Universitat Oberta de Catalunya

uoc.edu

Noche de las Walpurgis

Luis Mario Cucurull Cárcel
Jordi López Belzunces
Rodrigo Puig Cabrera
Informática y telecomunicaciones
Máster en diseño y desarrollo de videojuegos

Helio Tejedor Navarro
Jordi Duch Gavaldà
Joan Arnedo Moreno

30 de junio de 2018

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Noche de las Walpurgis*
Nombre del autor:	*Luis Mario Cucurull Cárcel* *Jordi López Belzunces* *Rodrigo Puig Cabrera*
Nombre del consultor/a:	*Helio Tejedor Navarro*
Nombre del PRA:	*Jordi Duch Gavaldà*
Fecha de entrega (mm/aaaa):	06/2018
Titulación:	*Máster en diseño y desarrollo de videojuegos*
Área del Trabajo Final:	*Trabajo final de máster*
Idioma del trabajo:	*Castellano*
Palabras clave	*Aventura, retro, 2D*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

Noche de las Walpurgis es un videojuego de aventuras al estilo clásico, como por ejemplo The legend of Zelda a Link to the pass [1]. Encarnaremos a un héroe que necesitará determinados objetos para poder completar su misión, dichos objetos los conseguiremos realizando puzles en las diferentes pantallas.

Para poder realizarlo se han utilizado diferentes tecnologías, Unity 3D como motor, Photoshop para el tratamiento de imágenes y tiles entre otros. Al ser un trabajo en grupo obliga a coordinarse para poder llegar al objetivo, para ello se ha utilizado Trello como herramienta de gestión del proyecto y Bitbucket como sistema de control de versiones.

Cada uno de los miembros ha abarcado más de un área del desarrollo del juego, repartiéndonos el trabajo y haciendo lo que cada uno nos propusimos en un principio. También algunas partes que estaban proyectadas han sido abandonadas por no haber dado tiempo a todo.

Como conclusión decir que estamos orgullosos del trabajo realizado, aunque han quedado muchas cosas en el tintero. Haber trabajado en grupo ha sido sin duda una gran experiencia de cara a la gestión de tareas y tiempo, así que con lo aprendido aquí podremos evitar bastantes de los errores que hemos cometido durante el TFM.

Abstract (in English, 250 words or less):

Noche de las Walpurgis is a classic styled adventure videogame like The Legend of Zelda A Link to the Pass [1]. We'll embody a hero in need of specific objects to be able to resolve his misión. Said objects can be achieved solving puzles at different screens.

Different technologies have been used in the creation process, among others Unity 3D as the engine and Photoshop for image and tile processing. Being a team work forces coordination to achieve the objectives: Trello for project management and Bitbucket as a version control system are the tools chosen.

Every member has embraced more than one development áreas of the game, distributing tasks and doing all that each member proposed at the beginning. Also, some projected parts have been abandoned since there hasn't been enough time.

As a conclusión, we're proud of the work done, even though lots of things have been put aside. Without any doubt, team working has been a great experience facing task and time management so we will avoid the good few errors we've commited during the TFM.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del Trabajo

La idea comienza a rondarnos por la cabeza unos meses antes de comenzar el TFM, ¿Y si intentamos hacerlo en grupo para abarcar mucho más de lo que haríamos si estuviésemos solos?, así podríamos experimentar lo que realmente es la producción de un video juego, ya que nos repartiríamos el trabajo por áreas y cada uno tendría que adoptar uno o varios roles de dentro de la industria.

Ilustración 1 - Madness Factory logo

Empezamos a tener reuniones y hacer brainstorming en algunas de ellas, tuvimos varias ideas, pero la que acabó por ser la elegida fue la de hacer un juego de rol en 2D, no demasiado extenso, algo realista que pudiésemos llevar a cabo los 4 integrantes que éramos en un principio. Los juegos de rol tienen un gran número de seguidores y si encima son tipo pixel art además tienen una legión de fans de lo retro, así que decidimos de seguir por esta línea.

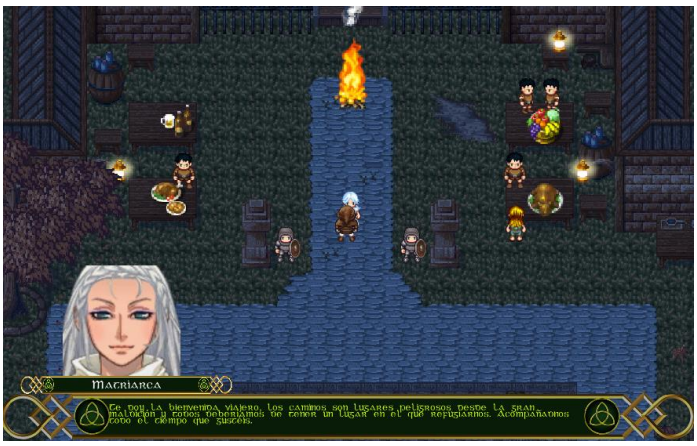


Ilustración 2 - Matriarca

Ilustración 3 - Aldea

Una vez que tuvimos claro el género empezamos a decidir cual sería la trama y donde estaría ambientado, se nos ocurrieron muchas historias, pero la que más cuajó fue la de hablar sobre la noche de las Walpurgis[2], una noche en la mitología celta en la cual las brujas y demonios campaban a sus anchas por las tierras de los hombres, y estos tenían que tener cuidado con no toparse con ellas hasta el amanecer, momento en el que todo volvía a la normalidad.
Decidimos que envergadura tendría que tener el juego para que fuera un trabajo de acuerdo a 4 personas y fue así como empezamos a escribir el GDD[3], tenemos que decir que para esa parte tuvimos muchísimo tiempo ya que empezamos un mes antes y pudimos pulirlo mucho, si ahora volviésemos atrás dedicaríamos parte de ese tiempo al desarrollo sin duda. Antes de terminar el GDD completamente perdimos a un integrante, pero decidimos seguir adelante con todo el contenido, un error sin duda, pero eso lo vimos después y tuvimos que acabar recortando igualmente, quizás no lo suficiente ya que el tiempo que hemos tenido ha sido muy justo para el trabajo que queríamos presentar y la motivación inicial no fue suficiente para poder abarcarlo todo.
Una vez acabado el GDD cada uno asumió su parte y nos pusimos manos a la obra.

Ilustración 4 - Trinqueta celta

1.2 Objetivos del Trabajo

Los objetivos iniciales del trabajo eran entregar un juego completo compuesto por:

6 escenas de juego completas.

- La aldea – Serviría como una escena de nexo que uniese las demás, es una zona segura desde donde lanzaremos las demás. Elegimos esta forma para simular que estamos asistiendo a la narración de unos hechos, en los cuales participaremos controlando al protagonista. Cada que vez que acabásemos una pantalla volveríamos aquí, para poder seguir escuchando la historia o descansar.

Ilustración 5 - Mapa de la aldea

- El bosque maldito – Primer nivel de nuestra historia, un oscuro bosque donde empezaríamos a introducir las mecánicas de nuestro juego poco a poco, sería casi como un nivel tutorial. En este nivel inicialmente no podríamos pelear, tendríamos que apañárnosla para evitar a los enemigos y trampas hasta encontrar una forma de defendernos, para ello pusimos el primer puzle del juego. Una vez superado tendríamos en nuestro poder una espada que ya nos permitiría pelear con los enemigos, a partir de este momento ya serian accesibles nuevas zonas y habría nuevos enemigos más agresivos con los que medirnos, así hasta llegar al jefe del nivel que nos pondría a prueba con una mecánica de vulnerabilidad/invulnerabilidad, mientras no lo pudiésemos dañar invocaría los enemigos vistos hasta ahora y al acabar con ellos pasaríamos a fase en la que podríamos vencer, repitiendo todas las veces que fuese necesario.
- El mundo de las hadas – Por una reacción mágica al acabar con el jefe anterior nos veríamos transportados a este mundo, donde como en cada escena habría un objeto poderoso para recoger. Añadimos nuevos enemigos y mecánicas, como la de dañar por la espalda a algunos enemigos para

poder eliminarlos y la del objeto del nivel, que nos permitiría lanzar un destello de luz que aturdiría a los enemigos a nuestro alrededor unos instantes y encender algunas antorchas mágicas, cosa indispensable para poder avanzar. Entonces ahora tendríamos las mecánicas de la espada del primer nivel y las del casco de este, que nos serían imprescindibles para poder acabar con el guardián de la zona.

- La isla de los piratas – Al escapar del mundo de las hadas llegaríamos a esta isla al azar, nuestra misión volvía a ser salir de allí. En esta pantalla volvemos a usar las mecánicas introducidas hasta ahora y añadimos otras nuevas, como por ejemplo un barril de ron que podemos envenenar para que todos los enemigos de alrededor mueran, eso sí, para que funcione deberemos de tener la capa de invisibilidad y hacerlo sin que nos vean.

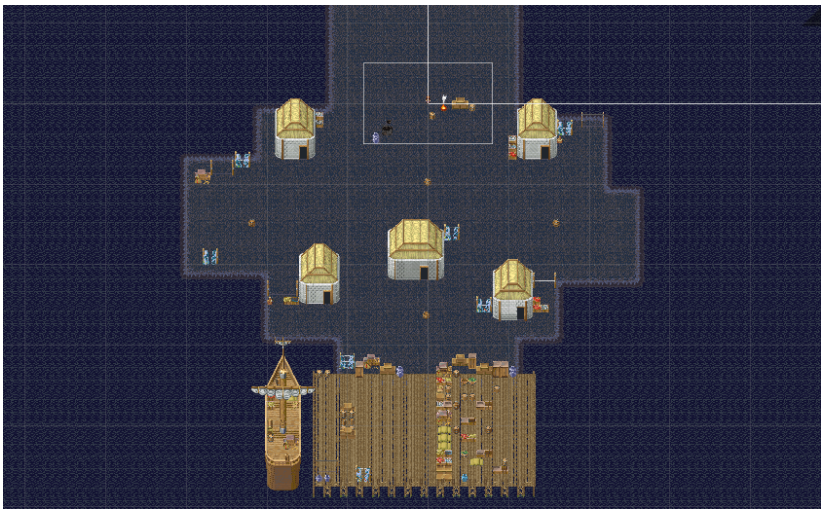


Ilustración 6 - La isla nivel 5

- La fortaleza – Este nivel representaba una fortaleza abandonada sonde debíamos de conseguir una armadura que nos hacía invulnerable durante un corto periodo de tiempo. Al final decidimos eliminarla y no estará en el trabajo final.

- La montaña maldita – Último nivel de nuestra aventura, se desarrolla en un volcán en erupción. En este nivel conseguiremos 2 objetos ya que la armadura de la fase anterior la decidimos de pasar a este para aprovechar sus mecánicas, el otro objeto será la capacidad de lanzar tornados con nuestra espada y poder afectar así blancos a distancia. Con este último poder también podremos activar objetos para continuar el viaje y será clave para eliminar al último enemigo ya que en un momento dado solo podremos alcanzarlo con el.

Enemigos para todas las pantallas.

- El juego originalmente se iba a entregar con 22 enemigos y trampas distintos, pero por la falta de tiempo hubo que eliminar 8 de ellos, con lo que

tuvimos que rellenar los niveles con enemigos ya usados en otros sitios. La idea era que cada lugar tuviese sus propios enemigos.

Ilustración 7 - Boceto araña

- 4 jefes de final de pantallas con sus comportamientos y mecánicas específicas, para añadir un reto al final de cada nivel.

Ilustración 8 - Baalberth

5 Objetos místicos que representarían nuestras mecánicas principales.

- Espada – La mecánica de combate más básica, con ella podríamos golpear a nuestros enemigos y despejar el camino de algunos obstáculos.

- Casco de la luz – Gracias a este objeto tendríamos la habilidad de cegar a nuestros enemigos, dejándolos inmovilizados unos instantes y además podríamos encender fuegos mágicos en algunos sitios.

- Capa del ladrón – Con esta capa podríamos hacernos invisibles durante unos instantes para no ser detectado por los enemigos y así poder evitarlos.

- Armadura de los titanes – Un objeto que nos volvería invulnerable durante un pequeño periodo de tiempo.

- Poder del viento – Desde el momento de conseguirlo nuestra espada lanzaría tornados, sirviéndonos para eliminar enemigos y para apagar fuegos mágicos en determinados sitios.

Loop de juego completo.

- El juego se entrega con un total de 30 escenas, si bien 4 de ellas son para pruebas o se desecharon en su momento, 26 entran en el loop del juego.

- Sistema de guardado y carga.

- UI completa, tanto in game como de menú.

- Pequeña historia con guion en 3 idiomas.

- Sistema propio de pathfinding.

Ilustración 9 - Nodos del pathfinding

1.3 Enfoque y método seguido

La idea inicial era muy clara, íbamos a hacer un tipo de juego que ya existe y del que hay varias muestras en el mercado. Decidimos arriesgarnos y hacer algo que ya está más que inventado como son los juegos de aventuras con un toque rolero, estos juegos tienen una legión de seguidores y la idea de hacerlo con un aire retro nos daba acceso también a los fans de este tipo de estética.
El toque de diferencia se lo intentamos dar con el trasfondo de la historia, si bien ya hay algunas obras que hacer referencia a la noche de las Walpurgis, como la

película española de 1970 La noche de las Walpurgis[4], o el tema musical del grupo Los carniceros del norte de 2012 también con un título parecido, La noche de Walpurgis[5], en lo referente a videojuegos no encontramos nada digno de mención, así que decidimos de tirar la narrativa en esa dirección ya que para la mayoría de la gente es algo relativamente desconocido.

Ilustración 10 - Cartel película La noche de las Walpurgis

1.4 Planificación del Trabajo

Una vez acabado el GDD y teniendo claro lo que queríamos hacer tocaba plantearse como nos íbamos a organizar, así que después de darle muchas vueltas decidimos que, por ser 3 personas para hacer el proyecto y tener horarios bastante dispares necesitábamos un sistema de control de versiones sí o sí. Casi siempre habíamos usado GitHub[6] como servidor de git pero queríamos que nuestro repositorio fuese privado así que al final nos decantamos por Bitbucket[7], que salvo por alguna diferencia menor es exactamente lo mismo y además nos permitía mantener nuestro repositorio en el anonimato . Una vez solucionado el tema del

repositorio necesitábamos algo para organizarnos y que de un vistazo supiésemos en que punto nos encontrábamos y cual era el siguiente paso a dar. Trello[8] parecía la herramienta perfecta, con su diseño por tarjetas podías asignar tareas individuales, marcar una fecha de finalización, quien debía de terminar el trabajo, subir archivos de apoyo, añadir subtareas en forma de checklist y un montón de funcionalidades más. No lo pensamos más, organizamos todo el trabajo en Trello, añadimos los dead lines, hicimos un código de color para las etiquetas y así de un vistazo saber a que área pertenecían y por último decidimos quien haría cada cosa.

1.5 Breve sumario de productos obtenidos

- 5 pantallas de juego.
- 3 idiomas.
- 26 escenas totales.
- 14 trampas y enemigos.
- 8 NPCs.
- UI completa.
- 4 Jefes de final de fase.
- Sistema de guardado y cargado.
- Sistema propio de pathfinding.
- Sistema de eventos por triggers.
- Guion para la historia principal y diálogos.

1.6 Breve descripción de los otros capítulos de la memoria

Este documento está formado por X capítulos, además del 1, que es donde nos encontramos, tenemos los siguientes:

- 2. Diseño de niveles, UI y escenas – En este capítulo tratamos todo lo referente a la construcción de las escenas que conforman nuestro juego, tanto de juego como menús y sus respectivas UI.

- 3. BGM, eventos e internacionalización – En este capítulo hacemos un recorrido por todo el sistema de triggers que controlan los eventos del juego,

el sistema de diálogos de los NPCs, la biblioteca de internacionalización usada y los temas musicales elegidos para el proyecto.

- 4. Lógica de juego – En este capítulo veremos las soluciones aportadas para controlar las mecánicas del juego, así como la inteligencia artificial usada, nuestro sistema de pathfinding propio y todo el loop de juego.

- 5. Conclusiones – Alegato final de la defensa del proyecto.

- 6. Bibliografía – Citas y reseñas de terceros usadas para ilustrar este documento.

- 7. Anexos – Documentos extra generados durante el desarrollo del proyecto.

2. Diseño de niveles, UI y escenas

2.1 El diseño de niveles

Durante la redacción del GDD ya se empezó a introducir tanto la descripción como el diseño de los niveles que compondrían el juego. Así, primero creamos una historia principal y a partir de esa historia imaginamos que lugares podríamos visitar que estuviesen en consonancia con nuestro trasfondo argumental. En un principio fueron 6 las pantallas que ideamos, todas con sus enemigos y desafíos propios, queríamos que tuviesen una continuidad pero que cada una fuese totalmente diferente, los enumeramos a continuación:

- La aldea – Este nivel existe solo como excusa para poder acceder a los demás, es decir, ideamos que nuestra historia transcurriese en un pasado cercano y que la forma de revivirla fuese a través de unas narraciones en una tradicional fiesta celta. Siempre que volviésemos al juego empezaríamos en la aldea, en el papel de un viajero que está de paso, y podríamos seguir escuchando la narración en cualquier momento y así seguir con nuestra aventura, pero ya de la mano de nuestro héroe principal, el joven Cedrick. En la aldea también podemos explorar y hablar con los habitantes. Al ser una pantalla de paso es bastante sencilla, un pequeño pueblo amurallado en forma de cruz, lo podemos explorar por completo en unos instantes.

- El bosque maldito – Esta es nuestra primera pantalla de juego propiamente dicho y es la que nos servirá a modo de tutorial por su sencillez y por lo escalado de su dificultad, ya que al principio los peligros serán una mera anécdota y poco a poco irán aumentando en cantidad y peligrosidad. El principal punto de cambio de dificultad sería recoger la espada ya que a partir de aquí los enemigos serían más agresivos.

Ilustración 11 - Plano cad nivel del bosque

- El mundo de las hadas – Mundo onírico que visitaríamos por un accidente mágico, a partir de aquí ya se introducirían las mecánicas de objetos que utilizaríamos durante todo el juego, ya que cada nivel tendría un objeto que habría que recoger y utilizar correctamente para poder superar dicho nivel, además en los siguientes niveles tendríamos la opción de usar los anteriores pudiendo así montar puzles interesantes.

- La isla de los piratas – Escapando del nivel anterior por un portal acabamos en esta isla de la que también debemos huir. El diseño de esta pantalla y de las siguientes es como en el mundo de las hadas, introducción de un nuevo objeto, reutilización de los anteriores y puzles relacionados, también tendremos la oportunidad de acabar o evitar a los enemigos, ya sea volviéndonos invisibles con la capa que recogeríamos en este nivel, aturdiéndolos con el casco y escapando o bien enfrentándonos a ellos con nuestra espada.

- La Fortaleza – Este nivel iba a ser un intermedio entre la isla y la montaña maldita, al final se decidió eliminar por falta de tiempo. El objeto de poder relacionado con esta pantalla aparecería en la montaña maldita para aprovechar su mecánica.

- La montaña maldita – Final de nuestro destino, en esta montaña se encuentra el enemigo a batir para restaurar el día y que la noche de las Walpurgis no vuelva a tener lugar.

Ilustración 12 - Pentagrama

Los niveles en un principio se diseñaron en papel y una vez que ya estaba claro que era lo que queríamos, se hacían en Autocad[9], estaban diseñados de manera que un solo nivel estaba en una sola escena, esto nos planteo un problema de rendimiento que descubrimos cuando el primer nivel, el bosque encantado estuvo acabado. El nivel era enorme y tenia una gran cantidad de trampas, enemigos y colliders, esto hacia que cayeran los FPS y diera una sensación como de ir flotando. Para evitar esto decidimos hacer los niveles en 5 escenas diferentes en vez de en una, esto no solo mejoró el rendimiento si no qué, además, al dibujar los niveles en porciones más pequeñas se hacia más rápido y ameno, por lo que al final cuando estuvieron todos terminados el boque también se volvió a hacer para que estuviese en 5 escenas como los demás. El nivel original del bosque aún está en el proyecto y se puede ver.

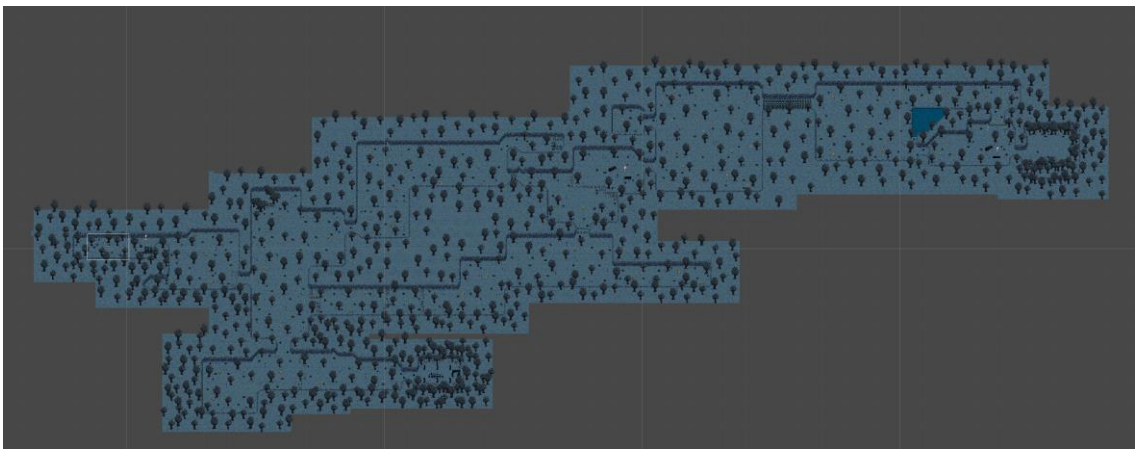


Ilustración 13 - Mapa original del bosque

2.2 Construcción de los niveles

Nuestro juego está desarrollado completamente con assets libres de uso, algunos solo para proyectos personales y otros incluso para profesionales. El proyecto incluye un README con todas las referencias del material utilizado.

Estuvimos semanas buscando assets que mantuviesen la coherencia tanto entre ellos como con nuestra idea, fue una labor tediosa ya que si bien, hay muchísimo material libre, es muy difícil que lo que hay quede bien entre ello, pero como no encontramos dibujante tuvimos que ponernos las pilas. En nuestro juego casi todas las escenas transcurrían de noche y no había lo que necesitábamos, así que al final tratamos varias hojas de sprites con Photoshop[10] para, aplicando capas, oscurecer las imágenes y hacer que pareciese de noche, funcionó bastante bien, así que repetimos el tratamiento de colores cuando nos hizo falta.

Una vez elegidos y tratados todos los assets para las pantallas elegimos la herramienta que incorpora Unity[11] desde la versión 2018, el tileMap, esta herramienta permite pintar el nivel utilizando varias capas creadas por nosotros según necesitemos. La principal ventaja de esta herramienta es la velocidad con la que se trabaja, se arrastra la hoja de sprites y se crea una paleta, desde esa paleta elegimos el tile con el que pintar y a partir de aquí utilizamos el puntero como pincel. Tiene funcionalidades muy útiles como el cubo de rellenado que va muy bien en ocasiones en las que tengamos que pintar una gran superficie con el mismo tile y otras como el borrado o el copiar selección.

Ilustración 14 - Aldea original

Ilustración 15 - Aldea filtro noche

Una vez dominado, los niveles salían como churros, pero no todo era así de bonito. TileMap incorpora un gameObject que nos permite seleccionar una de las capas de nuestro nivel y hacer que tengan collider para así, ahorrar muchísimo tiempo a la hora de declarar que partes son transitables y que partes no. Como hemos dicho más arriba nosotros tenemos un sistema de pathfinding propio que además es incompatible con esta forma de poner colliders ya que en el fondo tileMap es un elemento de canvas y nuestro pathfinding no lo reconoce. ¿Solución? Poner todos y cada uno de los colliders a mano, si los niveles los pintábamos super rápido, esto nos retrasaba de mala manera.
Una vez que nuestro nivel tenía colliders se añadían prefabs de trampas, enemigos, obstáculos y objetos, y con esto ya teníamos listo uno de nuestros niveles.

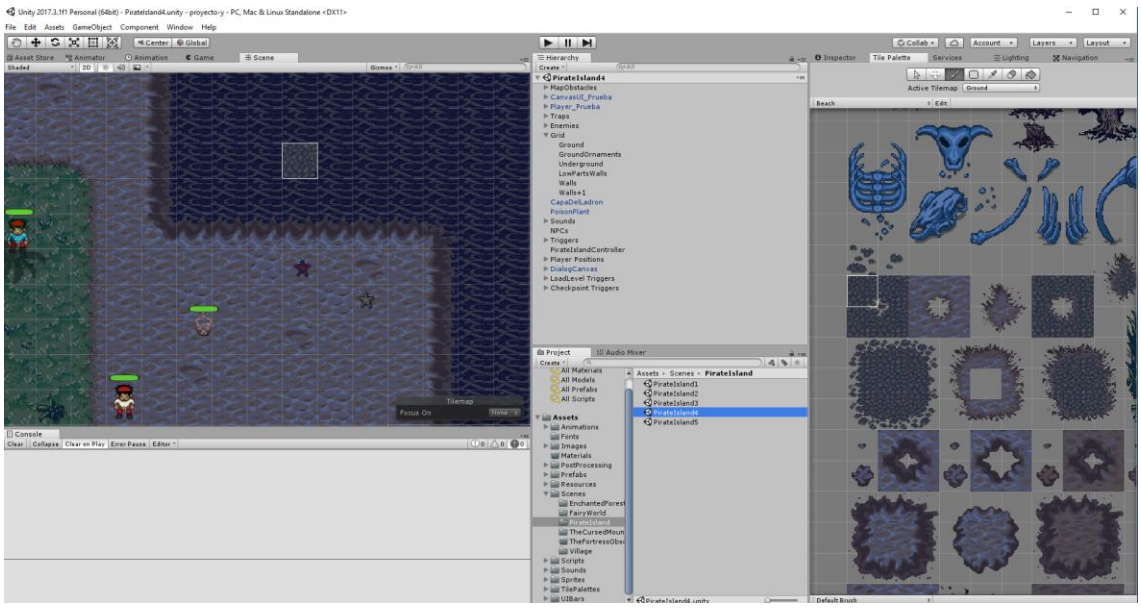


Ilustración 16 - Pintando un nivel con TileMap

2.3 Otras escenas

A parte de las escenas de juego nuestro proyecto contiene otras escenas de tránsito y de opciones:

- Escena de Splash – Arrancamos con esta escena mostrando el logo de la UOC y el logo de nuestro equipo.
- Escena de selección de idioma – El juego está traducido a tres idiomas seleccionables desde esta pantalla.
- Escena de título – Desde esta escena podremos abandonar el juego, empezar una nueva partida, cargar una existente o ver las opciones.
- Escena de game over – Veremos esta escena al ser eliminados y nos permitirá cargar la partida o volver al título.

- Escena de carga – Originariamente esta pantalla se creó por los problemas de carga y rendimiento que daban los niveles cuando no eran modulares, aún prevalece y muestra el progreso de carga entre niveles.
- Escena de créditos – Podemos acceder a ella desde las opciones o acabando el juego.

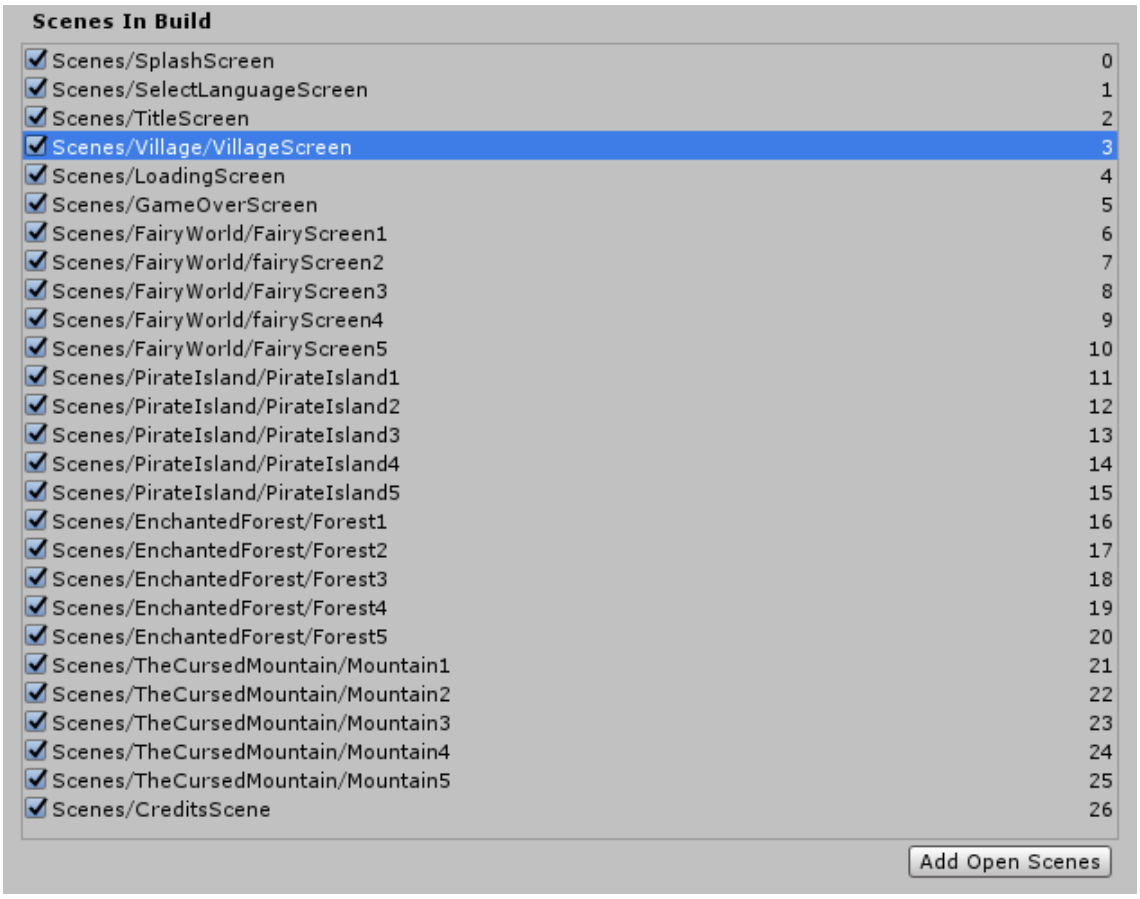


Ilustración 17 - Escenas

Casi todas permiten la transición entre ellas por botones actuables con el ratón.

2.4 Elementos de UI

La interfaz de nuestro juego está dividida en 2 categorías, la interfaz de juego y la interfaz de pantallas de tránsito.

- La interfaz de las pantallas de transito – Para estas pantallas hemos elegido una transitabilidad a base de botones actuables con el puntero del ratón. Por ejemplo, la pantalla de selección de idioma nos muestra 3 banderas, cada una es un botón actuable, al meter el puntero del ratón en una de ellas vemos que la bandera aumenta de tamaño ligeramente y a la vez suena un chasquido, si decidimos pulsarla suena un click y las demás desaparecen. La idea de esto es que el jugador reciba una señal clara de lo que está

haciendo, al hacer sonar el chasquido sobre una bandera acentuamos la sensación de estar haciendo algo y al aumentarla de tamaño señalamos claramente donde esta actuando el jugador, y finalmente al hacer click sobre ella lo confirmamos con otro sonido y al hacer desaparecer las otras dos damos la sensación final de confirmación de lo elegido.
El resto de botones del juego funcionan igual, solo que en vez de banderas tienen más aspecto de botón ya que llevan un texto explicativo de la acción que realizan, pero la forma de actuar sobre ellos es exactamente la misma.

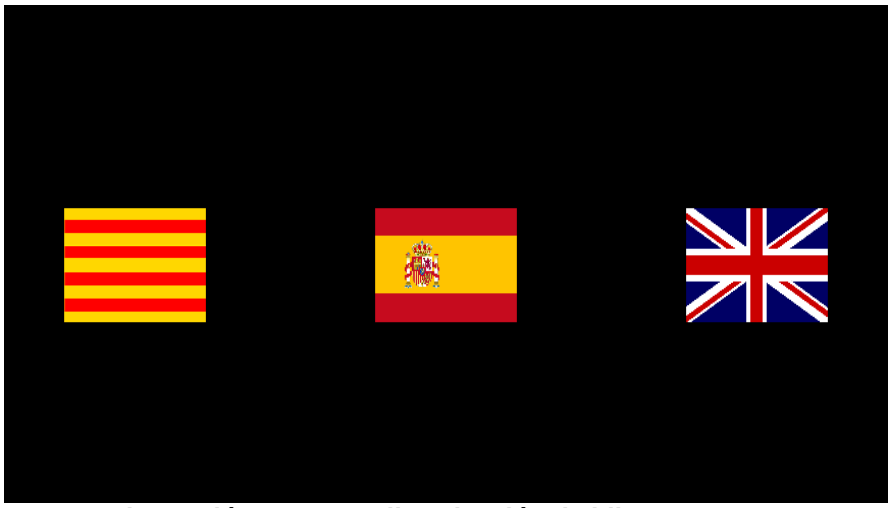


Ilustración 18 - Pantalla selección de idioma

- La interfaz del juego – Para el juego hemos elegido una interfaz sencilla y que deje clara nuestra situación y las acciones que podemos llevar a cabo. En la parte inferior izquierda de la pantalla tenemos dos indicadores de barra, uno rojo para representar la vida y uno azul para representar el maná, no es necesario indicar cual es cual con textos ya que los colores asociados son universales y no dan pie al equivoco. Estas barras reaccionan según la vida o maná que tengamos, la de vida parpadea cuando tenemos poca mientras que la de maná lo hace cuando la tenemos a tope.
  Justo a la derecha de estas barras se alinean varios iconos que representan las acciones que podemos llevar a cabo y además llevan la letra asociada del teclado que los activa para tener claro en todo momento como hacerlos funcionar. Estos iconos se irán activando en la interfaz de juego según vayamos consiguiendo la habilidad asociada, así pues, al empezar el juego no habrá ninguno y ya casi al final de este tendremos 5 iconos.

Ilustración 19 - Interfaz de juego

Durante las peleas con los jefes se activará en la parte superior izquierda un retrato de este junto a su nombre y barra de vida.
Al pausar el juego tendremos una ventana emergente con botones actuables como los que hemos visto anteriormente.

3.BGM, eventos e internacionalización

3.1 Selección de músicas

Para esta tarea se buscan piezas que tengan algún elemento cercano al folklore celta. Por suerte, en la FMA (Free Music Archive) estaba disponible la música de Sláinte, uno de los grupos pioneros de la cultura copyleft. Entre todas las melodías, se intentó descartar las que contenían letra cantada puesto que podían despistar un poco al jugador y se busca un sentimiento en cada una de las piezas. Algo que conecte con el nivel o la pantalla correspondiente. Por ejemplo, para Fairy World se usa una melodía algo melancólica puesto que se trata de un mundo fantástico pero en decadencia (la reina ha perdido la cabeza).
Con las luchas contra jefes hubo más problemas porque era necesario que la melodía fuera más energética, incitando al jugador a la acción, a la premura. Para esto se buscan melodías electrónicas, que muchas veces imitan al folk y tienen ese toque vintage sin dejar de ser músicas contemporáneas. El oído sibarita notará un cambio quizás un poco discordante, pero se ha intentado que no desentonaran demasiado entre ellas.
Desgraciadamente la selección inicial, con la que quedamos bastante satisfechos, se vió afectada por la retirada de la música de uno de los artistas (Art of Escapism) de FMA. Por lo visto consiguieron un contrato con una discográfica y retiraron su música de Internet. Se tuvo que improvisar una solución y quizás el resultado es algo menos impresionante de lo que podría haber sido. Finalmente seleccionamos algunas canciones de Damiano Baldoni y de Dee Yan-Key que dan el pego.
A la hora de implementarlo, primeramente se pensó utilizar un objeto en cada escena que reprodujera la música. Pero cuando se divieron los niveles en secciones, se vio la necesidad de crear una experiencia más continua ya que la pantalla de carga era muy aburrida si simplemente aparecía ahí sin más. Por esta razón se crea un objeto único en la pantalla de selección de título que, a través de la opción DontDestroyOnLoad queda en memoria durante toda la ejecución.
Este objeto contiene un script BGMManager y un listado de canciones que se van cargando en el AudioSource según en qué nivel o pantalla nos encontramos, además de cambiar a una canción de boss si es que estamos en una lucha contra un boss.

Ilustración 20 - Lista de temas

De esta manera la pantalla de carga sigue teniendo un elemento musical que siempre ameniza la espera y, en el momento de ir al siguiente nivel, se cambia la melodía a la que corresponda creando una experiencia sin cortes.

3.2 Lógica del juego

Siguiendo la directiva de facilitar la vida al equipo, se construyen los cimientos del juego entorno a una clase estática. Esta clase estática no es accesible desde el Editor, pero está ahí detrás, accesible desde todas partes y, quizás lo más importante, es única. Muchas veces para implementar este tipo de comportamiento se abusa del patrón Singleton cuando realmente no es necesario. La envergadura del proyecto es mínima, por lo que una clase estática es una solución muy sencilla y, a la vez, poderosa y flexible de construir esta base. En nuestro caso, esta clase se llama Game.
Además, sirve de punto de encuentro para los diferentes equipos. Imaginemos que un programador está escribiendo el código de los Bosses mientras que otro está haciendo los Triggers. Bueno, no hace falta imaginarlo, es lo que ha pasado en este proyecto :P Ambos están trabajando en sistemas independientes entre sí pero que pueden necesitar comunicación de algún tipo. Por ejemplo:
El Boss X muere y, a su muerte, deja caer una llave. Esa llave es la que abre la puerta para poder salir del nivel.
Esta acción requiere de varios elementos:
- Un Boss (programado por A)
- La llave (creada por un diseñador y guardada en un prefab)
- Un diálogo que salte a la muerte del Boss (programado por B)

Tres personas trabajando en cosas independientes pero que, en última instancia, han de ver su trabajo coordinado. Hay muchas maneras de realizar esto, algunas más complejas que otras, pero con la clase estática se puede conseguir de manera muy sencilla
1. El Boss realiza sus tareas mientras su valor de vida es superior a 0. Cuando llega a 0, muere y el programa debe gestionar esta muerte y todas sus consecuencias.
2. Como Game está disponible desde cualquier punto del código, el script del Boss puede llamar a una variable (Game.bossDied) o a un método

(Game.SetBossDied()) e informar de la muerte. Evidentemente, el método o la variable debe ser público. Una vez hecho esto el objeto del Boss ya se puede eliminar, desactivar o lo que se crea conveniente.
3. Una vez Game ya sabe que el boss ha muerto, se pueden activar triggers que a) hagan aparecer la llave y b) haga saltar el diálogo. El script correspondiente puede consultar la misma variable o método de Game desde, por ejemplo, su método Update().
Pero esto genera otro problema: ¿y qué pasa si la acción sólo ha de ejecutarse una vez? Los tres puntos anteriores, a priori, parece que funcionaran sin problemas. Pero ¿qué pasa si se tiene un código como éste?

```
private void Update()
{
    if(Game.bossDied)
    {
        object.SetActive(!object.activeSelf);
    }

}
```

Pues pasa que, suponiendo 60 FPS, el gameObject estará activándose 30 veces por segundo y desactivándose 30 veces por segundo. ¿Cómo solucionamos esto? Con una máquina de estados.

```
private void Update()
{
    switch(currentState)
    {
        case (ObjectState.WAITING):

            if (Game.bossDied)
            {
                object.SetActive(!object.activeSelf);
            }
            currentState = ObjectState.DONE;
            break;
    }

}
```

En este nuevo ejemplo desactivamos el objeto si estamos en estado WAITING. Y sólo lo desactivaremos una vez porque justo después cambiamos el estado a DONE. De esta manera nos aseguramos que la acción se realiza una sola vez. Parece contraproducente, pero el hecho de no saber en qué orden se van a ejecutar los scripts puede crear mucha confusión.

```
public class Game : MonoBehaviour {

    // Vars
    public static GameState currentState = GameState.UNINITIALIZED;
    public static GameState lastState;
    public static SaveGame savedGame = new SaveGame();
    // Creamos un save temporal para checkpoints, si morimos deberíamos resetear el savedGame con el checkpointSave
    public static SaveGame checkpointSave = new SaveGame();
    public static CoreConfig config = new CoreConfig();
    public static int currentSaveSlot = 0;
    public static bool isPlayerFrozen = false;
    public static bool loadedFromSaveGame = false;
    public static bool isBossFight = false;
    //public static string currentLevel = Constants.LEVELS_NAME_SCREEN_VILLAGE; // por defecto, la aldea
    public static Player player;

    public static I18n i18n = I18n.Instance;
```

Ilustración 21 - Estado del juego

3.3 Guardado

Al principio los videojuegos ni se planteaban guardar los datos de juego y, una vez perdidas todas las vidas o continuaciones, el juego terminaba obligando al jugador a empezar de cero. Esto permitía que juegos sencillos en cuanto a planteamiento tuvieran una vida muy extensa basada en la dificultad. Los jugadores podían intentar batir el récord de puntos, acabar el juego sin perder una sola vida (o combinar ambos para la partida perfecta), y otros retos similares.
Algunos juegos eran endiabladamente difíciles y han pasado a la historia por ello. Y es un equilibrio difícil de mantener porque al final la manera más simple de elevar la dificultad de un videojuego es permitir a la IA hacer trampas. La sensación de ganar a un juego que hace trampas no es la misma que la de ganar a un juego legítimamente difícil de batir.
Todo esto cambió cuando se empezaron a crear videojuegos que permitían guardar la partida. Algunos géneros se adaptan muy bien a lo comentado anteriormente (las recreativas, los juegos arcade,…) pero hay otros en los que no es saludable adentrarse sin una opción de guardar el progreso (RPG, aventuras gráficas, etc). La dificultad se relajó un poco, aunque no está reñida con el guardado, puesto que los gustos también cambiaron. Se pudieron crear, entre otras cosas, juegos más largos (algunos RPG pueden tener cientos de horas de juego).
Para nuestro juego, siguiendo la línea de no complicarse la vida, se ha optado por guardar un objeto en Game con una serie de campos que contienen información sensible de la partida. ¿Qué ventajas tiene esta forma de proceder?
- Fácil de resetear. Tan simple como hacer uso de new.
- Centralizado. Todos los equipos saben dónde ir a buscar o dejar la información.
- Fácil de duplicar. Por ejemplo, para guardados temporales.

Una de estas ventajas permite implementar la funcionalidad de los checkpoint muy fácilmente. Tan sólo hay que guardar otro objeto del mismo tipo dentro de Game. Cuando se llega a un checkpoint, se guarda el estado actual en el objeto temporal. En caso de morir, el proceso inverso. Cargamos los datos del checkpoint en el estado actual, revirtiéndolo al momento del checkpoint. Esto es parecido a lo que hacen los emuladores para guardar estados rápidamente. Los emuladores de hardware antiguo guardan una copia del estado actual de la memoria de la máquina y, para volver a él tan sólo tienen que restaurar esa copia tal cual.

El tipo SaveGame guarda información relativa al estado del juego, de los triggers y algunas estadísticas de los NPC que se podrían usar en una pantalla de datos curiosos:

```
[System.Serializable]
public class SaveGame
{
    public string currentLevel;
    public string lastLevel;

    public bool isNewGame = true;
    public GameState gameStatus;

    public bool[] villageTriggerStatus;
    public int[] villageNPCDialogCount;


    public bool[] fairy1TriggerStatus;
    public int[] fairy1NPCDialogCount;

    public bool[] fairy2TriggerStatus;
    public int[] fairy2NPCDialogCount;

    public bool[] fairy3TriggerStatus;
    public int[] fairy3NPCDialogCount;

    public bool[] fairy4TriggerStatus;
    public int[] fairy4NPCDialogCount;

    public bool[] fairy5TriggerStatus;
    public int[] fairy5NPCDialogCount;

    public bool[] pirate1TriggerStatus;
    public int[] pirate1NPCDialogCount;

    public bool[] pirate2TriggerStatus;
    public int[] pirate2NPCDialogCount;

    public bool[] pirate3TriggerStatus;
    public int[] pirate3NPCDialogCount;

    public bool[] pirate4TriggerStatus;
    public int[] pirate4NPCDialogCount;

    public bool[] pirate5TriggerStatus;
    public int[] pirate5NPCDialogCount;

    public bool isOnVillage = false;

    public bool hasFoundKey = false;
    public bool hasFoundHelmet = false;
```

```
    public bool hasFoundCloak = false;
    public bool hasFoundArmor = false;

    public bool hasFoundHerbs = false;

    public bool hasCompletedLightTemple = false;
    public bool hasCompletedFireTemple = false;
    public bool hasCompletedWindTemple = false;
    public bool hasCompletedLavaTemple = false;

    public bool neverVisitedFairyWorld = true;
    public bool neverVisitedPirateIsland = true;
    public bool neverVisitedEnchantedForest = true;
    public bool neverVisitedMountain = true;


        ...
}
```

En el savegame guardamos algunas variables acerca del estado de algunos elementos del juego. Por ejemplo, hasFoundKey será true si se ha encontrado la llave del nivel FairyWorld. Con esta información podemos decirle a la barrera correspondiente si debe desactivarse o no, puesto que depende de dicha llave. También tenemos un paquete de variables que nos dicen qué niveles ha visitado el jugador, de forma que la Matriarca sabe qué niveles debe mostrar. En el savegame guardamos también el estado de los triggers, de forma que podemos reiniciarlos al cargar la escena.

Por otra parte el objeto Game puede tener otras variables volátiles como por ejemplo si el sistema de diálogos está activo. No hay que obsesionarse con guardarlo todo, tan sólo aquella parte de los datos que no se pueden regenerar durante la ejecución.
Unity permite guardar y cargar datos de forma muy sencilla usando JSON. Para que JSONUtility reconozca la clase hay que hacerla Serializable y utilizar tipos de datos soportados por esta clase. El resultado es que podemos guardar y cargar partidas sin demasiado esfuerzo. La pega es que los ficheros son fácilmente editables, pero para cumplir el objetivo de este proyecto nos sirve.

Había previsión de usar varios slots de guardado aunque al final no se ha utilizado. Para conseguir esto tan sólo hay que pasar un parámetro a la hora de guardar el fichero (por ejemplo, un numero del 0 al 9) de forma que se guarde en un nombre de fichero diferente (por ejemplo, savegame0 … savegame9).

Se barajó también la posibilidad de tener escenas duplicadas para poder cubrir algunos escenarios. Por ejemplo, una escena para la primera visita a la aldea (con un guía o una serie de eventos automáticos) y otra escena exactamente igual pero con libre albedrío. Pero era mucho más sencillo guardar el estado de todo y poderlo cargar desde disco.

3.4 Triggers y NPCs

Aunque se planteó al principio crear un sistema de Eventos independientes, al final resultó más fácil usar triggers independientes. En un proyecto más grande deberíamos pensarlo dos veces, puesto que la complejidad se multiplica exponencialmente. Para un proyecto pequeño y con tiempo limitado, no hace falta (ojo, aunque puede ser recomendable) construir un sistema independiente de eventos.
Se crea una clase base de Triggers que, como se ha comentado antes, quizás necesite otra vuelta de hoja para acabar de especializarla pero que cubre el caso general (trigger automático) con algunas propiedades típicas.

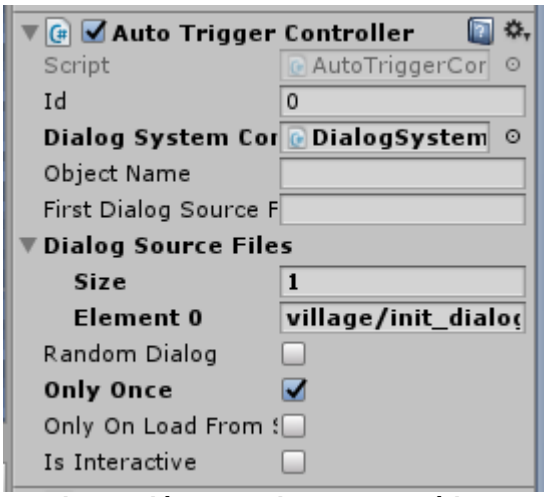


Ilustración 22 - Trigger automático

A partir de esta clase base, otras clases heredan y complementan funcionalidades como por ejemplo un trigger que varía el status de un objeto o uno que vigila varios objetos. ¿Por qué le falta un poco de especialización? Porque originalmente estos triggers se crearon para uso exclusivo del sistema de diálogos. Pero luego se vió que su utilidad podía ser mucho más diversa y, sobretodo al descartar el sistema de Eventos, pasó a ser más genérico.
Existen los siguientes tipos de triggers:

Nombre	**Descripción**
AutoTrigger	El trigger automático de toda la vida.
CheckpointTrigger	Un trigger que guarda la partida temporalmente.
CombinedTrigger	Este trigger puede vigilar el estado de una serie de objetos para saber cuándo puede actuar.
LoadLevelTrigger	Un trigger para cargar un nivel o sección de nivel
NPCController	Trigger para controlar NPCs

StatusTrigger	Este trigger permite cambiar el estado de un objeto
VariableStatusTrigger	Este trigger sirve para cambiar algunas variables importantes de Game

Con estos triggers cubrimos una serie de situaciones que nos permiten construir el juego según el GDD. El original es el NPCController que se creó para gestionar la interacción con NPCs. Más tarde, cuando se decidió dividir los niveles en fragmentos, hubo la necesidad de crear otro tipo de trigger y se estaban creando duplicidades. Por esta razón se decide reimaginarlo todo y crear un trigger básico del que hereden todos los demás.
Aunque ya se ha insistido suficiente en que le falta un hervor, también cabe decir que los triggers actuales permiten gestionar algunas casuísticas interesantes como, por ejemplo, cargar un nivel una única vez. Pero realmente lo ideal sería que el trigger base contenga la mínima funcionalidad posible, todo aquello que es común en todos los casos, lo que define a un trigger; dejando el resto de cosas para las diferentes especializaciones. Todos los triggers aparecen fruto de una necesidad: cargar un nivel, activar una variable, etc. Esa necesidad define unos parámetros y, a partir de ahí, deberíamos escribir el objeto que hereda.

3.5 AutoTrigger

El AutoTrigger se activa al detectar que el player entra o aparece dentro de su collider. Es el trigger básico y ejecuta la acción por defecto. En nuestro caso la acción por defecto es mostrar un diálogo. Sus propiedades son:

Propiedad	**Descripción**
Id	Define un id único para el trigger. Este id se utiliza para asegurar una posición en el vector de triggers de SaveGame.
DialogSystemController	Referencia a la instancia de DialogSystemController presente en la escena.
Object Name	Nombre del objeto
First Dialog Source File	Path del archivo que contiene el diálogo que se debe mostrar la primera vez que se habla con el NPC. Si no se define, se utiliza el Dialog Source Files.
Dialog Source Files	Array de paths de los archivos que contienen los diálogos que se deben mostrar al hablar con el NPC.
Random Dialog	Si está activado, recorrer el Dialog Source

	Files aleatoriamente.
Only Once	Activar el trigger una única vez.
Only On Load From Save Game	Activar únicamente al cargar la partida
Is Interactive	Si está activado, se activa únicamente al pulsar Intro.

3.6 CheckPointTrigger

Este trigger actualiza la partida temporal. Sus propiedades son:

Propiedad	**Descripción**
Notification System	Referencia al NotificationSystem presente en la escena.
Default Status Trigger State	El estado por defecto que debe tener el Checkpoint al iniciarse.
Status Trigger State	Estado actual.

3.7 CombinedTrigger

Este trigger tiene dos listas de triggers. Una contiene los triggers que debe vigilar para saber si se tiene que actualizar o no. Mientras no estén todos en estado DONE, no hará nada. La otra lista contiene los triggers que debe activar una vez los triggers de la primera lista estén finalizados. Actualmente es compatible con StatusTrigger, AutoTrigger y VariableStatusTrigger.

Propiedad	**Descripción**
Blocked By Triggers	Lista de triggers que bloquean el CombinedTrigger.
Triggers To Activate	Lista de triggers que se deben activar cuando los triggers de “Blocked By Triggers” estén finalizados.
Default Status Trigger State	El estado por defecto que debe tener el CombinedTrigger al iniciarse.
Status Trigger State	Estado actual.

3.8 LoadLevelTrigger

Este trigger simplemente carga una sección/nivel. Cuando el jugador entra o aparece dentro del collider del trigger se carga la escena SectionName a través de Game.LoadLevel. Sus propiedades son:

Propiedad	**Descripción**
SectionName	Nombre de la escena a cargar.

3.9 NPCController

Este trigger controla un NPC. Hereda de AutoTrigger la capacidad de gestionar los diálogos y, además, permite que el NPC se mueva por el mapa. Se pueden definir una serie de waypoints para formar una ruta que el NPC puede recorrer. Además, hay parámetros que definen la velocidad con la que se va a mover y si debe moverse. También controla las animaciones.

Propiedad	**Descripción**
Default Animation	Animación que se debe ejecutar por defecto
Waypoints	GameObject con los waypoints de la ruta que sigue el NPC.
Speed	Velocidad de desplazamiento.
Patrol Waiting Time	Tiempo que espera el NPC al llegar a un waypoint
IS Moving	Si está activado, el NPC se moverá por la ruta definida por sus waypoints
Infinite Patrol	Si está activado, cuando llegue al último waypoint de la ruta, se dirigirá al primero. Es importante intentar que el flujo de la rutas sea circular.
Random Idle Animation	Si está activado, cuando el NPC se pare se elegirá una animación por defecto de entre las animaciones Idle.

3.10 StatusTrigger

En este trigger podemos guardar una lista de Components y otra de GameObjects a los que queremos cambiar el estado. Cuando el trigger se active, todos los Component y GameObjects de las listas invertirán su estado actual. Es decir, si estaban activados pasaran a desactivados (y viceversa)

Propiedad	**Descripción**

Component List	Lista de Component que queremos gestionar.
GameObject List	Lista de GameObject que queremos gestionar.
Default Status Trigger State	El estado por defecto que debe tener el CombinedTrigger al iniciarse.
Status Trigger State	Estado actual.
Is Interactive	Si está activado, se activará al pulsar la tecla Intro.

3.11 VariableStatusTrigger

Este trigger se utiliza para activar/desactivar variables importantes de Game.

Propiedad	**Descripción**
Has Completed Light Temple	Activar/desactivar hasCompletedLightTemple
Has Completed Fire Temple	Activar/desactivar hasCompletedFireTemple
Has Completed Wind Temple	Activar/desactivar hasCompletedWindTemple
Has Completed LavaTemple	Activar/desactivar hasCompletedLavaTemple

3.12 Notificaciones

Aunque no estaba planeado al principio, era necesario informar al usuario de algunas acciones (principalmente de que su partida se ha guardado). Por esta razón se improvisa un sistema de notificaciones muy sencillito que consiste en un texto que se muestra durante unos segundos. Sólo hay que pasarle el texto que se quiere mostrar y llamar a ShowNotification(). Se puede utilizar para cualquier cosa que requiera mostrar un mensaje en pantalla temporalmente. Por ejemplo: los checkpoints o los logros.
De haber tenido tiempo habría estado bien decorar un poco la UI (por ejemplo, añadir iconos) pero cumple su función y estuvo listo en unos minutos, que es lo importante aquí.

3.13 Diálogos

El sistema de diálogos dió muchos quebraderos de cabeza porque se quería hacer algo funcional, flexible pero a la vez poderoso. Los prototipos se complicaban bastante por lo que al final se optó por simplificar y basarlo también en JSON. Primeramente pensábamos que no se podía hacer porque JSONUtility no parsea tipos complejos pero luego descubrimos un pequeño truco.

```
{
	"list": [
		{
			"text": "¿Ya has encontrado la espada?",
			"portrait": 5,
			"textSize": 32,
			"characterName": "Viejo druida"
		}
	]
}
```

El uso de un elemento llamado “list” permite que JSONUtility cargue elementos a objetos List<T>. Con esto podíamos crear un sistema de diálogo basado en JSON flexible y bastante poderoso. Sólo es necesario una clase con la lista y un objeto que defina sus elementos.

```
using System.Collections.Generic;

public class DialogList
{
   public List<Dialog> list;
}
```

Sin esta clase JSONUtility no es capaz de parsear los diálogos. Luego, en Dialog se pueden establecer todos los campos que se crean convenientes. En nuestro caso tenemos los siguientes:

Campo	**Tipo**	**Descripción**
Text	String	Frase a mostrar.
Portrait	Int	Índice del retrato en DialogSystem
TextSize	Int	Tamaño de letra
CharacterName	String	Nombre del personaje
Answer1	String	Texto de la respuesta 1
Answer2	String	Texto de la respuesta 2
Answer3	String	Texto de la respuesta 3
Reaction1	String	Texto de la reacción a la respuesta1
Reaction2	String	Texto de la reacción a la

		respuest2
Reaction3	String	Texto de la reacción a la respuest3
Requisite1	String	Requisito de la respuesta1
Requisite2	String	Requisito de la respuesta2
Requisite3	String	Requisito de la respuesta3
MoveNPC	Int[]	Array con los indices de los NPC que se deben mover
NeedsHelmet	Bool	Saltar si aún no se tiene el casco.
NeedsKey	Bool	Saltar si aún no se tiene la llave.
NeedsCloak	Bool	Saltar si aún no se tiene la capa.
NeedsArmor	Bool	Saltar si aún no se tiene la armadura.
NeedsHerbs	Bool	Saltar si aún no se tiene la hierba.
NeedsLightTemple	Bool	Saltar si aún no se ha completado el templo de la luz
EndDialog	Bool	Finaliza el diálogo inmediatamente.
MovePlayer	float[]	Mover al jugador (x,y)

Con todos estos elementos podemos gestionar diferentes situaciones. Especialmente a través de los campos reactionX, que permite ejecutar comandos como por ejemplo newlevel o jump. Con estos comandos se puede manipular el diálogo pero también el estado del juego. Los comandos disponibles son:

Comando	**Descripción**
ActivateTomb	Activa una de las tumbas del nivel FairyWorld. Se le debe pasar un parámetro con el nombre de la tumba.
Stop	Detiene el diálogo
Continue	Pasa al siguiente elemento del diálogo
Jump	Salta X elementos del diálogo. Se le debe pasar el número de saltos como parámetro.
Newlevel	Carga nivel/sección. Se le debe pasar el nombre de la escena como parámetro.

Savegame	Guarda la partida.
exit	Sal al título.

Estos comandos se ejecutan al seleccionar una de las respuestas, que se muestran como botones en pantalla.

Una de las ventajas de este sistema es que podemos editar los diálogos mientras el juego se está ejecutando, pudiendo ver las modificaciones en tiempo real y también hacer diferentes pruebas sin tener que reiniciarlo.

3.14 Internacionalización

Para la internacionalización se usa una librería externa llamada i18n muy simple que permite cargar las traducciones desde JSON. Se define un JSON para cada idioma y luego se utilizan parejas clave-valor para definir los textos.

Ilustración 23 – JSON

Este sistema es muy sencillo y se usa a través de un objeto que incluye Game llamado i18n (casualmente). Simplemente se llama a un método al que se le pasa la clave y la librería te devuelve el valor correcto según el locale seleccionado. Permite cambiar el locale on-the-fly por lo que podemos cambiar el idioma desde el menú de opciones, por poner un ejemplo.
Para la internacionalización de los diálogos esta solución no era idónea, por lo que tuvimos que pensar un sistema híbrido. La librería i18n nos dice qué locale estamos usando y a partir de ahí definimos unas rutas diferentes para cada idioma en las que guardamos ficheros JSON que se llaman igual. Sólo hay que montar las rutas a la hora de cargar usando el código de locale que nos diga i18n (por ejemplo es para Español → Dialogs/es/fichero.json).
De esta manera podemos traducir todos los díalogos del juego editando simples ficheros de texto y sin tener que desarrollar un nuevo sistema de internacionalización.

3.15 Otras cuestiones

- Lo que hemos aprendido:
  A primera vista, puede parecer fácil crear un videojuego. De alguna manera, es una aplicación como cualquier otra. Pero la realidad no es tan simple, puesto que una aplicación cualquiera no necesita sorprender ni retar al usuario. Tampoco necesita generar sentimientos ni recuerdos épicos. Simplemente ha de cumplir una función: hacer una foto, reproducir un vídeo, etc.
  Pero un videojuego va más allá. Es una experiencia de ocio interactiva que además, de estar bien hecha, es fuertemente immersiva. Puede hacer que

perdamos la noción del tiempo y, desde luego, que queramos volver a su universo. Como un buen libro o una buena película, pero haciéndonos partícipes de la acción (o la inacción).
El núcleo es el game loop, que debe tener en cuenta unas especificaciones muy especiales. Por ejemplo, debe intentar ceñirse a una cantidad de imágenes por segundo (30 o 60 son los más usados) como si de una película se tratara. De esta forma el ojo humano tiene una sensación de continuidad, al bajar de 24-25 fps el ojo humano es capaz de percibir los saltos de imagen. Pero eso no es todo, puesto que un video se podría ir guardando en un bufer de memoria para ganar algo de tiempo de procesamiento. En el caso del videojuego esto no sirve porque hay que dar respuesta immediata a las acciones del jugador. Esto nos deja con un tiempo máximo para procesado entre imagen e imagen.
Por estas razones, es muy importante que los diferentes elementos del videojuego estén funcionando correctamente y bien engrasados, generando una experiencia creíble y sin interrupciones derivadas de errores o malos funcionamientos. De otra manera, el jugador pierde la sensación de inmersión y puede también perder el interés.
Para ello es muy importante tener un buen diseño al que acudir cuando no se tienen las cosas claras o, por ejemplo, cuando ya hace un par de días que no se toca un área concreta del proyecto. También es clave poner en claro unas ideas o unos procedimientos estándar en los que todo el equipo se pueda apoyar cuando tenga dudas, homogeneizando lo máximo posible el trabajo de todos. Si uno tiene que corregir o ampliar el trabajo del otro, debe poder entenderlo lo más rápido posible puesto que así será más efectiva la búsqueda de la solución.

- El GDD:
  Quizás el documento más importante para definir la idea y el objetivo. Muchas veces se pasa por alto su importancia, creyendo que el resultado puede ser similar. Lo cierto es que en proyectos pequeños (principalmente los de una sola persona) puede que sea cierto, porque al final se va improvisando sobre la marcha. Pero lo ideal sería definir primero de forma exacta qué se quiere conseguir para tener una referencia futura. Y esta referencia es el GDD. Incluso para el equipo de programación es una herramienta esencial para poder recordar o definir los diferentes aspectos del juego.
  A primera vista puede parecer algo meramente formal, pero en etapas avanzadas muchas veces cuesta recordar para qué era aquél objeto o qué rutas se habían definido para aquél otro objetivo. Y en esos momentos, los documentos de diseño sirven para refrescar la memoria así cómo para resolver dudas y/o conflictos. Es el reglamento de juego.
  En nuestro proyecto no se ha hecho, pero ahora sabemos que sería muy positivo apoyar este GDD con otro documento de diseño de programación.
  En aplicaciones tradicionales se suele incluir en el documento de diseño el esquema ER (si procede) así como una especificación de los diferentes objetos. Para el caso de los videojuegos no sería necesario centrarse tanto en la especificación de objetos si no, más bien realizar un estudio de qué patrones y herramientas se ajustarán mejor a lo que vamos a hacer. Un documento que especifique los diferentes sistemas, así como los estándares

a seguir y las diferentes convenciones que se acuerden (notación, estilo, etc).
De no tener este documento, se tiende a ir cada uno por su propio camino y al final se crean incompatibilidades o, lo que es peor, “imprescindibilidades”. Es decir, necesito a X para que haga esto porque nadie más sabe hacerlo. Es importante que el equipo sea lo más flexible posible. Aunque cada uno esté a una tarea, es muy útil que todo el mundo se pueda adaptar rápidamente.
En nuestro caso hay que decir que nos hubiera ido muy bien tener este documento, así como el GDD nos ha ayudado en muchas ocasiones. Aunque pueda parecer una pérdida de tiempo (y quizás en calendarios muy ajustados lo pueda ser), después se gana mucho tiempo en resolución de dudas y conflictos.

- Control de versiones:
  Trabajar en grupo provoca desastres. Para evitarlos: control de versiones.
  Sistemas como git o svn permiten tener un control de todos los cambios que se realizan en unos ficheros concretos. Además, permiten sincronización, con lo que se pueden crear copias remotas de los repositorios. A partir de estas copias, otros programadores pueden descargar el trabajo realizado o cargar mejoras. Además, permiten branching con lo que podemos crear diferentes “ramas” de nuestro código que pueden ser para nuevas funcionalidades o incluso para nuevas versiones escritas de cero.
  Pero el control de versiones, aunque muy poderoso, no es infalible. Hay que establecer unas pautas de trabajo que todo el mundo debe aprender o puede haber desastres irrecuperables. Esta es otra de las cosas que podrían incluirse en el documento mencionado anteriormente.
  Durante el proyecto ha habido algunos “accidentes” leves causados principalmente por desconocimiento. El control de versiones, bien gestionado, permite solucionar accidentes bastante graves puesto que tenemos copia de los diferentes cambios realizados. En caso de perderlos o corromperlos, podemos volver a un estado anterior o recuperarlo de otra rama. Siempre y cuando se haya ido sincronizando los cambios paulatinamente. Además, el hecho de tener una copia remota y X copias para X usuarios, complica la pérdida de información. Pero puede darse el caso que no se hayan guardado los cambios desde hace mucho tiempo o que se mezclen versiones de archivos complicando un poco la recuperación. En algunos casos, especialmente si se trata de archivos binarios o que no son de texto, tocará repetir la faena.

- Motor:
  Se escoge Unity3D como motor de juego puesto que nos permite centrarnos en la creación del videojuego en sí abarcando la mayor cantidad de dispositivos y configuraciones.
  Se estudia durante un tiempo si se puede optimizar el rendimiento usando diferentes patrones pero, al final, se decide simplificar puesto que Unity ya implementa algunos de estos patrones (p.ej: Component) y no hace falta reinventar la rueda.
  Por ello se intenta seguir tres convenciones:
  - Facilitar la vida a los diseñadores.

- Intentar que todos los elementos sean independientes unos de otros.
- Lo más fácil posible

Para la primera se intenta que los scripts se puedan configurar lo máximo posible desde el Editor de Unity. De esta manera, el equipo de diseño puede hacer diferentes pruebas en tiempo de ejecución y anotar los ajustes que haya que hacer y los bugs que encuentren para notificárselos a los programadores. Además, hay que intentar que sean suficientemente flexibles para cubrir diferentes situaciones. Esto se puede conseguir a través de herencia.

Para la segunda se debe tomar como punto de partida el ciclo de vida de Unity. Sabiendo que Unity maneja una lista de objetos que va gestionando en los diferentes momentos de su ciclo de vida es fácil imaginar cómo hacer que los componentes sean independientes. Básicamente se necesita una máquina de estados. Teniendo una referencia de en qué estado se encuentra el juego (o cualquier otro objeto) podemos manejar la situación en cada objeto de forma individual a través de los métodos que proporciona Unity (Awake(), Start(), Update(), …).

La tercera es evidente. Se trata de intentar buscar la solución más sencilla posible, sin complicarse la vida con sistemas complejos. Es un proyecto pequeño y, con un calendario tan ajustado, el problema principal no va a ser el rendimiento.

Muchas veces no se ha conseguido ajustarse a estas convenciones ya sea por las prisas o por falta de análisis. Como apuntábamos antes, nos hubiera ido bien dedicar un tiempo a establecer unas bases para la programación igual como se hizo con el diseño con muy buenos resultados. Pero, en general, creemos que se ha conseguido el objetivo.

La pega de este modelo es que algunos componentes no son suficientemente específicos y heredan algunas funcionalidades que no necesitan (p.ej: los Triggers) y que pueden confundir al equipo de diseño. Estaría bien hacer otra iteración para acabar de pulir estos aspectos.

4. Lógica de juego

En este documento se realizará una explicación de cada clase y función creada.

Nota: En este documento se utilizarán dos colores principalmente. el **negro** y el **verde**. El color verde se usará para mostrar que es lo nuevo que se ha implementado frente a lo que ya estaba (negro).

4.1 Clases relacionadas con el jugador

Player.cs: Esta clase es la encargada de gestionar el comportamiento y el estado del jugador, es decir, de nuestro protagonista.

Funciones: Descripción.

- **Void Start () y void Awake ():** Funciones inicializadoras de estados (vida, energía, animaciones, ancho y alto del jugador, etc).

- **Void Update:** Actualiza cada ciclo el estado del personaje.

- **Void InputControl ():** Asocia a cada pulsación relacionada con el movimiento del personaje un evento que después será procesado y producirá unos efectos determinados. Se han añadido las mejoras que hacen posible el uso de la capa, armadura y el tornado.

- **Void EnergyUses ():** Relaciona los eventos de combate o usos de habilidades con su respectivo gasto energético.

- **Void Movement():** Relaciona los eventos de movimiento con un desplazamiento instantáneo.

- **Void EnergyRecoveryUpdate():** Se asegura de que se cumplen los límites aplicados a la barra de energía. La energía no puede ser inferior a cero o mayor a la máxima. Establece en caso de necesidad una recuperación constante de energía.

- **Void UpdateEnergyUI():** Actualiza la barra de estado de la energía del héroe en la UI.

- **Void UpdateLifeUI():** Igual que la función anterior pero relacionada con la barra de vida.

- **Void UpdateEffects():** Gestiona los Debufs y los Bufs del personaje. Efectos añadidos a la lista.
- **Void Notify():** Gestiona las acciones del jugador y las transforma en animaciones o notifica a NodeMap de que el jugador a realizado un ataque.

- **void OnEnemyCollision(Enemy enemy) :** Se activa cuando se colisiona con un enemigo. En este caso solo se gestiona si el enemigo produce daño por colisión.

- **void TakeDamage(Attack attack):** Gestiona el ataque del enemigo cuando el personaje se ve afectado por uno.Se ha modificado un poco internamente para impedir que entre ningún tipo de daño al jugador cuando éste utiliza la armadura.

- **void AnimationUpdate():** Según las acciones del jugador, ejecuta una u otras animaciones.

- **Void OldPositionReturn():** Devuelve al jugador la posición que tenía en el ciclo anterior.

- **Float GetWidth():** Devuelve el ancho del personaje.

- **Float GetHeight():** Devuelve el alto del personaje.

- **Actions GetActions():** Devuelve las acciones del personaje.

- **void SkillManagement():** Organiza si podemos usar una habilidad o no según nuestra energía actual. Si tenemos energía suficiente se utiliza la habilidad. En caso contrario no podremos usar habilidades y en el caso de que estuvieran activas (como la capa) se desactivarán.

- **Vector2 GetMidPosition() & Vector2 GetPosition():** Las dos funciones devuelven la posición del player pero de manera distinta. GetMidPosition() devuelve el punto medio del sprite del personaje mientras que GetPosition() devuelve el transform.position del personaje que está ubicado en los pies de éste. Esto se ha hecho así porque existía la necesidad de utilizar una posición u otra según las circunstancias.

- **Attack GetSowrdAttack():** Devuelve el ataque asociado a la espada.

- **Attack GetHelmetPower():** Devuelve el ataque asociado al Casco.

- **Attack GetCapPower():** Devuelve el ataque asociado a la capa.

- **void TakeSword():** Permite utilizar los poderes asociados a la espada (ataque básico).

- **void TakeHelmet():** Permite utilizar los poderes asociados al casco (ceguera).

- **void TakeCap():** Permite utilizar los poderes asociados a la capa (invisibilidad).
- **void TakeArmor():** Permite utilizar los poderes asociados a la armadura (invencibilidad).
- **void TakeTwister():** Permite utilizar los poderes asociados al control del viento (lanzar tornados).
- **bool GetCapIsActivated():** Devuelve true si la capa está activada.

NodeMap.cs: Gestiona todos los elementos del juego y proporciona las estructuras y funciones necesarias para el pathfinding.

- **Void Awake():** Inicializa las variables principales y crea el mapa que posteriormente se usará para hacer el pathfinding.
- **Void Start():** inicializa otras variables que deben de crearse o inicializarse en esta función.
- **Void Update():** Gestión de colisiones con enemigos, obstáculos o proyectiles.
- **Bool InObstacle(Vector2 point, GameObject obj):** devuelve true si el punto “point” esta dentro del objeto “obj”.
- **Void OnDrawGizmos():** Pinta la malla de puntos que permite el pathfinding, dibujando esferas rojas en los puntos por donde no se pueda pasar y azules en las zonas “walkables”.
- **List<Vector2> CreatePath(Vector2 or, Vector2 dest):** Crea una lista de puntos con el camino a recorrer para llegar de un punto “or” al punto “dest” utilizando el algoritmo a*. Si se ha encontrado el camino devuelve la lista con los puntos que conforman el camino válido. En caso contrario devuelve null.
- **Node [] SearchNeighbours (Node n, bool [,] map, int rows, int columns):** Busca los nodos que están alrededor del nodo “n”, verificando que sean válidos. Si se encuentran nodos válidos alrededor del objeto se envían en forma de Array de nodos.
- **Node SearchNodeInGridByPositionCoords(Vector2 vec):** Traduce un Vector2 “vec” en el nodo que se corresponde con la posición dada dentro de

la malla de puntos. Se mejoró la búsqueda del punto dadas unas coordenadas así como inicialmente coger una cuadrícula de aproximación y descartar todos los puntos que fueran “no walkables” para evitar algunos problemas en la búsqueda de caminos.

- **int Displacement(Node a, Node b):** Calcula el desplazamiento de un nodo “a” a un nodo “b”.

- **int CalculateH(Node a, Node b):** Es una función heurística que calcula la distancia de un punto “a” a un punto “b” de manera aproximada.

- **bool IsInside(List<Node> list, Node n):** Devuelve true si el nodo “n” está dentro de la lista “list”.

- **Node GetSmallestElement(List<Node> lista):** devuelve el nodo con menor coste (F) de la lista list.

- **int SearchNode(List<Node> list, Node n):** Devuelve la posición que ocupa el nodo N dentro de la lista “list”. En caso de no encontrarse el elemento devuelve -1.

- **bool IsObstacle(Node n):** Devuelve true en caso de que el nodo “n” sea “walkable” y false en caso contrario.

- **void CollisionCheck():** Gestiona las colisiones entre todos los elementos que conforman el mapa. Mejoras relacionadas con la colisión a otros elementos como trampas de diferentes tipos o proyectiles.

- **public void PlayerAttackCheck(Player.Actions action, int damage):** Dependiendo de la acción “action” del player se comprobará a que enemigos afecta este ataque y les quitará “damage” puntos de vida.

- **float CalculateAngle(float x_1, float y_1, float x_2, float y_2, Vector2 direction):** Calcula y devuelve al ángulo entre el vector formado por dos puntos y el vector “direction”.

- **List<Vector2> ValidatePoints(Node[] array):** Devuelve una lista con los nodos válidos del array de nodos “array”.

- **void EnemyTargeted():** Asigna a los enemigos el target(que es el player) cuando éste está lo suficientemente cerca. Ahora sólo permite dar a los enemigos el target siempre que el player no esté utilizando la capa de invisibilidad.

- **Projectile CreateProjectile(Projectile p):** Crea un proyectil de tipo ProjectileType. Inclusión de más tipos de proyectiles (Shadow ball, twister, musical note 1, musical note 2, musical note 3, fireball). Ha cambiado el tipo que devuelve.

- **bool[,] GetWalkableNodes():** Devuelve la lista de booleanos que indican si se puede pasar por parte de la malla de pathfinding o no.

- **Enemy InstantiateEnemy(string name, Vector2 position):** Instancia enemigos desde NodeMap dado el nombre del monstruo name y lo posiciona en position.

- **void CreateTrap(Vector2 position, string name):** Permite crear trampas dado el nombre y la posición.

- **void DestroyColliders(List<int> IDs):** Destruye una serie de Colliders relacionados con Eventos y taggeados inicialmente.

- **void SetNoTargetToEnemy():** Los enemigos pierden el target. Es decir, dejan de ver al jugador.

Node.cs: Clase que almacena la posición dentro de la malla de pathfinding, y las variables F, G, H necesarias para la aplicación del algoritmo a*.

Enemy.cs: Establece la base del comportamiento de todos los enemigos del juego.

- **Enemy(Transform tr):** Constructor de la clase, inicializa variables.

- **void SetTarget(GameObject target):** Asigna el target “target” al enemigo.

- **void Behaviour():** Se establecen funciones que satisfacen cada estado alcanzado por el enemigo.

- **virtual void IdleActions():** Función con la lógica necesaria para satisfacer el estado Idle.

- **virtual void ChaseActions():** Función con la lógica necesaria para satisfacer el estado Chase.

- **virtual void CombatActions():** Función con la lógica necesaria para satisfacer el estado Combat.

- **virtual void DeadActions():** Función con la lógica necesaria para satisfacer el estado Dead.

- **virtual void OnCollision():** Respuesta a una colisión.
- **virtual void TakeDamage(int damage):** Gestiona el daño recibido.
- **void ChangeState(States state):** Cambia de estado al enemigo según el parámetro “state”.
- **void AnimationControl():** Ejecuta las animaciones bajo la demanda de los subestados del enemigo.
- **List<Vector2> CreatePath(Vector2 or, Vector2 dest):** Se sirve de la función CreatePath de NodeMap para trazar una ruta de un punto “or” a uno “dest”.
- **void CreatePatrolPath(GameObject go_path):** Crea una lista con los puntos necesarios para crear una patrulla dado un gameobject con hijos que establezcan los puntos principales de la ruta.
- **void FollowPath(List<Vector2> fpath, ref int index):** Hace al enemigo seguir la ruta fpath.
- **void FollowPlayer():** Igual que FollowPath pero utilizado para crear bajo demanda caminos al player para perseguirlo.
- **Vector2 CreateDirection():** Devuelve un vector con una dirección distinta a la actual que utilizará para desplazarse en algunos casos o estados.
- **void ChangeDirection():** Invierte la dirección de movimiento actual.
- **States GetState():** Devuelve el estado actual del enemigo.
- **int GetHealth():** Devuelve la vida actual del enemigo.
- **int GetMaxHealth():** Devuelve la vida máxima del enemigo.
- **float GetAttackSpeed():** Devuelve la velocidad de ataque del enemigo.
- **float GetWidth():** Devuelve el ancho del enemigo.
- **float GetHeight():** Devuelve el alto del enemigo.
- **float GetVisionRange():** Devuelve el rango de visión del enemigo.
- **bool GetHurtTouch():** Devuelve true en caso de que el enemigo pueda producir daño por contacto/colision y false en caso contrario.

- **bool IsReadyToFight():** devuelve true cuando el enemigo puede volver a atacar y false en caso contrario.

- **bool GetHasPatrol():** Devuelve la variable “hasPatrol”.

- **Vector2 GetPosition() y Vector2 GetPivotPosition():** Igual que para el player. Según circunstancias se coge uno u otro.

- **Atributes GetAtributes():** Devuelve la estructura con los atributos.

- **void SetAtributes(Atributes atributes):** Modifica los atributos actuales por otros dados por parámetro.

Nota: La clase enemigo está pensada para ser la base de la que hereden todos los demás enemigos. Al haber enemigos inteligentes que utilizan patrol y otros que no, añadí una variable para diferenciarlos y poder darles la funcionalidad de seguir una patrulla solo a aquellos enemigos que sean inteligentes. Ha habido muchas mejoras con respecto a la externalización de las variables de todos los enemigos para facilitar el balanceo de los mismos.

- **Vector2 GetPosition():** Devuelve el punto medio del enemigo.

- **Vector2 GetPivotPosition():** Devuelve el punto donde se situa el pivot del enemigo.

- **Attack GetAttack():** Devuelve el ataque del enemigo.

- **GameObject GetTarget():** Devuelve el target del enemigo.

MeleeSkeleton.cs: Clase derivada de Enemy. Gestiona las peculiaridades de este tipo de enemigo. Enemigo cuerpo a cuerpo.

SwordOrc.cs: Clase derivada de Enemy. Gestiona las peculiaridades de este tipo de enemigo. Enemigo cuerpo a cuerpo.

StabbingPirate: Clase derivada de Enemy. Gestiona las peculiaridades de este tipo de enemigo. Enemigo cuerpo a cuerpo.

ArcherSkeleton.cs: Clase derivada de Enemy. Gestiona las peculiaridades de este tipo de enemigo. Enemigo a distancia.

- **Void Runaway():** Crea una ruta de huida en caso de que el player se acerque mucho.

- **void FollowPathForRunAway(List<Vector2> fpath, ref int index):** Sigue la ruta de huida de manera similar a como lo hace la función base de Enemy Void FollowPath(List<Vector2> fpath, ref int index).

ArcherPirate.cs: Clase derivada de Enemy. Gestiona las peculiaridades de este tipo de enemigo. Enemigo a distancia.

Spider.cs: Clase derivada de Enemy. Gestiona las peculiaridades de este tipo de enemigo.
Rat.cs: Clase derivada de Enemy. gestiona las peculiaridades de este tipo de enemigo.

Attack.cs: Clase que se encarga de gestionar el ataque de cada enemigo. Recoge aspectos como el tipo de ataque (físico, venenoso, etc), el daño que causa y si es simple o un área. [En fase de desarrollo: faltan implementar más tipos de ataque].

- **public AttackType GetAttackType():** Devuelve el tipo de ataque.
- **public SubAttackType GetSubAttackType():** Devuelve el subtipo de ataque.
- **public int GetDamage():** Devuelve el daño que hace el ataque.

Effects.cs: Clase que contiene un efecto negativo o beneficioso para el jugador o enemigos. Tiene efecto en el tiempo.

- **Bool IsEnd():** Devuelve true en caso de que el efecto ya haya expirado y false en caso contrario.
- **Virtual void Update(int health)**: actualiza el efecto en el objetivo. [En proceso de desarrollo].
- **Attack GetAttack()**: Devuelve el ataque del que procede el efecto.

PoisonEffect.cs: Clase derivada de Effects. Aplica daño en el tiempo por envenenamiento.

StunEffect.cs: Clase derivada de Effects. Inmoviliza a los enemigos durante un tiempo.

BlindEffect.cs: Clase derivada de Effects. Impide ver al personaje (se aplica solo a enemigos).

Strucst.cs: Archivo donde encontramos las estructuras que usaremos en el juego.

WitchBehaviour.cs: Clase encargada del comportamiento del primer jefe dentro del nivel 2: El Bosque. La lógica del enemigo es muy sencilla. Invocará enemigos al principio del turno y se volverá inmune a nuestros ataques. Cuando matemos los

enemigos invocados, se quedará unos segundos aturdida por el esfuerzo que supuso la invocación. En este momento la bruja es vulnerable. Una vez se recupere del aturdimiento vuelve a utilizar una barrera que la vuelve inmune al ataque e invoca más esbirros que antes. Cada invocación aumenta el número de enemigos en pantalla.

- **Void CheckEnemies():** Verifica si algún enemigo invocado ha muerto.
- **void GetHurt():** Estudia si ha sido alcanzado por un ataque por parte del jugador.
- **void AjustHealthBar():** Ajusta la barra de vida según se necesite.
- **Enemy InvokeEnemy(string enemyName, Vector2 position):** Instancia un enemigo en la lista local de enemigos y en la de la clase NodeMap.

SidheBehaviour.cs: Clase encargada del comportamiento del segundo jefe del juego. Este enemigo tiene varias fases. Cuando estemos a distancia nos atacará lanzándonos bolas de sombra que podremos evitar utilizando la habilidad del casco.Cuando estemos cerca se transformará en una sombra y cargará contra nosotros constantemente. En esta segunda fase deberemos utilizar el casco para sacarla de la forma de sombras y poder golpearla. Cuando pierde cierta salud, la reina sidhe se enfurece y carga contra nosotros lanzando bolas de las sombras al final de cada carga. Nuevamente con el casco podremos sacarla de su forma sombría y causarle daño.

- **void Start():** Inicializa al boss.
- **void Update():** Actualiza su comportamiento.
- **void UpdateHealth():** Actualiza la barra exterior de vida.
- **void PlayerAttackCheck():** Comprueba los estados del personaje y verifica si ha sido alcanzada por algún ataque.
- **PlayerHelmetCheck():** Utilizado para informar a la clase si el casco ha sido utilizado y cambiar de estado al boss.
- **void GetHurt():** Gestiona el daño del boss (se le hace daño fijo).
- **void Charge():** Función encargada de realizar las cargas contra nosotros en forma de sombra.
- **void ChargeWithShadowBall():** Igual que la anterior función pero instanciando al final una bola de las sombras.

- **void SelectPhase():** Según la vida de nuestro boss, cambiará a un estado u otro.

BossPirate.cs: Clase encargada del comportamiento del tercer Boss. El boss pirata tiene dos modos. Melee y a distancia. Cada vez que le quitemos un cierto porcentaje de vida se teletransportará encima de unos barriles y los aporreará generando notas musicales. Estas notas musicales son lanzadas hacia el personaje y al azar en caso de que se use la capa de invisibilidad. El estado Melee es un estado en el cual nos persigue y golpea. Para ayudarnos a superar esta fase nos serviremos del casco para aturdirlo brevemente y poder golpearlo.
Las funciones de este boss no voy a comentarlas pues son muy similares a las de los bosses anteriores.

Ascarth: Es el boss final y tiene varios estados. el primer estado sería el de golpearnos cuerpo a cuerpo. Deberemos usar la armadura constantemente o nos destrozará con sus golpes. Otra posibilidad es utilizar la capa si nos vemos muy apurados ya que al no vernos adelantará su siguiente fase. Después de cierto tiempo, el boss se teletransporta a una plataforma a la cual no podemos llegar y nos lanzará bolas de fuego. Éstas al chocar contra el suelo crearán una zona en llamas que nos hará daño al contacto. El boss siempre nos lanzará bolas de fuego a nosotros a menos que utilicemos la capa de invisibilidad. En ese caso lanzará de manera aleatoria las bolas de fuego por toda la plataforma. Un rato después según la vida que le falte seguirá pegándonos cuerpo a cuerpo o iniciará su tercera fase. La tercera fase no es más que la incorporación de una nueva habilidad, la explosión. La explosión es una habilidad cargada, es decir, empieza a castearla y cuando termine de castearla tendremos que usar la armadura puesto que sino moriremos de inmediato.
La única función a mencionar interesante es la de **UpdateSkillBar(float percentage)** que actualiza la barra de carga para la habilidad final.

EnemyBehaviour.cs: Clase donde se almacena la clase Enemy. Lo más interesante de esta clase es que podemos utilizar la funcionalidad de Monobehaviour y hacer uso de la función Update para actualizar a nuestro enemigo.

- **void Init(int maxHealth, float speed):** Crea un enemigo.
- **void AjustHealthBar():** Ajusta la barra de vida.
- **void DestroyEnemy():** Destruye al enemigo después de un tiempo.
- **Enemy CreateEnemyByName(string name):** Crea un enemigo dado el name.
- **void SetAtributesInit():** Completa la tabla de atributos del inspector. Si en esta aparece un cero, se coge un valor por defecto y se muestra en el editor.

Si por el contrario en el editor se encuentra un valor distinto de cero, este se mantiene y pasa a formar parte de la lista de atributos.

- **void SetAtributes():** Asigna al enemigo encapsulado en la clase EnemyBehaviour los atributos previamente seleccionados.

- **void OnValidate():** Cada vez que desde fuera (editor) se cambia un atributo, este se actualiza.

- **void Init():** Función que inicializa el objeto.

MonsterFabric.cs: Clase utilizada para instanciar enemigos en las clases Monobehaviour de manera sencilla. [Incompleto hasta tener todos los enemigos programados de todos los niveles].

Obstacle.cs: Clase que almacena la posición, el ancho y alto de un obstáculo.

Projectile.cs: Clase de la que derivarán los demás proyectiles.

- **virtual Vector3 Update():** Actualiza el proyectil.

- **Vector2 GetPosition():** Devuelve la posición del proyectil.

- **float GetRotation():** Devuelve la rotación del proyectil.

- **Attack GetAttack():** Devuelve el ataque del proyectil.

- **ProjectileType GetProjectileType():** Devuelve el tipo del proyectil.

- **bool IsDead():** Devuelve true en caso de que el tiempo de vida del proyectil haya pasado.

- **void KillProjectile():** Elimina el proyectil.

- **void Fix():** Deja fijo el proyectil en un punto. Usado para proyectiles estilo flecha al chocar contra muros, árboles y demás. [En proceso de desarrollo].

Arrow.cs: Clase derivada de Projectile.

ProjectileShadowBall.cs: Clase derivada de Projectile. Este proyectil sigue al jugador allá donde vaya y puede ser destruida con el poder del casco.

FireBall.cs: Clase derivada de Projectile. Este proyectil se dirige de un punto A a un punto B. Si impacta en el camino, hará daño y creará una zona llameante en el suelo. Si llega al punto B sin colisiones se crea en la zona un charco de fuego.

Twister.cs: Clase derivada de projectile. Este proyectil se crea gracias a los poderes que tiene el personaje. Tiene la capacidad de eliminar los charcos de fuego del suelo.

ProjectileBehaviour.cs: La misma manera de proceder que EnemyBehaviour con respecto a Enemy.

ProjectileFabric.cs: La misma manera de proceder que en MonsterFabric.

BramblesBehaviour.cs: Clase de la primera trampa implementada. Las zarzas bloquean el camino al personaje y a los monstruos, pero pueden romperse con la espada despejando el camino haciéndolo "walkable" para el player y los enemigos.

- **void CheckCollision():** Calcula si el player ha colisionado con las zarzas y en caso positivo le hace daño.
- **void DestroyBramble():** Destruye la zarza y despeja el camino.
- **void GetHurt():** Comprueba si el jugador ha atacado y verifica si le ha dado. En caso afirmativo procedemos a eliminar la zarza.
- **float CalculateAngle(float x_1, float y_1, float x_2, float y_2, Vector2 direction):** Calcula el Angulo entre el vector formado a raíz de los floats y el vector "direction".
- **void LockArea():** Al crearse modifica el "walkableMap" bloqueando el paso a enemigos y al player.
- **Void UnlockArea():** Desbloquea los puntos que LockArea() bloqueó inicialmente.

SpikeTrapBehaviour.cs: Esta trampa se activa por proximidad. Cuando el personaje se acerca a la trampa, los pinchos salen hacia afuera haciendo mucho daño.

- **void CheckCollision():** Comprueba si el personaje está dentro del área y en caso afirmativo le inflige daño.

WildFire.cs: Esta trampa consiste en un fuego fatuo que recorre dos puntos constantemente. Al acercarnos a el nos provoca daño de fuego cada 0.1 segundos. [El daño por fuego está pensado que se implemente en la siguiente entrega por lo que de momento solo hace daño físico].

- **void CheckCollision():** Igual que las trampas anteriores.

ElectricTrap.cs: Clase derivada de Trap. Hace daño cada cierto tiempo y se desplaza entre dos puntos.

FireTrap.cs: Clase derivada de Trap. Hace daño cada cierto tiempo y puede ser eliminada utilizando un tornado.

Armor.cs: Clase que permite la recogida y eliminación del objeto Armor. Una vez cogido este objeto se desbloquean las habilidades relacionadas con la armadura.

Cap.cs: Clase que permite la recogida y eliminación del objeto Cap. Una vez cogido este objeto se desbloquean las habilidades relacionadas con la Capa.

Helmet.cs: Clase que permite la recogida y eliminación del objeto Helmet. Una vez cogido este objeto se desbloquean las habilidades relacionadas con el casco.

Sword.cs: Clase que permite la recogida y eliminación del objeto Sword. Una vez cogido este objeto se desbloquean las habilidades relacionadas con la espada.

EventSystemLevel.cs: Clase encargada de gestionar algunos eventos según la escena.
Inicialmente recoge el nombre de la escena en la que se está trabajando para saber con qué eventos se va a interactuar.

- **void Update():** Actualiza el sistema de eventos.

- **void UpdateFairyScreen_01():** Actualiza el sistema de eventos de la escena 01 del mundo de las hadas. **No implementado. No necesaria la implementación.**

- **void UpdateFairyScreen_02():** Actualiza el sistema de eventos de la escena 02 del mundo de las hadas. **No implementado. No necesaria la implementación.**

- **void UpdateFairyScreen_03():** Actualiza el sistema de eventos de la escena 03 del mundo de las hadas. **Implementado.**

- **void UpdateFairyScreen_04():** Actualiza el sistema de eventos de la escena 04 del mundo de las hadas. **Implementado.**

- **void UpdateFairyScreen_05():** Actualiza el sistema de eventos de la escena 05 del mundo de las hadas. **No implementado. No necesaria la implementación.**

- **void SetDestroyableCollider(int ID):** Selecciona de cada nivel los obstáculos que se van a destruir por un determinado evento y le pasa la información a NodeMap.
- **void ActivateKey():** activa la llave de Fairy3.

- **void SetTombA(bool b):** Activa la tumba A en Fairy4.

- **void SetTombB(bool b):** Activa la tumba B en Fairy4.
- **void SetTombC(bool b):** Activa la tumba C en Fairy4.

4.2 Dificultades

A lo largo del desarrollo me he encontrado con muchas dificultades a la hora de programar la parte que me tocaba. La parte que me ha tocado, como queda reflejado en el Trello son los enemigos, bosses, trampas y player y, por ende, también el tema de colisiones y demás.
Al ser un juego 2D no podíamos usar el navMesh de Unity así que decidí hacer mi propio pathfinding para que los enemigos pudieran desplazarse por el mapa sin que todo fuera un caos. Esto acarreaba muchos problemas ya que para realizar el pathfinding con éxito necesitaba una malla de nodos con los que poder interactuar. Pensé entonces que al crear la clase NodeMap, debería de construir un array de nodos reflejando los puntos por donde un enemigo puede o no puede pasar. Una vez conseguí hacer la malla de puntos sin errores empecé a implementar el pathfinding, que de todos los que hay, me quedé con el algoritmo a* por "simplicidad y velocidad". Luego me di cuenta de que no era tan sencillo realizar este algoritmo porque hay muy poca información en internet sobre este método y además según la disposición del mapa puede llegar a tardar mucho en calcular la ruta de un punto A, a un punto B. Tenía muchos problemas de eficiencia que conseguí arreglar como por ejemplo que para situar el punto de la malla más cercano que corresponde a un personaje o enemigo, lo que hacía era recorrer toda la malla calculando las distancias del enemigo o del player a cada punto de la malla y quedándome con aquel nodo que estuviera más cerca del objetivo. Ahora simplemente haciendo un sencillo cálculo matemático calculo aproximadamente dónde se encuentra el player o el enemigo dentro de la malla.
Esto nuevamente traía un problema, cuando el enemigo o el player se encontraban muy cerca de un obstáculo, el juego se ralentizaba o los enemigos se quedaban parados en exceso porque no existía ruta posible entre un punto y otro que fuera un punto "no walkable" de la malla. La forma de solucionarlo era comprobando las posiciones adyacentes del enemigo o player y cogiendo la más favorable o la primera disponible.
He tenido problemas también para programar los enemigos pues al estar constantemente interactuando con el entorno y con el player, daba a muchos errores raros, como que los enemigos saliesen "disparados" hacia fuera del mapa a grandes velocidades, se quedaran atascados en las esquinas de los obstáculos, mataran al player de un golpe, etc...
Ha sido mucho tiempo de testeo y pruebas y en esta versión tengo mucho código mal organizado que pienso ordenar y estructurar cuando hayamos avanzado más en el proyecto puesto que aún queda mucho por hacer.

Las colisiones también están hechas “from scratch”. En un juego del estilo que estamos haciendo no necesitábamos físicas de ningún tipo, y vi por internet que colliders sin rigidbody traía problemas de rendimiento así que también realicé un sistema sencillo de colisiones.
Los puntos destacados de esta entrega que he programado serían: Enemigos (Melee Skeleton, Archer Skeleton, Spider), Bosses (Witch), Effectos (Envenenado), Movilidad (pathfinding, colisiones, funciones se seguimiento automático de rutas), Player(Movimiento, Combate, Colisiones), Trampas (Brambles, Spike trap, wildfire).

Las dificultades durante esta última parte han sido mayores y es que a medida que crece el proyecto todo se complica mucho más. Se han implementado nuevas trampas, proyectiles, enemigos, con sus lógicas y efectos. He aprendido muchísimas cosas durante el proceso de creación del videojuego y esas mismas cosas que aprendía las he estado aplicando a cosas anteriores ya implementadas. Como por ejemplo el utilizar el editor para actualizar los enemigos y usar la función OnValidate para no tener que estar constantemente mirando si alguna variable ha cambiado o no. La clase NodeMap se ha vuelto un caos difícil de entender para quien no ha tocado esa clase. Esto último es normal ya que empezó siendo una clase de organización y en un principio solo se organizaban unos cuantos enemigos y un sistema de pathfinding. A medida que ha ido creciendo se han añadido más elementos tanto de los mismos tipos como diferentes. Muchos de estos elementos eran de uso exclusivo como las brambles, el twister, la fire trap y otras.

Siguen existiendo muchos bugs que están identificados como por ejemplo una excesiva ralentización por parte de los arqueros cuando tocan zonas “no walkables”. Este problema ocurre desde la tercera vez que modifiqué el pathfinding, ya que fue en ese punto donde toqué la manera que tenían los enemigos de seleccionar los puntos por los que circulaban para llegar de A a B. Es solucionable con tiempo al igual que el resto de bugs que arrastramos.
Nos ha faltado tiempo y eso está claro, pero estoy muy contento con lo que hemos hecho porque ha servido para crecer y aprender. Muchas personas Top del mundo de los videojuegos y del campo del desarrollo de software dicen que siempre has de plantearte un reto mayor del que puedas realizar porque ahí es donde aprendes de verdad ya que sales del “área de confort” y te obliga a estudiar y practicar lo estudiado. Aunque nuestro objetivo no fuese el de tener un proyecto inabarcable en estos meses, al final ha sido un poco eso, hemos lidiado con un proyecto grande y aunque el resultado final no es lo que hubiéramos esperado, ya que falta mucho trabajo por hacer, la experiencia ha sido muy buena.

5. Conclusiones

Habiendo llegado al final del tiempo establecido para poder entregar el trabajo tenemos una sensación muy clara, nos ha faltado tiempo. Durante el desarrollo del proyecto hemos tenido 2 reveses importantes, el primero el abandono por parte del miembro número 4, y por este hecho en sí, si no más bien por no haber sabido en su momento eliminar parte de lo proyectado para ir más desahogados. Y el segundo fue el hecho de perder a otro miembro del equipo durante un mes, si ya estábamos con el agua al cuello esto ya acabó de hundirnos.

El trabajo entregado es una sombra de lo que debería haber sido, podemos tratarlo como una beta, ya que tenemos numerosas cosas que hay que mejorar. No hemos podido probar lo que hay a fondo, así que cuando hacíamos pruebas de alguna funcionalidad solían aparecer problemas relacionados con otras áreas y en ese mismo momento se intentaban de resolver, pero aun así han quedado algunos en el tintero.

Otro problema importante al que nos hemos enfrentado ha sido el individualismo, si bien todos teníamos claro que éramos un equipo, a la hora de hacer las tareas que estaban marcadas como más importantes en Trello, no todos lo entendimos de la misma forma y el proceder de cada uno de los miembros no fue más que en contadas ocasiones el más optimo para las entregas, hacia falta quien tomase las riendas y organizara a los demás ya que solos no pudimos. Una cosa que pensamos que resultaría es que, alguno de los miembros adquiriese el papel del Producer en videojuegos, y se encargara de organizar el trabajo y gestionar el tiempo para las entregas.

El producto final es una beta como hemos dicho antes, pero con más trabajo puede ser un gran juego, la idea de base es buena y nos hemos acercado bastante a lo que queríamos en un principio, ahora ya solo sería cuestión de pulir el producto, acabar de implementar el sonido y balancearlo a base de pruebas. Con el tiempo necesario para estas tareas pensamos que el juego mejoraría exponencialmente.
Pese a que el trabajo no ha dado el resultado todo lo satisfactorio que nos hubiese gustado, lo hemos dado todos estos meses, este juego se lleva parte de nuestras almas y solo nosotros sabemos la cantidad de horas que le hemos dado, así que por esto aún tenemos motivos para sentirnos orgullosos del trabajo entregado.

6. Bibliografía

- [1] The legend of Zelda a Link to the pass, 1991, Nintendo.

- [2] La noche de las Walpurgis, festividad del 30 de mayo al 1 de abril, https://es.wikipedia.org/wiki/Noche_de_Walpurgis visitada el 30/06/2018

- [3] GDD, siglas de Game Design Document, https://en.wikipedia.org/wiki/Game_design_document visitada el 30/06/2018

- [4] Noche de las Walpurgis, película de 1970, la foto que ilustra el comentario también está sacada de la siguiente web https://www.filmaffinity.com/es/film442746.html visitada el 30/06/2018

- [5] La noche de las Walpurgis, canción de 2013 de Los carniceros del norte, versión web https://youtu.be/-Wej_xZpPDA visitada el 30/06/2018

- [6] GitHub, servidor de git web https://github.com/ visitada el 30/06/2018

- [7] Bitbucket, servidor de git web https://bitbucket.org visitada el 30/06/2018

- [8] Trello, software de administración de proyectos en web https://trello.com/ visitada el 30/06/2018

- [9] Autocad, software de diseño para dibujo y 2D y modelado 3D, Autodesk 1982.

- [10] Photoshop, editor de gráficos rasterizados, Adobe Systems Incorporated 1988.

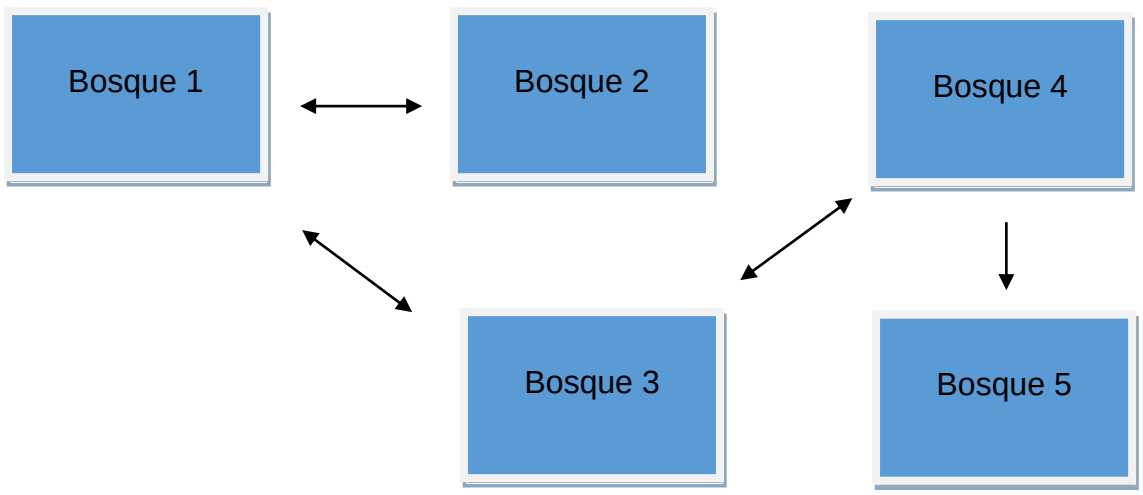
- [11] Unity 3D, motor de videojuegos multiplataforma, Unity Technologies 2005.

7. Anexos

A continuación añadimos algunos documentos que se han generado durante el desarrollo del proyecto, los cuales creemos que son necesarios para poder comprender todo el trasfondo del mismo.

7.1 Documento de transiciones del bosque

Correspondencias del nivel:



Bosque 1

Al poco de empezar nos encontramos con el druida, trigger de conversación, al acabar volvemos a tener el control, si nos vamos de pantalla y volvemos el druida ya no está.
Al norte tenemos una salida a bosque3 y al sur una salida a bosque2.

Bosque 2

Salida a bosque1 por donde acabamos de entrar.
En la parte izquierda inferior del mapa hay un pergamino, trigger con frase y recoger. Se trata de un objeto que podemos coger con una frase cachonda.
En la parte derecha del mapa está el puzle de las columnas, primero trigger de la charla con el druida y luego 1 interruptor por columna, al acertarlo aparecerá la espada para recogerla, ahora mismo está conectada, hay que desconectarla y hacer los triggers a los interruptores.

Bosque 3

Salida a Forest2 por donde hemos entrado.
En el centro de la pantalla, en la hoguera tras las zarzas haremos checkpoint.
Todo a la derecha y abajo del mapa tendremos la entrada a Forest4.

Bosque 4

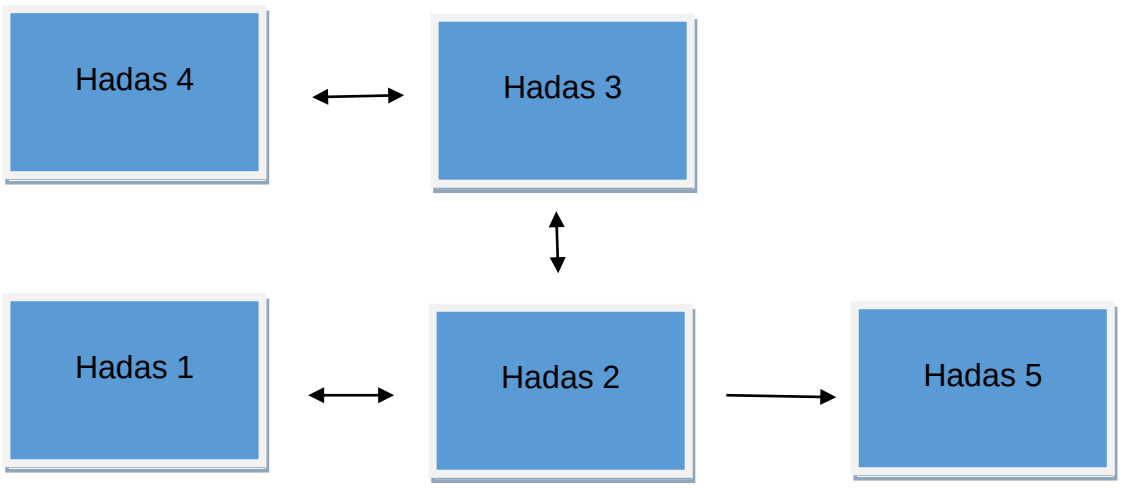
Salida a Forest3 por donde hemos entrado.
Parte derecha del mapa entrada a Forest5.

Bosque 5

Hacemos un checkpoint al inicio del nivel.
Triggers de conversación con la bruja y empieza la pelea.
(desconectarla al principio porque empiezas la pantalla con enemigos)
Tras derrotarla hay que instanciar una explosión o algo así, tras esta explosión nos teletransportaremos a hadas1.

7.2 Documento de transiciones de las hadas

Correspondencias del nivel



Hadas 1

Aquí el trigger deberá estar al final del camino de la derecha, es el que comunica con hadas 2, ha de estar antes de que se vea el final de la pantalla.

Hadas 2

Empezamos en el camino, un poco más atrás deberá estar el trigger para volver a hadas1, todo a la derecha hay una barrera y 2 antorchas con pedestal (estas tienen ya la animación de fuego, pero apagada), se tienen que poder encender con el casco y una vez encendidas se ha desconectar la barrera mítica para poder acceder al teletransportador a hadas5.
Al sur de la pantalla hay una tumba, ahí tendremos un trigger para el fantasma que nos da la misión (mirar el GDD)
Al norte podremos encontrar una cueva, es el trigger a hadas3.

Hadas 3

Aparecemos un poco más delante de la entrada que a su vez será el trigger para volver a hadas2, estamos también junto a un campamento que será un trigger de checkpoint.
En la habitación grande de la derecha hay una pelea con un enemigo importante que suelta una llave, la llave está ahí y también es prefab lo que aún no hace nada. Con esa llave
podremos desactivar la barrera mítica del camino de la izquierda, y justo en la puerta de más adelante tendremos el trigger a hadas4.
Hadas.

Hadas 4

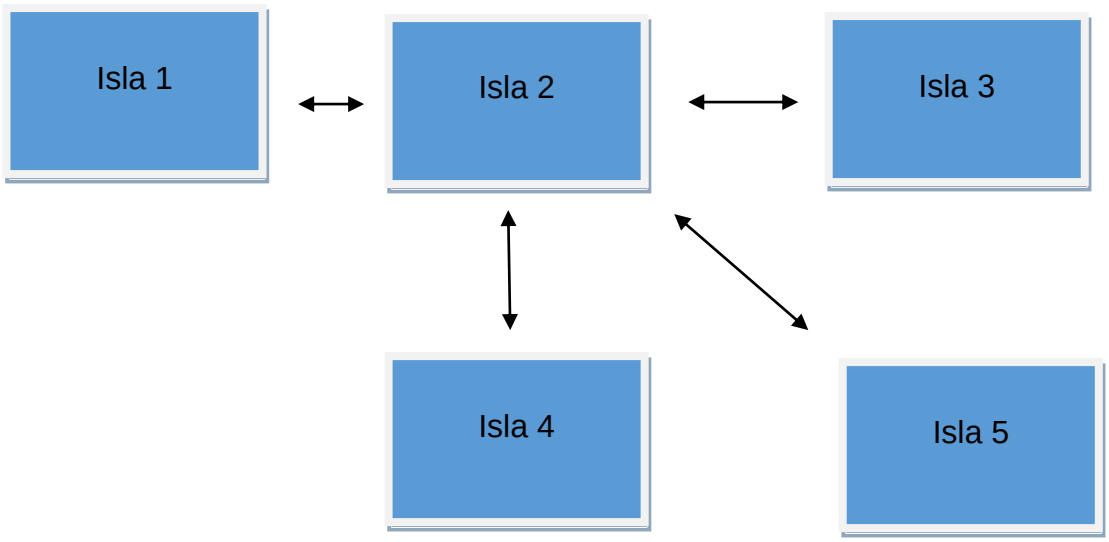
Hay tres tumbas en esta pantalla, cada una de ellas ha de tener un trigger para que al pulsar sobre ellas tengamos una frase de contexto y acto seguido una oleada de enemigos. Cuando lo hagamos en las 3 y venzamos se activará un puente que nos dará acceso a la isla central donde se encuentra el casco, como la llave hecho prefab pero sin funcionalidad.
El puente está ya colocado en su lugar, pero desconectado, habrá que conectarlo en el momento oportuno y además habrá que desconectar un hijo de MaopObstacles llamado puenteCollider para poder atravesar el lago.

Hadas 5

Aquí es la pelea con el jefe, una vez vencido ha de caer la barrera mística y a través del portal tendremos que poder avanzar hasta Isla1 que aún no existe.
También habría que ponerle un efecto al portal en algún momento para que molase más.
PD: Supongo que aquí hay partes de Rodrigo también.

7.3 Documento de transiciones de la isla

Correspondencias del nivel



Isla 1

En esta pantalla tenemos un solo trigger en la parte de abajo a la izquierda para pasar a piratas2.

Isla 2

Desde esta pantalla podemos acceder a las otras 4, piratas3 por la derecha, piratas4 por la izquierda, piratas1 al norte y pirats5 al sur.
La entrada a piratas 5 estará tapada con una barrera mística, para poder derribarla necesitaremos eliminar a los guardias con la trampa del barril de ron que hay cerca de la barrera. Si los matamos de otro modo la barrera no caerá y deberemos de salir de la pantalla y volver para poderlo volver a intentar. Una vez eliminada la barrera permanecerá así.

Isla 3

Podremos volver a piratas2 desde la parte superior izquierda de la pantalla, también será la entrada de la pantalla.
Al final de la pantalla hay un edificio con 2 antorchas, deberemos de encenderlas con el casco para derribar una barrera mística que hay en piratas4. Una vez encendidas las antorchas permanecerán así.

Isla 4

Entraremos a la pantalla por la parte de la derecha y justo por ahí podremos volver a piratas2.

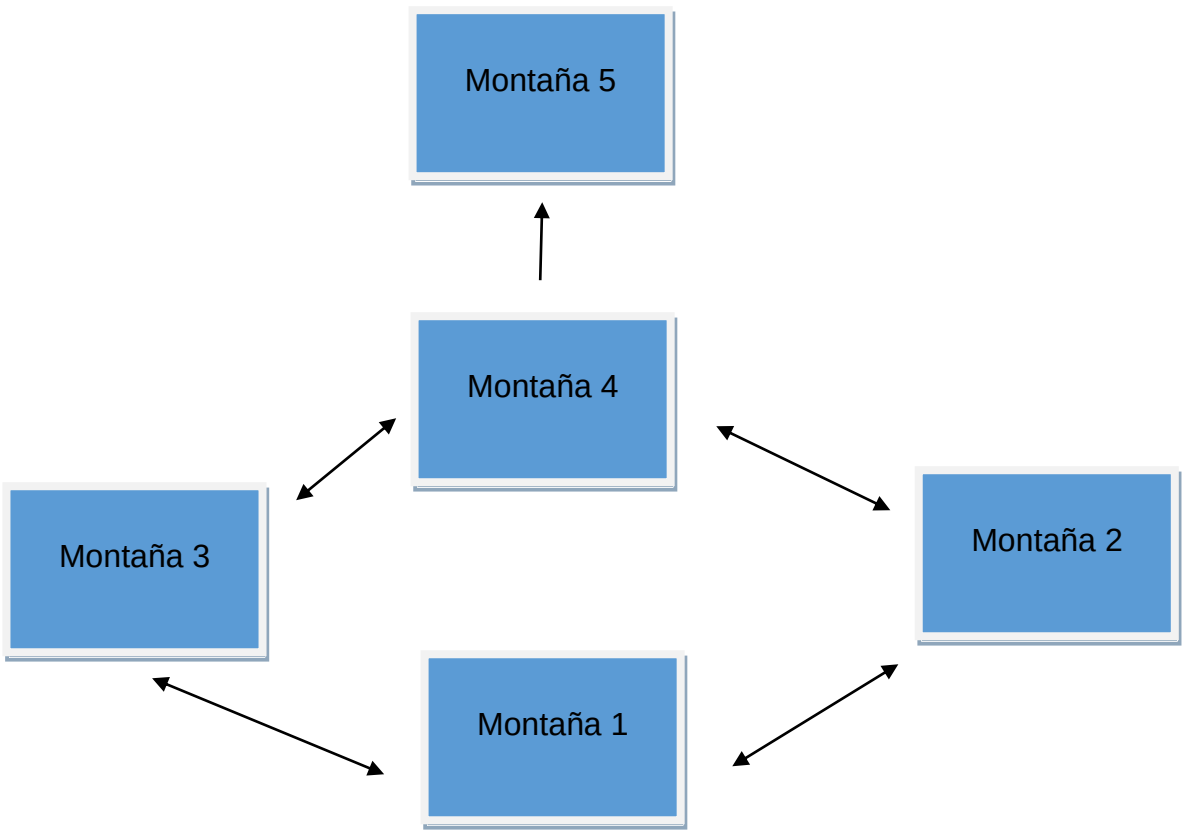
En la parte sur de la pantalla encontraremos una barrera mística que no nos dejará pasar, para derribarla deberemos de encender las antorchas del templo de la luz de piratas3. Una vez que logremos pasar tendremos un checkpoint en la hoguera de los monolitos y además podremos recoger la capa del ladrón y las hierbas venenosas que nos servirán para activar las trampas de barril de ron.

Isla 5

Entraremos a la pantalla por el norte y por ahí también podremos volver a piratas2.
En esta pantalla hay 4 trampas de ron, podremos eliminar a la gran cantidad de enemigos muy fácilmente con ellas y la capa, o podemos hacerlo a lo animal. En la parte sur de la pantalla se encuentra el muelle, justo al entrar se encenderá una barrera mística que nos impedirá salir y empezará la pelea con el jefe. Al derrotarlo pasaremos a la pantalla de la montaña.
Nota: En el muelle hay unos bloques de cajas y entre ellos barriles, es aquí donde el jefe hará percusión para lanzarnos proyectiles.

7.4 Documento de transiciones de la montaña

Correspondencias del nivel.



Montaña 1

Casi nada más empezar tenemos la armadura de los titanes, al acercarnos trigger de conversación y hacer que la armadura se pueda coger, ya está el prefab.
Este nivel tiene dos salidas, al norte a la derecha Mountain2 y al norte a la Izquierda Mountain3, triggers de salida.

Montaña 2

Tenemos un trigger para volver a montaña 1 por donde hemos venido, todo al norte hay otro trigger para pasa a montaña 4.
En la parte derecha del mapa vemos un pasillo que lleva a una plataforma donde podremos recoger el poder del tornado, el prefab ya está hecho faltará actuar sobre el.
En el pasillo hay que poner un efecto de fuego constante, y debe de hacernos daño, la única forma de pasar ha de ser con la armadura.
Una vez en la plataforma deberemos de activar los 4 pilares para conseguir el tornado.

Montaña 3

Tenemos un trigger para volver a montaña 1 por donde hemos venido y todo al norte tenemos otro para pasar a montaña 4.
A la izquierda del mapa tenemos el templo del viento, tendremos los triggers de conversación. Además, tenemos 2 antorchas fuera de la zona caminable, tendremos que apagarlas con los tornados para poder eliminar la barrera de montaña 4.

Montaña 4

Tenemos 2 triggers en la parte inferior de la pantalla, el derecho nos llevará a Mountain2 y el izquierdo a mountain3.
En la parte superior de la pantalla tenemos una marca en el suelo que hará de teletransportador hacia mountain5, para llegar a el deberemos apagar la barrera qua nos impide el paso.

Montaña 5

Los pasos del jefe se describen en el GDD.

8. Links

Direcciones web con el video de la defensa del proyecto y el repositorio donde se encuentra el trabajo.

Video:

- https://www.youtube.com/watch?v=oDHXFOeifdY&t=1063s

Repositorio:

- https://bitbucket.org/uoctfmteam/proyecto-y/src