Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Videojuegos para usuarios con discapacidad visual: creación con LibGDX de un RPG accesible

Autor: Mario Cobos Maestre

Tutores: José María Gutiérrez Martínez y Juan Aguado Delgado

2019

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Fin de Grado

Videojuegos para usuarios con discapacidad visual: creación con LibGDX de un RPG accesible

Autor: Mario Cobos Maestre

Directores: José María Gutiérrez Martínez y Juan Aguado Delgado

Tribunal:

Presidente: .....

Vocal 1º: .....

Vocal 2º: .....

Calificación: .....

Fecha: .....

Agradecimientos

A mis antiguos compañeros de N31 por todos los buenos momentos y por haberme brindado la oportunidad de afrontar este TFG.
A mi familia, por el apoyo recibido durante estos meses en los que el frenesí de terminar ha causado más de un roce.
A Adri y Bea, por los ratos que me han aguantado divagando sobre ideas que no parecían llevar a ninguna parte.
A Haven, por estar ahí cuando necesitaba alguien que me escuchase con mis paranoias.
Y a mi yo pasado, por decidir emprender este camino, y no rendirse hasta el final.

Palabras clave

Accesibilidad en videojuegos, RPG, LibGDX, Dungeon Crawler

Keywords

Videogame accessibility, RPG, LibGDX, Dungeon Crawler

Resumen corto

El proyecto consiste en un estudio de la accesibilidad y el estado de la herramienta LibGDX para dilucidar y, posteriormente, aplicar técnicas específicas orientadas a un perfil de diversidad funcional visual. Para ello, se ha desarrollado un prototipo de videojuego Dungeon Crawler de corte clásico, sobre el que se han aplicado técnicas para suplementar la información disponible de forma visual mediante el uso de sonidos, y compatibilizar el resultado con lectores de pantalla.

Short summary

This project consists, first, of the study of the state of the art in accessibility, and the possibilities of the LibGDX framework. This is done in order to, then, design and apply specific techniques, aimed at visually impaired people. In order to achieve this goal, a prototype Dungeon Crawler game, fashioned after old-school classics, has been developed. A number of techniques and ideas have been applied to it in order to reinforce the information supplied via text using sounds and to make it compatible with several screen reader applications.

Índice general

Índice general **vii**

1 Introducción **1**

2 Objetivos **3**

3 Estado del arte **5**

3.1 Videojuegos RPG 5

3.1.1 Breve historia del RPG 6

3.1.2 Características comunes 7

3.1.3 Subgéneros más comunes 7

3.1.3.1 RPG de acción 8

3.1.3.2 RPG estratégico 8

3.1.3.3 RPG multijugador masivo en línea 9

3.1.3.4 Roguelikes 9

3.1.3.5 Dungeon Crawlers 10

3.2 Accesibilidad 10

3.2.1 Diversidad funcional sensorial 11

3.2.2 Pautas de accesibilidad 12

3.2.3 Pautas de accesibilidad para videojuegos 12

3.2.4 Estado de la accesibilidad en la industria del videojuego 13

3.3 LibGDX 14

3.3.1 LibGDX 1.0 15

3.3.2 LibGDX 1.1 16

3.3.3 LibGDX 1.2 16

3.3.4 LibGDX 1.3 16

3.3.5 LibGDX 1.4 16

3.3.6 LibGDX 1.5 16

3.3.7 LibGDX 1.6 17

3.3.8 LibGDX 1.7 17

3.3.9 LibGDX 1.8 . . . 17
3.3.10 LibGDX 1.9 . . . 18

4 Desarrollo del proyecto 21

4.1 Análisis: LibGDX . . . 21
4.1.1 Generación y uso de un proyecto LibGDX . . . 22
4.1.2 Estructura de un proyecto LibGDX . . . 24
4.2 Análisis: Tiled . . . 24
4.3 Diseño del videojuego . . . 25
4.3.1 Objetivo del juego . . . 25
4.3.2 Ambientación del juego . . . 26
4.3.3 Personajes de jugador . . . 26
4.3.3.1 Características . . . 26
4.3.3.2 Clases . . . 27
4.3.4 El pueblo . . . 29
4.3.5 El laberinto . . . 30
4.3.5.1 Estructura del laberinto . . . 30
4.3.5.2 Enemigos . . . 36
4.3.6 Habilidades . . . 37
4.3.7 Objetos . . . 41
4.3.8 Pantallas . . . 42
4.3.8.1 Consideraciones de accesibilidad . . . 42
4.3.8.2 Menú principal . . . 42
4.3.8.3 Menú del pueblo . . . 43
4.3.8.4 Menú de la taberna . . . 44
4.3.8.5 Menú de la tienda . . . 44
4.3.8.6 Menú de la posada . . . 45
4.3.8.7 Pantalla de exploración . . . 46
4.3.8.8 Menú de pausa . . . 47
4.3.8.9 Menú de inventario . . . 49
4.3.8.10 Menú de habilidades . . . 49
4.3.8.11 Pantalla de estado . . . 50
4.3.8.12 Pantalla de combate . . . 51
4.4 Implementación del videojuego . . . 54
4.4.1 Aproximación arquitectónica . . . 54
4.4.1.1 Patrón Modelo-Vista-Controlador . . . 54
4.4.1.2 División en niveles de abstracción . . . 54

4.4.1.3 Estructura modular . . . 57
4.4.2 Estructuras de datos fundamentales . . . 59
4.4.2.1 Implementación del sistema de entidades . . . 59
4.4.2.2 Implementación del sistema de inventario . . . 63
4.4.2.3 Implementación de las habilidades . . . 64
4.4.2.4 Implementación del laberinto . . . 64
4.4.3 Implementación del sistema de guardado . . . 68
4.4.4 Implementación de la máquina de estados finita . . . 70
4.4.4.1 Implementación de la gestión de estados . . . 71
4.4.5 Implementación de la capa de scripting . . . 71
4.4.5.1 Interfaz Script . . . 72
4.4.5.2 Clase ScriptManager . . . 72
4.4.5.3 Clase LuaScript . . . 74
4.4.5.4 Clase ExecuteScript . . . 74
4.4.6 Implementación de la capa de accesibilidad . . . 75
4.4.7 Implementación del sistema de menús . . . 78
4.4.8 Implementación del sistema de mensajes . . . 79
4.4.9 Implementación del sistema de control . . . 80
4.4.10 Implementación de la exploración . . . 81
4.4.11 Implementación del sistema de combate . . . 83
4.5 Correspondencia con las *Game Accessibility Guidelines* . . . 85

5 Coste del proyecto 87
5.1 Coste material . . . 87
5.1.1 Coste por tiempo de trabajo . . . 87
5.1.2 Gastos generales y beneficio industrial . . . 87
5.1.3 Presupuesto de ejecución por contrata . . . 88
5.1.4 Importe total . . . 88

6 Conclusiones y líneas futuras 89
6.1 Resumen . . . 89
6.2 Conclusiones . . . 90
6.3 Líneas futuras . . . 91

Bibliografía 93

A Detalles de implementación 95
A.1 Clases . . . 95

Capítulo 1

Introducción

La industria del videojuego se ha consolidado, pese a sus relativamente recientes inicios, como una de los negocios más lucrativos que existen en la actualidad. Los medios disponibles para la producción de productos audiovisuales de entretenimiento han experimentado, como consecuencia, un gran desarrollo.

No obstante, la misma naturaleza del medio puede suponer un obstáculo para que personas con determinadas dificultades accedan a él. Quizá uno de los perfiles más notables y evidentes sea el de aquellas personas que, debido a alteraciones o degradaciones en la visión, sean incapaces de procesar correctamente el contenido ofrecido por un producto cuyo soporte es, preeminentemente, visual.

El último censo oficial sobre discapacidad realizado en 2008 en España por el Instituto Nacional de Estadística, estima en torno a 3,8 millones de personas pertenecientes a diversos perfiles de diversidad funcional, representando un 8,5 % de la población, aproximadamente[1].

Existen, no obstantes, múltiples técnicas que se pueden emplear para facilitar el acceso de estas personas a las tecnologías de la información. Dichas técnicas impactan, en su mayoría, a las fases de análisis y diseño del proceso de desarrollo, evidenciando que, al menos hasta cierto punto, las dificultades para presentar el contenido de una forma adecuada a sectores de la población que necesitan adaptaciones especiales se deben a una falta de conocimiento o aplicación de estas directivas.

De entre los diversos géneros en los que se puede englobar un videojuego, uno de los más longevos es el de los RPG, o juegos de rol, caracterizado por un énfasis en la narrativa, el desarrollo de personaje y la resolución de problemas, habitualmente mediante sistemas basados en estadísticas y habilidades. Dada la imperiosa necesidad de comunicar información mediante, normalmente, texto escrito, se presenta, no obstante, un claro problema para aquellas jugadores que presentan ceguera total, o ciertos grados de reducción de la visión: la imposibilidad de leer información esencial para el desarrollo de la partida. Desde diálogos hasta información suplementaria sobre objetos y habilidades, pasando por los numerosos sistemas de menú que se utilizan para navegar por la distintas opciones que caracterizan al género, jugar resulta completamente imposible en esas circunstancias.

Este trabajo de fin de grado pretende demostrar, mediante un enfoque específico en los videojuegos RPG y un perfil de diversidad funcional visual, la posibilidad de, empleando técnicas de accesibilidad adaptadas al contexto de los videojuegos, generar un producto accesible, sin renunciar por ello a un cierto grado de profundidad mecánica.

A un nivel individual, se desea que este trabajo sirva, asimismo, como una experiencia de aprendizaje y concienciación personal, permitiendo interiorizar las múltiples directivas en él aplicadas, así como explorar, de manera superficial, áreas transversales al campo de la informática en el desarrollo de un videojuego.

Capítulo 2

Objetivos

> El objetivo principal de este trabajo de fin de grado es el desarrollo de un prototipo de juego RPG bidimensional multiplataforma mediante el uso del motor LibGDX, condicionado a que dicho prototipo sea lo más accesible posible para un perfil de diversidad funcional visual.

Para ello, será necesario:

- Investigar y analizar en mayor profundidad las pautas de accesibilidad ya existentes.
- Determinar el grado de aplicación posible de las mismas en el contexto del género de juego seleccionado.
- Adquirir nociones elementales de diseño de juego.
- Comprender la aplicación del proceso de ingenierá del software al campo de los videojuegos.
- Estudiar la aplicación de patrones de diseño software en el ámbito concreto del videojuego.
- Profundizar en el uso del motor LibGDX, y, por consiguiente, de la herramienta Gradle.

Será la aplicación conjunta de todos los conocimientos adquiridos como resultado lo que permitirá llevar a buen término el proyecto desarrollado.

Capítulo 3

Estado del arte

Este capítulo pretende ofrecer una visión más profunda de los elementos involucrados en este TFG, divididos en tres secciones: el género de videojuegos RPG, la accesibilidad desde el punto de vista de un perfil de diversidad funcional visual, y la propia herramienta LibGDX.

En primer lugar, se explicará el concepto de videojuego RPG, su origen y características generales más notables, y una breve historia del desarrollo del género, incluyendo breves referencias a sus subgéneros más notorios. Dicha lista no pretende ser exhaustiva, dada la vasta variedad de posibilidades que los mismos ofrecen.

A continuación, se procederá a ofrecer una visión general del concepto de accesibilidad, que se procederá a detallar y matizar de cara al perfil anteriormente citado. Asimismo, se tratará de forma breve el estado de la accesibilidad en la industria del videojuego, centrando la atención, una vez más, en un perfil de diversidad funcional visual.

Finalmente, se presentarán las características e historia del motor LibGDX, buscando ofrecer una visión superficial del estado en que se encuentra en el momento en el que se está redactando este documento. Para ello, se mencionarán brevemente los cambios más significativos introducidos por cada versión desde 1.0 hasta 1.8.0, y a continuación se enumerarán las características que presenta el motor en su versión actual.

3.1 Videojuegos RPG

Los videojuegos RPG (del inglés Roleplaying game, o juego de rol) han sido desde hace años grandes exponentes en la industria internacional del videojuego. Nombres como Final Fantasy, World of Warcraft o Skyrim figuran ya en el imaginario colectivo gracias al impacto del que el género goza, y diversas compañías han construido su éxito en torno a este mercado.

Como todo videojuego, su soporte es fundamentalmente visual, y, dado el enfoque a la narrativa habitual en el género, estos juegos tienden a apoyarse en grandes extensiones de texto para comunicar información al jugador, ya sea relevante a la jugabilidad del título en cuestión, o a la construcción del mundo con el que el jugador ha de interactuar.

A continuación se ofrece, en primer lugar, un vistazo a la historia y desarrollo de los RPG como género de videojuego, y, en segundo lugar, una breve discusión de las características comunes de los mismos, así como de algunos subgéneros notables.

3.1.1 Breve historia del RPG

Pese a que el género no se abriría al gran público hasta 1980, podemos encontrar los primeros precursores del mismo en el año 1975, publicados para el sistema de entrenamiento asistido por ordenador PLATO (acrónimo de Programmed Logic for Automatic Teaching Operations). Títulos como Dungeon y Moria, influenciados por el recientemente creado juego de rol de mesa Dungeons and Dragons, establecerían los conceptos básicos del género, así como el modelo de presentación que se utilizaría en los RPG occidentales durante la siguiente década: sistemas de progresión de personaje basados en clases y estadísticas; un enfoque principalmente orientado a la exploración de laberintos hostiles en busca de tesoros, y una presentación gráfica principalmente apoyada en texto, en la que los laberintos se representaban en primera persona mediante gráficos de estilo wireframe.

El nuevo género gozó de un éxito moderado, lo que, unido a la popularidad del anteriormente citado Dungeons and Dragons, llevaría, en 1980, a la publicación de Rogue, creando el primer subgénero de RPG hasta la fecha. No obstante, la consolidación del género no se produciría hasta 1981, cuando se lanzaron al mercado las primeras entregas de las series Ultima y Wizardry para Apple-II. Ambos gozaron de un gran éxito, y no solo contaron con numerosas secuelas a lo largo de los años, sino que influenciaron una gran mayoría de desarrollos posteriores.

Para comprender el estado actual del videojuego RPG, tenemos, no obstante, que dirigir la mirada a Japón, en el año 1986, cuando un pequeño equipo, influenciado por el Wizardry original, desarrolló Dragon Quest para NES. Pese a no ser el primer juego del género publicado en consola, o para dicha plataforma, su arrollador éxito en su país de origen marcaría a sus múltiple sucesores, definiendo las características básicas del subgénero conocido como JRPG. Lo que es más, serviría de referencia para el desarrollo del primer Final Fantasy, una de las sagas más reconocidas de la actualidad.

Así pues, se crearía una brecha entre el RPG occidental y el japonés. Donde en occidente se optaría por mantener un estilo más clásico, orientado a la exploración y la resolución de situaciones de diversa índole, en Japón se daría preferencia a la narración de historias más complejas, a costa de mayor rigidez. Esto no quiere decir que ambos conceptos fueran completamente incompatibles: títulos como Megami Tensei (Atlus, 1987), y su saga sucesora, Shin Megami Tensei (Atlus, 1992) tratarían de unificarlos, creando líneas argumentales detalladas, pero influidas por las decisiones del jugador, así como incluyendo nuevos conceptos, como la posibilidad de negociar con el enemigo para formar alianzas u obtener nuevos recursos. Lo que es más, la saga Wizardry pasaría a estar en manos de desarrolladoras japonesas, tales como Starfish y Acquire, a principios de los 2000, y contaría con numerosos sucesores desarrollados en el país del sol naciente, muchos de los cuales nunca llegarían a territorio occidental de manera oficial.

La década de los 90 está considerada como la edad de oro del RPG japonés gracias a la explosión que el género experimentó en SNES y, posteriormente, PlayStation, y a títulos como Final Fantasy VI, Chrono Trigger e, incluso, las primeras entregas de Pokémon. La enorme producción permitió la experimentación con nuevas ideas mecánicas y narrativas, popularizando sistemas más dinámicos y dando fuerza a subgéneros que, si bien ya habían sido establecidos, no habían terminado de despegar todavía, como podrían ser los RPG de acción o los RPG de estrategia. En el lado occidental, los RPG de exploración de mazmorras en primera persona comenzaban a perder popularidad, a favor de juegos en tercera persona con una mayor profundidad estratégica, como el ampliamente reconocido Baldur's Gate, o Fallout.

Esta tendencia permanecería durante la primera mitad de la primera década de los 2000. Los RPG, como género ya ampliamente establecido, gozaban de una popularidad y salud más que aceptables. Sagas como Shin Megami Tensei comenzaban a ganar reconocimiento en occidente, y la experimentación continuaba. Lo que es más, los primeros RPG multijugador masivos en línea, o MMORPG, comenzaban a experimentar con las tecnologías disponibles en la época. Pese a la existencia de títulos anteriores al año

2000, el nuevo subgénero no cobraría excesiva fuerza hasta la publicación de World of Warcraft (Blizzard, 2004).

No obstante, la llegada de la generación de consolas compuesta por Playstation 3, Xbox 360 y Wii anunciaba el declive del mismo. El protagonismo pasó a pertenecer a los juegos de disparos en primera persona, y la producción de RPG, si bien no se detuvo completamente, sí que se vio parcialmente estancada. Cabe destacar que sería en esta época cuando los juegos de exploración de mazmorras en primera persona experimentarían un aumento de popularidad en los círculos todavía afines al género, gracias a la saga Etrian Odyssey (Atlus, 2007).

En la actualidad, los videojuegos RPG gozan de una popularidad moderada. Algunas sagas consideradas clásicas, como Final Fantasy o Pokémon, conservan un reconocimiento más que respetable. Incluso, desarrollos más recientes, como Dark Souls, Bravely Default y Octopath Traveler, son considerados obras maestras.[2]

3.1.2 Características comunes

Habiendo observado el origen del género, resulta sencillo entender cuáles son los puntos comunes más habituales, así como el origen de los mismos.

En primer lugar, resulta habitual la presencia de un sistema de estadísticas, que puede estar expuesto o no al jugador. Las posibles estadísticas dependen directamente del título en cuestión, pero un elemento recurrente es la vitalidad de los personajes, que determina su capacidad para seguir funcionando en el mundo, así como la presencia de un recurso tal como la resistencia o el maná, que gestione el acceso a habilidades especiales.

Del mismo modo, la narrativa suele ocupar, con mayor o menor grado de acierto, un papel central. Salvo ciertos casos específicos, normalmente juegos que imitan a los RPG de mazmorreo de estilo más clásico, los personajes y el mundo toman gran relevancia en la presentación del título. No es de extrañar, dado que el objetivo principal de los juegos de rol de mesa que sirvieron como precedente para el nacimiento de este género era, precisamente, la creación de historias de manera colectiva.

Los dos puntos anteriores son, en la mayor parte de los casos, los pilares fundamentales sobre los que se sustenta el juego. Adicionalmente, suelen aparecer también sistemas de inventario, con diversas implementaciones; misiones secundarias para ampliar el tiempo de vida del producto final, así como incentivar la exploración, mediante la promesa de nuevos objetos e información que mejoren la experiencia general del jugador, y diversos sistemas de mejora de personajes, que interactúan de forma continua con el sistema de estadísticas.

Adicionalmente, resulta también bastante común que buena parte de la interacción del jugador con el contenido del juego se produzca mediante menús y cuadros de texto, aunque la carga de estos elementos está directamente ligada al subgénero específico del mismo.

3.1.3 Subgéneros más comunes

El género RPG cuenta con multitud de subgéneros. Tratar la totalidad de los mismos resultaría una labor titánica, por no decir imposible, pero algunos destacan por encima de los demás.

Es relativamente habitual la aparición de juegos que aúnan elementos de múltiples subgéneros. Un ejemplo notorio de esta hibridación entre subgéneros es el sistema de batalla en tiempo activo presente en los juegos de la saga Final Fantasy entre el cuarto y el noveno juego numerado, ambos incluídos, así como en Chrono Trigger, también desarrollado por Square-Enix, que introducía el tiempo como factor en

un sistema de batalla por turnos propio de los RPG más clásicos, acercando así el resultado a un RPG de acción, sin por ello presentar todas las características de este subgénero.

Precisamente la enorme variedad resultante de esta clase de combinaciones es la razón por la que el estudio de todas las posibles variantes escapa al propósito de este trabajo de fin de grado, o de esta sección dentro del mismo.

3.1.3.1 RPG de acción

Probablemente el subgénero más extendido en la actualidad, los RPG de acción se caracterizan por desarrollarse en tiempo real. A diferencia de subgéneros más clásicos, en que las batallas se desarrollan por turnos y usando sistemas de menús, los ARPG abogan por presentar sistemas de combate en los que las distintas acciones están asignadas a distintos botones, y en los que la precisión a la hora de ejecutar diversas acciones es tan importante como la acción a ejecutar, sino más, requiriendo pericia además de una buena gestión de las estadísticas del personaje. La jugabilidad resultante muestra claros parecidos con la de los juegos *Beat 'Em Up* o la de los juegos de lucha, exigiendo una interacción más activa al jugador en todo momento.

Ejemplos notables publicados en la última década incluyen Dark Souls (From Software, 2009), Dragon's Dogma (CAPCOM, 2012) y Final Fantasy XV (Square-Enix, 2016).

Figura 3.1: Captura de pantalla de Dark Souls 3 (©FromSoftware, 2016)

3.1.3.2 RPG estratégico

Menos populares históricamente, pero aún así relevantes, son los RPG de estrategia. En ellos, el posicionamiento de los integrantes del grupo pasa a ser un factor fundamental. Son habituales conceptos como el rango de movimiento de cada personaje, el alcance de un arma, e, incluso, la línea de visión. En algunos casos, el juego incluye mecánicas de fuego amigo, que fuerzan al jugador a considerar el posicionamiento relativo de sus unidades en combate. También resulta habitual que el terreno juegue un papel fundamental, premiando así la toma de objetivos para obtener la ventaja estratégica sobre el enemigo.

Un buen ejemplo de ese último punto podría ser el aumento de alcance proporcionado por la altura a las armas de larga distancia, como arcos, en la saga Tactics Ogre, o el aumento de evasión proporcionado por una casilla de bosque en la saga Fire Emblem.

Ejemplos notables publicados en la última década incluyen Fire Emblem Awakening (Intelligent Systems, 2012), Tactics Ogre: Let us cling together (Square-Enix, 2011) y Disgaea 5: Alliance of Vengeance (Nippon Ichi Software, 2015).

Figura 3.2: Captura de pantalla de Tactics Ogre: Let Us Cling Together (©Square-Enix, 2012)

3.1.3.3 RPG multijugador masivo en línea

Caracterizados únicamente por el hecho de ser juegos exclusivamente multijugador y requerir una conexión a Internet. Los jugadores deben cooperar para poder progresar, aunque, en ocasiones, existen facciones en oposición mutua.

Pese al precedente sentado por World of Warcraft (Blizzard, 2004), existe gran diversidad en términos de jugabilidad y diseño. Resultaría, por tanto, imposible describir en mayor detalle todas las posibilidades.

Ejemplos notables incluyen el ya mencionado World of Warcraft, Everquest (Verant Interactive, 1999), Runescape (Jagex, 2001), Dofus (Ankama Games, 2004) y Final Fantasy XIV: A Realm Reborn (Square-Enix, 2013).

Figura 3.3: Captura de pantalla de World Of Warcraft (©Blizzard, 2004)

3.1.3.4 Roguelikes

Llamados así por imitar al clásico Rogue (Glenn Wichman, 1980), presentan normalmente una perspectiva cenital y el objetivo principal del jugador es, generalmente, explorar laberintos generados aleatoriamente, tratando de conseguir la mayor cantidad de tesoros posibles antes de ser derrotados. Es habitual que la muerte de un personaje sea permanente, obligando al jugador a empezar desde el principio una vez termina la partida.

Ejemplos notables incluyen Shiren the Wanderer: The Tower of Fortune and the Dice of Fate (Spike Chunsoft, 2010), Pokémon Mundo Misterioso: Exploradores del Cielo (The Pokémon Company/Chunsoft, 2009) y Etrian Mystery Dungeon (Atlus, 2015).

Figura 3.4: Captura de pantalla de Shiren the Wanderer: The Tower of Fortune and the Dice of Fate (©Spike Chunsoft, 2010)

3.1.3.5 Dungeon Crawlers

Junto a a los roguelike, el subgénero más clásico. Consisten, en su versión más básica, en explorar una laberíntica mazmorra en busca de tesoros. Normalmente, el juego se representa en primera persona, y el jugador ha de crear un grupo de aventureros con el que superar los diversos obstáculos que el juego pondrá en su camino, desde monstruos hasta trampas, pasando por puertas cerradas con llave y caminos secretos.

Este subgénero perdió popularidad a finales de los años 80 y principios de los noventa, debido a su complejidad. No obstante, experimentó un resurgimiento en el año 2007, de la mano de Etrian Odyssey, desarrollado por Atlus.

Algunos ejemplos notables de la última década incluyen Etrian Odyssey V: Beyond the Myth (Atlus, 2016), Stranger of Sword City (Experience Inc., 2014), Class of Heroes 2 (Zerodiv, 2009) y Elminage Original (Starfish SD, 2011).

Figura 3.5: Captura de pantalla de Etrian Odyssey IV: Legends of the Titan (©Atlus, 2012)

3.2 Accesibilidad

El concepto de accesibilidad se refiere a la posibilidad para cualquier tipo de usuario de aprovechar diversos servicios, contenidos y productos con independencia de sus limitaciones personales. En el contexto concreto de la informática, la accesibilidad se refiere a un conjunto de consideraciones de diseño y técnicas que, al aplicarse, permiten el uso de un producto o servicio, sin importar su perfil.[3]

Este concepto suele cobrar importancia cuando se trata de garantizar el acceso de personas con diversidad funcional de cualquier índole a toda clase de servicios, dado que ciertas necesidades especiales

exigirán consideraciones concretas.

Dado el creciente desarrollo que la industria del videojuego ha experimentado desde su popularización a principios de los años 80, y las dificultades específicas que la misma naturaleza del medio plantea para la creación de contenido accesible, no es de extrañar que la accesibilidad en videojuegos se haya convertido en un área de investigación digno de mención.

Cabe destacar que la aplicación de técnicas de accesibilidad repercute positivamente en todos los usuarios, y no solo en aquellos con diversidad funcional, al proveer más herramientas para facilitar la comprensión y disfrute del contenido ofrecido.

Los tipos de impedimento que un usuario puede experimentar debido a sus condiciones personales al tratar de acceder a un videojuego se dividen en tres grandes grupos: sensoriales, motrices y cognitivas.

A continuación se estudiará, brevemente, el estado de la accesibilidad tanto en un contexto general como en el ámbito específico de los videojuegos, centrándose fundamentalmente en un perfil de diversidad funcional visual por ser el foco de este trabajo de fin de grado. Para una visión más generalista al respecto, se recomienda consultar el trabajo realizado por Sergio Sánchez López.[4]

3.2.1 Diversidad funcional sensorial

La diversidad funcional sensorial engloba todos aquellos tipos de diversidad funcional que involucren a los sentidos. Así pues, los tipos más notables en el ámbito de la tecnología son la diversidad funcional visual y la diversidad funcional auditiva.

Diversidad funcional visual

La diversidad funcional visual se refiere a toda la gama de defectos o alteraciones de la visión que pueden dificultar la visualización o comprensión de imágenes. Se identifican tres tipos principales: ceguera total, ceguera parcial y daltonismo.

- La ceguera total se refiere a la falta de visión que no puede ser corregida con lentes. Así pues, imposibilita jugar a aquellos juegos que requieran de un soporte visual para transmitir información esencial a los jugadores. La imposibilidad para el disfrute del producto audiovisual viene dada de la depencencia de un soporte de carácter visual para obtener información esencial, de modo que, en aquellos casos en que exista una alternativa acústica o háptica, no habrá impedimentos notables.

- La ceguera parcial se refiere a la capacidad de detectar luz de forma muy limitada. En algunos casos podría ser posible detectar movimiento. En términos más específicos, se considera que una persona padece ceguera parcial cuando su agudeza visual, tras aplicar las debidas correcciones, es de 20/70, o su campo de visión de 20° o menos. Es importante, asimismo, evaluar la definición de ceguera legal acuñada por la OMS, la cual incluiría a aquellos individuos con una agudeza visual de 20/200 tras aplicar medidas correctivas, o, una vez más, a aquellos con un campo de visión de 20° o menos. En estos casos, el usuario podría ser capaz de acceder a información visual, asumiendo que cuente con magnificación suficiente.

- El daltonismo se refiere a la incapacidad del individuo para detectar determinados colores. Se distinguen múltiples perfiles, en función del número de colores que no resulta posible detectar, y de qué colores conforma dicha gama. La gama de posibilidades comprende desde la rara acromatopsia, esto es, la incapacidad para distinguir cualquier color, hasta condiciones más comunes como la deuteranopia (incapacidad para distinguir el rojo y el verde), la protanopia (incapacidad para

distinguir tonos rojizos), o la tritanopia (incapacidad para distinguir los azules). Los usuarios con este perfil serán capaces de acceder a cualquier información visual que no esté ligada al color, pero podrían encontrarse con dificultades distinguiendo elementos con bajo contraste entre sí. Ciertas gamas de colores acentúan estos problemas, al poner énfasis en tonos como el rojo y el verde, de modo que es aconsejable proveer paletas alternativas en caso de recurrir a esa clase de combinación de colores.

Diversidad funcional auditiva

La diversidad funcional auditiva comprende todas aquellas alteraciones que dificulten o imposibiliten el acceso a información sonora. Así pues, se distinguen dos tipos fundamentales: la sordera, y la dificultad auditiva.

- La sordera es la imposibilidad de percibir o distinguir estímulos auditivos. Puede atender a causas variadas, de carácter genético, patológico o accidental, y se debe a una pérdida de audición profunda. Pese a la posibilidad de utilizar ayudas para el oído, estas son habitualmente insuficientes para acceder a toda la información sonora disponible. Resulta, por ello, fundamental proveer un soporte visual alternativo con el que transmitir información esencial a esta clase de usuarios.

- Se define la dificultad auditiva como un espectro de condiciones que impiden percibir sonidos inferiores a cierta intensidad. El umbral de entrada para considerar que un usuario padece dificultad auditiva son los 25 dB. Puede afectar a uno o ambos oídos, aunque ambos han de ser incapaces de detectar sonidos por debajo del umbral de 25 dB anteriormente mencionado para que se considere que esta condición está presente. Esta clase de usuarios puede a menudo requerir de un mayor volumen sonoro o el uso de ayudas externas para que las pistas auditivas les resulten genuinamente útiles, por lo que, al igual que con un perfil de sordera, es recomendable la inclusión de alternativas visuales para facilitar el acceso del usuario a la información, así como su interacción con el producto.

3.2.2 Pautas de accesibilidad

Existen a día de hoy múltiples pautas de accesibilidad dependientes de su contexto de aplicación. Destacan ejemplos como WCAG (actualmente en su versión 2.1), orientado a web; la Microsoft Human Interface Guidance, orientada a aplicaciones de escritorio; la guía de Accesibilidad de Apple, o la guía de accesibilidad para aplicaciones móviles elaborada por Juan Aguado Delgado y Francisco Javier Estrada Martínez, y ofrecida por la Secretaría General de Administración Digital.[5][6][3]

Todos estos ejemplos contienen elementos comunes que pueden aplicarse a desarrollos de índoles diferente al contexto de aplicación original de la correspondiente guía, de modo que resulta posible estudiar qué pautas tendrían sentido en el contexto de un videojuego y aplicarlas.

3.2.3 Pautas de accesibilidad para videojuegos

No obstante, existe también una guía de accesibilidad elaborada específicamente para su aplicación en el desarrollo de videojuegos. Más concretamente, se trata de la Game Accessibility Guidelines, disponible como recurso web, y que, además de incluir numerosas recomendaciones organizadas por perfil y complejidad de aplicación, incorpora ejemplos prácticos de la aplicación de cada una de esas pautas.

3.2.4 Estado de la accesibilidad en la industria del videojuego

La misma naturaleza de un videojuego supone un reto de cara a la creación de experiencias accesibles para determinados perfiles. Es un hecho que se han realizado progresos al respecto, dada la existencia de un compendio de pautas específicas para su aplicación en este contexto, pero queda un largo camino por recorrer.

La aplicación de ideas útiles de cara al desarrollo de videojuegos accesibles no es, de hecho, algo nuevo. Debido a limitaciones técnicas, algunos de los títulos mencionados en el breve vistazo realizado en la sección anterior a la historia del RPG como género de videojuegos ya incorporaban, si bien de forma involuntaria y totalmente accidental, elementos que podrían facilitar el proceso. Un buen ejemplo podría ser la sencilla gama de colores utilizada por Wizardry: Proving Ground of the Mad Overlord, que, al basarse fundamentalmente en el uso de blancos y negros, ofrecía un alto nivel de contraste. Lo que es más, la mayor parte de información esencial se transmitía al jugador de manera textual, lo que podría servir como base para un desarrollo accesible en la actualidad, siempre y cuando el sistema de presentación de dichos textos fuese compatible con el lector de pantalla de la plataforma de destino, o el juego contase con algún otro medio para ofrecer una alternativa acústica al texto escrito.

Figura 3.6: Ejemplo de la interfaz de usuario del primer juego de la saga Wizardry

Huelga decir que, pese a todo, estos títulos presentaban multitud de problemas desde el punto de vista de la accesibilidad, precisamente por no tratarse de un elemento contemplado a nivel de diseño.

Para poder encontrar casos de verdadera accesibilidad en videojuegos deberemos cambiar nuestro foco a productos más recientes. Referentes como el uso de elementos del entorno con alto nivel de contraste en la versión de Doom lanzada al mercado en 2016 por Bethesda son mejores referencias, si bien incompletas, de accesibilidad.

No obstante, podemos encontrar ejemplos verdaderamente notables de aplicación de técnicas de accesibilidad, incluso para perfiles tan desafiantes como la ceguera total. Killer Instinct, en su edición de 2013, presentaba diferentes sonidos para cada acción de cada personaje, así como un completo sistema de panning sobre el mismo para permitir a los usuarios invidentes calcular la distancia entre su personaje y el del adversario al tiempo que se identificaba, de forma rápida y unívoca, las acciones realizadas por cada personaje en cada momento. Esta misma técnica se aplicaría también en diversas entregas de la afamada saga Mortal Kombat. Se evidencia, por consiguiente, que un buen diseño de sonido puede resultar una herramienta fundamental de cara a transmitir información a aquellos jugadores que no pueden acceder a la misma de forma visual. Sin embargo, podemos observar que los dos títulos aquí mencionados comparten una serie de características específicas: el entorno es limitado, y la interacción con el mismo es mínima; asimismo, en todo momento hay únicamente dos entidades impactando al desarrollo del juego,

Figura 3.7: Doom utiliza luces en el entorno para marcar objetivos al jugador

y, pese al gran avance que supone ser capaz de jugar en absoluto, los jugadores con discapacidad visual siguen encontrándose en una cierta desventaja debido, precisamente, a la falta de información visual complementaria utilizada, por ejemplo, para telegrafiar grandes ataques, reduciendo como consecuencia la ventana de tiempo disponible para su reacción.

En una línea similar, aunque de impacto más limitado, encontramos la posibilidad de personalizar la visualización del juego que ofrecen determinados productos. Desde la alteración de elementos importantes de la interfaz gráfica y modificación de la posición de la cámara respecto al jugador hasta personalización de la paleta de colores en los casos más avanzados. Un buen ejemplo de estas ideas es Final Fantasy XIV, que en su parche 4.3 incluyó precisamente la gestión de la paleta de colores. Dicha gestión se encontraba limitada a la selección de un perfil de daltonismo, siendo protanopia, deuteranopia y tritanopia, y al grado de influencia del filtro aplicado mediante el slider incluido como parte de la interfaz. Pese a que este cambio fue parcialmente criticado por el público al que estaba destinado a ayudar, precisamente debido a las limitaciones que presenta, sí que se reportaron beneficios para ciertos perfiles, pudiendo considerarse, por tanto, un éxito parcial, así como un paso en la dirección correcta. [7]

No es sorprendente, no obstante, que fuese Final Fantasy XIV uno de los primeros MMORPG en introducir esta opción dentro de su ya de por sí amplia variedad de elementos de interfaz configurable. El título ya había presentado, desde su actualización 2.0, numerosas consideraciones para facilitar el disfrute de personas con disfunciones visuales de carácter menor: la configuración del tamaño de todos los elementos de la interfaz, así como de su posición; la personalización de los colores empleados en la ventana de chat, así como la posibilidad de mostrar etiquetas para indicar el canal en que ha sido enviado un mensaje, y los bordes de alto contraste en los marcadores de área de efecto, tanto de habilidades empleadas por enemigos como por aliados, eran características básicas desde dicho parche.

Queda, por supuesto, mucho camino por recorrer antes de que la totalidad de la industria pueda considerarse accesible, debido, en gran medida, a la naturaleza más artística del medio, que suele plantear mayores dificultades de cara a implementar pautas de accesibilidad tradicionales, con independencia de su perfil. Los ejemplos aquí citado demuestra, no obstante, que la posibilidad existe, y que numerosos desarrolladores han comenzado ya a intentar abrir la puerta a aquellos usuarios que, por sus circunstancias personales, no podrían disfrutar este medio de ningún otro modo.

3.3 LibGDX

LibGDX es un motor de videojuegos multiplataforma escrito en Java y basado en Gradle con un profundo enfoque en el desarrollo de juegos 2D. El motor provee un wrapper para OpenGL, así como herramientas

Figura 3.8: Ventana de ajuste de la opción de filtrado de color para personas con daltonismo en Final Fantasy XIV

de gestión y compilación de proyectos compatibles con Android, HTML5, iOS, Linux, MacOS y Windows.

En la actualidad se encuentra en su versión 1.9.9, con la versión 1.9.10 disponible en fase de desarrollo, y cuenta con numerosos módulos externo compatibles que facilitan el desarrollo de software multimedia de entretenimiento. Entre los más destacables se encuentran Box2D y gdxAI, aunque más adelante se ofrecerá una lista más completa de las funcionalidades aportadas por este motor.

El motor se ofrece como un ejecutable Java llamado gdx-setup.jar, que permite establecer la configuración básica de cada nuevo proyecto en términos de módulos de manera sencilla, y genera un proyecto Gradle importable en cualquier entorno de desarrollo integrado compatible con dicha tecnología. Cabe destacar que dicho proyecto Gradle cuenta con un subproyecto por cada plataforma de despliegue seleccionada, así como con un subproyecto adicional, llamado core, para implementar todos aquellos elementos independientes de la plataforma de destino, es decir, que conformen el grueso de la aplicación final.

3.3.1 LibGDX 1.0

Liberada oficialmente el 20 de abril de 2014, y tras un largo proceso de desarrollo que comenzó en 2009 [8], la primera versión estable del motor presentaba múltiples características que permanecerían en sus versiones subsiguientes: desarrollo basado en Gradle; soporte multiplataforma; manejo de gráficos y audio; compatibilidad con Box2D como motor de físicas; funcionalidades de cámara y viewport; soporte integrado para manejo de interfaz de usuario, y soporte para todos los IDE de Java.

En gran medida, el producto final fue el reflejo de desarrollos similares de la época, como el PlayN de Google, que ganó presencia durante 2012 y 2013. No obstante, el soporte de funcionalidades más complejas, incluso en el backend web del motor, acabaría dando prevalencia a LibGDX en los años venideros. Cabe destacar que, incluso antes del lanzamiento de la primera versión estable de LibGDX, ya se habían llevado a cabo desarrollos profesionales utilizándolo, de acuerdo con su autor. [8]

Adicionalmente, ya en esta versión, Box2D se consideraba una extensión del motor, en contraste con versiones anteriores, en las que era una piedra angular del motor. [9]

3.3.2 LibGDX 1.1

Liberada el 23 de mayo de 2014, la versión 1.1.0 de libGDX no solo arreglaba múltiples errores conocidos, sino que añadía múltiples funcionalidades y utilidades nuevas. Además de ofrecer compatibilidad con la versión 19.1.0 de las herramientas de compilación de Android. Además, añadió soporte para internacionalización; utilidades como funciones de manejo de color para Strings y de modificación de densidad de píxeles para facilitar el proceso de pruebas en un ordenador, y compatibilidad con JPGD. Asimismo, se alteró el comportamiento de funciones relacionadas con el manejo de matrices de traslación. [10]

3.3.3 LibGDX 1.2

Liberada el 22 de junio de 2014, los cambios más notables son la inclusión de utilidades de profiling relacionadas con OpenGL; compatibilidad con fuentes de tipo TTF mediante AssetManager; funciones de comparación para rectángulos, así como otros métodos de utilidad a clases pre-existentes, y la inclusión gdx-ai como motor para algoritmos de inteligencia artificial. Asimismo, se realizaron cambios notables en el comportamiento de determinados componentes, tales como Animation#frameDuration y Animation#animationDuration, que pasaron ha estar ocultos tras un conjunto getter/setter de métodos para estandarizar su manejo. [11]

3.3.4 LibGDX 1.3

Liberada el 9 de agosto de 2014, esta versión se centró fundamentalmente en ampliaciones de API.

Dichas ampliaciones se centraron, fundamentalmente, en ampliar funcionalidades previamente implementadas: se añadieron funcionalidades a las clases relacionadas con figuras geométricas básicas, se añadió un nuevo modelo de máquina de estado al módulo de IA, y se alteró el funcionamiento de métodos preexistentes en múltiples clases. Asimismo, se añadió compatibilidad con el nivel 19 de la API de Android. [12]

3.3.5 LibGDX 1.4

Liberada el 10 de octubre de 2014, esta versión se centró fundamentalmente en ampliaciones de API, así como ampliación de compatibilidades con versiones más recientes de los IDE soportados.

Además de actualizar la compatibilidad con Gradle y las herramientas de compilación de Android, las actualizaciones a la API se centraron en proveer mayor variedad de posibilidades al programador, mediante el reemplazo, por ejemplo, del método Actor#setCenterPosition(x, y) por Actor#setPosition(x, y, align). [13]

3.3.6 LibGDX 1.5

Liberada en diciembre de 2014, esta versión tuvo como foco principal mejorar la compatibilidad con iOS, así como mejorar las funcionalidades de red ofrecidas por el motor mediante diversas modificaciones a las clases relacionadas con Http, tales como la inclusión de constantes para las cabeceras de peticiones Http.

Asimismo, en posteriores versiones de LibGDX 1.5, además de mejorarse la compatibilidad con Gradle, se realizaron múltiples refactorizaciones para ampliar la funcionalidad de métodos preexistentes, incluyendo la exposición del método ControllerManager#clearListener.

Finalmente, los métodos Node#children y Node#parent se declararon como obsoletos, siendo reemplazados por la bandera inheritTransform. [14]

3.3.7 LibGDX 1.6

Liberada en mayo de 2015, trajo consigo revisiones a las herramientas auxiliares empaquetadas con el motor, fundamentalmente a 2D ParticleEditor, así como mejores a la gestión de gráficos, y expansiones a la API de reflexión de código. En este último caso, dichas mejoras estuvieron fundamentalmente enfocadas a mejorar la anotación de código en proyectos multiplataforma.

Como ya había ocurrido con la versión 1.5 del motor, esta versión contó con múltiples revisiones, las más notables enfocadas, por un lado, a mejorar la compatibilidad con RoboVM, y por otro a mejorar la compatibilidad con funcionalidades específicas del editor Tiled, tales como el uso de elementos animados para la generación de mapeados.

Cabe destacar que se realizaron numerosas mejoras de errores, y que se mejoró la estabilidad general del motor mediante la adición de sistemas de gestión de errores a la clase AssetManager. [15]

3.3.8 LibGDX 1.7

Liberada en septiembre de 2015, y distribuida en tres revisiones, su primera revisión se enfocaba únicamente a la expansión del manejo de cursores para interfaces gráficas de usuario, y arreglaba problemas de compatibilidad con UTF-8, al mismo tiempo que expandía la compatibilidad con RoboVM y, por consiguiente, con iOS.

Sus revisiones, no obstante, mejoraron el manejo de elementos gráficos y de audio, añadieron nuevas posibilidades para el uso de vectores, y mejoraron la interpretación de archivos JSON. [16]

3.3.9 LibGDX 1.8

Liberada en enero de 2016, esta versión del motor no contó con posteriores revisiones regulares, a diferencia de sus predecesoras. En ella, se introdujeron numerosos cambios a nivel de API.

En primer lugar, se redefinió el modo en el que el módulo Graphics daba acceso a los monitores disponibles en un equipo, facilitando la detección del monitor principal para hacer más sencilla la configuración en caso de ser necesario. Asimismo, se alteró el acceso a la información referente a los modos de representación disponibles, para poder dar cabida a la posibilidad de que distintos monitores presenten compatibilidad con distintas gamas de configuraciones. Dichos cambios impactaron también el manejo de resolución en pantalla completa y ventana.

Además, se introdujo por primera vez compatibilidad con pantallas HDPI para dispositivos móviles. Estos cambios separaron finalmente la generación de gráficos de su representación en dispositivos móviles, permitiendo estandarizar las operaciones realizadas entre distintas plataformas.

Se modificaron, también, las opciones de cursor, para arreglar problemas relacionados con fugas de memoria, y para permitir el uso de cursores del sistema y no solo de cursores personalizados.

Finalmente, se suprimió el método Sound#setPriority, dada su escasa utilización, y se migró el backend de LWJGL 2 a LWJGL 3, lo que supuso cambios profundos en el comportamiento del lanzador de aplicaciones, al tiempo que introdujo por primera vez listeners para eventos de ventana y soporte para múltiples ventanas en una sola aplicación, además de incluir soporte nativo para controladores, como podrían ser joysticks y gamepads. [17]

3.3.10 LibGDX 1.9

Liberada en enero de 2016, 1.9.x es la versión actual del motor. Cuenta con 11 revisiones, desde 1.9.0 hasta 1.9.10 en la actualidad. [18]

Dado que se trata de la versión más reciente, a continuación se listarán sus características más significativas, algunas de las cuales estaban presentes en versiones anteriores. Esta lista ha sido extraída de la página oficial del motor[19].

- Desarrollo multiplataforma basado en Gradle. Capaz de desplegar en Windows, Linux, Mac OS X, Android (versiones 2.2 en adelante), BlackBerry, iOS, como Java applet o como aplicación web (basada en Javascript y WebGL).
- Compatibilidad con Spine, Nextpeer y Saikoa.
- Basado en OpenGL. Ofrece una API de alto nivel para acceder a sus funciones de dibujo en caso de ser necesario, además de permitir aprovecharlas con una completa librería gráfica ya incluída con el motor.
- API de escenas 2D y generación de interfaces en la forma de Scene2D.
- Compatibilidad con el editor de niveles Tiled, y con archivos en formato TMX.
- Compatibilidad con modelos en formato Wavefront OBJ, MD5 y FBX.
- Capacidad de reproducir música y sonidos en streaming, en formato WAV, MP3 u OGG.
- Soporte de acceso directo a dispositivos de audio para reproducción de audios en formato PCM y grabación de sonidos.
- API abstracta para manejo de ratón, pantalla táctil, teclado, acelerómetro y brújula.
- Capacidad para procesar gestos en pantallas táctiles.
- Completar librería matemática, acelerada mediante el uso de código C nativo.
- Compatibilidad con físicas 2D basadas en Box2D.
- Compatibilidad con físicas 3D basadas en bullet physics.
- Abstracción del sistema de archivos para proporcionar compatibilidad con todas las plataformas.
- Emulación de solo lectura para el backend de Javascript.
- Soporte de archivos binarios en el backend de Javascript.
- Sistema ligero de preferencias.
- Definición personalizada de colecciones, con soporte de tipos primitivos.
- Sistema de escritura y lectura de archivos JSON, compatible con POJO.
- Sistema de escritura y lectura de XML.
- Editor de partículas 2D y 3D.
- Empaquetador de texturas.
- Generador de fuentes en formato bitmap.

Además, el motor facilita el prototipado rápido y el desarrollo iterativo aprovechando características de la versión de escritorio de Java como hot swapping, lo que permite modificar partes específicas del proyecto en tiempo de ejecución sin necesitar compilarlo en su totalidad.

En su estado actual, LibGDX carece de un módulo de accesibilidad nativo, o de compatibilidad directa con herramientas de accesibilidad tales como lectores de pantalla.

Capítulo 4

Desarrollo del proyecto

A continuación se detalla el proceso seguido para el desarrollo del proyecto, incluyendo un análisis de las herramientas utilizadas, el diseño a nivel de juego, y la implementación software realizada. Dicha implementación software contemplará, asimismo, tanto la implementación del prototipo de videojuego propiamente dicho, como la implementación de pautas de accesibilidad a nivel técnico en el mismo.

El proceso de desarrollo software ha sido realizado de manera iterativa, con el objetivo de facilitar una comprensión gradual de los componentes fundamentales de las herramientas empleadas para dicho desarrollo, así como de las técnicas de accesibilidad aplicables. Lo que es más, esta mejor comprensión de las herramientas empleadas permitirá pulir, de manera gradual, el resultado final, así como reevaluar el enfoque adoptado en aquellos elementos que no funcionen del modo esperado.

4.1 Análisis: LibGDX

LibGDX es un motor de videojuegos multiplataforma escrito en Java. Pese a presentar soporte para desarrollo 3D, se trata de un motor claramente diseñado para desarrollo de videojuegos 2D, y se caracteriza por su gran extensibilidad gracias a la multitud de paquetes compatibles existentes. Permite el despliegue en cualquier sistema operativo de PC compatible con Java, así como Android, iOS, y web en formato HTML5.

El motor propiamente dicho se presenta en forma de librería, accesible mediante el uso de Gradle, y viene empaquetado con un generador de proyectos para simplificar el comienzo del proceso de desarrollo. En su forma más básica, ofrece un wrapper para OpenGL y OpenAL, así como para la gestión de entradas en todas las plataformas de despliegue anteriormente descritas. Además, la estructura de proyecto más básica permite diversificar claramente el desarrollo de funcionalidades internas del de funcionalidades específicas de cada plataforma de destino, así como hacer interfaz entre ellas.

Dicho esto, LibGDX carece de un editor de niveles propio, así como de compatibilidad con sistemas de scripting, e incluso de un sistema de máquina de estados básico. Todo ello son funcionalidades que, en caso de ser necesarias, deberán añadirse externamente. Por fortuna, LibGDX contiene un intérprete para niveles implementados mediante Tiled, y no presenta incompatibilidades de ningún tipo con bibliotecas de interfaz con lenguajes de scripting.

LibGDX presenta también compatibilidad directa con las librerías Scene2D, Box2D y GDX-AI. La implementación de una interfaz gráfica de usuario debería, por tanto, verse simplificada, gracias a Scene2D,

y la implementación de físicas bidimensionales, en caso de ser necesaria, no debería de suponer excesivos problemas gracias a Box2D. Algunas de las funcionalidades incluidas en GDX-AI podrían suponer también una gran ayuda de cara a implementar el comportamiento de determinados oponentes.

Cabe destacar que, dado el funcionamiento interno del motor, el mismo no presenta compatibilidad nativa con lectores de pantalla, pese a tener un completo sistema de generación de interfaces. Será necesario, por tanto, diseñar una solución para poder cubrir ese frente. Será necesario, también, implementar una máquina de estados adaptada a las necesidades del proyecto, así como un sistema de entidades completo.

Como entorno de desarrollo puede emplearse cualquier IDE compatible con Gradle. Se ha seleccionado NetBeans 8.2 por poseer experiencia previa con su manejo. Asimismo, la versión de Java requerida para poder desplegar en plataformas Android el resultado final es Java 6, de modo que resultará imposible recurrir a algunas herramientas modernas, como inferencia de tipos en la instanciación de colecciones, o determinadas estructuras reminiscentes de una aproximación funcional. Sí será posible, no obstante, recurrir a librerías de terceros compatibles con Java, lo que resultará fundamental de cara a la adecuada implementación del proyecto.

Dadas las características del prototipo a implementar, se prescindirá de la mayoría de librerías accesorias incluídas con LibGDX. El motivo es simple: si bien en un proyecto de mayores dimensiones simplificarían el flujo de trabajo durante el desarrollo, el exceso de funcionalidad supone, de cara a nuestro caso de uso, un perjuicio, fácilmente evitable mediante la implementación directa de nuestros propios sistemas. Dada la necesidad, en todo caso, de personalizar el código ofrecido por el motor, o, al menos, construir en base a él, resulta razonable realizar implementaciones personalizadas de aquellas utilidades que se consideren imprescindibles.

4.1.1 Generación y uso de un proyecto LibGDX

LibGDX se distribuye como una utilidad para generar proyectos Gradle preconfigurados. El proceso es extremadamete sencillo: basta lanzar el generador y ajustar la configuración de acuerdo a nuestras necesidades. A continuación, la utilidad generará un proyecto Gradle, que podremos importar en el IDE de nuestra elección, asumiendo que sea compatible con Gradle en primer lugar. Como resultado, Eclipse resulta una elección clásica para trabajar con LibGDX, dada su integración directa con Gradle, pero la mayoría de versiones de NetBeans, así como de IntelliJ IDEA, son opciones perfectamente válidas.

Figura 4.1: Interfaz del generador de proyectos de LibGDX.

Las opciones de configuración regulan los siguientes aspectos:

- **Name**: nombre del proyecto a generar. No puede ser refactorizado posteriormente de forma nativa.
- **Package**: nombre del paquete de código generado inicialmente. Puede ser refactorizado posteriormente.
- **Game Class**: nombre de la clase Java que actuará como punto de entrada al juego. El generador la almacenará en el paquete definido en **package**.
- **Destination**: directorio local en el que generar el proyecto.
- **Android SDK**: ruta a la instalación local del SDK de Android. Solo es obligatorio si se pretende realizar despliegue a dicha plataforma.
- **Sub Projects**: lista de plataformas de despliegue deseadas. Deberemos marcar todas aquellas para las que queramos poder realizar despliegue posteriormente, y será necesario marcar al menos una para que el juego pueda siquiera ejecutarse.
- **Extensions**: extensiones nativas de LibGDX que deseamos utilizar en nuestro proyecto. Solo se incluirán aquellas que marquemos en este punto.

Asimismo, podemos ganar más control sobre la configuración del proyecto mediante el uso de extensiones desarrolladas por terceros y las opciones de configuración avanzadas. Concretamente, las opciones de configuración avanzadas permiten generar proyectos para Eclipse o IDEA sin usar Gradle.

En cuanto a las extensiones incluídas con el motor, sus funciones son las siguientes:

- **Bullet**: Detección de colisiones en 3D y manejo de cuerpos rígidos.
- **FreeType**: Manejo de fuentes escalables. No es compatible con distribuciones HTML, que se compilarán usando el sistema de fuentes incluído con LibGDX por defecto.
- **Tools**: Conjunto de utilidades accesorias, tales como un editor de partículas, un generador de fuentes en bitmap y un empaquetador de texturas.
- **Controller**: Manejo de entrada desde controladores externos, tales como mandos de Xbox360 o joysticks.
- **Box2d**: Manejo de físicas en 2D.
- **Box2dlights**: Manejo de iluminación, usando **Box2d** como framework, y OpenGL ES 2.0 para dibujar en pantalla.
- **Ashley**: Framework de manejo de entidades.
- **Ai**: Framework de inteligencia artificial. Especialmente útil para resolver problemas de pathplanning en el manejo de NPCs.

Una vez hayamos establecido la configuración deseada, bastará con hacer click en *Generate*, y el generador se encargará de configurar el proyecto. Dado que para la elaboración de este TFG se ha utilizado NetBeans 8.2, el proyecto generado será compatible con Gradle. Además, se prescindirán de la mayor parte de extensiones, manteniendo, únicamente, **Tools** como parte del proyecto.

4.1.2 Estructura de un proyecto LibGDX

Una vez completada la generación, podremos importar el proyecto Grade en el IDE de nuestra elección. Dicho proyecto contendrá, a su vez, un subproyecto por cada plataforma de despliegue elegida durante la generación, así como un subproyecto adicional llamado 'core'. La función de cada posible subproyecto es la siguiente:

- **core**: Código y recursos comunes a todas las distribuciones. La mayor parte del código formará parte de este subproyecto, dado que las estructuras de datos utilizadas para manejo de estados y entidades, así como las mecánicas de juego, serán independientes de la plataforma utilizada por el jugador. Permite, además, el acceso a sus elementos públicos de forma directa desde el resto de subproyectos, lo que permite, por ejemplo, generar interfaces para interactuar con código específico de cada plataforma de despliegue de forma sencilla y elegante.
- **desktop**: Código y recursos específicos para distribuciones de ordenador. Dado que el resultado es un programa Java, este código será compatible con cualquier sistema operativo compatible con dicha tecnología.
- **android**: Código y recursos específicos para distribuciones de Android. Contiene, además, un archivo Manifest.xml, así como directorios específicos para esta distribución.
- **html**: Código y recursos específicos para distribuciones HTML. Contiene también una plantilla para generar automáticamente el archivo WAR necesario para el despliegue de la aplicación en un servidor web.
- **ios**: Código y recursos específicos para distribuciones iOS.

Esta estructura permite simplificar en gran medida el flujo de trabajo, al no ser necesario realizar implementaciones específicas para cada plataforma excepto en aquellos casos en los que se necesite interactuar de forma directa con elementos concretos del sistema, como podrían ser las entradas generadas por el jugador. Lo que es más, al poder realizarse estas variaciones de forma transparente al núcleo de la aplicación, la lógica de negocio resultante será mucho más limpia.

Nótese que la filosofía general del motor enfatiza la aplicación de un modelo de arquitectura Modelo-Vista-Controlador, permitiendo realizar variaciones únicamente sobre la vista y partes muy específicas del controlador para generar código multiplataforma de manera ágil y sencilla.

4.2 Análisis: Tiled

Tiled es un editor de niveles 2D genérico basado en cuadrículas. El editor permite almacenar el resultado en formato TMX, que no es sino un archivo XML que define las propiedades del mapa, incluyendo tanto el conjunto de patrones empleado, como las propiedades que hayamos decidido definir para el mismo. Resulta, así, una herramienta flexible, que permite parametrizar en profundidad el resultado final.

El motor distingue, asimismo, entre capas de patrones y capas de objetos. Las capas de patrones definen, fundamentalmente, el aspecto gráfico del mapa generado. En circunstancias normales, ese aspecto se usará directamente para generar la representación gráfica del mapeado en el resultado final, al tiempo que se establecen las propiedades específicas de cada casilla para su interpretación en tiempo de ejecución. La capa de objetos, por otro lado, permite definir regiones con propiedades arbitrarias, que pueden posteriormente interpretarse por separado.

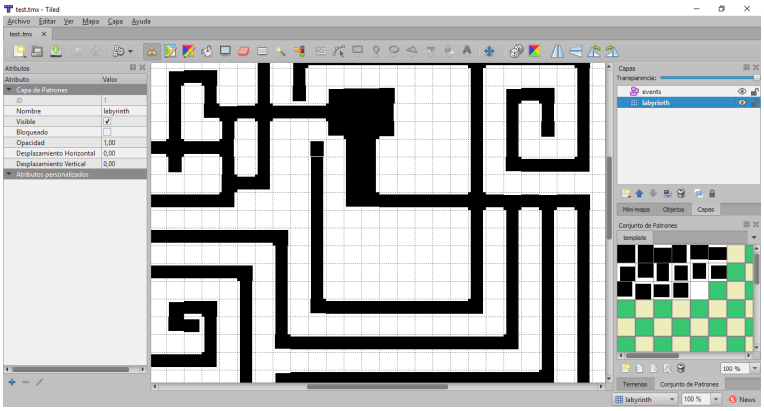

Figura 4.2: Interfaz de Tiled. En ella podemos ver el panel principal de edición de mapas, el panel de conjunto de patrones, el panel de capas, y las propiedades de una de dichas capas.

Así, por ejemplo, podremos utilizar una capa de patrones únicamente para establecer el aspecto visual de cada nivel, y definir las propiedades de colisión mediante una capa de objetos, o establecer propiedades de colisión para cada patrón, de modo que se interpreten directamente en la propia capa de patrones. Otros usos posibles para la capa de objetos incluyen la definición de eventos, la definición de puntos de origen para entidades e, incluso, la definición de elementos visuales de manera vectorial, que después podremos representar en el mapa.

Para el desarrollo de este TFG, nos interesan dos funcionalidades fundamentalmente: la posibilidad de establecer propiedades asociadas a patrones concretos, y la capa de objetos. Concretamente, las propiedades de patrones han sido utilizadas para facilitar la representación del escenario, como se detallará más adelante, mientras que la capa de objetos se ha aprovechado para establecer las posiciones de eventos concretos.

4.3 Diseño del videojuego

Procederemos, ahora, a detallar el prototipo que se ha implementado, así como explicar las decisiones que se han tomado durante el diseño del mismo. Dicho prototipo utiliza como ejemplo juegos del género dungeon crawler en primera persona, tales como Wizardry o Etrian Odyssey, e implementa versiones simplificadas de las mecánicas típicas en los mismos. Así pues, el prototipo contiene:

- Múltiples clases de personaje con las que poder formar un grupo de exploración.
- Sistemas de inventario y habilidades simplificados para demostrar su funcionalidad.
- Varios enemigos, con distintos comportamientos y características, que hacen las veces de retos.
- Un laberinto relativamente corto que explorar.
- Algunos eventos con los que interactuar en el laberinto.

4.3.1 Objetivo del juego

El objetivo del jugador es sencillo: explorar un laberinto plagado de peligros para conseguir riquezas y fama en un universo ficticio. La premisa básica es típica de esta clase de videjuegos, y sirve únicamente como justificación para explorar el laberinto principal. Dicha exploración se realiza en primera persona, y los

retos planteados al jugador aumentan en dificultad de manera gradual. El motivo es sencillo: al progresar en el juego, la comprensión de las mecánicasde juego por parte del jugador se irá incrementando, por lo que será necesario incrementar la dificultad para mantener el mismo nivel de tensión.

Es importante tener en cuenta que la misma naturaleza del juego implica la existencia de progresión de personaje. Esta progresión se produce de forma vertical: al ganar suficiente experiencia, el nivel de los personajes aumenta, y eso se traduce en mejores características y nuevas habilidades disponibles para su uso. Para el propósito de este TFG se ha optado por prescindir de la progresión horizontal, presente en la forma de múltiples opciones de equipo, habitual en el género. En su lugar, las características de cada personaje dependen exclusivamente de su clase, un pequeño componente aleatorio durante la generación de personaje, y su nivel actual.

4.3.2 Ambientación del juego

El juego se ambienta en una pequeña aldea situada junto a unas ruinas recién descubiertas. Dichas ruinas están plagadas de monstruos y, presumiblemente, de riquezas, lo que rápidamente llamó la atención de aventureros y cazadores de tesoros de todo el mundo. Conscientes de que la repentina afluencia de personas podía llevar a la prosperidad económica de la aldea, sus habitantes se esforzaron por promocionar la exploración del infame laberinto.

El jugador controla a un grupo de dichos aventureros, novatos en primera instancia, que tratarán de explorar las profundidades de las mortiferas ruinas para obtener riquezas y gloria. Los personajes que conforman este grupo serán generados al comienzo de la partida, pudiendo un equipo de exploración contener a un máximo de seis integrantes, y pudiendo el jugador tener múltiples personajes adicionales en reserva para ajustar su estrategia y aproximación a diversos retos en caso de ser necesario.

Las actividades del jugador se dividen en dos secciones claramente diferenciadas: la preparación en la aldea, donde puede configurar su grupo de aventureros como desee, así como gastar su oro en objetos útiles, y el laberinto, donde tendrá que hacer frente a grupos de enemigos de manera ocasional, y gestionar cuidadosamente sus recursos para evitar perecer de manera prematura.

El estilo visual elegido es minimalista: imitando la presentación del clásico Wizardry para MS-DOS, el laberinto se representa en un estilo *wireframe* en primera persona. Los menús y cajas de diálogo que componen el grueso de la interacción del jugador con el mundo del juego se representan en blanco y negro, con el objetivo de ofrecer un alto nivel de contraste que facilite el acceso de personas con visión reducida.

4.3.3 Personajes de jugador

A continuación se discutirán los aspectos fundamentales del diseño a nivel de juego de los personajes de jugador como entidades. Notablemente, se tratarán las características que los componen, en rasgos generales, así como las distintas clases de personajes disponibles y las propiedades de cada clase.

4.3.3.1 Características

Todos los personajes de jugador cuentan con las siguientes características, generadas de manera semi-aleatoria durante la creación de personaje. El valor de cada una de ellas depende de la clase de personaje y del resultado de múltiples tiradas de dado, y el jugador tiene siempre la opción de volver a generar una distribución de estadísticas antes de aceptar al personaje, en caso de considerarlo necesario:

- **Max_HP**: Valor máximo de salud para este personaje. Siempre que comience la exploración, el personaje tendrá este valor de salud, y nunca podrá recuperarse por encima de él.
- **Max_MP**: Valor máximos de los puntos de magia para este personaje.Siempre que comience la exploración, el personaje tendrá este valor de puntos de magia, y nunca podrá recuperarse por encima de él.
- **Cur_HP**: Valor actual de salud para este personaje. Si se reduce a 0, el personaje está muerto, y no puede hacer nada hasta que se le resucite. El valor de esta característica se reduce al recibir daño, y se incrementa al recibir curación, ya sea mediante habilidades u objetes. Se inicializa a *Max_HP* al comenzar la exploración.
- **Cur_MP**: Valor actual de puntos de magia para este personaje. Si se reduce a 0, el personaje no puede usar habilidades. Se reduce al utilizar habilidades especiales, y se recupera mediante el uso de objetos. Se inicializa a *Max_MP* al comenzar la exploración.
- **Agility**: Determina la agilidad del personaje, y, como consecuencia, su velocidad de actuación, así como su capacidad para acertar o esquivar ataques. No puede modificarse más allá de los cambios propiciados por una subida de nivel. Puede ser relevante para la resolución de determinados eventos en el laberinto.
- **Strength**: Determina la fuerza del personaje, y, como consecuencia, su capacidad para causar daño físico. Es un componente fundamental en las fórmulas de daño para habilidades que utilicen armas de corta distancia. No puede modificarse más allá de los cambios propiciados por una subida de nivel. Puede ser relevante para la resolución de determinados eventos en el laberinto.
- **Wisdom**: Determina la sabiduría del personaje, y regula su capacidad para sanar las heridas de sus compañeros de equipo mediante el uso de habilidades curativas. No puede modificarse más allá de los cambios propiciados por una subida de nivel. Puede ser relevante para la resolución de determinados eventos en el laberinto.
- **Defense**: Determina la defensa del personaje, y regula su capacidad de mitigar daño. No puede modificarse más allá de los cambios propiciados por una subida de nivel. Puede ser relevante para la resolución de determinados eventos en el laberinto.

4.3.3.2 Clases

A continuación se detallarán las distintas clases de personaje disponibles para el jugador. Los detalles de implementación se tratarán más adelante, en la sección correspondiente. Este apartado pretende únicamente proveer una visión preliminar, de carácter puramente conceptual, para facilitar la comprensión de las decisiones ténicas adoptadas.

Paladín

Especialistas en el aspecto defensivo, los paladines se centran en proteger al grupo. Con un apartado ofensivo mediocre, y habilidades centradas en la mitigación de daño, su papel es el de defensores, única y exclusivamente. Poseen las mejores habilidades defensivas de todas las clases, y, pese a su lentitud y torpeza, son expertos en proveer apoyo al grupo mediante objetos y hechizos que limitan los daños recibidos. Útiles en cualquier composición de grupo, asumiendo que se suplan sus carencias ofensivas con el uso de clases más adeptas en ese aspecto.

Los paladines suelen presentar el mayor valor de HP de todo el juego, siendo murallas naturales, y lo complementan con una cantidad de MP nada desdeñable. Sus hechizos se centran en crear barreras en torno al grupo, limitando el daño recibido en determinadas instancias, pero han de ser mantenidos de manera constante. Como consecuencia, el uso inteligente de estas barreras se vuelve fundamental en el trabajo de todo paladín: su papel es casi exclusivamente defensivo en las batallas más duras, pero no por ello menos importante, y gestionar sus recursos puede llegar a volverse un auténtico reto dependiendo del oponente al que se esté plantando cara.

No es recomendable contar con más de un paladín en la mayoría de grupos, gracias a que sus barreras protegen a la totalidad de los integrantes del mismo. Las habilidades defensivas de los paladines siempre se ejecutan con prioridad, siendo superadas por escasos ataques enemigos, pero cualquier intento de un paladín por ponerse a la ofensiva será lento y carente de precisión.

Guerrero

Poderosos en el aspecto físico, los guerreros son los atacantes por excelencia. Su resistencia física no es nada desdeñable, comparable a la de los clérigos, pero su papel fundamental es la de destruir a cualquier enemigo que trate de cortar el paso al grupo. Carecen de habilidades propias, y su mayor aporte es su capacidad para limitar el gasto de recursos en encuentros. Sus defensas pueden fácilmente verse beneficiadas por sus compañeros, pero su limitada inteligencia y sabiduría convierten a la mayoría de guerreros en cañones de cristal al hacer frente a oponentes con capacidad mágica. Un recurso habitual para todo tipo de grupo, pero que puede verse beneficiado por una buena sinergia con cualquier clase de carácter defensivo.

Los guerreros presentan el valor de fuerza más elevado de entre todas las clases, y poseen también una de las agilidades más altas. Su ratio de HP es aceptablemente alto, pero sus bajas capacidades mentales los vuelven vulnerables a ataques de carácter mágico, y susceptibles de ser eliminados rápidamente por enemigos poderosos. En una carrera de daño, contar con un guerrero y protegerlo cuidadosamente puede suponer la diferencia entre el éxito y el fracaso: su independencia de los puntos mágicos implican una capacidad para mantener el asalto de manera constante, de modo que el principal recurso a gestionar es, precisamente, el HP.

Clérigo

Poderosos curadores, los clérigos flaquean únicamente en el aspecto defensivo. Su durabilidad es más que respetable, aunque no comparable a la de los paladines, y poseen la capacidad de sanar a sus compañeros en caso de necesidad. Se dice, incluso, que los clérigos de más alta categoría son capaces de llevar a cabo milagros tales como la resurrección. Integrantes habituales de grupos de toda índole gracias a sus capacidades curativas, los clérigos agradecen la presencia de compañeros capaces de eliminar al enemigo rápidamente, para limitar la carga situada en el curador del grupo.

Los clérigos presenta el valor de sabiduría más elevado de entre todas las clases. Su durabilidad es similar a la de un guerrero, pero carecen de la fuerza para causar verdaderos daños por el lado físico. Sus habilidades curativas dependen de su MP, que, si bien no es terriblemente limitado, puede suponer un problema en caso de no ser gestionado adecuadamente. La clase en su conjunto gira, por tanto, en torno a determinar adecuadamente cuándo defender y cuándo curar, haciendo un uso efectivo de las habilidades de la clase.

Mago

Sabios, pero frágiles, los magos se especializan el uso de toda suerte de artes arcanas para facilitar la exploración al conjunto del grupo. Poseedores de hechizos tanto ofensivos como de alteración, los magos cumplen un papel fundamental de cara a limitar el riesgo presentado por los oponentes más resistentes físicamente. Pero, cuidado: su fragilidad física es su mayor debilidad, suponiendo un riesgo en toda clase de situaciones. Todo mago que se precie agradecerá la presencia de personas más fuertes y resistentes en las inmediaciones, a las que utilizar como escudo humano mientras preparan sus poderosos hechizos para aniquilar al enemigo. Otro componente habitual en cualquier grupo de exploración, gracias a la utilidad que pone sobre la mesa.

Los magos poseen el valor de inteligencia más alto de entre todas las clases del juego. Su daño mágico puede, incluso, superar al daño físico de los guerreros, pero está limitado a la cantidad de MP disponible. Además, el bajo HP convierte a los magos en los verdaderos cañones de cristal, haciendo imprescindible determinar cuándo es necesario adoptar una aproximación más defensiva. Esta clase gira, por tanto, en torno a la idea de gestionar los tiempos del combate para determinar el mejor momento para arrasar con la salud del enemigo, tratando de limitar los daños recibidos el resto del tiempo.

Pícaro

Truhanes, ladrones, criminales y asesinos, los pícaros se especializan en el subterfugio. Con multitud de habilidades para aprovechar el más mńimo descuido del enemigo, estos hábiles delincuentes adolecen únicamente de su extrema fragilidad. Quién sabe, quizá disfrutar de compñías tan indignas de confianza pueda tener sus ventajas en determinadas situaciones. Dadas sus propiedades especiales, los pícaros suelen presentar dificultades para integrarse en cualquier grupo; sin embargo, cuando lo consiguen, suelen convertirse en pilares fundamentales del mismo, facilitando la exploración gracias a sus habilidades para el engaño y el asesinato.

Los pícaros poseen características equilibradas en casi todos los aspectos, y poseen el mayor valor de agilidad de todo el juego, así como el segundo menor valor de HP de todas las clases, solo por encima del de los magos. Su valor de MP es reducido, pero la mayor parte de habilidades de los pícaros pueden emplearse sin usar puntos mágicos. La mayor parte de estas habilidades se centran en actuar antes que el enemigo, permitiendo rematar a aquellos oponentes que de otro modo podrán realizar acciones peligrosas antes de ser eliminados definitivamente, y en garantizar ataques críticos, facilitando la labor del resto del grupo. De forma similar a como pasa con los magos, su fragilidad implica que la clase en su conjunto requiere leer cuidadosamente el flujo del combate, para poder actuar en el momento preciso para optimizar el resultado obtenido, al tiempo que se limita el riesgo para el pícaro en cuestión.

Un pícaro bien empleado puede dar la vuelta a las situaciones más complicadas, pero, dadas sus características, esta clase puede dificultar encuentros de otro modo mundanos si no se utiliza cuidadosamente.

4.3.4 El pueblo

El pueblo es el lugar donde el jugador puede prepararse para las aventuras. Se presenta como un sencillo menú, desde el que acceder a todas las utilidades del pueblo, a saber:

- **taberna**: En la taberna es posible generar nuevos personajes, así como gestionar el grupo de aventuras actual. El jugador podría querer regresar ocasionalmente para reestructurar su estrategia y aproximación, y adaptarse más fácilmente a los restos planteados por el laberinto.

- **Tienda**: En la tienda pueden comprarse objetos curativos, útiles para la exploración, asumiendo que el jugador posea el dinero y espacio en el inventario necesarios para hacerlo. Como se explicará más adelante, para el propósito del prototipo desarrollado, se han implementado únicamente objetos curativos, pero, en caso de haber mayor variedad, estará disponible aquí. El único tipo de objeto que no puede comprarse en la tienda son los objetos clave, por tratarse de un tipo de objeto especial.

- **Posada**: La posada tiene como única función guardar la partida. El sistema de guardado de datos se detallará más adelante, pero en este prototipo es un servicio completamente gratuito, permitiendo guardar la partida tantas veces como se desee.

4.3.5 El laberinto

El laberinto es el lugar donde el jugador pasará la mayor parte del tiempo, y donde transcurre el grueso del juego. Plagado por peligros inmaginables, y compuesto por retorcidos pasillos, el interior del laberinto se encuentra envuelto en un perpetuo manto de oscuridad que dificulta la visión en todo momento. Nunca se sabe qué puede esconderse tras una esquina, o a dónde llevará una intersección. La única forma de descubrirlo es, obviamente, explorar.

La exploración se realiza con una vista de primera persona, en la que los controles permiten al jugador avanzar, retroceder o girarse a izquierda y derecha. En su interior pueden tener lugar diversos eventos, narrados empleando texto para facilitar la accesibilidad, o encuentros con enemigos. Los encuentros con enemigos pueden, en si mismos, producirse de manera aleatoria o en puntos prefijados, por lo que es fundamental gestionar cuidadosamente todos los recursos del grupo a lo largo de la exploración, y determinar cuándo es el momento de desandar el camino recorrido hasta el momento para poder reabastecerse.

4.3.5.1 Estructura del laberinto

El laberinto se compone de cinco plantas, con un nivel de dificultad creciente, en las que la atención del jugador es fundamental. Los grupos de enemigos y tipos de eventos presentes en las distintas plantas del laberinto también aumentan de dificultad según el jugador progresa, y para ello han de emplearse las escaleras. Nótese que en todo momento es posible avanzar a una nueva planta o retroceder a una anterior, asumiendo que se sepa llegar a las escaleras correspondientes.

A continuación, se detallarán las características de cada planta, y se presentará su trazado para facilitar el siguimiento. Las características de cada planta incluyen una descripción general, así como los enemigos presentes como encuentros aleatorios, y los distintos eventos existente. En cuanto a los eventos, se especifica su localización, sus condiciones de activación, y el resultado del evento.

Primera planta

La primera planta tiene como único objetivo servir como una toma de contacto con el juego, y su trazado es relativamente sencillo. Pese a la presencia de numerosos callejones sin salida, o caminos que vuelven sobre si mismos, no es difícil regresar a la entrada, o avanzar hasta la siguiente planta si se conoce el camino. Los enemigos en esta planta son poco amenazantes, y no deberían de suponer un gran riesgo para ningún grupo mínimamente bien construido. El objetivo, es, de hecho, presentar los conceptos más básicos en lo referente al combate: la necesidad de priorizar ciertos tipos de enemigos respecto a otros, la existencia de distintas especialidades entre los propios enemigos, y el hecho de que los grupos pueden contar con hasta tres enemigos a la vez.

Existe una sección aparentemente inaccesible, a la que ha de regresarse bajando desde la segunda planta, y que permite acceder a otra sección avanzada de dicha planta. Hay un combate prefijado antes de acceder a esas últimas escaleras, con un nivel de dificultad ampliamente superior similar al de los encuentros de la tercera planta; dicho encuentro ha de ser superado cada vez que se desee atravesar esta zona después de regresar al pueblo. Esta sección ha de recorrerse para llegar al final del prototipo.

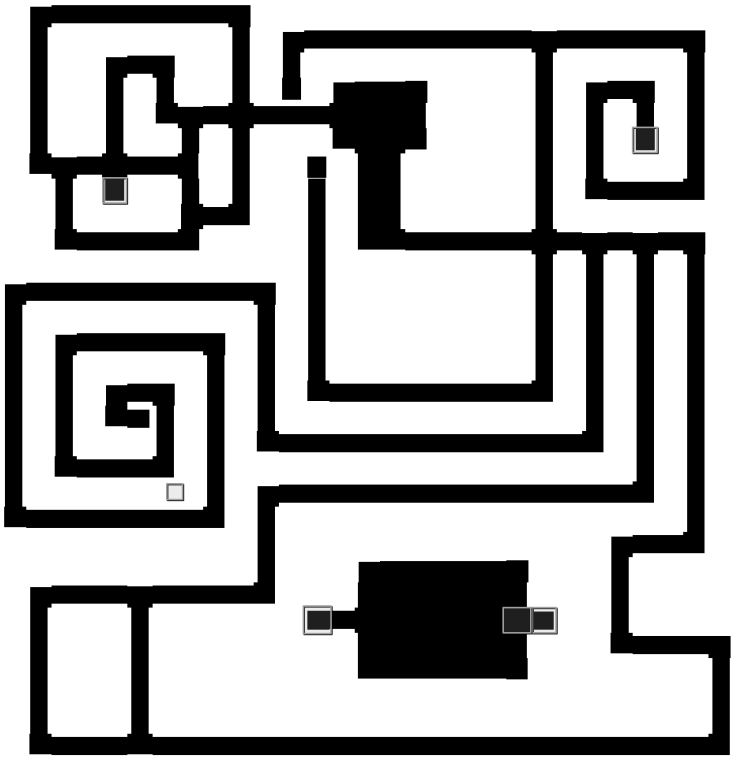

Figura 4.3: Mapa de la primera planta del laberinto.

En esta planta, el jugador solo puede enfrentarse a grupos de **Slimes**, **Heal Slimes** y **Goblins**. El ratio de encuentros permanentemente reducido, con el objetivo de facilitar la exploración y toma de contacto con el juego en sus primeros compases.

Los eventos presentes en esta planta son los siguientes:

- Ejemplo.
  - **Localización**: (x, y)
  - **Condiciones**:
    - Condición 1.
  - **Desarrollo**: explicar.
- Escaleras planta 2, parte 1.
  - **Localización**: (26, 6)
  - **Condiciones**:
    - Inicio automático al colisionar con el jugador.
  - **Desarrollo**: Desplaza al jugador a la segunda planta, en la posición (x, y).
- Escaleras planta 2, parte 2.

  - **Localización**: (13, 25)
  - **Condiciones**:
    * Inicio automático al colisionar con el jugador.
  - **Desarrollo**: Desplaza al jugador a la segunda planta, en la posición (x, y).

- Escaleras planta 2, parte 3.
  - **Localización**: (22, 25)
  - **Condiciones**:
    * Inicio automático al colisionar con el jugador.
  - **Desarrollo**: Desplaza al jugador a la segunda planta, en la posición (x, y).

- Combate prefijado 1.
  - **Localización**: (21, 25)
  - **Condiciones**:
    * El combate no se ha superado todavía en esta sesión de exploración.
    * Inicio automático al colisionar con el jugador.
  - **Desarrollo**: El grupo es atacado por un grupo de enemigos pertenecientes a la tercera planta, y debe derrotarlos o huir para continuar. El número y tipo de enemigos son seleccionados aleatoriamente.

Segunda planta

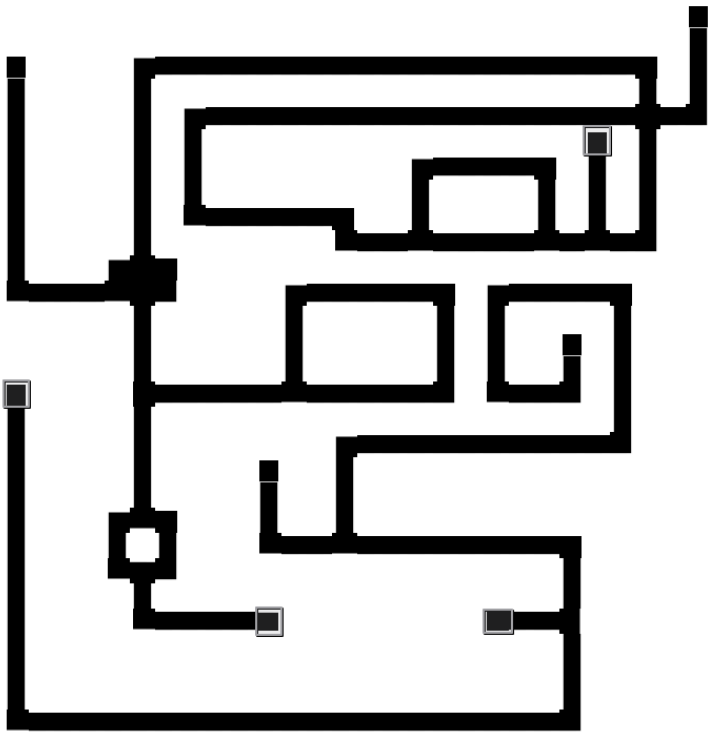

Figura 4.4: Mapa de la segunda planta del laberinto.

La segunda planta tiene como misión afianzar los conocimientos adquiridos en la primera, actuando como una extensión de la misma, y preparando al jugador para los retos que están por venir. El trazado es más directo, pero la mayor dificultad de los enemigos en esta planta supone un desgaste superior para el grupo.

- Escaleras planta 2, parte 1.
  - **Localización**: (26, 6)
  - **Condiciones**:
    - Inicio automático al colisionar con el jugador.
  - **Desarrollo**: Desplaza al jugador a la primera planta, en la posición (x, y).
- Escaleras planta 1, parte 2.
  - **Localización**: (13, 25)
  - **Condiciones**:
    - Inicio automático al colisionar con el jugador.
  - **Desarrollo**: Desplaza al jugador a la primera planta, en la posición (x, y).
- Escaleras planta 1, parte 3.
  - **Localización**: (22, 25)
  - **Condiciones**:
    - Inicio automático al colisionar con el jugador.
  - **Desarrollo**: Desplaza al jugador a la primera planta, en la posición (x, y).
- Escaleras planta 3.
  - **Localización**: (3, 16)
  - **Condiciones**:
    - Inicio automático al colisionar con el jugador.
  - **Desarrollo**: Desplaza al jugador a la tercera planta, en la posición (x, y).

Tercera planta

La tercera planta presenta una mayor complejidad en el laberinto, y encuentros aún más duros. Se trata de una planta corta, que acúa como preludio para la cuarta planta, y obliga al jugador a considerar si su equipo está preparado para afrontar el reto.

- Escaleras planta 2.
  - **Localización**: (3, 16)
  - **Condiciones**:
    - Inicio automático al colisionar con el jugador.
  - **Desarrollo**: Desplaza al jugador a la tercera planta, en la posición (x, y).
- Escaleras planta 4.
  - **Localización**: (25, 12)
  - **Condiciones**:
    - Inicio automático al colisionar con el jugador.
  - **Desarrollo**: Desplaza al jugador a la cuarta planta, en la posición (x, y).

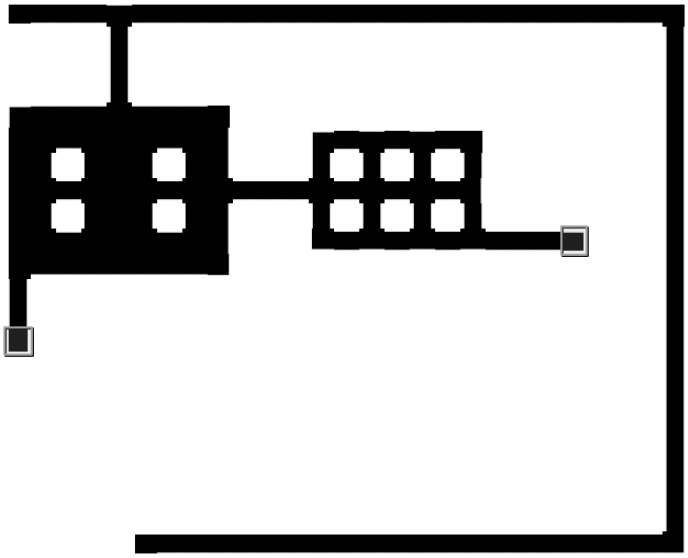

Figura 4.5: Mapa de la tercera planta del laberinto.

Cuarta planta

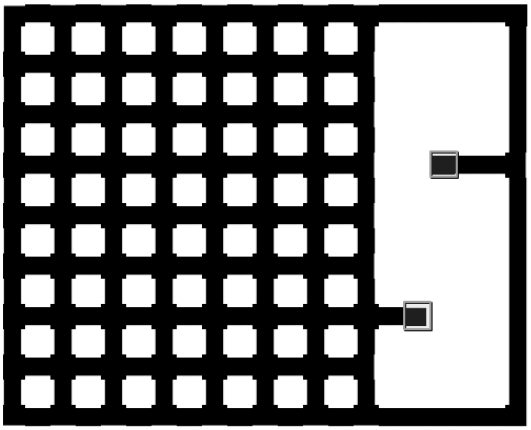

Figura 4.6: Mapa de la cuarta planta del laberinto.

La cuarta planta es la prueba de fuego. Potencialmente larga y enrevesada, el jugador necesita aplicar todos los conocimientos adquiridos en las plantas anteriores, así como gozar de un equipo debidamente

preparado, si quiere poder progresar. Los encuentros alcanzan aquí su pico de dificultad.

- Escaleras planta 3.
  - **Localización**: (25, 12)
  - **Condiciones**:
    - Inicio automático al colisionar con el jugador.
  - **Desarrollo**: Desplaza al jugador a la tercera planta, en la posición (x, y).
- Escaleras planta 5.
  - **Localización**: (24, 18)
  - **Condiciones**:
    - Inicio automático al colisionar con el jugador.
  - **Desarrollo**: Desplaza al jugador a la quinta planta, en la posición (x, y).

Quinta planta

Figura 4.7: Mapa de la quinta planta del laberinto.

La quinta planta acúa como un reto final. Idealmente, el jugador no encontrará un gran número de enemigos antes de enfrentarse al Jabberwock.

- Escaleras planta 4.
  - **Localización**: (24, 18)
  - **Condiciones**:
    - Inicio automático al colisionar con el jugador.

  - **Desarrollo**: Desplaza al jugador a la quinta planta, en la posición (x, y).

- Jabberwock.
  - **Localización**: (15, 15)
  - **Condiciones**:
    * Inicio automático al colisionar con el jugador.
  - **Desarrollo**: Comienza el combate final contra Jabberwock.

4.3.5.2 Enemigos

Pasemos ahora a hablar de los moradores del laberinto. Esta sección cubre únicamente los aspectos únicamente de diseño, concretamente: su aspecto visual, el concepto principal de la criatura, su esquema de comportamiento básico, y el nivel de dificultad estimado, así como su hábitat dentro del laberinto, y si se trata de un enemigo especial. Los detalles de implementación, tales como sus características específicas, o la descripción a alto nivel de su inteligencia artificial, se cubrirán más adelante.

Slime

Enemigo básico, con estadísticas básicas y la única capacidad de atacar al azar. No suponen una amenaza, incluso en grupos de tres, dada su fragilidad, escasa precisión y baja fuerza, y su única función es permitir al jugador acostumbrarse al sistema de combate mientras explora la primera planta. Las ganancias obtenidas por derrotar slimes son nimias, para compensar la prácticamente inexistente dificultad.

Healing slime

Slimes con la capacidad de curar a sus compañeros. Su capacidad de atención es extremadamente limitada, por lo que existe una probabilidad relativamente alta de que no se percaten de que sus compañeros o ellos mismos han sido heridos antes de que sea demasiado tarde. Moradores de la primera planta, sirven como una introducción al concepto de enemigo de apoyo. Son incluso más débiles que los slime normales, pero pueden suponer una molestia mayor debido a su capacidad de curar a otros enemigos. Las ganancias obtenidas por derrotarlos son también indignas de mención, dada su limitado nivel de peligro.

Goblin

Seres ruines y estúpidos, con un amor malsano por lo ajeno. Conocidos por asaltar y asesinar a aventureros novatos que bajan la guardia al llegar a la segunda planta del laberinto, los goblins pueden suponer un reto para aquellos que no esperan la agresividad de estos pequeños matones. Es común que trabajen para un líder, habitualmente un goblinoide más poderoso como un bugbear o un hobgoblin, quien controla al grupo mediante el miedo y la intimidación.

Hobgoblin

Goblins de gran tamaño y potente físico, es mejor no subestimar a estos guerreros implacables. Conocedores de su potente físico, pero más inteligentes que sus hermanos pequeños, los hobgoblin no dudan en recurrir a toda clase de artimañas, y en fortalecerse mediante el número. No se recomienda a los aventureros novatos enfrentarse a ellos: la recompensa no merece la pena en comparación con el elevado riesgo.

Bugbear

Los goblinoides de mayor tamaño presentes en el laberinto, peludos y malolientes. Pese a que no es extraño que un bugbear solitario tome el control de una tribu de goblins, un hobgoblin puede fácilmente establecer su control sobre grupos de estas estúpidas criaturas mediante el subterfugio. Naturalmente violentos y agresivos, los bugbears son oponentes dignos de mención. Â¡Cuidado! Pese a su vulnerabilidad a la magia, no dudarán en acabar con ese enclenque lanzador de conjuros que amenaza con destruirles.

Zombie

Cadáveres reanimados, los zombies se caracterizan por su lentitud y la potencia de sus golpes. Su condición de no-muertos los hace especialmente resistentes al daño de cualquier índole, incapaces de sentir el más mínimo dolor. Quizá no sean una gran amenaza por su cuenta, pero, en grupo, pueden suponer un problema incluso para los más avezados aventureros.

Skeleton

Esqueletos vivientes, producto de la nigromancia más perversa, los esqueletos son seres más inteligentes y ágiles que sus putrefactos hermanos. Con un odio visceral hacia la vida, no dudarán en cortarle el paso a cualquier aventurero en busca de riquezas. Subestimar a estas retorcidas criaturas es el peor error posible: sus brutales instintos les convierten en asesinos implacables.

Imp

Diablillos malevolentes, los imp son conocidos por destruír a los incautos que se atrevan a subestimarlos por su juvenial apariencia. Poderosos magos, no dudan en chamuscar a sus víctimas en un abrir y cerrar de ojos, mofándose a continuación de la estupidez cometida por los incautos aventureros.

Jabberwock

El jefe final de este prototipo, y un oponente digno de mención. Inteligente y poderoso, el Jabberwock es capaz de adaptar sus tácticas al comportamiento de sus adversarios, abrumando a aquellos que adopten una estrategia excesivamente defensiva u ofensiva por igual. Un monstruo resistente y brutal, solo se recomienda afrontar el combate a aquellos aventureros avezados, que, conscientes de sus habilidades, deseen alcanzar la fama... O perecer en el intento.

4.3.6 Habilidades

A continuación se detallarán las distintas habilidades presentes en el juego, incluyendo su descripción, así como aquellas fórmulas matemáticas que lo rijan.

Attack

Esta habilidad regula el ataque básico. No forma parte de ningún listado de habilidades visible para el jugador, pero se representa internamente como una habilidad más. Se encuentra gestionada mediantes dos fórmulas concretas: una para determinar si el ataque alcanza su objetivo, y otra para determinar el daño resultante en caso afirmativo. En primer lugar, se determina si el golpe acierta mediante la siguiente fórmula:

$$(a_0 + f_0) * r_0 > a_1 * r_1 \tag{4.1}$$

Donde a_0 y a_1 son la agilidad del atacante y del objetivo, respectivamente; f_0 es la fuerza del atacantes, y r_0 y r_1 son factores aleatorios generados en tiempo de ejecución. Esta fórmula permite equilibrar la balanza entre clases que tengan dependencia de su capacidad de ataque físico, al tiempo que establece un riesgo constante de fallar debido a un bajo valor aleatorio. Asimismo, crea la posibilidad de acertar a enemigos que, de otro modo, podrÃan resultar inalcanzables, si bien tal probabilidad es ínfima. Un ataque se considera exitoso cuando el primer término de la inecuación es mayor que el segundo término. En tal caso, se determina el daño del siguiente modo:

$$(a * r_0 - d) * r_1 \tag{4.2}$$

Donde a es el valor combinado de ataque del agresor, r_0 es un factor aleatorio entre 1 y 1'2, d es la defensa de la víctima y r_1 es un valor aleatorio entre 0'5 y 1. Esta fórmula es una simplificación de la fórmula de daño típica de múltiples RPG por turnos clásicos, utilizando únicamente los valores de ataque y defensa, así como dos factores aleatorios que generan variación en la potencia del impacto. Nótese que el primer valor aleatorio permite desplazar el centro de la campana de Gauss resultante de la fórmula y, por consiguiente, el rango de valores normales dentro de la misma, mientras que el segundo factor permite escoger el valor exacto dentro de dicho rango.

Defend

Esta habilidad regula la capacidad defensiva. No forma parte de ningún listado de habilidades visible para el jugador, pero se representa internamente como una habilidad más. Simplemente hace que la entidad entre en un estado defensivo, reduciendo el daño recibido.

Flee

Esta habilidad regula la opción de huir de un combate. No forma parte de ningún listado de habilidades visible para el jugador, pero se representa internamente como una habilidad más. Permite huir del combate en base a la siguiente fórmula:

$$r < a * 5 \tag{4.3}$$

Donde r es un valor entero aleatorio entre 0 y 100, y a es la agilidad de la entidad que está intentando huir del combate. El motivo por el que se multiplica la agilidad por 5 de cara a determinar el éxito de la acción es la necesidad de mejorar las probabilidades de escapar del usuario. Así, un personaje con un valor de agilidad de 4 tendrá un 20 % de probabilidades de huir, escasas, pero superiores al 4 % que implicaría, por ejemplo, el uso directo del valor de la característica.

Cure

Esta habilidad recupera una pequeña cantidad de salud al objetivo. La cantidad de salud regenerada se calcula en base a la siguiente fórmula:

$$\lfloor w * r * 1'5 \rfloor \tag{4.4}$$

Donde w es la sabiduría del usuario del hechizo, y r es un factor aleatorio entre 0 y 1. Efectivamente, el resultado es que la potencia de la curación fluctúa, desde su total inutilidad hasta un aumento del 50 % respecto al valor esperable dada la sabiduría del usuario, creando incertidumbre y, por consiguiente, un elemento de riesgo en el uso de esta habilidad.

Protect

Esta habilidad protege a un aliado del usuario, mitigando el daño recibido por el mismo este turno de forma equivalente a si se hubiese defendido por su cuenta. Permite a una entidad atacar desde el estado de defensa, a cambio de que un compañero de equipo renuncie a su turno.

Flash

Esta habilidad causa daño luminoso a un enemigo. Contra la mayor parte de enemigos, emplea las mismas fórmulas que **Attack**, pero contra demonios duplica el daño ocasionado en caso de acertar. De este modo, tanto la habilidad como sus usuarios ganan una cierta identidad propia, reforzando la ambientación del prototipo.

Smite

Esta habilidad causa daño sacro a un enemigo. Contra la mayor parte de enemigos, emplea las mismas fórmulas que **Attack**, pero contra no-muertos duplica el daño ocasionado en caso de acertar. De este modo, tanto la habilidad como sus usuarios ganan una cierta identidad propia, reforzando la ambientación del prototipo.

Cleave

Esta habilidad causa gran daño a un único enemigo. La fórmula para determinar el acierto es idéntica a la empleada por **Attack**, pero la fórmula de daño pasa a ser

$$(a * r_0 - d/3) * r_1 \tag{4.5}$$

Donde a es el valor combinado de ataque del agresor, r_0 es un factor aleatorio entre 1 y 1'2, d es la defensa de la víctima y r_1 es un valor aleatorio entre 0'5 y 1. Así, **Cleave** se trata de un ataque más potente que un ataque normal, apropiado para un guerrero, lo que afianza la identidad de la clase en un rol ofensivo dentro del equipo.

Zap

Causa daño mágico a un enemigo. Utiliza la misma fórmula para determinar si se produce un impacto que **Attack**, pero reemplazando fuerza por sabiduría. En cambio, la fórmula de daño pasa a ser

$$(w * r_0 - d) * r_1 \tag{4.6}$$

Donde w es el valor combinado de sabiduría del agresor, r_0 es un factor aleatorio entre 1 y 1'2, d es la defensa de la víctima y r_1 es un valor aleatorio entre 0'5 y 1. En esencia, **Zap** es una habilidad diseñada para suplir el escaso ataque físico de un mago, y dotarle de un lugar dentro de cualquier equipo.

Fireball

Causa daño mágico a un enemigo. Utiliza la misma fórmula para determinar si se produce un impacto que **Zap**. En cambio, la fórmula de daño pasa a ser

$$(w * r_0 - d/3) * r_1 \tag{4.7}$$

Donde w es el valor combinado de sabiduría del agresor, r_0 es un factor aleatorio entre 1 y 1'2, d es la defensa de la víctima y r_1 es un valor aleatorio entre 0'5 y 1. Se trata del equivalente mágico del grupo, y afianza la posición del mago como una clase ofensiva, comparable al propio guerrero una vez esta habilidad es adquirida.

Flare

Causa daño mágico a un enemigo. Tiene garantizado el acierto, y utiliza la siguiente fórmula de daño:

$$(w * r_0) * r_1 \tag{4.8}$$

Donde w es el valor combinado de sabiduría del agresor, r_0 es un factor aleatorio entre 1 y 1'2, d es la defensa de la víctima y r_1 es un valor aleatorio entre 1'2 y 2. La independencia del valor de defensa del enemigo convierte a esta habilidad en un arma de gran calibre, finalmente encumbrando a la clase **Mago** en términos ofensivos, para compensar sus carencias en otros aspectos importantes, y para justificar su presencia prolongada en un grupo de aventuras. Si tardía adquisición exige una recompensa comparable al esfuerzo realizado, y es por ello que la fórmula de daño escogida resulta tan beneficiosa al usuario.

Assassinate

Causa daño físico a un único enemigo. Comparte su fórmula de acierto con **Attack**, y su fórmula de daño es

$$(a * r_0 - d/3) * 3 \tag{4.9}$$

Donde a es el valor combinado de agilidad del agresor, r_0 es un factor aleatorio entre 1 y 1'2, y d es la defensa de la víctima. Esta habilidad sirve para afianzar la identidad de clase del pícaro, y dotarle de una herramienta ofensiva que le permita aportar a los encuentros que su grupo deba afrontar.

Resumen

Se adjunta, a continuación, una tabla resumen comparando todas las habilidades, la fórmula empleada para calcular su acierto y, en caso de ser necesario, la fórmula empleada para calcular el grado de efecto, esto es, los puntos de golpe ganados o perdidos, de la misma.

Nombre	Fórmula de impacto	Fórmula de efecto
Attack	$(agilidad_0 + fuerza_0) * r_0 > agilidad_1 * r_1$	$(ataque * r_0 - defensa) * r_1$
Defend	No aplicable	No aplicable
Flee	$r < agilidad * 5$	No aplicable
Cure	No aplicable	$\lfloor sabiduria * r * 1'5 \rfloor$
Protect	No aplicable	No aplicable
Flash	$(agilidad_0 + fuerza_0) * r_0 > agilidad_1 * r_1$	$(ataque * r_0 - defensa) * r_1$
Smite	$(agilidad_0 + fuerza_0) * r_0 > agilidad_1 * r_1$	$(ataque * r_0 - defensa) * r_1$
Cleave	$(agilidad_0 + fuerza_0) * r_0 > agilidad_1 * r_1$	$(ataque * r_0 - defensa/3) * r_1$
Zap	$(agilidad_0 + sabiduria_0) * r_0 > agilidad_1 * r_1$	$(sabiduria * r_0 - defensa) * r_1$
Fireball	$(agilidad_0 + sabiduria_0) * r_0 > agilidad_1 * r_1$	$(sabiduria * r_0 - defensa/3) * r_1$
Flare	No aplicable	$(sabiduria * r_0) * r_1$
Assassinate	$(agilidad_0 + fuerza_0) * r_0 > agilidad_1 * r_1$	$(agilidad * r_0 - defensa/3) * 3$

Tabla 4.1: Tabla comparando todas las habilidades diseñadas para el prototipo.

Nótese que todos los elementos denominados r son valores aleatorios generados en tiempo de ejecución, para ese uso concreto de la correspondiente habilidad.

4.3.7 Objetos

A continuación se detallan los distintos objetos presentes en el juego, especificando si se trata de objetos consumibles o clave, así como su efecto.

Vulnerary

Objeto básico de aventura, barato y útil. Puede comprarse sin demasiadas dificultades.

- **Tipo**: Consumible.
- **Efecto**: Recupera 30 HP al objetivo.

Health Potion

Objeto curativo potente. Puede encontrarse en determinados puntos del laberinto, pero es un bien escaso.

- **Tipo**: Consumible.
- **Efecto**: Recupera 100 HP al objetivo.

Resumen

A continuación, se presenta una tabla resumen comparando los objetos presentes en el prototipo:

Nombre	Tipo	Efecto
Vulnerary	Consumible	Recupera 30 HP al objetivo
Health Potion	Consumible	Recupera 100 HP al objetivo

Tabla 4.2: Tabla comparando los distintos objetos diseñados para el prototipo.

4.3.8 Pantallas

Esta sección cubre las consideraciones tomadas en el diseño de las distintas pantallas del juego, así como las múltiples versiones de las mismas por las que se ha pasado hasta llegar al diseño final.

4.3.8.1 Consideraciones de accesibilidad

Para garantizar en la medida de lo posible la accesibilidad de las pantallas generadas, se han tenido en cuenta tres pilares fundamentales:

- El nivel de contraste ha de ser el mayor posible, con el objetivo de facilitar la lectura y comprensión a personas con visión reducida que no utilicen un lector de pantalla. Por este mismo motivo, la mayor parte de elementos del juego se representarán usando blanco y negro.[5]
- La información imprescindible para el disfrute del videojuego ha de ser accesible mediante el uso de un lector de pantalla. De este modo, esta infomación habrá de transmitirse, en la mayor parte de casos, de manera textual. Asimismo, dicha información visual deberá encontrarse disponibles mediante múltiples canales de distinta índole.
- Los controles deben ser sencillos, y permitir rotar desde el principio hasta el final de manera ininterrumpida en los menús. Por desgracia, esto no es posible en otras circunstancias, por lo que habrán de integrarse mecanismos auxiliares que faciliten el disfrute por parte de usuarios con un elevado grado de ceguera. Estos controles, sencillos e intuitivos, benefician también a otros perfiles de diversidad funcional, tales como el cognitivo, en el que permiten limitar la frustración que experimenta el jugador debido a la dificultad para interactuar con el juego.

4.3.8.2 Menú principal

El menú principal contiene únicamente tres opciones:

- **New game**: Comienza una nueva partida desde el principio, advirtiendo al jugador de que el juego contiene un único archivo de guardado, lo que podría suponer perder el progreso realizado en una partida previa en caso de guardar la partida.

- **Load**: Carga la partida actualmente guardada, o informa al jugador de que no existe en caso de ser necesario.
- **Exit**: Cierra el juego y regresa al sistema operativo.

En circunstancias ideales, este menú daría acceso también a un menú de configuración desde el que personalizar los controles y diversos elementos de la interfaz, pero, dado la carga técnica presente con los requisitos actuales, se ha prescindido de ella para esta versión del prototipo, quedando como trabajo futuro.

Figura 4.8: Diseño esquemático del menú principal.

4.3.8.3 Menú del pueblo

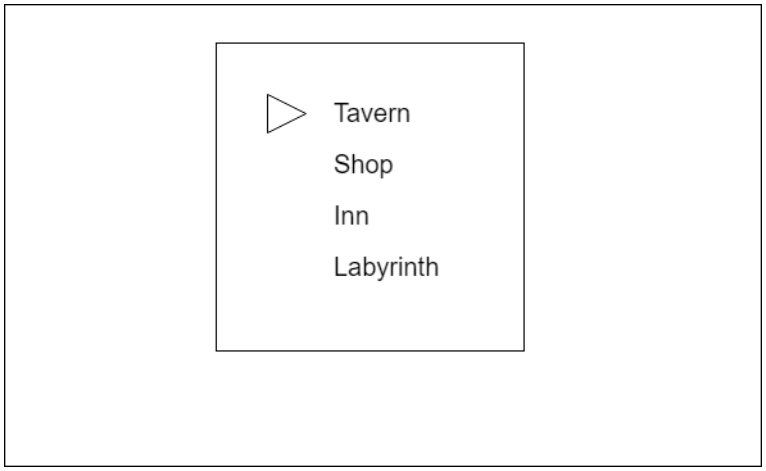

Figura 4.9: Diseño esquemático del menú del pueblo.

El menú del pueblo permite gestionar el grupo de aventuras, así como acceder al laberinto. Para que esta última opción se habilite, el grupo debe contener al menos un aventurero. Este menú contiene las siguientes opciones:

- **Tavern**: Permite acceder a la taberna, para gestionar el grupo.
- **Shop**: Permite acceder a la tienda, para comprar consumibles útiles en caso de poseer fondos suficientes.

- **Inn**: Permite acceder a la posada, donde se puede guardar la partida.
- **Labyrinth**: Comienza la exploración desde la entrada de la primera planta del laberinto.

4.3.8.4 Menú de la taberna

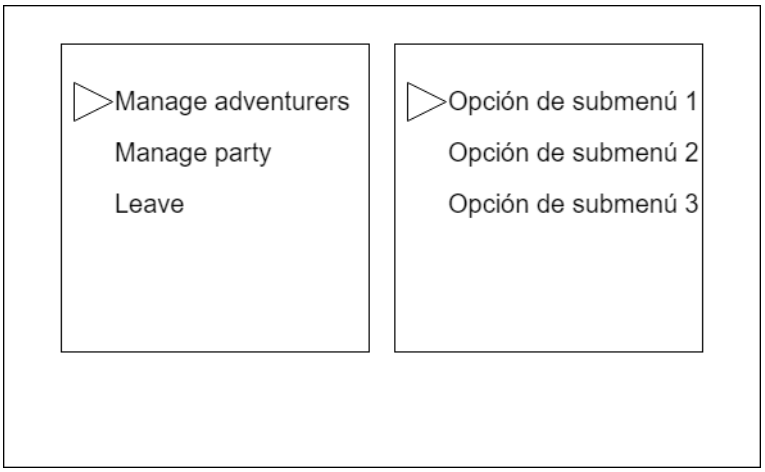

Figura 4.10: Diseño esquemático del menú de la taberna.

El menú de la taberna permite acceder a todas las opciones de gestión del grupo de aventuras. En caso de ser necesario, se detallan a también las opciones de los submenús correspondientes.

- **Manage adventurers**: Da acceso al submenú de gestión de aventureros, el cual contiene las siguientes opciones:
  - **Register**: Accede a la creación de personaje, y permite generar un nuevo aventurero.
  - **Dismiss**: Permite eliminar personajes ya creados, solicitando previamente la confirmación por parte del usuario para minimizar el riesgo de eliminaciones accidentales.
- **Manage party**: Da acceso al submenú de gestión del grupo de aventuras, el cual contiene las siguientes opciones:
  - **Add to party**: Permite añadir aventureros al grupo. En caso de que el grupo esté lleno, informa al jugador.
  - **Replace adventurer**: Permite reemplazar un aventurero en el grupo con otro en reserva. En caso de que el grupo no contenga ningún aventurero, informa al jugador.
  - **Remove from party**: Permite retirar aventureros del grupo. En caso de que el grupo no contenga ningún aventurero, informa al jugador.
- **Leave**: Regresa al menú del pueblo.

4.3.8.5 Menú de la tienda

El menú de la tienda es, en realidad, bastante sencillo. Permite acceder a las opciones de compra, compuestas únicamente por un listado de objetos disponibles así como sus precios. Contiene las siguientes opciones:

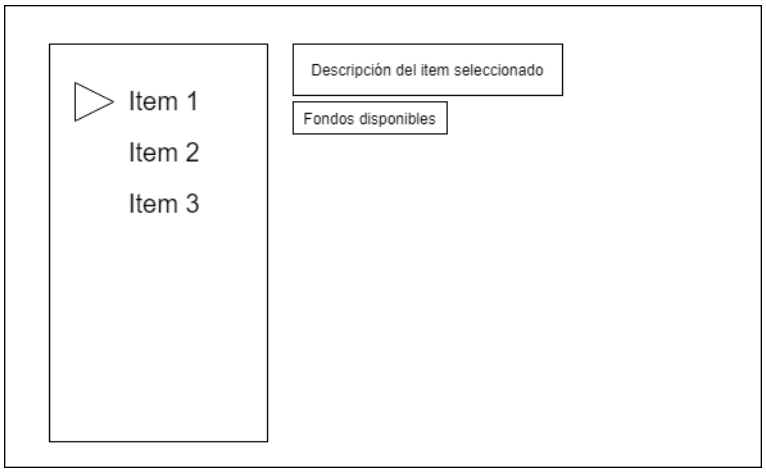

Figura 4.11: Diseño esquemático del menú de compra en la tienda.

- Buy: Permite acceder al menú de compra propiamente dicho. Al confirmar un elemento en dicho menú, se solicita la confirmación del jugador y, en caso afirmativo, se pasa a la compra. En caso de que el inventario de todos los integrantes del grupo esté lleno, o que no se disponga de los fondos suficientes para adquirir el objeto, se informa al usuario y la compra se cancela.
- **Leave**: Regresa al menú del pueblo.

En un caso ideal, se incluiría también una opción de venta de objetos, pero, dado que este prototipo ha de ser relativamente sencillo, se ha prescindido de ella para esta versión, quedando como trabajo futuro.

4.3.8.6 Menú de la posada

Figura 4.12: Diseño esquemático del menú de la posada.

El menú de la posada permite, únicamente, acceder al sistema de guardado. El menú resultante contiene las siguientes secciones:

- **Save**: Guarda la partida, previa confirmación del usuario. En caso de haber datos anteriores, se sobrescriben.
- **Leave**: Regresa al menú del pueblo.

4.3.8.7 Pantalla de exploración

La pantalla de exploración es, en realidad, la pantalla más imporante del juego. En ella se presentan numerosos elementos de ayuda para la navegación del laberinto, muchos de los cuales cuentan, además, con ayudas sonoras mediante el uso de lectores de pantallas o efectos de sonido auxiliares.

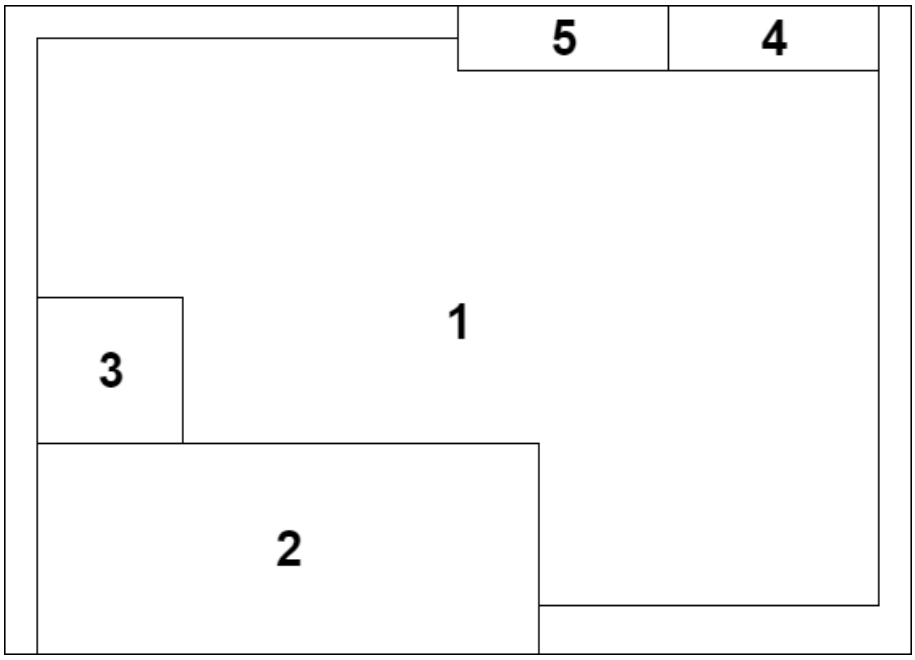

Figura 4.13: Diseño esquemático de la interfaz de exploración.

1. Pantalla principal de exploración. En ella se representa lo que el grupo de aventureros ve, en primera persona, permitiendo al jugador tener un conocimiento aproximado de su entorno.
2. Display de estado del grupo. En este recuadro se presenta información auxiliar sobre el estado del grupo, tal como los nombres de los personajes, su HP y su MP. No cuenta con audiodescripción directa, pero es accesible mediante el menú de pausa.
3. Minimapa. Una pequeña respresentación auxiliar de un area de 3x3 casillas alrededor del grupo. No es accesible mediante audiodescripción, pero tampoco es imprescindible para la navegación del laberinto.
4. Indicador de dirección. Indica hacia donde está mirando el grupo ahora mismo para facilitar la navegación. Cada vez que el grupo se gira, el lector de pantalla, en caso de estar activo, informa al jugador del cambio realizado.
5. Indicador de posición. Representa las coordenadas auxiliares del grupo para facilitar la elaboración de mapas por parte del jugador en caso de considerarse necesario. No es accesible mediante audiodescripción, pero puede ser ignorado sin perjuicios notables. Al igual que el minimapa, pretende ser únicamente una pequeña ayuda auxiliar a la navegación.

Asimismo, en esta pantalla, las entradas generadas por el jugador no afectan a un menú, sino que permiten al grupo desplazarse por el laberinto, o interactuar con él. Las opciones disponibles son las siguientes:

- Avanzar. Simplemente mueve al grupo una casilla en la dirección en la que está mirando, en caso de no haber un muro en el camino.
- Retroceder. Simplemente mueve al grupo una casilla en la dirección contraria a la que está mirando, en caso de no haber un muro en el camino.
- Girar a la derecha. Cambia la orientación del grupo en sentido horario.
- Girar a la izquierda. Cambia la orientación del grupo en sentido antihorario.
- Interactuar con el laberinto. Activa el evento presente en la actual casilla, en caso de haberlo, siempre y cuando se cumplan las condiciones necesarias para su activación.
- Abrir el menú de pausa.

4.3.8.8 Menú de pausa

El menú de pausa permite acceder a múltiples opciones útiles durante la exploración, y se vio rediseñado respecto a la primera aproximación para simplificar las opciones disponibles.

Inicialmente, el menú contendría las siguientes opciones:

- **Status**: Permite comprobar el estado de los personajes, y dota al lector de pantalla de acceso a la información simplificada presente también en la pantalla de exploración.
- **Items**: Permite acceder al inventario de los personaje.
- **Equipment**: Permite gestionar el equipo de los personajes.
- **Magic**: Permite acceder a las habilidades de los personajes, y usar aquellas utilizables fuera de combate.
- **Quests**: Permite acceder al menú de misiones.
- **Formation**: Permite reorganizar la formación del grupo.

Con el objetivo de simplificar el prototipo resultante, este menú fue simplificado en gran medida, obteniendo así una segunda versión, que es la que finalmente se implementó.

- **Status**: Se comporta del mismo modo que la versión inicial.
- **Items**: Se comporta del mismo modo que la versión inicial.
- **Skills**: Se comporta del mismo modo que **Magic** en la versión inicial.

El resultado es un menú más compacto y de más ágil manejo, y una menor complejidad de implementación. Dado que se optó por no implementar un sistema de equipo, misiones o formación en el prototipo actual, todos ellos quedan como trabajo futuro para el desarrollo del mismo.

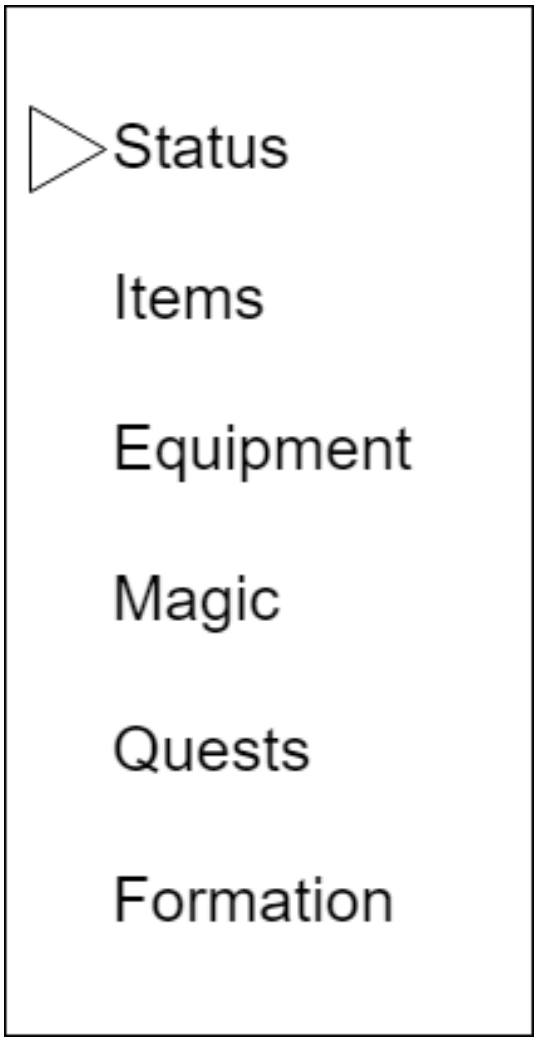

Figura 4.14: Diseño esquemático original del menú de pausa.

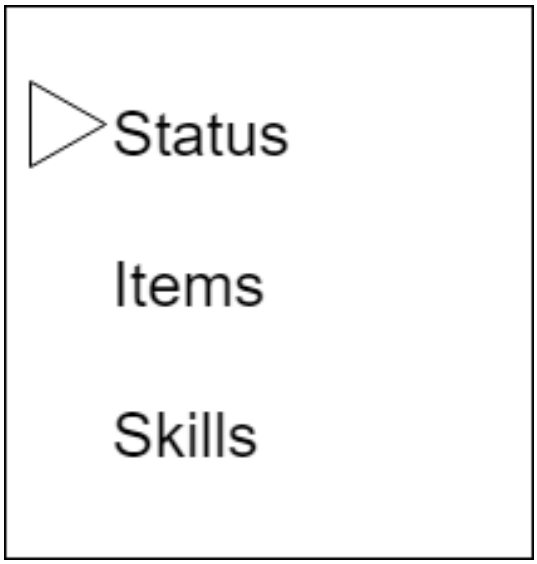

Figura 4.15: Diseño esquemático final del menú de pausa.

4.3.8.9 Menú de inventario

El menú de inventario dota al jugador de acceso a los objetos consumibles de cada miembro del grupo.

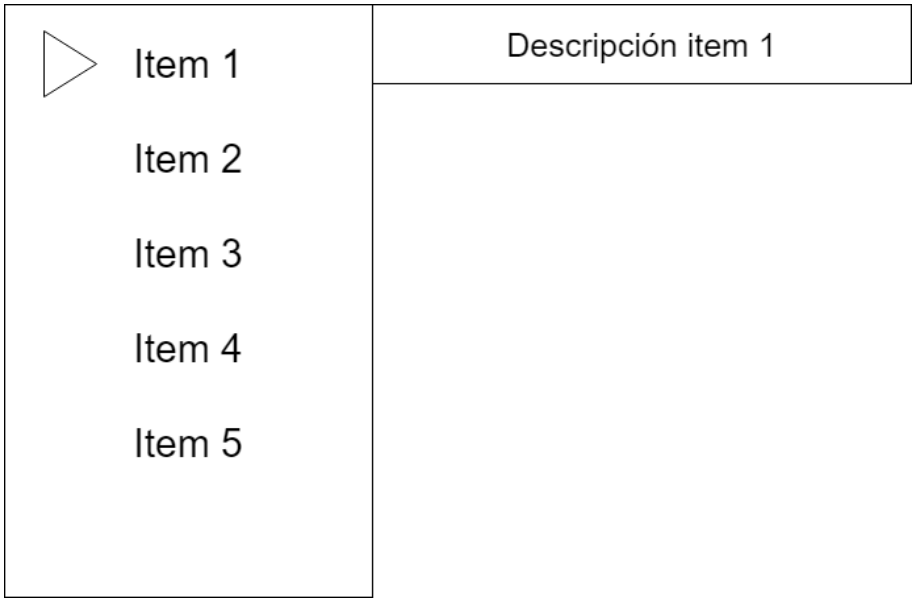

Figura 4.16: Diseño esquemático del menú de inventario.

Los mismos controles empleados para la exploración se utilizan para navegar este menú. **Avanzar** y **Retroceder** permiten desplazar el cursor arriba y abajo respectivamente, mientras que **Girar izquierda** y **Girar derecha** permiten pasar al inventario del siguiente miembro del grupo o el anterior. **Abrir el menú de pausa** permite utilizar el objeto actualmente seleccionado, y **Interactuar con el laberinto** permite salir del menú de inventario.

4.3.8.10 Menú de habilidades

El menú de habilidades dota al jugador de acceso a las habilidades utilizables de cada miembro del grupo. Los controles son idénticos a los del menú de inventario.

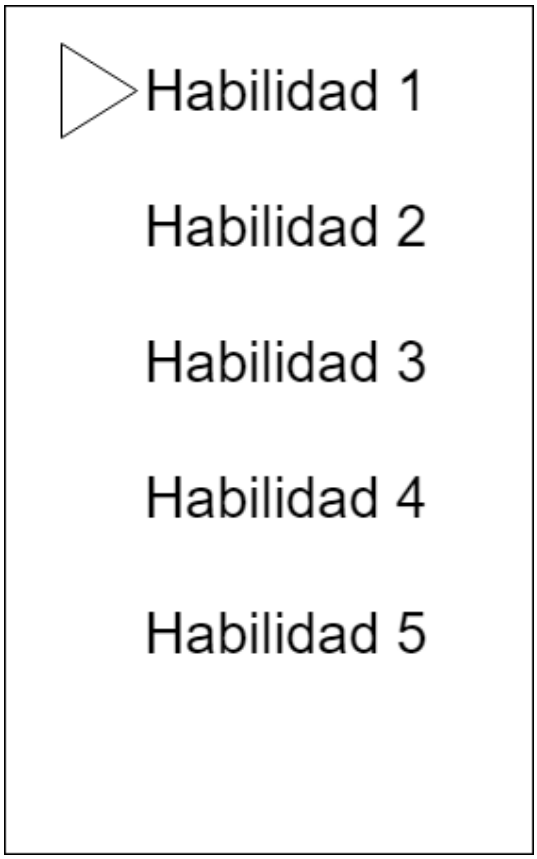

Figura 4.17: Diseño esquemático del menú de habilidades.

4.3.8.11 Pantalla de estado

La pantalla de estado permite confirmar las estadísticas y estado actuales de un personaje determinado. La interacción es extremadamente limitada, permitiendo únicamente desplazar el cursor por las distintas opciones para que el lector de pantalla pueda leerlas, o salir de la pantalla de estado. En un primer momento, se pensó en implementar un sistema similar al de los menús de inventario y habilidades para cambiar ágilmente entre habilidades, pero se descartó para facilitar la navegación con audiodescripción.

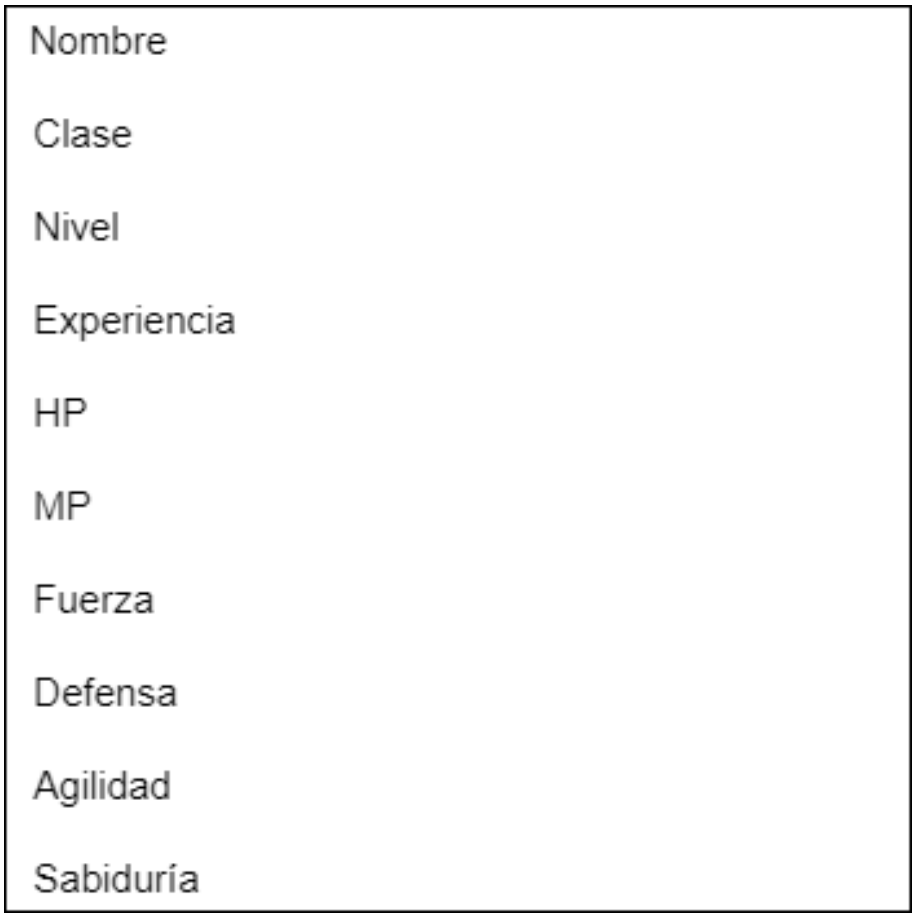

Figura 4.18: Diseño esquemático de la pantalla de estado.

4.3.8.12 Pantalla de combate

Si la pantalla de exploración del laberinto es la pantalla más importante del juego, la pantalla de combate es la segunda pantalla más importante. El jugador pasará una gran parte del tiempo enfrentándose a enemigos, y será en este proceso en el que interactuará con la mayor parte de mecánicas. La pantalla de combate cuenta con dos subpantallas fundamentales, pero ambas comparten determinados elementos de interfaz.

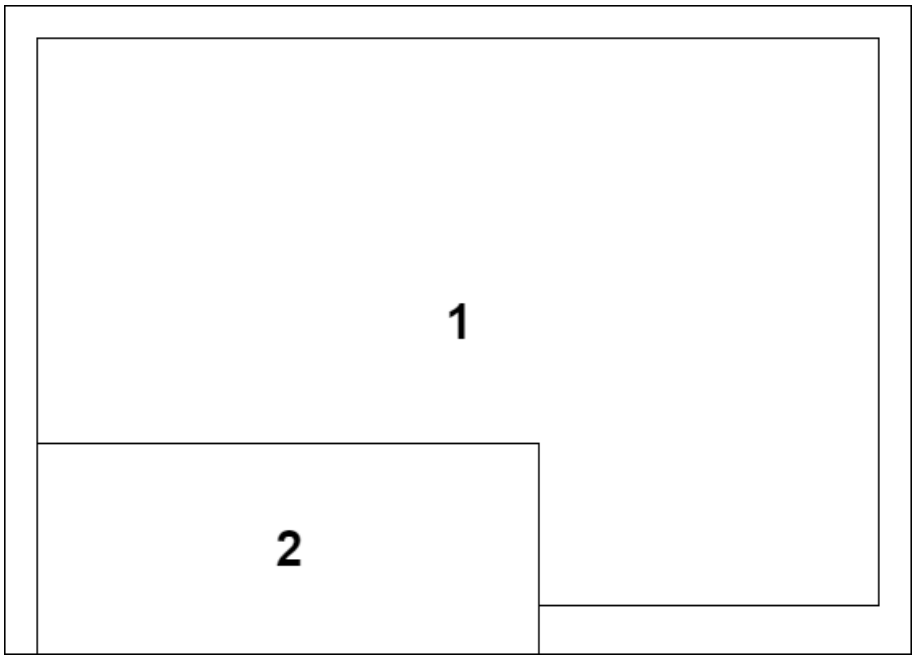

Figura 4.19: Diseño esquemático de la interfaz de combate.

1. Sección de representación del laberinto modificada. Los enemigos se representan superpuestos al laberinto, con un máximo de tres enemigos en combate simultáneamente.
2. Sección de estado del grupo. Idéntica a la presente en la exploración del laberinto.

La presentación del sistema de combate se vertebra en base a estos dos elementos. Cabe destacar que todos los enemigos han de tener un nombre textual asociado para su acceso mediante lector de pantalla.

Menú de selección de acciones

La primera subpantalla fundamental para el funcionamiento del sistema de combate es el menú de selección de acciones. Desde ella, se pueden elegir las acciones de cada miembro del grupo, así como los objetivos de las mismas, en caso de ser necesario.

Figura 4.20: Diseño esquemático del menú de selección de acciones en combate.

El funcionamiento de las distintas opciones es el siguiente:

- **Attack**: Ejecuta la habilidad **Attack** sobre el enemigo seleccionado durante la ejecución del turno del personaje.
- **Defend**: Ejecuta la habilidad **Defend** sobre el propio personaje al comenzar el turno.
- **Skills**: Previamente llamado **Magic**, permite seleccionar una habilidad de la lista de habilidades aprendidas por el personaje, para su uso sobre una entidad aplicables. Cabe destacar que el jugador no puede seleccionar habilidades para las que no se disponga de suficientes MP.
- **Item**: Permite usar un objeto consumible durante un turno, de forma similar a como se usaría desde el menú de inventario.
- **Run**: Permite al jugador intentar escapar, usando la habilidad **Flee** sobre el propio usuario.

Una vez todos los miembros vivos del grupo han seleccionado una acción, se pasa a la pantalla de ejecución de turno.

Pantalla de ejecución de turno

En esta pantalla, las distintas entidades ejecutan, de manera ordenada, las acciones seleccionadas. El orden de está ejecución depende del valor de agilidad de cada entidad, así como de un factor de modificación oculto asociado a la acción a realizar. Una vez calculado el efecto de una acción, se presenta un mensaje en pantalla informando al jugador del resultado, antes de pasar a resolver la siguiente acción. Nótese que los enemigos han seleccionado sus acciones en el momento de entrar a esta pantalla, y no pueden modificarlas más adelante, con independencia de la validez de las mismas en el momento de tratar de ejecutarlas, por lo que tanto el jugador como la inteligencia artificial son susceptibles de cometer errores de cálculo.

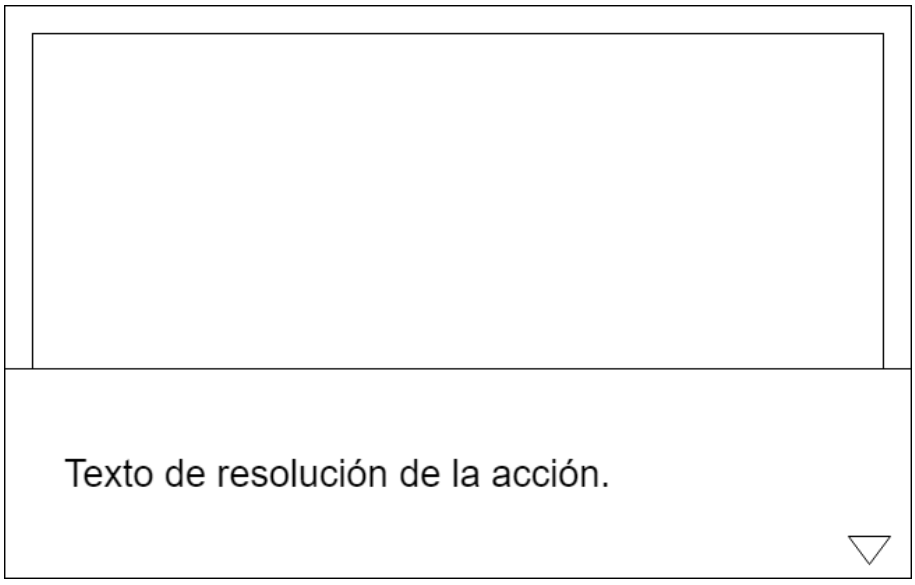

Figura 4.21: Diseño esquemático de la interfaz de ejecución del turno.

4.4 Implementación del videojuego

A continuación, se detallará la implementación realizada respecto al prototipo, indicando, de ser necesario, las modificaciones aplicadas al diseño inicial para mejorar el funcionamiento del conjunto. En primer lugar, se explica la aproximación arquitectónica software realizada, tanto desde un punto de vista de abstracción como desde el punto de vista de la estructura modular del prototipo. A continuación, se detallan las estructuras de datos que vertebran el conjunto del prototipo, explicando, asimismo, el razonamiento que ha llevado a su aplicación, así como potenciales alternativas en caso de considerarse oportuno. Finalmente, se procede a explicar la implementación específica de los distintos elementos que componen el prototipo.

4.4.1 Aproximación arquitectónica

La arquitectura del prototipo se construye en base a dos ideas fundamentales: el uso de múltiples niveles de abstracción que cooperan entre sí, y la integración de diversos módulos que implementan esta idea y se comunican para obtener las funcionalidades deseadas. El patrón de diseño arquitectónico básico empleado es Modelo-Vista-Controlador, por lo que, en primer lugar, se explican los conceptos fundamentales referentes al mismo, y esta sección culmina con su aplicación al prototipo generado.

4.4.1.1 Patrón Modelo-Vista-Controlador

El patrón de diseño Modelo-Vista-Controlador se basa en la subdivisión de los componentes de una aplicación en tres grupos fundamentales: un modelo, que establece cómo han de estructurarse los datos que la aplicación manejará para su almacenamiento y proceso; una vista, que regula cómo se presentará la información al usuario, así como las interacciones del mismo con la aplicación, y un controlador, que se encargará de procesar datosy modificarlos, y de tratar las interacciones del usuario con la vista.

Así pues, el nivel de acoplación entre los distintos componentes se minimiza, existiendo preeminentemente entre los mecanismos que el controlador tiene para coordinar su funcionamiento con el modelo y la vista, y la capa con la que se trata de coordinar, pero se obtiene un elevado de grado de cohesión. No es de extrañar, por tanto, que sea un patrón ampliamente aplicado en la industria del videojuego: la necesidad de gestionar datos y presentarlos de forma eficiente es una de las grandes problemáticas del sector desde el punto de vista técnico, especialmente en la actualidad, con la tendencia existente a generar parches que corrijan errores en el producto final para contentar al usuario.

Precisamente la agilidad de la que dota al proceso de desarrollo una vez implementada la infraestructura requerida por el patrón es el motivo por el que encaja perfectamente con este prototipo: la posibilidad de generar contenidos de manera ágil sin necesitar alterar grandes partes de la lógica de negocio básica del juego, o de la presentación del mismo, supone una gran baza de cara a minimizar costes y tiempos de desarrollo.

4.4.1.2 División en niveles de abstracción

Para facilitar la implementación del modelo MVC, así como delimitar de forma clara las responsabilidades de cada componente del sistema se decidió crear una estructura lógica en forma de pila con múltiples niveles de abstracción. Los niveles inferiores proveerían funcionalidad a los superiores, creando una cadena hasta llegar a la interacción con el usuario. Así pues, podemos distinguir las siguientes capas:

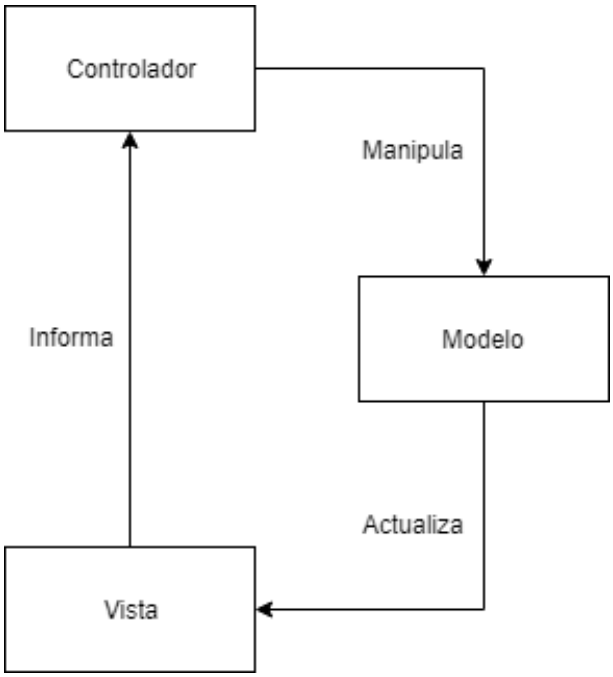

Figura 4.22: Representación del patrón Modelo-Vista-Controlador.

- Una capa de funcionalidad básica, que se encarga de las operaciones de bajo nivel necesarias para la gestión de entrada y salida, así como otras funcionalidades auxiliares que se detallan un poco más adelante.
- Una capa de motor de juego, que implementa las mecánicas propias del género Dungeon Crawler en los RPG.
- Una capa de datos, en la que se implementan los contenidos del juego propiamente dichos mediante abstracciones de alto nivel.

Cabe destacar que estas capas se subdividen, a su vez, en componentes, que actúan de forma transparente al resto de capas.

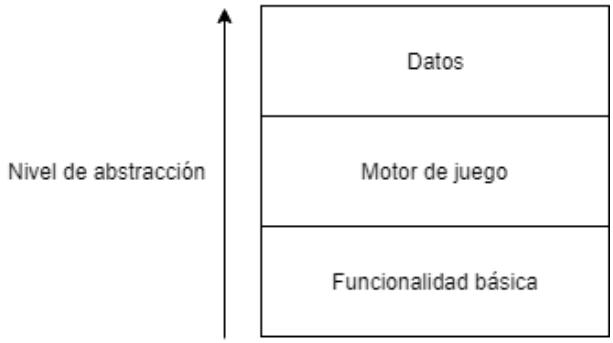

Figura 4.23: Representación de los distintos niveles de abstracción como pila.

Explicación de los niveles de abstracción

Procedemos, por tanto, a detallar qué componentes conforman cada nivel de abstracción, así como la funcionalidad que ofrece cada uno. Esta estructura establece, en gran medida, la colaboración entre los

distintos componentes del sistema desde un punto de vista modular, por lo que su estructura es definitoria para el resto del proyecto elaborado.

En la capa de funcionalidad básica encontramos dos componentes: el motor LibGDX y la librería Tolk. LibGDX se encarga de proveer todas las funcionalidades referentes al manejo de un videojuego propiamente dicho: interacción con el hardware de entrada y salida, manejo de archivos, gestión de datos a bajo nivel, y ejecución del bucle de juego básico, asegurándose de llamar a las funciones de inicialización, limpieza y eliminación de los componentes del juego de acuerdo a las necesidades del momento actual. Tolk, por contra, ofrece una interfaz con múltiples lectores de pantalla; su funcionamiento, uso y razones para integrarla en el proyecto se explicarán más adelante, en la sección sobre la implementación de accesibilidad.

En la capa de motor de juego, encontramos únicamente el propio motor construido sobre LibGDX. Dicho motor implementa la máquina de estados básica, así como métodos para gestionarla mediante un sistema de mensajes internos que se explicará más adelante. Implementa, también, la arquitectura básica de los sistemas de entidades, inventario, objetos, habilidades, enemigos, exploración, combate, creación de personajes, gestión de grupo y guardado de partida, estableciendo la aproximación necesaria para el manejo de los datos específicos referentes a estos sistemas.

En la capa de datos, encontramos, por un lado, una capa de scripting, y por otro, los archivos que representan la información propiamente dicha, como podrán ser imágenes, archivos de audio y mapas en formato TMX. La función de la capa de scripting es sencilla: permitir le generación rápida de elementos dinámicos dentro del juego, tales como eventos, habilidades, enemigos u objetos, para facilitar la modificación durantes las pruebas, y el ajuste de parámetros concretos que impacten a la funcionalidad de los mismos. Por contra, el resto de archivos se utilizan para representar información de carácter puramente estático: recursos gráficos o sonoros, así como estructuras de escenarios y, potencialmente, diálogos en múltiples idiomas. Cabe destacar que esta última idea fue descartada dado el coste adicional de implementarla en este prototipo, pero se considera un pilar fundamental de cara a desarrollar trabajo futuro.

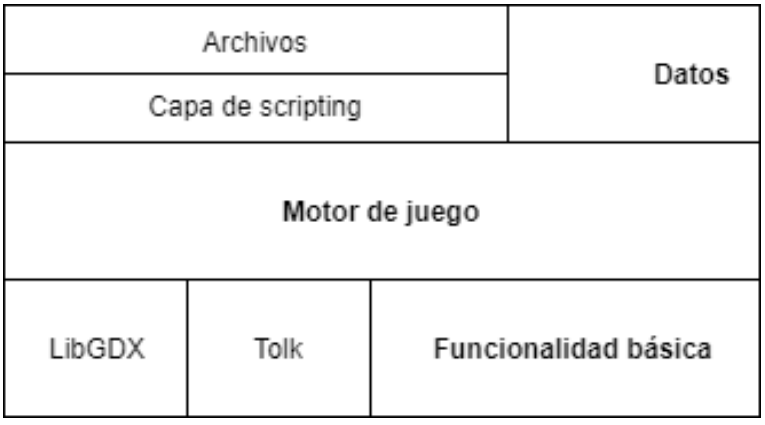

Figura 4.24: Representación de los distintos componentes de los niveles de abstracción.

Colaboración entre niveles de abstracción

Habiendo establecido la responsabilidad de los diversos niveles de abstracción, no resulta difícil imaginar cómo se integran. Procedemos, por tanto, a explicar la cooperación entre los mismos, dado su impacto fundamental en el resto del proyecto.

La capa de motor se encarga de centralizar el grueso de la funcionalidad del prototipo, solicitando información específica a la capa de datos cuando se vuelve necesaria, procesándola y solicitando los servicios pertinentes a la capa de funcionalidad básica en cada momento. Así, si el jugador está, por ejemplo, explorando el laberinto, la capa de motor procesará los movimientos realizados, solicitará a la capa de datos información sobre la nueva posición del grupo, así como el entorno inmediato del mismo, la procesará, y solicitará a la capa de funcionalidad básica la representación de la imagen resultante, así como la reproducción de sonidos que se consideren pertinentes. Esta interacción se extiende a absolutamente todos los ámbitos de la ejecución del videojuego, por lo que podemos, por fin, establecer el papel que cada capa juega en el patrón modelo vista controlador:

- La capa de funcionalidad básica se comporta, de hecho, como la vista, gestionando, únicamente, las entradas y salidas de acuerdo a las demandas del motor.
- La capa de motor de juego se comporta como el controlador, centralizando el grueso del proceso y coordinando a las otras dos capas.
- La capa de datos hace el papel del modelo, como cabía de esperar, estructurando la información empleada por el motor para su funcionamiento.

La integración de todas estas ideas es, en realidad, natural y transparente, gracias a la conceptualización de LibGDX como un framework de creación de videojuegos más que como un motor completo, lo que permite un altísimo grado de personalización, a costa de requerir un elevado esfuerzo para la generación de la lógica de negocio básica en cualquier proyecto de una mínima envergadura.

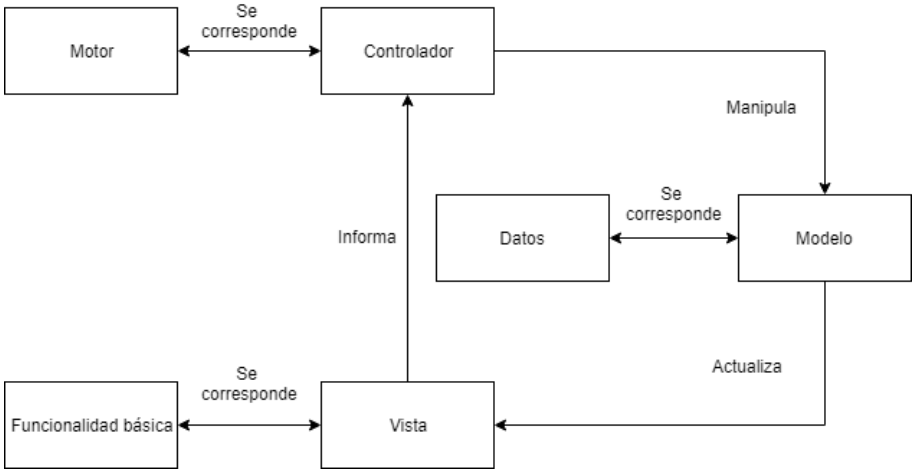

Figura 4.25: Correspondencia entre los distintos niveles de abstracción y MVC.

4.4.1.3 Estructura modular

Una vez explicada la estructura del prototipo desde un punto de vista de la abstracción, podemos estudiar su estructura en lo referente a módulos. En este caso, podemos distinguir cuatro grandes módulos, cada uno con múltiples componentes, que intractúan de manera constante:

- LibGDX.
- Un módulo de accesibilidad, en este caso, Tolk.
- El motor de juego.
- El módulo de scripting.

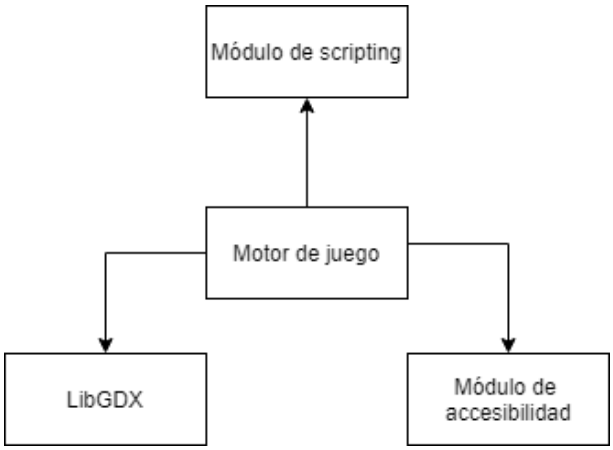

Figura 4.26: Esquema representando los cuatro grandes módulos que componen el funcionamiento del prototipo.

Explicación de los módulos

Para poder explicar apropiadamente la integración de esos cuatro módulos, es necesario, primero, establecer sus responsabilidades. En algunos casos, existe correspondencia directa con su papel en la estructura lógica, pero se retirará dicha información con el objetivo de ofrecer una visión lo más completa posible.

LibGDX provee la funcionalidad de entrada y salida básica, así como algunas herramientas auxiliares para el manejo interno de archivos. Su función en el conjunto es idéntica a la que tiene desde el punto de vista de la abstracción: servir como base para contruir el resto de elementos que permiten al prototipo funcionar.

El módulo de accesibilidad, una vez más, provee interacción con las disintas tecnologías de asistencia ofrecidas por el sistema operativo sobre el que se está ejecutando el prototipo. En el caso concreto de este proyecto, esta capa está compuesta únicamente por la librería Tolk, pero, en versiones futuras, se pretende incluir múltiples sistemas de índole similar para dar cabida a mayor número de plataformas y lectores de pantalla de destino.

El motor de juego mantiene el mismo papel que se le atribuía desde el punto de vista de la abstracción: integrar y coordinar al resto de elementos, tomando un papel central en el funcionamiento del conjunto.

El módulo de scripting no aparece de forma directa en la estructura que refleja la arquitectura abstracta del prototipo, pero su función es fácilmente discernible: permitir al motor interactuar con aquellos elementos de la capa de datos de carácter dinámico, esto es, scripts, haciendo de intermediario entre los lenguajes de programación involucrados, traduciendo y adaptando las estructuras de datos pertinentes, y permitiendo un flujo transparente de información entre las partes involucradas en el proceso.

Colaboración entre los módulos

Una vez más, la colaboración entre módulos fluye de manera natural una vez establecidas sus responsabilidades. No obstante, es adecuado revisarla para establecer los conceptos más fundamentales del flujo de información resultante de manera clara y concisa.

El motor de juego centraliza toda la lógica de negocio básica del prototipo, recurriendo a LibGDX y el módulo de accesibilidad para gestionar la presentación de información al usuario, y a LibGDX exclusivamente para interactuar con las entradas generadas por el mismo, así como facilitar el manejo interno de memoria en cuestiones de carga y almacenamiento de archivos. En cuanto a su interacción con

el módulo de scripting, esta se produce cuando un componente solicita una funcionalidad de alto nivel, indicando como se explicará más adelante qué archivo contiene la función a ejecutar, de modo que se produce de manera habitual.

Como cabe de esperar, no existe interacción directa entre LibGDX y el módulo de accesibilidad, LibGDX y la el módulo de scripting, o el módulo de accesibilidad y el módulo de scripting, dado que todos ellos se limitan a proveer servicios al motor de juego, que es el encargado de gestionar todas las funcionalidades resultantes.

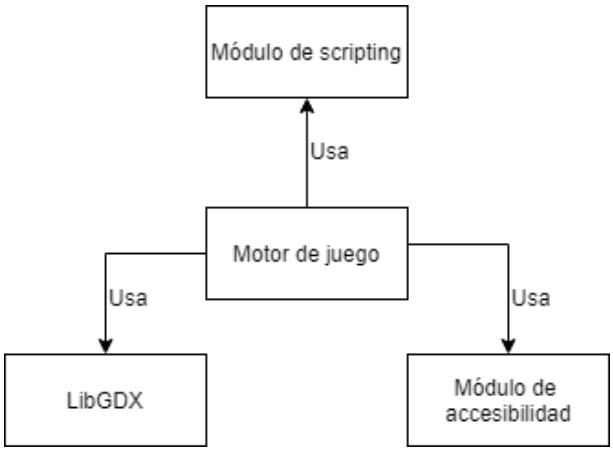

Figura 4.27: Esquema representando la interacción entre los cuatro módulos.

4.4.2 Estructuras de datos fundamentales

A continuación, se detallan las estructuras de datos implementadas, fundamentales para el funcionamiento del prototipo. En primer lugar, se describe la implementación realizada del sistema de entidades que vertebra la mayor parte de sistemas del juego. A continuación, se explica la implementación del sistema de inventario. Finalmente, se describe la implementación del laberinto desde un punto de vista de estructura de datos.

4.4.2.1 Implementación del sistema de entidades

El sistema de entidades estructura, en realidad, los sistemas de combates y eventos. Resulta, por tanto, uno de los pilares centrales en el funcionamiento del prototipo producido, de modo que las consideraciones tomadas en su diseño e implementación informaron numerosas decisiones posteriores.

La primera consideración importante es la distinción entre dos tipos de entidades: personajes de jugador y enemigos. Existen determinados puntos comunes, tales como la presencia de estadísticas que siguen una nomenclatura concreta, así como numerosos metadatos auxiliares que flexibilizan el sistema de habilidades, por lo que se creó una clase abstract Entity, que definiese todos estos conceptos de manera globalizada, y que se vería implementada por las clases PartyMember y Enemy. La propia clase Enemy es, de hecho, una clase abstracta que define todos los elementos comunes a los enemigos, como se explica más adelante. Asimismo, se generó una interfaz de carácter puramente semántico, PlayableCharacter, para flexibilizar el sistema de equipo y permitir, en caso de considerarse necesario, la generación de miembros del grupo no controlados por el jugador. Pese a que esta idea no sea utilizó en la implementación final del prototipo, el código resultante se mantuvo para facilitar la ampliación del mismo en el futuro.

Las características que una entidad puede tener son comunes a todos los posibles tipos de las mismas, cambiando únicamente sus valores específicos, y se almacenan en un atributo de tipo mapa, llamado stats.

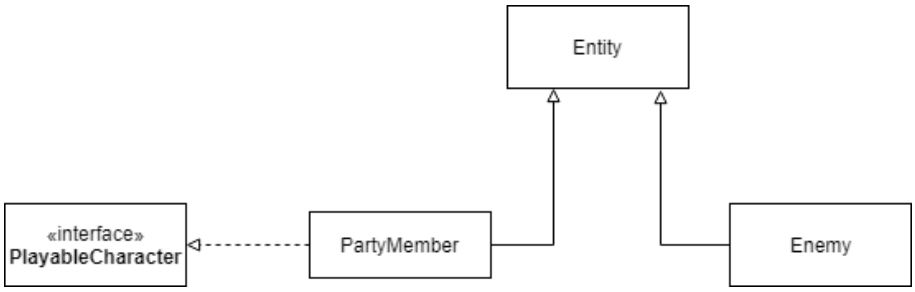

Figura 4.28: Esquema básico de herencia en el sistema de entidades.

El motivo por el que se almacenan asés muy sencillo: simplificar el acceso por parte de la capa de scripting a dichos atributos, solicitando únicamente el valor deseado en cada momento, y permitiendo obviar el resto de elementos de la clase. Lo que es más, al tratarse de una estructura de datos implementada por la práctica totalidad de lenguajes de scripting que podrían tener aplicación en el desarrollo de un videojuego, y, en concreto, por Lua de forma nativa, no es necesario generar una interfaz específica para el acceso a esta información desde la capa de scripting.

Además de sus estadísticas base, la clase entidad también almacena los modificadores aplicadosa las estadísticas, y la lista de habilidades que la entidad en cuestión tiene a su disposición, siempre como mapas. Más concretamente, el atributo modifiers, que almacena los modificadores, es un mapa que relaciona una string con un nuevo mapa, permitiendo diversificar aún más la utilidad del atributo, al permitir durante la implementación de un juego mediante el uso de este motor su uso para almacenar múltiples tipos de modificadores positivos y negativos de manera dinámica. En cuanto al atributo skills , vincula el nombre de una habilidad disponibles para la entidad a la instancia unívoca de la misma en el juego, como se explica al desarrollar el sistema de habilidades.

La clases Entity almacena, también, el nombre de la entidad, y una variable booleana que permite determinar si se está defendiendo o no para facilitar el cálculo de daño. Además de los métodos getter y setter estándar para todos estos atributos, que no se representan en el diagrama de clases extendido, la clase provee, a su vez, numerosos métodos auxiliares para facilitar la gestión del estado de las entidades del juego, de la forma más transparente y limpia posible.

La abstracción resultante permite simplificar múltiples componentes de sistemas tales como el sistema de habilidades o el sistema del combate, en los que es posible, en su mayoría, referenciar únicamente las entidades involucradas en las diversas posibles interacciones, e ignorar la clase específica a la que las mismas pertenecen.

Personajes de jugador

Como ya se ha señalado, la clase PartyMember representa a cualquier entidad que pueda formar parte del grupo. Al implementar, asimismo, la interfaz PlayableCharacter, todas estas entidades se consideran, a su vez, personajes jugables, pero se podría aplicar una sencilla modificación para separar los dos conceptos.

A los miembros de la clase entidad, PartyMember añade, además, un atributo privado de tipo String llamado status, que representa cualquier posible estado alterado que un personaje de jugador pueda sufrir; un atributo privado de tipo CharacterClass llamado clss para evitar conflictos con el compilador y que representa la clase del personaje, y un atributo privado de tipo Inventory llamado inventory que representa el inventario del personaje en cuestión y, por tanto, los objetos que posee.

La propia clase CharacterClass es una clase abstracta que define, únicamente, cómo han de estructurarse las propiedades de una clase de personaje. Más concretamente, provee información referente al

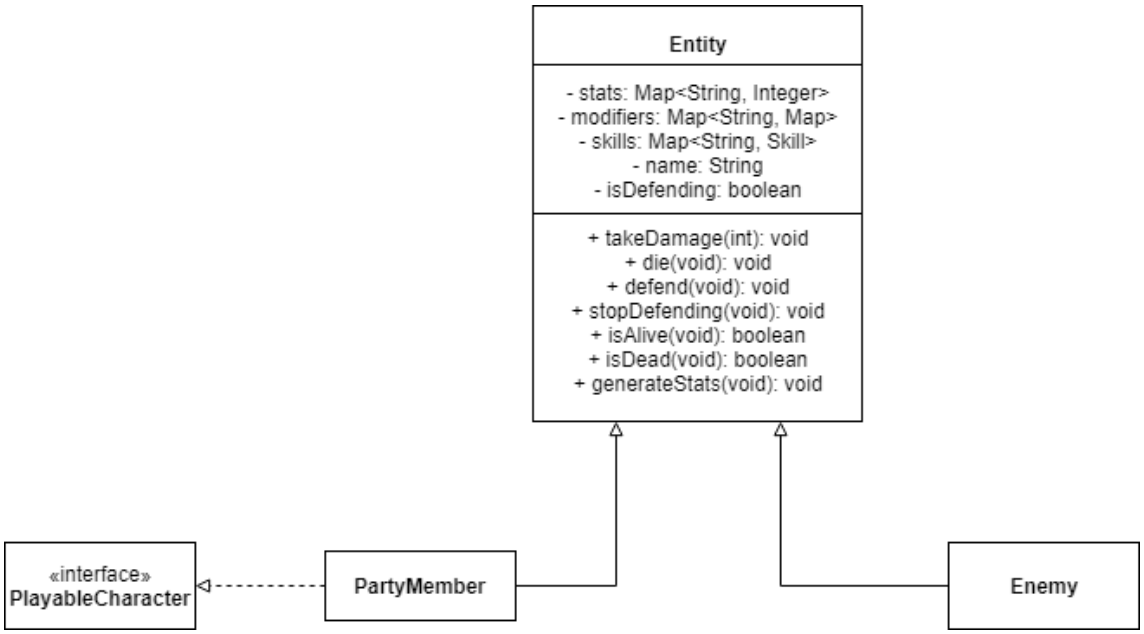

Figura 4.29: Jerarquía de clases extendida con atributos y métodos comunes.

nombre, la representación simbólica de la clase, los niveles a los que se aprenden diversas habilidades, e información necesaria para la generación de estadísticas, y variaciones de las mismas durante la subida de nivel. Todas las clases se almacenan de manera unívoca en una clase auxiliar CharacterClassManager, y se referencian en tiempo de ejecución de acuerdo a las necesidades del momento, lo que permite ahorrar memoria y mejorar el rendimiento del juego al necesitar una única instancia de cada clase para proveer servicio a todos los personajes que la posean.

Un buen ejemplo de la colaboración entre las clases PartyMember y CharacterClass se encuentra en el método createStat de la propia clase PartyMember, y en su relación con el método generateStats, tambiÃ©n de la misma clase, observable en el siguiente código fuente:

```
private int createStat(String name)
    {
        return clss.getBaseStat(name) +
                new Random().nextInt(clss.getStatModifierDice(name));
    }

@Override
    public void generateStats() {
        setStat("Max_HP", createStat("Max_HP"));
        setStat("Cur_HP", getStat("Max_HP"));
        setStat("Max_MP", createStat("Max_MP"));
        setStat("Cur_MP", getStat("Max_MP"));
        setStat("AC", 10);
        setStat("Agility", createStat("Agility"));
        setStat("Strength", createStat("Strength"));
        setStat("Attack", getStat("Strength"));
        setStat("Defense", getStat("AC"));
    }
```

Además, los miembros del equipo amplían el contenido de stats con una entrada para la experiencia obtenida, lo que permite llevar a cabo progresión de personaje individualizada.

El conjunto de miembros del equipo de exploración se gestiona mediante una clase llamada PartyData, que gestiona los personajes presentes en el grupo mediante un array estático con un número arbitrario de posiciones, y el oro que el grupo posee en un momento dado para realizar compras. Esta clase es la que la mayor parte de sistemas utilizan para gestionar la interacción con el grupo de personajes del jugador. Con el objetivo de reproducir, en la medida de lo posible, las convenciones habituales del género, se ha limitado el número de miembros máximo en un grupo a seis. Nótese que, pese a la existencia de una clase MapPartyData, la misma no contiene referencia alguna a la información de los componentes del grupo, sino a información necesaria para la exploración, diversificando así las responsabilidades.

Enemigos

La clase Enemy extiende el concepto de entidad y, de hecho, de personaje no jugador, para dar cabida a conceptos propios de los enemigos a los que los jugadores se enfrentarán. Para poder entender apropiadamente la clase Enemy, no obstante, debemos estudiar en primer lugar la clase NPC, obviada en los diagramas anteriores por no servir otra función que la de ofrecer una base ampliables desde la que construír personajes no jugador genéricos, y empleada en este TFG únicamente para crear enemigos.

A las funcionalidades de la clase Entity, NPC añade soporte para los conceptos de raza, clase y alineamiento, todos ellos aplicables para modificar el efecto de diversas habilidades que los personajes de jugador podrían aplicar a los personajes no jugador. Sobre estos conceptos, Enemy añade soporte para el concepto de resistencias, permitiendo crear puntos débiles y puntos fuertes, y una referencia al Sprite utilizado para representar al enemigo en cuestión en combate. Para reducir la complejidad del sistema de combate, ninguna de las habilidades implementadas en el prototipo final utiliza el sistema de resistencias, pero su aplicación resultaría casi trivial en caso de considerarse necesaria en un trabajo futuro basado en el mismo prototipo.

Los propios enemigos pueden heredar de la clase Enemy, o de su clase hija ScriptedEnemy, que permite especificar scripts externos para manejar la inteligencia artificial del enemigo en combate. Todos los enemigos contienen entre sus estadísticas el oro y la experiencia generados como recompensa por ser derrotados, y los mismos métodos estándar para acceder a su inteligencia artificial, con independencia de cómo esté implementada la misma, así como un método render para permitir su dibujado en pantalla de forma sencilla.

La instanciación de enemigos se maneja mediante una factoría, Enemies, que interpreta el tipo de enemigo a crear en base a su identificador, y lo instancia dinámicamente. Este sistema se utiliza de forma continua a la hora de transicionar al sistema de combate, y la propia factoría contiene funcionalidades que permiten traducir una cadena de caractéres a un identificador de enemigo de forma completamente transparente.

Cabe destacar que, a propósitos de este TFG, todas las inteligencias artificiales de enemigos que utilizan un script para determinar sus acciones se han implementado en Lua, pero se podría haber usado cualquier otro lenguaje de scripting, asumiendo que se hubiese implementado el módulo de interfaz correspondiente con anterioridad, sin perjuicio alguno para el sistema.

4.4.2.2 Implementación del sistema de inventario

El sistema de inventario se divide en dos vertientes: la implementación de los objetos propiamente dichos, y la implementación del sistema de gestión de los mismos. Por ello mismo, esta sección cubre, en primer lugar, la implementación de los objetos desde el punto de vista de las estructuras de datos y, a continuación, la implementación del sistema de inventario en su conjunto.

Implementación de los objetos

Todo objeto hereda de la clase abstracta Item, que define las propiedades elementales de los mismos. Esta clase provee la interfaz necesaria para la interacción con un objeto, así como la definición de la información imprescindible para su estructura y manejo interno. Al nombre, descripción e identificador del objeto, se suma una lista de banderas, implementada mediante un EnumSet, que define las propiedades del mundo: condicionantes de uso, potenciales objetivos o si se trata de un objeto clave, por ejemplo. Es esta lista de banderas la que se utiliza para el procesado de objetos en otros apartados del prototipo, como se explica un poco más adelante.

Además, la clase Item incluye un método para interpretar la lista de banderas de un objeto a partir de un valor númerico que las almacene en formato bit, permitiendo generar la información de objeto en un archivo externo, que sea interpretado al cargar el juego. Esta funcionalidad no se ha utilizado en la implementación del prototipo, y existe únicamente para facilitar el desarrollo de trabajo futuro.

Dos clases extienden Item en este prototipo: Vulnerary y ScriptItem. Vulnerary sirve como ejemplo de la implementación directa de un objeto en Java: basta con heredar de la clase Item, implementar el constructor correspondiente, y escribir el código que regule el efecto en el método use, y la nueva clase de objeto será totalmente funcional. ScriptItem, por el contrario, define las estructura necesaria para generar la funcionalidad de un objeto desde un script. Cabe destacar que, para poder usar ScriptItem, es necesario pasar una serie de argumentos adicionales al contructor: además del nombre y descripción, es imprescindible pasar también las banderas necesarias para la gestión del objeto en cuestión, así como el identificador de la función que implementa la funcionalidad del objeto en el archivo items.Lua, como se explica en la sección sobre la capa de scripting.

La instanciación de objetos se utiliza mediante una factoría, llamada ItemFactory, que recibe el identificador del objeto y devuelve la instancia correspondiente del mismo. Cabe destacar que, en el prototipo desarrollado, todos los objetos, incluyendo Vulnerary, se instancian como objetos de clase ScriptItem, con el objetivo de demostrar la versatilidad de dicha clase a la hora de generar contenidos.

Implementación de la estructura de inventario

Una vez explicada la clase Item, resulta fácil conceptualizar la implementación de la clase Inventory: se trata únicamente de una clase que provee múltiples métodos para interactuar con un array de tipo Item de 10 posiciones, permitiendo añadir o retirar objetos de manera ágil y dinámica, sin importar las posiciones que se encuentren libres del array, y determinar de formar sencilla si el inventario se encuentra lleno o no. El tamaño máximo del inventario se encuentra, en realidad, definida como una constante estática local de tipo **int**, llamada MAX_ITEMS, lo que permite la reconfiguración de dicho valor de manera sencilla para proyectos con características distintas a las de este prototipo.

Como se ha indicado anteriormente, cada personaje jugador tiene acceso a su propio inventario, lo que permite almacenar un máximo de 60 objetos a la vez en el inventario del grupo, asumiendo que el grupo tenga la máxima cantidad de miembros posibles. Cabe destacar que dicho inventario es unipersonal y que,

si bien la clase PartyData provee ciertas funcionalidades auxiliares para gestionar el inventario conjunto del grupo de manera más sencilla, un personaje de jugador solo tiene acceso a su propio inventario en situaciones de combate. Esta decisión ha sido premeditada, para forzar un mayor componente estratégico en la gestión de objetos por parte del jugador.

4.4.2.3 Implementación de las habilidades

El sistema de habilidades es, en realidad, uno de los pilares ocultos de la implementación del prototipo. La flexibilidad de su implementación permitió llevar a cabo ciertas mecánicas básicas sin un esfuerzo adicional. Antes de explicar cómo se ha implementado la estructura de datos básica que define el comportamiento de una habilidad, es importante mencionar que los comandos Fight, Defend y Flee del sistema de combate, así como el uso de objetos en general, utilizan internamente el concepto de habilidad, como vehículo auxiliar para facilitar su integración en el conjunto del prototipo, y su interacción con el resto de sistemas.

La clase abstracta Skill define la estructura básica de toda habilidad, así como su interfaz. Así pues, todas las habilidades en el sistema contendrán, por definición, un nombre, una descripción, una plantilla para generar texto que informe al jugador del resultado y un factor de modificación de velocidad, cuyo uso se explica al tratar el sistema de combate. Asimismo, además de los métodos getter y setter habituales, toda habilidad deberá implementar un método execute, que recibe una entidad de origen y una entidad de destino, y lleva a cabo las modificaciones de estado pertinentes.

Cinco clases descienden de Skill: Attack, Defend, Flee, UseItem y ScriptSkill. Como se puede comprobar, las cuatro primeras clases se corresponden con las funcionalidades que, como ya se ha mencionado, utilizan la estructura de la clase Skill para facilitar su implementación. La clase ScriptSkill, por su parte, permite la implementación de nuevas habilidades mediante el uso de script, de manera similar a cómo ScriptItem permite implementar nuevos objetos utilizando una función en un script.

Nuevamente, ScriptSkill requiere, además de los parámetros esperables en el contructor para Skill, el identificador de la función que implementa la funcionalidad de la habilidad en cuestión dentro del archivo skills.Lua.

Para limitar el uso de memoria por parte del prototipo, y facilitar su ejecución en sistemas con escasez de recursos, la clase Skills contiene una copia de todas las habilidades existentes en el juego, con la excepción de UseItem, que se utilizan para acceder a su funcionalidad de manera dinámica sin necesidad ampliar el uso de memoria. Es esta aproximación la que llevó a requerir las entidades de origen y destino únicamente en el método execute, flexibilizando el conjunto del sistema de combate. En cuanto a la motivación para no incluir UseItem como parte de esta estructura, es, en realidad, bastante sencilla: la necesidad de modificar en tiempo de ejecución el objeto almacenado en la instancia de la habilidad dispararía el número de llamadas y, de hecho, dificultaría su integración en el sistema de combate, actuando de forma contraria a a la filosofía general de simplificar, en la medida de lo posible, la implementación e integración de nuevos contenidos, y siendo, por consiguiente, contraproducente.

4.4.2.4 Implementación del laberinto

La implementación del laberinto es, en realidad, uno de los aspectos más fundamentales de todo este prototipo. Su conceptualización informó numerosas decisiones respecto al almacenamiento de información de estado del jugador, la construcción de la máquina de estados finita del juego, y la contrucción de numerosos sistemas, concretamente, el sistema de exploración, el sistema de combate, el sistema de generación de encuentros, el sistema de eventos, y la gran mayoría de métodos empleados para la representación del

juego en su conjunto. Se trata, por tanto, de una estructura fundamental en la elaboración del TFG, con una importancia equiparable, casi, a las técnias de accesibilidad aplicadas.

Por fortuna, las estructuras de datos implementadas para la creación del laberinto son, en realidad, relativamente sencillas. Para la gestión de estado del laberinto, encontramos MapData, una estructura de datos estática que contiene la información prefijada sobre cada planta concreta del laberinto, y MapPartyData, una estructura de datos dinámica que gestiona la información referente a la posición y orientación del grupo de personajes del jugador dentro del laberinto. Así, la información queda claramente dividida en dos conceptos funcamentales: la estructura del laberinto propiamente dicho y la situación del jugador dentro del mismo.

Comencemos, por tanto, por estudiar la estructura de MapData. Esta clase contiene como atributos una referencia a la música de fondo del mapa, en la forma del atributo bgm; un array bidimensional de casillas, llamado mapData; información sobre los eventos del mapa, almacenada en el atributo eventsLayer; una lista de enemigos presentes en la planta, almacenada en el atributo enemies, y el número máximo de enemigos en combate en esta planta, en el atributo maxEnemies. Además de los correspondientes getter y setter, la interfaz de MapData ofrece métodos para gestionar la música de fondo, acceder a información de casillas concretas, y gestionar la presencia de muros y pasabilidad de casillas en múltiples direcciones, a raíz de unas coordenadas dadas.

MapPartyData, por su parte, es una estructura de datos significativamente más sencilla, conteniendo únicamente la posición y orientación del grupo, y ofreciendo en su interfaz, además de los correspondientes getter y setter, métodos para abstraer las operaciones de girar a izquierda y derecha, avanzar, y retroceder, además de métodos auxiliares para predecir la posición resultante de un desplazamiento, y para obtener la dirección de desplazamiento hacia adelante y hacia atrás en un momento dado.

Como se puede comprobar, ambas clases son, en realidad, independientes la una de la otra, y requieren de un puente que las una. El razonamiento es sencillo: limitar el acoplamiento entre clases, y delimitar de forma clara las responsabilidades de las mismas, para facilitar el mantenimiento y ampliación de las mismas.

Estructura de las casillas

La conceptualización de las casillas que componen el laberinto resulta una parte integral de la implemenación del mismo, por lo que se ha decidido dedicar este espacio a explorar y explicar las decisiones tomadas para tal propósito.

La estructura de datos que almacena la información de una casilla es la clase Tile, que contiene información fundamental sobre la misma, así como métodos para facilitar su interpretación e interpretación en tiempo de ejecución. Más concretamente, la información contenida es un conjunto de banderas, representado como un EnumSet, e información sobre el alto y ancho de la casilla respecto a la pantalla. Así, es posible acceder a esta información para navegar el laberinto, para representar la casilla desde un punto de vista en primera persona, y para dibujar la casilla en el minimapa.

Resulta necesario, por tanto, observar cómo se define la enumeración de posibles valores de un casilla, de modo que se adjunta el correspondiente código fuente a continuación:

```
public enum TileFlag{NORTH_WALL, SOUTH_WALL, EAST_WALL, WEST_WALL, FREE,
        OUT_OF_BOUNDS};
```

Como se puede comprobar, las banderas han sido nombradas con nombres fácilmente identificable, que permiten comprender de forma rápida el contenido de una casilla, y calcular la pasabilidad de la

misma en una dirección determinada. Es importante tener en cuenta que FREE y OUT_OF_BOUNDS son banderas incompatibles con ninguna otra, y que los muros pueden usarse en cualquier combinación que se desee. Sin embargo, las banderas de las casillas tienen mayor profundidad de la que podría parecer en un primer momento, así que se tratan en mayor detalle en el siguiente apartado.

Banderas de las casillas

Para poder generar mapas mediante un programa externo, resulta imprescindible, dada la conceptualización de las casillas, contar con algún método para almacenar las mismas en un archivo externo en un formato coherente, independiente de la implementación del archivo, y fácilmente interpretable por parte del motor a la hora de leer y cargar dicha información. Finalmente, se optó por establecer una correspondencia directa entre las banderas anteriormente descritas como parte de una enumeración, y banderas definidas a nivel bit.

La estructura resultante se muestra a continuación, ignorando determinados métodos auxiliares que no contribuyen a la comprensión del resultado:

```
public enum TileFlag{
    NORTH_WALL(1), SOUTH_WALL(4), EAST_WALL(8), WEST_WALL(16),
    OUT_OF_BOUNDS(128),
    FREE(0){
        @Override
        public boolean isSet(int flags){return flags == 0;}
    };

    private int value;

    TileFlag(int flags)
    { this.value = flags;}

    public boolean isSet(int flags){return (flags & value) == value;}
}
```

Como se puede comprobar, se extendió la funcionalidad de las banderas, dotándolas de métodos auxiliares para poder comprobar si cada una de ellas estaba establecida a nivel bit, y facilitar la interpretación. El funcionamiento esperado de este sistema es en realidad bastante sencillo: basta utilizar el método isSet para determinar si una bandera está en estado alto dentro de un valor numérico, aprovechando el funcionamiento de la operación AND a nivel bit, y gestionar el resultado de dicha operación.

En cuanto a la implementación de un método isSet específico para la bandera FREE, el motivo es, una vez más, sencillo: dado que su valor es cero, la operación realizada en el método general resultaría siempre en **true**, al anular cualquier posible valor de entrada, de modo que la única forma de confirmar que realmente se está dando el caso correspondiente es realizar una comparación directa.

Finalmente, la presencia de huecos arbitrarios se debe a la preparación de posibles escenarios en las que dichos bits deban utilizarse para nuevas posibilidades en versiones futuras. En este prototipo, los bits 1, 5, y 6 del primer byte, asumiendo que el primer bit es el bit 0, no se utilizan para nada.

Estructura de los eventos

La implementación de los eventos sigue, una vez más, una filosofía similar a la aplicada para implementar los objetos y habilidades: la generación de una estructura abstracta que sirva como base para generar contenidos de forma ágil y sencilla, y que reuna todos los elementos fundamentales de un evento en una única clase. Dicha estructura es la clase MapEvent, que, en colaboración con la interfaz Condition, vertebra el conjunto del sistema de eventos.

Describamos, en primer lugar, la clase MapEvent. Esta clase contiene información sobre la posición del evento en el mapa correspondiente, las condiciones que han de cumplirse para que se active, y si su activación se produce de manera automática. Además, ofrece métodos para determinar si se ubica en una posición determinada, si las condiciones se cumplen y disparar el evento. Finalmente, es capaz de interpretar las condiciones necesarias para disparar el evento a partir de un array de String, generando automáticamente su propia lista de condiciones en base a datos en texto plano.

La interfaz Condition, por su parte, establece únicamente un método con tipo de retorno booleano llamado isFulfilled , que determina si la condición se ha cumplido, y que deberá ser debidamente implementado. La motivación para usar una interfaz es doble: por un lado, era únicamente necesario definir métodos abstractos, carentes de atributos, característica definitoria de las interfaces en el paradigma de programación orientada a objetos; por otro, era necesario poder agrupar objetos de diverso caracter, que cumpliesen esta misma función, en el interior de un objeto de evento. Así pues, dada la mejor adecuación a la función deseada que la de una clase abstracta, se optó por utilizar una interfaz.

Tanto los eventos como sus condiciones se almacenan, de manera estática, en sendas estructuras de datos auxiliares, idénticas a la empleada para almacenar las habilidades, y llamadas MapEvents y Conditions, respectivamente. La motivación es, parcialmente, la misma que para realizar esta operación con las habilidades: reducir el uso de memoria en tiempo de ejecución, para facilitar el despliegue en plataformas con recursos más limitados. Existe, no obstante, una motivación secundaria: permitir la sencilla interpretación de los archivos de mapeados en que se referencian los eventos, y la generación de la correspondiente factoría de eventos.

Uso de Tiled

Hasta ahora hemos tratado la implementación de las estructuras de datos que sustentan la exploración, pero resulta imprescindible analizar la integración del editor de niveles Tiled en el conjunto de las mismas. Como ya se ha mencionado, dicho editor de niveles almacena los mapas generados en formato TMX, un archivo XML que LibGDX es capaz de interpretar de manera nativa. Esta funcionalidad simplifica en gran medida el proceso, pues basta con definir apropiadamente las propiedades necesarias para su aplicación al prototipo, así como un intérprete adecuado, para integrarlo en el flujo de trabajo del proyecto.

La clase TiledMapParser es la encargada de realizar la interpretación del archivo de mapa, siguiendo un proceso automático relativamente sencillo:

1. Se capturan las propiedades del mapa.
2. Se extraen la altura y anchura del mapa de las propiedades capturadas.
3. Se interpretan las casillas de la capa de patrones, utilizando para ello las banderas anteriormente descritas.
4. Se interpretan los eventos almacenados en la capa de eventos, y se almacenan en formato String.

5. Se interpreta el nombre de la música de fondo del mapa, y se utiliza para cargar el archivo correspondiente.
6. Se interpreta la lista de enemigos presentes en el mapa.
7. Se interpreta la cantidad máxima de enemigos en la planta actual.
8. Se devuelve el MapData completamente configurado.

Sin embargo, para poder ofrecer soporte a este sistema de interpretación, es necesario configurar el archivo Tiled con algunas consideraciones específicas. Concretamente:

- El conjunto de patrones ha de haber sido configurado previamente con las banderas escritas a nivel bit, de modo que sea posible intepretarlas correctamente.
- La capa de patrones que representa el mapeado ha de llamarse "labyrinth", independientemente de su posición en la jerarquía de Tiled.
- La capa de objetos que representa los eventos del mapeado ha de llamarse .events", independientemente de su posición en la jerarquía de Tiled.
- El mapa ha de contener una propiedad "bgm", de tipos String, que represente la ruta relativa al archivo de música ha reproducir durante la exploración.
- El mapa ha de contener una propiedad .enemies", en la que se escriban los identificadores de los enemigos presentes en el mapa separados por ';'.
- El mapa ha de contener una propiedad de tipo entero llamada "max_enemies", que deberá ser mayor que cero, e igual o menor que tres.

Asumiendo una configuración adecuada, el motor es ahora capaz de interpretar el mapa de manera dinámica y correcta, y la tarea de implementar mapeados se ve simplificada en gran medida respecto a la posibilidad de implementarlos directamente en código.

4.4.3 Implementación del sistema de guardado

Para implementar el sistema de guardado, se han distribuido los datos necesarios para el sistema en su conjunto en dos grupos: datos volátiles y datos persistentes. Ambos se almacenan mediante un patrón pizarra, permitiendo el acceso y modificación desde diversos puntos del programa sin necesidad de intermediarios, y utilizan un patrón singleton como soporte para dicho comportamiento. Más concretamente, se han implementado las clases VolatileData y PersistentData.

VolatileData contiene la información respecto al mapa actual y a la posición del grupo dentro del mismo, que no es necesario recuperar más adelante, de modo que se pueda acceder tanto desde el estado de exploración del laberinto como desde eventos que puedan necesitar leer o modificar esta información.

PersistentData, por su parte, es fundamentalmente más compleja. Provee funcionalidad para guardar y cargar información, y expone, de manera consciente, información sobre los datos persistentes del juego. Dicha información, a su vez, se almacena en una clase interna, GameData, y está compuesta por:

- Información sobre banderas, necesaria para crear condiciones de eventos y generar una secuencia de juego en narrativas más complejas. Dada la sencilla naturaleza del prototipo, esta funcionalidad no se utiliza en su totalidad, pero se ofrece soporte para comportamientos complejos.

- Información sobre la composición actual del grupo, concretamente, la instancia única de PartyData.

- Información sobre todos los personajes de jugador registrados, en forma de un array de tipo PartyMember.

El proceso de guardado de datos se divide en dos pasos: en primer lugar, se codifica la información contenida en GameData en un JSON; en segundo lugar, el JSON se cifra usando Base64 y se almacena en el archivo "bin/saveData.bin".

En cuanto al proceso de cargado de datos, se traduce en invertir los dos pasos del proceso de guardado: en primer lugar se lee y descifra el archivo "bin/saveData.bin", y, a continuación, se regenera GameData a partir del JSON resultante. Cabe destacar que la función encargada de este proceso contiene comprobaciones de seguridad que generarán una nueva partida en caso de no haber ninguna guardada previamente.

Cabe destacar que, para poder guardar apropiadamente el inventario del jugador, ha sido necesario escribir código de serialización personalizado. Concretamente, el atributo type de la clase Item, implementado como una EnumSet en el que se almacenan las banderas que configuran el tipo concreto de objeto, entraba en conflicto con el funcionamiento estándar de la serialización JSON, debido a la ausencia de un constructor vacío para la clase EnumSet. Ha sido, por tanto, necesario, establecer una estrategia de escritura y lectura que circunvalase el uso directo de dicha instancia, para lo que se recurrió a la escritura de las banderas a nivel bit. Fue necesario implementar la interfaz Serializable, contenida en el paquete com.badlogic.gdx.utils.Json, para poder establecer esta funcionalidad. El código de escritura a JSON resultante, implementado por comodidad en la clase ScriptItem, es el siguiente:

```
@Override
    public void write(Json json) {
        json.writeValue("name", this.name);
        json.writeValue("description", this.description);
        json.writeValue("id", this.id);
        json.writeValue("type", this.generateFlagSet());
        json.writeValue("behaviorId", this.behaviorId);
    }

    @Override
    public void read(Json json, JsonValue jv) {
        name = json.readValue("name", String.class, jv);
        description = json.readValue("description", String.class, jv);
        id = json.readValue("id", Integer.class, jv);
        type = parseType(json.readValue("type", Integer.class, jv));
        behaviorId = json.readValue("behaviorId", String.class, jv);
    }
```

Como se puede comprobar, la se utilizan métodos auxiliares para convertir el valor del EnumSet a un valor entero, y para descodificar dicho entero. La implementación de este código permitió resolver, de forma limpia y sencilla, los problemas ocasionados por la colisión entre las necesidades de las estrategias de serialización implementadas por LibGDX y la implementación estándar de la clase EnumSet en el lenguaje Java.

4.4.4 Implementación de la máquina de estados finita

La máquina de estados es un componente fundamental de la mayor parte de videojuegos. No es de extrañar: la posibilidad de abstraer el manejo de la actualización y representación del videojuego respecto al bucle de ejecución supone una herramienta fundamental a la hora de implementar productos con un mínimo de complejidad.

Dada la naturaleza del prototipo implementado, existe un gran número de estados y subestados posibles, cada uno asociado a uno o varios sistemas, y un flujo constante entre los mismos. Resultaba imprescindible, a consecuencia, implementar la maquina de estados de tal manera que el flujo entre los mismos fuese lo más razonable y transparente posible, al tiempo que se minimizase el número de veces que un mismo estado era instanciado a lo largo de una misma partida. Tras un cierto estudio de diversos títulos dentro del género de los RPG por turnos, se evidenció la necesidad de poder regresar al estado anterior de manera ágil, presente en la práctica totalidad de los estados que los videojuegos de este tipo utilizaban, por lo que se determinó que, en contraposición con una máquina de estados clásica en la que se conservaría información sobre un único estado y se establecerían mecanismos para transicionar a los estados adyacentes, se implementaría la máquina de estados como un estructura de pila. Esta implementación permitiría regresar de forma ágil con independencia del estado actual, permitiendo reducir la cantidad de información a almacenar y facilitando el manejo.

Así, la clase MyGdxGame, generada automáticamente por el framework, contiene un Stack de tipo GameState que utiliza para simplificar el método de actualización. Por restricciones del propio LibGDX, este método recibe el nombre de render, y se encarga de actualizar y dibujar el estado actual del juego. Aplicada la máquina de estados, el método resultante es el siguiente:

```
@Override
public void render () {
        if(!cm.isEmpty()) processCommand(cm.retrieve());
        stateStack.peek().update();
        stateStack.peek().render();
        if(debug) printFPS();
}
```

En primer lugar, el método determina la necesidad de gestionar el estado y, en caso afirmativo, procesa la gestión a realizar, como se explica más adelante. Una vez completado este proceso, el juego actualiza el estado activo, situado en la parte superior de la pila, y lo representa. Finalmente, en caso de tratarse de una build de depuración, el juego muestra en pantalla el ratio de frames por segundo, para permitir evaLuar el rendimiento del juego.

Lo que es más, esta estructura básica nos permite vislumbrar los métodos más importantes comunes a todos los estados: update y render. Cada estado ha sido implementado como una clase hija de GameState, pero todos ellos tienen en común el contener información sobre el SpriteBatch principal del juego, para poder dibujar en pantalla; sobre su interfaz, en caso de considerarse necesario registrar elementos con nombre específicos (funcionalidad que prácticamente no se ha utilizado en el desarrollo del prototipo, pero que resultaría fundamental para el desarrollo de proyectos de mayor calibre), e información respecto a si el estado está activo. Este último elemento, aparentemente superfluo, se incluyó como una revisión durante el desarrrollo, al presentarse la necesidad de realizar actualizaciones cosméticas sobre estados inactivos. Una aplicación clara de este concepto es la actualización de la animación de un cursor inactivo, son modificar su estado cada vez que el jugador realiza una entrada.

Cada una de las pantallas descritas en la sección de diseño de juego se traduce en un estado, que puede

contar con numerosos subestados dependiendo del caso, por lo que las necesidades de cada estado son claramente variables. Es por ello por lo que surge la necesidad de establecer un sistema de comandos que permita agilizar la gestión del estado de juego, de manera completamente transparente para el framework, al tiempo que se evalúan y suplen las necesidades de cada uno de esos estados durante su instanciación.

4.4.4.1 Implementación de la gestión de estados

La gestión de estados se estructura en base a un sistema de mensajes y comandos internos, que permite transicionar de forma casi transparente, y centralizar la responsabilidad de gestionar los cambios en la máquina de estados en un único punto del bucle de ejecución. Para ello, se han generado dos familias de clases: los comandos y los procesadores.

Los comandos, que extienden la clase abstracta Command, contienen un mensaje, así como, opcionalmente, un conjunto de argumentos y un payload, que, se asume, serán necesarios para la correcta ejecución del comando. El mensaje está implementado como una valor dentro de una enumeración, con el objetivo de simplificar su interpretación mediante el uso de una estructura tipo **switch**, mientras que los argumentos y payload del comando están caracterizados como elementos de tipo Object. Así, los comandos contienen, únicamente, la información pertinente a la solicitud de ejecutar una operación, esto es, el equivalente a cualquier orden o invocación de función.

Los procesadores, por su parte, se encargan de procesar y llevar a cabo comandos concretos. Heredan de la clase CommandProcessor, y son los verdaderos en cargados de asegurarse de que la orden se ejecuta correctamente. Son los procesadores los que se encargan de realizar la modificación de estado pertinente sobre el juego, para lo que utilizan los argumentos y payload del correspondiente comando.

En cuanto a la interacción entre ambos tipos de componente, es el propio juego el encargado de mediar, dado que se trata de operaciones que impactan directamente sobre la máquina de estados. Concretamente, el juego selecciona el procesador adecuado para procesar un comando determinado, y solicita su ejecución en el momento correcto del bucle de juego. Además, la propia clase juego pone a disposición del resto del sistema una cola de mensajes, sobre la que depositar sus solicitudes en forma de comandos, de forma que la ejecución de los mismos sea ordenada y controlada, y se puedan aplicar medidas de seguridad para evitar situaciones anómalas en el estado del programa en la medida de lo posible.

Esta estructura permite ocultar la complejidad del proceso de la lógica de los estados en su mayor parte, y simplifica la lectura y mantenimiento de las mismas. Lo que es más, al diversificar las responsabilidades, los efectos de un cambio en la especificación se ven minimizados, permitiendo agilizar los procesos de refactorización, corrección de errores y expansión que, como en toda aplicación informática, habrán de producirse eventualmente como resultado del proceso normal de mantenimiento y desarrollo de la aplicación.

4.4.5 Implementación de la capa de scripting

Para facilitar la creación de contenidos específicos, se ha desarrollado una capa de scripting. La motivación para ello es doble: en primer lugar, la mayor cercanía al lenguaje natural, unido al hecho de que se trata de lenguajes interpretados, permite agilizar los procesos de escritura y depuración del código; en segundo lugar, el uso de una capa de scripting para desarrollar contenidos a alto nivel es un estándar habitual en la industria. Precisamente por su extendido uso en proyectos reales relacionados con videojuegos, el módulo específico de scripting se ha desarrollado para proveer soporte a Lua.

La capa de scripting se divide en dos secciones claramente diferenciadas: un módulo que hace de puente entre Java y los diversos scripts empleados en el proyecto, y los scripts propiamente dicho. Es

importante, por consiguiente, observar la filosofía de diseño del puente que se ha implementado para poder comprender en su totalidad el encaje de los diversos scripts generados en el conjunto del proyecto.

El módulo de scripting en Java está compuesto, en la versión actual del prototipo, por cuatro elementos: la interfaz Script, la clase ScriptManager, la clase ExecuteScript y la clase LuaScript. De entre todas ellas, solamente LuaScript y ExecutScript son dependientes del lenguaje de scripting escogido, y el resto de elementos sirven para crear una fachada que simplifica el uso del módulo. Pasemos, por tanto, a detallar cada uno de estos elementos por separado.

4.4.5.1 Interfaz Script

La interfaz Script es, de hecho, uno de los dos pilares fundamentales de la capa de scripting: define todas las funcionalidades que permiten utilizar y gestionar el estado de un script dado, con independencia del lenguaje empleado o los detalles de implementación específicos de la comunicación con el mismo, y permite la creación de mecanismos universales de interacción con los scripts empleados con el juego. La filosofía de diseño es, en realidad, realmente sencilla: todo script habrá de proveer soporte a métodos de control, que permitan determinar si es posible realizar la ejecución de funciones específicas o si el script ha sido inicializado; un método de inicialización, que permita garantizar que la comunicación entre los dos lenguajes de programación ha sido establecida correctamente; un método de ejecución de funciones específicas, y un método de recarga, en caso de detectarse alteraciones al archivo fuente.

El método canExecute es el encargado de informar al resto del sistema de si se pueden ejecutar funciones del archivo fuente representado por el objeto Script correspondiente. Normalmente, devolverá **true** siempre y cuando el archivo al que referencia el objeto exista, y **false** en cualquier otro caso, pero determinadas librerías podrían alterar ligeramente este comportamiento. En cualquier caso, la función es la misma: permitir al motor detectar, de manera preventiva, situaciones en las que resulte imposible acceder a un script dado, y gestionarlas de la forma más segura posible.

El método polimórfico executeInit permite ejecutar la inicialización del script, en caso de que exista. Admite cualquier número de objetos como entrada, y los utiliza para realizar la inicialización en caso de ser necesario. Precisamente para poder implementar esta funcionalidad de la forma más transparente posible, se ha recurrido al polimorfismo: una versión del método no recibe ningún argumento, mientras que la otra recibe una lista de objetos de tamaño arbitrario definida como Object ... en la signatura de la función. En cualquier caso, el resultado es el mismo: la inicialización del script, de acuerdo a las necesidades específicas de la implementación. Ambas versiones devuelven **true** en caso de que la inicialización se haya ejecutado de forma existosa, o **false** si la inicialización ha fallado.

El método polimórfico executeFunction permite la ejecución de funciones arbitrarias dentro de un script. Una vez más, existen dos versiones: una que recibe únicamente el nombre de la función como argumento, y una que, además, recibe una lista de parámetros para llamar a la función. Ambas versiones devuelven **true** si la ejecución de la función solicitada ha sido existosa, o **false** si ha fallado.

Finalmente, el método reload permite forzar al motor a recargar e interpretar el archivo representado por el correspondiente objeto Script, devolviendo **true** en caso de éxito, o **false** en caso de error.

4.4.5.2 Clase ScriptManager

Implementada siguiendo una aproximación similar a una clase estática, dentro de los límites establecidos por Java, la clase ScriptManager se encarga de gestionar la carga, descarga, recarga y acceso a scripts de manera centralizada. Cuenta con dos atributos estáticos, scripts y json, que se inicializan en el momento de llamar al constructor, y numerosos métodos estáticos. Antes de explicar cuál es la función de cada

uno de estos métodos, es conveniente detallar la estructura y comportamiento de los dos atributos, por tratarse de elementos fundamentales en el comportamiento del gestor en su conjunto.

- scripts es un objeto de tipo Map<String, Script> que establece la correspondencia entre un identificador y un archivo de script previamente cargado, y que se utiliza para permitir el uso de scripts sin recurrir a su ruta como método de selección.
- json es un objeto de tipo JsonValue que actúa como índice de los scripts conocidos por el motor. Para ello, carga sus valores del archivo ubicado en /data/scripts/scripts.json". Este archivo contiene las correspondencias entre identificador y ruta, y permite cargar diversos scripts utilizando únicamente su identficador. Lo que es más, su presencia permite dificultar la modificacón del código fuente del juego con fines maliciosos por parte de terceros, añadiendo en el proceso una pequeña, aunque reconocidamente débil, capa de seguridad adicional.

El uso conjunto de estos dos atributos permite agilizar el uso de scripts desde cualquier otro módulo del sistema, centralizando el acceso a la capa de scripting mediante una fachada efectiva, que cuenta con los siguientes métodos estáticos para solicitar las diversas funcionalidades de la misma:

- load: Recibe un objeto de tipo String como entrada, correspondiente a uno de los identificadores recogidos en json, y trata de cargar el correspondiente archivo, en caso de existir. Devuelve **true** si la carga del archivo ha resultado exitosa, o **false** en cualquier otro caso.
- add: Añade al diccionario interno del gestor un nuevo script, utilizando para ello la combinación de clave identificadora y la ruta al archivo correspondiente. El objeto de tipo Script se instancia bajo demanda y, tras comprobar que el archivo existe y se puede ejecutar, se añade al diccionario. Este método devuelve **true** en caso de que el script se haya añadido exitósamente al diccionario interno del gestor, o **false** en cualquier otro caso, y es utilizado internamente por load.
- reload: Recarga el archivo correspondiente a una clave dada. Devuelve **true** si la operación tiene éxito, o **false** si falla, modificando el contenido del diccionario interno únicamente en caso de éxito.
- executeFunction: Ejecuta la función indicada del archivo fuente correspondiente a la clave provista. En caso de que la clave no exista en el diccionario interno, el método intenta cargar el archivo en el sistema. Este método es polimórfico, permitiendo ejecutar funciones con cualquier número arbitrario de argumentos. Si la función se ejecuta con éxito, devuelve **true**, y si no, devuelve **false**.
- executeInit: Ejecuta la función init del archivo indicado. Más allá de ejecutar esa función específica, el comportamiento es idéntico al de executeFunction, incluyendo el uso del polimorfismo.
- executeUpdate: Ejecuta la función update del archivo indicado. Más allá de ejecutar esa función específica, el comportamiento es idéntico al de executeFunction, incluyendo el uso del polimorfismo.
- dispose: Libera la memoria utilizada por el gestor, para evitar fugas de memoria. No debería llamarse en tiempo de ejecución, y existe únicamente para facilitar el proceso de limpieza al salir del juego mediante una llamada a Gdx.app.exit() en algún punto del código.

Como se puede comprobar, la interfaz resultante permite la interacción transparente del resto del sistema con el módulo de scripting, sin por ello necesitar un conocimiento profundo del funcionamiento del mismo. Basta con conocer los contenidos del archivo json que actúa como índice, y del script al que sea acceder, para porder utilizarlos efectivamente, dado que el propio gestor se encarga de cargar los archivos en caso de ser necesario, y los métodos implementados para dicho fin se encuentran expuestos únicamente para flexibilizar la interacción con esta capa.

4.4.5.3 Clase LuaScript

La clase LuaScript implementa de manera directa la interfaz Script, utilizando la biblioteca LuaJ para mediar entre los dos lenguajes de programación involucrados. Para ello, la libreria registra, en primer lugar, las variables globales del entorno de ejecución Java al entorno de ejecución Lua generado, permitiendo la interacción entre ambos lenguajes a un nivel elemental. Dicho proceso se realiza de manera automática al invocar al contructor de la clase, proceso que también se encarga de realizar la carga del script.

La carga del script almacena los datos leídos como un objeto de tipo LuaValue llamado chunk, que representa el conjunto de funciones que componen al archivo de script fuente. Este chunk se ejecuta, a continuación, para añadir dichas funciones, ya interpretadas, al espacio de variables globales definido en globals.

En cuento a la ejecución específica de funciones Lua, el proceso se ve ampliamente simplificado por la aproximación adoptada en la definición de la interfaz Script. Una vez cargado el script, y solicitada la ejecución de una función específica dentro del mismo, el proceso seguido es el siguiente:

1. El script comprueba si es posible realizar la ejecución.
   - Si no es posible, devuelve **false** y termina.
   - Si es posible, procede al siguiente paso.
2. A continuación, se recupera el elemento con el nombre indicado.
3. Se comprueba si el elemento es una función.
   - Si resulta no serlo, devuelve **false** y termine.
   - Si resulta serlo, se procede al siguiente paso.
4. Se capturan los parámetros, y se traducen a formato Lua mediante una llamada a CoerceJavaToLua#coerce, función provista por LuaJ.
5. Se ejecuta la función con los parámetros traducidos, capturando cualquier excepción que pueda producirse.
   - Si se produce alguna excepción, ésta se captura y muestra en el log, devolviendo **false** a continuación.
   - Si, por el contrario, la ejecución se completa sin errores, se devuelve **true** para indicar el éxito.

4.4.5.4 Clase ExecuteScript

La clase ExecuteScript es una clase implementada utilizando el patrón de diseño **command**, y el patrón de instanciación **singleton** con el objetivo de facilitar la interacción entre scripts Lua usando Java como intermediario. Su diseño es relativamente complejo, y su implementación resulta poco clara, de modo que es adecuado analizarla parte por parte.

En primer lugar, es importante denotar que la clase ExecuteScript contiene una clase interna, ExecuteScriptImplementation, que se encarga del verdadero manejo de llamadas a scripts desde Lua. La clase ExecuteScript se encarga únicamente de actuar de interfaz para simplificar el acceso a esta funcionalidad, del siguiente modo:

- getInstance es un método estático común a toda clase que implemente el patrón singleton: instancia, en caso de ser necesario, el objeto único, y devuelve la correspondiente referencia común.

- call es el verdadero encargado de actuar de interfaz: lo que hace es instanciar un nuevo objeto de la clase ExecuteScriptImplementation, lo registra en el entorno Lua correspondiente, y devuelve el resultado, efectivamente habilitando el acceso a la funcionalidad encargada de comunicar distintos scripts.

Nótese que ExecuteScript implementa la interfaz TwoArgFunction, provista por la librería LuaJ, lo que identifica a la clase como un wrapper para una función escrita en Java y accesible en Lua que recibe dos argumentos para su funcionamiento. El método call es un método requerido por dicha interfaz, al igual que los argumentos y tipos de salida específicados.

En cuanto a ExecuteScriptImplementation, su funcionamiento está regido por un único método, call, que se encarga de traducir los parámetros requeridos por ScriptManager#executeFunction de formato Lua a formato Java, y de traducir el resultado de dicha función de vuelta a formato Lua, para su uso directo desde un script. Esta clase interna implementa la interfaz ThreeArgFunction, también provista por LuaJ, y que una vez más identifica a la clase como un wrapper para el uso de una funcionalidad Java en Lua, que, en este caso, requiere tres argumentos para su invocación.

4.4.6 Implementación de la capa de accesibilidad

La capa de accesibilidad basa su funcionalidad en el uso de la librería Tolk, desarrollada por Davy Kager, Leonard de Ruijter, Axel Vugts y QuentinC, y distribuida bajo licencia LGPLv3. La librería está escrita en C++, y crea una interfaz con numerosos lectores de pantalla mediante el uso de DLLs y COMs para acceder a las correspondientes APIs. La librería se distribuye con wrappers para Java, C#, VB.NET, Python, Autolt y PureBasic, y se ha escogido por su gran compatibilidad con múltiples lectores de pantalla, a la par que oculta la complejidad del proceso mediante el uso de una fachada.

A continuación, se adjunta una tabla detallando los lectores de pantalla con los que la librería es compatible, extraída de la documentación oficial, a la que se ha añadido, además, información sobre los sistemas operativos compatibles con el correspondiente lector de pantalla.

Lector de pantalla	Voz	Braille	Estado	x86	x64	Sistemas operativos
JAWS	Sí	Sí	No	Sí	Sí	Microsoft Windows
Window-Eyes	Sí	Sí	No	Sí	Sí	Microsoft Windows
NVDA	Sí	Sí	No	Sí	Sí	Microsoft Windows
SuperNova	Sí	No	No	Sí	No	Microsoft Windows
System Access	Sí	Sí	No	Sí	Sí	Microsoft Windows
Zoom Text	Sí	No	Sí	Sí	Sí	Microsoft Windows
SAPI	Sí	No	Sí	Sí	Sí	Microsoft Windows

Tabla 4.3: Listado de lectores de pantallas compatibles con Tolk, en orden en que la librería intenta acceder a ellos, incluyendo sus características fundamentales.

Como se puede comprobar, la gran debilidad de la librería elegida es el hecho de que solo ofrece compatibilidad con sistemas Windows. La generación de un modulo de accesibilidad universal para LibGDX, detallada en líneas generales más adelante, queda como trabajo futuro, debido a las limitaciones en los recursos disponibles para la elaboración de un TFG.

Integración de Tolk

La integración de Tolk requirió ligeras modificaciones al script de compilación Gradle generado durante la creación del proyecto. Concretamente, fue necesario indicar la necesidad de compilar como dependencia,

también, el contenido de la carpeta libs del proyecto, añadida para alojar el JAR que hace las veces de wrapper para Tolk, así como las correspondientes DLL para su correcto funcionamiento. Cabe destacar que, dado su carácter local, el resultado es completamente portable, dado que el sistema busca las DLL mediante una ruta relativa.

Una vez salvado ese primer obstáculo, la compatibilización no podría ser más sencilla. Dos métodos, cuyas funciones se explican a continuación ganan especial protagonismo: Tolk#silence y Tolk#output. Tolk#silence solicita al lector de pantalla interrumpir el output sonoro que se esté produciendo en un momento dado, en caso de ser necesario, y liberar la cola de sonidos correspondiente. Tolk#output, por el contrario, solicita al lector de pantalla el producir una salida de texto determinada de acuerdo a la configuración del usuario, de modo que, aquellos usuarios que prefieran usar únicamente una pantalla braille y tengan un lector de pantalla compatible, recibirán la salida por dicho medio, en lugar de mediante una señal auditiva.

Para poder acceder a estas funcionalidades es necesario, no obstante, haber ejecutado el método Tolk#load en primer lugar, de modo que se haya establecido el puente con el correspondiente lector de pantalla. En caso contrario, las funcionalidades correspondientes no producirán una salida. Finalmente, es necesario ejecutar el método Tolk#unload al finalizar la ejecución, para evitar problemas de fuga de memoria.

Así pues, el método Tolk#load se ejecuta durante la inicialización del juego, antes de finalizar la carga de la máquina de estados, para garantizar su disponibilidad cuando sea necesario. Todos los elementos de interfaz cuyo contenido es susceptible de ser accedido por un usuario de lector de pantalla cuentan con un método feedReader, que se encarga de ejecutar las llamadas oportunas de acuerdo a la información que se ha de transmitir al usuario. En general, estas llamadas implican únicamente ejecutar Tolk#silence y Tolkd#output de manera consecutiva, utilizando la cadena de text correspondiente como único argumento para el último método, pero existen interfaces específicas que requieren de un procesado extra para garantizar que la información aportada es correcta y en un formato comprensible.

Además, en el estado de exploración, cuyo funcionamiento se detalla más adelante, se incluyen llamadas auxiliares a las funciones Tolk#silence y Tolk#output para garantizar que los usuarios con ceguera total sean capaces de orientarse durante su exploración del laberinto, quedando solo algunos elementos auxiliares, pero no por ello imprescindibles, fuera de su alcance.

Finalmente, la llamada a Tolk#unload se realiza en el interior del método dispose de la clase principal del juego, que se ejecutará únicamente como respuesta a la señal producida por una llamada a Gdx.app.exit, garantizando la limpieza de los recursos empleados por el puente con el lector de pantalla siempre y cuando el proyecto se cierre en circunstancias normales.

Alternativas exploradas

Procedamos a explicar los motivos por los que se optó por utilizar Tolk, explorando, para ello, las diversas alternativas exploradas. En todos los casos, se indica el motivo por el que la opción se descartó en favor de la librería seleccionada, de modo que resulte posible seguir la lógica aplicada para la toma de tal decisión.

La primera alternativa planteada fue el uso de Java Accessibility API, or JAAPI para abreviar. Se trata de una API provista de forma nativa por Oracle junto al lenguaje de programación Java para dotar de acceso a funcionalidades de lectores de pantalla compatibles con Java Access Bridge, tales como audiodescripción o acceso a pantallas braille. En otras palabras, el planteamiento es similar al realizado por Tolk, con la salvedad de no depender de la presencia de DLLs específicas en el entorno de ejecución. Además, dado que Java Access Bridge es una tecnología exclusiva de Windows, el rango de usuarios que

podría beneficiarse del uso de esta opción es similar al que podría aprovechar el uso de Tolk. No obstante, una lectura en mayor profundidad de la documentación de JAAPI revela un detalle fundamental: la librería está diseñada para su uso específico en interfaces generadas mediante las librerías gráficas nativas de Java, tales como Swing, y requieren escribir código adicional para compatibilizar componentes personalizados. Lo que es más, no permite solicitar al lector de pantalla la producción de una salida desde el propio programa. El resultado es, por consiguiente, una opción inferior para las necesidades del proyecto, al no ofrecer ninguna ventaja, y suponer un aumento del coste de producción, agravado por la pérdida de características fundamentales.

La segunda alternativa planteada fue la creación de una fachada personalizada que hiciese de interfaz con los diversos lectores de pantalla presentes en el mercado, utilizando para ello las correspondientes APIs, y un selector capaz de detectar el lector de pantalla en ejecución de manera automática durante el arranque. El funcionamiento resultante sería muy similar al que ya ofrece Tolk, con la salvedad de permitir mayor diversidad de lectores de pantalla al poder implementar módulos específicos para cada uno de ellos, facilitando, una vez completada, la elaboración de proyectos accesibles multiplataforma mediante el uso de LibGDX. Los costes de implementación en términos de tiempo suponían, no obstante, la necesidad de enfocarse a un único lector de pantalla para el propósito del desarrollo de este TFG, contrarrestando inmediatamente cualquier ganancia que la versión completa pudiese haber ofrecido.

Cualquier otra opción sería, en realidad, una variante de una de estas alternativas, o de la opción escogida. Así pues, dada la amplia compatibilidad, facilidad de integración y magnífico resultado ofrecido por Tolk, se escogió esta última librería para la elaboración del prototipo.

Propuestas para la ampliación del módulo de accesibilidad

Para poder salvar los problemas ocasionados por la complejidad inherente a dotar de accesibilidad a un proyecto multiplataforma de estas características, se propone la siguiente aproximación. Cabe destacar que el único motivo por el que no ha sido empleada para la elaboración de este TFG han sido las limitaciones de tiempo, unidas al extenso trabajo necesario para su correcta implementación y depuración.

LibGDX está diseñado con la posibilidad de generar código específico de cada plataforma de destino y utilizarlo en el fuente principal en mente, dado el objetivo de permitir un desarrollo ágil de videojuegos multiplataforma. Para ello, pone a disposición de cada subproyecto correspondiente a una plataforma de despliegue el código generado en el subproyecto core, y permite la libre modificación del constructor para la clase principal de juego del mismo, de modo que sea posible requerir parámetros específicos que no formen parte del listado previsto por BadLogicGames. Esta decisión de diseño puede emplearse para generar una interfaz común, una implementación de la cual será requerida por el constructor del juego, y cuyos detalles específicos dependerán de la plataforma en que se esté ejecutando. Este proceso es, de hecho, común a la hora de generar interfaces y esquemas de control adaptados a diversas plataformas, al abstraer los detalles de los mismos de la lógica de negocio principal.

Esta misma idea puede fácilmente emplearse para ocultar los detalles de la implementación de la capa de accesibilidad de la lógica de negocio, estableciendo lógica específica para cada plataforma de destino. Así, por ejemplo, se podrá acceder a un módulo compatible con TalkBack al realizar el despliegue de un proyecto en Android, y compatible con Tolk al desplegar el mismo proyecto en Windows.

Es necesario, no obstante, tener en cuenta una consideración adicional al tratar con esta idea: el proyecto desktop está destinado a cualquier ordenador personal o portátil, con independencia del sistema operativo ejecutado, de modo que se habrá de añadir una capa adicional a la implementación de la interfaz para garantizar la ejecución de las funciones correctas. Existen dos posibilidades fundamentales:

1. Generar una única implementación de la interfaz correspondiente, y delegar la responsabilidad sobre un componente destinado a cada plataforma de destino, que será seleccionado durante la instanciación del módulo correspondiente, haciendo el proceso transparente al resto del sistema.

2. Generar una implementación de la interfaz para cada una de las posibles familias de sistemas operativos de destino, y seleccionar la más adecuada en tiempo de ejecución, utilizándola a continuación en core.

Ambas aproximaciones parecen suponer un esfuerzo similar en términos del desarrollo, y ninguna parece ofrecer ventajas significativas sobre la otra. Para determinar los detalles prácticos sería necesario realizar un estudio en mayor profundidad de ambas ideas, contemplando en el proceso el impacto sobre la legibilidad, mantenibilidad y extensibilidad del proyecto de ambas opciones.

4.4.7 Implementación del sistema de menús

La implementación del sistema de menús se estructura en torno a dos ejes fundamentales: el primero es la reutilización de conceptos modulares, y el segundo, la accesibilidad de dichos elementos. A excepción de algunos aspectos auxiliares de las interfaces de exploración e inventario, y del sistema de mensajes, todos los menús heredan de la clase MenuBox. Esta clase abstracta define la estructura por componentes seguida por todas sus subclases, concretamente la presencia de huecos habilitados para insertar opciones de manera modular, la existencia de opciones que den funcionalidad a dichos huecos, y la presencia de un cursor, que puede encontrarse oculto.

La estructura de la mayoría de los menús en una jerarquía de componentes tiene una única función: facilitar la personalización, y la generación de nuevas interfaces. Los huecos para opciones definen la posición de la pantalla donde la opción se representará, y contienen un puntero a la opción correspondiente. Del mismo modo, las opciones implementan un patrón comando para contener su funcionalidad, permitiendo insertar, literalmente, funcionalidades en los menús de manera transparente; así, si existe un comando encargado de abrir el menú de inventario, dicho comando se podrá emplear en múltiples menús, sin necesidad de realizar modificaciones al mismo, simplemente asignándolo a la correspondiente opción. Asimismo, las opciones del menú contienen una etiqueta, que en un caso general se emplea tanto para determinar el texto representado en el menú, como para alimentar al lector de pantalla con información que transmitir al jugador.

Todos los menús han sido diseñados con una estructura vertical, y permiten al cursor rotar de un extremo al otro según sea necesario, para facilitar el manejo de los mismos. Cabe destacar que el cuadro de información del grupo está implementado como un menú, para facilitar su interacción tanto con determinados subsistemas como con la capa de accesibilidad.

En cuanto a los elementos que se alejan de la aproximación general, podemos dividirlos en dos grandes grupos: accesibles y no accesibles. Todos los elementos esenciales forman parte del grupo de elementos accesibles, poseyendo siempre una representación alternativa utilizada por el lector de pantalla, y los elementos no accesibles son meras ayudas visuales que permiten a determinados jugadores facilitar su exploración.

Entre los elementos no estándar accesibles encontramos:

- El sistema de mensajes, que se explica más adelante.

- El indicador de orientación del sistema de exploración del laberinto, que se explica más adelante.

- El viewport en el que se representa el laberinto, mediante el uso de efectos de sonidos complementarios para suplir la falta de información visual para usuarios con ceguera total.
- La caja de descripción de objetos en el menú de inventario.
- El selector de enemigos, que ofrece una representación alternativa al correspondiente sprite mediante el nombre del mismo.

Entre los elementos no estándar no accesibles encontramos:

- El minimapa del sistema de exploración del laberinto.
- El indicador de coordenadas del sistema de exploración del laberinto.

4.4.8 Implementación del sistema de mensajes

El sistema de mensajes utiliza un buffer para almacenar todos los textos que han de mostrarse secuencialmente como resultado de una interacción o evento, y genera un cuadro sobre el que mostrarlos. El cuadro ocupa el total del ancho de la pantalla, y está diseñado, como el conjunto del juego, en blanco y negro para mejorar el contraste.

Con el objetivo de minimizar los sobresaltos producidos de manera instintiva al ver aparecer grandes elementos de forma repentina en pantalla, se han implementado dos sistemas de suavizado: uno referente a la caja en la que se representan los mensajes, y otro referente a la representación gradual de los propios mensajes.

El proceso de apertura de la caja de mensajes sigue la siguiente lógica:

Algoritmo 4.1: Proceso de apertura de la caja de diálogo.

```
Result: Caja de diálogo abierta de forma gradual
1 var tiempo = 0.5f;
2 var delta_w = width/tiempo;
3 var delta_h = height/tiempo;
4 while actual_width != width and actual_height != height do
5 |   width = width + delta_w;
6 |   height = height + delta_h;
7 end
```

Si bien la implementación exacta realizada en Java es ligeramente distinta, la filosofía de implementación es idéntica.

La verdadera lógica implementada en Java se muestra a continuación, con el único objetivo de permitir la comparación entre el planteamiento algorítmico inicial, y la implementación final.

```
width += targetWidth/(FPS*targetTime);
height += targetHeight/(FPS*targetTime);
if(width > targetWidth) width = targetWidth;
if(height > targetHeight) height = targetHeight;
createBox();
if(isReady()) this.state = MsgBoxState.SHOWING_TEXT;
```

Nótese que, dado que forma parte del método update, se utiliza el ratio de frames por segundo, junto al tiempo de apertura objetivo, para suavizar el proceso de actualización de la caja. Una vez se completa

este proceso, se realizan dos comprobaciones de control para ajustar los parámetros de altura y anchura en caso de haberse superado los valores deseados. A continuación, se prepara la caja para su dibujo con los nuevos valores. Finalmente, si se han alcanzado las dimensiones deseadas, la caja se prepara para mostrar mensajes.

Cabe destacar que el mismo proceso se realiza, a la inversa, para cerrar de forma gradual la caja de diálogo, y de ese modo evitar sobresaltos al jugador por su repentina desaparición.

En cuanto a la representación gradual del texto, simplemente se emplea una variable auxiliar, cuyo valor se actualiza de manera progresiva, para indicar el número de caracteres a representar. De este modo, en cada actualización aumenta la cantidad de caracteres en pantalla, y se crea la ilusión de que el texto se está representando de forma gradual.

Respecto al uso del buffer de texto, llamado messagePipeline, se trata de una cola de tipo String, de la que se va extrayendo gradualmente el primer elemento hasta que está vacía. Cuando un mensaje se está representando en su totalidad, si quedan mensajes en cola, aparece un pequeño indicador para invitar al jugador a seguir leyendo, y el proceso se repite hasta que no queda ningún mensaje por mostrar, momento en el que se cierra la caja de diálogo y se regresa al estado anterior.

4.4.9 Implementación del sistema de control

En condiciones normales, para garantizar la accesibilidad del prototipo desarrollado, sería necesario permitir la personalización del sistema de control. Sin embargo, dado las características de LibGDX, este proceso requiere el desarrollo de múltiples abstracciones que permitan diversificar la entrada del procesado de la misma desde un punto de vista de máquina de estados. Es por ello, junto al hecho de que, dadas las condiciones establecidas anteriormente, el prototipo solo es completamente compatible con sistemas Windows, que se ha optado por un compromiso intermedio, utilizando un esquema de control sencillo, y dejando las abstracciones necesarias como trabajo futuro, que requerirá de una cierta refactorización de código. La distribución realizada es la siguiente:

- Flecha arriba:
    1. Desplazarse hacia delante en el laberinto.
    2. Desplazar el cursor hacia arriba en menús.
- Flecha abajo:
    1. Desplazarse hacia detrás en el laberinto.
    2. Desplazar el cursor hacia abajo en menús.
- Flecha izquierda:
    1. Girarse hacia la izquierda en el laberinto.
    2. Cambiar al anterior personaje del grupo en submenús que lo permitan.
- Flecha derecha:
    1. Girarse hacia la derecha en el laberinto.
    2. Cambiar al siguiente personaje del grupo en submenús que lo permitan.
- Tecla Z:
    1. Acceder al menú de campo durante la exploración.

2. Confirmar una opción seleccionada en un menú.

- Tecla X:

1. Accionar un evento durante la exploración.

2. Regresar a la pantalla anterior en el sistema de menús.

El acceso a estos controles se realiza, actualmente, de forma directa, mediante los métodos de lectura que ofrece LibGDX de forma nativa. Para poder realizar la abstracción, se generaría un archivo de configuración, y un mapa de símbolos a constantes de LibGDX, que permitiese leer el estado de mediante múltiples niveles de indirección, y crease la posibilidad de configurar el juego al gusto del usuario.

4.4.10 Implementación de la exploración

El sistema de exploración es uno de los elementos fundamentales del videojuego desarrollado. Gran parte del atractivo del género, para sus seguidores más fieles, es, precisamente, la exploración y descubrimiento de entornos hostiles en los que, por norma general, mayor riesgo implica mayor recompensa. Resulta razonable, por tanto, prestar atención a este apartado del prototipo, especialmente dado el calibre del reto presentado por el mismo desde el punto de vista de la accesibilidad.

La exploración del laberinto se realiza desde una perspectiva en primera persona, en la que se representa el entorno a su alrededor en una perspectiva cónica de un punto de fuga, centrado en la pantalla. La representación del entorno percibido por el grupo se calcula en tiempo de ejecución, utilizando para ello la información referente a cada baldosa visible desde el punto en el que está situado en ese momento, así como la dirección en que está mirando el jugador. El algorimo empleado para la representación del algoritmo es el siguiente:

Algoritmo 4.2: Algoritmo de representación gráfica del laberinto

Result: Representación gráfica del laberinto

```
1  var x = jugador.x;
2  var y = jugador.y;
3  var dirección = jugador.dirección for cada baldosa en la distancia de visión do
4  |  var escalado = 1/(distancia + 1);
5  |  if muros a la izquierda en dirección then
6  |  |  dibujar muros a la izquierda con escalado y distancia;
7  |  end
8  |  if muros a la derecha en dirección then
9  |  |  dibujar muros a la derecha con escalado y distancia;
10 |  end
11 |  if muros al frente en dirección then
12 |  |  dibujar muros al frente con escalado y distancia;
13 |  end
14 end
```

En cuanto al dibujo de los muros, existen dos variantes principales: si el muro está a izquierda o

derecha, se dibuja un trapecio, y si está al frente, un cuadrado. El proceso es el siguiente:

Algoritmo 4.3: Algoritmo de representación gráfica de un muro a izquierda o derecha

Result: Representación de un muro a izquierda o derecha

```
1  var w = calcular anchura del muro;
2  var h0 = calcular altura cercana del muro;
3  var h1 = calcular altura lejana del muro;
4  var x0 = calcular desplazamiento horizontal en base a distancia;
5  var x1 = x0 + w;
6  var y0 = calcular desplazamiento vertical en base a distancia;
7  var y1 = y0 + (h0 - h1)/2;
8  var dirección = jugador.dirección;
9  calcular vértices usando las variables generadas;
10 if los vértices colisionan then
11 |  ajustar vértices;
12 end
13 dibujar paralelogramo formado por los vértices calculados;
14 registrar región de la pantalla como ocupada;
```

Algoritmo 4.4: Algoritmo de representación gráfica de un muro al frente

Result: Representación de un muro al frente

```
1  var x0 = calcular x de origen en función de distancia;
2  var w = w0 * escalado;
3  var x1 = x0 + w;
4  var y0 = calcular y de origen en función de distancia;
5  var h = h0 * escalado;
6  var y1 = y0 + h;
7  generar cuadrado con vértices basados en x0, x1, y0 e y1;
8  if los vértices colisionan then
9  |  ajustar vértices;
10 end
11 dibujar paralelogramo formado por los vértices calculados;
12 registrar región de la pantalla como ocupada;
```

De este modo, la visión del jugador se calcula en tiempo de ejecución, en cada llamada al método render del estado de exploración. Los cálculos de visibilidad en pantalla tienen en cuenta la presencia de otros elementos de interfaz, así como los límites del viewport utilizado para representar la visión del jugador.

Además del laberinto desde una perspectiva en primera persona, mientras el jugador está explorando también se representa un minimapa desde una perspectiva cenital, con unas dimensiones de 3x3 baldosas, el estado del grupo, e información referente a la posición y orientación del mismo dentro del mapeado. De entre todos estos elementos, el único accesible de forma directa mediante lectores de pantalla es la orientación del grupo, cuyo estado se indica al jugador cada vez que se gira, con el objetivo de evitar potenciales efectos de cacofonía que puedan suponer una pérdida de información relevante.

Para evitar movimientos accidentales, la lectura de la entrada en este estado tienen un breve período de refresco de 10 frames, esto es, una sexta parte de segundo, de modo que exista un mínimo tiempo de reacción para el jugador sin por ello comprometer la agilidad de la jugabilidad.

Cada paso tomado por el jugador tiene dos efectos: en primer lugar, se comprueba la presencia de eventos activables en la casilla de destino y, en caso de haberlos, se determina qué hacer con ellos; en segundo lugar, se ajusta el contador oculto de pasos que gestiona la activación de una batalla aleatoria. Si el contador de pasos alcanza un valor establecido aleatoriamente cada vez que se entra al laberinto, o que se completa un combate aleatorio, se produce un encuentro de manera automática.

El movimiento se realiza mediante las teclas de dirección: las flechas izquierda y derecha actualizan la orientación del jugador, mediante un sencillo proceso en el que se identifica la orientación actual, se determina la dirección del giro y la orientación resultante, y se actualiza la información del grupo. Dado que el dibujo del laberinto utiliza la información contenida en dicha estructura de datos para determinar qué parte del laberinto representar, y como, el proceso es totalmente transparente de cara al dibujado. Asimismo, las teclas arriba y abajo permiten realizar el desplazamiento propiamente dicho: presionar la tecla arriba permite desplazarse hacia delante, mientras que la tecla abajo hace que el grupo retroceda. El proceso es algo más complejo en este caso: en primer lugar, se determina la dirección del movimiento, y se calcula si se producirían colisiones utilizando la información almacenada en las banderas de las casillas; en caso de que no se produzca colisión alguna, se actualiza la información de la posición del jugador de la forma pertinente, lo que, por supuesto, se traduce en una modificación de la información representada en pantalla; por contra, si el movimiento no pudiese realizarse, se reproduce el correspondiente sonido de colisión y se mantiene la posición del jugador.

Para interactuar con el juego en este estado, se utilizan dos botones más: la tecla Z abre el menú de campo, mientras que la tecla X permite interactuar con eventos cuya activación no sea automática.

4.4.11 Implementación del sistema de combate

El sistema de combate se estructura en torno a su propia máquina de estados interna. De cara al juego, todos ellos se identifican como parte del estado de combate, y es el propio estado de combate el encargado de gestionarlos. Esta máquina de estados está implementada de una forma mucho más ortodoxa que la máquina de estados general del juego: un mismo estado puede transicionar a varios otros estados dependiendo de las interacciones que se produzcan, y este proceso suele ser unívoco.

En general, el flujo sigue un bucle constante: una vez comenzado el combate, el jugador selecciona las acciones de sus personajes para el turno, los enemigos seleccionan sus propias acciones, las acciones se ordenan en base a su prioridad de ejecución, y se procede a resolverlas. Si el combate no ha terminado todavía, se vuelve al comienzo del bucle. En caso de haber concluído, se determina el resultado del mismo, con tres posibilidades:

- Si los enemigos han sido derrotados, los personajes del jugador reciben como recompensa la correspondiente experiencia y dinero.
- Si los personajes han huido del combate, vuelven a la exploración sin recibir recompensa alguna.
- Si todos los personajes han muerto, el juego pasa al estado de Game Over.

Del mismo modo, el proceso de inicialización del estado de combate pasa por múltiples fases:

1. Se determina el número de enemigos en combate, como mínimo uno, y como máximo el valor establecido en el mapa correspondiente.
2. Se determinan al azar los enemigos que aparecen en combate, utilizando para ello la lista de posibles enemigos presente en la configuración del mapa.

3. Se instancian los enemigos, y asignan a posiciones específicas dentro de la pantalla.

4. Se genera el mensaje de comienzo de combate.

5. Se presentan el estado de combate, y el correspondiente mensaje.

A partir de aquí, comienza el bucle habitual de ejecución de este estado, hasta que el combate sea concluido de una de las formas anteriormente indicadas. Los enemigos y las acciones quedan almacenados en una lista y una cola de prioridad respectivamente, de modo que sean accesibles para múltiples subestados de manera independiente.

El proceso es de selección de acciones es, en si misma, importante a la hora de entender el flujo interno de información. Concretamente, los pasos seguidos, con independencia de la entidad cuya acción se está seleccionando, son los siguientes:

1. Se selecciona la acción a realizar.

2. Se selecciona el objetivo de la acción, en caso de ser necesario.

3. Se encapsula la combinación de acción, entidad que la realiza y entidad objetivo.

4. Se añade la acción encapsulada a la cola de acciones.

5. La cola se reorganiza automáticamente utilizando la prioridad real de cada acción a ejecutar como criterio.

Concretamente, la prioridad real de una acción se calcula mediante la siguiente fórmula:

$$p = src.Agility * skill.priority \tag{4.10}$$

Donde p es la prioridad real de la acción, *src.Agility* es el atributo de agilidad de la entidad que la ejecuta, y *skill.priority* es el factor de prioridad de la habilidad. Un mayor valor de prioridad efectiva supone que la acción se resuelva antes, de modo que los personajes con mayor agilidad, así como las habilidades con prioridad elevada, tienden a mover primero.

Finalmente, la resolución de acciones sigue su propio proceso:

1. Se recuperan la habilidad a ejecutar, así como las entidades involucradas.

2. Se solicita a la instancia única de la habilidad que realice las modificaciones pertinentes a ambas entidades, y que genere el mensaje de resolución correspondiente.

3. Se transiciona al estado de mensajes en la máquina de estados global del juego, almacenando en su buffer los mensajes generados por la resolución de la correspondiente habilidad.

Cabe destacar que las acciones solo se resuelven en caso de que las entidades involucradas sigan vivas. En caso de que al menos una de ellas haya muerto, la ejecución se cancela, y,de tratarse únicamente del objetivo, la entidad que ejecuta la habilidad pierde automáticamente el turno. Esta mecánica es una decisión de diseño intencional, dado que la búsqueda de un nuevo objetivo potencial está implementada como parte del sistema de batalla, dad la necesidad de evitar que el jugador pudiese seleccionar enemigos muertos como objetivos de un ataque.

4.5 Correspondencia con las *Game Accessibility Guidelines*

A continuación, se analiza la correspondencia entre el trabajo realizado y las *Game Accessibilty Guidelines*[20]. Se ha estudiado únicamente la correspondencia con aquellas pautas referidas a un perfil de diversidad funcional visual, por ser aquellas en las que se centra el proyecto. Asimismo, se han omitido aquellas pautas que no correspondan a juegos en dos dimensiones.

- Garantizar que ninguna información esencial se transmita mediante color únicamente.
  - Cumplida en su totalidad. El juego no depende en absoluto de colores para transmitir información esencial.
- Utilizar un tamaño de fuente fácil de leer.
  - Cumplida en su totalidad. El tamaño de fuente se ha especificado para ocupar el espacio disponible y resultar sencillo de ver.
- Utilizar un formato de texto simple.
  - Cumplida en su totalidad. La información textual se presenta siempre de la forma más sencilla posible.
- Utilizar un alto nivel de contraste entre el texto de la interfaz y el fondo.
  - Cumplida en su totalidad. La interfaz fue diseñada desde un primer momento para garantizar el máximo nivel de contraste.
- Utilizar sonido envolvente.
  - Pendiente. Dadas las limitaciones de LibGDX, conseguir un sonido envolvente apropiada al tiempo que se maneja un lector de pantalla suponía un esfuerzo adicional que, por desgracia, no pudo realizarse.
- Garantizar compatibilidad con lectores de pantalla.
  - Cumplida parcialmente. El prototipo es compatible con una pequeña lista de lectores de pantalla comerciales, pero dicha lista debería verse ampliada.
- Garantizar que los efectos de sonido de opciones esencialmente diferentes son distintos.
  - Cumplida parcialmente. Gran parte de los efectos de sonido empleados son unívocos, pero sería conveniente ampliar la gama.
- Ofrecer una opción para ajustar el nivel de contraste
  - Descartada. La ausencia de un menú de configuración imposibilitaba usar esta pauta, pero el diseño visual del juego está orientado a anular cualquier problema derivado del nivel de contraste.
- Ofrecer una indicación clara de que los elementos interactivos son interactivos.
  - Cumplida parcialmente. Notablemente, los efectos activables reproducen un efecto de sonido para indicar su presencia.
- Evitar situar información esencial fuera de la línea de visión del jugador.

  - Cumplida en su totalidad. Los elementos fundamentales se encuentran siempre situados en el centro de la pantalla, o de tal manera que no haya distracciones.

- Permitir el redimensionamiento de la interfaz.
  - Pendiente. Parte del trabajo futuro.
- Permitir el ajuste del tamaño de fuente.
  - Pendiente. Parte del trabajo futuro.
- Ofrecer un mapa de audio estilo sónar.
  - Cumplida parcialmente. El juego utiliza sonidos específicos para ayudar al jugador a orientarse, pero el sistema puede refinarse en gran medida.
- Ofrecer audio pregrabado para todos los textos, incluyendo menús e instaladores.
  - Descartada, en favor de la compatibilidad con lectores de pantalla.
- Usar audio distintivo para cada escenario, evento y situación.
  - Pendiente. Parte del trabajo futuro.
- Simular sonido binaural.
  - Pendiente. Parte del trabajo futuro.
- Ofrecer una pista de audiodescripción.
  - Pendiente. Parte del trabajo futuro. Potencialmente no aplicable.

Capítulo 5

Coste del proyecto

Se presenta, a continuación, una estimación del coste de ejecución del proyecto. En este coste se contemplan el coste material y el coste de personal, pero no se evalúan aquellos materiales que forman parte del funcionamiento habitual de la empresa y que, por consiguiente, no necesitarían haber sido adquiridos específicamente para desarrollar este proyecto.

5.1 Coste material

Se considera que el coste material es la combinación del coste de equipos y el coste de personal involucrados en el proyecto. Dado que solamente se ha empleado un portátil, propiedad anterior de la empresa y parte normal del funcionamiento de la misma, el coste de equipo se considera cero, por ser un coste de funcionamiento de la propia empresa, y no estar directamente vinculado al desempeño del proyecto.

5.1.1 Coste por tiempo de trabajo

El desarrollo del proyecto ha llevado un tiempo total estimado de 480 horas, considerando una jornada laboral de 8 horas al día. El salario estimado del ingeniero involucrado es de 30 euros la hora, por lo que el coste total por personal queda desglosado a continuación:

FUNCIÓN	**N° DE HORAS**	**€/HORA**	**TOTAL**
Ingeniería	480	30	14.400€

Dadas las condiciones especificadas, esta cantidad se corresponde también con el coste de ejecución material del proyecto.

5.1.2 Gastos generales y beneficio industrial

Con el objetivo de cubrir los gastos de mantenimiento de las instalaciones, así como otros costes incurridos durante el funcionamiento normal de la empresa, se aplica un margen del 20 % sobre el coste por tiempo de trabajo, obteniendo la siguiente estimación:

Gastos generales y beneficio industrial	2.880€

5.1.3 Presupuesto de ejecución por contrata

El valor del presupuesto de ejecución por contrata resulta de la suma de los costes de ejecución materia, y la combinación de gastos generales y beneficio industrial. Así pues, el presupuesto de ejecución por contrata será:

Presupuesto de ejecución por contrata	17.280€

5.1.4 Importe total

Al presupuesto de ejecución por contrata ha de aplicársele un IVA del 21 %, de lo que obtenemos un coste total de:

CONCEPTO	VALOR
Presupuesto de ejecución por contrata	17.280€
Impuesto sobre valor añadido	3.628,8€
IMPORTE TOTAL	20.908,8€

Por tanto, el coste total del proyecto asciende a 20.908,8 euros.

Capítulo 6

Conclusiones y líneas futuras

A continuación, se ofrece un resumen del trabajo realizado, acompañado de las condiciones resultantes, y se presentan las líneas de trabajo futuras para este TFG.

6.1 Resumen

Este trabajo de fin de grado pretendía la elaboración de un prototipo de videojuego RPG, mediante el uso del motor LibGDX, que fuese accesible en el mayor grado posible para personas con diversidad funcional visual. Para ello, se debían estudiar los distintos perfiles de usuario potencial que comprendían tal sector de población, plantear posibles formas de facilitar su acceso a la información necesaria para disfrutar del videojuego desarrollado, e implementarlas, en la medida de lo posible, en un proyecto funcional que sirviese de prototipo.

Una vez realizado el estudio previo, se ha decidido implementar una capa de accesibilidad mediante la librería Tolk, dado que, si bien ofrece soporte únicamente para sistemas Windows, es la opción que mayor variedad de perfiles de usuario ofrece de entre las estudiadas, y se ha procedido a construir un motor completo basado en LibGDX para la ejecución del juego. En este documento se han explicado las consideraciones involucradas y decisiones tomadas durante el desarrollo de dicho motor, incluyendo aspectos relacionados con estructuras de datos fundamentales, la implementación de la máquina de estados de juego, la implementación de las mecánicas fundamentales del mismo, la creación de una capa de scripting para facilitar la implementación de contenidos, la generación de un módulo para integrar mapas creados mediante Tiled, e, incluso, la implementación del sistema de guardado.

Se ha dedicado, también, una extensa sección a explicar y detallar las decisiones de diseño de juego que se han ido tomando, haciendo especial mención de aquellas que impactan la accesibilidad del producto resultante, y se han pormenorizado los detalles de diseño de juego de mayor relevancia, demostrando, en el proceso, que el coste del trabajo de diseño a desarrollar no se ve impactado, en gran medida, por la necesidad de crear un producto accesible, siempre y cuando se tenga en cuenta dicha necesidad desde un primer momento.

Finalmente, se ha cruzado el trabajo desarrollado con la sección sobre accesibilidad para personas con diversidad funcional visual de las *Game Accessibility Guidelines*, indicando si cada directiva se ha cumplido, si era de aplicación, y, en caso de haberse prescindido de ella, el motivo para hacerlo.

6.2 Conclusiones

El desarrollo del presente TFG ha evidenciado numerosos aspectos referentes al desarrollo de videojuegos accesibles, aunque todos ellos han de matizarse dentro de un contexto determinado. Concretamente, muchas de las facilidades encontradas para el desarrollo del trabajo se deben a la selección específica de género para el prototipo: la independencia de factores temporales, la simplicidad de la representación de los mapeados y la clara dependencia en información textual son todos factores que han facilitado la labor de diseño e implementación del trabajo realizado y, en algunos casos, auspiciado la consecución de determinadas metas que, en otras circunstancias, podrían haber resultado del todo inviables.

Uno de los hechos más relevantes que han quedado demostrados durante la elaboración del prototipo aquí detallado es la influencia del factor tiempo en la integración de la accesibilidad. Cuanto más avanzado se encuentra el proyecto, más difícil resulta integrar las correspondientes pautas, mientras que la implementación de muchas de ellas se ve trivializada siempre y cuando se contemple desde un primer momento. Asimismo, es importante recalcar, una vez más, el impacto de la complejidad de las mecánicas de juego elegidas: la integración de lectores de pantalla ha resultado posible, únicamente, gracias a la dependencia directa que el proyecto presentaba del texto escrito. Del mismo modo, habría sido necesario dilucidar mecanismos más ágiles para la transmisión de información fundamental en caso de haber prescendido de un sistema de combate por turnos, en favor de un sistema de batalla en tiempo real o en tiempo activo. En otras palabras, toda decisión de diseño puede tener un impacto directo en la implementación de la accesibilidad en un videojuego y, por ende, ha de ser meditada cuidadosamente antes de ser establecida de manera definitiva.

En referente a las herramientas manejadas, LibGDX se demuestra como un motor flexible, si bien requiere mayor esfuerzo por parte del programador que otros motores comerciales para obtener resultados similiares, y capaz de ofrecer resultados sólidos cuando se utiliza cuidadosamente. El hecho de estar implementado en Java, su compatibilidad con numerosas librerías propias de este lenguaje, y el uso activo de Gradle como sistema de compilación son todos bazas a favor de esta herramienta, al menos en el contexto de pequeños desarrollos con ambición limitada. Sin embargo, la necesidad de prácticamente construir un motor de juego a partir de lo que es, en esencia, un pequeño framework que ofrece preeminentemente funcionalidades de entrada y salida supone un riesgo digno de valorar antes de comenzar un proyecto apoyándose en él.

Merece la pena mencionar, también, la extensa documentación existente para facilitar el desarrollo mediante LibGDX, producto del esfuerzo tanto de la comunidad de desarrolladores que lo emplea, como de Badlogic Games, desarrolladores del motor. La encomiable labor realizada para facilitar la comprensión del motor, mediante ejemplos y explicaciones sencillas, pero completas, ha resultado fundamental en la consecución de este proyecto y, de hecho, en la integración exitosa tanto de LuaJ como de Tolk con la herramienta.

Como un último aparte, la existencia de la librería Tolk, y su gran compatibilidad con LibGDX, evidencia las posibilidades, en su mayoría todavía inexploradas, de la accesibilidad en videojuegos. Resulta significativo que títulos de gran calibre, como el Killer Instinct desarrollado por Double Helix Games y publicado en 2013 hayan servido como una fuente de inspiración para diseñar algunos de los mecanismos de accesibilidad aplicados en este proyecto. Es innegable que se trata de una labor compleja, y muy probablemente la visión artística de determinados proyectos entre en conflicto directo con las acciones necesarias para conseguir que el producto resultante sea completamente accesible, pero continuar esta línea de trabajo ofrece un beneficio potencial que no ha de ser ignorado.

6.3 Líneas futuras

El proyecto desarrollado es, simplemente, un prototipo, y ofrece, por tanto, numerosas opciones para su ampliación. A continuación se listan las más fundamentales:

- La extensión del módulo de accesibilidad para dar cabida a sistemas Linux, Mac, Android y iOS es, probablemente, la línea de trabajo futuro más importante. El esfuerzo necesario para llevarlo a cabo es notable, pero no por ello deja de merecer la pena.
- La refactorización y el refinamiento del motor desarrollado son, asimismo, fundamentales para cualquier trabajo futuro adicional. Muchas de las decisiones de implementación tomadas son fruto de la necesidad, y de un proceso de aprendizaje que sigue en curso. Mejorar los mecanismos ya establecidos en este TFG para el desarrollo de videojuegos es una meta evidente. Entre estas ampliaciones, se encuentra la inclusión, por ejemplo, de un sistema de equipamiento completo.
- La implementación de un sistema de configuración para que el usuario personalice su experiencia de juego es otra de las grandes líneas de trabajo futuro. Se tuvo que prescindir de la idea por motivos de tiempo, pero hubiese sido deseable llevarla acabo. Tal deseo permanece inalterado.
- La introducción de mayor variedad mecánica, tanto en lo que a enemigos como a habilidades se refiere, ampliando la progresión del jugador, y presentando mayor variedad de situaciones.
- La creación de más elementos interactivos dentro del laberinto, en su mayoría amenazas, que supongan el planteamiento de nuevos retos para el jugador. A la mente vienen clásicos del género, tales como laberintos de casillas teletransportadoras o secciones que causan daño al atravesarlas.
- La implementación de un sistema de misiones opcionales que amplíen la vida del juego.
- La integración de aquellos perfiles de diversidad funcional que han tenido que ser dados de lado en favor del perfil visual, fundamentalmente, el perfil de diversidad funcional auditiva. Para ello, sería necesario realizar un estudio en profundidad previo a la refactorización anteriormente contemplada.
- La creación de un juego completo, utilizando como base este prototipo, que integre las ideas básicas aquí recogidas, las líneas de trabajo futuras ya planteadas, y una línea argumental coherente y disfrutable, que entretenga al jugador durante horas.

La integración de estas ideas, y el desarrollo de un producto comercial, permitiría a su vez recabar datos reales, gracias a la colaboración de los usuarios, para continuar el estudio de la accesibilidad a este campo y, deseablemente, dotar de una oportunidad de disfrutar del medio a aquellos que a día de hoy no pueden. Es más, permitiría sentar un nuevo precedente, y demostraría que sí es posible lograr la accesibilidad en videojuegos, más allá de toda duda razonable.

Además, el conocimiento adquirido durante el desarrollo de este proyecto es de clara aplicación en otras áreas de la informática. Sería conveniente aprovecharlo para reevaluar soluciones existentes a determinados problemas de accesibilidad en interfaces y, potencialmente, ofrecer nuevas ideas que puedan suplir algunas de las carencias ya conocidas.

Finalmente, sería importante reevaluar las decisiones tomadas durante este desarrollo una vez adquirida mayor experiencia en el campo de la accesibilidad para, idealmente, mejorarlas, y crear un producto capaz de llegar a una variedad mayor de usuarios.

Bibliografía

[1] *Encuesta sobre discapacidades, autonomía personal y situaciones de dependencia.* Instituto Nacional de Estadística, http://www.ine.es/dyngs/INEbase/es/operacion.htm?c=Estadistica_C&cid=1254736176782&menu=resultados&secc=1254736194716&idp=1254735573175.

[2] F. Pepe, *The CRPG Book Project: Sharing the History of Computer Role-Playing Games*, https://crpgbook.files.wordpress.com/2018/02/crpg_book_1-0-1.pdf.

[3] *Guía de accesibilidad de aplicaciones móviles (apps).* Aguado Delgado, J & y Estrada Martínez, F. J., https://sid.usal.es/libros/discapacidad/27481/8-4-1/guia-de-accesibilidad-de-aplicaciones-moviles-apps.aspx.

[4] *Posibilidades de la realidad virtual para mejorar la accesibilidad en desarrollos realizados con Unity.* Sánchez-López, S., Aguado Delgado, J. & Gutiérrez Martínez, J.M., https://sid.usal.es/libros/discapacidad/27481/8-4-1/guia-de-accesibilidad-de-aplicaciones-moviles-apps.aspx.

[5] W3C, "Web content accessibility guidelines (wcag) 2.1," https://www.w3.org/TR/WCAG21/.

[6] Apple, "Accessibility on ios," https://developer.apple.com/accessibility/ios/.

[7] Square-Enix, "Patch 4.3 notes," https://na.finalfantasyxiv.com/lodestone/topics/detail/13e322580acb8e9861160a6e08ccabfaff09eeee.

[8] M. Zechner, "libgdx 1.0 released," https://www.badlogicgames.com/wordpress/?p=3412.

[9] ——, "Box2d," https://github.com/libgdx/libgdx/wiki/Box2d.

[10] ——, "libgdx 1.1.0 released," https://www.badlogicgames.com/wordpress/?p=3451.

[11] ——, "libgdx 1.2.0 released," https://www.badlogicgames.com/wordpress/?p=3461.

[12] ——, "libgdx 1.3.0 released," https://www.badlogicgames.com/wordpress/?p=3484.

[13] ——, "libgdx 1.4.0 released," https://www.badlogicgames.com/wordpress/?p=3533.

[14] ——, "libgdx 1.5.0 released," https://www.badlogicgames.com/wordpress/?p=3617.

[15] ——, "libgdx 1.6.0 released," https://www.badlogicgames.com/wordpress/?p=3682.

[16] ——, "libgdx 1.7.0 released," https://www.badlogicgames.com/wordpress/?p=3758.

[17] ——, "libgdx 1.8.0 released," https://www.badlogicgames.com/wordpress/?p=3870.

[18] ——, "libgdx/changes," https://www.badlogicgames.com/wordpress/?p=3412.

[19] B. Games, "Goals and features," https://libgdx.badlogicgames.com/features.html.

[20] "Game accessibility guidelines," http://gameaccessibilityguidelines.com/.

Apéndice A

Detalles de implementación

A.1 Clases

A continuación, se expone información sobre la implementación real de cada una de las clases, representando mediante tablas el contenido de las estructuras de datos de referencia que las mismas contienen. Nótese que la implementación de las clases se detalló en la sección sobre estructuras de datos, y que este fragmento pretende, únicamente, complementar la información dipuesta en la misma y en el diseño de juego, ofreciendo datos reales.

Paladín

Estadística	**Valor**
Max_HP	15
Max_MP	5
Strength	10
Defense	10
Agility	3
Wisdom	4

Tabla A.1: Valores de base del paladín

Estadística	Valor
Max_HP	12
Max_MP	6
Strength	8
Defense	4
Agility	8
Wisdom	10

Tabla A.2: Modificadores del paladín

Habilidad	Nivel
Protect	2
Cure	5
Flash	8
Smite	10

Tabla A.3: Habilidades del paladín

Guerrero

Estadística	Valor
Max_HP	12
Max_MP	0
Strength	15
Defense	8
Agility	5
Wisdom	2

Tabla A.4: Valores de base del guerrero

Estadística	Valor
Max_HP	10
Max_MP	4
Strength	12
Defense	4
Agility	8
Wisdom	4

Tabla A.5: Modificadores del guerrero

Habilidad	Nivel
Cleave	4

Tabla A.6: Habilidades del guerrero

Clérigo

Estadística	Valor
Max_HP	10
Max_MP	6
Strength	5
Defense	8
Agility	4
Wisdom	15

Tabla A.7: Valores de base del clérigo

Estadística	Valor
Max_HP	8
Max_MP	6
Strength	5
Defense	6
Agility	4
Wisdom	10

Tabla A.8: Modificadores del clérigo

Habilidad	Nivel
Cure	2

Tabla A.9: Habilidades del clérigo

Mago

Estadística	Valor
Max_HP	4
Max_MP	10
Strength	2
Defense	5
Agility	5
Wisdom	10

Tabla A.10: Valores de base del mago

Estadística	Valor
Max_HP	5
Max_MP	12
Strength	3
Defense	3
Agility	8
Wisdom	12

Tabla A.11: Modificadores del mago

Habilidad	Nivel
Zap	2
Fireball	5
Flare	11

Tabla A.12: Habilidades del mago

Pícaro

Estadística	Valor
Max_HP	7
Max_MP	0
Strength	7
Defense	7
Agility	12
Wisdom	4

Tabla A.13: Valores de base del pícaro

Estadística	Valor
Max_HP	7
Max_MP	4
Strength	7
Defense	6
Agility	12
Wisdom	10

Tabla A.14: Modificadores del pícaro

Habilidad	Nivel
Assassinate	10

Tabla A.15: Habilidades del pícaro

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá