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 todos los juegos vendidos a diario.

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 lo están 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.

1.2. Objetivos

Descripción de los objetivos del trabajo.

1.3. Plan de trabajo

Aquí se describe el plan de trabajo a seguir para la consecución de los objetivos descritos en el apartado anterior.

1.4. Herramientas y Metodología

Descripción de las herramientas utilizadas en el trabajo y de la metodología seguida para llevar a cabo las etapas del plan de trabajo.

1.5. Estructura de la memoria

Explicación breve de la estructura de la memoria (qué se encuentra en cada parte) y algunos enlaces de interés al repositorio con el proyecto, etc...

1.6. Enlaces adicionales

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,

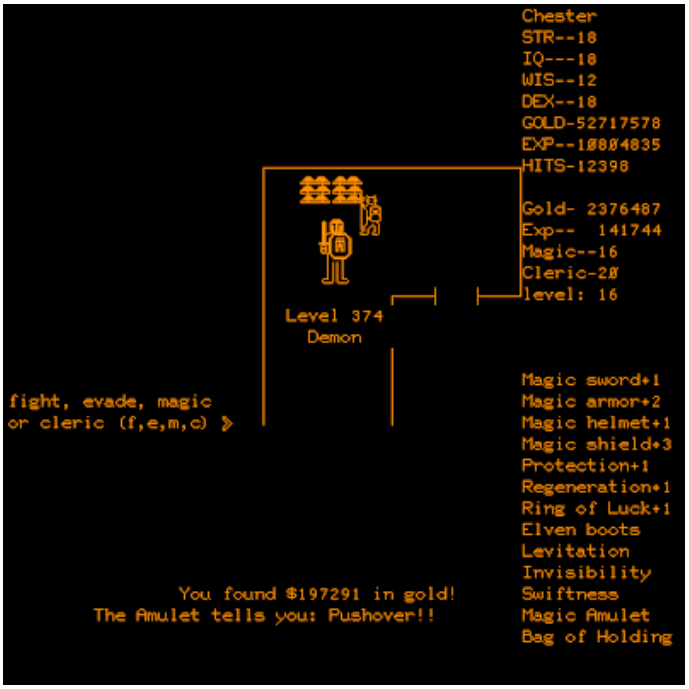


Figura 2.2: Interfaz de *dnd* (Whisenhunt y Wood, 1975), extraída de Addict (2019).

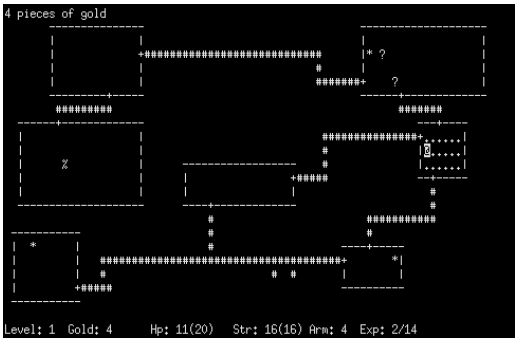
temas y estética, que varían de juego a juego.

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.

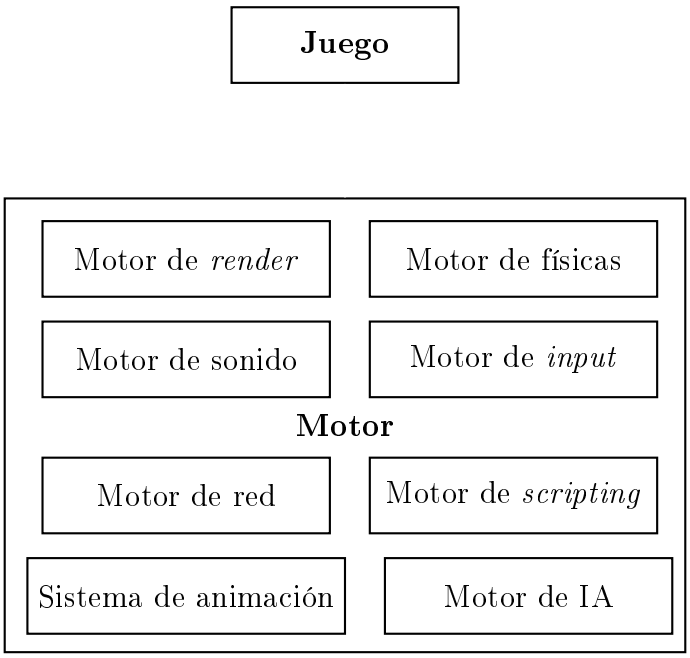


Figura 2.4: Representación esquemática de la estructura de un juego y su motor con algunos de los componentes principales

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.

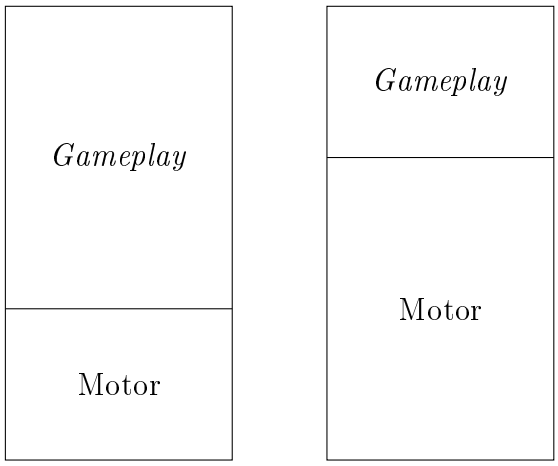
- *Motor de scripting*: permite escribir lógica específica del juego utilizando lenguajes de programación de alto nivel (como por ejemplo JavaScript, Lua, C# o Python), separando la implementación del *gameplay* («jugabilidad») del motor. El *scripting* se suele hacer mediante lenguajes de programación 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

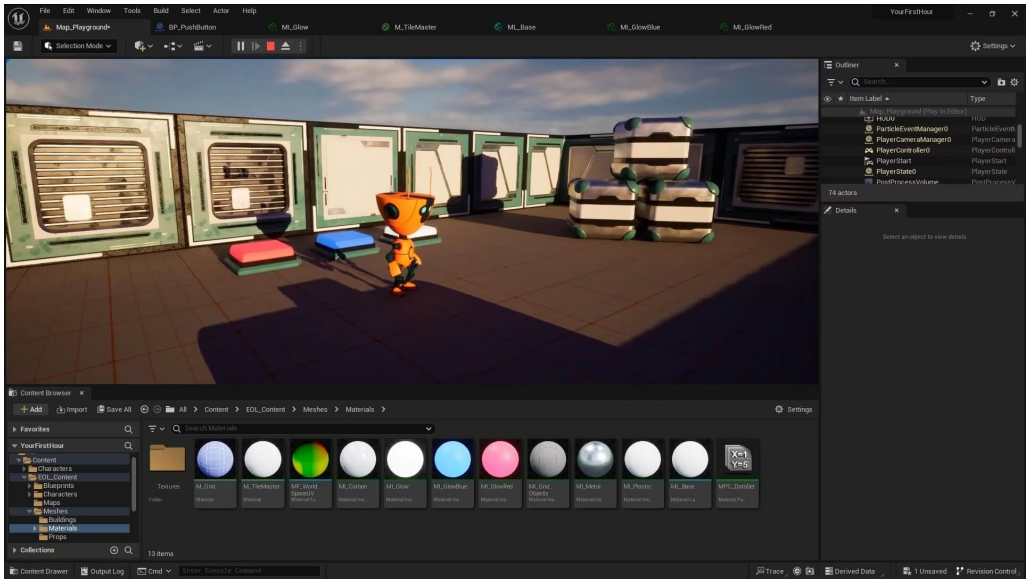


Figura 2.6: Vista de la interfaz de *Unreal Engine* 5.2, extraída de Wadstein (2023).

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*

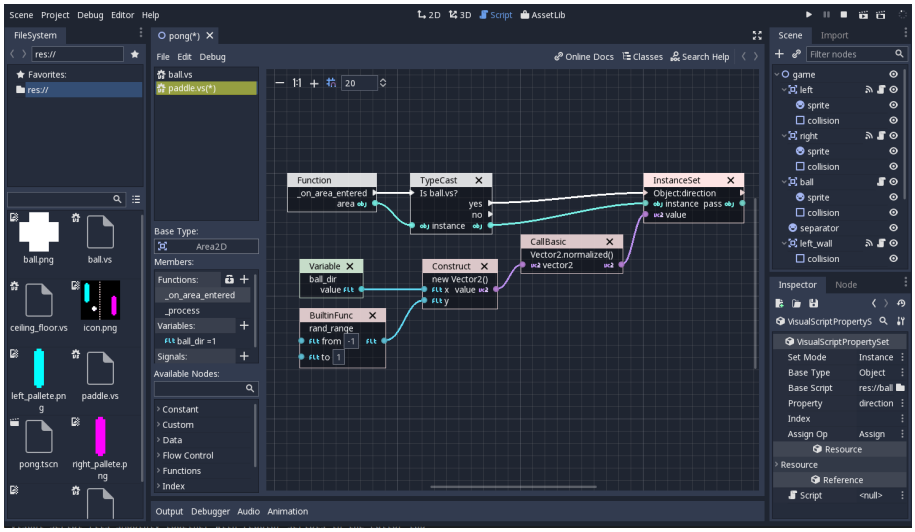
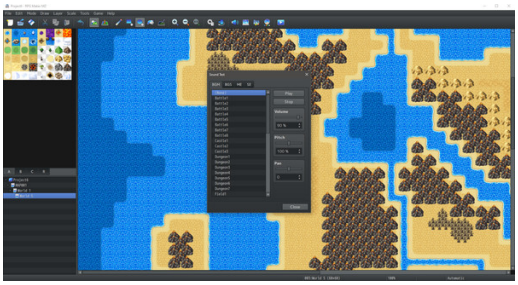


Figura 2.7: Vista del *scripting* visual mediante nodos de *Godot*, extraída de Linietsky y Manzur (2022).

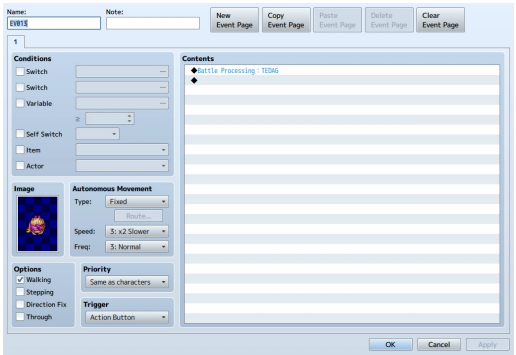
Editor), lo cual agiliza las iteraciones durante el desarrollo.

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 vamos a hablar de los objetivos principales del proyecto, así como de las decisiones tomadas en cuanto a diseño del motor y del editor.

3.1. Objetivos principales del Proyecto

Nuestra idea principal era el desarrollo de un motor de videojuegos, enfocado a los RPG 2D, acompañado de un editor que permitiese un desarrollo rápido y sin mucha complicación de juegos de este tipo, sobre todo para gente no programadora o sin experiencia en el desarrollo de videojuegos. El videojuego generado por el editor debería ser multiplataforma, y poderse ejecutar en Windows, Mac, Linux y Android. Por otra parte, el editor también sería multiplataforma, para las anteriormente mencionadas excepto Android, ya que no es la plataforma más preferible a la hora de poder ejecutar un editor[1].

El editor debería ser capaz de poder generar las cuatro versiones del juego distintas para cada plataforma, permitir al usuario elegir para qué plataforma generar el ejecutable, y volcar todo el contenido desarrollado a estos, asegurando que el comportamiento programado fuese el mismo que en el juego final, y que sea esta versión la que el usuario pueda empaquetar y distribuir, sin la necesidad de hacer nada más.

El editor también debe ser capaz de poder 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 este haya podido perder ningún cambio que haya realizado. Y, finalmente, debe poder exportar proyectos, e importar otros que otros usuarios puedan haber generado en otras plataformas 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 visual. Tendrá que tener soporte para periféricos de entrada-salida tradicionales (teclado y ratón), así como entrada táctil (para los dispositivos móviles).

[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.

3.2. Toma de decisiones

La primera decisión a tomar fue la plataforma de desarrollo del Proyecto. El Proyecto se iba a desarrollar íntegramente en C++, ya que se quería aprovechar la potencia que ofrece con respecto a otros lenguajes, 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 nos facilitarían a la hora de desarrollar el Proyecto.

Debido a nuestra premisa de un desarrollo multiplataforma, se necesitaba usar un IDE (*integrated development environment*, entorno de desarrollo integrado) que fuese compatible tanto con Windows, Mac, 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 Mac y Linux[2], 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 (la compilación utilizando CMake, en nuestra experiencia, es considerablemente más rápida que la compilación utilizando el compilador por defecto de *Visual Studio*) 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 han supuesto a la hora del manejo de las distintas dependencias externas frente a la alternativa que se había valorado[3] han hecho que esta opción haya merecido la pena.

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*.

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), una FFI (*foreign function interface*, interfaz de funciones foráneas, es decir, un mecanismo por el cual un lenguaje de programación puede llamar a funciones o rutinas programadas o compiladas en otro lenguaje de programación distinto) integrada en el SDK de Java que permite la comunicación con C/C++. Esto sería necesario para poder ejecutar el juego, pese a que la entrada de la aplicación estuviese en Java.

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

[2]Mac y Linux cuentan con *Visual Studio Code*, que si bien sirve para poder compilar C/C++, es más complicado de configurar para proyectos más complejos como este.

[3]La alternativa era utilizar *submódulos* en GitHub, pero eso suponía un elevado coste extra a la hora de clonar el proyecto (en lugar de clonar únicamente el código a una máquina, todas las librerías que se utilizasen tendrían que ser clonadas), actualizar las librerías, etc...

3.2.1. Diseño del motor

En cuanto al diseño del motor, las primeras decisiones giraban en torno al contenido *gameplay* que se quería que este tuviese. La principal idea era poder diseñar juegos al estilo de las primeras generaciones de la saga *Pokémon* (Game Freak, 1996) (es decir, un juego *pokemonlike*). Finalmente, se determinó que el contenido fuese el siguiente:

- Mapas, esenciales para el desarrollo de videojuegos. Estos estarían formados por dos elementos, un *tilemap* (es decir, un conjunto de baldosas individuales unidas en un mismo archivo) y objetos.
- Personajes, con características personalizables y que pudiesen interactuar con la escena mediante el uso de eventos.
- Objetos, que son elementos que contienen atributos, como por ejemplo *sprites*, eventos, sonidos, etc...

El motor seguiría un patrón EC (entidad-componente), con las entidades agrupadas en distintas escenas.

3.2.2. Diseño del editor

Sobre el editor, la decisión más importante fue elegir qué librería se iba a utilizar para poder desarrollar toda la interfaz y la funcionalidad básica de este. 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* para el motor, se optó por el uso del mismo para la generación de *scripts* del editor, acompañados por la librería `sol2`, que proporciona *bindings* (código que permite el manejo de Lua desde C/C++ de una manera muy sencilla sin tener que lidiar con la a veces rebuscada sintaxis de la librería `lua` de C/C++) para facilitar el desarrollo.

En cuanto a diseño, se decidió independizar el desarrollo del editor del motor para poder avanzar en el Proyecto de manera más rápida, por lo que el editor tiene sus propias clases encargadas del dibujado en pantalla, de la gestión de la entrada del usuario y del manejo del *scripting* y ficheros.

Si bien es cierto que una de las técnicas comunes en el diseño de motor-editor es la de implementar el editor utilizando las propias funcionalidades del motor, ya que se evita tener que implementar dos veces las mismas funcionalidades (en cuanto al *renderizado*, *input* y *scripting*), creemos que la opción escogida ha supuesto una menor carga de trabajo final en el Proyecto y ha permitido flujos de iteración en el desarrollo más ágiles.

En cuanto a la estructura de código, se optó por utilizar un patrón típico en el desarrollo de *software* no relacionado con videojuegos, basado en ventanas (o subventanas) anidadas otras ventanas, con el uso de ventanas modales que apareciesen

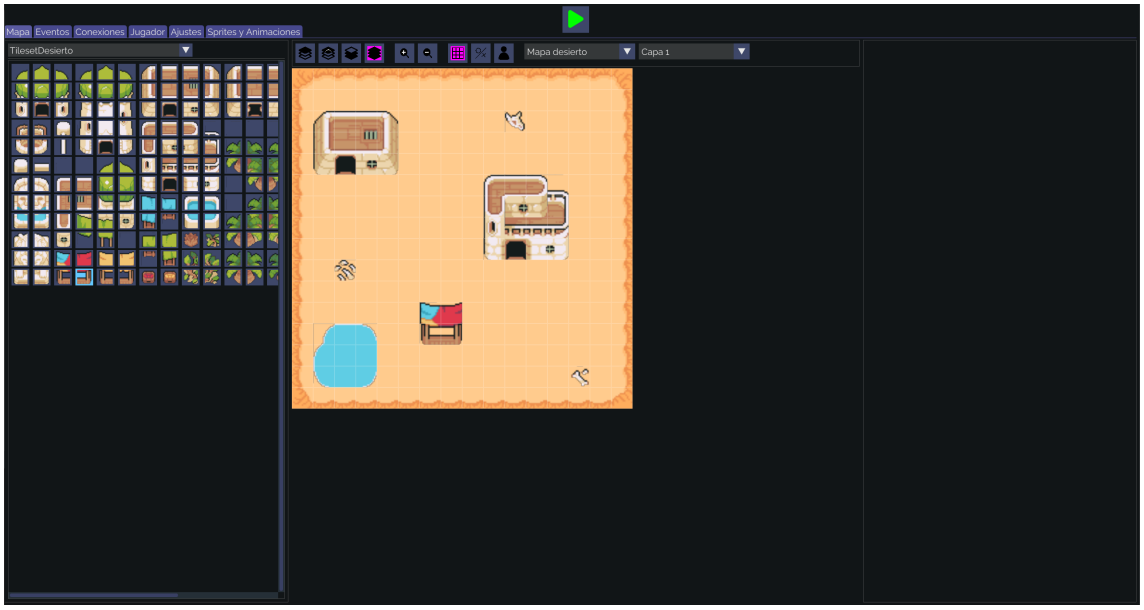


Figura 3.1: Editor de mapas de *RPGBaker*.

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 clases como `Project`, que almacenaría toda la información referente a cada uno de los proyectos 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.

Para la persistencia de los ficheros de configuración del proyecto, se optó por utilizar archivos de Lua simulando el uso que otros formatos, como JSON o XML pudiesen tener, ya que permitía reusar Lua no solo como lenguaje de *scripting* y no complicaba el tener que utilizar otra librería específica para ello. Estos archivos Lua difieren de los que el motor interpretará al final, ya que el editor necesita muchos más datos que para el motor son irrelevantes, por lo que se tendría que tener una parte de compilación o generación de la *build* del juego que «tradujese» los archivos del editor a archivos del motor. Al igual que como ocurre en los motores más generales, se decidió que el editor no pudiese interpretar los archivos finales del juego[5]

[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.

El editor, se diseñó teniendo en mente las características principales que se querían poder editar del motor, es decir, un editor de mapas que permitiese al usuario cargar sus propias *tilesets*, decidir el tamaño de los mapas, añadir objetos de manera visual y establecer las colisiones, aprovechándose de la posibilidad de tener varias capas de dibujado, así como poder situar la posición de los mapas en el mundo (ver figura 3.1); un editor de eventos, que permitirían flujos condicionales y la ejecución de animaciones o lanzamiento de otros eventos; un editor del personaje, que permitiese la personalización y configuración de elementos de este; y un editor de *sprites* y animaciones, que permitiesen al usuario poder elegir un fichero de imagen, generar los *sprites* decidiendo las coordenadas iniciales y el alto y ancho de la imagen, y con esos *sprites* poder generar las animaciones, eligiendo el tiempo entre cada uno de los fotogramas y el orden de estos. Todos estos elementos tendrían que poder ser editados y eliminados una vez creados.

En un principio, se contaba con la idea de que el editor tuviese un *viewport* para que el usuario final pudiese ver en tiempo real el desarrollo que estaba haciendo, y poder ejecutar el juego desde el propio editor, al estilo de *Unity*, pero debido a complicaciones con la API de `SDL3` decidimos descartarlo y permitir al usuario ejecutar su juego con una instancia del motor aparte.

Capítulo 4

Desarrollo del Proyecto

4.1. Desarrollo del motor

4.2. Desarrollo del editor

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

Introduction to the subject area. This chapter contains the translation of Chapter 1.

Conclusions and Future Work

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

Contribuciones Personales

Miguel Curros García

Al menos dos páginas con las contribuciones del estudiante 1.

Alejandro González Sánchez

Al menos dos páginas con las contribuciones del estudiante 2. En caso de que haya más estudiantes, copia y pega una de estas secciones.

Alejandro Massó Martínez

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.

GAME FREAK. *Pokémon*. Nintendo Game Boy, Nintendo DS, Nintendo 3DS y Nintendo Switch. Nintendo. 1996.

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.