Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Desarrollo de un juego roguelike adaptado para invidentes empleando técnicas de generación de texto.

Estudante: Viteri Letamendía, Jorge
Dirección: Gómez Rodríguez, Carlos
Dirección: Vilares Ferro, Jesús

A Coruña, setembro de 2019.

A mi familia.

Agradecimientos

Muchas gracias a mi familia y amigos por todo su apoyo durante la carrera.
Muchas gracias a Jesús Vilares Ferro y a Carlos Gómez Rodrígues por sus consejos y recomendaciones con las que he podido avanzar el proyecto.
Y muchas gracias al café de Lázaro y al mixto de Ori, los añoraré.

Resumen

Este proyecto trata del desarrollo de un juego del género *roguelike* accesible para las personas con algún tipo de deficiencia visual grave. El juego conserva las características clásicas de un *roguelike*: se desarrolla por turnos en una mazmorra y el héroe o heroína deberá avanzar por esta superando los obstáculos que se le presenten. También sigue presente la *permadeath* o muerte permanente lo que implica que se perderá todo avance realizado una vez el protagonista muera. Además, la mazmorra será generada pseudo-aleatoriamente, dando al juego un mayor grado de rejugabilidad. Finalmente, se tiene un entorno gráfico *ASCII* para el juego y dos áreas de texto en las que se mostrará información sobre la partida, y que además son compatibles con lectores de pantalla. Al disponer de ambas representaciones, gráfica y textual, se permite que tanto jugadores videntes como invidentes puedan disfrutar de la experiencia de juego.

Abstract

This project is about the development of a *roguelike* game accessible to people with some kind of serious visual impairment. The game retains the classic characteristics of a *roguelike*: it takes place in a dungeon in which the progress is made in turns and the hero or heroine must advance through it overcoming obstacles that arise. The *permadeath* or permanent death is also present, which implies that any advance made will be lost once the protagonist dies. In addition, the dungeon will be generated pseudo-randomly giving the game a greater degree of replayability. Finally, there is a graphical environment textit ASCII for the game and two areas of text in which information about the game will be displayed, and which are also compatible with screen readers. By having both graphic and textual representations, both sighted and blind players are allowed to enjoy the gaming experience.

Palabras clave:

- Accesibilidad.
- Roguelike.
- Generación Automática del Lenguaje.
- Java.
- WordNet.
- Jugadores invidentes.
- Generación procedural.

Keywords:

- Accessibility.
- Roguelike.

- Automatic Language Generation.
- Java.
- WordNet.
- Blind players.
- Procedural generation.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introdución

1.1 Motivación

Siempre he tenido mucho interés por lo videojuegos, especialmente por los de rol y es la razón principal de que comenzase esta carrera. El medio de los videojuegos es enorme y variado, ofreciendo experiencias únicas y complejas creadas por la colaboración de multiples doctrinas. Esta colaboración es aún más notable dentro del aspecto de informática de un videojuego, ya que se pueden definir características en el videojuego que requieran el trabajo de ingenieros de diferentes campos de la informática, como redes e inteligencia artificial. En resumen, en mi opinión es el tipo de proyecto informático más completo y más interesante.

Tengo deseos de realizar varios trabajos en la industria que todos puedan disfrutar. De este modo, también me resulta importante mejorar la accesibilidad en esta industria que no para de crecer. A medida que pasa el tiempo, los videojuegos van formando más parte de la vida diaria para personas de todas las edades y género. Esta expansión no debería verse restrigida por algunas limitaciones que no han sido consideradas durante el desarrollo, por ello considero también muy importante el diseño orientado a la accesibilidad. Esto puede ir desde ofrecer diversos niveles de dificultad a configuraciones de la interfaz.

1.2 Objetivos

El objetivo del proyecto es el desarrollo de un videojuego del género *roguelike* que sea accesible tanto para jugadores habituales de este género como aquellos que tengan algún tipo de deficiencia visual grave. Además, este juego contaría con una interaz cómoda, un generador te textos y que esté internacionalizado. Con un carácter más específico, se considerarían como objetivos básicos:

- *Desarrollo del juego.* Ofrecer a los usuarios un *roguelike* divertido que no sacrifique ninguna de las características clave que definen al género.

- *Definir mecanismos para la obtención del vocabulario.* El vocabulario debería estar disponible en múltiples idiomas y estructurado de tal modo que sea fácil de tratar por el programa y que le aporte información importante sobre las palabras solicitadas.

- *Desarrollo de un generador de texto automático.* Para dar un experiencia aún más dinámica y entretenida, se crearía un generador de texto capaz de ofrecer diversos resultados equivalentes para una misma situación. Es decir, que ante una misma situación a describrir, no genere siempre el mismo texto, sino que sea capaz de introducir variedad. También debería soportar varios idiomas.

- *Habilitar una interfaz adecuada para todos los jugadores.* Armonizar la pantalla de juego y las ventanas centradas al texto para ofrecer una interfaz que se pueda disfrutar por cualquier tipo de usuario, vidente o invidente.

1.3 Metodología de desarrollo

Tal y como se planificó, se ha seguido una metodología iterativa-incremental que se englobaría en dos etapas principales. La primera fue la implementación del juego en si, comenzando con la generación de la mazmorra ya que sería de lo más complejo del juego y del que depende el resto de esta etapa. Con la mazmorra ya funcional, se desarrolló a las criaturas e items con sus respectivos comportamientos y efectos. Finalmente, teniendo ya la mazmorra habitada y con herramientas, se implementaron las múltiples pantallas que el juego cuenta para las diferentes situaciones tales como subir de nivel o disparar a un objetivo con un arco.

En la segunda etapa el foco del desarrollo se centró en la generación de texto, comenzado por implementar una interfaz que sea cómoda para mostrar el texto, compatible con varios idiomas y que sea legible por lectores de pantalla. Una vez implementada la interfaz, los esfuerzos se centraron en la obtención y organización de los vocabularios que se emplearían. Después, se implementó la clase que gestionaría estos recursos y, finalmente, la clase que construiría las frases descriptivas con las palabras y restricciones obtenidas por el gestor de los recursos.

1.4 Plan del proyecto

El proyecto se inició en el segundo cuatrimestre del curso 2017-2018, período en el cual se realizaron las reuniones iniciales y se tomaron varias decisiones, pero no se pudo avanzar mucho más a causa del trabajo del curso. Al acabar con dicho trabajo, en el verano de 2018 se planificó el desarrollo del proyecto. En al Tabla 1.1 se pueden observar las actividades que se habían planificado en ese verano.

Tarea	**Duración de la Tarea**
Investigación y documentación.	40 horas
Diseño inicial del juego.	15 horas
Diseño inicial de la gramática.	25 horas
Diseño inicial de la integración de la Wordnet en el proyecto.	20 horas
Comienzo del desarrollo del juego: la terminal.	5 horas
Generador de mazmorras.	40 horas
Diseño e implementación de criaturas y su IA.	25 horas
Diseño e implementación de items y efectos.	15 horas
Diseño e implementación de acciones y sus pantallas.	30 horas
Implementación de la interfaz.	15 horas
Incorporación de la Wordnet al juego.	40 horas
Implementación del generador de texto.	50 horas
Correcciones de código.	15 horas
Pruebas con los betatesters.	30 horas
Total	315 horas

Tabla 1.1: Planificación ideal del tiempo de cada tarea de proyecto.

Debido a diferentes responsabilidades y a dificultades inesperadas del proyecto, el tiempo que le podía dedicar menguó a la vez que el requerido por este se incrementó considerablemente. Un problema especialmente complejo fue la integración de *Wordnet*, que supuso prácticamenete un subproyecto a parte para poder obtener los recursos necesarios para el juego. Se hablará en más profundidad de esto en el AnexoAppendix A. Junto a este, también existieron problemas trabajando con algunos otros recursos, como los problemas de codificación de los *WikiCorpus*, del que también hablaremos en la Sección**??**. Esto implicó su división en tareas que, en conjunto, tenían una mayor duración que la esperada. Con los retrasos, el testeo y QA no fueron tan profundos ni amplios como se pretendían inicialmente. Del mismo modo, algunas tareas acabaron siendo menos complejas de lo previsto, por lo que fueron más cortas, por ejemplo, la implementación de criaturas. Todo esto se puede ver en la Figura 1.2

que representa el tiempo real empleado en el proyecto.

Tarea	Duración de la Tarea
Investigación y documentación.	40 horas
Diseño inicial del juego.	15 horas
Diseño inicial de la gramática.	25 horas
Diseño inicial de la integración de la Wordnet en el proyecto.	20 horas
Comienzo del desarrollo del juego: la terminal.	5 horas
Generador de mazmorras.	60 horas
Diseño e implementación de criaturas y su IA.	15 horas
Diseño e implementación de items y efectos.	10 horas
Diseño e implementación de acciones y sus pantallas.	30 horas
Implementación de la interfaz.	20 horas
Creación de un proyecto capaz de hacer consultas a la Wordnet.	15 horas
Creación de las funciones que crean los recursos del juego desde la Wordnet.	45 horas
Implementación del generador de texto.	50 horas
Correcciones de código.	15 horas
Pruebas con los betatesters.	No realizado
Total	370 horas

Tabla 1.2: Tiempo real empleado en el proyecto

Como se puede ver, el proyecto al final contó con 370 horas de desarrollo. Las reuniones realizadas durante el comienzo de este duraron aproximadamente 2 horas y se realizaron 3. Durante el resto del desarrollo, se realizaron unas 7 reuniones de 1 hora cada una. Esto implica que el proyecto además contó con 13 horas de reuniones para las que hay que sumar el coste por hora del tiempo del director y el analista. Puede verse la estimación del coste del proyecto en las Tablas 1.3 1.4. Se empleó el convenio definido por el *BOE*[1].

Perfil	Saldo/Hora sin impuestos aplicados	Horas totales	Total sin impuestos aplicados
Programador	25€	370 horas	9250€
Analista	35€	13 horas	455€
Director de proyecto	45€	13 horas	585€
Total			10290€

Tabla 1.3: En esta tabla, se muestra el coste de proyecto pero teniendo en cuenta la aplicación de los impuestos.

Perfil	Saldo/Hora con impuestos aplicados	Horas totales	Total con impuestos aplicados
Programador	17.809€	370 horas	6589.33€
Analista	32.78€	13 horas	491.14€
Director de proyecto	42.14€	13 horas	547.82€
Total			7625.29€

Tabla 1.4: En esta tabla, se muestra el coste de proyecto pero teniendo en cuenta la aplicación de los impuestos.

El coste de producción es prácticamente el mismo de desarrollo, ya que únicamente se ha empleado software cuyas licencias son gratuitas y no requiere ningun hardware específico para funcionar.

1.5 Estructura de la memoria

Junto a esta introducción, se encuentran en este documento los posteriores capítulos cuyo contenido se compendia a continuación:

- **Capítulo 2. Fundamentos.** Es este capitulo se describe los *roguelike*, respecto a las características de estos juegos y su historia. También se hace una introducción a la tiflotecnología y a la accesibilidad en los videojuegos.
- **Capítulo 3. Datos técnicos.** Aquí se describen las herramientas y tecnologías empleadas en el proyecto.
- **Capítulo 4. Desarrollo.** En él se realiza una descripción detallada del trabajo realizado en sus diferentes etapas. Adicionalmente, se detallan las estructuras y patrones utilizados en el desarrollo y la explicación del funcionamiento de las características más particulares de este.
- **Capítulo 5. Conclusiones y trabajo futuro.** Se comenta la experiencia adquirirda en el marco de este trabajo y los planes futuros para mejorar y expandir el juego.

Capítulo 2

Fundamentos

2.1 Concepto del *Roguelike*

El *Roguelike* es un subgénero del videojuego de rol que nació en los 1980 con el juego llamado *"Rogue"*[2], del que mostramos una captura en la Fugura 2.1(página 10). En él, el jugador controla a un personaje que explora una mazmorra con el objetivo de obtener un objeto y salir de esta. *Rogue* estableció las bases de un subgénero muy diverso ya que ni los mismos desarrolladores ni fans de estos títulos han consensuado unas características determinadas, de ahí el termino *roguelike*, "similar al *Rogue*".Sí existen una serie de llamadas "interpretaciones" al respecto, que pueden verse, en cierto modo, como convenios:

- **Interpretación de Berlín:** Se manifiesta en la *"Roguelike Development Conference"* del año 2008 [3] que se realizó en Berlín. Esta interpretación pretende definir, por así decirl, *cuánto* de *roguelike* tiene un juego. Para esto se definieron unas pautas en base a juegos que sí son considerados puros *roguelikes*, como el *NetHack*. Estas pautas pueden subir el "nivel" de *roguelike* de un juego más o menos según la categoría del patrón. Finalmente, añadir que algunas de estas pautas han sido criticadas, ya que esta interpretación puede llegar a ser un tanto restrictiva, como que el hecho de que el juego emplee gráficos basados en *ASCII* haga que el juego se considere más *roguelike*. A continuación, presentamos algunas de las principales características que harían que un juego fuese más *roguelike*:

  - **Reto táctico.** Se debe aprender sobre las tácticas antes de poder hacer un avance significativo importante. El juego debe proporcionar retos tácticos al jugador. Una de los medios del juego para realizar esto, es la obligación de tener que volver a empezar desde cero cuando se vuelve a jugar, eliminando la posibilidad de ser capaz de enfrentarse a retos de los últimos niveles.
  - **Juego no modal.** Las diversas acciones deben ocurrir todas en el mismo mo-

do, implicando que en todo momento del juego se pueda realizar cualquiera de las acciones definidas. En otros juegos, por ejemplo, en el momento de entrar en combate el modo de juego cambia, pasando a otro tipo de pantalla que funciona de forma diferente.

  - **Gráficos ASCII.** Método tradicional de mostrar la rejilla del mundo.
  - **Hack'n'slash.** Durante el juego, no habría ningun comportamiento amigable por parte de las otras criaturas, la única opción siempre es el combate.
  - **Un único personaje.** En el juego, sólo se puede controlar un personaje y la mazmorra debe estar centrada en este.

- **Interpretación de *Temple of the Rogue*:** El foro *Temple of the Rogue* es uno de los más populares entre los jugadores de este género. En esta web se definió en 2014, coincidiendo con el resurgir de los *roguelikes*, una nueva interpretación más generalista y abierta que la de Berlín, si bien en este caso se trata de una definición estricta. Se la conoce como *Interpretación clásica* [4] y está formada por 7 pautas entra las que se encuentran el *mapa de rejillas* y la *generación procedural*. Es de carácter más actual y más aceptada tanto por desarrolladores como por jugadores. Estas pautas son:

  - **Basado en turnos.** El jugador interactua por turnos, decidiendo la acción a tomar. Tras actuar, el resto de elementos del juego también realiza su acciones. El jugador puede no hacer nada en un turno, pero el resto del mundo sí llevará a cabo igualmente sus acciones.
  - **Basado en rejilla.** Puede ser ortogonal o hexagonal, siendo la superficie donde se disponen los elementos del juego. El movimiento es atómico, de una celda a otra.
  - **Fallos permanentes.** Los jugadores deben arriesgarse y responsabilizarse de sus decisiones, ya que no hay forma de guardar partida y luego volver a cargarla tras realizar una acción.
  - **Entorno procedurales.** En cada partida, la mazmorra es regenerada proceduralmente para dar una mayor rejugabilidad.
  - **Resultados aleatorios de los enfrentamientos.** Al igual que en los juegos de rol, los combates en *roguelike* se verán afectados por el azar. Por ejemplo, el daño realizado con un arma no debería ser el mismo siempre.
  - **Inventario.** Por toda la mazmorra hay items que el jugador puede recoger y emplear, tales como armas, pociones mágicas, comida, etc., pero debe tener cuidado ya que hay un límite de objetos a llevar y su uso puede ser limitado.
  - **Un único personaje.** El jugador está representado en la partida por un único personaje. Esta pauta es similar a la definida en la Interpretación de Berlín.

Más recientemente, en el año 2018 se realizó una nueva interpretación, la llamda del *roguelike tradicional* [5]. Esta viene influenciada por la necesidad de determinar mejor cuales son las características más tradicionales de un *roguelike* que se puedan encontrar en un juego actual. Con esto se busca favorecer a los juegos que siguen más el testigo de los *roguelike* clásicos. En esta ocasión, las pautas se reducen a cuatro y constan de unas definiciones aún más estrictas:

- **Centrado en un personaje.** El jugador controla únicamente a un personaje, al contrario que en otros juegos en los que no se controlan personajes, o en los que el jugador toma el papel de un director que maneja a un grupo de entidades.

- **Consecuencias permanentes.** Cualquier tipo de acción tomada no puede ser anulada de ningún modo. Con esto se busca que el jugador actúe con tácticas precavidas y estrategias a largo plazo, y con ello aumentar la diversión al avanzar por la mazmorra generada proceduralmente.
- **Contenido procedural.** La mayor parte del juego debe generarse proceduralmente para cada partida. Con esto, se incentiva al jugador a que se sienta menos dolido por las consecuencias permanentes, ya que sus efectos desaparecen una vez se comience una nueva partida.
- **Juego basado en turnos.** Se debe tener el tiempo que uno quiera para pensar la decisión a tomar. Esto es importante dado que, al tener toda acción consecuencias permanentes, no se buscan decisiones rápidas por parte del jugador, sino movimientos bien planificados en situaciones críticas.

Partiendo de las diversas interpretaciones existentes, vamos a considerar como los patrones para nuestro *"roguelike"* los listados a continuación:

- **Permadeath o Muerte permanente.** Característica común en juegos con dificultad, significa que si tu personaje muere pierdes todos los avances y se debe empezar la partida desde el principio. Tampoco hay posibilidad de guardar o cargar partida. Por otro lado, ante este tipo de juego, dicha carácterística se ofrece como un aliciente para partidas cortas, intensas y divertidas.

- **Mapa de rejilla.** La mazmorra está divida en rejillas cuadradas en las que se sitúan los componentes del juego, un tanto similar a un tablero de ajedrez tal y como se puede ver en la Figura 2.1(página 10). Esto limita los movimientos, ya que sólo se podrá transportar un elemento de una mazmorra a una casilla que sea vecina.

- **Desarrollo de la partida por turnos.** Cada acción que realice el jugador es un turno y en un turno se realizarán también las acciones de los otros elementos de la mazmorra.

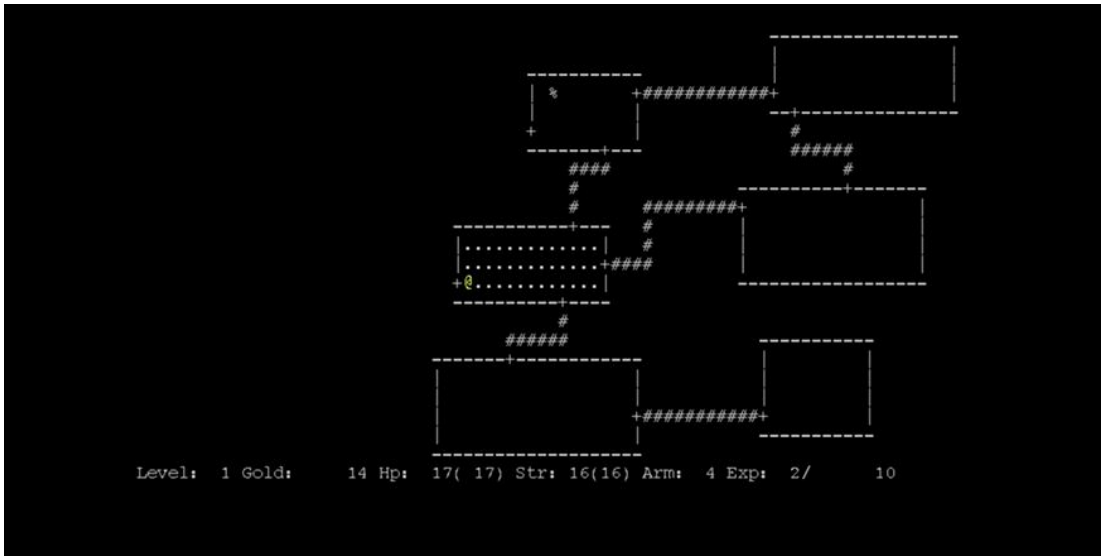


Figura 2.1: Captura de *Rogue* en el que se puede observar el clásico mapa de rejillas representado mediante caracteres *ASCII*

Por tanto, mientras el jugador no realice ninguna acciónb tampoco el resto de elementos de la mazmorra harán las suyas. Asimismo, significa que cuando el jugador realice una acción habrá movimiento en zonas de la mazmorra que no sean perceptibles por este. Cabe recordar que ”pasar turno” es considerado como una acción, por lo que el resto de elementos sí actuarían.

- **Aleatoriedad.** Junto a la *permadeath* esta es la característica que más favorece la rejugabilidad y diversión. Esto se puede ver claramente en la generación de la mazmorra, cuyas salas se organizan de forma aleatoria así como los elementos que se disponen en esta. Hay incluso algunos elementos, caso de pociones, cuya apariencia cambia de partida a partida, por lo que el jugador no sabe qué es hasta que lo usa.

- **Inventario y su gestión.** Durante la exploración por la mazmorra, el jugador encontrará diversos items, tales como armas, pócimas, etc., que le servirán en la aventura pero que también tendrá que gestionar, ya que no tienden a aparecer con frecuencia y tampoco suelen poder usarse más de una vez. También se deberá tener en cuenta que el tamaño del inventario es limitado, por lo que el jugador no puede cargar con todo lo que se encuentre y debe ser más selectivo.

(a) Captura de *GTA V* que sigue a día de hoy siendo una de los más exitosos, batiendo records.

(b) *PokemonGO* el fenónemo de los juegos par móviles.

Figura 2.2: 3 de los juegos que más beneficios dan a día de hoy.

2.2 Estado del arte: situación actual de la industria del videojuego

La industria del videojuego es ahora mismo la mayor industria de entretenimiento del mundo, superando a la música y cine [6]. Como ejemplo, el videojuego *Grand Theft Auto V* ha alcanzado las 90 millones de unidades vendidas, generando beneficios de 6 mil millones de dolares[6], lo que lo convierte en el lanzamiento más grande de la industria del entretenimiento. Asimismo, en el año 2018, el mercado de videojuego generó 115 mil millones de dolares y se espera que alcance los 300 mil millones para el 2025 [7]. En EE.UU., su principal mercado, 211 millones de personas juegan asiduamente a videojuegos [8], seguido ya por China, siendo la casa de *Tencent*, la mayor empresa del sector en estos momentos [9].

Respecto a las consolas, podemos comentar que la *PlayStation 4* ha sido la que más rápido ha alcanzado los 100 millones de unidades vendidas [10], y que la *Nintendo Switch* lleva ya unas 37 millones de consolas vendidas en dos años a la venta [11].

Para finalizar con las ventas, los juegos para móviles siguen siendo los que más beneficios generan y los que mayor crecimiento tienen [12]. De entre los juegos de móvil más exitosos se debe citar a *"Pokemon Go"* que a día de hoy lleva 2500 millones de dolares generados [13]. Algunos de estos otros juegos se pueden ver en la Figura 2.2(página 11).

Sobre los nuevos avances tecnológicos en la industria, tenemos los nuevos servicios de juego por *streaming* que buscan llevar una experiencia similar a la dada por *Netflix*, ofreciendo un catalogo de juegos bajo subscripción. En este ámbito destacaría el caso *Google*, que se adentrará en esta industria mediante su plataforma *Stadia* [14]. También se lanzarán el año que vienen las nuevas iteraciones de las consolas de *Sony* y *Microsoft*, que parece que intentarán favorecer en la industria el uso del *Path tracing*[15], tecnología con la que se simula una iluminación más realista. Mientras, el trabajo con equipos de realidad virtual sigue avanzando

y destaca el lanzamiedo del nuevo *headset* de *Valve* [16].

A modo de conclusión de este apartado, se puede decir que esta industria sigue siendo la que más crece dentro del mundo del entretenimiento, así como que más tecnología introduce a la sociedad. Así mismo, poco a poco se está convirtiendo en una de las más influyentes a nivel cultural.

2.3 Historia de los *Roguelike*

Para entender mejor los *Roguelike* y su historia, es conveniente conocer el precedente de este: el juego de rol. Un *juego de rol* es un juego de mesa en el que los participantes deben interpretar un rol de un personaje a lo largo de una aventura que está planificada por un tercero conocido como el *director de juego master*. Los jugadores deberán enfrentarse a la situaciones que se les presente según como quieran ellos, ya que tienen libertad en decidir cómo actuar. Para aumentar las capacidades de las acciones que los jugadores pueden tomar, los roles tienen diferentes habilidades según sus atributos; por ejemplo, el jugador con rol de mago será capaz de realizar hechizos. También interviene el factor del azar, ya que no se puede tomar ninguna acción importante en el juego sin lanzar unos dados para determinar el resultado. Las partidas de rol suelen durar mucho tiempo por lo que los personajes van ganando lo que se llama ”puntos de experiencia”, que le permiten aquirir mejoras en sus diferentes capacidades y así permitirles enfrentarse a problemas más complicados. El juego más conocido de este tipo es ”Dragones y Mazmorras”(*”Dungeons & Dragons”*), el cual está ambientado en un mundo de fantasía medieval de espada y brujería y que ha acabado por definir los clichés que nos solemos encontrar en buena parte de los videojuegos del género de rol [17].

Como ya hemos comentado en la Sección 2.1 *”Rogue”* nació en 1980 de la mano de Glenn Wichman y Michael Toy[]RogueHistGam. Este tiene la premisa de que el jugador aparece en la primera planta de una mazmorra y este no puede escapar sin el *”Amuleto de Yendor”*, que se encuentra al fondo del laberinto. El jugador deberá avanzar por los niveles de la mazmorra para obtener el amuleto y luego regresar a la superficie con este para escapar. La ambientación del juego era de un mundo de fantasía como el ”Dragones y Mazmorras”, compartiendo también con este algunos puntos como el azar tan presente en las acciones y en el mundo.

Si bien no son *Roguelikes*, llegados a este punto es importante mencionar otros dos juegos de rol: *”Wizardry”* y *”Ultima”* 2.3 (página 13). Cada uno de estos juegos representa una aproximación diferente pero acertada al *juego de rol* de mesa clásico. Estos títulos se influirían mutuamente en sucesivas entregas y asentarían las bases del videojuego de rol tal y como lo conocemos hoy en día [18].

Lamentablemente, este subgénero iría perdiendo popularidad con el paso del tiempo por diversas razones, como la migración de los jugadores a las videoconsolas lo que generó una

(a) Wizardry establecería las bases de los de un subgéreno propio.

(b) Ultima ofreció un mundo enorme a explorar y permitía también a acceder mazmorras, las cuales se navegaban en primera persona.

Figura 2.3: *Wizardry* y *Ultima* los cuales definirían con *Rogue* los videojuegos de rol.

gran pérdida de popularidad del ordenador como plataforma de juego [19]. Aún así, llegó a salir algún juego del género para consolas como, por ejemplo, *"Azure Dreams"* para *PlayStation* (véase Figura 2.4 página 14). Para finales de la década del 2000 haría acto de presencia *"Dwarf Fortress"*, que sigue estando en evolución y desarrollo a día de hoy [20]. Se trata de un juego de *rol* muy complejo que se caracteriza por el inmenso mundo que ofrece para explorar y la variedad acciones que permite. Además, tiene un modo *roguelike*. También son peculiares las descripciones de los combates debido a la irracionalidad de situaciones que describe y el lenguaje que usa. He de decir, que en este aspecto ha influido en cierta manera en las descripciones generadas por mi juego, sobre todo en el uso de vocabulario.

Sin embargo, en los últimos años los ordenadores han retomado su posición como unas de las plataformas de juego más relevantes. Esto ha sido así gracias a diversos factores, como la aparición de palataformas de distribución digital como *Steam*[21] y *Good Old Games*[22], y el crecimiento de desarrolladores *indie*, los cuales no siguen las tendencias del momento ni dependen de decisiones de productoras, lo que les permite desarrollar juegos no tan *mainstream* caso de los *roguelike*. Con este retorno del PC, el *roguelike* también ha recuperado estatus gracias a nuevos lanzamietos en PC y consolas e incluso la aparición de nuevas variaciones del género, caso de los llamados *"roguelite*, que son juegos que constan de características de los *roguelike* pero no todas o no de un modo totalmente fiel a los *rogelike*. Algunos de los *roguelite* más populares actualmente serían *"Spelunky"*, la saga *"Pokemon Mystery Dungeon"* y *"Enter the Gungeon"*. En este último, del cual mostramos una captura en la Figura 2.5(página 14) el combate pasa a ser en tiempo real y orientado a disparos, si bien conserva aún la estructura de mazmorra de un *roguelike*. También otras sagas de juegos de rol han probado suerte recientemente con adaptaciones del género, como el *"Wizrogue"* de la saga *"Wizardry"*.

Está claro que el género goza de buena salud a día de hoy y que le queda mucho por ofrecer. Seguir dando soporte a estos títulos y hacerlos más accesibles resulta de gran interés.

Figura 2.4: Captura de *"Azure Dreams"* de la primera *PlayStation*.

Figura 2.5: Captura de *"Enter the Gungeon"* un buen ejemplo de *rogue-lite*.

2.4 Tiflotecnología y videojuegos para invidentes

La **tiflotecnología** es una tecnología de apoyo centrada en permitir el acceso de personas con ceguera u otros problemas visuales graves a las nuevas tecnologías. Respecto a su historia, es muy difícil de datar su origen aunque tendríamo sus precursore en la **tiflomecánica**, la cual buscaba el mismo objetivo pero empleando mecanismos. Dicho esto, se suele considerar como hitos importantes el prototipo de máquina parlante de Wolfgang von Kemplelen de 1791[23] y al rafígrafo de Foucault[24] de 1841 los cuales se pueden ver en la Figura 2.6 (página 15). La primera es una máquina que consistía en un conjunto de tubos de organo con los que s epretendía reproducir las vocales, siendo así el primer sintetizador de voz. El segundo dispositivo consiste en diez palancas, compuestas cada una de tecla y punzón, con las que se

(a) Rafígrafo.

(b) Maquina parlante de Kemplelen.

Figura 2.6: Hitos de la tiflomecánica.

realiza la impresión mecanizada de caracteres visuales en relieve punteado.

Más fácil de trazar es su historia en España, a la que llegó a comienzos del siglo XX en forma de una máquina Pitch [25], de las primeras de escribir Braille. También habría que tener en cuenta los audio-libros de los años 60, pero el desarrollo comienza de verdad con la microelectrónica. De este momento nace finales de los años 70 el dispositivo conocido como Optacon, el cual reproducía en una matriz táctil las imágenes que capturaba con una cámara. Este aparato se puede ver en la Figura 2.7(página 16).

Ya en la década de los 80, los ordenadores pasan a estar en el foco. Aparece entonces el *COBRA*, el primer software español dedicado a la tiflotecnología. Este software era capaz de convertir textos *ASCII* a Braille y formatearlos para poder imprimirlos en una impresora de Braille. También cabe destacar en esta década la aparición de las denominadas **líneas braile**, dispositivos que cuentan con unas celdas en las cuales se representan los caracteres braille mediante una matriz de agujas que suben y baja. Un dispositivo conocido de este grupo sería el *Braille Lite* que se puede apreciar en la Figura 2.8(página 16).

A día de hoy, posiblemente la tecnología de este campo que más destaca sería la de los **lectores de pantalla**, un software capaz de interpretar lo que hay en pantalla y que luego convierte en audio al usuario. Cabe destacar la gran variedad de lectores como *NVDA*[26] y *ORCA*[27].

En lo que respecta a los videojuegos, en concreto primero deberíamos retornar a comienzos de los años 70. Durante esta época, los ordenadores no disponían de una interfaz gráfica como las de hoy en día y únicamente eran capaces de representar textos. Debido a ello, existían por entonces los llamados *juegos basados en texto*, juegos que completamente delegaban en texto para su interfaz y sí mostraban imágenes, estas solían ser secundarias. Los jugadores podían tomar decisiones según las teclas que se pulsasen o introduciendo frases en lenguaje

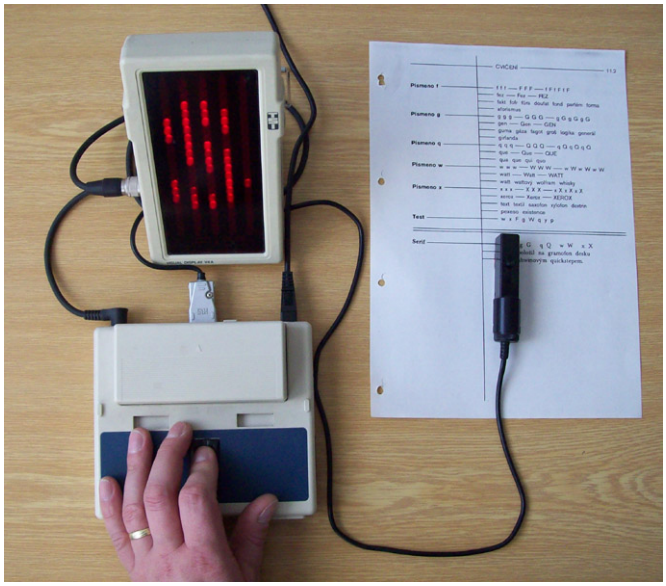


Figura 2.7: Foto del Optacon en funcionamiento.

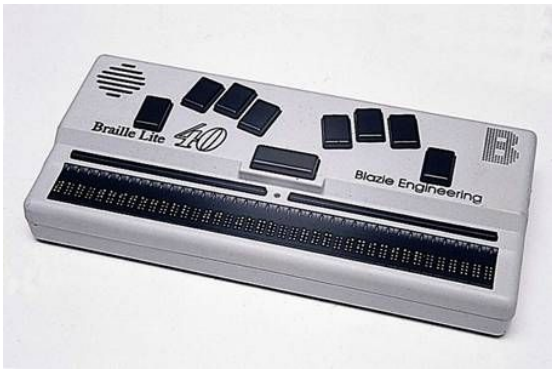


Figura 2.8: Foto del *Braille lite*.

natural. Debido a que confiaban tanto en el texto, estos juegos eran relativamente fáciles de usar por los invidentes mediante el uso de sintetizadores de texto. En la Figura 2.9(página 17) se puede ver una captura de *"Oregon Trail*, el juego más conocido e influyente de esta clase [28] aunque el primero que ofreció compatibilidad fue *"Colossal Cave Adventure"*(1976) [29].

Con el avance tecnológico, los ordenadores irían ganando capacidad para representar gráficos y los videojuegos pasarían a usar más este tipo de representación en lugar de la de texto. Esto redujo considerablemente la accesibilidad para los invidentes, viéndose progresivamente relegados a los juegos más simples. Como respuesta a este problema, algunos desarrolladores se pusireron a trabajar en videojuegos centrados en el aspecto del sonido, aunque basándose en otros ya existentes. Se acabarían creando videojuegos únicamente basados en audio y conocidos como *audiojuegos*.

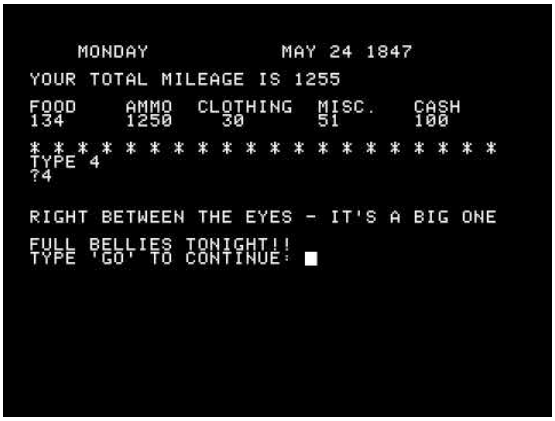


Figura 2.9: *Oregon Trail* en su verisón de 1978. El juego se creó en 1971 y aún sigue recibiendo nuevas versiones a día de hoy.

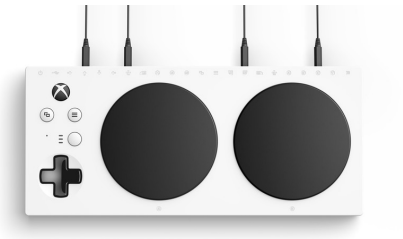
Con las mejoras en la tecnología de audio llegaría el sonido envolvente o posicional, el cual permite dar la sensación de que un sonido procede de una dirección determinada. Esta tecnología sería aprovechada para el desarrollo de nuevos y poder representar situaciones espaciales. Un buen ejemplo moderno sería el proyecto *RAD* [30], que consiste en una interfaz de audio aplicable a cualquier juego de carreras y que da la información necesaria para que el jugador invidente pueda disfrutarlo sin problemas.

Llegdos a este punto, es obligatorio destacar el papel que ha tenido internet en estos juegos accesibles. Hoy en día, existen numerosas comunidades y *webs* dedicadas a los *audiojuegos* y la accesibilidad. Un buen ejemplo es la web *audiogames* [31], la cual contiene un foro activo dedicado a este tipo de juegos y un archivo con el que se puede acceder a diversos títulos de este tipo.

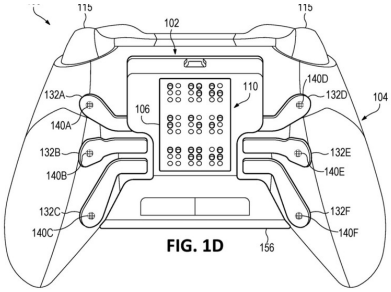
En el caso del hardware, tenemos a *Microsoft*, que ha lanzado un mando adaptativo altamente personalizable para dar apoyo a las personas con discapacidad. También registraron recientemente la patente de un mando con panel braille [32]. Ambos mandos se muestra en la Figura 2.10(página 18).

En lo que respecta a juegos *roguelike*, durante los últimos años ha habido diversos avances en ofrecer accesibilidad a personas invidentes. En la *Roguelike Celebration* del 2016 fue uno de los temas abordados los retos a la hora de adaptar un *roguelike* accesible a personas con problemas visuales[33]. Actualmente, existen ya diversos *roguelikes* accesibles para invidentes, destacando *Dungeon Crawl Stone Soup*, que se caracteriza por la amplia oferta de atajos y acciones que permiten al jugador actuar rápidamente [34]. Otro juego sería *Brogue-SPEAK*, una versión del juego *Brogue* que incluye un lector de pantalla propio y realiza un gran trabajo con los efectos de sonido [34].

Con nuestro proyecto buscamos iniciar un nuevo juego que se una a la tendencia actual



(a) Mando adaptativo anunciado el año pasado.



(b) Imagen de la patente del mando con panel braille.

Figura 2.10: Hardware accesible en videojuegos.

buscando así aportar nuevas ideas, experiencias y que pueda ser disfrutado por todo tipo de jugadores, videntes o invidentes, favoreciendo así su integración también en este ámbito.

Capítulo 3

Datos técnicos

3.1 Lenguaje Empleado

Desde el inicio del proyecto se decidió trabajar con el lenguaje *Java*, siendo la principal razón que la comunidad de desarrolladores de *roguelike* emplean este lenguaje por lo que no es difícil encontrar recursos, tutoriales, foros que traten el desarrollo de esto juegos con *Java*. Este lenguaje también da las ventajas de ofrecer universalidad entre diferentes plataformas por lo que no debería de haber problemas según el sistema operativo. Finalmente, hay que destacar que es un lenguaje muy orientado a la accesibilidad por lo que componentes del *Swing* no tienen problemas con los lectores de pantalla más comunes [35], aunque en *Windows* requiera activar el *Java Accessibility Bridge* [36] pero no implica mucho problema.

3.2 Equipo de desarrollo

Para el desarrollo se empleó el equipo especificado en la Tabla 3.1. Como se puede ver, el equipo supera con creces las necesidades de la tarea.

Modelo CPU	Intel Core i7 6700HQ
Velocidad CPU	2.60GHz
Núcleos por CPU	4
Hilos por núcleo	2
Caché L1/L2/L3	256KB/1MB/6MB
Memoria RAM	12GB
GPU	NVIDIA GeForce GTX 950M

Tabla 3.1: Especificaciones ténicas del hardware empleado.

3.3 Software de desarrollo

Para evitar problemas, se decidió emplear las librerías de terceros más necesarias procurando trabajar dentro de todo lo posible con lo ofertado por la API de *Java*. Al comienzo del desarrollo, se estudió emplear *"Necklace of the Eye"* [37] para la interfaz gráfica, pero se descartó por problemas de compatibilidad con *Java*. Algunas de las librerías de terceros que se usaron fueron *json.org* [38] con la que se pudo trabajar con los *json* que almacenan los recursos de texto y *asciiPanel* [39] que simula un monitor *ASCII* soportando 256 caracteres, colores en primero y en segundo plano y varias dimensiones para el terminal.

Respecto a la herramienta que obtiene los datos de la *Wordnet*, se ha empleado también la librería *json.org* y se ha añadido la librería *mySQLConnector* [40] con la que se puede acceder a la *Wordnet MCR 3.0* [41] la cual incluye inglés, español, gallego, euskera, catalán y portugués. Para obtener los datos de las palabras se comprobó la información contenida en el *Wikicorpus*[42], una colección de textos de *Wikipedia* con información lingüística.

A continuación, la Tabla 3.2 en la que se muestra el software empleado para el desarrollo del propio juego.

Sistema Operativo	Windows 10
IDE	Eclipse Neon.3
Versión de Java	8
Librería Gráfica	asciiPanel

Tabla 3.2: Software empleado en el desarrollo.

3.4 Licencias

Al momento de escoger una licencia para nuestro programa, se tuvieron en cuenta las licencias de las librerías y recursos de terceros empleadas en el proyecto. Las librerias que se han empleado en el juego constan de las siguientes licencias:

- **MIT License.**[43]: correspondiente a la librería *ascciPanel*.
- **JSON License.**[44]: de la librería *org.json*.
- **Apache 2.0.**[45]: de *commons-io*.

Ante estas licencias, se decidió que el juego tendría una **GNU General Public License 3.0** [46], dado que es compatible con las licencias mencionadas.

Respecto al generador de recursos tenemos además otras licencias, siendo algunas de recursos:

- **Apache 2.0.** [45]: empleada por *commons-io* y *java native access*.
- **JSON License.** [44]: una vez más, *org.json*.
- **GNU General Public License v2.**[46]: por *mySQLConnector*.
- **GNU Free Documentation License.**[47]: la misma que *Wikipedia*, empleada por *WikiCorpus*.
- **Wordnet License.**[48]: licencia propia de la *Wordnet* original del habla inglesa.
- **Attribution 3.0 Unported (CC BY 3.0).**[49]: licencia empleada por el resto de componentes de la *Wordnet MCR 3.0*.

Una vez más, se ha decidido licenciar con *GNU General Public License 3.0* ya que evitaba problemas de compatibilidad con las licencias mencionadas.

Capítulo 4
Desarrollo

4.1 Análisis de Requisitos

El proyecto se propuso con la idea de que pudiera ser jugado por personas invidentes, es por esto que fue necesario pensar en una interfaz competente. Desde el principio se decidió que el juego contaría con una interfaz *ASCII* para jugar, pero esto daría problemas de compatibilidad con los lectores de pantalla por lo que no se podía relegar toda la información a esta pantalla. Con este problema, también surge el de la localización ya que se deseaba que el juego fuera sencillo de localizar a diferentes idiomas y el formato *ASCII* iba a ser un gran inconveniente. Otro problema que se quería solventar era la ”monotonía” de los mensajes ya que lo normal es que para cada situación haya un mensaje que el juego muestra. Para una persona invidente esto sería muy cansino y molesto ya que el lector no pararía de repetir el mismo texto una y otra vez.

Del juego cabe decir que no se planteó hacerlo muy diferente al *Rogue* básico. La idea era crear un juego de partidas de corta duración, pero intensas y divertidas, buscando la rejugabilidad. También, el juego debería ser fácil de modificar para añadir nuevas criaturas o objetos al juego.

4.2 Diseño del juego

Ante los problemas planteados se decidieron tomar las siguientes opciones:

- Para solventar los problemas de interfaz y los lectores de pantalla, se decidió que se debería añadir algún tipo de ventana complementaria que acompañarían al terminal de juego, y en los que se mostraría la información del juego de forma repartida: una estaría centrada en los mensajes que se suceden cuando el jugador avanza por la mazmorra mientras que la otra actuaría con las diferentes pantallas de equipo como la de subir nivel. Al trabajar con *Java*, se sabía de la disponibilidad del *TextArea* así que se tuvo en

mente realizar la implementación con esta clase. En la Figura 4.1 (página 25), se puede ver la forma final de la interfaz. Como ya se comentó, también se estudió el emplear *Necklace of the Eye* [37], pero debido a incompatibilidades se decidió no emplearlo. Se pensaba emplear para dar una interfaz gráfica tridimensional.

- El conocimiento del *TextArea* comentada en el anterior apartado también serviría para solventar el problema de la localización respecto a la interfaz, ya que no habría problema para soportar caracteres unicode. También permitiría más comodidad en otros puntos de la localización, como cuidar el tamaño de un mismo texto en dos idiomas diferentes que puede dar problemas a una interfaz. TextArea ofrece las herramientas para solventar estos problemas fácilmente. Otro recurso que se pensaba emplear, era el de la *Wordnet* el cual podía controlar varios idiomas e identificar una palabra entre estos, lo que permite obtener la traducción de una palabra con tan solo un identificador. También se siguieron pautas definida en el libro *"The Game Localization Handbook"*[50], como evitar *"hardcodear"* texto en el juego, guardar los recursos en archivos de texto de fácil acceso, que no se requiera un editor de texto propietario para tratarlos y que los textos dentro del archivo estén identificados por una llave única. Con estas pautas, se decidió que los recursos serían almacenados en archivos *json* estructurados.

- Respecto el problema de la monotonía, se optó implementar un generador de textos. Este funcionaría en base al estado de la partida describiendo, por ejemplo, el combate entre dos criaturas. También debería ser capaz de trabajar con sinónimos y alternar sus usos en las frases, esto se resolvería mediante el uso de la *Wordnet* ya que agrupa las palabras en *synsets*, es decir, conjuntos de sinónimos identificados por una llave. Como es lógico, también debería ser fácil de preparar para que trabaje con un idioma nuevo, identificando los componentes de la frase y juntandolos según la gramática definida.

  Al principio se estudió realizar una *"Probabilistic Unification Grammar"* [51], en la cual las unidades gramaticales están asociadas con sets de valores de unas características. Estas unidades, se combinarían únicamente cuando otras características comunes se unían. Estos valores de características son, pues, la información del contexto de la frase a crear, reglas a cumplir. La implementación de esta habría sido mediante algún lenguaje lógico como *PROLOG*, ya que se ajusta bastante a la gramática.

  Finalmente, se decidió realizar una mucho más simple independiente del "contexto" de la frase, ya que iba a ser muy complejo y ya tratar el tema del "contexto" que daría para un trabajo propio. Si bien no se perfiló totalmente desde el principio, la gramática final ya se esbozó como una basa en plantillas por cada acción, en las que se irían asignando las palabras necesarias. Otra causa de esto fue que las *Wordnet* no almacenan ciertos tipos de palabras como determinantes y preposiciones.

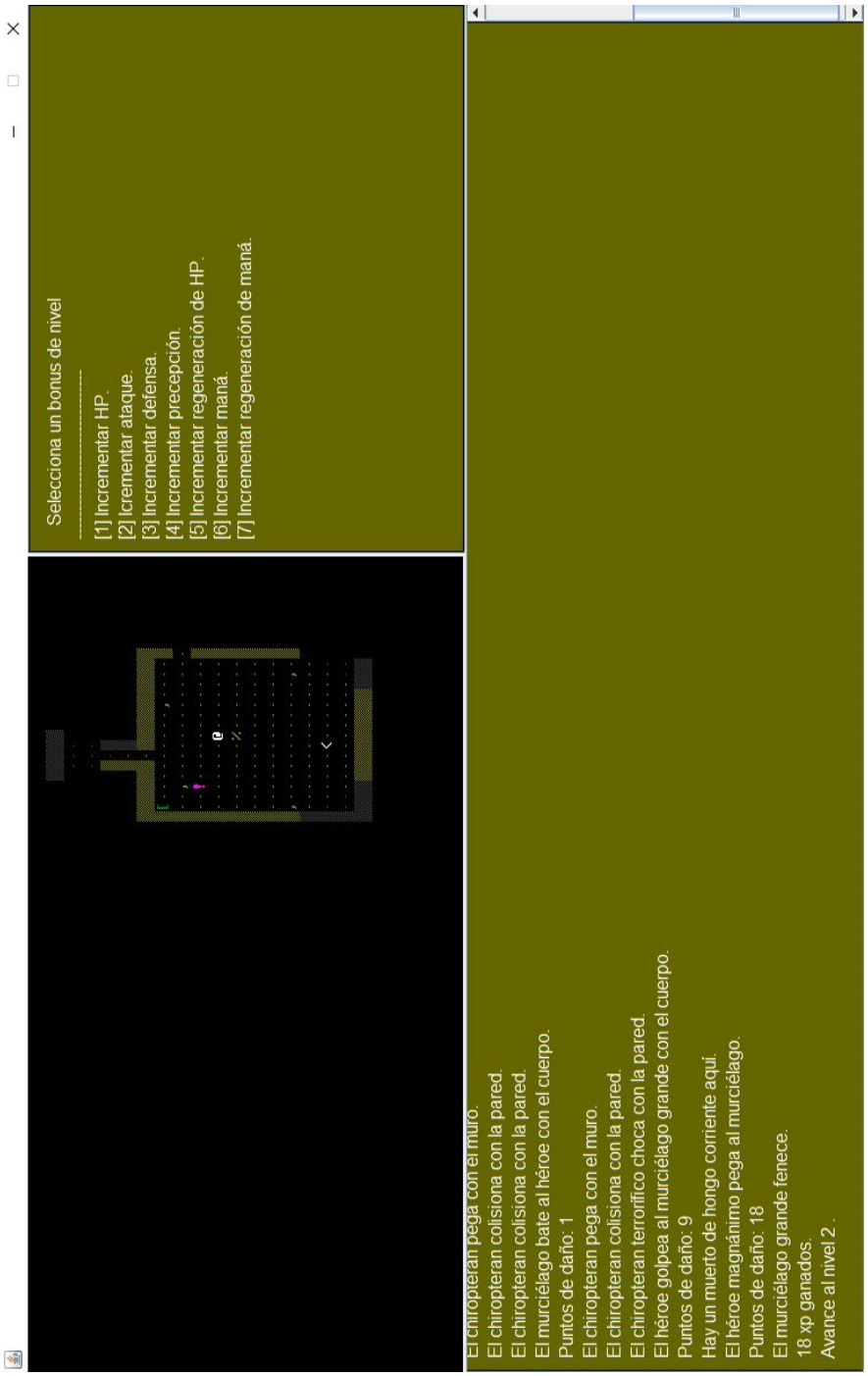


Figura 4.1: Imagen en la que se puede ver la interfaz del juego compuesta por un terminal *ASCII* y dos *JTextAreas*. En la inferior se muestran los mensajes de las cosas que van sucediendo en la mazmorra mientras que en la superior derecha se muestra el estado del jugador siempre y también las diferentes pantallas que puedan surgir como la de subir de nivel.

Para modificar palabras según las necesidades de la situación, se estudió implementar código que lo realizase o emplear NodebBox:Linguistics, más finalmente esta decisión se tomó más adelante en base al estado de los recursos en la *Wordnet*.

- Con el objetivo de mantener el juego lo más simple y divertido posible se decidió no hacerlo muy grande pero sí bien definido y con opciones divertidas para que el jugador pudiera experimentar. También, algunas de las criaturas de la mazmorra tendrían comportamientos heteróclitos que podrían llamar la atención del jugador.

- Con motivo de conseguir un juego fácil de modificar se decidió realizar la implementación del juego siguiendo consejos de la comunidad e incluso algún tutorial. Esto permitiría poder comparar resultados y tener también referencias de código por si se cometía algún error.

Estas decisiones se realizaron durante un tiempo en el que se dedicó a estudiar los recursos disponibles, el cual se centró principalmente en las *Wordnet* y las gramáticas dado a que eran algo que sería complicado de realizar y totalmente nuevo para mi. Investigando las primeras descubrí que existe un estándar conocido como el *KYOTO* [52] a cumplir, por lo que decidí que emplearía una que lo cumpliese. También, seguí la recomendación de mi directos de que buscase alguna que funcionase con algún sistema de gestión de bases de datos.

4.3 Implementación

La parte más costosa y duradera del proyecto con diferencia, comenzó con la realización de la ventana de juego con el terminal ASCII teniendo que definir una mazmorra inicial y las primeras pantallas del juego. Tras eso, me dispuse a la implementación de un constructor de mazmorras, la cual fue la parte más compleja de todo el proyecto. Al principio se estudio realizar la construcción de la mazmorra mediante el algoritmo *A estrella*[53], pero finalmente se decidió seguir el definido en *"Gamasutra"* [54] dado que sus resultados se ajustaba más a las necesidades del proyecto. El problema fue a causa de que muchos de los componentes del método tenían implementaciones que no se ajustaban a mis necesidades, como las de *Delaunay*, por lo que me ví obligado a implementarlas por mi parte. Curiosamente, de todos los componentes que tuve que implementar aquí, el que más problemas me dió fue el de obtener el centro de un círculo dados tres puntos. Esto fue debido a que no comprobaba bien todas las posibles situaciones de este cálculo, dando errores matemátios graves. Al final, este código se implementó adaptando el código desarrollado por Danylo Malyuta para *Matlab* [55].

Otros problemas con la creación de la mazmorra fueron habitaciones superpuestas, lo que obligó a definir en la clase *Room* una función que comprobase que dos habitaciones fueran disjuntas.

Una vez creada la mazmorra, ahora los problemas provenían del dibujado de esta los cuales eran causados por el cambio de coordenadas a las de una imagen, que el dibujado de los pasillos no estaba bien definido y que algunas posiciones de las habitaciones no eran generadas dentro del rango del terminal. Estos se solucionaron controlando el paso de coordenadas en el eje *"y"*, con una definición del dibujado en el que se tendrían en cuanta las posiciones y distancias entre las posiciones de dos habitaciones, y restringiendo las posiciones que se podrían tener en cuenta a la hora de generar la mazmorra.

Tras completar el constructor y su dibujado, comencé la creación de los diferentes elementos que habría en la mazmorra empezando por el jugador el cual sería representado por el carácter @. Tras tenerlo en la mazmorra y poder desplazarlo por esta, se porcedió a añadir criaturas fijas y, con estas, las primeras interacciones básicas entre el jugador con otros elementos. En esta parte no hubo mucho cambio de lo esperado, ni tantos problemas. Los comportamientos eran muy básicos, orientados al combate, y las criaturas no era complejas.

Después se porcedió a añadir varios niveles a la mazmorra y conectarlos mediante escaleras. Este punto tuvo algo de dificultad ya que había que controlar las posiciones de las escaleras permitiendo así que al volver a un nivel anterior el personaje apareciera en el punto donde estaban las escaleras previas. Esto también significo que el jugador debería aparecer en la escalera de salida del primer nivel siempre. Seguidamente, se porcedió a añadir criaturas que se comportaban de forma más inteligente, se estableció un campo de visión y los items. Lo primero requirió definir máquinas de estados finitos y una implementación del algoritmo *A estrella*[53] con el que las criaturas podría perseguir a los objetivos que se encontraban a su vista. Siguiendo con la vista, esto requirió implementar un campo de visión el cual se realizó generando una línea del tamaño del radio de visión y se comprobaba lo que había al alcance. Esta línea se dibujaba siguiendo el *algoritmos de la línea de Bresenham* [56]. Respecto a los items, se implementó la clase y se definió el atributo *inventory* que permitiría a las criaturas llevar un número de objetos.

Tras completar la parte del juego se volvió a la interfaz. Ya se tenía algo que se podía jugar y que mostraba los mensajes mas había que adaptar la interfaz para que fuera accesible a los lectores de pantalla. Para ello, se modificó la clase principal añadiendo dos *TextAreas*, mas estas no funcionaban bien ni permitían tantas modificaciones como me esperaba. Por ello, se pasó a usar *JTextAreas* las cuales no requirieron mucho cambio en el código ya que comparten muchas funciones. Llevó un poco de tiempo modificarlas para que se ajustaran al terminal del juego, pero se consiguió. Con las dos *JTextAreas*, surgió el problema de como permitir el acceso a estas desde el resto de pantallas del juego, por lo que se definió una clase *singleton* que se iniciaría con estas dos componentes, y se encargaría de manejarlas. Luego se tuvo que permitir su lectura por parte de los lectores de pantalla, lo que implicó permitir que fueran *focusable* y había que controlar las entradas del juego. Si esto último no se hacía, el

usuario se vería incapaz de salir de las *JTextAreas* por lo que no podría jugar.

Una vez acabada la interfaz, comenzó el trabajo con la *Wordnet* el cual fue bastante más complicado de lo esperado. El primer problema fue que pensaba que se podría decidir el *synset* adecuado para el contexto de la situación de alguna forma automática, e intenté buscar algún modo de realizarlo. Desistí de esta opción, ya que era muy compleja y se trataba de un trabajo de *Word Sense Desambiguation*[57], una problemática que ya daba para un trabajo propio. Con esto, yo mismo tuve que ir comprobandos los *synsets* de la *Wordnet* e ir apuntando los identificadores de estos para poder obtenerlos en el proceso de crear los recursos.

Una vez obtenidos los identificadores, se porcedió implementar las peticiones en java a la *Wordnet*. Hubo éxito, ya que los grupos de palabras se escribían agrupados en archivos *json* bien estructurados, mas surguió el problema de que sólo se tenían las palabras, no los datos morfológicos. Se estudió realizar peticiones al *API* de la RAE (Real Academia Española) *NLP Linguakit*[58], pero finalmente se opto por emplear el *WikiCorpus* [42], que indirectamente también resolvió el problema de la conjugación de los verbos. Aún así, esto llevo a un punto crítico, ya que se tenía pensado que toda esta parte del proyecto iría integrada en el juego, ya que se deseaba que el juego tuviera un proceso de instalación que consistiría en la creación de todos estos recursos, pero se estaba haciendo demasiado grande. Es por esto, que el trabajo con *Wordnet* acabó siendo un proyecto propio del cual se aprovecharían los recursos generados. Finalmente, comentar que el *WikiCorpus* también fue problemático, debido a que pese tener archivos en español, no estaban en *unicode*, por lo que las palabras estaban con caracteres incorrectos.

Finalmente, se porcedió a la creación del generador de texto. Este motor se diseñó en base a la teoría de *Natural Language Generation* que define dos componentes importantes, el *Planificador del Discurso* y el *Realizador de Superfie* [59]. Aproximandome a estos dos conceptos, implementé una clase que obtendría las palabras necesarias desde los recursos del programa y otra que realizase las frases juntando estas palabras mientras respetaba las restricciones definidas por estas. Algo que se tuvo que decidir fue el momento adecuado en el que el *Planificador del discurso* debía actuar para obtener algunas palabras como los nombres y características de las criaturas, ya que podría hacer al crealas o de forma dinámica en ejecución. Se optó por la primera opción, que sería más cómoda de tratar y daría resultados igual de buenos y más estables, además, así se podría establecer el género del jugador de forma aleatoria. Una cosa que cambió un poco, fueron las plantillas ya que al final no se definieron para cada acción, esto se hizo sólo en parte ya que están definidas en grupos de acciones y, en ocasiones, algunas plantillas más específicas. Las diferencias en estas plantillas, se intuiría en qué componentes forman parte de esta y cuales no. Gracias a la investigación previa, este apartado no fue tan costoso como pensé inicialmente, aunque sí que requirió escribir mucho código delicado aunque totalmente operativo y fácil de aprovechar. Donde sí surgieron problemas fue en la

definición de los recursos, que significo modificar la herramienta de la *Wordnet*, especialmente por el verbo *to be*.

Ahora, se procederá a describir en más detalle como están implementados los componentes del proyecto.

4.3.1 La interfaz

Como se comentó previamente, la interfaz consta de tres elementos: un terminal *ASCII* y dos *JTextAreas* tal y como se puede ver en la Figura 4.4 (página 32).

Inicialmente, se estuvo trabajando únicamente con la terminal, mediante la cual se irían probando las pantallas iniciales del juego. Para implementar las pantallas, se creó una interfaz *Screen* en la que se establecían las funciones básicas de una pantalla: *displayOutput()* que se encargaría de mostrar la imagen necesaria y *respondToUserInput()* con la que las pantallas serían capaces de capturar las entradas del juegador y tratarlas. La primera implementación de la interfaz fue *StartScreen* que simplemente contiene un mensaje de introducción indicando que se pulse cualquier tecla para comenzar partida. Tras pulsar alguna tecla, se pasa a la pantalla *PlayScreen* que es donde transcurre la mayor parte del juego. Es en esta pantalla en la que el mundo realiza sus actividades y el jugador lleva a cabo sus acciones. Según las acciones ejecutadas, *PlayScreen* llamará a otra *Screen* más especializada en tratar la acción. Una vez realizada, se vuelve a *PlayScreen*.

PlayScreen es también responsable del posicionamiento de las criaturas e items en la mazmorra. Esto es útil, ya que se ha establecido que aparezcan nuevas criaturas cuando el jugador obtenga el amuleto al fondo de la mazmorra. Otras responsabilidades de esta clase son la impresión de la mazmorra en el terminal y controlar el *scroll* de esta. Finalmente, también es la encargada de comprobar si el jugador cumple la condición de victoria.

De las otras pantallas, primero hay que destacar las clases abstractas *InventoryBasedScreen()* y *TargetBasedScreen()*, que se usan como base para situaciones en las que se requiera visitar un listado de elementos y en las que se mueve un cursor por la zona perceptible del mapa. Un ejemplo del empleo de las dos clases sería la acción de lanzar un objeto, por la cual se accede primero a la pantalla *ThrowScreen()* que extiende a *InventoryBasedScreen()*, mostrando así los objetos que tienen el jugador en un lista. Una vez seleccionado el objeto, se pone la pantalla *ThrowAtScreen()* en la que el jugador puede seleccionar con un cursor el objetivo del lanzamiento.

Otra pantalla importante es *CheckEnviromentScreen* que no implementa a *TargetBasedScreen* aunque sea similar al contar con un puntero. Esto es debido a que requiere un mayor esfuerzo por la clase principal, pues en esta pantalla se lista todo lo que esté a la vista del jugador, si se encuentra en una habitación, su posición, y la posición de un puntero que podrá mover libremente por la zona perceptible, permitiéndole ver los detalles de los elementos que

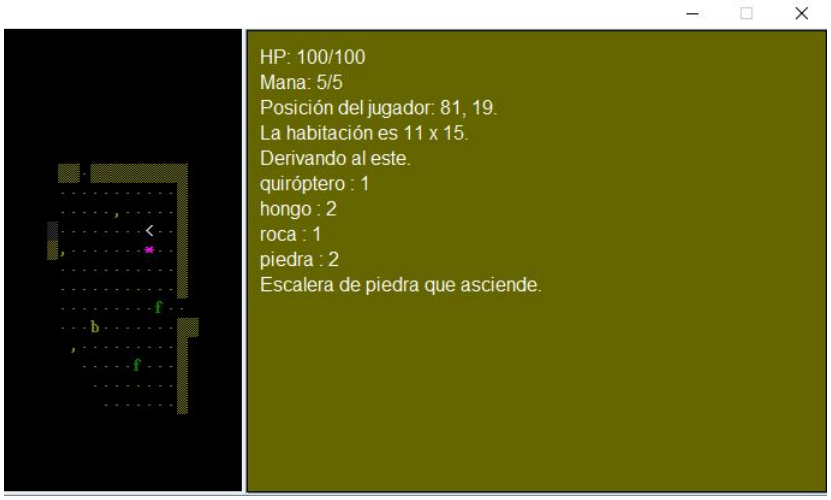


Figura 4.2: CheckEnviromentScreen, cuando el puntero se encuentra sobre el jugador. En este caso, se muestra múltiple información de lo que tiene a su alrededor, como las dimensiones de la sala, número de cada criatura y posición respecto al centro de la mazmorra.

hay. Al igual que las *TargetBasedScreen*, la línea del puntero y lo que hay a la vista funciona gracias al *algoritmo de la línea de Bresenham*[56]. En la Figura 4.2 (página 30) y 4.3 (página 31) se pueden ver unos ejemplos de su uso.

Esta pantalla requirió bastante testeo y tiempo, debido a que tenía que distinguir entre un pasillo y una habitación. Esto se podía complicar en algunas posiciones como las esquinas y las entradas a una habitación.

Una vez las pantallas estaban implementadas y los textos iniciales de las acciones se generaban, se porcedió a añadir los *JTextAreas* a la ventana de juego. No resultó complicado añadir estas componentes ya que sólo hubo que modificar la clase principal del juego. Ahora, para que los textos generados aparecieran en las *JTextAreas* hubo que implementar un clase *singleton* llamada *TextManager*. Esta clase es iniciada una vez en la clase *StartScreen* que le pasaría las dos *JTextAreas* que se encargaría de manejar. Así, cuando se necesitase escribir algo, se llamaría al *getInstance* de *TextManager* y se le llamaría a su función *writeText()* en la que se le pasa el texto a escribir en formato *String* y un 1 o un 2 para identificar la *JTextArea* en la que se quiere escribir. Finalmente, añadir que este *manager* también controla el eliminado de la primera línea, el borrado de contenido de una *JTextArea* y reemplazar líneas.

Para controlar las entradas y evitar problemas con el *focus* de los *JTextAreas*, se definieron dentro de estas *KeyListeners* y, en estos, se estableció en la función *keyPressed* que se controlasen las teclas de dirección, que se usarían para desplazarse por la pantalla de texto. El resto de entradas serían dirigidas a la pantalla que estuviese presente en ese momento para que se traten de la forma correspondiente. Si hay algún cambio de pantalla a causa de la entrada, el *KeyListener* se encargará de establecerla como la pantalla actual y se llamara a que se vuelva

Figura 4.3: CheckEnviromentScreen, cuando el puntero se coloca sobre otro elemento de la mazmorra. En este caso, se indicarán los datos del objetivo, como su vida en el caso de que sea una criatura. También se puede observar la línea del puntero dibujada mediante el algoritmo ya comentado.

a *pintar* la pantalla.

También se ha limitado la escritura en el segundo *JTextArea*, haciendo que cada turno su contenido se borre completamente y se añada el nuevo, en el cual se colocará el *caret* en la primera de las nuevas líneas. Para leer el resto de líneas nuevas, el jugador deberá desplazar el *caret* con la flecha direccional hacia abajo. Esto se ha hecho para evitar la sobresaturación de texto y que el lector de pantalla lea todos los turnos la última línea presente en el *JTextArea*.

Pese a que los *JTextAreas* se añadieron pensando principalmente en los jugadores invidentes no hay que olvidar que también se añadieron para favorecer la localización del juego a diversos idiomas. Esto se ha controlado estando definiendolas con fuentes compatibles para varios idiomas como, por ejemplo, el japonés.

4.3.2 La mazmorra

La mazmorra como tal, está definida por la clase *World* la cual contiene un array tridimensional de la clase *Tile*, esto viene a ser la mazmorra en sus varios pisos como una rejilla de tres dimensiones. La clase también tiene listas de las criaturas, escaleras, items y pasillos para poder tener acceso a cualquiera de estos cuando sea necesario. También posee las funciones necesarias para colocar elementos en posiciones de la mazmorra o eliminarlos de esta. Estas funciones serán invocadas externamente por criaturas o pantallas.

Antes de seguir, sería conveniente explicar algunas de las clases que guarda en listas, ya que son clave en la generación de la mazmorra

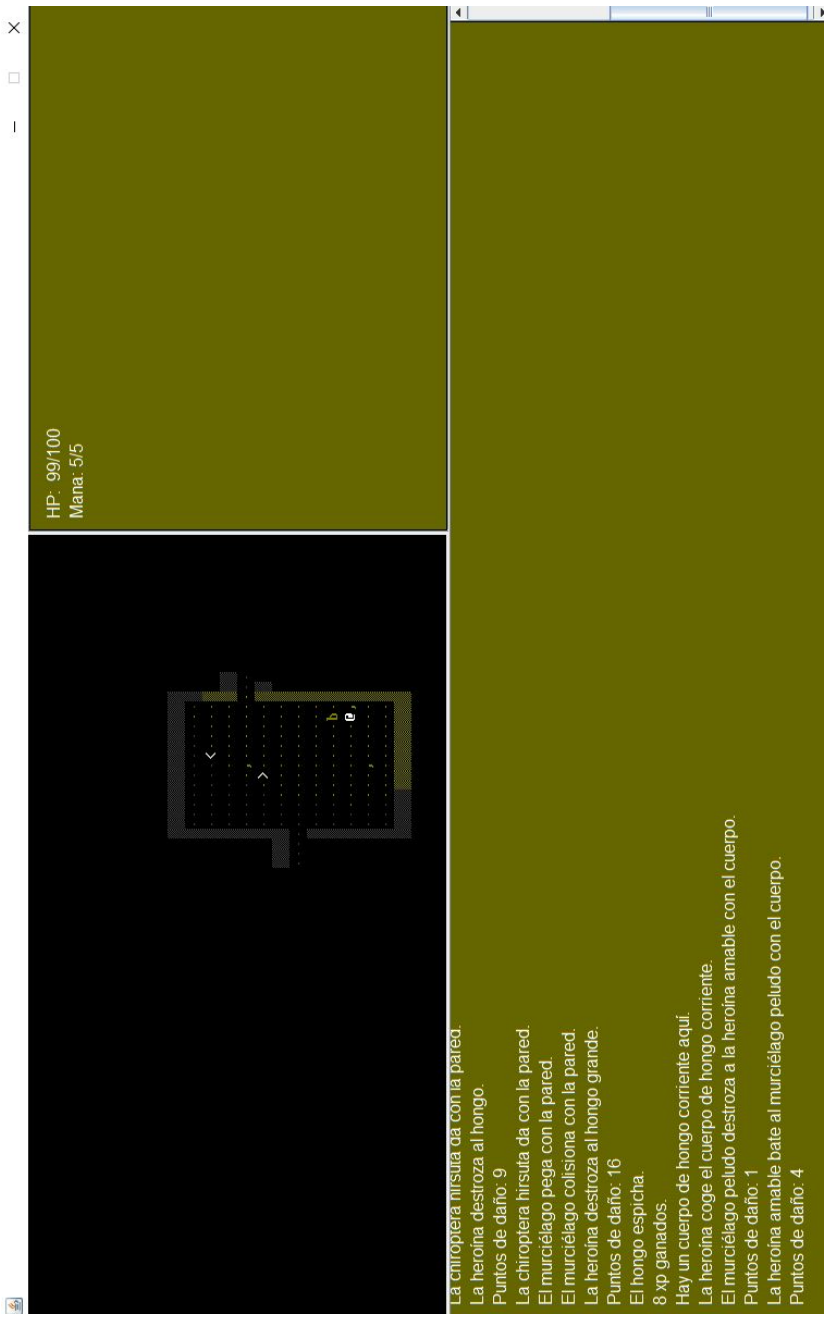


Figura 4.4: Captura en la que se pueden observar las tres partes de la interfaz.

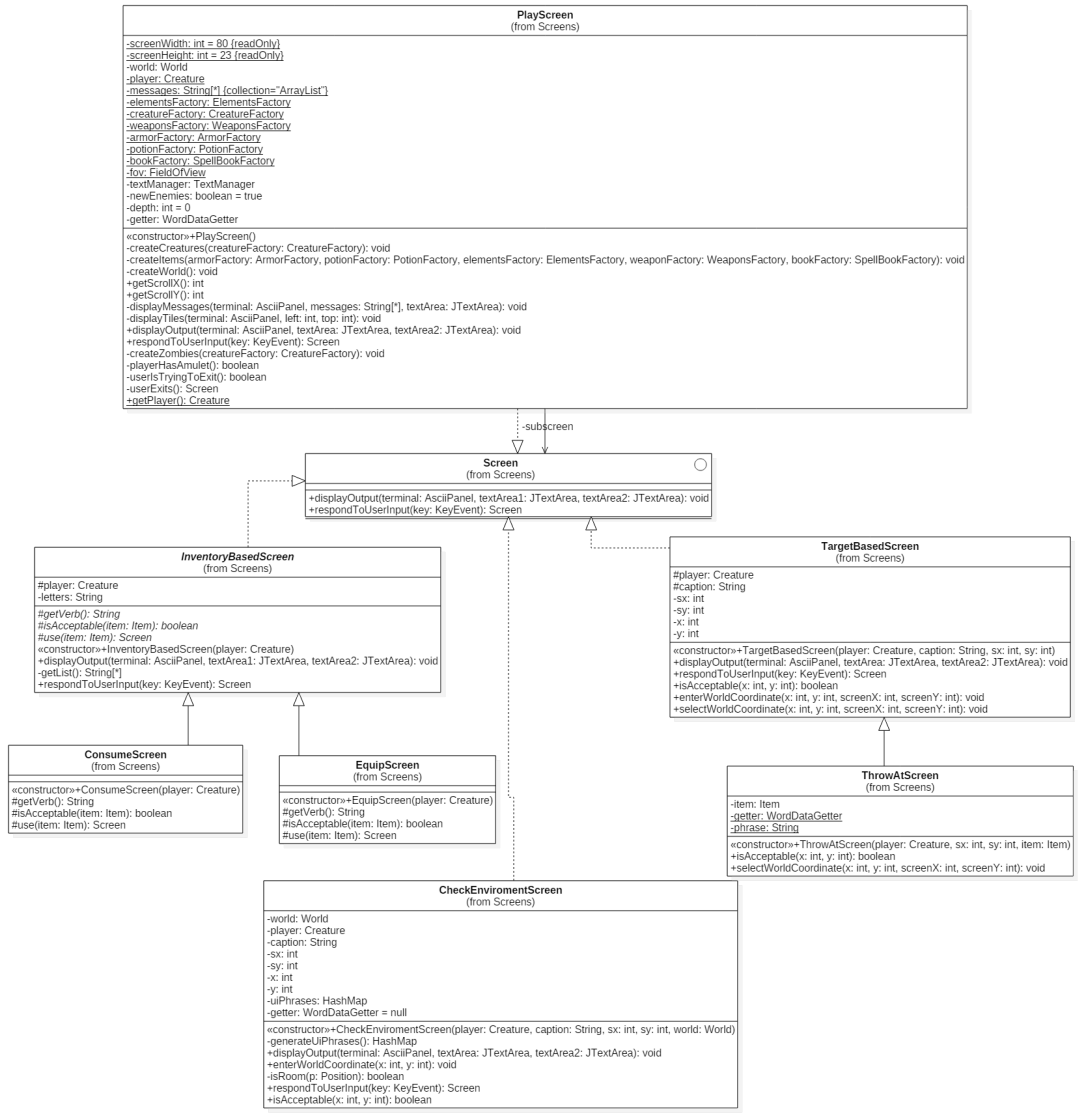


Figura 4.5: Diagrama que muestra algunas de las pantallas del juego. Se puede ver a la interfaz *Screen* en el centro y que algunas de las pantallas que la implementan tienen otras clases que las extienden.

- **Position.** Define una posición espacial con coordenadas en 3 dimensiones. Tiene funciones con las que también controla que las posiciónes no salgan del rango establecido y funciones con las que se pueden obtener sus vecinos en vecindad 4 y 8. También es capaz de generar un cuadrado respecto a otra posición que se le paso por parametro. Finalmente, pueden devolver la posición respecto a otro punto en términos de texto de los puntos cardinales.

- **Node.** Esta clase ha sido diseñada para trabajar con el algoritmo de Prim de árbol mínimo. Tiene de atributos un *Position* que representa la posición del mismo nodo, *Position* del nodo padre, un *ArrayList* de posiciones adyacentes y una *key* que es un valor numérico. Sus funciones son modificadores de atributos si bien la función que modifica la lista de adyacentes ha de comprobar que no se inserte ninguna posición repetida.

- **Room.** Esta clase define a una habitación de 4 paredes. Tiene un atributo del tipo *Position* por cada una de las esquinas y otro más para la posición central de la habitación. También tiene un *ArrayList* de la posiciones que forman los muros de la habitación.

- **Corridor.** Esta clase representa los pasillos que unen a las habitaciones. Sus atributos son dos *Position*, los cuales se corresponden a los centros de las habitaciones que conectan.

- **Tile.** Representa la unidad atómica del mapa del juego. Se caracteriza por ser un tipo *enum* ya que así se define cómo puede ser: *Tile.FLOOR*, *Tile.WALL*, *Tile.STAIRSUP*, *Tile.STAIRSDOWN* o *Tile.UNKNOWN*. Tienen múltiples atributos como color y símbolo que les identifica y vienen dados por los tipos comentados previamente.

- **Staircase.** Clase muy simple que almacena dos posiciones, la de de una escalera que baja y una que sube. Así se sabe el destino al tomar unas escaleras en el juego.

Este proceso es llamado por la pantalla *StartScreen* para obtener el objeto *World*. Debido a la complejidad de sus componentes, esta clase es generada por otras dos clases en colaboración: textitWorldBuilder y *WorldBuilderMathOperations* siguiendo así el patrón *Builder*. Para la creación, se decidió implementar un algoritmo de varias partes [54]:

1. **Generación de las habitaciones.** La clase *WorldBuilder* irá creando por cada nivel entre ocho y doce habitaciones, primero eligiendo la *Position* que será la esquina inferior izquierda de una habitación. Después determina el incremento a aplicar a esta posición para obtener la esquina contraria, la superior derecha. Con esto se podrían definir ya las cuatro esquinas de la habitación. Finalmente, se añade la habitación a la lista de habitaciones aunque antes se comprueba si es adyacente con alguna de las ya creadas.

Si esto es así, se vuelve a generar otra habitación. Cabe decir, que los incrementos son calculados entre unos rangos controlador para evitar problemas.

2. **Triangulación por Delaunay.** Ahora, *WorldBuilder* obtendrá con la lista de habitaciones una lista de centros. Pasará estas dos listas a *WorldBuilderMathOperations* para tratarlos con la triangulación de Delaunay. Este método matemático permite definir entre un conjunto de puntos triángulos cuyas circunferencia circunscrita no contiene un vértice de otra triangulación. Esto se puede apreciar mejor en la Figura 4.6 (página 35).

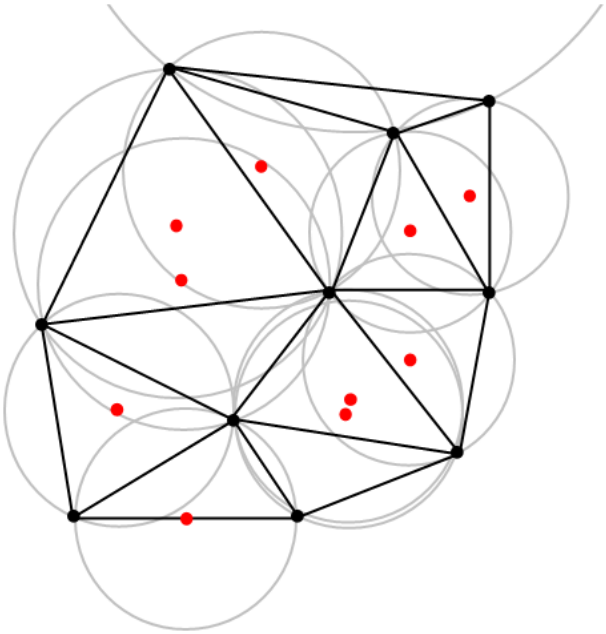


Figura 4.6: En esta imagen se pueden ver los triangulos definidos por tres puntos y las circunferencias que no contienen ningun vértice de otro triángulo.

Sabiendo esto, primero se obtienen las permutaciones en grupo de 3 posiciones de la lista de centros. Estos resultandos serán comprobados uno a uno, primero obteniendo el centro del círculo que definen, y luego el radio de este para comprobar que no haya ninguna otra posición a su alcance que no forme parte del grupo de 3 posiciones con las que se calculó. Si se cumple la condición, se procederá a crear *Node*s por cada posición de la permutación, estableciendo a las otras como posiciones adjuntas. Luego son añadidos a una lista de *Node*. Hay que tener en cuenta que previamente se comprueba si ya hay un *Node* con esa posición en la lista, dado el caso se actualiza la lista de adjuntos.

Una vez finalizado, se devuelve el la lista de nodos, un grafo conexo, a la clase *WorldBuilder*. Con este grafo se realizará un filtro sobre la lista de habitaciones ya que es posible que alguna haya sido eliminada durante el proceso.

3. **Arbol mínimo de Prim.** Con el grafo conexo, se vuelve a delegar en *WorldBuilder-MathOperations*, ahora para que aplique el algoritmo de árbol mínimo de *Prim*[60]. El grafo se irá procesando a la vez que generará objetos del la clase *Corridor* a medida que se van sacando los nodos a los que se les ha asignado un padre. Estos pasillos tendrán de atributos los valores de posición del nodo y su padre. Finalmente, la lista de pasillos será pasada a *WorldBuilder*.

4. **Pintado de las habitaciones y de los pasillos.** Volviendo a la clase *WorldBuilder*, ahora se procede a dibujar las habitaciones en el array tridimensional de *Tile*. Para esto, simplemente se va asignando al *Tile* el valor de *Tile.WALL* o *Tile.FLOOR* que están definidos en su clase *enum*. También se irán determinando y dibujando los pasillos en base a la información contenida en la clase *Corridor*, y la situación espacial entre dos habitaciones que tienen una conexión.

   Entrando en más detalle en el último paso, este se realiza considerando si entre dos habitaciones que hay que conectar, hay algún rango de valores en alguno de los ejes que comparten, es decir, si hay alguna posición de sus paredes en paralelo. Ante este caso, se dibujaría un pasillo simple recto. En el resto de casos, se requiere dibujar un pasillo con cambio de dirección. Esto se hace primero determinando el punto medio del pasillo a dibujar. Después, se resuelve para cada una de las habitaciones involucradas la esquina más próxima a este punto medio. Finalmente, se va dibujando desde la dicha esquina, incrementando en los ejes hasta llegar al punto medio. Luego, se hace lo mismo con la esquina de la otra habitación, resultando en un pasillo formado por dos rectas cruzadas. En el array de *Tile*, las casillas de un pasillo también son marcadas como *Tile.FLOOR*.

5. **Unión de niveles mediante escaleras.** Con los niveles construidos, ya sólo queda conectarlos. Para esto, primero se obtienen dos posiciones aleatorias de dos habitaciones de unos niveles adjuntos. Un nivel es el superior y el otro el inferior. Luego, se crea una instancia de la clase *Staircase* en la que se establece a las posición de comienzo y final, es decir, dónde está el extremo superior de la escalera y el inferior de esta. Finalmente, se dibujan en el array de *Tile* las escaleras en las dichas posiciones.

Con esto la mazmorra ya estaría completamente construida, y el objeto *World* sería inicializado con la función *build* en la que ya se le pasan todos los componentes necesarios. Finalmente, un diagrama en el que se pueden ver las principales clases involucradas en la Figura 4.7 (página 37).

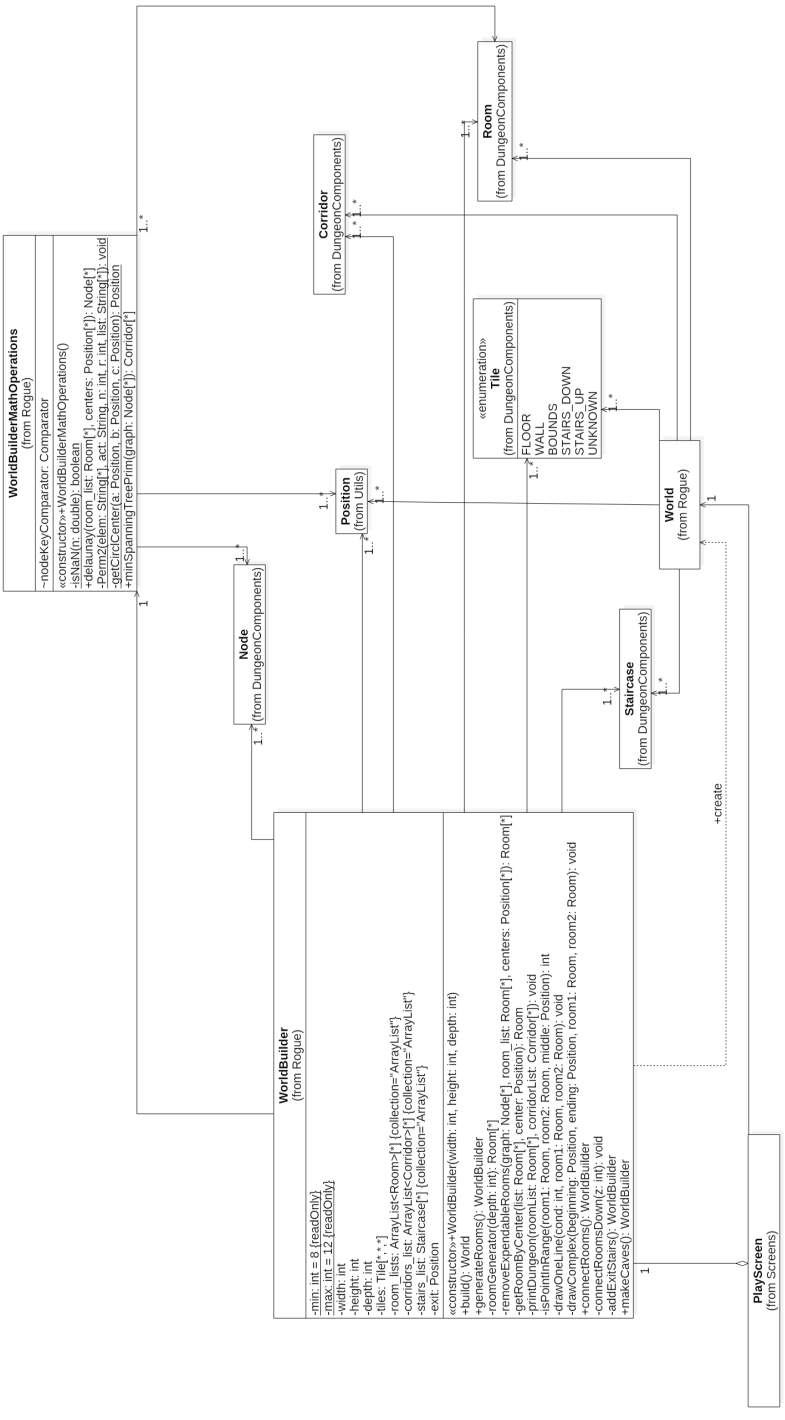


Figura 4.7: Diagrama que muestra las clases encargadas de la construcción del *World*.

4.3.3 Elementos de la mazmorra

En esta sección se describe el proceso de implementación de los elementos que se puede encontrar el jugador en la mazmorra y el de sus características.

Criaturas

El juego consta de una clase *Creature* empleada para crear la fauna de la mazmorra. Esta clase consta de un gran número de atributos, muchos necesarios para definir las diferentes características y capacidades de los múltiples tipos de criaturas. De los que definen características podemos comentar:

- **visionRadius.** Define la cantidad de casillas que se pueden percibir, determinando así qué hay en estas.
- **maxHp.** Indica la la máxima cantidad de vida que puede tener la criatura.
- **maxMana.** Similar a la anterior, pero define la cantidad máxima de maná que puede tener. Este valor es empleado para realizar ciertas actividades con algunos items.
- **level:.** Valor que se va incrementando a medida que la criatura luche más y gane experiencia. Al subir de nivel, se puede subir una de las características de la criatura.

Por otro lado, de los atributos que definen capacidades hay que mencionar a *inventory*, un atributo de la clase con el mismo nombre que permite gestionar un array de elementos del tipo *item*. No todas las criaturas van a tener un inventario.

Respecto a funciones, esta clase consta de varias con las que se definen las acciones que pueden realizar una criatura. Unos ejemplos de estas funciones son *pickUp* que permite recoger objetos y *getVisibleThings* que obtiene un *HashMap* de pares *String* y *ArraList* de posiciones. Estos pares se emplean para guardar las posiciones de criaturas, items y escaleras que hay bajo el *visionRadius*. Esta función está pensada para emplearse en la pantalla *CheckEnviromentScreen* la cual se diseñó para dar información completa de la situación espacial de la partida en un momento dado.

Otra función muy importante es *doAction*, por la cual la criatura comunica sus acciones al otras que se encuentren en su radio de percepción. Hay que comentar sobre esta función la presencia de otra, versión, *doActionComplex* que realiza lo mismo que la normal pero relega en el generador de texto. Se hablará de ella más adelante en section 4.5.

Aún con todas estas definiciones, *Creature* no define al comportamiento de la criatura, esto se delega en extensiones de la clase *CreatureAi*. Esta clase abstracta define múltiples funciones centradas especialmente en comprobar si una acción se puede realizar o no, por ejemplo, *canThrowAt* comprueba si se puede lanzar a un objetivo algún objeto. También define algunas

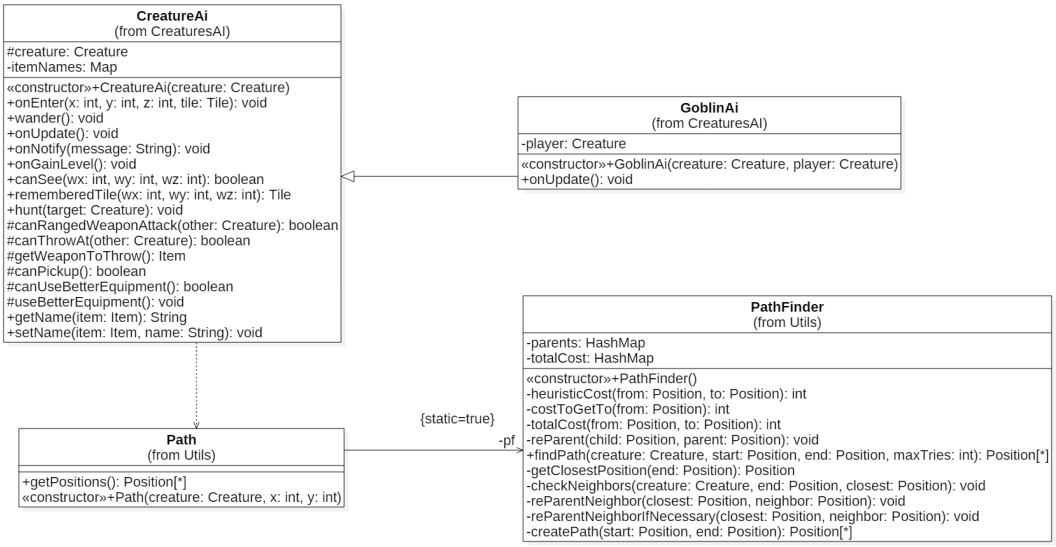


Figura 4.8: Diagrama en el que se muestra las clases involucradas en la realización de la función *hunt* por parte de un *goblin*.

funciones como *useBetterEquipement* que permite equipar automáticamente equipamiento de mejores características si es que se dispone de estos en el inventario. Otra función, es *canSee* que determina si una posición es perceptible por el jugador. Dicho esto, la función que hay que destacar dentro de esta clase abstracta es *hunt*, la cual permite a una criatura perseguir a otra que se encuentre en su campo de visión. Este carácter agresivo es llevado a cabo gracias a las clases *Path* y *PathFinding*. La primera clase tiene de atributo un*ArrayList* de posiciones el cual es generadon por *PathFinder* mediante una implementación del algoritmo *A estrella* [53]. El elemento de la clase *Path* es por tanto las posiciones a seguir para llegar a la criatura objetivo. Se puede ver un diagrama de las clases implicadas en el ejemplo en la Figura 4.8 (página 39).

CreatureAi cuenta también con un atributo llamado *itemNames* el cual es un *HashMap* de pares de *Strings*. Con este atributo se controla el conocimiento, o desconocimiento, de un item por parte de la criatura. Este atributo será gestionado por dos funciones y será llamado desde la función *nameOfItem* definido en la clase *Creature*. Esta situación se dará cuando la criatura realiza una acción, por lo que se define un nuevo *doAction* que recibe también el item como parámetro.

Como hemos visto, *CreatureAi* define muchos comportamientos inteligentes, pero no todas las criaturas tienen porqué usarlos o no tener algunos propios. Por ello se realizan extensiones de esta clase por cada tipo de criatura, las cuales suelen implicar la implementación de la función *onUpdate* de la clase *CreatureAi*. Se puede ver un diagrama de estas extensiones

en la Figura 4.9 (página 41). Es en esta donde se define el comportamiento de la criatura, las decisiones que ha de tomar durante el turno. A continuación, la implementación de la dicha función en *GoblinAi*, y se puede apreciar que se ha realizado la aproximación de una máquina de estados finitos.

```
1  public void onUpdate(){
2  if (canUseBetterEquipment())
3  useBetterEquipment();
4  else if (canRangedWeaponAttack(player))
5  creature.rangedWeaponAttack(player);
6  else if (canThrowAt(player))
7  creature.throwItem(getWeaponToThrow(), player.getX(),
       player.getY(), player.getZ());
8  else if (creature.canSee(player.getX(), player.getY(),
       player.getZ()))
9  hunt(player);
10 else if (canPickup())
11 creature.pickup();
12 else
13 wander();
14 }
```

Como se puede ver, en todo este trabajo con el comportamiento de las criaturas se ha empleado un patrón *Delegation* y en la creación del *Path* el patrón *Builder*.

Items

Todos los elementos inanimados del juego entran dentro de la categoría *Item*. Al igual que la clase *Creature*, esta cuenta con una gran cantidad de atributos pero son bien diferentes de los de *Creature*. Un ejemplo interesante serian *attackValue*, *throwAttackValue* y *rangedAttackValue*. Como se puede intuir, el item tendrá un diferente valor de ataque según como se define y el tipo de ataque que se use, por ejemplo, un arco tiene más valor de ataque al disparar que al golpear con el directamente. Ahora, hay dos atributos en esta clase de carácter singular:

- *quaffEffect*. Atributo del tipo *Effect*, una clase que define efectos de items en el juego. Un efecto es un estado que dura unos determinados turnos afectando a la criatura que se le ha aplicado. Pueden ser beneficiosos o benignos como el veneno. Este efecto ha sido creado únicamente para algunos tipos de items.

- *writtenSpells*. Consiste en una lista de elementos *Spell*. Esta clase define habilidades especiales que tienen un coste de maná y un efecto. Cuando se usan, hay que seleccionar un objetivo al que se le aplicará el efecto del hechizo. Los hechizos son variados, desde curar vida hasta invocar murcielagos. Este atributo también está pensando para algunos items determinados.

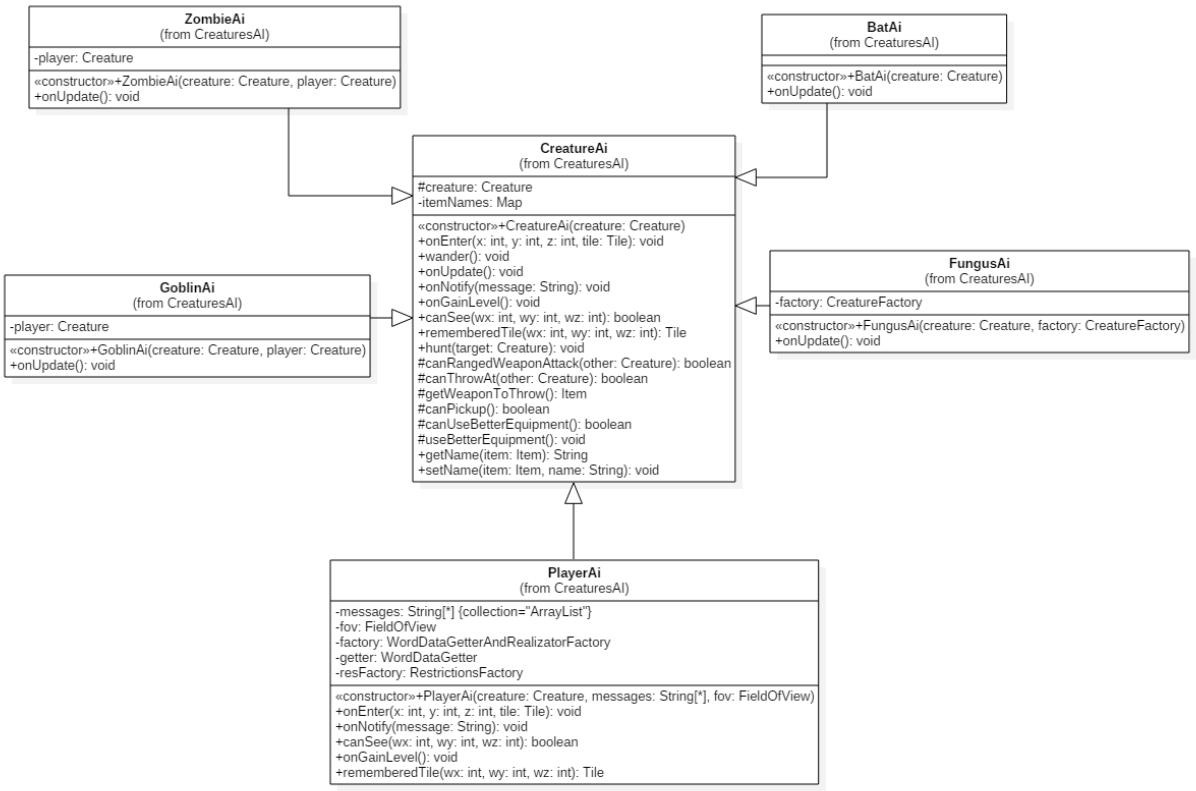


Figura 4.9: Diagrama en el que se muestra la clase *CreatureAi* y sus diversas extensiones.

Para finalizar con los atributos, es necesario destacar *appearance*. Este valor de tipo *String* es asociado a determinados objetos cuando son creados y los valores dados son escogidos aleatoriamente de un conjunto definido previamente. Con esto en el juego se verán objetos cierto tipo con una apariencia, pero que en otra partida serán diferentes. Esto da al jugador la incógnita de qué item se trata, ya que hasta no se use no se sabrá el nombre verdadero del objeto, únicamente su apariencia. Como ya se dijo, el juego debe tener una componente de aleatoriedad y generación procedural, y este atributo es el que aporta esos principios a los items.

A continuación, habría que tener en cuenta a los items que entrarían más bien en una categoría de ”equipamiento”. Estos son items que la criatura puede llevar puestos, obteniendo así algún incremento en sus características como en la defensa y en el ataque. Según estos incrementos se pueden dividir en dos grupos: armaduras y armas. Los primeros simplemente incrementan la defensa de la criatura, mientras que los segundos dan puntos extras al ataque de la criatura según cómo los empleen. Entre estos últimos cabe destacar al arco, el cual es especial por permitir atacar a distancia y sufrir desgaste al usarse para disparar. Cuando se dispara con el arco, su daño a distancia se reducirá un punto hasta que llegue a cero, situación que se interpreta como que se ha roto la cuerda de este.

Finalmente, hay unos pocos items que no entran en ninguna categoría y se que hablará de ellos en el siguiente apartado.

Factorias de los elementos

Para generar los elementos de la mazmorra, se tienen varias *Factories* cada cual especializada y con un atributo de la clase *World* para poder solicitar la colocación del elemento creado en el mapa y su incorporación a las listas que tiene este. Para las criaturas se tiene una factoría que consta con una función por cada tipo de criatura que se crea, incluyendo al jugador.

Para los items, se tienen cinco factorías:

- **ElementsFactory.** Esta factoría es la encargada de crear items que no encajan en ninguna de las otras categorías. Ahora mismo, dos items son los que se encuentran en la situación: piedras y el amuleto de *Hamsun*. El primer item es muy básico y sirve únicamente como proyectil, mientras que el segundo es el item que hay que recuperar para obtener el juego. También puede ser lanzado.

- **ArmorFactory.** Esta factoría es la que se encarga de generar los items de equipo que aumentan la defensa de quien los porte.

- **WeaponFactory.** Similar a la anterior, sólo que genera las armas que se podrán emplear.

- **PotionFactory.** En esta se crean las pociones que se podrán encontrar la mazmorra. Se

caracteriza por la función *setUpPotionAppearances* por la que se crean todas las posibles apariencias para las pociones y las mete en un array en orden aleatorio. Cuando se quiera asignar una apariencia, se obtendrá una de ese array mediante una posición. También hay que destacar que, junto a las pociones, se defines los efectos de estas.

- **SpellBookFactory.** La encargada de generar los libros de magia. A la vez que se definen los libros también se establecen los hechizos con sus efectos y son asignados a la lista de hechizos del libro.

Se puede ver un diagrama de las facotrias en la Figura 4.10 (página 44)

Para finalizar, las factorías de armaduras, armas, pociones y libros de magia constan de una función que permite generar uno de sus items de forma aleatoria. También, muchas de las factorías y funciones de estas constan con arrays de *Strings* que actuaran como llaves, pero de esto se entrará en más detalle en el apartado de **Generación de texto** section 4.5.

4.4 Creación de recursos léxicos

El primer paso fue escoger una *Wordnet* que se ajustara a los requisitos de localización del proyecto. Es por esto que se escogió el *Multilingual Central Repository* [41] dado a su soporte de inglés y español entre otros. También se trata de una base de datos *MySQL* por lo que no fue complicado de instalar.

Tras haber instalado y configurado la *Wordnet*, se procedió a probarla desde un proyecto diferente al del juego. En este, creó la clase *MySQLAccess*, necesaria para realizar las consultas a la base de datos y se comprobó que funcionaban correctamente.

Tras poder realizar solicitudes *SQL*, se implementó una interfaz *JsonCreator*, la cual debería ser implementada según el idioma que se desease tratar, ya que esta clase generaría los *json* estructurados con la información que precise el idioma. Estos *json* son de verbos, sustantivos, adjetivos y adverbios ya que la *Wordnet* únicamente controla estos tipos de palabras.

Ya en este punto, se decidió que este proyecto pasaría a ser una herramienta propia para generar los recursos del juego. No se implementaría dentro del juego por complejidad. Una vez decidido esto, se añadieron los archivos del *Wikicorpus*[42] del idioma inglés y español. Para tratarlos, se definió una interfaz *FindStr* que también debía ser implementada según el idioma que se quería tratar. El problema de esta clase es que ahora mismo delega en la función *findstr* de *Windows*, por lo que la herramienta sólo funciona con ordenadores *Windows* y puede llegar a ser muy lento ya que se ejecuta sobre los archivos del *Wikicorpus* que son numerosos y de gran tamaño.

Con todo esto, se tiene una herramienta bastante interesante por la facilidad de modificar y con muchas posibilidades a la hora de crear recursos lingüísticos estructurados en múltiples idiomas.

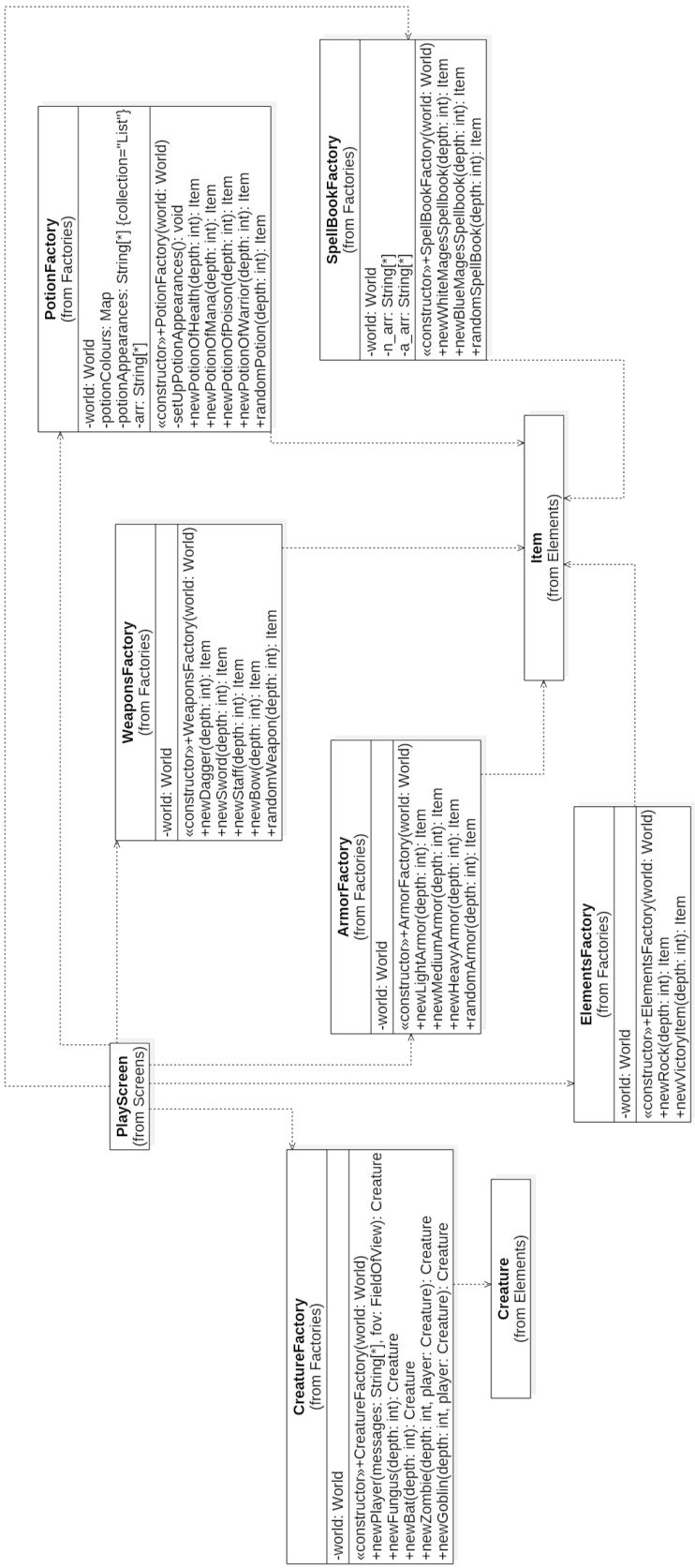


Figura 4.10: Se pueden observar las múltiples factorias, las cuales son llamadas por el *PlayScreen* y genera elementos del tipo *Item* o *Creature*

Se hablará en más detalle del funcionamiento de la herramienta en su propio anexo Appendix A.

4.5 Generación de texto

Como ya se ha comentado, para esta parte me inspiré en la teoría de *Natural Language Generation* [59] por lo que primero tenía que trabajar con algún medio para poder obtener las palabras de los recursos *json* que tenía para cada idioma. Lo que surguió fue la interfaz *WordDataGetter*, que establecía las funciones con las que se realizaban las consultas para obtener la información almacenada en los *json*. Algunas de estas funciones devuelven un *HashMap* con diversos datos y variables de la palabra, ya que en ocasiones es necesario como tener el plural del nombre de una criatura o su femenino en caso del español. Estas funciones son las que hacen consultas para sustantivos, adjetivos, verbos y adverbios. También cuenta con funciones que devuelven un *String* simple como determinantes y preposiciones. A estas palabras se les crearon unos recursos a mano debido a que las *Wordnet* no las manejan.

Se definió como interfaz ya que no todos los idiomas van a requerir la misma información de una palabra, por ejemplo, en inglés no existen las variantes de género de adjetivos [61] como en el español. Así se obtiene el *WordDataGetter* adecuado para un idioma. Es por esto, que es muy importante al trabajar con estas funciones el controlar en todo momento las entradas que se les pasa y las salidas que generan ya que en ocasiones las peticiones al *WordDataGetter* se realizan fuera del generador de texto, donde no se controla el idioma empleado. Estas situaciones, son el nombramiento de criaturas e items y su asociación características estéticas. Esto es para lo que sirven los comentados *arrays* de *Strings* que hay en las funciones de las factorías. Son diversas llaves de los *json* de recursos léxicos, permitiendo así al *WordDataGetter* seleccionar una de esas llaves de forma aleatoria, y luego obtener esa palabra, o un sinónimo, con toda la información necesaria.

Después del *WordDataGetter*, tocaba ponerse con el *Realizator* que sería el encargado de agrupar las palabras según el tipo de frase a formar y las restricciones morfológicas de las palabras. Los tipos de frase se definieron como unos modelos en dos *json*: *templates.json* y *PhraseComponents.json*. El primero definiría diferentes tipos de frase para una situación indicando los componentes presentes. A continuación, el código correspondiente a las plantillas de las frases en español:

```
1  {
2    "BasicActionsTemplates":[
3      "SUJ VB CD ADV"
4    ],
5    "ThrowAttack":[
```

```
      "SUJ VB CD CI ADV"
    ],
    "WeaponsAttacks":[
      "SUJ VB CI ADV",
      "SUJ VB CI CC ADV"
    ],
    "ChangeDungeonLevel":[
      "SUJ VB CD CC"
    ],
    "ToBeInTemplate":[
      "VB CD ADV"
    ],
    "ToBeSomething":[
      "SUJ VB ATR"
    ],
    "MostBasicTemplate":[
      "SUJ VB"
    ],
    "CollideWall":[
      "SUJ VB CC"
    ]
  }
```

Como se puede ver, las plantillas van identificadas por una llave que depende del tipo de situación a describir. *"BasisAtionsTemplates"* se usa para acciones bastante comunes como beber o comer un objeto. Luego hay otras como *"WeaponsAttacks"* en la que se definen dos frases, así no siempre se indicará el objeto con el que se realizó el ataque. También hay que mencionar que en *"BasicActionsTemplates"*, el componente *ADV* no siempre va a salir, esto se decidirá según la situación.

Las componentes han sido nombradas según los términos de la sintaxis española, mas en la práctica, pueden ser tratados sin problema al trabajar en otros idiomas. Los componentes empleados son:

- **SUJ.** Sujeto.
- **VB.** Verbo.
- **CD.** Complemento directo.
- **CI.** Complemento indirecto.

- **CC.** Complemento circunstalcial, el cual puede ser de lugar o instrumento en el juego.
- **ADV.** Adverbio.
- **ATR.** Atributo.

Mientras, en *PhraseComponents.json* se definen las partes de un componente de las frases de *templates.json*. Aquí se pueden ver las componentes definidas para el español:

```
{
  "SUJ":[
    "ART NOUN",
    "ART NOUN ADJ"
  ],
  "CD":[
    "ART NOUN",
    "ART NOUN ADJ"
  ],
  "CD_be_space":[
    "DetUn NOUN ADJ"
  ],
  "CI":[
    "PR ART NOUN",
    "PR ART NOUN ADJ"
  ],
  "CCI":[
    "PR ART NOUN",
    "PR ART NOUN ADJ"
  ],
  "CCL":[
    "PR ART NOUN ADJ"
  ],
  "ATR":[
    "DetUn NOUN ADJ",
    "DetUn NOUN"
  ]
}
```

Se pueden observar dos cosas en este código. Lo primero, son los elementos atómicos como*NOUN* y *PR*, que son identificados y manejados ya dentro del código. Luego, que hay dos

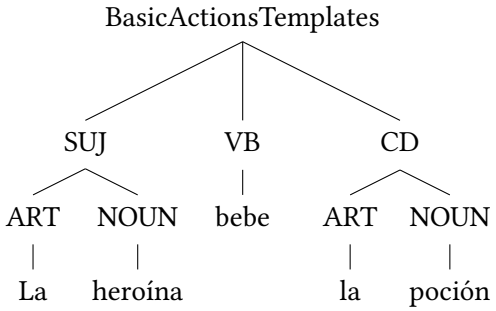
keys CD y esto es debido a la necesidad de emplear el determinante adecuado, por ejemplo, en ”La heroína bebe la poción” y en ”Hay una poción verde aquí” tenemos dos complementos directos pero con artículos muy diferentes.

Estos elementos atómicos son:

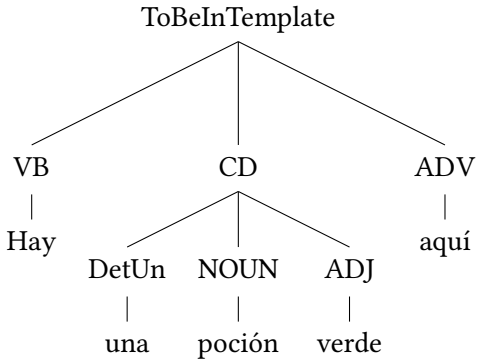
- **ART.** Artículo.
- **NOUN.** Nombre.
- **ADJ.** Adjetivo
- **PR.** Preposición.
- **DetUn.** Determinante indefinido.

Para comprender mejor esta organización, representaré las dos frases del anterior párrafo en un árbol sintátcio que es como lo percibe el programa.

- ”La heroína bebe la poción.”:



- ”Hay una poción verde aquí.”



Ahora es preciso hablar de las restricciones morfológicas que el *Realizator* debe respetar. Estas se almacenan en la clase *Restricctions* la cual tienen atributos para cada posible restricción como género y número del sujeto de la frase. Debido a la gran cantidad de entradas que requiere el constructor de *Restriccitions*, se implementó *RestricctionsFactory*, un *singleton* en el cual están definidas funciones para construir *Restricctions* pero empleando las entradas que reciba la función, el resto serían *null* para el constructor.

Ya explicados los *templates* y la clase *Restricctions*, es necesario hablar de dónde y cómo se llama al *Realizator*, qué entradas se le pasan y qué hace con estas.

Como ya se ha comentado, los *templates* representan acciones y es que *Realizator* es llamado desde los elementos de clase *Creature*. En las funciones de las acciones, se han añadido variables del tipo *HashMap* en las cuales se definen las caracteríctcas de los componentes de la acción, como el tipo de verbo a emplear (*actionType*) y el sujeto. Algunos de estos datos, como el "número", son escritos directamente en el código debido a que depende de la situación. Esto se puede ver bien en el caso de "invocar murcielagos" frente a la acción de "beber". En el primer caso, el objeto directo es plural siempre mientras que en el otro va a ser singular, no se pueden beber dos pociones a la vez.

Una vez definidos los *HashMaps* de los componentes de la acción, se define también el *Restricctions* correspondiente a la acción, y se pasan todos estos a la función *doActionComplex* junto a un *String* que indica el tipo de *template* que se deberá cargar. Dentro de esta función, se hace la llamada a la función *realizatePhrase* del *Realizator* para obtener la oración final. Una vez obtenida, pasará a ser notificada.

Una vez llamada la función *realizatePhrase*, lo primero que se hace es cargar el *template* indicado por la entrada *templateType* y, también, se carga una instancia del *WordDataGetter* ya que hay algunas palabras, como determinantes, que no vienen dadas en la entrada y deben decidirse en el *Realizator*. A continuación, determina la llave que identifica a los verbos que se pueden usar en la frase pero, teniendo en cuenta la presencia del componente *CC* que representa en español al *Complemento Circunstancial*. Si este tiene una llave equivalente al *String "CCI"*, significa que en la frase se ha usado una herramienta por lo que la llamada al *WordDataGetter* incluirá una entrada en la que se especifica la herramienta usada. Esto permite elegir un verbo en base al utensilio usado, como "ensartar" y otros sinónimos cuando se usa un puñal. En otro caso, se cargará el verbo directamente de *screenActions-verbs.json* en el cual se corresponden el tipo de acción con el verbo a emplear.

Una vez obtenida la llave del verbo, se obtiene el verbo final y sus características con el *WordDataGetter* y se llama a phraseConsructor. Es en esta función donde se comprueba la presencia de los posibles componentes en la plantilla cargada. Si el componente está presente, se llama a *getObjectTemplate* para obtener el template del componente, y luego se llama a la función encargada de construir el componente pasándole por entrada la plantilla, el componente y las restricciones. A medida que se van construyendo los componentes como *Strings*, estos se irán añadiendo a un *HashMap* en el que irán en par con el identificador del componente. Al acabar, serán devueltos a *realizatePhrase*, donde finalmente serán organizados gracias al orden de los componentes en la plantilla de la frase y unidos en un *String*, la oración final.

A medida que avanzó la implementación, se añadieron nuevas funcionalidades extras al *WordDataGetter* y al *Realizator*, haciendo que el primero fuera también responsable de obtener

las traducciones completas de algunas frases almacenadas en los recursos y el segundo que construyera nombres compuestos o posesivos.

Para acabar, estaba el problema del idioma del juego para el cual se aplicaron los patrones *facade*, *factory* y *singleton*. Primero se establecieron como *singleton* los *WordDataGetter* y los *Realizator*. Seguidamente, se implementó *WordDataGetterAndRealizatorFactory* una *factory singleton* que obtendría las instancias de *WordDataGetter* y los *Realizator* según el idioma determinado en el archivo de configuración de idiomas. Con esto, el juego no tendría que distinguir entre un *WordDataGetter* o un *Realizator* en su codificación teniendo aquí el ejemplo del patrón *facade*. Toda esta relación se puede ver en el diagrama de la Figura 4.11 (página 51) y un ejemplo de los juego en inglés en la Figura 4.12 (página 52).

Figura 4.11: La facotría *WordDataGetterAndRealizatorFactory* con las interfaces *WordDataGetter* o y *Realizator* y las implementaciones de estas.

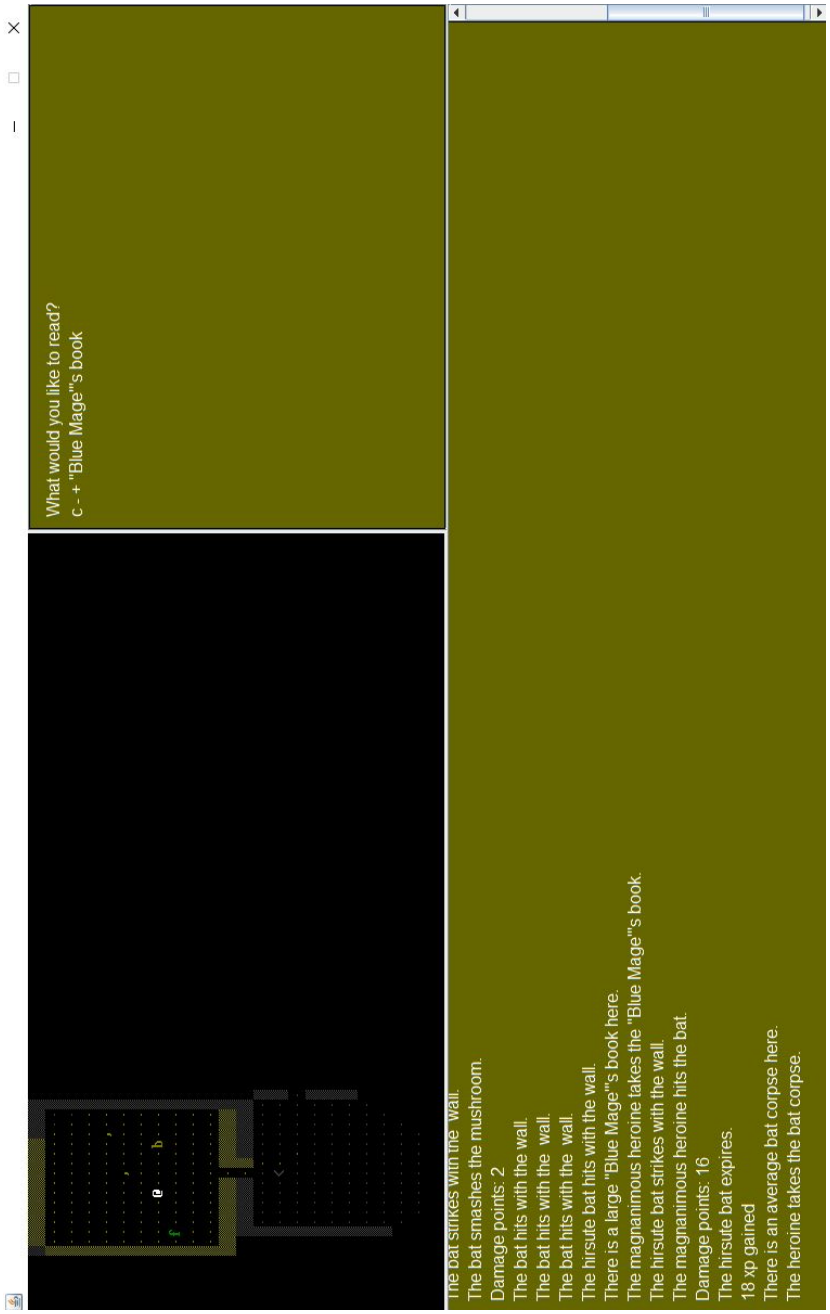


Figura 4.12: Se puede apreciar la localización incluso en los textos generados

Capítulo 5

Conclusiones y trabajo futuro

5.1 Conclusiones

A este día, se han completado la implementación del juego, su interfaz compatible con el lector de pantalla y el generador de textos, así como la internacionalización del juego. El proyecto ha demostrado ser bastante complejo debido a que combina diversas doctrinas y no las usa de una forma sencilla. Paralelamente, también ha repercutido mucho en mí como desarrollador, ya que he ampliado y mejorado mis habilidades y mis conocimientos respecto a varias tecnologías como *Java* y recursos como *Wordnet*. Debido a la complejidad del proyecto, también se ha ampliado mi experiencia en el uso de patrones de diseño dando al juego un carácter maleable.

Siento orgullo principalmente por dos componentes del juego: el constructor de mazmorras y el generador de texto. El desarrollo de estos también fue algo complicado, pero creo que denotan un buen diseño y funcionamiento, de tal modo que ahora añadir una nueva frase para una acción resulta muy sencillo e incluso puede no requerir cambio alguno en los recursos del generado de texto.

Con esto, siento que el trabajo ha merecido la pena y que el esfuerzo y tiempo dedicados han dado sus frutos por mucho que haya costado llegar hasta aquí.

5.2 Trabajo futuro

Como siempre ocurre en proyectos de este tipo, varias de las ideas que fueron surgiendo durante su desarrollo no se llegaron a materaializar por limitaciones de tiempo. No obstante, la estructura del juego es potente y flexible por lo que su implementación en futuras ampliaciones debería resultar sencilla. Algunas de estas ideas de cara a desarrollos futuros son:

- **Localización al japonés.** Uno de los planes iniciales que también se vio relegado por restricciones de tiempo fue la localización a este idioma. Sin embargo, particularidades

específicas de la *Wordnet* japonesa lo impidieron. Dicho esto, debemos recordar que nuestro generador de texto está pensando para que pueda manejar varios idiomas, así que su inclusión no debería ser tan difícil disponiendo de tiempo.

- **Configuración más accesible y diversa.** Actualmente las opciones del juego sólo se pueden modificar ambiando los valores en los archivos de configuración. Esto no es cómodo y debería poder realizarse desde el juego con una buena interfaz. También me gustaría añadir nuevas opciones que enriquecerían el juego, tales como modificar la dificultad de la partida o cambiar el tamaño y estilo de la fuente.

- **Ofrecer diferentes tipos de mazmorras.** En el juego podemos disfrutar de un tipo de mazmorra clásica, pero podríamos ofrecer otras que aporten no sólo cambios de localización, si no también en los elementos y en la propia estructura de esta. Por ejemplo, podría añadirse una nueva mazmorra ambientada en una zona helada en la que el jugador debería preocuparse de rejillas especiales que le den una penalización de movimiento por congelación.

- **Comportamientos más complejos.** Mediante mejoras de la *IA* del juego se podría añadir un ladrón que se dedicase a buscar el amuleto al igual que el jugador y que, si logre escapar de la mazmorra con este, el jugador pierde la partida. Otra posibilidad interesante sería definir un comportamiento entre las criaturas que, cuando estén en peligro, decidan desplazarse para buscar criaturas de la misma clase para que le den apoyo. Este comportamiento cooperativo también podría definirse para alguna criatura que pueda colaborar con el jugador.

- **Efectos de sonido.** Una de las ideas más interesantes y que más beneficiarían a los usuarios del juego. En este apartado también surgió la idea de implementar una base de datos de sonidos, similar a la *Wordnet*, lo que permitiría tener a los sonidos agrupados por sus significados y contextos.

- **Estilo del narrador.** Actualmente el generador de texto emplea la tercera persona, activa del presente a la hora de realizar las descripciones. Añadir nuevas formas y tiempos requeriría definir nuevas plantillas y revisar bien los recursos de vocabulario, especialmente los verbos.

- **Selección de palabras por estadísticas.** Ahora mismo el *WordDataGetter* obtiene las palabras desde unos recursos *json* que contienen *arrays* de sinónimos, devolviendo a una de ellas escogida aleatoriamente. La idea sería modificar el mecanismo de selección para que se tengan en cuenta las estadísticas de uso de estas palabras en el idioma; es decir, favorecer a aquellas cuyo uso sea más común. Así, el lenguaje empleado por el juego sería más natural al usuario.

- **Registro de la partida.** Como se ha explicado, el juego únicamente muestra mensajes sobre lo que ha sucedido durante un turno y en el momento que se pasa a otro, se borra. Sería de ayuda ofrecer un registro en el que se guarden los mensajes que se han mostrado y permitir su acceso desde el juego en cualquier momento.

Apéndices

Apéndice A

Herramienta *Wordnet*

A.1 Introducción

En este apéndice se tratará de forma más detallada la herramienta empleada para la creación de los recursos léxicos del juego. Como ya se explicó, en principio esta funcionalidad iba a estar incorporada en el juego y se ejecutaría al realizar la instalación de este. Esto demostró no estar al alcance de nuestro proyecto, por lo que finalmente se descartó y se dejó a este como un medio independiente que se puede emplear libremente para crear los recursos que se necesiten. Es importante saber que esta herramienta no consta de ninguna interfaz gráfica para su uso. Si se quiere emplear, se puede ejecutar su *JAR* o el código directamente en un *IDE* compatible, pero debe de hacerlo respetando las normas a la hora de modificar los recursos para el buen funcionamiento. El código está disponible en un repositorio *GitHub* [62].

A.1.1 Wordnet: qué es?

Primero, me gustaría explicar qué es una *Wordnet*, dado que se la ha mencionado muchas veces pero no se ha aclarado su naturaleza. Una *Wordnet* es una base de datos relacional, más concretamente, relaciones léxicas. En esta nos encontramos a las palabras agrupadas por sus significados, es decir, grupos de sinónimos. Estos grupos se los conoce como *synsets* y constan de un identificador único, un *Offset*. Estos conjuntos están relacionados en la *Wordnet* mediante relaciones semánticas, como por ejemplo para los sustantivos:

- **Hiperonimia.** En este caso, un *synset* cuyos términos son más genéricos que otros *synsets* y que se pueden utilizar para nombrar una misma realidad. Por ejemplo: ”animal” es hiperónimo de ”gato”.

- **Hiponimia.** Un *synset* posee en sus términos todos los rasgos semánticos de otro, pero es más específico. Ejemplo: ”diciembre” es un hipónimo de ”mes”.

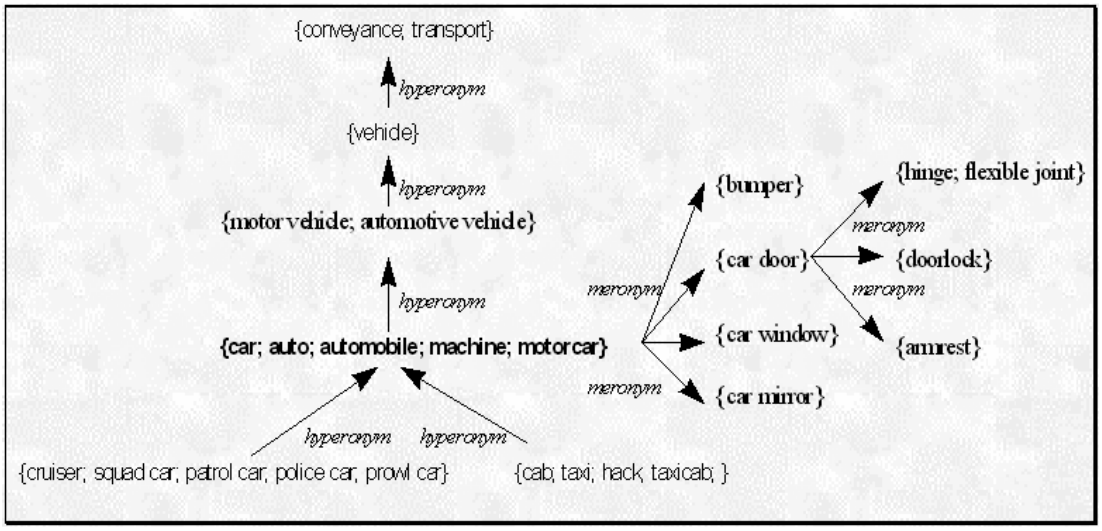


Figura A.1: En esta imagen se puede observar la clasificación que hace la *Wordnet* de las palabras. Este ejemplo se centra en el grupo de palabras {car; auto; automobile; machine; motorcar}, el cual se trata de un *synset*. Las diversas flechas en la imagen indican la dirección de las relaciones y están nombradas con los tipos de estas. Se puede ver que el *synset* principal es hiperónimo de {cab, taxi, hack, taxi, cab}, mientras que conveyance, transport son hiperónimos de este. Finalmente a la derecha, unas relaciones que se indican algunos *synsets* merónimos, como car door.

- **Holonimia.** En este caso, se hace referencia a una inclusión ”física” de los términos. Por ejemplo, ”escuela” es holónimo de ”clase” ya que las ”escuela” contiene ”clases”.

- **Meronimia.** El otro extremo de la relación de holonimia. En este caso, el *sysnset* es el incluido. Siguiendo con el ejemplo anterior, ”clase” es el merónimo de ”escuela”.

En la Figura A.1 (página 60) se puede ver un ejemplo de la agrupación y relación dentro de la *Wordnet*.

Aclarado el término, prosigue pues el apéndice, tratando primero la implementación de la herramienta en detalle, y después se indicaran los pasos a realizar para añadir alguna palabra de lla que se quiera obtener su información.

A.2 Implementación

En esta sección se tratará en diferentes apartados el curso de la implementación de la herramienta, entrado en detalle en las clases involucradas y sus funciones definidas.

A.2.1 Instalación de la *Wordnet* y obtención de *Offsets*

Lo primero fue obtener la *Wordnet MCR (Multilingual Central Repository) [41]*, el cual consiste en un directorio con diversas carpetas. Entre ellas, existe una llama *mysql* en la que se definen los términos de la instalación y hay dos archivos *sql* que al ejecutarse llenarían una base de datos vacía de todas las estructuras y recursos de la *Wordnet*. Por tanto, se instaló *MySQL*, se creó la base de datos y se procedió a la ejecución de los dos archivos *sql*. Una vez finalizada la inicialización, se procedió a realizar consultas para las palabras que interesaba tener algún *synset* determinado. Esto llevó mucho tiempo, ya que algunas palabras llegaban a tener más de 40 *synsets* asociados y había que comprobarlos uno a uno hasta encontrar el necesario. A medida que se iban encontrando los synsets adecuados, se iba anotando u *ili-Offset*, el identificador del *synset* dentro de todos los idiomas controlados por la *Wordnet*. Con este identificador, sería muy sencillo obtener sus traducciones en otros los idiomas.

A.2.2 Concultas a la *Wordnet* desde *Java*: *MySQLAccess*

Unas vez obtenidos los *Offsets* necesarios, se procedió a la implementación de código *Java* que fuera capaz de realizar las consultas a la *Wordnet*. Esto requirió la librería de *MySQL* de *Java*[40], la cual una vez obtenida se creó la clase *MySQLAccess*. Dentro de esta se implementó la función *wordnetQuery*, la cual recibiría por entrada el *iliOffset* de la palabra a buscar y un valor *boolean* con el que se determinaba el idioma a buscar. Esta función ejecuta múltiples operaciones necesarias para tratar la base de datos, muchas veces incorporando al *String* que representa la solicitud datos necesarios para la consulta. La mayor parte de estos son cargados desde un archivo *.properties*, mientras que el del idioma se decide por la entrada comentada.

Sobre el funcionamiento interno, el proceso simplemente realiza dos solicitudes: en la primera se pide el *Offset* que se obtiene del *ili-Offset* respecto al idioma que se esté tratando. Después, se hace la solicitud del *synset* identificado por el *Offset* obtenido. El resultado es devuelto como un *ArrayList* de *Strings*.

A.2.3 Construcción de Json a partir de las consultas: *JsonCreator*

Ya pudiendo realizar las consultas desde código, se procedió a implementar las clases que construirían los recursos estructurados con los resultados obtenidos. Para esto, se decidió definir una *interface JsonCreator*, dado que los recursos serían *json* estructurados de forma diferente según el idioma. Dentro de la interfaz, se nombró a cuatro funciones:

- **nounJson()**
- **adjJson()**
- **verbJson()**

- **advJson()**

Cada una debía de ser implementada de tal manera que se crease el *json* de las palabras correspondientes. Esto significa que cada una debe obtener el *ili-Offset*, pasar el *ili-Offset* a la consulta con la *Wordnet* para obtener la lista de palabras y, finalmente, escribirlas en el archivo *json* agrupadas correctamente.

Para esto, se ha creado previamente un archivo *json* por cada tipo de palabra, en el que están todos los los *ili-Offsets* identificados por una de las palabras que designa. Gracias a la librería *json.org*[38], se va iterando cada uno de los pares de este archivo y se va realizando una consulta. Cuando se acaba la consulta, se escribe en un archivo la palabra que identificaba al *ili-Offset* como llave de las palabras obtenidas desde la consulta. El archivo creado se localizará en el directorio *resources*, dentro de la carpeta del idioma que le corresponde. A continuación, se puede ver un ejemplo del *json* que contiene los *ili-Offsets* de sustantivos del juegos.

```
{
  "bat":"ili-30-02139199-n",
  "creature":"ili-30-00004475-n",
  "corridor":"ili-30-03895585-n",
  "potion":"ili-30-07883251-n",
  "fungus":"ili-30-07734744-n",
  "zombie":"ili-30-10805638-n",
  "rock":"ili-30-09416076-n",
  "goblin":"ili-30-09543748-n",
  "dagger":"ili-30-03158885-n",
  "sword":"ili-30-04054361-n",
  "staff":"ili-30-07267573-n",
  "tunic":"ili-30-04497570-n",
  "chain_mail":"ili-30-03000247-n",
  "plate_armour":"ili-30-02740764-n",
  "bow":"ili-30-02879718-n",
  "book":"ili-30-06410904-n",
  "tome":"ili-30-06413579-n",
  "perchment":"ili-30-14975779-n",
  "grimoire":"ili-30-06422032-n",
  "hero":"ili-30-05929670-n",
  "heroine":"ili-30-10173305-n",
  "health":"ili-30-14447908-n",
  "poison":"ili-30-15032376-n",
  "magic":"ili-30-05967977-n",
```

```
26      "warrior":"ili-30-10768585-n",
27      "stairs":"ili-30-04298171-n",
28      "corpse":"ili-30-05218119-n",
29      "level":"ili-30-03365991-n",
30      "body":"ili-30-05216365-n",
31      "wall":"ili-30-04546855-n",
32      "amulet":"ili-30-02706586-n"
33  }
34
35
```

A.2.4 Obtención de variaciones y datos morfológicos: *FindStr*

Finalmente, se observó que se necesitaban muchos datos y variaciones de estas palabras que simplemente no se podían obtener desde la *Wordnet*. Para solventar este problema, se decidió aprovechar el recurso *WikiCorpus*[42], dado que precisamente contiene las variaciones e información que se necesitan en forma de anotaciones. Estas anotaciones siguen un formato conocido como *Freeling* [63]. Este formato define etiquetas compuestas por caracteres que según su valor y posición, indican una característica morfológica de la palabra. Luego, el *WikiCorpus* indica las palabras ”base”, permitiendo así buscar conjugaciones de verbos. Para mayor comodidad, la palabra base y la etiqueta se encuentran separadas por tan sólo un espacio, favoreciendo así la búsqueda en estos textos anotados.

Ante esto, se probó la función de *Windows findstr*, para ver si había éxito obteniendo, por ejemplo, el femenino de un adjetivo. Funcionaba sin problemas hasta el momento que se tenían que tratar caracteres que no aparecen en el alfabeto inglés. En primera instancia, esto es culpa de la propia función *findstr*, la cual debe leer el *String* a buscar desde un archivo, y escribir los resultados en otro para poder mostrar estos caracteres. Pese a esto, siguió fallando y se estuvo un tiempo dudando si cambiar de estrategia completamente. Finalmente, resultó que la culpa recaía en los propios archivos del *WikiCorpus* que no estaban codificados en *UTF-8* pese a contener texto en español. Una vez se convirtieron los archivos a *UTF-8*, no se tuvo más problemas en las consultas de *findstr*.

Con esto, se procedió a crear la *interface FindStr*, con la que se nombraban las funciones que obtendrían las variaciones y datos morfológicos de la palabra deseada. Se creó esta interfaz dado que los *Freelings* son diferentes según su idioma, no sólo en la información que aporta si no en su composición.

Internamente de una de estas clases implementadas, se tiene un atributo el cual es un *ArrayList* de *Strings* llamado *fileNames*. Este atributo se inicializa con la clase, guardando

los nombres de todos los archivos que hay que tratar. Luego, cuando se llama a alguna de las funciones *getWordData*, se crea una *query* juntando la palabra que se desea buscar con unas etiquetas de información complementarias para mejorar la búsqueda. A continuación, se escribe esta *query* en archivo y se pasa a iterar la *fileNames* con la función *findWordData*, que es donde se ejecuta la función *findstr*. Cuando la busqueda tenga algún resultado, se devolverá la línea completa en la que se encontró la *query* realizada. Para poder obtener lo necesario, esta se parte mediante los espacios en un *array* de *Strings*, se obtiene la palabra base de este *array* y se genera un array de *char* a partir del *string* que representa a la etiqueta. Los valores *char* se irán comprobado ir obteniendo los datos y palabras que se buscaban. Finalmente, los resultados son retornados a la clase *JsonCreator*, la cual irá almacenando con la etiqueta adecuada en un *json*.

Para finalizar, un diagrama de la implementación en la Figura A.2 (página 64).

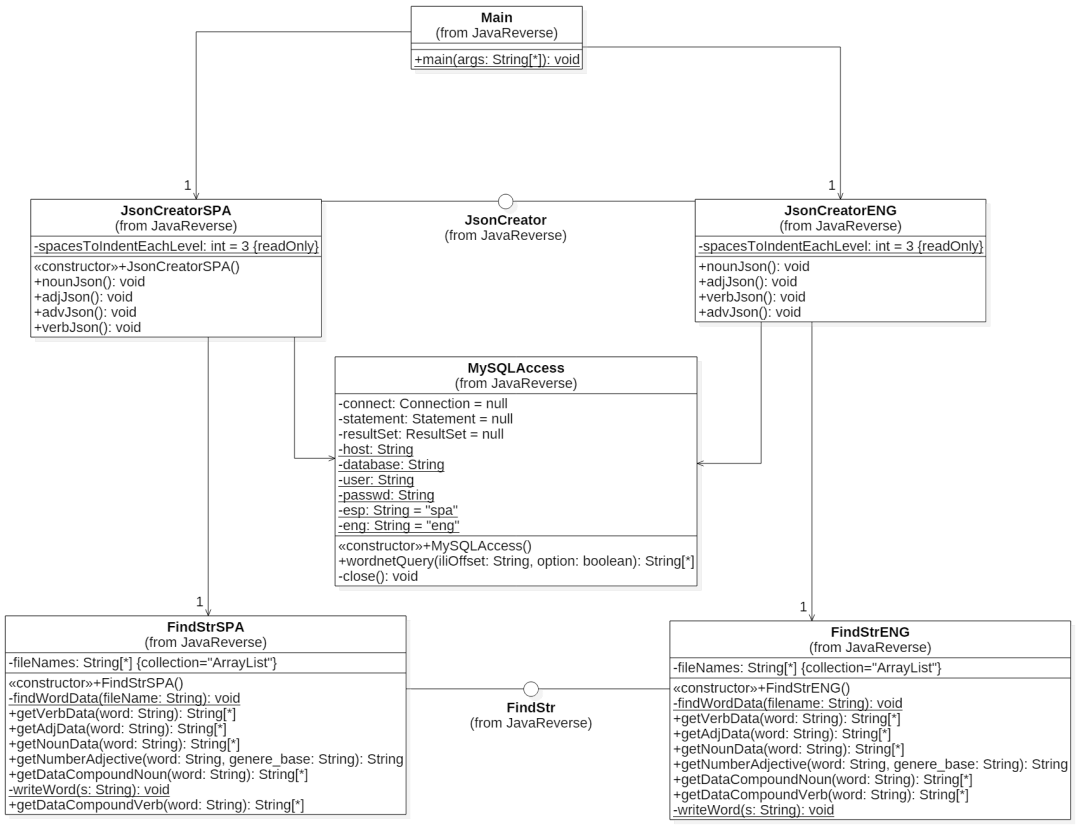


Figura A.2: Diagrama en el que se puede ver las relaciones entre las clases de la herramienta.

A.2.5 Conclusiones

Se ha conseguido implementar una herramienta que puede llegar a ser muy potente a la hora de realizar traducciones e incluso obtener detalles sobre la palabra. Aún así, la herramienta tiene múltiples problemas, como que los resultados no son totalmente fiables ya que ni los contenidos de la *Wordnet* ni del *WikiCorpus* son siempre exactos, por lo que requiere revisar los resultados siempre. También está el problema de que ahora mismo esta herramienta únicamente puede operar en *Windows* ya que depende de una función propia de este sistema operativo, y que esta función debe procesar un conjunto enorme de ficheros bastante grandes por lo que si la palabra no está presente se habrá desperdiciado mucho tiempo. Tampoco cuenta con una implementación muy buena, ya que se podrían aprovechar las *interface* para hacer *facade* con las clases que las implementan, de una forma similar a la que hago en la implementación del juego.

Pese a todo, creo que tiene un potencial enorme y me gustaría darle los retoques necesarios para resolver todos esos problemas y dejarla libre al publico para que la use a su parecer.

A.3 Pasos para añadir un *synset* a los recursos

1. Determinar el *ili-Offset*, por lo que es necesario que se busque dentro de la *Wordnet*.
2. Abrir el archivo contenido en el directorio *ILI data* según el tipo de palabras que haya en el *synset*.
3. Añadir el nuevo *ili-Offset* con alguna de las palabras de su *synset* como identificador.
4. Poner en el *Main* la función de *JsonCreator* en base al tipo de palabras que se van a tratar y el idioma a usar.
5. Ejecutar el *Main*. Los resultados se escribirán en el archivo correspondiente al tipo de palarabs que se tratraron y el idioma.

Apéndice B

Manual del juego

En este apéndice se describe el uso y la instalación del juego. La parte de manual está centrada a jugadores novicios en el género del roguelike. El manual está definido respecto a la versión 1.0, la que existe en el momento de entrega de este proyecto. Por eso, es importante saber que este juego no está acabado aún, así que se ruega al usuario comprensión. El proyecto seguirá expandiéndose con nuevas criaturas, items y otras características.

¡Muchas gracias por probar el juego!

B.1 Instalación

El código fuente y el *release 1.0* están disponibles en *Github* [64]. El *release* incluye un ejecutable *.jar* al que con darle click se inicia el juego. Para seleccionar el idioma se debe modificar el valor en el archivo *language.properties* y para cambiar el número de niveles se debe modificar el *dungeon.properties*. En el primero los valores pueden ser *ENG* para seleccionar inglés o *SPA* para español. El número mínimo de niveles en la mazmorra es 2, si se pone menos se generarán 5 niveles por defecto.

Para poder emplear un lector de pantalla en *Windows*, es importante tener activado el *Java Access Bridge* y comprobar que se tiene instalada la versión completa del mismo. Lo recomendable es tener las versiones para 32 y 64 bits instaladas. Para activarlo, es necesario realizar la configuración desde el panel de control:

1. Acceder al Panel de control.
2. Seleccionar la opción de Accesibilidad.
3. Después, seleccionar Centro de Accesibilidad.
4. Dentro del la pantalla, dar click a Facilitar uso del equipo.

5. En esta pantalla, ir a la sección Otros programas instalados, en la que se debería checkear la opción del *Java Access Bridge*.

También se puede activar desde el terminal *Windows* mediante el comando:
`\%JRE_HOME\%\bin\jabswitch-enable`
En el que *%JRE_HOME%* es el directorio en el que se encuentra instalado *Java*.

B.2 El juego

"Desciende por las mazmorras de Uqbar, encuentra el amuleto perdido de Hamsun, y regresa a la superficie para ganar. Emplea lo que encuentres para evitar la muerte."

El juego es un roguelike de corte clásico en el que hay que descender por una mazmorra generada proceduralmente. El objetivo es llegar al fondo del laberinto, obtener el amuleto, y regresar a la salida del primer nivel. Esta mazmorra es por defecto de 5 plantas que constaran con varias habitaciones conectadas por pasillos. Mientras, los niveles de esta serán conectados por escaleras y en el primer piso se encontrará una escalera ascendente que es la salida de la mazmorra y donde el jugador empieza la partida. El juego se desarrolla por turnos, esto es, que cada vez que el jugador haga una acción se dará lugar el turno por lo que el resto de elementos de la mazmorra también harán sus acciones.

El jugador debe jugar estratégicamente dado que los enemigos pueden ser muy fuertes y, al desarrollarse en turno, se podrá tardar lo que uno quiera para tomar una decisión.

B.3 El héroe y sus acciones

El jugador controlará a un héroe/heroína que podrá desplazar por la mazmorra empleando las teclas:

U para desplazarse hacia norte.

J para desplazarse hacia el sur.

H para desplazarse hacia el oeste.

K para desplazarse hacia el este.

Las teclas de dirección son empleados por los *JTextArea* para que el lector de pantalla pueda desplazarse por los textos mostrados. Este personaje consta de cualidades que podrá incrementar al subir de nivel. Estas características son:

- Salud del personaje: que representa la cantidad de daño que puede recibir hasta morir.
- El maná: un valor de energía necesario para lanzar hechizos.

- Ataque: determina la cantidad de daño base que hará el personaje.
- Defense: valor que reduce el daño recibido.
- Percepción: controla la distancia máxima a la que el personaje puede detectar elementos.
- Regeneración de maná: a medida que pasen turnos, se va recuperando algo de maná. Este valor determina la cantidad que se recupera.
- Regeneración de salud: a medida que pasen turnos, se va recuperando algo de salud. Este valor determina la cantidad que se recupera.

Para el combate cuerpo a cuerpo, el personaje simplemente deberá moverse hacia donde se encuentre la otra criatura y al entrar en contacto con esta se producirá el daño. El personaje cuenta con otras múltiples acciones que se comentan en el siguiente sub-apartado:

B.3.1 Controles

- **[g] o [,] para recoger.** Permite al personaje recoger objetos del mapa, aunqu hay un límite de objetos que se pueden llevar en el inventario.
- **[d] para soltar.** Permite al personaje dejar en una casilla cerca uno de los objetos que llevaba.
- **[e] para comer.** Permite al personaje consumir un objeto del tipo comida, normalmente cuerpos de los enemigos.
- **[w] para equiparse.** Permite al personaje equiparse una armadura o una espada. Sólo puede llevar una de cada clase a la vez.
- **[?] para ayuda.** Muestra una pantalla en la que se pueden revisar los controles
- **[x] para examinar tus items.** Permite comprobar las características de los items que lleva el jugador.
- **[Ctrl] para mirar alrededor.** Se describe al jugador todo lo que hay al alcance de su percepción y permite comprobar las posición individualmente con un puntero.
- **[f] para disparar un proyectil.** El personaje podrá disparar con un arma a distancia, como un arco.
- **[t] para lanzar un objeto.** Se selecciona un objeto del inventario y se lanza a una posición seleccionada mediante un puntero. Puede servir como ataque.
- **[q] para beber una poción.** Con esta acción se podrá ingerir una pócima.

- **[r] para leer algo.** Leer libros de magia para lanzar hechizos.
- **[<] para subir escaleras.** Volver al nivel anterior de la mazmorra.
- **[>] para bajar escaleras.** Descender al siguiente nivel de la mazmorra.

B.4 Items

B.4.1 Armaduras y armas

Durante el juego, el jugador se encontrará con múltiples armas y armaduras. Los primeros dan bonificación de defensa, mientras que las armas dan de ataque aunque depende del tipo de ataque en el que se usen. Las armas son diversas, ofreciendo cetros, puñales, espadas… cada una con sus cualidades. Una arma especial es el arco, el cual permite atacar a enemigos desde la distancia, mas esta se irá desgastando, perdiendo fuerza hasta romperse.

B.4.2 Consumibles

Estos items pueden ser comida o pociones. Los primeros suelen recuperar vida y ser los cuerpos de los enemigos derrotados. Los segundos se encuentran por toda la mazmorra y poseen diversos efectos. Estos efectos pueden ser positivos o negativos, pero no se puede saber hasta que no se haya tomado la poción ya que en principio únicamente se las conocer por el color.

B.4.3 Tomos de magia

Estos items permiten elegir un hechizo de los que contienen y lanzarlo a un objetivo seleccionado con un cursos. Hay dos tipos de tomos: el blanco que está centrado a sanación y el azul que está orientado a confundir a los enemigos. Es importante saber que, para lanzar un hechizo, es necesario tener los puntos de maná indicados por este.

Bibliografía

[1] "Resolución de 18 de marzo de 2009, de la dirección general de trabajo, por la que se registra y publica el xvi convenio colectivo estatal de empresas de consultoría y estudios de mercado y de la opinión pública." 2009. [En línea]. Disponible en: https://www.boe.es/buscar/doc.php?id=BOE-A-2009-5688

[2] M. Barton and B. Loguidice, "The history of rogue: Have @ you, you deadly zs," 2009. [En línea]. Disponible en: https://www.gamasutra.com/view/feature/4013/the_history_of_rogue_have__you_.php

[3] M. Authors, "Berlin interpretation." [En línea]. Disponible en: http://www.roguebasin.com/index.php?title=Berlin_Interpretation

[4] Slash, "A classic roguelike," 2014. [En línea]. Disponible en: https://blog.roguetemple.com/roguelike-definition/

[5] ——, "What is a traditional roguelike," 2018. [En línea]. Disponible en: https://blog.roguetemple.com/what-is-a-traditional-roguelike/

[6] A. M. D'Argenio, "Statistically, video games are now the most popular and profitable form of entertainment," 2018. [En línea]. Disponible en: https://www.gamecrate.com/statistically-video-games-are-now-most-popular-and-profitable-form-entertainment/20087

[7] L. Lanier, "Videogames could be a $300 billion industry by 2025," 2019. [En línea]. Disponible en: https://variety.com/2019/gaming/news/video-games-300-billion-industry-2025-report-1203202672/

[8] B. Crecente, "Nearly 70% of americans play video games, mostly on smartphones," 2018. [En línea]. Disponible en: https://variety.com/2018/gaming/news/how-many-people-play-games-in-the-u-s-1202936332/

[9] B. Gilbert, "The biggest game company in the world isn't nintendo — it's a chinese company that has a piece of everything from 'fortnite' to 'league of legends'," 2019. [En línea]. Disponible en: https://www.businessinsider.com/what-is-tencent-games-explainer-2019-8?IR=T

[10] C. M. PÉREZ, "Ps4 ha llegado a los 100 millones más rápido que cualquier otra consola," 2019. [En línea]. Disponible en: https://vandal.elespanol.com/noticia/1350725203/ps4-ha-llegado-a-los-100-millones-mas-rapido-que-cualquier-otra-consola/

[11] N. Summers, "Nintendo switch sales continue to thrive," 2019. [En línea]. Disponible en: https://www.engadget.com/2019/07/30/nintendo-switch-mario-maker-2-sales/

[12] T. Wijman, "The global games market will generate $152.1 billion in 2019 as the u.s. overtakes china as the biggest market," 2019. [En línea]. Disponible en: https://newzoo.com/insights/articles/the-global-games-market-will-generate-152-1-billion-in-2019-as-the-u-s-overtakes-china-as-the-biggest-ma

[13] P. Cao, "Pokémon go brings in $2.45 billion in revenue, 550 million downloads globally," 2019. [En línea]. Disponible en: https://9to5mac.com/2019/03/07/pokemon-go-revenue-downloads/

[14] A. González, "Google stadia: Todos los detalles - lanzamiento, requisitos, juegos y características," 2019. [En línea]. Disponible en: https://vandal.elespanol.com/reportaje/todo-sobre-stadia-el-futuro-de-los-videojuegos-de-google-fecha-de-lanzamiento-requisitos-caracterist

[15] A. Bradley, "How important will ray tracing be for sony's ps5 and xbox's project scarlett?" 2019. [En línea]. Disponible en: https://www.gamesradar.com/how-important-will-ray-tracing-be-for-sonys-ps5-and-xboxs-project-scarlett/

[16] D. Hardawar, "Valve index review: Next-level vr," 2019. [En línea]. Disponible en: https://www.engadget.com/2019/06/28/valve-index-review-vr/

[17] J.-P. Dyson, "The influence of dungeons and dragons on video games," 2011. [En línea]. Disponible en: https://www.museumofplay.org/blog/chegheads/2011/05/the-influence-of-dungeons-and-dragons-on-video-games

[18] Adell, "Historia de los videojuegos: Los orígenes del género rpg," 2019. [En línea]. Disponible en: http://www.destinorpg.es/2015/08/historia-de-los-videojuegos-los.html

[19] M. Fahey, "The many, many deaths of pc gaming," 2010. [En línea]. Disponible en: https://kotaku.com/the-many-many-deaths-of-pc-gaming-5674097

[20] M. Authors, "Dwarf fortress," 2019. [En línea]. Disponible en: http://www.bay12games.com/dwarves/

[21] ——, "Steam." [En línea]. Disponible en: https://store.steampowered.com/?l=spanish

[22] ——, "Good old games." [En línea]. Disponible en: https://www.gog.com/

[23] ——, "Talking heads: Simulacra," 2008. [En línea]. Disponible en: http://www.haskins.yale.edu/featured/heads/SIMULACRA/kempelen.html

[24] ——, "Rafígrafo de fouçault," 2019. [En línea]. Disponible en: http://museo.once.es/home.cfm?id=42&nivel=1&detallep=131

[25] X. D. Riera, "Historia de la tiflotecnología en españa," 2010. [En línea]. Disponible en: http://www.nosolousabilidad.com/articulos/tiflotecnologia.htm

[26] M. Authors, "Nv access." [En línea]. Disponible en: https://www.nvaccess.org/

[27] ——, "Lector de pantalla orca." [En línea]. Disponible en: https://help.gnome.org/users/orca/stable/

[28] J. Clemens, "The oregon trail computer game," 2018. [En línea]. Disponible en: https://mncomputinghistory.com/oregon-trail-computer-game/

[29] E. S. Raymond, "A brief history of colossal cave adventure," 2019. [En línea]. Disponible en: http://www.catb.org/~esr/open-adventure/history.html

[30] H. Evarts, "For blind gamers, equal access to racing video games," 2018. [En línea]. Disponible en: https://engineering.columbia.edu/press-releases/rad-blind-video-games

[31] M. Authors, "Audiomgames," 2019. [En línea]. Disponible en: https://audiogames.net/

[32] C. Fisher, "Microsoft designs an xbox controller with braille," 2019. [En línea]. Disponible en: https://www.engadget.com/2019/05/07/microsoft-patent-braille-controller/

[33] A. Pepers, "Accessibility - alexei pepers," 2016. [En línea]. Disponible en: https://www.youtube.com/watch?v=Y_8sHtr62Bo

[34] K. Sutherland, "Playing roguelikes when you can't see," 2017. [En línea]. Disponible en: https://www.rockpapershotgun.com/2017/04/05/playing-roguelikes-when-you-cant-see/

[35] M. Authors, "Package javax.swing." [En línea]. Disponible en: https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

[36] ——, "Java se desktop accessibility." [En línea]. Disponible en: https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136191.html

[37] ——, "Necklace of the eye," 2016. [En línea]. Disponible en: http://roguetemple.com/z/noteye/

[38] S. Leary, "Json in java [package org.json]," 2019. [En línea]. Disponible en: https://github.com/stleary/JSON-java

[39] T. Spangler, "Asciipanel," 2018. [En línea]. Disponible en: https://github.com/trystan/AsciiPanel

[40] M. Authors, "Mysql connectors." [En línea]. Disponible en: https://dev.mysql.com/downloads/connector/

[41] ——, "Multilingual central repository," 2016. [En línea]. Disponible en: http://adimen.si.ehu.es/web/MCR

[42] ——, "Wikicorpus, v. 1.0: Catalan, spanish and english portions of the wikipedia." [En línea]. Disponible en: https://www.cs.upc.edu/~nlp/wikicorpus/

[43] ——, "The mit license." [En línea]. Disponible en: https://opensource.org/licenses/MIT

[44] ——, "The json license," 2002. [En línea]. Disponible en: https://www.json.org/license.html

[45] ——, "Apache license, version 2.0," 2004. [En línea]. Disponible en: https://www.apache.org/licenses/LICENSE-2.0

[46] ——, "Gnu general public license," 2007. [En línea]. Disponible en: https://www.gnu.org/licenses/gpl-3.0.en.html

[47] ——, "Gnu free documentation license," 2008. [En línea]. Disponible en: https://www.gnu.org/licenses/fdl-1.3.en.html

[48] ——, "License and commercial use of wordnet." [En línea]. Disponible en: https://wordnet.princeton.edu/license-and-commercial-use

[49] ——, "Attribution 3.0 unported (cc by 3.0)." [En línea]. Disponible en: https://creativecommons.org/licenses/by/3.0/

[50] H. M. Chandler and S. O. Deming, *The Game Localization Handbook*, 2nd ed. Jones & Barlett Learning, 2012.

[51] T. C. Smith and J. G. Cleary, "Probabilistic unification grammars," 1997. [En línea]. Disponible en: https://www.cs.waikato.ac.nz/~ml/publications/1997/TCS-JGC-Prob-Gram.pdf

[52] M. Authors, "Kyoto project." [En línea]. Disponible en: http://kyoto-project.eu/xmlgroup.iit.cnr.it/kyoto/index.html

[53] J. Smed and H. Hakonen, *Algorithms and Networking for Computer Games*, 1st ed. WILEY, 2006.

[54] A. Adonaac, "Procedural dungeon generation algorithm," 2015. [En línea]. Disponible en: https://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php

[55] D. Malyuta, "Fit circle through 3 points," 2016. [En línea]. Disponible en: https://es.mathworks.com/matlabcentral/fileexchange/57668-fit-circle-through-3-points

[56] C. Flanagan, "The bresenham line-drawing algorithm." [En línea]. Disponible en: https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html

[57] S. Ruder, "Word sense disambiguation." [En línea]. Disponible en: http://nlpprogress.com/english/word_sense_disambiguation.html

[58] M. G. C. P. R. M.-C. Gamallo, Pablo and J. C. Pichel, "Llinguakit," 2018. [En línea]. Disponible en: https://citius.usc.es/transferencia/software/linguakit

[59] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 2nd ed. PRENTICE HALL, 2008.

[60] M. Authors, "Prim's spanning tree algorithm." [En línea]. Disponible en: https://www.tutorialspoint.com/data_structures_algorithms/prims_spanning_tree_algorithm.htm

[61] D. Madhavan, "Grammatical gender," 2015. [En línea]. Disponible en: http://www.english-language-grammar-guide.com/grammatical-gender.html

[62] J. V. Letamendía, "Jsonlanguagestructuregenerator," 2019. [En línea]. Disponible en: https://github.com/JorViteri/JsonLanguageStructureGenerator

[63] M. Authors, "Spanish freeling part-of-speech tagset." [En línea]. Disponible en: https://www.sketchengine.eu/spanish-freeling-part-of-speech-tagset/

[64] J. V. Letamendía, "Hamsun-s-amulet," 2019. [En línea]. Disponible en: https://github.com/JorViteri/Hamsun-s-amulet