1. Introducción
2. Marco teórico o estado del arte
   1. Definición de RPG

RPG es el acrónimo de Rol Playable Game, en otras palabras, juego de rol.

Un rol es un papel, representación o interpretación de un personaje de forma imaginaria o interpretativa.

* 1. Videojuegos RPG

El RPG como videojuego es aquel en el que el jugador encarna a uno o varios personajes que atraviesan una serie de situaciones en las que el jugador tiene la capacidad de decidir (o no) el transcurso de la historia.

Las características principales de estos videojuegos vienen dadas por la adaptación de algunos elementos de los juegos de rol tradicionales o de mesa como por ejemplo, el desarrollo estadístico del personaje.

Normalmente, dentro del género RPG, los personajes controlados por el jugador pueden acumular puntos de experiencia y de esta forma, al llegar a un número determinado de puntos (50, 100,…), subir de nivel (junto con sus estadísticas). A su vez, los jugadores pueden conseguir dinero e intercambiarlo por bienes para su causa como pociones, comidas, armas, armaduras, habilidades,…

Actualmente, predomina la propuesta de videojuego donde se controla y representa cabalmente a un personaje (o varios), que debe cumplir con una serie de objetivos o misiones establecidos. Usualmente, se crea un mundo perteneciente a un tema de fantasía épica. Para ello, se viene utilizando una interfaz gráfica cada vez más vistosa para utilizar un sofisticado inventario de poderes humanos y sobrenaturales (que el jugador desarrolla poco a poco con práctica y muchas horas de juego), recursos monetarios y objetos diversos en propiedad (comprados o encontrados de manera fortuita), para el logro de las metas.

Ejemplos de estos videojuegos son: Final Fantasy, Dragon Quest, Bravely Default, Golden Sun, Chrono Trigger, The Legend of Zelda, Tales of…

* + 1. Géneros dentro de los videojuegos RPG
       1. JRPG

JRPG es como se conoce a los juegos de rol japoneses. Suele llamárseles de esta forma para diferenciarlos de los RPG occidentales.

Las diferencias entre un JRPG y un RPG occidental suelen ser:

* Los JRPG por lo general tienen unos gráficos estilo “anime” (al menos así era en sus orígenes, luego con el paso del tiempo se optaron por gráficos más realistas) mientras que por lo general, los RPG occidentales suelen caracterizarse por tener un mundo amplio y tramas más oscuras.
* Mientras que los JRPG tienen una progresión más lineal haciendo que tanto la historia como los personajes que en ella aparecen tengan una mayor profundidad. Por cambio, los occidentales, tienen historias más ramificadas que vienen dadas por las diferentes elecciones que hace el jugador a lo largo de la historia.
* Mientras que los RPG occidentales tienen unos combates en tiempo real, los JRPG están caracterizados por los combates por turnos de forma general.
  + - 1. Action RPG

Los Action RPG son videojuegos donde la acción y los combates en tiempo real tienen una mayor importancia.

Así pues, los combates son bastante frenéticos ya que hay acción constante. Y el jugador debe reaccionar a todo lo que ocurre mediante sus reflejos de una forma rápida para poder acabar con todos los enemigos.

* + - 1. Tactic o Strategic RPG

Este subgénero se caracteriza por tener varias unidades dispuestas sobre el terreno de batalla donde, mediante turnos, el jugador las irá moviendo mediante la rejilla (que es la característica que más distingue este tipo de juegos) para poder derrotar a la IA rival.

* + - 1. MMORPG

MMORPG son las siglas correspondientes a “Masive Multiplayer Online Role Playing Game”. Este tipo de juegos conectan a grandes cantidades de jugadores a la vez para que jueguen entre ellos. Ya sea o bien avanzando en la historia mediante mazmorras derrotando enemigos o bien para que peleen entre ellos.

* 1. Motores de videojuegos

Actualmente existe una gran variedad de motores de videojuegos, tanto gratuitos como de pago, de código abierto y cerrado e incluso motores propios de las empresas que solo son utilizados en su ambiente cerrado a la hora de desarrollar sus videojuegos.

* + 1. Unreal Engine



Unreal Engine[] es un motor de videojuegos programado en C++ y desarrollado por la compañía Epic Games (desarrolladores de juegos como Unreal Tournament y Gears of War). Actualmente en su versión 4, es un motor totalmente gratuito donde sólo hay que aportarles un 5% de los beneficios obtenidos con la comercialización del videojuego.

A nivel gráfico, es compatible con OpenGL, Vulkan y DirectX 11 y 12, lo que hace que pueda exportar a distintas plataformas como PC (Windows y Linux), MAC, Web(HTML5), dispositivos móviles(Android e iOS) y la mayoría de las consolas actuales (PS4, Xbox One y Nintendo Switch) además de dar soporte a VR.

Gracias a su gran potencia gráfica, también es usado fuera del ámbito de los videojuegos, como por ejemplo para realizar cinemáticas de animación.

En cuanto a funcionalidades, Unreal Enginge permite el scripting en C++ (previo desuso de su propio lenguaje UnrealScript similar a Java) lo cual apoya el sistema de Blueprints (bloques de código encapsulados que definen acciones y comportamientos) que permite hacer videojuegos sin tener muchos conocimientos sobre scripting. Tambien incluye todo tipo de editores: texturas, mapas, terrenos, partículas…

Por último, Unreal Engine tiene una amplia comunidad que se apoya mutuamente mediante la compartición de assets, plug-in, etc.

* + 1. GameMaker:Studio



GameMaker:Studio[] es una herramienta escrita en Delphi y un kit de desarrollo (SDK) pensada para la creación de videojuegos. Software gratuito (existe una versión comercial de pago con mayores características), esta pensado para aquellos desarrolladores novatos que tienen un conocimiento menor de programación. Aunque aquellos más experimentados, mediante el lenguaje de scripting propio del motor, Game Maker Language(GML), pueden personalizar aún más sus juegos y expandir enormemente sus características.

Los juegos creados con GameMaker:Studio pueden ser distribuidos bajo cualquier licencia sujeta a los términos del EULA de GameMaker y para cualquiera de las siguientes plataformas: PC (Windows), móviles(Andriod) y Web (HTML5).

La interfaz de GameMaker, al estar pensada para aquellos usuarios no familiarizados con el desarrollo de videojuegos, funciona con el método “drag&drop”. Siendo así muy intuitiva ya que aprovechando las bibliotecas de acciones que trae por defecto el motor (movimiento, dibujo básico…) simplemente arrastrando dichas acciones al marco de trabajo.

Como se ha mencionado anteriormente, GameMaker utiliza su propio lenguaje. Y aunque el motor esté pensado para realizar videojuegos en 2D, mediante el uso de este lenguaje, se pueden lograr grandes resultados en videojuegos 3D.

* + 1. Cry Engine



Cry Engine[] es un motor de videojuegos desarrollado por la compañía Crytek e introducido por primera vez en su videojuego Far Cry. El motor, al igual que Unreal Engine, tiene una potencia gráfica muy grande y es por eso que los resultados son muy vistosos. Está preparado para exportar a varias plataformas como PC, PS4, Xbox One y VR.

El motor es totalmente gratuito, sin ninguna obligación de aportar a Crytek ningún porcentaje de los beneficios obtenidos.

Cuenta con una gran comunidad y un amplio mercado de assets, pero en contraparte cuenta con una amplia curva de dificultad.

* + 1. Xenko Engine



Xenko Engine[] es un motor 2D y 3D de código abierto[] desarrollado en 2015 por la compañía nipona Silicon Studios (desarrolladores de varios videojuegos como la saga Bravely) capaz de exportar a múltiples plataformas (PC, iOS, Android, VR y Xbox One, con las siguientes versiones del motor vendrá la exportación al resto de consolas).

El software es gratuito, pero al igual que con motores como Unity o Unreal, tiene versiones en función de para que se usen (para desarrollos indie, para estudios profesionales, para estudios profesionales que buscan tener más flexibilidad y crear sus herramientas dentro del motor o una configuración del motor totalmente customizada).

A nivel de funcionalidades es un motor completo de altísimo nivel orientado a componentes escrito en lenguaje C# (soporta la última versión del lenguaje, 7.0), que ofrece unos resultados gráficos altísimos (compatibilidad tanto como con DirectX12 y Vulkan).

La herramienta principal dentro de Xenko es el llamado Game Studio que permite importar los assets, crear y organizar las escenas,crear los scripts, compilar y ejecutar los juegos. Además de todo esto viene equipado con varios editores: partículas, sprites, rendering,UI… también contiene el famoso sistema de prefabs (objetos de la escena creados en el editor que se guardan como assets)[].

* + 1. Motores de videojuegos RPG, RPGMAKER.

Tanta es la fama del género rpg que incluso se han desarrollado motores para crear videojuegos de éste género.

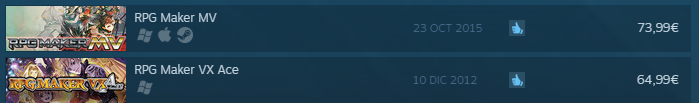
Ejemplo de ello es el motor RPG Maker, el cual cuenta con un gran número de revisiones y versiones que han ido saliendo con el paso de los años (desde 1988 hasta el día de hoy para ser exacto).

RPG Maker proporciona grandes facilidades al desarrollador tales como un editor de mapas, un editor de eventos y un editor de base de datos. A su vez, también incluye paquetes de assets como gráficos para mapeados, personajes, música, efectos de sonido, etc.

En su última versión RPG Maker MV aunque trae algunas funcionalidades nuevas como los nuevos “pluggins” para ayudar a los desarrolladores que no tengan muchos conocimientos de programación y algunas expansiones de tamaño de algunas bases de datos (como la de objetos que pasa de 999 a 2000 objetos) tiene algunas limitaciones o desventajas que cabe mencionar. Para empezar es de pago (73,99€) y actualmente cuenta con un modelo de negocio basado en “dlc”, es decir, expansiones de contenido descargable (nuevos assets, etc) también de pago. Pese haber ido avanzando de versión, el motor en sí no ha sido actualizado en mucho tiempo, es decir, sigue siendo lo mismo de siempre, aunque ofreciendo mejor resultado. Puede parecer que es poca cosa, pero al mirar en precio es algo a tener en cuenta puesto que la versión anterior RPG Maker VX Ace, no cambia mucho en cuanto a funcionalidad pero si en precio, ya que es mucho mas barato.

Además de eso, se arrastran problemas de versiones anteriores, por ejemplo:

si se desea cambiar una plantilla de las que ofrecer el motor, se debe hacer siempre desde la carpeta del programa.





* 1. Referencias
     1. Bravely Default y Bravely Second





Bravely Default y Bravely Second son JRPG desarrollados por la compañía Silicon Studio y publicados por Square Enix (en Japón) y Nintendo (en el resto del mundo) en 2012 y 2015 respectivamente. Ambos juegos forman parte de la familia de 3DS. Estos juegos han sido aclamados por el público, puesto que supone una vuelta a los antiguos JRPG con los elementos clásicos del género: historia que atrapa, personajes muy carismáticos, combates por turnos y el motivo por el cual es una de las principales referencias de este trabajo, el sistema de trabajos[] (presente también en otros juegos como Final Fantasy III).

* + 1. Final Fantasy 6 y 7





La saga Final Fantasy es famosa en todo el mundo con una gran trayectoria a lo largo de los años, y nos ha dejado grandes títulos como Final Fantasy VI y Final Fantasy VII. Juegos desarrollados y publicados por SquareSoft/Square.Co, conocida actualmente como Square Enix.

Estos dos juegos han tenido un gran impacto en la historia de los videojuegos por la narrativa de sus historias, en concreto el 7 que supuso toda una revolución en los videojuegos cuando salió al mercado.

Si bien es verdad que los JRPG son característicos por los combates por turnos, hay que decir que estos juegos tienen un componente que hace los turnos algo más dinámicos, y este es el ATB (Action Time Battle) que consiste en tener barras de tiempo que determinan cuando el jugador o la unidad enemiga puede hacer una acción en función de si dicha barra está llena o no.

* + 1. Digimon World DS



Digimon World DS es un videojuego de rol desarrollado por BEC y distribuido por Bandai Namco Games.

Este videojuego es bastante parejo a la saga RPG Pokémon, puesto que se basa en coleccionar y entrenar Digimon. A la hora de batalla, es su interfaz lo que se toma como referencia para este juego, ya que solo se ven los enemigos en la parte superior de la pantalla mientras que en la inferior solo se ven los retratos de los digimon empleados con sus datos de vitalidad y maná.

* + 1. Suikoden 2



Desarrollado y distribuido por Konami, Suikoden 2 es un JRPG para la primera Playstation (PSX) donde disponemos de todos los elementos propios del género, con la nota de que hay una gran cantidad de personajes obtenibles (108 por los 5,6 que solemos encontrar en la mayoría de JRPG). Y aunque el motivo por el cual es una referencia no es visible a simple vista, gracias a las distintas wikis, se puede ver como es el [**crecimiento de estadísticas de los personajes**](https://www.suikosource.com/games/gs2/guides/statgrowth.php)**[]**.

* 1. Conclusión del género

Como se ha podido observar, el RPG es un género a la orden del día que desde el momento de su nacimiento no ha parado de evolucionar y de generar cada vez más publico.

Primeramente fueron juegos con gráficos 2D por todas las plataformas, o casi todas, ya que desde tiempos de la Super Nintendo hay juegos de esta índole.

Seguidamente, pese que al principio de las consolas como PlayStation hubieron mchos juegos en 2D, se dio el salto al 3D con juegos icónicos como Final Fantasy VII o Chrono Cross, entre muchísimos otros títulos. Y desde entonces ya se entró plenamente en la época del 3D y casi todos los RPG llegaron al mercado en este formato.

A nivel mecánico ha habido bastante evolución desde los primeros turnos básicos (ej Pokémon), pasando por turnos con barra de tiempo individual (Final Fantasy VI, Final Fantasy VII), turnos con barra de tiempo colectiva (Grandia), combate en tiempo real (Final Fantasy XV),…

Y la evolución no se limita únicamente a nivel mecanico, puesto que la narrativa de los juegos también ha avanzado considerablemete. Pues con la aparición de los RPG occidentales y su sistema de elecciones y diálogos se abrió un nuevo horizonte capaz de hacer mas interesante la forma de ver el juego como tal, al punto de verlo como un libro interactivo (la saga The Witcher, Fallout o Mass Effect son ejemplos de narrativa moderna).

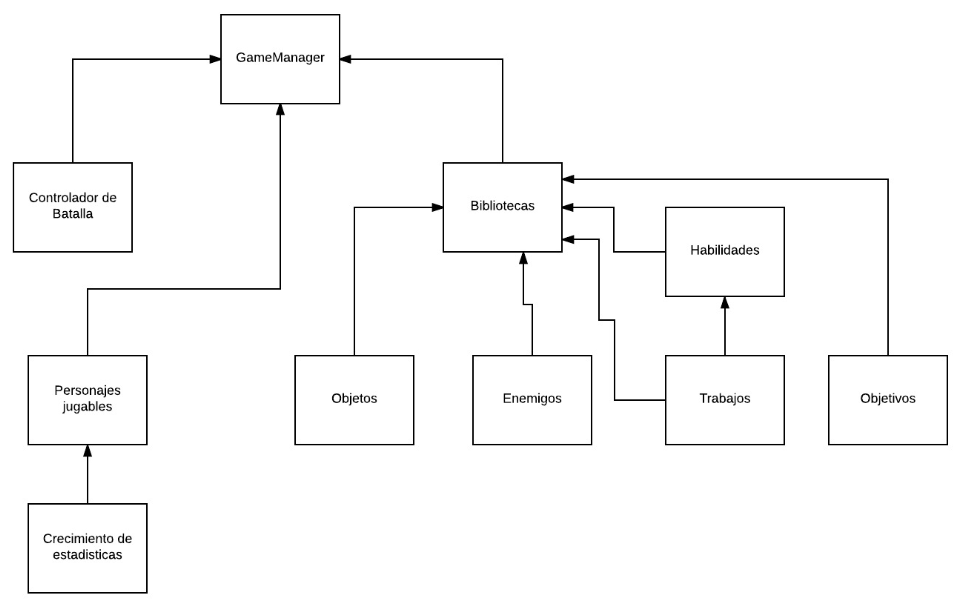
Videojuegos como la saga Persona, a partir de su tercera entrega supusieron una evolución dentro del género pues a parte de ser un videojuego RPG, introdujeron un componente social, es decir, un simulador de vida donde el jugador debe hacer la vida de un adolescente (estudiar, ejercitarse, relacionarse…).

En definitiva, el RPG es un género muy grande y expandido a lo largo del mundo a través de todas sus variantes. Y a día de hoy generan una gran cantidad de expectación entre el publico lo cual puede verse en los éxitos recientes de juegos como Persona 5, The Witcher 3, Zelda Breath of the Wild y Nier:Automata.

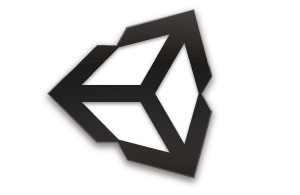
1. Objetivos y justificación

El objetivo de este proyecto es la creación de un videojuego en 2D incluido dentro del género RPG, más concretamente en su variante japonesa, es decir, JRPG.

El videojuego se construirá con el motor comercial Unity 5. Para ello se implementará todo un sistema de clases genérico para hacer este tipo de videojuegos. Este sistema tendrá soporte para varias funcionalidades para videojuegos jrpg entre las que se destacan: controlador del juego y batalla, sistema de trabajos, sistema de (crecimiento de) estadísticas, bibliotecas de objetos, enemigos, objetivos(misiones) y habilidades.



* 1. ¿Por qué Unity 5?



Se utilizará el motor Unity 5 para la construcción del videojuego porque es una herramienta potente, comercial y tiene versión gratuita. Pero el motivo más importante del porque usar Unity 5 es para aprender a utilizarlo junto con todas sus funcionalidades ya que es una herramienta de desarrollo muy asentada en el mercado.

Unity ofrece una gran cantidad de ventajas para el desarrollador que van desde la fácil implementación de prototipos hasta la exportación a 21 plataformas diferentes con un solo código, lo cual hace que tenga gran fuerza en el mercado. Por otro lado, la curva de aprendizaje es baja lo cual hace que sea rápido en comprender como funciona tanto el motor como la estructura basada en componentes. Además, para complementar dicha curva de aprendizaje, la comunidad de Unity es enorme y muy activa, al igual que la documentación, que se extiende más allá de los foros oficiales hasta llegar a Youtube donde hay una gran cantidad de tutoriales para llevar a cabo cualquier idea. Además de tener una tienda de assets que no para de actualizarse día a día con las aportaciones de la comunidad.

A nivel de scripting, el motor permite hacer scripts en dos lenguajes de programación diferentes: C# y Javascript, donde además son combinables dentro del mismo código.

A pesar de que no puede competir a nivel gráfico con motores como Unreal o Cry Engine, tras la introducción de la versión 5, Unity dio un salto de calidad en cuanto a potencia y se abrieron todo un mundo de posibilidades.

Dicho esto, lo que se pretende es crear una base para hacer videojuegos RPG en Unity 5.

* 1. ¿Por qué un JRPG?

Actualmente el JRPG es un género que está presente independientemente del paso de los años. Es un género bastante aclamado y que tiene un público bastante fiel tanto en su variante más clásica (juegos más estratégicos con turnos) como en su variable más moderna (juegos con más acción, rapidez y escenas cinematográficas).

Prueba de ésto son los distintos lanzamientos que tenemos hoy en día, como pueden ser:

* Bravely Default (Nintendo 3DS, año 2012)
* Bravely Second (Nintendo 3DS, año 2015)
* Persona 5 (PS3, PS4, año 2016)
* Tales of Berseria (PS3, PS4, PC, año 2017)
* Tales of Zestiria (PS3,PC, año 2016)
* Final Fantasy 15 (PS4, Xbox One, año 2016)
* Pokémon Sol y Luna (Nintendo 3DS, año 2016)
* Pokémon UltraSol y UltraLuna (Nintendo 3DS, año 2017)
* Dragon Quest XI (Nintendo 3DS, año 2017)
* Dragon Quest VII (Nintendo 3DS, año 2016)
* Dragon Quest VIII (Nintendo 3DS, año 2017)
* Nier: Automata (PS4, PC, año 2017)
* Xenoblade Chronicles (Wii, año 2012, New Nintendo 3DS, año 2015)
* Xenoblade Chronicles X (WiiU)
* Fire Emblem Fates (Nintendo 3DS, año 2015)
* Fire Emblem Echoes: Shadows of Velentia (Nintendo 3DS, año 2017)

1. Metodología
   1. Metodología de desarrollo iterativo/evolutivo

La metodología utilizada para este proyecto ha sido una metodología basada en iteraciones, donde en cada iteración se ha presentado un prototipo que ha permitido que el proyecto haya ido evolucionando con cada iteración al añadir cada vez más funcionalidades.

Las iteraciones se han producido en intervalos de tiempo de 2 semanas cada una.

* 1. Repositorio

Para el desarrollo del proyecto se ha escogido un repositorio en Github para tener un control de versiones.

De ésta forma se puede elegir que salvar con el fichero gitignore y tener una copia de seguridad en la red de todos los avances realizados. Cada copia de seguridad se ha hecho cada 2 semanas coincidiendo con la iteración de la metodología.

1. Cuerpo del trabajo
   1. Documento del juego (GDD)
   2. Estructura del juego

Como bien se ha podido ver en la figura X, el juego hace soporte a distintas funcionalidades, y en este apartado se va a explicar en que consiste cada apartado.

* + 1. GameManager

El GameManager es una clase siempre presente a lo largo de la ejecución del juego, y es la encargada de controlar el ciclo de ejecución del juego, así como el estado de éste (pausa, batalla, mapa…) ya que puede cambiarlo cuando es necesario. Además esta asociado con el resto de clases que almacenan datos como las bibliotecas y el grupo de personajes. También tiene control sobre parte de la interfaz del juego (menús) ya que es capaz de activarlos o desactivarlos.

En resumen es la clase encargada de controlar todo lo que pasa en el juego, así como de inicializar todos los componentes que permiten que funcione el juego.

* + 1. Controlador de batalla

El controlador de batalla es la clase que se encarga de inicializar todos los elementos que intervienen en la batalla (unidades, interfaz…) y de controlar el flujo de éste con una secuencia de estados (espera, decisiones de jugador/enemigo, selección de objetivo, calculo de daño, victoria, derrota y escapar).

El flujo de la batalla es el siguiente: primeramente la batalla esta en modo espera, donde se llenan las barras de tiempo y en el momento se llena una de las barras, pasa al siguiente estado (elección de jugador o elección de enemigo) donde se elige la acción a realizar. Seguidamente, se selecciona el objetivo y una vez hecho esto, se pasa al calculo de daño para poder aplicarlo. Una vez aplicado el daño al/los objetivos pertinentes, se comprueba la condición de victoria/derrota y en caso de cumplirse se pasa a ese estado que da fin a la batalla.

* + 1. Personajes jugables y crecimiento de estadísticas

Los personajes jugables son los personajes que el jugador puede controlar, es decir, pertenecer al grupo. Estos se caracterizan por poder tener, además de las estadísticas base, un trabajo asociado que potencia sus estadísticas. Así pues, también pueden subir de nivel, lo cual hace que sus estadísticas crezcan. El crecimiento de estadísticas se aplica mediante probabilidades, es decir, dependiendo de la estadística y su grado de crecimiento, tendrá una probabilidad de sumar más o menos puntos.

* + 1. Jugador y Cámara

El jugador como tal es únicamente el sprite que se mueve por pantalla, pero es importante puesto que siempre está presente a lo largo de la ejecución.

La cámara esta relacionada con el jugador, puesto que le sigue en todo momento.

* + 1. Bibliotecas

Las bibliotecas almacenan los datos de diferentes colecciones como los enemigos, los trabajos, las habilidades, los objetivos y los objetos.

Se cargan una vez al principio del juego y son siempre accesibles.

Dentro de las bibliotecas se pueden diferenciar dos tipos: las que se cargan como desde fichero y las que no.

Las que se cargan desde fichero son las bibliotecas de objetos y la de objetivos. En este caso se trata de un fichero json, que contiene los datos que identifican a cada objeto y a cada objetivo.

El resto son las que no se cargan desde fichero, se cargan como una instancia de prefab (ver siguiente apartado).

Pasando ya al funcionamiento de las bibliotecas, todos los datos se guardan en una lista contenida en el script que actúa como biblioteca en cada caso. Funcionan como bases de datos, es decir, en base a una consulta realizada mediante un método en concreto (buscar por ID, por nombre, por tipo…) devolverá el o los resultados que se correspondan con el criterio de búsqueda utilizado.

* + - 1. Construcción de los objetivos

Los objetivos marcan la línea de acción del juego, es decir, a donde debe ir el jugador, si el jugador debe de obtener algún objeto…

Así pues, los objetivos están pensados principalmente para funcionar en una estructura similar a la de un árbol, es decir, un objetivo puede desbloquear uno o más objetivos y a su vez un objetivo puede necesitar de uno o más objetivos completados para ser desbloqueado.

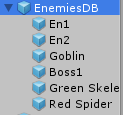
* 1. Prefabs

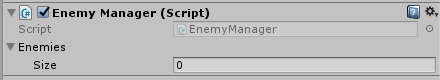
En este apartado se habla de los prefabs que han sido necesarios para la elaboración del videojuego

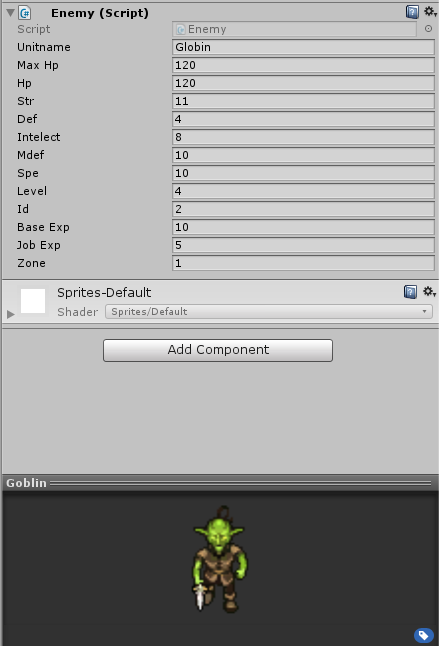
* + 1. Prefabs que hacen la labor de bases de datos

Estos prefabs como se ha mencionado anteriormente, forman parte de las bibliotecas del juego. Estan compuestos por un gameObject raíz que contiene el script (manager) que hace la función de biblioteca y diversos gameObject por debajo del raiz, es decir, los hijos que contienen a su vez el script del tipo de dato pertinente. El funcionamiento es el mismo en los 3 casos, primeramente, al instanciarse el prefab, el script (manager) de cada uno obtiene los datos de los scripts de los hijos y los guarda en una lista.

* + - 1. Enemigos

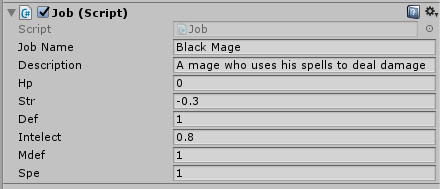






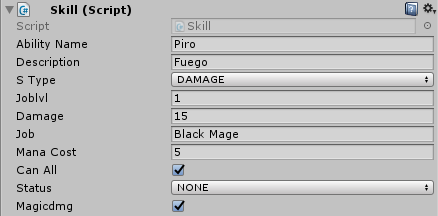
* + - 1. Trabajos





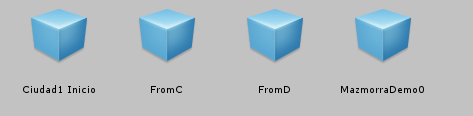
* + - 1. Habilidades





* + 1. Trasladores

A lo largo del juego, el jugador cambia de localización (de mapamundi a ciudad o a mazmorra por ejemplo), así pues los puntos en los que se cambia de localización (y de escena) están guardados como prefab. La necesidad de estos prefab estaba en que no era posible poner las coordenadas (x,y) del destino a mano, ya que debido a la organización de los gameObject y su posición de referencia respecto a la escena podía no ser la misma.



* + 1. Prefabs de interfaz

Los prefabs de interfaz son muy simples pues suelen contener pocos elementos de tipo UI (textos, paneles…). Su función es principalmente ayudar a mostrar la informacion por pantalla. Esto se consigue instanciando estos prefabs mediante código rellenando asi su panel contenedor.



* 1. Serialización de datos

La serialización[] de datos se utiliza para poder guardar datos del juego dentro de un fichero (en nuestro caso, un fichero json).

El guardado de datos mediante serializacion es muy fácil de hacer puesto que solo debemos indicar que la clase que queremos guardar en fichero sea serializable.

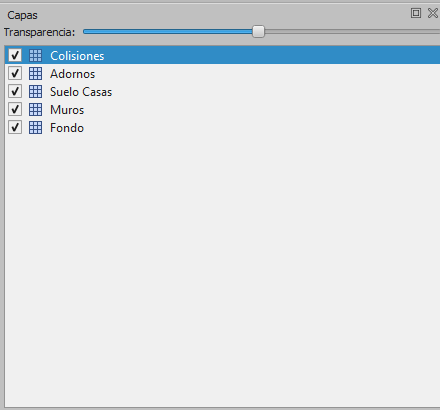
Aunque no permite serializar tipos de datos complejos como pueden ser los diccionarios, permite guardar todo tipo de datos comunes como int, float,string, listas… y en caso de necesitar que se guarde información de datos que no admiten serializacion se puede hacer un script para descomponer esa estructura mas compleja, extraer sus datos simples y poder guardarlos. Y a la hora de carga en este caso específico es hacer el proceso inverso cargar los datos del fichero en la estructura.

El proceso de serialización de datos se compone de una serie de llamadas a las funciones de la api de json que vienen por defecto en unity[] y de forma atuomática, indicándole el destino (fichero en caso de hacer guardado y variable en caso de hacer cargado de datos), se hacen todos los procesos para ambas operaciones.

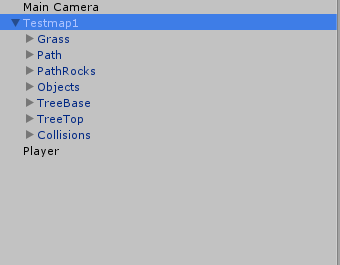
* 1. Creación de los niveles

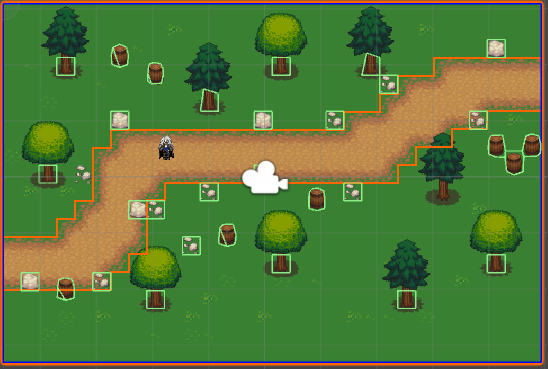
Al ser un videojuego en 2D, la herramienta utilizada para elaborar los distintos niveles/mapas que componen el juego ha sido Tiled.

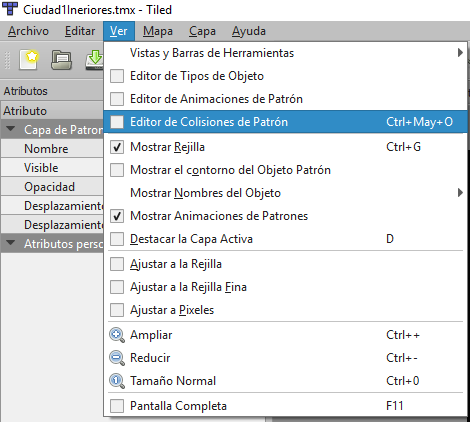
Esta herramienta permite dividir el escenario en capas pudiendo así ofrecer efectos de profundidad además de tener así organizados los distintos elementos del mapa.



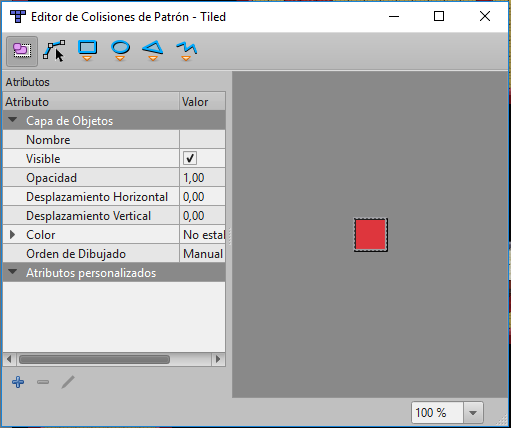
A su vez, también podemos establecer las colisiones existentes con Tiled, donde posteriormente al exportar se transformarán en *colliders* de Unity.

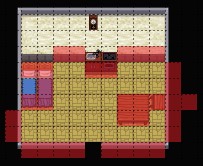
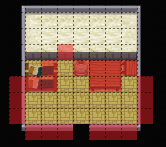




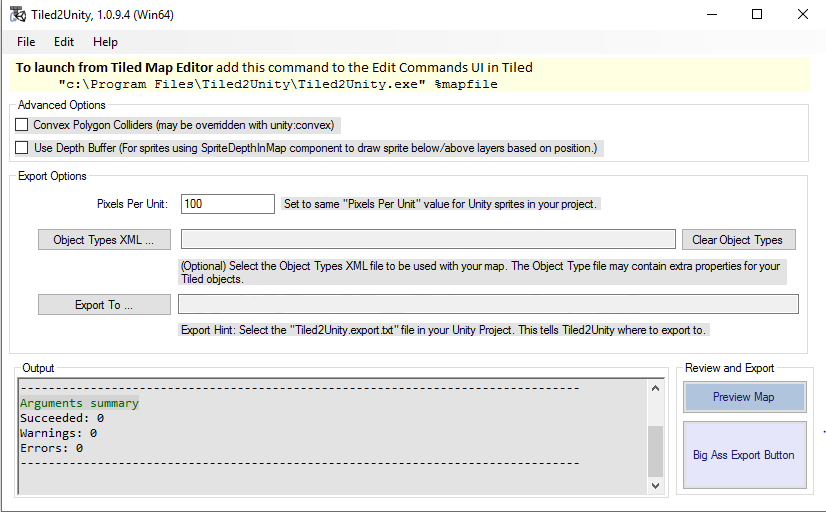


Para crear la colisión en Tiled, entramos en el Editor de Colisiones de Patrón y seguidamente se selecciona uno de los tiles que tengamos cargados, en este caso se ha utilizado un sprite de 32x32 de color rojo en su totalidad. Para terminar se selecciona la forma que queremos que tenga la colisión (personalizada, cuadrado, círculo) y seleccionamos todo el sprite con la forma seleccionada.





Para exportar los mapas creados en Tiled, se ha utilizado una herramienta gratuita llamada Tiled2Unity, que permite en un solo “click” convertir el archivo *.tmx* en un *prefab*de Unity.



El detalle más importante a tener en cuenta es el parámetro *Pixels Per Unit*, el cual hay que poner a 100 para que tenga la escala correcta y pueda visualizarse el mapa como realmente es.

* 1. Asset de diálogos RPGTalk

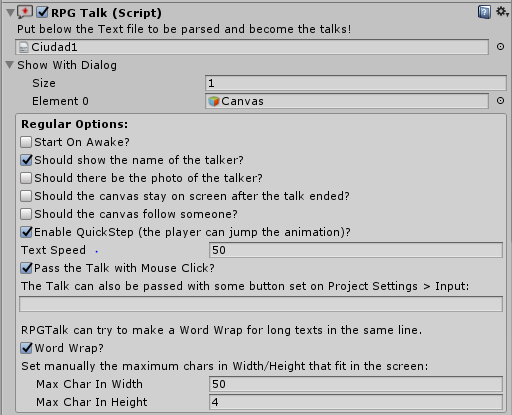
RPGTalk es un asset de la tienda de Unity desarrollado por seizestudios[referencia a la web], completamente gratuito, que ofrece al desarrollador una forma muy fácil de crear diálogos con varios efectos como la animación de texto, la animacion de los retratos al producirse el retrato,...

Para utilizar este asset lo único que hay que hacer es arrastrar el prefab que nos trae el asset y preparar la escena con un canvas y los diferentes paneles que componen la UI del diálogo.



A continuación se pueden ver los distintos parámetros/variables del asset junto con la multitud de opciones disponibles.

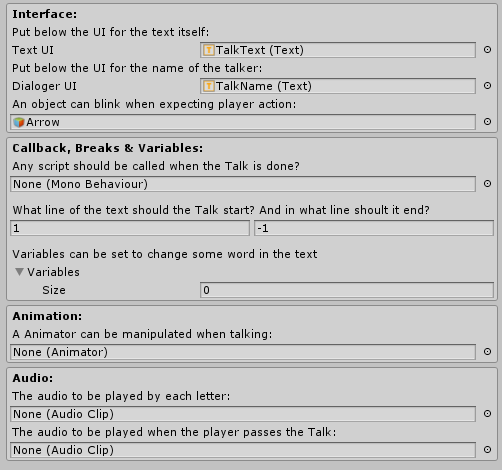
El primer parámetro es el más importante puesto que sin él, el diálogo no podría tener lugar. Estamos hablando del fichero .txt que contiene el texto del diálogo.



Seguido del fichero, se pueden ver distintas opciones y parámetros como el/los elemento/s UI al que debe estar enlazado el texto.

Por lo que respecta a las opciones, permiten bastante personalización como: empezar el diálogo al empezar la escena, mostrar o no el nombre del personaje que está hablando en ese momento, mostrar o no el retrato del personaje mientras habla,mantener el canvas una vez ha terminado la conversación, hacer que el canvas siga o no a un personaje para que el texto esté junto al personaje en todo momento, la velocidad del texto, el input con el que el jugador puede continuar con el diálogo una vez se ha llenado el cuadro de texto y hacer que el texto se ajuste al tamaño del cuadro texto.

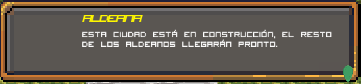
Siguen los bloques de interfaz,animación al hablar, audio y el bloque de callbacks, scripts y variables, que automatiza la llamada a otro método en caso de necesitarlo y con las variables se logra más personalización. Por ejemplo: si el jugador puede elegir su nombre, mediante una variable nombrada por ejemplo “nombreJugador” se puede mostrar el nombre deseado en el cuadro de diálogo. Además se pueden utilizar estas variables dentro del fichero de texto para de esa forma hacer los cambios en nombres o en cualquier parte del texto.

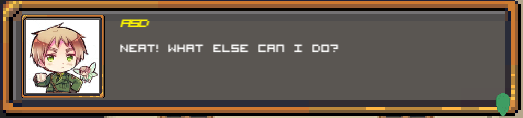


Para la interfaz se deben de enlazar los elementos UI en los cuales va a estar contenido el nombre del personaje,el texto del diálogo, los retratos de los personajes (en caso de estar activada la opción) y un objeto que indique la continuación del diálogo como por ejemplo una flecha.

En el bloque de callbacks, además de lo mencionado anteriormente, es donde se indica el principio y fin del diálogo, es decir, se indican las líneas del fichero que se deben leer.

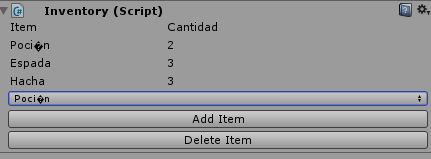
Finalmente se pueden añadir efectos de audio al diálogo tanto al pasar cada letra como al avanzar en el diálogo cuando el jugador pulsa el input correspondiente.





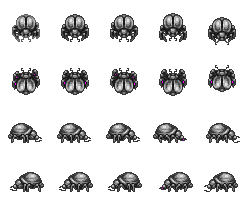
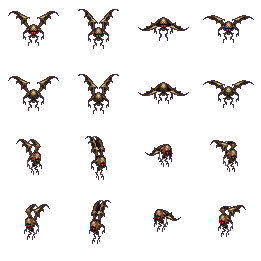
* 1. Pequeño editor de inventario para inspector de Unity

Para realizar pruebas en el inventario de una forma más rápida, se ha desarrollado una pequeña modificación del inspector de Unity para de esta forma poder hacer pruebas más visuales y además en tiempo de ejecución.

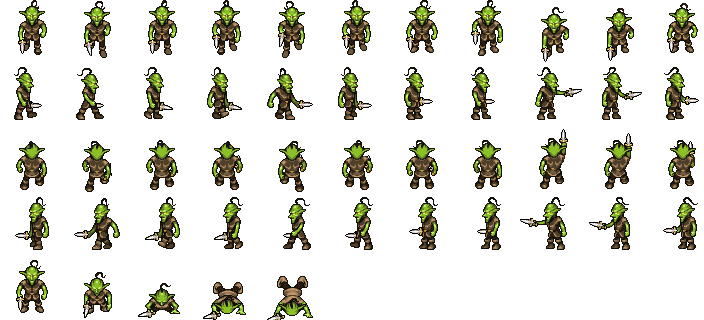


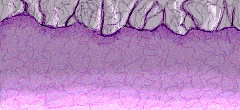
Se puede ver como se listan los elementos del inventario y además un desplegable en el que se muestran las posibles opciones de los objetos que pueden estar en el inventario. Dichos objetos que están en el desplegable son recogidos de la base de datos creada a partir de un fichero json. Y por ultimo los botones que permiten añadir o eliminar del inventario el objeto seleccionado mediante la lista desplegable.

* 1. Arte del juego





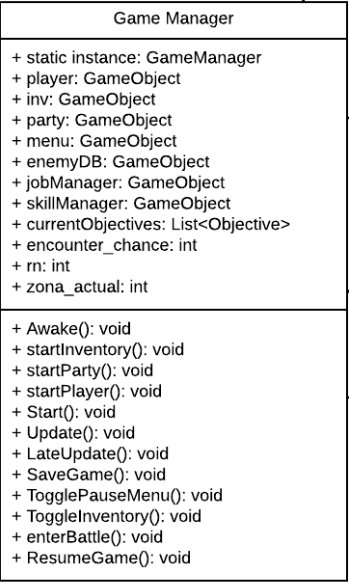








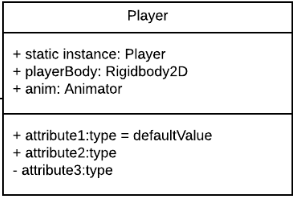
1. Conclusiones
2. Anexo
   1. Diagrama de clases
      1. Clase GameManager



Esta es la clase que controla el flujo del juego, es decir es la que se encarga de almacenar y mantener la información general del juego (% de encuentros, zona actual, referencias a las demás entidades independientes del juego…) además de cambiar el estado del juego (mapa, pausa, batalla…).

Entrando más en detalle, los métodos pertenecientes a la clase son:

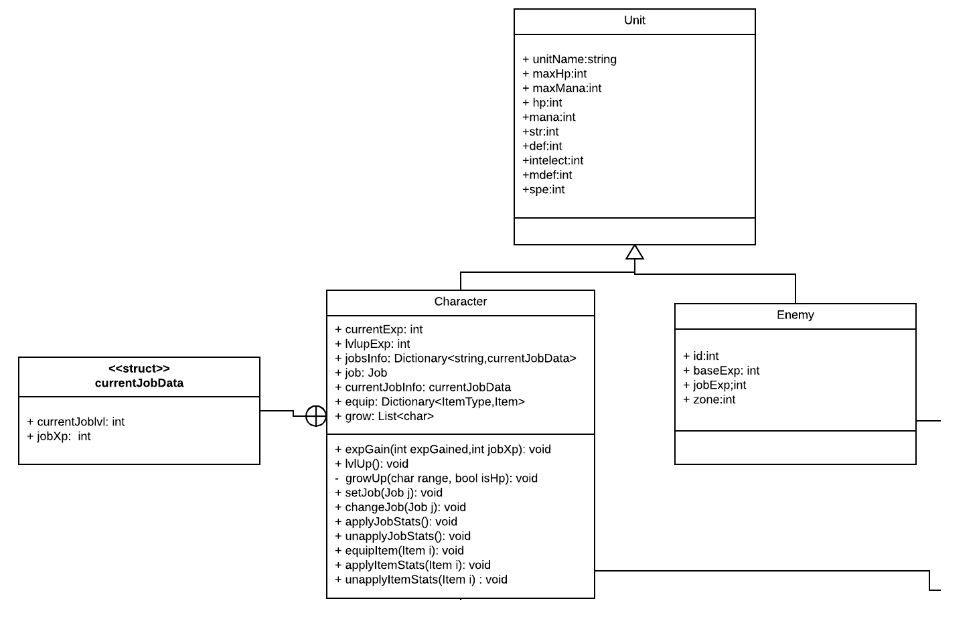
* Awake(): método que se llama al principio de la ejecución, donde se crea la instancia del propio GameManager como de otros singleton mediante la llamadas a los métodos pertinents, así como crear una instancia de las variables GameObject enemyDB y skillManager.
* StartInventory(): método que instancia el GameObject inv creando así una instancia para la clase Inventory.
* StartParty():método que instancia el GameObject party creando así una instancia para la clase Party.
* StartPlayer():método que instancia el GameObject player creando así una instancia para la clase Player.
* Start(): método que inicializa algunas de las variables de la clase como encounter\_chance o zona\_actual.
* Update(): método que se encarga de controlar los cambios de estado, principalmente se encarga de comprobar si se entra o no en batalla.
* LateUpdate(): método que se encarga principalmente de recoger entradas de teclado.
* SaveGame(): método que llama a las funciones de salvado de algunas clases como Inventory.
* TogglePauseMenu(): método que pone el juego en pausa y muestra el menú de pausa.
* ToggleInventory()
* EnterBattle(): método que se encarga de cargar la escena perteneciente a la batalla.
* ResumeGame(): método que en caso de haber pausado el juego devuelve el juego al estado normal.
  + 1. Clase Player



Esta clase, asociada a un sprite, se encarga de controlar el movimiento del jugador con las animaciones pertinentes.

Los métodos de esta clase son:

* Awake(): método donde se crea la instancia de la clase, ya que se un singleton.
* Start(): método en el que se realiza la inicialización de las variables playerBody y anim.
* Update(): método en el que se establecen los cambios de sprite del animator y además se modifica la posición del sprite al cual está asociada la clase.
  + 1. Clases Unit,Character y Enemy.



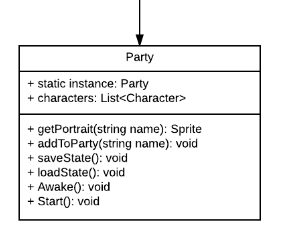
Estas 3 clases van prácticamente de la mano ya que la clase Unit es la clase base para las otras dos y se encarga de definir las estadísticas comunes tanto en los character como en los enemy, es decir, las estadísticas de vida, maná, fuerza, defensa, intelecto, defensa mágica y velocidad.

Por otro lado, tenemos la clase Enemy que hereda de la clase Unit, y tiene como atributos propios: id (para la base de datos de enemigos), la experiencia (para el personaje y para el trabajo) que da al derrotarlo y la zona a la que pertence.

Finalmente, tenemos la clase Character, que define a los personajes jugables del juego, y además de la información de la clase Unit, almacena información propia como un registro de los trabajos que ha tenido el personaje (con nivel y experiencia en el momento), el trabajo actual equipado con su información de experiencia correspondiente, la lista de Items que tiene equipados y una lista que define el crecimiento de estadísticas que va a tener el personaje.

Respecto a los métodos de la clase Character:

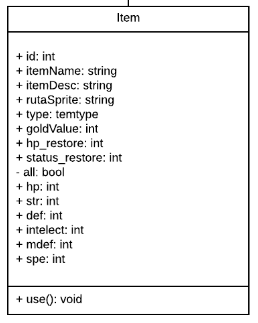
* ExpGain(int expGained, int jobXp): método que se encarga de sumar la experiencia obtenida a la experiencia actual y comprobar cuando se sube de nivel. Se aplica tanto a la experiencia propia del personaje como a la experiencia del trabajo que tiene asociado.
* lvlUp(): método que se encarga de la subida de nivel del pesonaje, actualizando así la experiencia necesaria para subir al siguiente nivel además de actualizar las estadísticas del personaje mediante la llamada el método growUP.
* growUp(char range, bool isHP): método que se encarga de aplicar cuanto crecen las estadísticas en función del rango especificado por parámetro.
* setJob(Job j): método que se encarga de asignar un trabajo al personaje.
* changeJob(Job j): método que se encarga de realizar el cambio de trabajo. Además guarda la información del trabajo anterior ,recupera, en caso de existir, la información del nuevo trabajo equipado y actualiza las estadísticas del personaje acorde al nuevo trabajo.
* applyJobStats(): método que se encarga de aplicar los multiplicadores del trabajo a las estadísticas del personajes.
* unapplyJobStats(): método que se encarga de volver las estadísticas del personaje a su valor base.
* equipItem(Item i): método que se encarga de añadir a la lista de equipo el ítem pasado por parámetro.
* applyItemStats(Item i): método que se encarga de aplicar los aumentos de estadísticas que proporciona el objeto pasado por parámetro.
* unapplyItemStats(Item i): método que se encarga de volver las estadísticas a su estado anterior quitándole las mejoras del objeto pasado por parámetro.
  + 1. Clase Party



La clase Party es otro de las clases singleton y se encarga de almacenar los personajes jugables.

Los métodos de esta clase son:

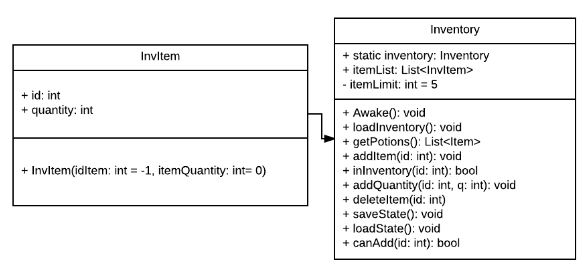
* Awake(): método que crea la instancia de la clase.
* Start(): método donde se almacenan los characters en la lista después de ser inicializada.
* GetPortrait(string name): método que devuelve el retrato del personaje que tenga por nombre el string pasado por parámetro.
* SaveState(): método que guarda en un fichero la información del grupo.
* LoadState(): método que recupera la información del grupo proveniente del fichero donde están guardados los datos.
  + 1. Clase Item



La clase Item define como va a ser un objeto en el juego, es decir, si va a ser una poción o un objeto equipable… Esta diferencia viene principalmente dada por la variable type.

El método use(), está pensado para las pociones y proporciona la información necesaria al BattleController para aplicar la curación pertinente.

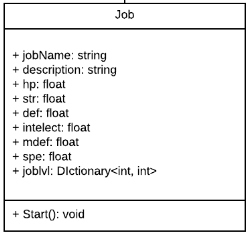
* + 1. Inventory



La clase inventario almacena los objetos que están en posesión del jugador, para ello se ha creado la clase InvItem que contiene un ID con el que se buscara el objeto en la base de datos y la cantidad que se tiene en posesión.

Los métodos que componen la clase son:

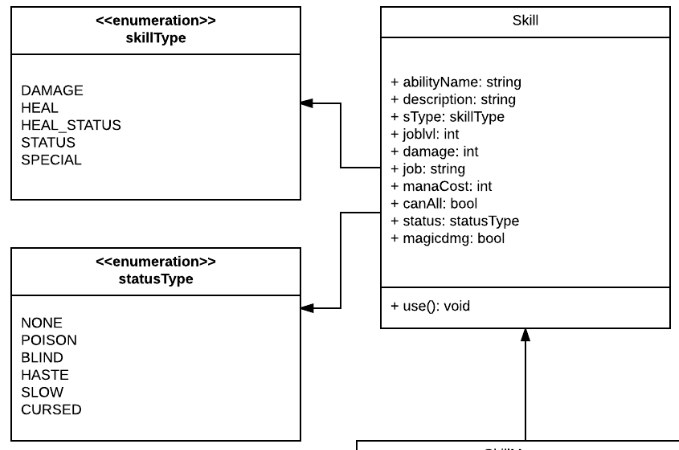
* Awake(): método que se encarga de crear la instancia de la clase.
* AddItem(int id): método que añade un objeto al inventario en función de su id (se comprueba que existe en la base de datos) en caso de que pueda añadirse (no se ha alcanzado el límite del inventario).
* AddQuantity(int id, int quantity): método que se encarga de buscar si el objeto cuyo ID es pasado por parámetro se encuentra dentro del inventario y en caso de estar dentro y no superar la cantidad de 99, se procede a aumentar la cantidad del objeto.
* InInventory(int id): método que busca el objeto en el inventario y devuelve verdadero o falso en función de si se encuentra o no.
* CanAdd(): método que devuelve verdadero o falso en caso de que puedan añadirse más objetos o no al inventario.
* GetPotions(): método que devuelve aquellos objetos en el inventario que son pociones.
* DeleteItem(int id): método que elimina del inventario el objeto especificado por parámetro en caso de estar contenido en el.
* SaveState(): método que guarda la información del inventario en un fichero json.
* LoadState(): método que recupera la información del fichero json para guardarla en la lista de objetos del inventario.
  + 1. Clase Job



La clase Job es la que define las estadísticas de cada trabajo, es decir, los multiplicadores que afectan a las estadísticas de los personajes.

En su único método, Start(), se almacena la información en el diccionario joblvl, donde se establece la relación entre el nivel de trabajo y la experiencia necesaria para llegar al siguiente nivel de trabajo.

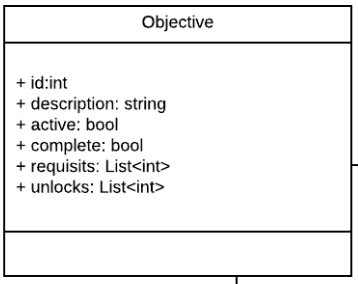
* + 1. Clase Skill



La clase Skill define las habilidades que podrán ser utilizadas por los personajes a través de los trabajos, ya que cada habilidad está directamente relacionada con un trabajo concreto.

El método use() proporciona información al BattleController para aplicar el daño o curación pertinente.

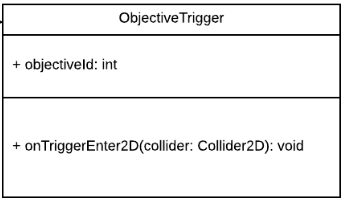
* + 1. Clase Objective



La clase Objective define los objetivos que tiene el juego. Está pensado para que sea una cadena de objetivos/eventos donde un objetivo puede depender o no de un objetivo anterior y que un objetivo pueda desbloquear o no varios objetivos. Esto incluye también el que un objetivo puede depender de 1 o más objetivos para ser debloqueado.

Un objetivo se desbloquea, en caso de tener un predecesor, una vez se han completado todos aquellos objetivos que están presentes en la lista de requisitos.

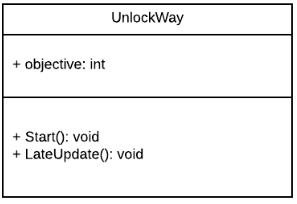
* + 1. Clase ObjectiveTrigger



La clase ObjectiveTrigger está pensada para completar objetivos que requieran pasar por cierto lugar o llegar a cierto sitio.

El método onTriggerEnter2D se encarga de llamar intermanente el controlador de objetivos y completar el objetivo en el caso de haber pasado por el lugar correspondiente.

* + 1. Clase UnlockWay



La clase UnlockWay está pensada para desbloquear caminos en función de si un objetivo determinado se ha completado o no.

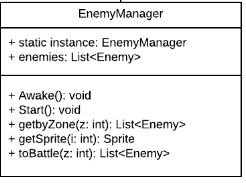
Los métodos de esta clase son:

* Start(): método que comprueba durante la inicialización de la clase si el objetivo correspondiente ha sido completado o no, y en consecuencia eliminar el GameObject asociado o no hacer nada. Esto es debido a que hay objetivos que se completan en una escena diferente a donde está el camino bloqueado.
* LateUpdate(): método que al final de cada frame comprueba si el objetivo ha sido completado o no, y en caso de estar completado, elimina el GameObject al cual está asociado la clase.
  + 1. Clases que actúan como bases de datos

Las siguientes clases actúan como bases de datos, es decir, como biblioteca de datos que está siempre disponible para extraerlos cuando es necesario.

Con la excepción de ItemDBController y ObjectiveController, el resto de bibliotecas no están hechas mediante informacion de fichero, sino que está realizada a mano en Unity mediante prefabs.

* + - 1. EnemyManager



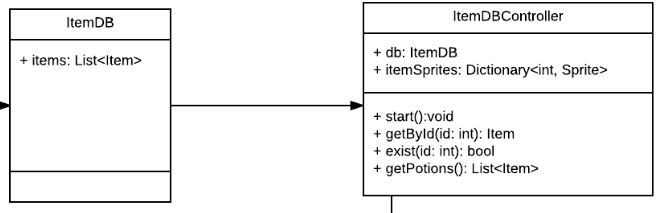
EnemyManager representa la biblioteca de enemigos del juego donde se almacenen en una lista todos los enemigos.

Respecto a los métodos Awake() y Start():

* Awake(): método que genera la instancia de la biblioteca.
* Start(): método que, al ser una instancia de un prefab de prefabs, recorre todos los GameObject que están asociados a la instancia y a partir de ahí recoje el script Enemy en cada uno de ellos y guardar así los enemigos en la lista.

Y como toda base de datos, la principal función de esta clase es devolver información, en este caso en forma de búsquedas o filtros:

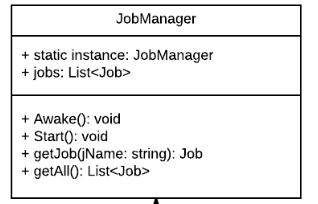
* getbyZone(int z): Devuelve todos los enemigos que se encuentran en la zona especificada con el int pasado por parámetro.
* getSprite(int i): Devuelve el sprite del enemigo cuyo ID coincida con el int pasado por parámetro.
* toBattle(int z): Devuelve una lista de 3 enemigos escogidos al azar de entre todos los que están en la zona especificada por parámetro.
  + - 1. ItemDBController



Esta biblioteca se genera mediante la lectura de un fichero json que contiene la información de todos los objetos del juego.

Su método Start() es el encargado de leer el fichero y almacenar toda la información, incluidos los sprites que se cargan en base a la ruta que hay en el fichero.

* + - 1. JobManager

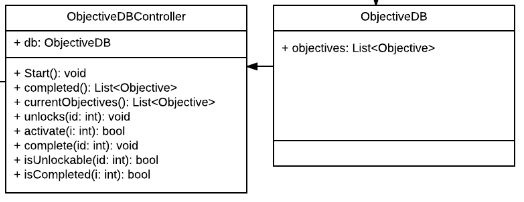


La clase JobManager representa la biblioteca de trabajos disponibles en el juego. La biblioteca es un prefab donde en el primer nivel esta el script de la clase y en los niveles inferiores están los GameObject que representan los trabajos.

Mediante los métodos Awake() y Start() se crea la estancia de la clase y se recorren los niveles inferiores o hijos del prefab para recoger el script Job asociado a cada uno de ellos.

Respecto al resto de métodos:

* getJob(string jName): método que devuelve el trabajo pertinente en función del nombre pasado por parámetro.
* getAll(): método que devuelve una lista con todos los trabajos disponibles.
  + - 1. ObjectiveDBController

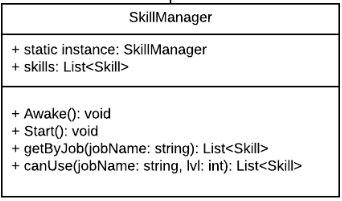


La clase ObjectiveController contiene todos los objetivos que estarán presentes durante el juego. La información de los objetivos se obtiene mediante la lectura de un fichero json.

Asi pues la clase se encarga de llevar control sobre los objetivos que hay en curso y se encarga de completar y activar los objetivos pertinentes.

Los métodos de la clase ObjectiveController son:

* Start(): método en el que se inicializa la lista de objetivos mediante la lectura de datos del fichero json.
* Completed(): método que devuelve una lista de todos los objetivos que ya han sido completados.
* CurrentObjectives(): método que devuelve una lista con los objetivos que están activos.
* Unlocks(int id): método que en caso de ser posible, desbloquea todos los objetivos que dependen del objetivo pasado por parámetro.
* Activate(int id): método que activa el objetivo pasado por parámetro.
* Complete(int id): método que marca como completado el objetivo que se le pasa por parámetro.
* IsUnlockable(int id): método que comprueba si el objetivo pasado por parámetro es desbloqueable o no. Para ello comprueba si sus requisitos han sido completados.
* IsCompleted(int id): método que comprueba si el objetivo pasado por parámetro esta completado o no.
  + - 1. SkillManager

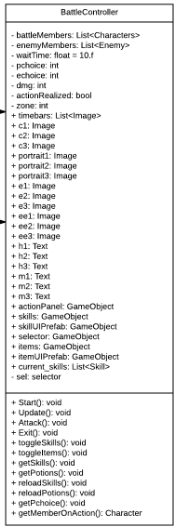


La clase SkillManager tiene el mismo funcionamiento que las clases JobManager y EnemyManager. Almacena en la lista de habilidades que tendrán los trabajos.

Con los métodos Awake() y Start() se crea la instancia de la clase y se recorren los hijos buscando los script de tipo Skill para guardarlos en la lista.

El resto de métodos son de tipo filtro:

* getByJob(string jobName): método que devuelve el trabajo que coincide con el nombre pasado por parámetro.
* canUse(string jobName, int lvl): método que devuelve todas las habilidades que se pueden utilizar con el nivel de trabajo especificados.
  + 1. Clase Battle Controller



La clase BattleController se encarga de controlar todos los aspectos de la batalla, desde la interfaz hasta todos los cálculos de daño.

Los método de esta clase son:

* Start(): método que inicializa todas las variables, entre las que se encuentran: las unidades que participan en la batalla (tanto aliadas como enemigas) y los componentes de interfaz (sprites de enemigos y portrait de los personajes se enlazan a su correspondiente componente de interfaz).
* Update(): Este método es el mas importante pues es el que se encarga de controlar todo el flujo de la batalla. Se divide en distintos estados: espera, elección del jugador, elección del enemigo, selección de objetivo, calculo de daño, victoria y derrota.

En el estado de espera se van llevando las barras de tiempo y en el momento se llena una, detecta cual es y pasa a elección de jugador o de enemigo en función de la barra llena.

En el estado de elección del enemigo, el enemigo elige al personaje con menos vida y le ataca.

En el estado de elección de personaje se abre el menú con las acciones que se pueden realizar (atacar, habilidades, objetos, escapar).

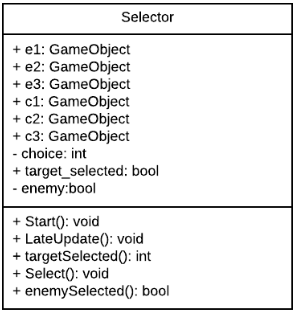
En el estado de selección de objetivo se muestra un marco selector que puede moverse a través de las unidades para seleccionar la unidad que va a ser objetivo del ataque, habilidad u objeto.

En el estado

En el estado victoria se suman todas las ganancias de experiencia y se aplican a los personajes, posteriormente el juego vuelve al mapa desde el cual se ha entrado en batalla.

En el estado derrota

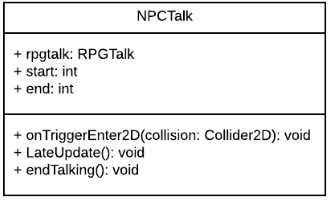
* Attack(): Este método se encarga de recoger que personaje hace la acción y posteriormente pasa al estado de selección de objetivo.
* Exit(): Este método se encarga de devolver el juego al mapa desde el cual se ha entrado en batalla.
* ToggleSkills(): Este método muestra las habilidades del personaje que tiene el turno.
* ToggleItems(): Este método muestra los objetos (pociones) del inventario.
* GetSkills(): Método que recoge las habilidades del personaje. Se utiliza para cargar las habilidades en la interfaz.
* GetPotions():Método que recoge las pociones del inventario. Se utiliza para cargar las pociones en la interfaz.
* ReloadSkills(): Este método se encarga de recargar el panel de habilidades, es decir, elimina el contenido del panel.
* ReloadPotions():Este método se encarga de recargar el panel de objetos, es decir, elimina el contenido del panel.
* GetPchoice(): Este método devuelve que personaje esta eligiendo.
* GetMemberOnAction():
  + 1. Clase Selector



La clase Selector se utiliza en la batalla, cuando el jugador debe decidir el objetivo de su movimiento. De esta forma se muestra un marco que se puede mover por las posiciones de cada unidad.

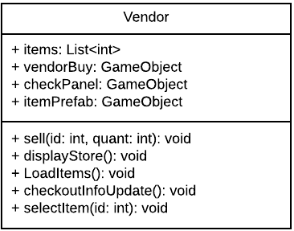
Los métodos que componen la clase Selector son:

* Start(): método que inicializa las variables choice, target\_selected y enemy.
* LateUpdate(): método que controla la entrada de teclado del selector, haciendo que pueda moverse arriba,abajo, derecha e izquierda.
* TargetSelected(): método que devuelve que objetivo se ha seleccionado en forma de índice.
* Select(): método que indica que se ha seleccionado un objetivo poniendo la variable target\_selected a true.
* EnemySelected(): método que indica si el selector esta encima de un enemigo o no.
  + 1. Clase NPCTalk



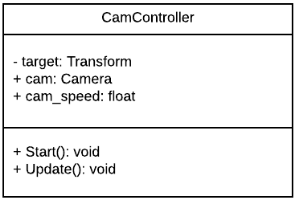
La clase NPCTalk esta enlazada a los NPC del juego y esta pensada para que se produzcan las conversaciones. Todo gracias al asset RPGTalk.

* onTriggerEnter2D(Collider2D collision): método que al activarse gracias a la colision del NPC con el jugador o viceversa, establece el principio y final de la conversación en la variable rpgtalk y además un cambio el estado del juego (pasa de estado MAP a estado TALKING).
* LateUpdate(): método que controla la entrada de teclado y es la encargado de, en caso de que se introduzca el input correcto, dar inicio a la conversación.
* EndTalking(): método que reestablece el estado del juego de vuelta a MAP al finalizar la conversación.
  + 1. Clase Vendor



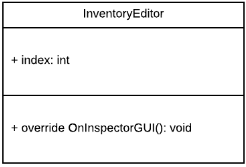
La clase Vendor, es la clase que almacenan y definen el comportamiento de los vendedores del juego.

* + 1. Clase CamController



La clase CamController se encarga del movimiento de la cámara, que estará siguiendo al personaje controlado por el jugador en todo momento.

* + 1. Clase InventoryEditor



1. Bibliografía