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

A Pedro Pablo y Marco Antonio, por crear
TeXiS e iluminar nuestro camino

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 el precio de un juego 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, con soporte multiplataforma, cuyo 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 the price of a game 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, with cross-platform support, whose objective is to ease the creation of this type of games for users without experience in development or programming, while maintaning 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

Índice de figuras

Índice de tablas

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 han sido uno de los géneros más influyentes de la industria, desde sus orígenes en la década de los años 80 hasta la actualidad, donde concentran una gran parte de la cuota de mercado.

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 nos han permitido evolucionar desde máquinas diseñadas exclusivamente para poder ejecutar un único juego a la amplia gama de dispositivos multimedia de los que disponemos actualmente.

Numerosos programas de edición y desarrollo de videojuegos han aparecido en las últimas dos décadas, pero aquellos que son más comerciales están pensados para juegos de todo tipo, por lo que suelen tener características más generales y no tan 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:

- 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, que pese a no ser muy elevado, hace que usuarios aficionados paguen por un *software* que rara vez utilizarán si no se afianzan finalmente en el mundo del desarrollo.

- Los que no están bajo un muro de pago, es decir, son *software* de código libre, suelen tener interfaces y sistemas complicados de entender para noveles,

quienes debido a la complejidad de estas herramientas deciden abandonar por completo el desarrollo.

La motivación principal de este Proyecto surge de la necesidad de contar con herramientas accesibles 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 Proyecto tiene marcados como objetivos el desarrollo de un motor de videojuegos 2D, pensado específicamente para videojuegos de rol, acompañado de un editor 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 que permita la implementación de videojuegos de rol.
- Se desarrollará un editor multiplataforma que facilite la implementación de los videojuegos de rol en el motor desarrollado.
- Se probarán ambas herramientas con usuarios para demostrar el funcionamiento del Proyecto 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 tres fases:

- 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 intentará 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 intentará proponer mejoras a los problemas que estos puedan tener de cara a nuevos usuarios poco experimentados.
- Diseño del Proyecto. Con las características abstraídas en la fase anterior, se planteará un diseño inicial que servirá como base durante el desarrollo del Proyecto. Este diseño, si bien no será inmutable, debería ser lo más «final» posible para evitar problemas durante la fase de desarrollo.

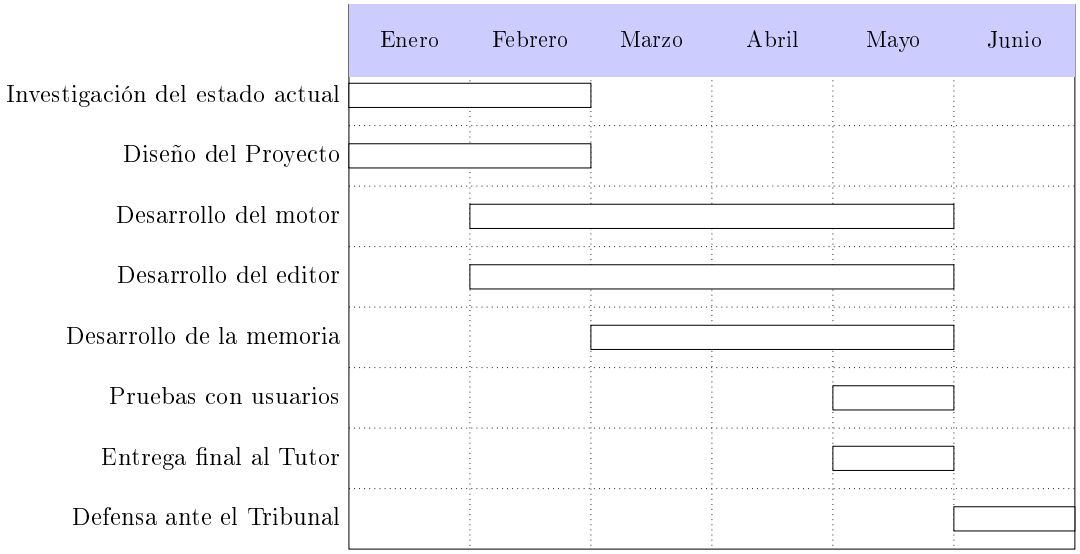


Figura 1.1: Diagrama de Gantt mostrando la planificación temporal del trabajo.

- Desarrollo del Proyecto. Una vez finalizado el diseño del Proyecto, 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 Proyecto, y se configurará todo para que sea lo menos trabajoso posible durante el desarrollo de ambas herramientas.
  - 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.

- Pruebas con usuarios. Para garantizar el correcto funcionamiento del Proyecto, 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 en el Proyecto 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.

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 del Proyecto, se utilizarán como entornos de desarollo 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 3.2**[1].

Por otra parte, la generación del PDF de la memoria se llevará a cabo mediante LaTeX, utilizando la plantilla de TeXiS, 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 Proyecto, 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, *Estado de la Cuestión*, se abordará la situación actual de los videojuegos de rol, del desarrollo de videojuegos, así como un poco de contexto histórico de ambos.

En el capítulo 3, *Planteamiento del Proyecto*, se tratará en profundidad el diseño planteado y las decisiones tomadas previas al desarrollo de las aplicaciones.

En el capítulo 4, *Desarrollo del Proyecto*, se hablará de la estructura interna del motor y del editor en cuanto a código, abordando también los problemas que se hayan tenido durante esta fase y las soluciones propuestas.

En el capítulo 5, *Evaluación y Conclusiones*, se expondrán las preguntas y objetivos de investigación, el desarrollo de las pruebas con los usuarios, y un análisis acerca de las pruebas, del que se extraerán conclusiones.

Finalmente, en el capítulo 6, *Trabajo Futuro*, se detallarán posibles mejoras para una futura actualización de las herramientas que podrían ser interesantes.

[1] ANOT: No estoy del todo de si poner las razones del uso aquí o referenciar la sección posterior donde se desarrolla.

Capítulo 2

Estado de la Cuestión

Resumen: En este capítulo vamos a hablar sobre los videojuegos de rol, introducir un poco su historia, hablar sobre el desarrollo de videojuegos, qué es un motor, para qué sirve, qué lo compone, qué es un editor, para qué sirve, y cuáles son los editores especializados en el desarrollo de videojuegos de rol.

2.1. Sobre los videojuegos de rol

Para poder definir qué es un videojuego de rol necesitaremos primero entender *qué es un videojuego* y *qué es un juego de rol*.

Definir qué es un videojuego es una tarea bastante complicada, sobre todo teniendo en cuenta los matices con los que podemos definirlo (académicos, de diseño, experimentales o tecnológicos). Una de las definiciones más acertadas nos 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.

Los juegos de rol, al haber nacido desde juegos de mesa tradicionales, han tenido el suficiente tiempo como para poder desarrollarse y poder ajustarse a una definición mucho más precisa, aunque, nuevamente, debido a la amplia variedad de juegos que se pueden catalogar como «de rol», es posible que hasta la definición más completa no pueda abarcarlos a todos. Lortz (1979, citado en Fine (1983)), define los juegos de rol como «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».

Por otra parte, Tychsen et al. (2006, citado en Hitchens y Drachen (2009)) enumera los elementos imprescindibles que un juego debe tener para ser considerado como juego de rol:

- Narrar una historia adecuándose a una serie de reglas. Tanto la historia como las reglas son únicas para cada juego.
- Reglas, múltiples participantes (al menos dos) y un mundo ficticio sobre el que se va a desarrollar la acción. Todos los participantes deben conocer previamente la ambientación, el entorno y las reglas.
- La gran mayoría de participantes controlarán, como mínimo, a un personaje durante toda la partida, y será con ese o esos personajes con quienes interactuará con el entorno.
- La existencia, por lo general, de un *director de juego* (GM, *gamemaster*), que será el responsable de gestionar aquellos elementos del juego o del entorno ficticio que no se encuentran bajo el control directo de los jugadores.

Ahora que ya sabemos *qué es un videojuego* y *qué es un juego de rol*, y pese a que no podemos dar una definición válida para todos los videojuegos que estén en esta categoría, definiremos un videojuego de rol (RPG, *role-playing game*, o también, más raramente, CRPG, *computer role-playing game*) como todo aquel programa *software*, con carácter lúdico, que permita a sus usuarios encarnar el rol de uno o varios personajes en un mundo ficticio, donde pueden tomar decisiones, interactuar con su entorno o mundo con cierta libertad y desarrollar una narrativa para conseguir un determinado objetivo.

Como se ha mencionado anteriormente, esta definición vale para la gran mayoría de RPG; pero, si pensamos en los videojuegos que se venden cada día en las tiendas, una proporción significativa de estos puede encajar dentro de la definición anterior sin necesariamente tener que ser un RPG, ya que prácticamente en todos se encarna el rol de un personaje, casi todos permiten una interacción con el entorno y todos tienen un objetivo final. Esto nos lleva a plantearnos si las clasificaciones de videojuegos según su género son demasiado limitadas para los videojuegos actuales.

2.1.1. El problema de los géneros de videojuegos

Antes de adentrarnos en los RPG, veremos por qué las distintas clasificaciones de videojuegos atendiendo a características similares tienen bastantes lagunas a la hora de categorizar las entregas más modernas.

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.

En la actualidad, los distintos géneros vienen dados por una mezcla de mecánicas, temas, elementos narrativos, estética, lugar de origen y plataforma, y están gravemente influenciados por las tendencias del mercado, discursos mediáticos y la

Figura 2.1: Escena de juego de *The Legend of Zelda: Breath of the Wild* (Nintendo, 2017), extraída de Barder (2017).

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.

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 tenemos ahora, resulta casi imposible poder agrupar la naturaleza multifacética de estos en etiquetas tan «tradicionales».

Pongamos el ejemplo de 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 podemos apreciar en la figura 2.1 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 (puedes ir consiguiendo nuevas habilidades o mejorando el equipamiento). Es por eso que muchas veces tenemos géneros intermedios, como en este caso, que podríamos 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,

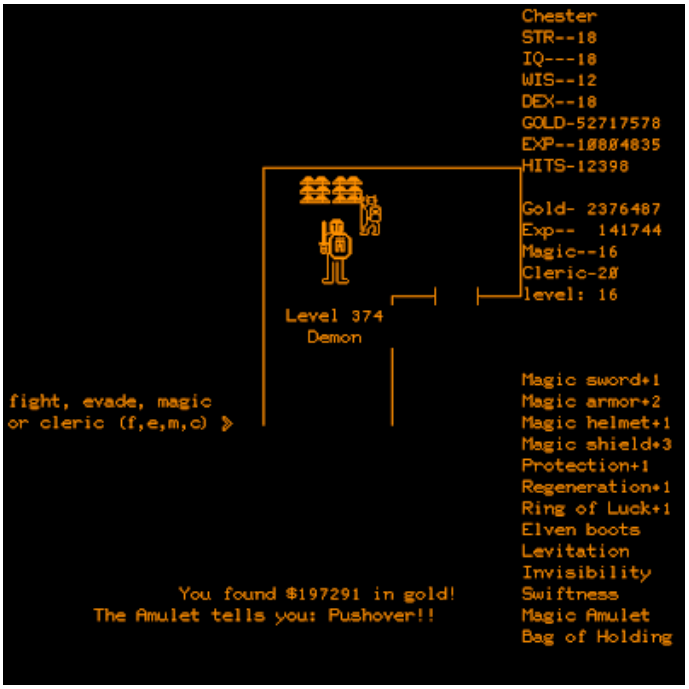


Figura 2.2: Interfaz de *dnd* (Whisenhunt y Wood, 1975), extraída de Addict (2019).

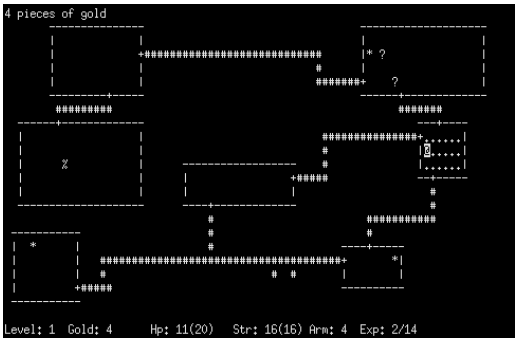
temas y estética, que varían de juego a juego.

2.1.2. Historia de los videojuegos de rol

Es a mediados de la década de los 70 cuando podemos 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 de *Dungeons & Dragons* (Gygax y Arneson, 1974), quizás el juego de rol más famoso, que mezcla mecánicas de combate con temas fantásticos. De *Dungeons & Dragons* se ha adoptado en la gran mayoría de juegos el lanzar los dados, las estadísticas de los personajes (por ejemplo, inteligencia o fuerza), o 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 2.2), 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» (en un símil con los años oscuros del Medievo europeo), principalmente por lo anteriormente mencionado, pero también por la escasa información que tenemos hoy en día sobre estos juegos, ya que muchos no se han conservado son *lost media* (aquellos 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:



(a) Captura de *Rogue: Exploring the Dungeons of Doom* (A.I. Design, 1980), extraída de Thedarkb (2021)

(b) Captura de *Ultima I: The First Age of Darkness* (Origin Systems, 1981), extraída de Lemon64 (2002)

Figura 2.3: Capturas de *Rogue* y *Ultima*.

- *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 2.3a) en el que se desarrolla una aventura basada por 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 2.3b).

Con el salto tecnológico que hubo a mediados de la década de los 80, los RPG comienzan a separase cada vez más de ser meras adaptaciones de juegos ya existentes a desarrollar sus propias historias. A partir de esta época encontramos 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, *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 por 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 nuestros días 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, especialmente en

los mercados 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 los RPG occidentales no tenían portabilidad, y los altos costes de desarrollo y producción que supuso el cambio de cartuchos tradicionales a nuevos formatos como CD-ROM, hicieron que los RPG, especialmente los WRPG, tuviesen este gran declive que solo pudo recuperarse con la entrada del nuevo milenio.

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 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) algunos de los juegos más jugados de la historia, e incluso ganando múltiples premios a juego del año.

2.2. Sobre el desarrollo de videojuegos

Para entender cómo las empresas o 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 veremos lo que es un *motor* y lo que es un *editor*.

2.2.1. 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». Ese *software* al que nos referimos 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, etc... A todas estas herramientas también se les denomina *motores* (por ejemplo, *motor de render*, *motor de físicas*...).

Todos estos «submotores» se suelen conocer como *motores de tecnología*, ya que 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 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 no había la suficiente cantidad de desarrolladores como para portar un juego a otra plataforma distinta con otra serie de requisitos y limitaciones.

No es hasta 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 podríamos llamar «motor». Uno de los primeros fue el que Shigeru Miyamoto desarrolló en Nintendo para la *Nintendo Entertainment System* (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 gran conocimiento. Por lo general, la gran mayoría de motores actuales suelen ser multiplataforma, algo de lo que hablaremos en el apartado 2.2.1.4.

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 ya vienen equipados con un editor, de lo cual hablaremos también posteriormente.

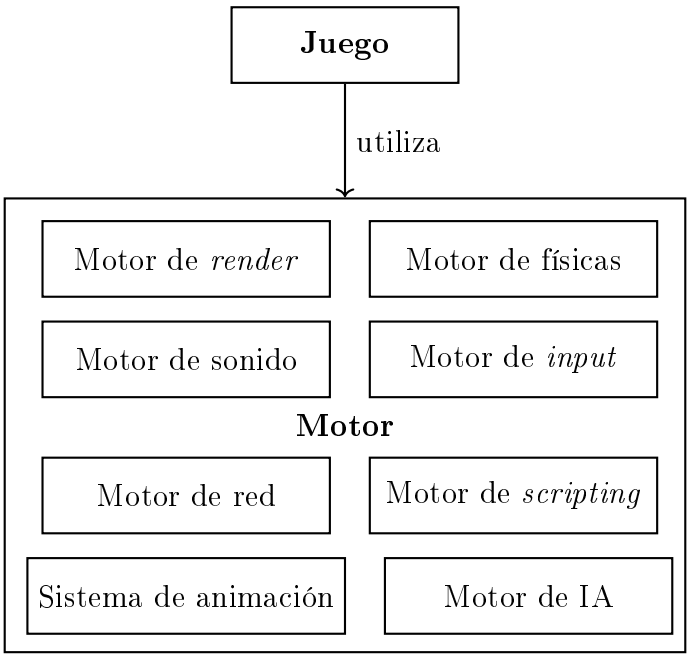


Figura 2.4: Representación esquemática de la estructura de un juego y su motor con algunos de los componentes principales

2.2.1.1. 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 se esté programando. Siguiendo el esquema provisto en la figura 2.4, 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 nuestro computador 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 encontramos 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). Suelen tener soporte para árboles de mezcla y cinemática inversa.

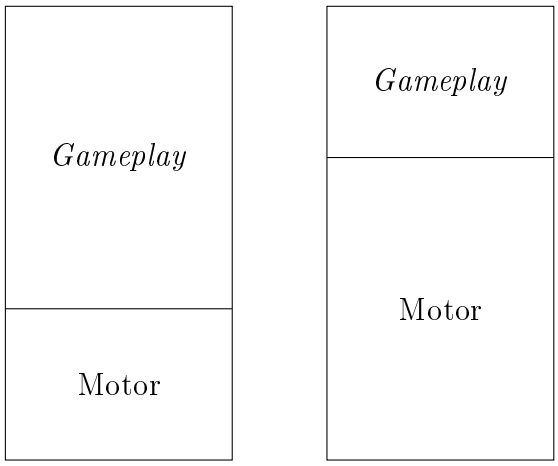
- *Motor de scripting*: permite escribir lógica específica del juego utilizando lenguajes de programación de alto nivel (como por ejemplo JavaScript, Lua, C# o Python), separando la implementación del *gameplay* («jugabilidad») del motor. El *scripting* se suele hacer mediante lenguajes de programación interpretados y no compilados, por lo que los cambios más pequeños no requieren de 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.



(a) Separación en motores generalistas.

(b) Separación en motores específicos.

Figura 2.5: Ejemplos de separaciones entre motor y *gameplay* en los distintos motores.

2.2.1.2. Separación entre motor y *gameplay*

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

Definimos jugabilidad o *gameplay* como la experiencia interactiva del jugador dentro de un videojuego. Ni Tekinbas y Zimmerman (2003) ni Schell (2019) dan una definición concreta, pero hablan de cómo la interacción entre el jugador, el sistema de reglas del juego y las mecánicas dan lugar a la jugabilidad. No solo cada juego tiene una jugabilidad distinta que depende de las mecánicas y de las reglas que este tenga, sino que puede haber una mecánica emergente que surge a través de las decisiones de cada jugador.

Los motores más generalistas que encontramos, como los anteriormente mencionados *Unity*, *Unreal Engine* o *Godot*, debido a que están pensados para poder desarrollar todo tipo de juegos, tienen una barrera de separación entre el motor y el *gameplay* suele estar a la mitad o incluso hacia abajo (ver figura 2.5a), es decir, que la implementación del juego final que tiene el motor es mínima (el motor es puramente un conjunto de motores de tecnología) y es el desarrollador el que se tiene que encargar de ello. Esta es la gran ventaja que tienen los motores generalistas frente a los específicos, la libertad que dejan al desarrollador para poder implementar la jugabilidad a su manera, con las reglas y mecánicas que él mismo establezca.

Si, como es el caso de este proyecto, nos interesa hacer un motor que fije gran parte de la jugabilidad de los juegos que se puedan hacer con él, y que lo único modificable sea la parte artística y visual, tendremos que introducir elementos propios del juego en nuestro motor, dejando menos libertad en el *gameplay* (ver figura 2.5b). Por ejemplo, nuestro motor dará a los desarrolladores herramientas para poder per-

sonalizar los escenarios de combate típicos en juegos RPG, así como un creador de mapas basado en baldosas o *tiles*.

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, que muchas veces se ven abrumados ante tal cantidad de opciones que ofrecen el resto de motores más flexibles, si bien es cierto que no ofrecen la libertad de estos, y la jugabilidad está fijada por el desarrollador del motor y no el del juego.

2.2.1.3. 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 nivel medio (como C o C++, que en la base son lenguajes de alto nivel pero poseen estructuras de acceso a *hardware* como los lenguajes de bajo nivel), ya que se espera tener una gran optimización en estos así como una portabilidad 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 reducir los tiempos en las iteraciones de desarrollo, la programación dirigida por datos permite a los diseñadores del juego, que no tienen que ser necesariamente programadores, 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.

Sin embargo, uno de los principales problemas que supone este paradigma es la depuración del código. Como los datos se suelen almacenar en lenguajes de *scripting*,

o en lenguajes de almacenamiento de datos (como JSON o XML), depurar un código o simplemente datos que nuestro motor interpretan, resulta extremadamente difícil si el motor no cuenta con herramientas de depuración integradas específicas para interpretar esos datos.

2.2.1.4. Programación multiplataforma

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 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, como CMake, 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.

Otra dificultad añadida es la de la obtención de las distintas SDK (*software development kit*, kit de desarrollo de *software*) y licencias de cada empresa, ya que no son código libre (todo el código está bajo acuerdos de confidencialidad), y muchas de ellas no suelen ser flexibles a la hora de entregar estas licencias a nuevas empresas.

Es por ello que muchas veces, en proyectos no profesionales, se suelen utilizar librerías multiplataforma, como SDL, que simplifican bastante la conversión de un sistema a otro y agilizan el desarrollo de un motor; o, directamente, se usan motores multiplataforma, como los ya mencionados *Unity*, *Unreal Engine* o *Godot*.

2.2.2. Editor de videojuegos

Un editor de videojuegos es una herramienta, generalmente visual, que permite a los desarrolladores crear, modificar y probar contenido de un juego sin la necesidad

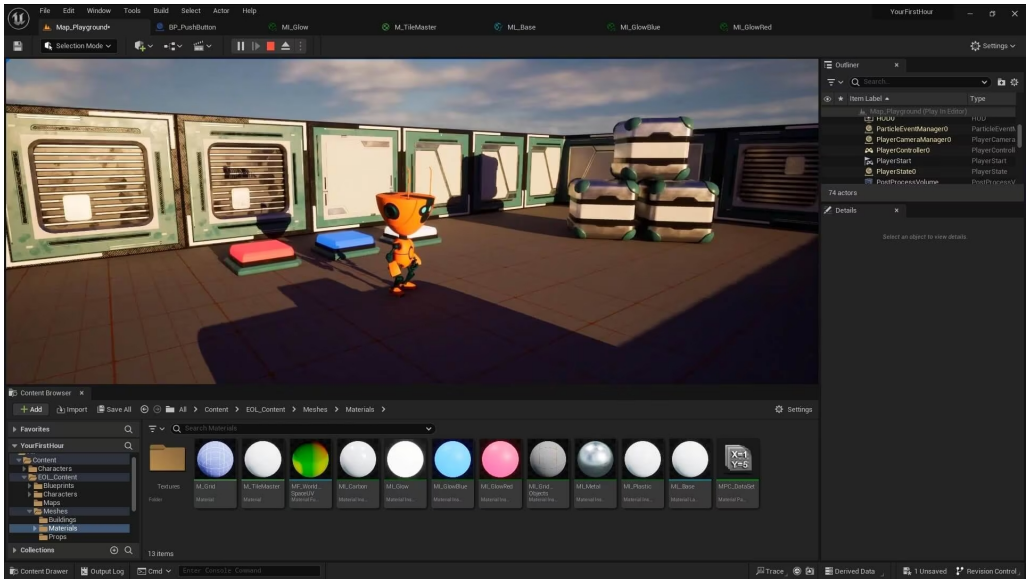


Figura 2.6: Vista de la interfaz de *Unreal Engine* 5.2, extraída de Wadstein (2023).

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, NPC (*non playable character*, personaje no jugable); editar *scripts* o crear eventos que definan 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 2.6, apreciamos varias de 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).

Los editores suelen estar integrados con el motor del juego y aprovechan los principios de la programación dirigida por datos (ver apartado 2.2.1.3), 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*

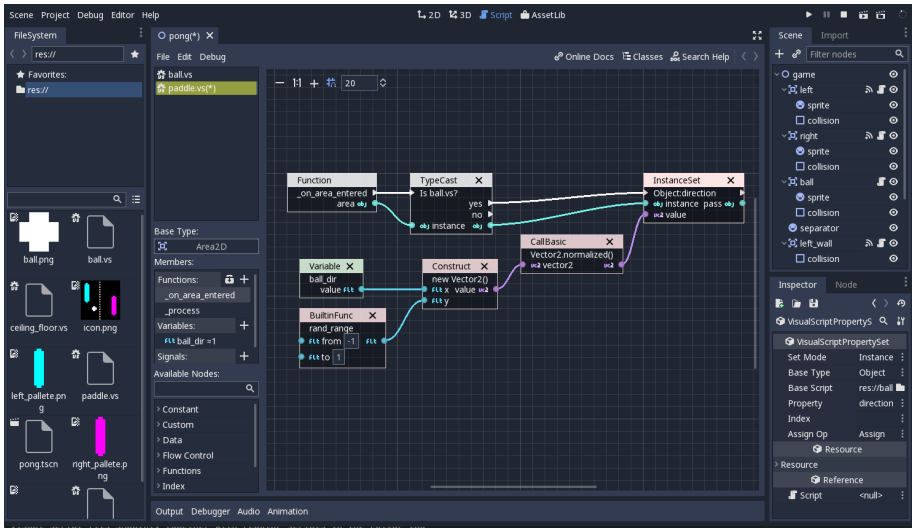
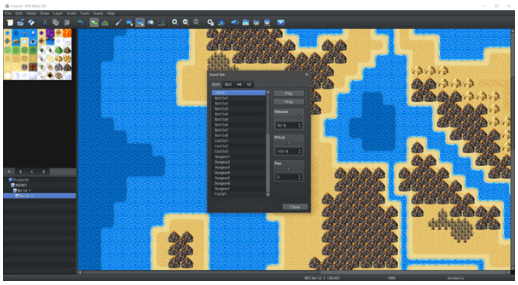


Figura 2.7: Vista del *scripting* visual mediante nodos de *Godot*, extraída de Linietsky y Manzur (2022).

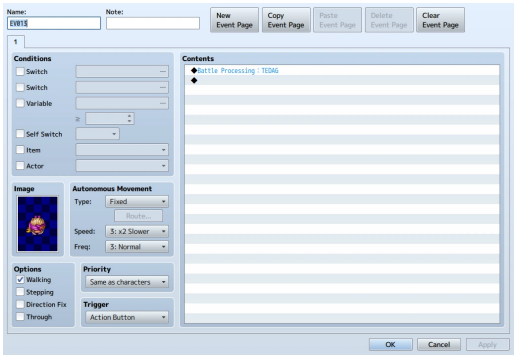
Editor), lo cual agiliza las iteraciones durante el desarrollo.

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 2.7), 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.

2.2.2.1. Editores específicos 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 2.8: Capturas de distintos elementos de la interfaz de *RPG Maker*.

Si bien es cierto que una gran parte de los juegos vendidos a diario son RPG, 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*, desarrollada desde 1992 por ASCII (ahora Enterbrain) y Gotcha Gotcha Games. *RPG Maker* está pensada para permitir a usuarios sin conocimientos avanzados de programación crear juegos completos mediante una interfaz sencilla (ver figura 2.8a).

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 2.8b).

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 (*mostrar diálogo*, *mover personaje*, *cambiar variable*...) y que puede ser utilizado por diseñadores sin experiencia alguna, así como desarrolladores más curtidos en la materia que pueden crear sistemas más complejos.

También se incluye un sistema de batallas por turnos tradicional, inspirado en los JRPG clásicos, aunque es posible modificarlo 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.

Además de *RPG Maker*, existen varias alternativas de código abierto, como *RPG Paper Maker*, que permite crear RPG en 3D con estética retro. Otras opciones, como *RPG JS* permiten desarrollar los juegos utilizando lenguajes como TypeScript o HTML5 para navegadores.

También encontramos proyectos como *EasyRPG*, que busca recrear el motor proporcionado en *RPG Maker 2000/2003* de manera gratuita y de código abierto, 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 *OHRRPGCE* (*Official Hamster Republic Role Playing Game Construction Engine*), un motor que se lleva manteniendo desde 1998 con un enfoque en juegos del estilo de *Final Fantasy*.

Capítulo 3

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.

3.1. Objetivos principales del Proyecto

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 estaría pensado principalmente para gente no programadora o sin experiencia en el desarrollo de videojuegos, por lo que la interfaz tendría que ser intuitiva y fácil de utilizar y aprender.

El editor debería 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 móviles[1].

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 capaz de guardar el estado de un proyecto y poder recuperarlo cuando el usuario desee, sin que se hayan podido perder los cambios que se hayan realizado. Y, también, debe poder exportar proyectos, e importar otros 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

[1]Los editores suelen tener elementos que son preferibles de ser utilizados mediante entrada de teclado y ratón. Si bien es cierto que Android permite la conexión de periféricos de entrada-salida, es una plataforma pensada para dispositivos móviles y táctiles.

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).

3.2. Toma de decisiones

La primera decisión a tomar fue la plataforma de desarrollo del Proyecto. Se decidió desarrollar el grueso del Proyecto en C++, ya que se quería aprovechar el alto rendimiento que ofrece en comparación a otros lenguajes[2], 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 Proyecto.

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 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, debido a que este IDE carece de versiones para MacOS y Linux[3], se optó por hacer el desarrollo del Proyecto 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 del Proyecto, 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, 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 lo que se decidió utilizar *Android Studio*, la herramienta oficial de desarrollo para Android, que proporciona máquinas virtuales de dispositivos Android de 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 tendría 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

[2]El 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.

[3]MacOS y Linux cuentan con *Visual Studio Code*, que si bien sirve para poder compilar C/C++ mediante el uso de *plug-in*, es más complicado de configurar para proyectos más complejos como este.

mecanismo por el cual un lenguaje de programación puede llamar a funciones o rutinas programadas o compiladas en otro lenguaje distinto, lo cual es necesario para poder ejecutar el juego, ya que la entrada de la aplicación Android estaría en Java.

El resto de decisiones son propias de cada una de las partes del Proyecto y se detallarán a continuación.

3.2.1. Diseño del motor

3.2.1.1. Base con el sistema entidad-componente

La base del motor se estructurará atendiendo a un patrón EC (entidad-componente). Los sistemas específicos de los juegos RPG se construirán sobre esta base modular, que se puede separar en dos partes diferenciadas: el bucle de juego y la carga de recursos.

El bucle de juego se estructurará en escenas. Cada escena estará formada por un conjunto de entidades que se actualizarán en paralelo. A su vez, cada una de estas entidades agrupará un conjunto de componentes, que encapsularán funcionalidades específicas, permitiendo una gran flexibilidad mediante el uso de polimorfismo. Gracias a esta estructura se podrán implementar una amplia variedad de mecánicas mediante la creación de distintos componentes asociados a las entidades.

En cuanto a la carga de recursos, este sistema se encargará de informar al motor sobre las escenas creadas y su contenido. Estas escenas estarán descritas cada una en un fichero Lua, estructurado de forma que puede descomponerse fácilmente en sus entidades, y, a su vez, en los componentes que las conforman.

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.

La parte más compleja de este sistema es la conversión de estructuras de texto en Lua a instancias de clases de C++ específicas. Para ello, se usará un *patrón factoría*: cada componente estará asociado a un identificador, que estas «factorías» utilizarán para instanciar el componente correspondiente y añadirlo a la entidad.

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*, menos usado recientemente), que liberará el recurso que más tiempo lleve sin usarse para hacer espacio al nuevo.

3.2.1.2. Componentes genéricos de juego

Para desarrollar videojuegos, es necesario contar con ciertas funcionalidades básicas que faciliten la implementación de los sistemas específicos del juego.

En el motor, se utilizará la librería `SDL` 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 *input*: se desarrollará un sistema de entrada 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 el *input* 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`. Desde el punto de vista del desarrollador, su uso es sencillo: 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.

3.2.1.3. Componentes específicos de RPG

En primer lugar, 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 de 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.

3.2.1.4. 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. Cada uno de estos eventos estará compuesto por dos elementos principales: una condición y un conjunto de comportamientos.

La condición determinará bajo qué circunstancias deberá activarse el evento. Existirán múltiples tipos de condiciones, encargadas de evaluar distintos aspectos del estado del juego. Si una de las condiciones se cumple, el 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,

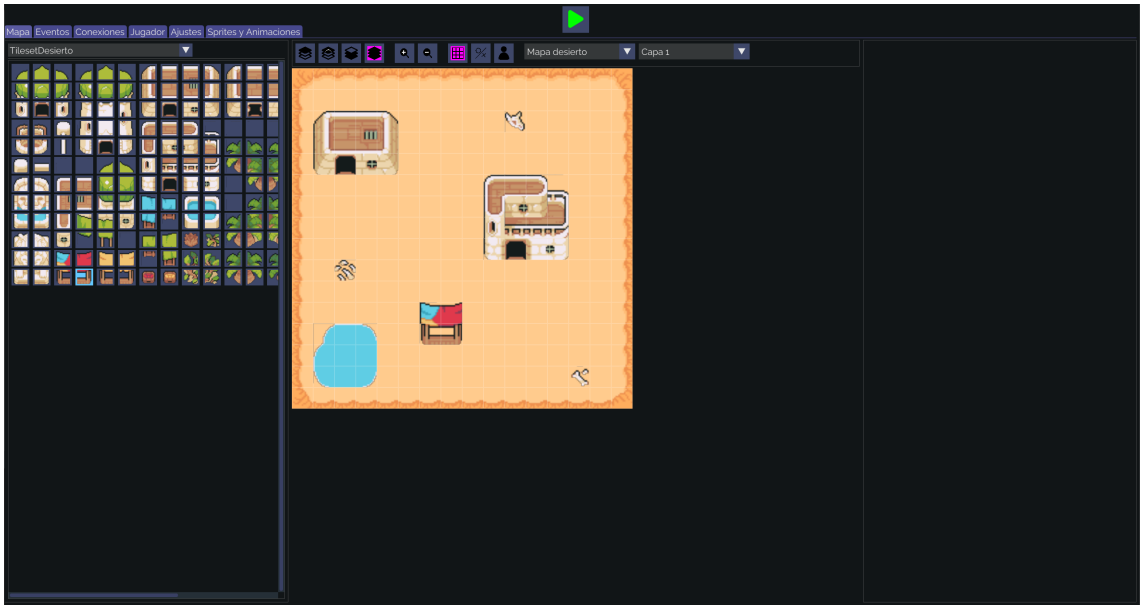


Figura 3.1: Editor de mapas de *RPGBaker*.

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 *scrpting* simplificado, será posible crear lógicas de juego complejas mediante herramientas accesibles y visuales, sin la necesidad de escribir código complejo.

3.2.2. Diseño del editor

3.2.2.1. 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 tendría correspondían con aquellos elementos que podían ser modificados en el motor, es decir:

- Editor de mapas (figura 3.1), que permitiese al usuario cargar sus propios *tilesets*; decidir el tamaño de la cuadrícula que ocuparía 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.

- Editor de eventos, que permitiese 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 tendría que tener 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 permitiese al usuario generar un *sprite* dada una imagen o *spritesheet*, y, posteriormente, animaciones dada una serie de *sprites*. Ambos editores tendrían una previsualización del *sprite* o animación, añadiéndose controles para la reproducción en este último caso.

- Editor de objetos, que permitiese 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 permitiese 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 permitiese 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 tendrían la capacidad de poder ser configurados al gusto del usuario, permitiendo ser editados y eliminados cuando este desee.

Además, el editor incluiría un *viewport* para que el usuario pudiese ver en tiempo real el diseño final del juego, sobre el cual se podría ejecutar y probar las funcionalidades implementadas, al estilo de *Unity*.

Por otra parte, el editor contará con un sistema de persistencia, que permita una carga y guardado de los *proyectos*, es decir, la representación de un juego en el editor. Todas los *assets* utilizados tendrán que estar referenciados en archivos de configuración del *proyecto*, y el guardado actualizará estos archivos de configuración.

3.2.2.2. Modularidad y arquitectura

En cuanto a la estructura de la interfaz, se optó por utilizar un patrón típico 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 apareciesen 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 tendrían comunicación unas con otras utilizando al *proyecto*, que almacenaría toda la información referente a cada uno de los juegos que el usuario crease (por ejemplo, referencias a los *tilesets*, *sprites*, animaciones o mapas creados).

Se tendrían 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 el Proyecto sea multilingüe[4] con una amplia escalabilidad en el caso de que se quisiesen añadir más idiomas.

3.2.2.3. 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 ya haber optado por utilizar `SDL3` como base para el motor, se optó también como base para el editor, acompañada de `DearImGui` para el dibujado y manejo de los elementos de la interfaz.

También, al ya haber optado por el uso de Lua como lenguaje de *scripting* y de definición de datos para el motor, se optó 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*, etc...). 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, al igual que como ocurre en la gran mayoría de motores[5].

Si bien es cierto que una de las técnicas más comunes en la implementación de un editor es la de utilizar el motor para el que se desarrolla como base, ya que esto evita el tener que implementar dos veces las mismas funcionalidades (principalmente en *renderizado*, *input* y *scripting*), se decidió independizar el desarrollo del editor del motor para poder avanzar más rápidamente en el Proyecto. Esta opción, pese a que ha supuesto una mayor carga de trabajo, ha permitido flujos de iteración en el desarrollo más cortos.

[4] En 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.

[5] Esto 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 4

Desarrollo del Proyecto

Resumen: En este capítulo se tratará la implementación del motor, la del editor, así como una puesta en marcha del entorno de desarrollo utilizado.

4.1. Puesta en marcha del *toolchain*

Antes de comenzar a desarrollar ambas herramientas, se desarrolló un *toolchain* (cadena de herramientas) que permitiese la generación de proyectos, tanto para el desarrollo en C++, como para el desarrollo en Java de Android.

Para ello, fue necesario configurar el archivo `CMakeLists.txt` de la raíz del Proyecto. Este fichero CMake permite la descarga de las dependencias externas mediante el uso del módulo `FetchContent`. Aquí se encuentra el primer «inconveniente», derivado del uso de este módulo: `FetchContent` descarga las librerías externas que se le pidan tantas veces como plataformas y tipos de compilación posibles haya. Esto provoca que una librería muy sencilla se necesite descargar ocho veces para poder generar el ejecutable final del juego para cada una de las plataformas destino deseadas (en total, cuatro plataformas destino y dos tipos de compilación por plataforma).

Pese a haber investigado alguna manera sencilla de poder evitarlo, el comportamiento de este módulo es fijo y no se ha podido encontrar una alternativa.

Por otra parte, atendiendo a la figura 4.1, cada uno de los proyectos contiene a su vez un fichero `CMakeLists.txt` que configura las direcciones de inclusión y de enlazado de cada una de las herramientas, así como la detección de los ficheros `.cpp` necesarios para la compilación. Esta estructura modular permite un mejor mantenimiento de cada proyecto, manteniendo cada módulo separado y encapsulado, lo que permite a su vez facilidad a la hora de las pruebas durante el desarrollo.

Debido a que el proyecto `Engine` ha sido concebido como una librería dinámica, se ha necesitado la creación de un proyecto `Executable`, que genera un ejecutable con llamadas a la API provista en `Engine`. Este será el ejecutable final que lance al juego.

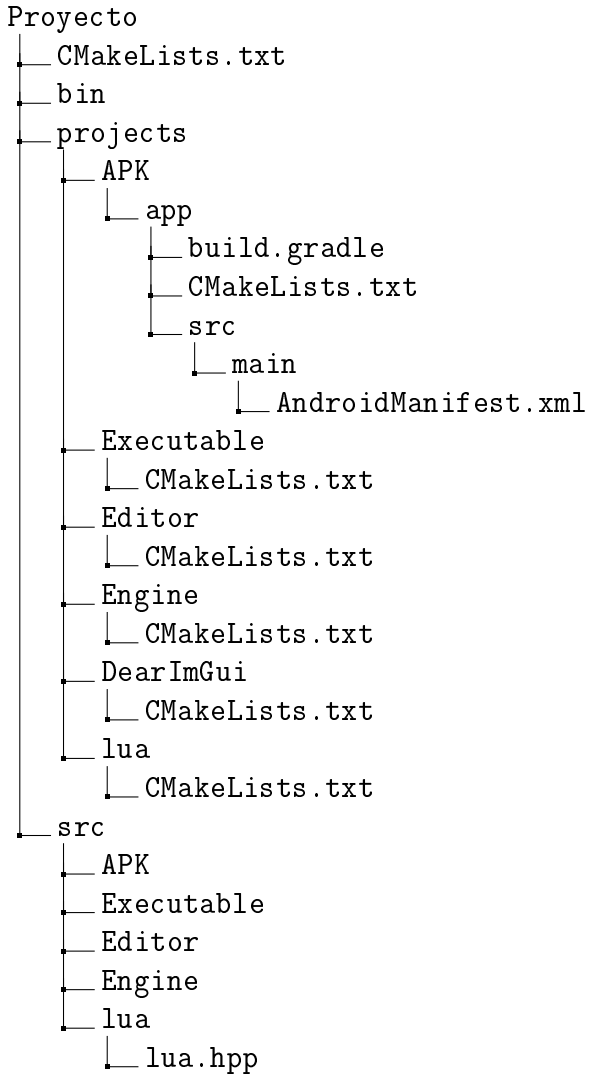


Figura 4.1: Estructura de carpetas del Proyecto.

En cuanto a los proyectos `lua` y `DearImGui`, se han generado los `CMakeLists.txt` correspondientes para su sencilla incorporación, debido a que estas dos librerías no disponen de soporte inicial para CMake.

También es necesario mencionar que se ha añadido un fichero de cabeceras de C++, `lua.hpp`, con el siguiente contenido:

```
extern "C" {
        #include <lua.h>
        #include <lauxlib.h>
        #include <lualib.h>
}
```

Esto es debido a que la librería de Lua para C/C++ está escrita en C, por lo que es necesario decirle al compilador de C++ que trate todo el contenido de la librería como si fuese código C. Así se evita cualquier problema a la hora de enlazar el proyecto con la librería de Lua debido al *name mangling* (o modificación de nombres) que el compilador de C++ pueda hacer.

En cuanto a la generación del proyecto en Android, se ha generado un fichero `build.gradle` que habilita el uso del NDK y JNI para poder ejecutar el motor desde una APK, así como la configuración de la entrada de la aplicación y del manifiesto de Android (archivo en formato XML que tiene información esencial de la aplicación).

Gracias a `SDL`, no ha hecho falta programar la entrada a la aplicación en Java, sino que es la propia librería la que se encarga de llamar automáticamente al código C++ mediante un extra para la programación en Android.

4.2. Desarrollo del motor

4.3. Desarrollo del editor

4.3.0.1. Estructura del código y organización de módulos

Capítulo 5

Evaluación y Conclusiones

Conclusiones del trabajo y líneas de trabajo futuro.

Antes de la entrega de actas de cada convocatoria, en el plazo que se indica en el calendario de los trabajos de fin de grado, el estudiante entregará en el Campus Virtual la versión final de la memoria en PDF.

5.1. Objetivo de la evaluación

5.2. Metodología

5.3. Resultados

5.4. Análisis de los resultados y conclusiones

Capítulo 6

Trabajo Futuro

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 have been one of the most influential genres in the industry, from their origins in the 1980s to the present day, where they account for a large portion of the market share.

The development of this type of game has undergone major changes over the years, driven by continuous hardware and software improvements, which have allowed us to evolve from machines designed solely to run a single game to the wide range of multimedia devices available today.

Numerous video game development and editing tools have appeared in the last two decades, but the more commercial ones are designed for games of all kinds. As a result, they tend to offer more 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.
- Those that are not locked behind a paywall, that is, 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 accessible and flexible tools, both for experimented 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.

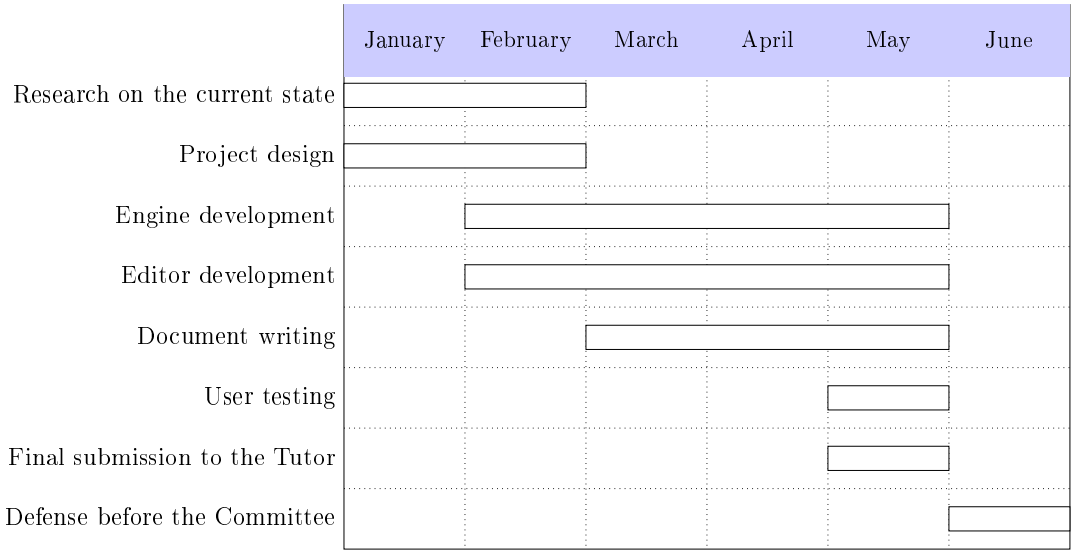


Figure 6.1: Gantt diagram showing the temporal planning of the work.

Objectives

This project aims to develop a 2D game engine specifically designed for role-playing games, along with an editor that 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 alowing more experienced users to create larger and more ambitious projects.

To accomplish this:

- A cross-platform engine that allows the user to implement role-playing games will be developed.
- A cross-platform editor that eases the implementation of a role-playing game in the developed engine will be developed.
- The Project 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 three 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.

- Project 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.
- Project 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 the Project's needs, and everything will be configured to minimize effort during the development of the applications.
  - 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.
- User testing. To ensure the correct functioning of the Project, 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 Project 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.

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 the Project, 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 3.2.

On the other hand, PDF generation of the document will be carried out using LaTeX, with the TeXIStemplate, 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 of the Project 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, *State of the Art*, the current situation of role-playing video games, video game development, and a bit of historical context of both will be discussed.

In Chapter 3, *Project Planning*, the design and decisions made prior to the development of the applications will be discussed in detail.

In Chapter 4, *Project Development*, the internal structure of the engine and editor in terms of code will be discussed, while addressing the problems encountered during this phase and the solutions proposed.

In Chapter 5, *Evaluation and Conclusions*, the research questions and objectives will be presented, along with the development of the user testing phase and an analysis of the tests, from which conclusions will be drawn.

Finally, in Chapter 6, *Future Work*, potential improvements for the final implementation of the applications that may be interest will be detailed.

Conclusions and Future Work

Conclusions and future lines of work. This chapter contains the translation of Chapter 5.

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 utilizaríamos 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 del *toolchain* de generación de dependencias y compilación de los distintos subproyectos, y la redacción de esta memoria.

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). 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.

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

Barder, O. «Breath Of The Wild» Is Not The Best «Zelda» Game Ever Made. 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.

CD Projekt Red. *The Witcher 3: Wild Hunt*. PlayStation 4, Xbox One, PC. CD Projekt. 2015.

Cenance, M. *RPG Maker MV* Event Editor. 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.

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.

Gotcha Gotcha Games. *RPG Maker MZ*. 2020. Imagen de la interfaz de *RPG Maker MZ* accedida desde la página de Steam del software 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.

Hitchens, M. y Drachen, A. The Many Faces of Role-Playing Games. *The International Journal of Role-Playing*, páginas 3–21, 2009. ISSN 2210-4909.

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.

Lemon64. Ultima I: The First Age of Darkness - Commodore 64 Game - Download Disk/Tape - Lemon64. 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. 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.

Nintendo. *The Legend of Zelda: Breath of the Wild*. Nintendo Wii U y Nintendo Switch. Nintendo. 2017.

NINTENDO. *Excitebike*. Nintendo Entertainment System. Nintendo. 1984.

NINTENDO. *Super Mario Bros.*. Nintendo Entertainment System. Nintendo. 1985.

ORIGIN SYSTEMS. *Ultima I: The First Age of Darkness*. PC, Atari 8-bit y Commodore 64. California Pacific Computer. 1981.

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.

SQUARESOFT. *Final Fantasy*. Nintendo Entertainment System. SquareSoft. 1987.

TEKINBAS, K. y ZIMMERMAN, E. *Rules of Play: Game Design Fundamentals*. ITPro collection. MIT Press, 2003. ISBN 9780262240451.

THEDARKB. Rogue Screenshot.png. 2021. Imagen de *Rogue* accedida desde la entrada en Wikipedia del juego (`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. 2023. Imagen de la interfaz de *Unreal Engine* 5, accedida desde el curso de *Unreal Engine* de Epic Games el día 29-04-2025.

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.