Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de un videojuego 2D de plataformas en Unity

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Diego Ruiz Muñoz

Tutor: Javier Lluch Crespo

2021-2022

Resumen

En este documento seguimos el desarrollo de un videojuego 2D de tipo plataformas desde cero, esto abarca el pensamiento de la idea, la planificación de esta idea, la organización, el proceso de desarrollo y el resultado final. Para ello utilizamos Unity2D y el lenguaje C#. Para el seguimiento de este TFG se utiliza una metodología ágil mediante el uso de la herramienta Trello con el añadido de una extensión para registrar tiempos en las tareas y con ello llevar un mejor seguimiento.

También se muestra el uso de patrones de programación y el porqué de utilizarlo ya que cada uno tiene un propósito diferente a la hora de utilizarlo, uno de estos patrones de programación es el Singleton.

El mapa del videojuego es uno fijo que está diseñado desde cero tomando como referencia mapas de videojuegos famosos como “Hollow Knight” o “Ori and the Blind Forest”. La historia del videojuego esta adornada con pequeños toques cómicos, referencias a memes y referencias a otros videojuegos.

Palabras clave: videojuego, Unity2D, plataformas, Trello, metodología ágil, C#, patrones de programación

Abstract

In this document we follow the development of a platform-type 2D videogame from scratch, this covers the thought of the idea, the planning of this idea, the organization, the development process and the final result. For this we use Unity2D and the C# language. For the follow-up of this TFG, an agile methodology is used through the use of the Trello tool with the addition of an extension to record times in the tasks and with it, better follow-up.

The use of programming patterns is also shown and why to use it since each one has a different purpose when using it, one of these programming patterns is the Singleton.

The video game map is a fixed one that is designed from scratch taking maps from famous video games like “Hollow Knight” or “Ori and the Blind Forest” as a reference. The history of the video game is adorned with small comic touches, references to memes and references to other video games.

Keywords: videogame, Unity2D, platform, Trello, agile methodology, C#, programming patterns

Tabla de contenidos

Índice de imágenes

Índice tablas

1 Introducción

Este proyecto es un trabajo conjunto con Adrián Botella Perpiñá compañero de ingeniería del software, surgió al cursar las asignaturas de Desarrollo de Videojuegos 2D y Desarrollo de Videojuegos 3D. Durante Desarrollo de Videojuegos3D programamos un juego llamado Wendigo que se presentó en la feria de proyectos, el programar un juego en unity nos pareció entretenido e interesante, como la asignatura solo dura unos meses no tuvimos el tiempo suficiente a probar todas las cosas que nos gustaría, más tarde mientras cursaba la asignatura de Desarrollo de Videojuegos2D le propuse la idea a mi compañero de hacer un TFG conjunto en relación con un videojuego 2D ya que nos había gustado la experiencia de programar un videojuego y queríamos aprender más sobre ello.

Se ha analizado las características de los videojuegos del género *metroidvania* para el desarrollo del proyecto, para así determinar los factores que caracterizan este tipo de videojuegos

El videojuego desarrollado se trata de un juego de plataformas, en el que el jugador deberá de explorar el mapa para encontrar las entradas a los jefes y obtener las gemas que dejan al ser derrotados.

El jugador puede obtener diversos objetos de cofres distribuidos por el mapa, la calidad del objeto depende del tipo de cofre que se abra ya que dependiendo del cofre se puede obtener mejor equipamiento, al mismo tiempo el equipamiento que se obtenga tendrá las estadísticas randomizadas para así que el daño de cada partida nunca sea el mismo, y los objetos que normalmente son buenos pueden llegar a aparecer con malas estadísticas.

Los sistemas desarrollados durante el proyecto tienen cada uno un objetivo en el proyecto, por ello cada uno compone una pieza importante en el proyecto para su correcto funcionamiento.

Durante el desarrollo del proyecto se ha llevado una metodología ágil para el correcto desarrollo del proyecto, al mismo tiempo se han implementado patrones de programación para solucionar ciertas necesidades de programación. Se ha seguido un estilo de programación para así poder estructurar de forma ordenada el código, facilitar su entendimiento y su mantenimiento.

Los sistemas del proyecto se han sometido a diferentes pruebas para comprobar su correcto funcionamiento y la detección de errores, los errores que se detectan mediante estas pruebas han sido analizados para su posterior solución.

En el siguiente enlace se puede encontrar un video que se ha realizado explicando las mecánicas principales y el funcionamiento del videojuego:

https://media.upv.es/#/portal/video/62ba3370-2ae9-11ed-ba3f-a3333974a7c0

El videojuego que se ha desarrollado se puede probar en el siguiente enlace:

https://diegoger222.itch.io/tfg-diego-adrian **la contraseña es patata**, esta ideado para jugarse en una resolución de 1920 x 1080, aunque esta modificado para que la interfaz se adapte a la resolución. Los controles se pueden ver en el punto 5.5 Controles.

1.1 Motivación

La forma en la que se realiza un videojuego desde el punto de vista del software es interesante ya que es diferente al desarrollo de una aplicación. Los enfoques que se pueden llevar durante el desarrollo de un videojuego son diferentes a lo que normalmente se produce en las aplicaciones ya que la finalidad de un videojuego es entretener al usuario que lo utiliza.

El mundo de la programación es tan amplio que es imposible el poder conocer todo este mundo en las asignaturas que se imparten en la universidad ya que no hay suficiente tiempo para que los profesores enseñen todas las posibilidades que contienen las asignaturas, por ello el poder profundizar un poco más en algunos conceptos de la informática es factor a tener en cuenta, aunque el mundo de la programación crece cada día más.

Los videojuegos tienen una gran variedad de géneros diferentes en los que estos mismo se pueden llegar a dividir en subgéneros y en ocasiones crear un nuevo subgénero dependiendo de su éxito. Esto nos lleva a elegir un género para la creación del proyecto, en concreto se ha elegido el género metroidvania para la realización de este mismo. Se planteo hacer un proyecto enfocado a un género de puzles, pero al final esto se descartó por la idea de hacer un videojuego más fluido.

1.2 Objetivos

El objetivo principal del proyecto es desarrollar un videojuego 2D de tipo metroidvania. El cumplimiento de este objetivo estará marcado por obtener una versión del juego jugable con varios jefes y con una variedad de enemigos, al mismo tiempo tiene que ser viable en el marco de la jugabilidad.

También como objetivos del proyecto está el aplicar una metodología ágil durante el desarrollo del proyecto, ya que la aplicación de una metodología ágil en el ámbito de una aplicación software a un desarrollo de un videojuego puede ser diferente a lo que normalmente se realiza. Otro objetivo es el de poder aplicar patrones de programación en el proyecto.

Como objetivos secundarios definidos para poder llegar a realizar los objetivos principales se han definidos los siguientes:

- **Conseguir que una zona del mapa obligue al jugador a explorar.** Este subobjetivo está pensado para mejorar la jugabilidad de la idea principal.
- **Conseguir que los enemigos sean diferentes entre ellos.** Este subobjetivo está pensado para apoyar la idea principal de variedad de enemigos.
- **Añadir elementos con los que se puedan interactuar en el mapa.** Este subobjetivo está pensado para añadir variedad al mapa que mejore la jugabilidad, ya que al añadir diferentes elementos se fomenta que el jugador explore el mapa.
- **Hacer que los enemigos tengan diferentes acciones.** El que los enemigos realicen otras acciones a parte de atacar hace que la jugabilidad sea mejor.

1.3 Metodología

Se ha desarrollado el videojuego utilizando una metodología ágil. Mientras se aplicaba la metodología debíamos tener en cuenta el trabajo que podía ser realizado por dos personas y ajustar las tareas de los sprints para que fueran viables para poder realizar el trabajo correctamente y sin exceso de carga, a este trabajo también se sumaba el trabajo de investigación que empezó un mes antes de empezar el sprint 0 y también durante el desarrollo del proyecto.

Los sprints han sigo 6 siendo la duración de cada uno de 2 semanas, cada script alberga las tareas realizadas durante el desarrollo del proyecto como se puede ver en la imagen 1.

Dentro del tablero podemos encontrar unas secciones que dividen las UTs que están creadas, las divisiones son las siguientes:

- Backlog: Es donde se almacenan la UTs pensadas para implementar en el proyecto, algunas de las Uts del Backlog es posible que no lleguen a desarrollarse por falta de tiempo.
- NextSprint: Es donde se almacenan las UTs que ya se han decidido para el siguiente sprint están a la espera de que termine el sprint actual.
- Sprint Backlog: Es donde se almacenan las UTs que están planeadas para el sprint actual pero todavía no se están realizando.
- Programar: Es donde se almacenan las UTs en las que actualmente se están trabajando.
- Aplicar Pruebas de Aceptación: Es donde se almacenan las UTs que el trabajo ha sido realizado y deben se aplicarse pruebas para comprobar su correcto funcionamiento.
- Done: Es donde se almacenan las UTs terminadas del sprint que han pasado las pruebas de aceptación.
- Sprints terminados: Los sprints terminados se guardan como listas estas listas tienen el nombre de cada sprint empezando por el sprint 0.

El protocolo que se siguió para el uso y creación de las UTs fue el de crearlas en el backlog con un nombre descriptivo de que trataría la UT dentro de la UT se definía en más detalle el que debía realizar, cuando era el momento de pasarla a preparación para el siguiente sprint se definían los requisitos que debía cumplir para considerarse que había sido cumplida, después de que fuera añadida En el sprint se seguía el trabajo como se ha explicado anterior mente en las secciones que dividen el tablero.

También creamos etiquetas que podíamos poner en las diferentes UTs para tener un mejor control del trabajo las etiquetas que creamos son: Terminado, Pausa, Revisar, Bug, Esperar requisitos, Sonido, Mapa Visual, Esperar Prioridad.

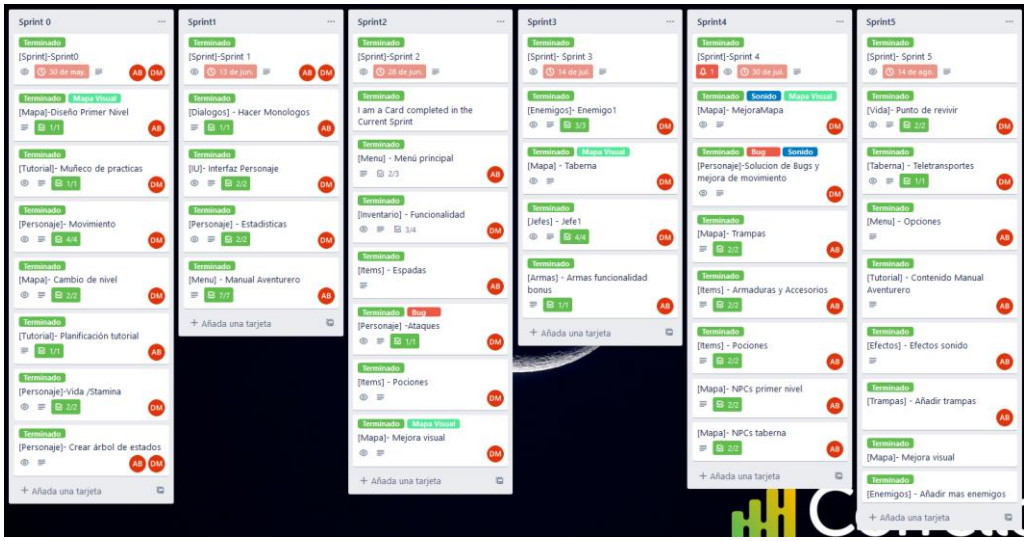

Imagen 1: Tablero Kanban

1.4 Estructura del documento

La estructura del documento se ha redactado de forma que permita entenderse el trabajo realizado.

El primer punto consiste en una introducción al proyecto, que muestra en que consiste este trabajo. En este punto se muestra el motivo del proyecto, los objetivos que se han planteado en el proyecto, la metodología usada y las colaboraciones.

En el segundo punto se analiza el estado del arte, explicando con ello alguna definición necesaria para entender conceptos del proyecto, se analiza los puntos fuertes que tienen algunos proyectos y se expone la propuesta sugerida.

El tercer punto contiene las herramientas utilizadas durante el desarrollo del proyecto, también por qué han sido utilizadas.

El cuarto punto explica la herramienta unity en más profundidad, que veneficios se pueden ver en ella y que funciones tiene que se van a utilizar, también en este punto se explican conceptos que se utilizan más adelante.

El quinto punto se muestran los diseños utilizados durante el desarrollo del proyecto, se muestran imágenes y explicaciones de para que se utiliza cada cosa.

En sexto punto se explica la implementación del videojuego desde el punto de vista de la programación, se explica que tiene cada componente importante y el funcionamiento de los scripts que componen el proyecto.

El séptimo punto contiene pruebas realizadas durante el desarrollo del proyecto, y los resultados que han dado estas apruebas, al mismo tiempo se explica si ha sido necesario aplicar modificaciones después de estas pruebas.

El octavo punto contiene las conclusiones, se resume lo logrado durante el proyecto y si se han logrado los objetivos deseados.

El noveno punto corresponde a los trabajos futuros, en este punto se exponen posibles trabajos que se podrían realizarse futuramente de continuar el proyecto.

El décimo punto es la biografía en ella se muestran lugares de donde se ha obtenido información.

1.5 Colaboraciones

Este proyecto esta creado con la ayuda de Adrián Botella Perpiñá. A continuación, se presenta unas tablas para mostrar la aportación de cada miembro, en la tabla 1 se refleja el trabajo realizado a lo que scripts se refiere y en la tabla 2 a otros trabajos realizados.

Scripts	Integrantes	
	Diego Ruiz Muñoz	Adrián Botella Perpiñá
Misiones		
Comprobador		X
ComprobadorMision		X
ManualAventurero		X
Menú		
MenuPrincipal		X
MostrarNumero		X
Settings		X
Diálogos		
ControlDialogos		X
PersonajeInteractuable		X
Textos		X
Personaje		
AtacandoBoolEspecialSalida	X	
AtacandoBoolEspecial	X	
AtacandoBool	X	
DesactivarColisiones	X	
BarraDeVida	X	
ControladorPersonaje	X	
Dano	X	
Escudo	X	
Mana	X	
MovimientoPersonaje	X	
AtaqueEspecial		X
InvocarEfecto		X
LanzarEfecto		X
LanzarMeteorito		X
Estadísticas	X	
Experiencia	X	
Cámara		
ControladorCMvcam	X	
ControladorMainCamp	X	
Enemigos		
Jefes		
CastEnem	X	
CastHab	X	
danyoHabi	X	
JefeControl	X	
Jefenumalt	X	
MovimientoAlmas	X	
MocimientoJefe	X	
VidaJefe	X	

Enemigos Normales		
ArbolCont	X	
DanyoVisible	X	
DanyoEnemigo	X	
EstadisticasEnemigo	X	
MovimientoCenti	X	
Muerte	X	
QuietoTime	X	
SeguirJugador	X	
Vida	X	
Enemigo Pacifico		
Cuervo	X	
CuervoGrito	X	
CuervoVuelo	X	
CuervoVuelta	X	
MovimientoCaw	X	
Inventario		
InformacionInventario	X	
Inventory	X	
Slot	X	
SlotEquipo	X	
Ítems		
Imodifier	X	
ItemControlador	X	
ItemObject	X	X
IU		
ControladorCanvas	X	
DanyoIU	X	
MovimientoIU	X	
ControladorEventos	X	
ControladorInventario	X	
Sonido		
ActivarSonido	X	
EjecutarSonido	X	X
ControladorSonido	X	
Mapa		
OcultarZona	X	
PilarTp	X	
InicioJugador	X	
SalidaJefes	X	
SalidaJugador	X	
TorreCheck	X	
TorreCuracion	X	
ActivarTrampa		X
ActivarTrampaOso		X
TrampaPinchoMovil		X
Destroy		X
Destroythis		X

Tabla 1: Scripts Realizados

Otros	Integrantes	
	Diego Ruiz Muñoz	Adrián Botella Perpiñá
Creación de ítems	X	X
Creación de IU	X	
Menú principal		X
Creación de mapa	X	
Efectos armas		X
Árbol estado personaje	X	X
Dialogos Personajes		X
NPCs		X
Manual Aventurero Texto		X
Historia	X	X
Árbol estado enemigos	X	
Trampas		X
Sonidos Añadidos	X	X

Tabla 2: Trabajo realizado

2 Estado del arte

La industria del videojuego crece más cada año, este crecimiento trae consigo muchos juegos que quedan marcados como referentes en su género y otros juegos que son eclipsados. Contra más avanza la industria del videojuego nuevos géneros de juegos aparecen y cada vez hay más opciones en el mercado, estas opciones crecen tanto en tipos de juego como en plataformas.

La temática de nuestro videojuego es Metroidvania [1] que es un subgénero de videojuegos de acción-aventura basado en un concepto de plataformas no lineal. El nombre de Metroidvania viene por dos juegos que quedaron marcados como referentes que son: Metroid y Castlevania. A continuación, se verán una serie de juegos relacionados con esta temática.

2.1 Videojuegos de tipo Metroidvania

Hollow Knight

Es un juego perteneciente al género metroidvania, publicado en 2017, desarrollado por Team Cherry.

Se enfoca principalmente en exploración, plataformas y combate.

Hollow Knight [2] (ver Imagen 2) es un juego que ha dejado su marca, tanto en su género como en general ya que fue muy bien recibido debido a su muy buena combinación de jugabilidad, historia, música, entorno. Una cosa interesante es que fue desarrollado en Unity y por ello mediante un proyecto de Unity que creo la comunidad podemos acceder a las escenas del juego para así ver cómo está construido el entorno.

Imagen 2: Hollow Knight

Dead Cells

Dead Cells [3] es un juego de plataformas inspirado en juegos tipo metroidvania, publicado en 2018 ha sido desarrollado por Motion Twin.

Dead cells (ver Imagen 3) es un juego que las partidas suelen ser rápidas dependiendo de cómo se juegue lo que favorece que se juegue muchas veces.

Una de las características del juego es el uso de diferentes armas, cada vez que se empieza una partida las armas te encuentras en su gran mayoría son distintas a la que encontraste en la anterior partida.

Imagen 3: Dead Cells

Ori and the Blind Forest

Es un juego perteneciente al género metroidvania, publicado en 2015, desarrollado por Moon Studios GmbH.

Ori and the Blind Forest [4] (ver Imagen 4) es un juego que se puede jugar de forma casual ya que la importancia del juego es adentrarte en la historia que cuenta.

El fuerte de este juego es la historia acompañada por la música, la jugabilidad del juego ayuda a que explores el mundo de este juego para profundizar en la historia.

Imagen 4: Ori and the Blind Forest

2.2 Motores de juego

En este punto mostraremos varios motores de juegos que existen en el mercado. Los motores de juego suelen estar especializados en cierta medida para una funcionalidad, esta funcionalidad generalmente está enfocada hacia un género de videojuego. Hay grandes empresas que utilizan varios motores dependiendo el juego que estén realizando en ese momento.

Amazon Lumberyard

Amazon Lumberyard [1] es un motor de videojuegos de código abierto, se ofrece de forma gratuita, tiene una interacción nativa hacia AWS y Twitch. Está enfocado en la producción de videojuegos 3D, al mismo tiempo facilita la interacción con los espectadores de Twitch.

Source Engine 2

Desarrollado originalmente por Valve [2] para creación de Half Life, es un motor de juego que permite el renderizado de escenas complejas sin perdida significativa del framerate. Tiene un sistema dedicado para la edición de niveles. Principalmente está destinado a la creación de mods para juegos ya existentes.

Unreal Engine

Unreal Engine [3] se ha posicionado como uno de los motores de juegos más completos. Comenzó como un motor para desarrollar Unreal pero con el paso del tiempo fue evolucionando para otorgar diferentes herramientas al usuario. Además, puede ser usado para crear juegos de diferentes géneros y plataformas.

CryEngine

CryEngine [4] fue el primer motor de juego que mostro impresionantes desarrollos gráficos para visualización a tiempo real de grandes espacios abiertos. Entre sus características se destaca una gran simulación de físicas que incluye simulación de líquidos, destrucciones y simulaciones avanzadas de cuerdas.

[1] https://aws.amazon.com/es/lumberyard/
[2] https://www.valvesoftware.com/es/
[3] https://www.unrealengine.com/en-US/
[4] https://www.cryengine.com/

2.3 Herramientas de gestión de proyecto

En este punto mostraremos varias herramientas que se pueden utilizar para gestionar un proyecto, estas herramientas fueron consideradas para gestionar el trabajo realizado.

Worki

Es la herramienta de apoyo para TUNE-UP Process [5]que es un framewoek para implantación del enfoque ágil. Es una herramienta muy completa que ofrece una forma completa y sencilla de aplicar una metodología ágil en un proyecto. Esta herramienta tiene un costo de 1500 € al año, pero ofrece una licencia gratuita para el uso académico. Esta herramienta es la que se usa en las asignaturas de PSW y PIN en la UPV.

Asana

Es una herramienta que permite visualizar el seguimiento de la metodología aplicada, ofrece una gran variedad de opciones para ajustar el tablero a las necesidades del usuario. Esta herramienta ofrece funcionalidades dependiendo del plan elegido, cada plan tiene un diferente coste monetario.

Monday

Monday [6] es una herramienta de gestión de proyectos ágiles, permite planificar, rastrear y entregar proyectos de equipo desde un solo espacio de trabajo. Tiene varios planes, entre ellos un plan gratuito para individuales con ciertas limitaciones.

[5] http://www.tuneupprocess.com/
[6] https://monday.com/lang/es/

2.4 Critica al estado del arte

Como hemos visto casi gran parte de los juegos siempre se caracteriza que al hacer distintas partidas siempre hay una parte que no suele cambiar y es las estadísticas de las armas que se utilizan o de los objetos, en algunos juegos varia la arma que te puede tocar y en otros esta parte es estática.

De los videojuegos del tipo metroidvania que actualmente dominan el mercado se han desarrollado en unity, esto nos muestra el potencial que se puede llegar a sacar de esta herramienta dependiendo de su uso y experiencia. También como se ha podido comprobar durante la investigación unity se puede usar para realizar muchas otras funciones como la creación de películas [5] [6].

En casi todos estos juegos la historia y la ambientación tiene una parte muy importante.

Muchos juegos dejan elegir el nivel de dificultad normalmente ofrecen las dificultades entre fácil, normal , difícil y en ocasiones ofrecen mayores dificultades para los jugadores que buscan algo más complicado, generalmente las dificultades suelen estar distribuidas de forma que la dificultad normal es para la gente casual, la fácil para la gente que busca el pasarse el juego rápido sin nada de dificultad, la difícil suele ofrecer un poco más de dificultad que la normal pero generalmente la diferencia de normal a difícil no suele variar, hay algunos juegos que no ofrecen el elegir dificultad, la dificultad es la misma siempre por lo que suele estar adaptado de forma que hay diferentes rutas de llegada al final, estas rutas serian la dificultad del juego.

2.5 Propuesta

El juego desarrollado pretende ofrecer una variante a lo que la jugabilidad normal está acostumbrada, esta variante se centra en que el equipamiento que se obtiene es tanto aleatorio a lo que te puede tocar y aleatorio en las estadísticas de estos, así las partidas jugadas serán diferentes cada una ya que un objeto no tendrá los mismos valores de bonificación en las partidas jugadas, además que cada arma realizará un ataque especial diferente.

La interfaz en los juegos toma una parte muy importante ya que al final es de donde sacas la información de que está pasando, al mismo tiempo una interfaz compleja puede ser un gran problema que puede llegar a desesperar al jugador y hacer que la experiencia de juego se arruine, por ello queremos optar por una interfaz simple que, de los datos necesarios para el jugador, la interfaz se detallara en otro punto más adelante.

Al mismo tiempo para acompañar a la jugabilidad se expone una historia más cómica y inesperada de acorde a la ambientación.

El nivel de dificultad no se podrá elegir ya que en mi opinion la mejor experiencia que se puede llegar a ofrecer es un reto acorte a la historia.

3 Herramientas utilizadas

En este apartado se muestra el software utilizado para el proyecto.

Unity

Unity [7] se trata de un motor de videojuego multiplataforma creado por Unity Technologies, disponible para las plataformas Microsoft Windows, Mac OS, Linux. Mas adelante se indagará más en los aspectos de Unity. Esta herramienta ha sido elegida ya que la ha sido utilizada anteriormente en asignaturas de Videojuegos 2D y Videojuegos 3D. El utilizar esta herramienta es por el motivo que en el momento de cursar las asignaturas se consideró que era bastante cómoda y completa para el desarrollo de este tipo de proyecto por esto la volvimos a utilizar. Los demás motores de videojuegos que se investigaron también eran buenos para la creación de videojuegos, pero se descartaron ya que están destinados a cubrir otro tipo de características que no necesitamos cubrir. La versión que utilizamos es la 2020.3.14f1 ya que es una versión bastante estable tiene algunos inconvenientes como que la extensión de GitHub no funciona correctamente, pero esto se puede solucionar con la aplicación de GitHub de escritorio, en otros aspectos no suele dar fallos de compatibilidad a la hora de programar y utilizar extensiones, algunas versiones más recientes posen ciertos fallos que afectarían a la hora del desarrollo o retrasarían ciertos avances.

Microsoft Visual Studio

Es un entorno de desarrollo integrado para Windows y macOS que es compatible con múltiples lenguajes de programación. En el proyecto se ha utilizado para realizar la programación en el lenguaje C#. El motivo por el que utilizar esta herramienta y no otra es porque al conectarse con el editor de Unity es muy fácil del programar y consultar bibliotecas, no se ha usado el Visual Studio Code porque en ocasiones daba un fallo a la hora de conectar el programa con las bibliotecas que ofrece Unity por ello se decidió el utilizar Microsoft Visual Studio.

Git

Git [8] es un sistema de control de versiones distribuido gratuito y de código abierto diseñado para manejar todo, desde proyectos pequeños hasta proyectos muy grandes, con rapidez y eficiencia.

> **GitHub:** Se trata de una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas. El elegir esta herramienta es porque se ha usado anteriormente para la creación de proyectos de otras asignaturas y es muy completa, también porque estamos muy acostumbrados a usar GitHub.
>
> **GitHub Desktop:** Se trata de la aplicación de escritorio de GitHub, nos permite subir el trabajo realizado a nuestro repositorio Git. Se ha usado esta herramienta por la necesidad de buscar una solución al problema de fallo de la extensión de GitHub en la

[7] https://unity.com/es
[8] https://git-scm.com/

versión utilizada en el proyecto de Unity, con esta herramienta podemos compartir el trabajo realizado para poder ponerlo en común a la hora de programar y de realizar cambios en el proyecto.

Trello

Trello [9] es una herramienta visual que permite a los equipos gestionar cualquier tipo de proyecto y flujo de trabajo, así como supervisar tareas. El motivo por el cual hemos elegido esta herramienta es para llevar un seguimiento de la metodología ágil aplicada, la ventaja de esta herramienta es que es simple y al mismo tiempo completa ya que se puede ajustar a lo que queremos con mayor facilidad que otras herramientas investigadas. Las demás herramientas que se investigaron y valoraron para gestionar la metodología que queremos aplicar se descartaron porque casi todas tenían funciones que requerían un plan de pago, con trello podemos cubrir casi todas las necesidades que queremos, y para aquellas que las funcionalidades no pueden cubrir se solucionó escribiendo en un documento.

UnityAssetStore

Es una tienda ofrecida por Unity en la que los usuarios pueden postear sus creaciones tanto de videojuegos como de assets, por lo que es útil para encontrar assets para nuestro proyecto. El motivo por el que se ha usado para obtener ciertos assets es que es una herramienta ofrecida por Unity y por ello también tiene una extensión en el editor de Unity haciendo muy fácil en añadir archivos al proyecto ya que solo debemos añadirlos desde la página a nuestros assets y desde la extensión del proyecto elegir los que queremos añadir al proyecto.

Itch.io

Itch.io [10] es una página que ofrece la posibilidad de compartir creaciones, de ella se pueden obtener también assets y se puede publicar los videojuegos creados en ella por lo que se pueden compartir con mayor facilidad. El motivo por el que se ha utilizado esta herramienta es porque ofrece una gran variedad de assets en su tienda y ofrece otras funciones como la de poder subir nuestro proyecto de videojuego a la página de forma en la que se pueda ejecutar el proyecto desde una página, de esta manera no se requiere que quien quiera probar el proyecto tenga que descargar archivos en su ordenador y así nos libramos de algunos problemas de incompatibilidad de software.

[9] https://trello.com/
[10] https://itch.io/

4 Unity

En este apartado se indaga más en profundidad Unity ya que es el motor utilizado en el desarrollo del proyecto.

4.1 Motor

Unity es un motor multiplataformas accesible para todo el mundo, dispone de versiones de pago, pero al mismo tiempo ofrece licencias gratuitas pensadas para usuarios normales, las licencias de pago están enfocadas a las empresas.

El motor ofrece soporte para gráficos en 2D y 3D, además ofrece un gran abanico de posibilidades debido las físicas, iluminaciones y colisiones que ofrece entre muchas opciones, aunque también se usa en otros ámbitos como en la arquitectura y en la creación de películas [5]. También se puede complementar las funcionalidades del motor añadiendo herramientas externas ya sean ofrecidas como complemento por el programa o hechas por otros usuarios.

A continuación, se profundiza en diferentes aspectos del motor que se utilizan en el desarrollo del proyecto.

4.1.1 Escenas

Las escenas son completamente necesarias en el proyecto, contienen todos los elementos que serán visibles o necesarios en el proyecto.

Una forma más fácil de entender las escenas es pensar en ellas como niveles de un juego, solo puedes estar en un nivel al mismo tiempo y el cambio de nivel o desplazamiento sería el cambio de escena a una nueva. Por ello a la hora de utilizar las escenas para cambiar entre ellas se debe planear de que escena a cuál se ira.

El proceso de carga de una escena depende de cuanto tenga que cargar la escena a la que se va, una escena con muchos GameObject tardara mucho más en estar lista que una que tenga pocos. En nuestro caso tenemos una escena con mucha carga en ella que tarda considerablemente más tiempo en cargar comparado con las otras escenas, aunque sigue sin ser un tiempo excesivo de espera.

4.1.2 GameObject

Un *GameObject* [7] es una unidad básica para construir cualquier cosa que se necesite en una escena, cada *GameObject* perteneciente a una escena se encuentra listado en la jerarquía de Unity.

Los elementos que tiene in *GameObject* se les llama componentes, estos componentes proporcionan una funcionalidad, por ello cada *GameObject* puede tener diferentes Componentes estos componentes se pueden modificar en el editor de Unity o mediante código.

A continuación, se muestran y explican los componentes más importantes que podemos utilizar en los *GameObject*.

- *Transform:* Es el componente que contiene la posición, la rotación y la escala del objeto en el espacio de la escena. Si un GameObject con
- *Rigibody:* Permite a los *GameObject* actuar bajo el control de la física. El *Rigibody* puede recibir fuerza en sus objetos para hacer que se muevan de una manera más realista. []
- *Colliders:* Da un volumen al objeto para poder recibir colisiones. Se pueden dar ciertas características como *Trigger* que permite ser atravesados por otros objetos.
- *Animator:* Permite añadir animaciones a un objeto mediante controladores se puede interactuar con este componente por código.
- *Audio source:* Permite reproducir sonidos de manera que parezcan ser realizados por el objeto o como entorno, se puede modificar los ajustes de sonido para que la reproducción se ajuste a la deseada.
- *Scripts:* Es el componente que permite manipular el *GameObject* y sus componentes mediante el uso del código, se puede ajustar la visibilidad los parámetros de estos, más delante de profundiza más en los *Scripts*.

Los *GameObject* pueden ser almacenados en la carpeta del proyecto pasando así a ser llamados *prefabs*. Los *prefabs* son Objectos que pueden crear copias de este, al modificar el *prefab* se modificaran todas sus copias, esto hace más fácil el cambiar parámetros de objetos que se repiten.

4.1.3 Programación

Mediante el uso de *scripts* Unity permite la programación para modificar las funcionalidades de los objetos. El motor es compatible con varis lenguajes de programación entre los lenguajes el que es más usado y el que usamos es C#.

Lo que hace que los scripts puedan comportarse como un componente más de los *GameObject* es la clase *MonoBehavior*, cualquier clase que herede de esta podrá adquirir el comportamiento de un componente.

Una de las ventajas de usar los scripts es que podemos modificar datos sin tener que entrar en el script siempre y cuando las variables sean públicas.

4.1.4 Flujo de ejecución

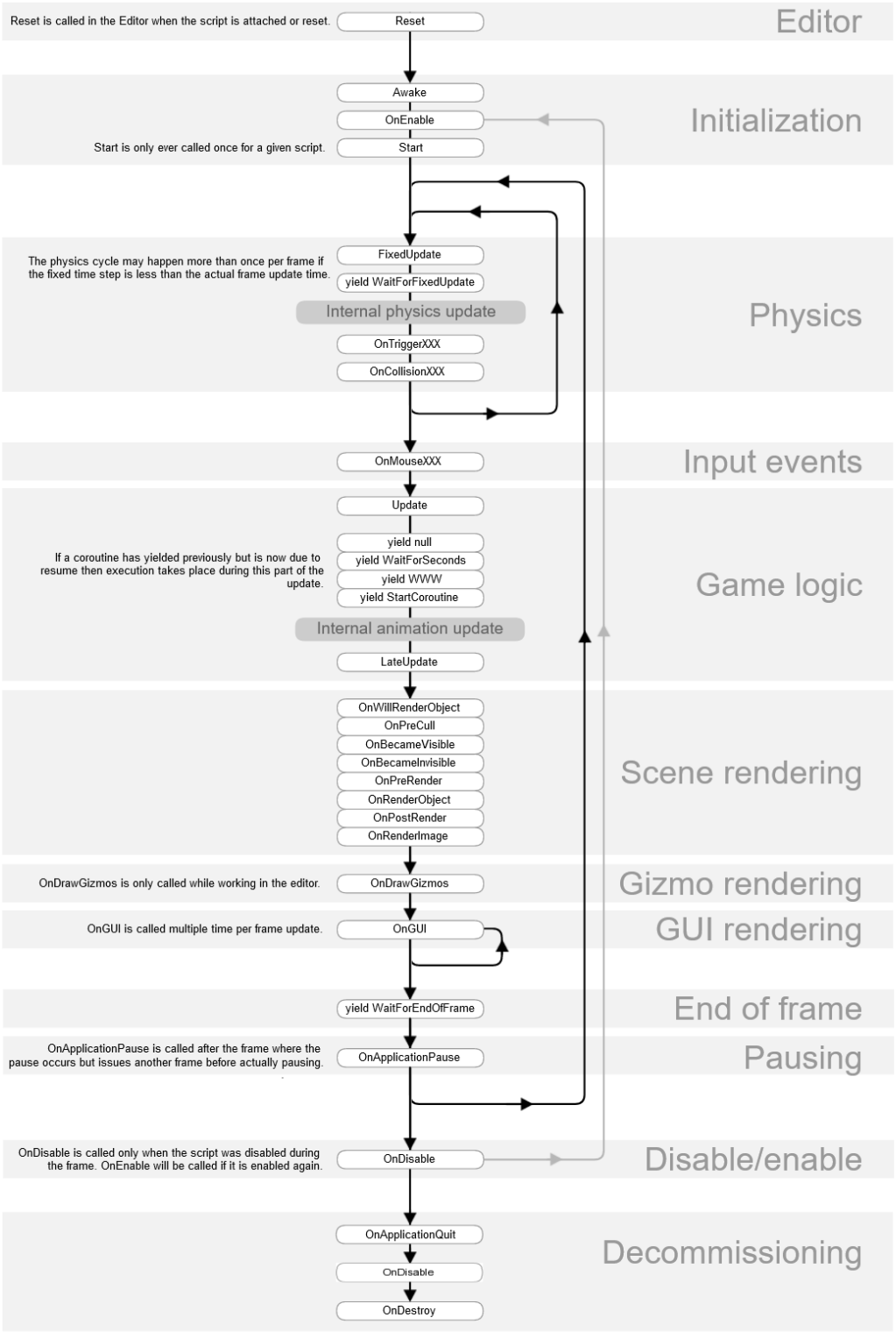

Imagen 5: Diagrama de flujo de ejecución de un script

El flujo de ejecución [8] (ver Imagen 5) a la hora de planear es importante ya que al final es el orden que seguirá un script a la hora de ejecutarse.

A continuación, se explica las llamadas utilizadas en el flujo de ejecución que se consideran más relevantes:

Editor

- **Reset:** El reset es llamado para inicializar las propiedades del script cuando es por primera vez adjuntado a un objeto y cuando el comando reset es utilizado.

First Scene Load (Primera escena cargada)

- **Awake**: Es una función que siempre se llama antes del Start y después de que un prefab sea instanciado.
- **OnEnable:** Es llamada si el objeto este activo al instanciarse.
- **OnLevelWasLoaded:** se ejecuta cuando toda la información del nivel actual ha sido cargada, es útil para hacer esperas para buscar objetos.

Antes del primer frame

- **Start:** El método se llama antes de la primera actualización de frame solo si la instancia del script está activa.

Entre frames

- **OnApplicationPause**: Esto es llamado al final del frame donde la pausa es detectada.

Update Order (Orden de Actualización)

- **FixedUpdate**: Puede ser llamada varias veces por frame.
- **Update:** Se llama una vez por frame. Es la función principal para las actualizaciones de frames.
- **LateUpdate:** es llamada una vez por frame después de que el Update termine.

Corrutinas

- **Yield:** La corrutina va a contar después de que todas las funciones del update hayan sido llamadas en el siguiente frame.
- **Yield WaitForSeconds:** Espera un tiempo determinado establecido.
- **Yield WaitForFixedUpdate:** Continua después de que todos los FixedUpdate hayan sido llamados en todos los scripts.
- **Yield StarCoroutine:** Encadena la corrutina.

Cuando el Objeto es Destruido:

- **OnDestroy;** Esta función es llamada después de que todas las actualizaciones de frame para el último frame de la existencia del objeto.

Cuando Salga

- **OnApplicationQuit:** Esta función es llamada en todos los game objects antes de que se salga de la aplicación.
- **OnDisable:** Esta función es llamada cuando el comportamiento se vuelve inactivo o deshabilitado.

También debemos de hablar de los métodos que ayudan a la detección de las colisiones dentro del proyecto:

- *OnTriggerEnter, OnTriggerExit, OntriggerStay:* son funciones que se activan cuando dos colisionadores se encuentran y alguno de ellos lleva activado el parámetro de *trigger*, este parámetro hace que pueda ser atravesado por otros objetos. Las funciones detectan la entrada, salida y la permanencia de los objetos con un *collider.*
- *OnCollisionEnter, OnCollisionExit, OnColissionStay:* son funciones similares a las anteriores explicadas con la diferencia que son utilizadas para objetos de no pueden ser atravesados.

4.2 Editor

El editor de Unity (ver Imagen 6) es donde tiene lugar gran parte del desarrollo del proyecto. A través de él podemos crear escenas, *GameObject* y administrar los componentes. Nos permite ejecutar pruebas de forma sencilla sin necesidad de exportar archivos. Podemos personalizar las ventanas que queramos ver para así ajustar las opciones a las deseadas en el momento.

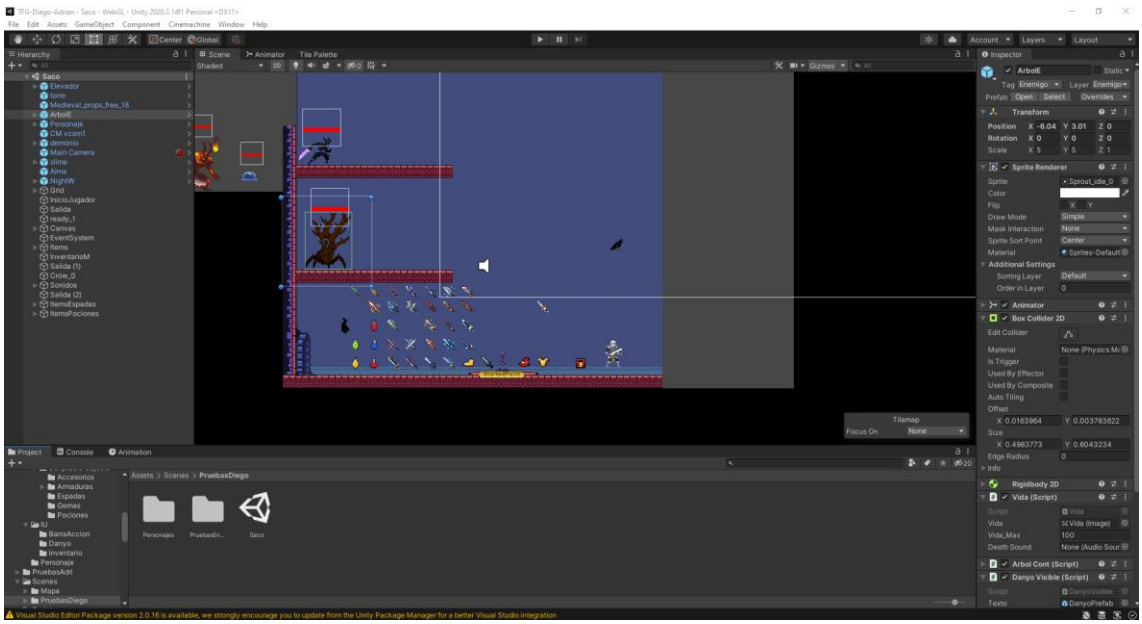

Imagen 6: Editor de Unity

4.2.1 Ventanas principales

Unity ofrece una gran variedad de ventanas para poder modificar y trabajar con una gran variedad de opciones, en este punto se introducen las ventanas más importantes utilizadas a la hora del desarrollo.

Scene

Esta ventana es la que nos permite poder interactuar con mapa en el que estemos trabajando

Game

Es la ventana en la que podemos ver lo que el objeto cámara nos muestra, desde esta ventana podemos ejecutar el proyecto sin tener que exportar nada así las pruebas se realizan con mayor facilidad.

Hierarchy

Es la ventana donde se muestra la jerarquía de los objetos en el proyecto desde ella podemos acceder a los objetos con mayor facilidad para así modificarlos o añadir más objetos, eliminar objetos….

Inspector

Esta ventana nos permite acceder a los detalles del objeto y así poder ver los componentes asociados al objeto y poder añadir, eliminar o modificar estos componentes.

Project

La ventana de project es el explorador de archivos de unity en el que nos permite guardar los scripts, prefabs, imágenes … es decir nos permite almacenar todo lo necesario para el proyecto, también nos permite crear carpetas para llevar una mejor organización.

Consola

La consola nos permite ver los warning, errores y los mensajes imprimidos por código, es una manera rápida de controlar si hay errores de código en las pruebas.

Animator

Es una ventana que nos permite crear mediante una máquina de estados un controlador de las animaciones asociadas a un objeto, el Animator se explicara en más profundidad más adelante.

4.2.2 Ejecutable

El editor de Unity ofrece una opción de generar un ejecutable en las opciones de build en esta ventana nos permite colocar las escenas de nuestro proyecto así poder pasar a otra escena mediante el código, también en esta parte nos ofrece una gran variedad de formas de hacer el ejecutable. Una vez se realice el ejecutable la primera escena a la que se entre será la que este en la parte superior.

Las opciones de plataformas del ejecutable se pueden ver en la imagen 7.

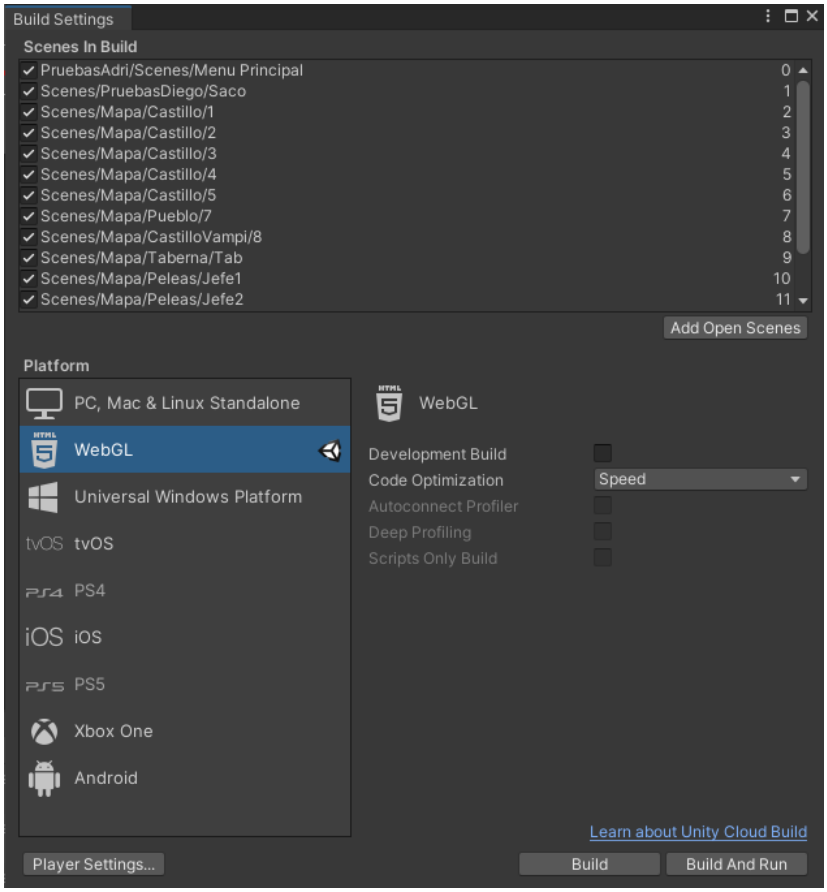

Imagen 7: Ventana Build

El ejecutable del proyecto esta creado en la plataforma de Web ya que así es más conveniente.

5 Análisis y diseño

En este apartado se exponen las diferentes partes que componen el videojuego, se hablara sin entrar en detalle del código, los detalles del código se mostraran en otro apartado.

5.1 Análisis de requisitos

En este punto se muestran los requisitos que tiene el sistema.

5.1.1 Requisitos Funcionales

Los requisitos funcionales se han dividido en los siguientes.

- Sistema del personaje
- Sistema de inventario
- Sistema de estadísticas
- Sistema de sonido
- Sistema de enemigos
- Entorno mapa
- Sistema de diálogos
- Sistema de misiones

Los requisitos funcionales contienen casos de uso. Los casos de uso ayudan a controlar el correcto funcionamiento de los sistemas. A continuación, se muestran algunos de los casos de uso de cada requisito funcional.

Sistema del personaje

El sistema del personaje engloba los requisitos relacionados con el correcto funcionamiento del jugador.

Referencia	CU-01
Nombre	Moverse
Descripción	Permite moverse horizontalmente en el mapa. 1. El jugador presiona la tecla de movimiento. 2. El sistema comprueba si al jugador se le permite moverse, en caso de poder se calcula hacia qué lugar se moverá.
Actor	Jugador

Referencia	CU-02
Nombre	Saltar
Descripción	Permite saltar en el mapa

	1. El jugador presiona la tecla de salto 2. El sistema comprueba si el jugador está en el suelo 3. En caso de estar en el suelo, se calcula la velocidad y dirección de desplazamiento.
Actor	Jugador

Referencia	CU-03
Nombre	Atacar
Descripción	Permite al jugador atacar 1. El jugador presiona la tecla de atacar. 2. El sistema comprueba si es posible atacar y si está en el suelo. 3. En caso de estar en el suelo el sistema ejecuta la acción de atacar. 4. En caso de estar en el aire el sistema ejecuta la acción de atacar en el aire.
Actor	Jugador

Referencia	CU-04
Nombre	Guardar/sacar espada
Descripción	Permite guardar/sacar la espada 1. El jugador presiona la tecla de guardar/sacar espada 2. El sistema comprueba el estado de la espada 3. El sistema guarda/saca la espada dependiendo del estado
Actor	Jugador

Referencia	CU-05
Nombre	Activar el escudo
Descripción	Permite al jugador activar un escudo 1. El jugador presiona la tecla del escudo. 2. El sistema comprueba si está disponible el escudo. 3. En caso de estar disponible el sistema activa el escudo.
Actor	Jugador

Referencia	CU-06
Nombre	Activar habilidad
Descripción	Permite al jugador utilizar la habilidad de la espada. 1. El jugador presiona la tecla de la habilidad. 2. El sistema comprueba si el jugador tiene mana suficiente y si tiene una espada equipada. 3. En el caso de tener mana y una espada con habilidad el sistema lanzara la habilidad correspondiente. 4. En caso de tener mana y no tener una espada con habilidad el sistema realiza solo el ataque especial sin habilidad. 5. En caso de no tener mana el sistema no activa nada.
Actor	Jugador

Referencia	CU-07
Nombre	Rodar
Descripción	Permite al jugador rodar. 1. El jugador presiona la tecla de rodar. 2. El sistema comprueba si el jugador está en el suelo y si rodar está disponible. 3. De estar en el suelo y disponible el sistema iniciara la acción de rodar.
Actor	Jugador

Referencia	CU-08
Nombre	Abrir/cerrar inventario
Descripción	Permite al jugador Abrir/cerrar inventario 1. El jugador presiona la tecla del inventario. 2. El sistema comprueba el estado del inventario. 3. Dependiendo del estado del inventario el sistema lo abre o lo cierra.
Actor	Jugador

Referencia	CU-09
Nombre	Vida jugador
Descripción	Permite al jugador recibir daño. 1. El jugador recibe daño 2. El sistema detecta que tipo de daño es. 3. El sistema calcula el daño que se recibirá dependiendo de las estadísticas del jugador. 4. El sistema resta la vida al jugador 5. El sistema comprueba la vida restante 6. Si la vida restante es menor o igual a 0 el sistema activa el proceso de muerte. 7. Si la vida no es inferior a 1 el sistema actualiza la barra de vida del jugador.
Actor	-

Referencia	CU-10
Nombre	Usar poción
Descripción	Permite al jugador recuperar vida o mana 1. El jugador usa el ítem de la poción. 2. El sistema comprueba que id tiene la poción. 3. Dependiendo del id el sistema suma una cantidad a la vida o al mana.
Actor	Jugador

Sistema de inventario

Referencia	CU-11
Nombre	Añadir un objeto al inventario
Descripción	Permite recoger un objeto del suelo 1. El jugador entra en contacto con un objeto. 2. El sistema comprueba si es un objeto stackable 3. En caso de ser stackable, el sistema comprueba si ya hay un mismo objeto en el inventario. 4. En caso de haber un mismo objeto se recoge el objeto y se le suma una unidad 5. En caso de no estar en el inventario el sistema comprueba si hay un espacio disponible 6. En caso de haber un espacio disponible el sistema recoge el objeto y lo añade a un slot 7. En caso de no ser stackable el sistema comprueba si hay un slot libre 8. En caso de haber un slot libre, el sistema recoge el objeto y lo añade a un slot.
Actor	Jugador

Referencia	CU-12
Nombre	Ver la información de un objeto.
Descripción	Permite al jugador ver la información de un objeto 1. El jugador presiona la tecla de ver información sobre un ítem en el inventario. 2. El sistema comprueba la información del objeto. 3. El sistema muestra una interfaz en la que se ve la descripción del objeto y sus estadísticas.
Actor	Jugador

Referencia	CU-13
Nombre	Usar Objeto
Descripción	Permite al jugador usar un objeto. 1. El jugador presiona la tecla usar sobre un ítem en el inventario. 2. El sistema comprueba si se trata de un objeto equipable. 3. De ser un objeto equipable el sistema comprueba si hay un espacio disponible en el inventario de equipamiento. 4. Si hay un espacio en el inventario de equipamiento el sistema mueve el objeto al espacio disponible. 5. De no tratarse de un objeto equipable el sistema usara el objeto.
Actor	Jugador

Referencia	CU-14
Nombre	Desequipar equipamiento
Descripción	Permite al jugador desequipar un objeto. 1. El jugador presiona la tecla de usar sobre un ítem equipado. 2. El sistema comprueba que haya espacio en el inventario. 3. En caso de haber espacio el sistema mueve el ítem a un slot libre en el inventario.
Actor	Jugador

Sistema de estadísticas

Referencia	CU-15
Nombre	Subida de nivel
Descripción	El jugador sube de nivel al llegar 1. El jugador obtiene la experiencia necesaria para subir nivel. 2. El sistema detecta que la experiencia obtenida es mayor que la necesaria para subir nivel. 3. El sistema resta la experiencia necesaria a la obtenida y sube un nivel al jugador.
Actor	-

Referencia	CU-16
Nombre	Añadir estadísticas de objeto
Descripción	El jugador obtiene las estadísticas de un objeto 1. El jugador se equipa un equipamiento nuevo 2. El sistema detecta el cambio en el slot de equipamiento 3. El sistema suma las estadísticas correspondientes del ítem a las estadísticas del jugador.
Actor	-

Referencia	CU-17
Nombre	Utilizar puntos de estadísticas
Descripción	El jugador usa los puntos obtenidos por el nivel 1. El jugador selecciona en que quiere gastar sus puntos. 2. El sistema añade un punto usado a la estadística seleccionada y resta un punto a los puntos disponibles del jugador.
Actor	Jugador

Sistema de sonidos

Referencia	CU-18
Nombre	Activar Sonido
Descripción	Una animación que tiene un sonido lo ejecuta. 1. La animación envía el nombre del sonido que se debe de ejecutar 2. El sistema comprueba el nombre del sonido con los nombres de sonidos guardados. 3. Si el nombre del sonido es encontrado entre los guardados el sistema ejecuta el sonido correspondiente.
Actor	-

Referencia	CU-19
Nombre	Ajustar Sonido
Descripción	El sonido se ajusta al valor seleccionado. 1. El usuario selecciona el valor de sonido deseado. 2. El sistema ajusta los sonidos al valor seleccionado.
Actor	Jugador

Sistema de enemigos

El sistema de enemigos es donde se analizan las acciones que hacen los enemigos.

Referencia	CU-20
Nombre	Movimiento enemigo
Descripción	El enemigo se mueve a la dirección 1. El enemigo elige la dirección a la que ir. 2. El sistema comprueba si es posible. 3. El sistema calcula la nueva dirección y velocidad del enemigo.
Actor	-

Referencia	CU-21
Nombre	Ataque enemigo
Descripción	El enemigo ataca al jugador 1. El enemigo intenta realizar la acción de atacar. 2. El sistema comprueba si el ataque está disponible. 3. De ser así el sistema empieza la animación de ataque del enemigo.
Actor	-

Referencia	CU-22
Nombre	Recibir daño
Descripción	El enemigo recibe daño del jugador. 1. El enemigo recibe daño del jugador. 2. El sistema calcula el daño recibido en base a la defensa. 3. El sistema resta la vida al enemigo. 4. El sistema comprueba que la vida sea superior a 0 5. De ser la vida inferior o igual a 0 el sistema activa la muerte del enemigo. 6. Al morir el sistema debe de entregar la experiencia al jugador.
Actor	-

Referencia	CU-23
Nombre	Habilidad enemigo
Descripción	El enemigo de tipo jefe atacara con una habilidad 1. EL enemigo accionara la habilidad. 2. El sistema comprobara si la habilidad está disponible. 3. De estar la habilidad disponible el sistema accionara la habilidad.
Actor	-

Entorno mapa

El entorno mapa hace referencia a las funcionalidades que están distribuidas por el mapa y no encajan en ninguna de las funcionalidades anteriores.

Referencia	CU-24
Nombre	Torre de TP
Descripción	Permite al jugador teletransportarse a la taberna 1. El jugador interactúa con la torre. 2. El sistema cambia la escena a la taberna.
Actor	Jugador

Referencia	CU-25
Nombre	Torre Curación
Descripción	Permite al jugador curarse la vida 1. El jugador entra en contacto con la torre. 2. El sistema comprueba si quedan curas disponibles en la torre. 3. En caso de quedar, el sistema cura al jugador. 4. En caso de no quedar, el sistema no hace nada.
Actor	Jugador

Referencia	CU-26
Nombre	Torre de guardado
Descripción	Guarda la ubicación del jugador. 1. El jugador entra en contacto con la torre. 2. El sistema guarda la torre como último lugar de paso.
Actor	Jugador

Referencia	CU-27
Nombre	Cofres
Descripción	Los cofres deben de dejar un objeto. 1. El jugador interactúa con el cofre. 2. El sistema comprueba el tipo de cofre, dependiendo del tipo del cofre selecciona las probabilidades de loot. 3. El sistema genera un numero aleatorio. 4. Dependiendo del numero que ha salido se elige una rareza. 5. Se elige un objeto aleatorio de la rareza seleccionada. 6. El sistema deja caer el objeto en el lugar del cofre.
Actor	Jugador

Referencia	CU-28
Nombre	Zona Oculta
Descripción	Las zonas ocultas deben de desactivarse al entrar en contacto con el jugador. 1. El jugador entra en contacto con una zona oculta. 2. El sistema comprueba si la colisión es del jugador. 3. Si es el jugador quien entro en la zona oculta el sistema desactiva la zona para dar visibilidad.
Actor	Jugador

El sistema de diálogos y sistema de misiones son desarrollados por otra persona, por ello se explicarán en su correspondiente TFG.

5.1.2 Requisitos No Funcionales

Los requisitos no funcionales especifican el atributo de calidad de un sistema de software. Algunos de los requisitos no funcionales identificados son los siguientes:

RNF-01	El juego deberá de funcionar en web.
RNF-02	El juego deberá de cargar los niveles en menos de 1 minuto.
RNF-03	El juego deberá de poder ser usado desde diferentes sistemas operativos.

5.2 Concepto

La idea principal del proyecto es desarrollar un videojuego de tipo *metroidvania* en el que debes avanzar en el mapa para llegar a los jefes y así conseguir las gemas. La dificultad del juego la marca el nivel del jugador y la zona en la que se encuentra.

El juego dispone de varias zonas que dependiendo de la zona se encuentras más o menos obstáculos, siendo así a zona de la mazmorra la que más obstáculos tiene.

El equipamiento del juego se obtendrá de forma aleatoria de los cofres que se encuentren por el mapa y al mismo tiempo las estadísticas del equipo variaran en cada partida, con esto el juego busca ser uno de partidas rápidas que ofrezcan cierta variación en las partidas para así intentar llegar al final con lo que se obtiene.

5.3 Ambientación

En este punto se introduce la historia en la que está ambientado el videojuego y la que se ha intentado crear todo en sentido a esta historia.

El protagonista es un mensajero que ha recibido la misión de entregar un importante mensaje al rey: El anterior héroe ha muerto que estaba encargado de reunir las amatistas necesarias para reparar el sello de un enemigo de gran amenaza que yace en las mazmorras del castillo. Dicho sello está a punto de romperse y solo reuniendo las amatistas puede repararse. El rey, sin ningún plan B, nombra al mensajero nuevo héroe y le encomienda la misión del antiguo héroe. Al mensajero no le gusta la idea, pero sabe que no tiene otra opción.

El protagonista empieza su aventura a lo desconocido con el equipamiento más básico que le han dado y va conociendo detalles del mundo mediante personas que va encontrándose que le explican cosas brevemente.

5.4 Diseños

En este punto se muestran los diseños del videojuego.

Mapa

La idea principal que se tenía con el mapa era realizar diferentes zonas ambientadas en diferentes biomas, empezamos ideando lo que sería un mapa de todo el mundo del videojuego de acorde a la historia que pensamos (ver Imagen 8), como el mapa era grande decidimos hacer unas zonas, el hacer primero un mapa general era para poder darle un sentido a la historia, aunque no se llegara a realizar todas las zonas.

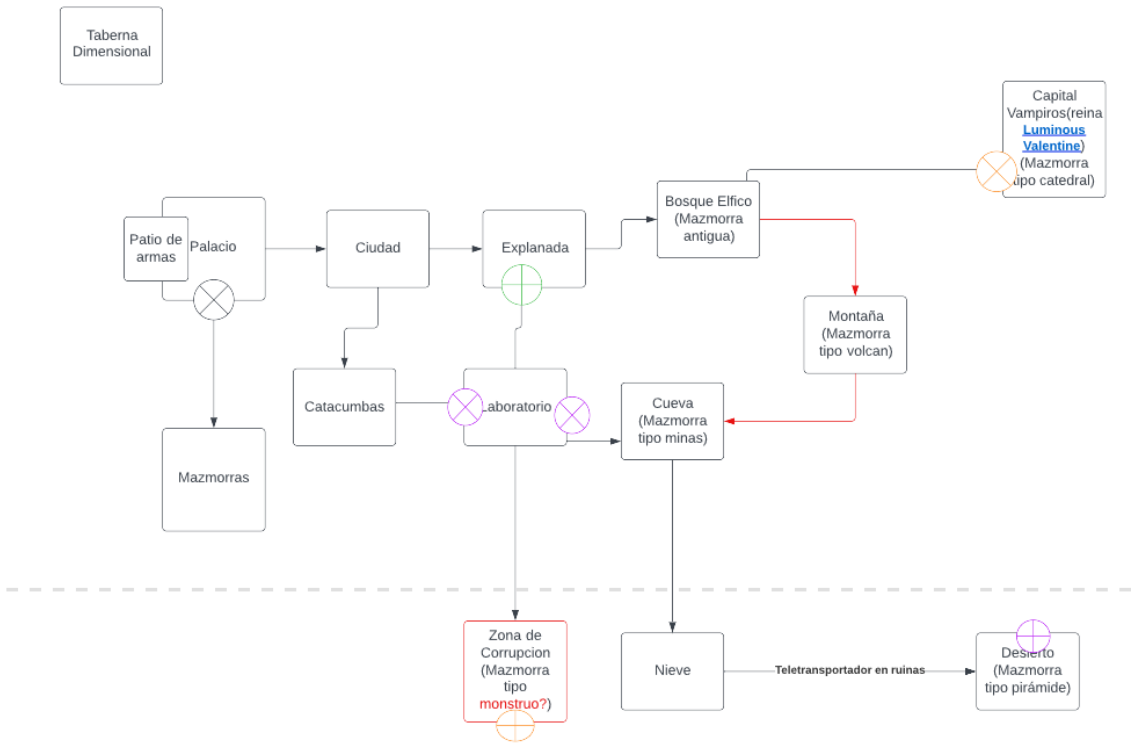

Imagen 8: Primera idea de mapa

Tras tener este mapa general decidimos acotar las zonas y al final se implementaron las siguientes zonas: Palacio, Ciudad, Bosque, Capital Vampiros, Taberna y las zonas de los jefes.

La zona más grande es la Capital Vampiros que se diseñó al final como una mazmorra (ver Imagen 9). Casi todas las zonas solo tienen una escena asociada, menos el castillo que son 5 escenas simples. La zona de la mazmorra fue diseñada desde cero tomando como ejemplo el cómo están diseñados las zonas en un juego de plataformas, la idea es que sea difícil encontrar el camino para así obligarte a explorar la zona. Varias ideas fueron sacadas al analizar un proyecto de unity que te permite ver cómo está montado el escenario de Hollow Knight [9].

Imagen 9: Mapa de la Capital Vampiros

El reto a la hora de realizar las zonas era que tuvieran sentido con la historia y que pudiéramos realizarlas con los assets disponibles que encontramos en diferentes sitios, también debíamos tener en cuenta que no debíamos gastar mucho tiempo en el mapa.

Las diferentes zonas se realizaron con un propósito las zonas del castillo intentan mostrar las interacciones con elementos del mapa.

En el castillo se muestra el cómo funciona el sistema de diálogos, el sistema de plataformas, la introducción a las trampas que puedes encontrarte y el manual del aventurero que se da.

La zona de la taberna (ver Imagen 10) está pensada para poder moverte entre diferentes zonas con mayor facilidad y contarte partes de la historia mediante los NPCs que están en ella. La taberna se pensó como un lugar de descanso que conectara los diferentes puntos del mapa de forma que se tuviera una forma rápida de ir de una zona del mapa a otra sin tener que hacer un recorrido demasiado largo. La música de la taberna es una música tranquila que concuerde con un ambiente calmado. Al ser una taberna se decoró de acorde a una con mesas platos bebidas adicionalmente se añadieron decoraciones de fondo como cuadros y estanterías, después se añadieron NPCs para interactuar con ellos lo que dio vida a la taberna.

Imagen 10: Zona de la taberna

Personaje

Aventurero del rango más bajo que siempre ha evitado pelear, trabaja a tiempo parcial como mensajero para ganarse la vida. Está equipado con el equipo básico que se le ha proporcionado.

El diseño del aventurero se puede apreciar en la imagen 11.

Este personaje puede realizar acciones como rodar, saltar atacar, protegerse. Más adelante se muestran todas las acciones que puede realizar el personaje. La intención con el personaje es que los movimientos sean lo más fluido posible, debe cumplir con varios requisitos para que se cumpla.

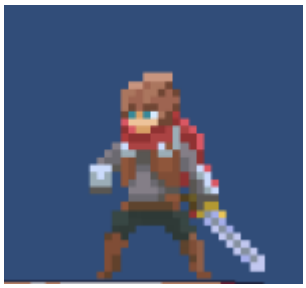

Imagen 11: Modelo del protagonista

Las acciones que puede realizar el personaje se pueden ver en la imagen 12 de entre ellas destacar las acciones que más se realizan como: correr, saltar, atacar, rodar, recibir daño, las acciones que puede realizar el personaje son bastantes estas se encuentran en estados como se explica en el punto 6.1. Al tener que realizar una variedad de acciones diferentes, se tiene que idear un sistema que haga posible esto, por ello se usa un árbol de estados que se verá más adelante.

Imagen 12: Animaciones del Personaje

Enemigos

Por las zonas pueden aparecer diferentes tipos de enemigos, los enemigos tienen diferentes tipos de estadísticas dependiendo de que enemigo sea. Hay 5 enemigos normales, 1 enemigo pacífico y 2 jefes.

Durante el desarrollo se probaron varios enemigos diferentes, se eligieron algunos y otros se descartaron. Al final los que quedaron se distribuyen por el mapa dependiendo del tipo de enemigo los enemigos se pueden ver en la imagen 13, cada uno tiene estadísticas ajustadas a su nivel de dificultad las estadísticas de cada enemigo se pueden ver en la tabla 3. Los más difícil de programar los enemigos es que su movimiento sea fluido y a la vez posible de esquivar sus ataques, al no disponer de mucho tiempo para crear una matriz de posibilidades se opto por programar a los enemigos por proximidad al jugador, de forma que fueran a por el jugador dependiendo de la distancia entre ellos.

Enemigo	Características				
	Daño	Vida	DefensaFisica	DefensaMagica	Experiencia
Demonio	30	100	70	40	20
Esqueleto	5	1	50	50	1
NightW	30	100	20	100	20
Arbol	24	100	54	20	20
Slime	7	100	30	40	20

Tabla 3: Estadísticas Enemigos

Imagen 13: Enemigos del juego

Los jefes que se ven en la imagen 14, son los que se están programados para el videojuego se estaba preparando un tercer jefe, pero no se terminó de programarlo, aunque en el proyecto se puede encontrar el jefe inacabado en la escena de Jefe3. Los jefes tienen estadísticas dependiendo de las habilidades que realizan, estas estadísticas se pueden ver en la tabla 3. Durante el desarrollo e implementación de los jefes se pensó en qué manera deben afectar en el juego, no tiene que ser difícil el pasar estos enemigos, pero al mismo tiempo no debe de ser del nivel de un enemigo normal, por ello se decidió el incorporarles habilidades especiales a cada uno, y a diferencia de los enemigos normales hacer que tengan una probabilidad de hacer su habilidad o atacar, la habilidad que tienen define su tipo de lucha, por ello se ajustaron sus estadísticas dependiendo de la habilidad, Senob tiene la habilidad de invocar esqueletos por ello su pelea se enfoca más en si el jugador tiene daño en área para poder limpiar los esqueletos antes de que se acumulen muchos y llegar al jefe, Uyako tiene la habilidad de generar portales que hacen daño al jugador por ello su combate se base más en el 1 contra 1 y del cuando acercarse a pegar.

Jefes	Características				
	Daño	Vida	DefensaFisica	DefensaMagica	Experiencia
Senob	15	150	125	60	250
Uyako	20	250	70	100	250

Tabla 4: Estadísticas jefes

Imagen 14: Jefes del Juego

Entorno

En esta parte vamos a mostrar los diseños de gran parte de decorados interactivos del mapa.

Cofres

Son en los que generalmente obtienes el equipamiento en el juego, estos cofres están distribuidos por diferentes zonas del mapa y dependiendo del cofre se consigue un buen equipamiento o un equipamiento básico, los cuatro tipos de cofres se pueden ver en la imagen 15.

Los cofres surgieron al implementar mucho equipamiento y armas, al principio se considera el que los enemigos suelten los objetos, pero al ser tantos se debía de implementar un sistema que permita la obtención de objetos.

Imagen 15: Cofres

Torres

Las torres son elementos que dotan al mapa de más vida hay tres tipos de torres que hacen las funciones de llevarte a la taberna, guardar tu punto de aparición, curarte (ver Imagen 16).

Las torres surgieron cada una en una etapa diferente del desarrollo, la primera fue la del pilarTP desde el principio del proyecto la idea de crear una taberna dimensional fue propuesta, el cómo conectar la taberna se decidió por medio de un teletransporte que se situarían en puntos importantes del mapa. Al principio solo con entran en contacto con el pilar eras teletransportado, pero se modificó para que tuvieras que interactuar con él. La siguiente Torre que surgió fue la Torre de Check, esta torre surgió ante el problema de que sucede cuando mueres, se decidió crear un Objeto que te permitiera guardar una ubicación para después volver a ella en caso de morir. La última torre es la de curación, esta torre se implementó para ser utilizadas en el comienzo de diversas zonas para regenerar la vida del jugador.

Imagen 16: Las tres torres

Elevador

El elevador (ver Imagen 17) es un elemento para conectar ciertas zonas del mapa para hacer la distancia más corta y no sobre usar las plataformas.

El elevador viene de un proyecto de Desarrollo de Videojuegos2D en ese momento se necesitaba un elemento que conectara dos zonas por ello se programó un ascensor para poder solucionar este problema, al principio dio unos problemas porque la palanca no subía correctamente, pero al final se logró solucionar modificando la jerarquía del objeto.

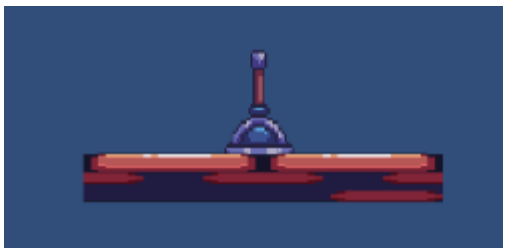

Imagen 17: Elevador

Trampas

Las trampas son elementos que están distribuidas por el mapa, estas trampas se pueden ver en la imagen 18. Las trampas tienen como objetivo el obstaculizar el camino del jugador.

Cuando se planteó la ida del proyecto se definieron diferentes funcionalidades, sistemas y elementos que se deseaban implementar uno de los elementos que se planeo fue el de las trampas, no se desarrollaron hasta que el proyecto paso la mitad del desarrollo ya que hay otros elementos de mayor importancia. Durante su desarrollo se eligieron diferentes trampas entre una selección que se encontraron durante el periodo de investigación del proyecto, de esta selección se programaron algunas que tendrían concordancia con alguna zona del mapa.

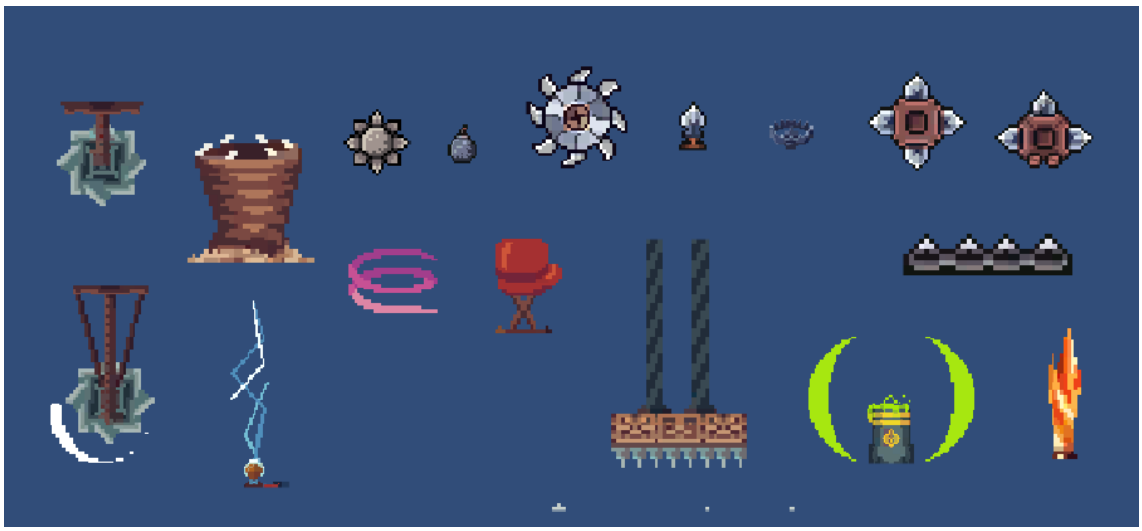

Imagen 18: Trampas

Ítems

En esta parte se mostrarán los diferentes objetos que existen en el juego.

Los ítems entran dentro del sistema de inventario, el sistema de inventario es uno que es bastante complejo tanto en programación como en planificación. Para poder hacer el sistema de inventario se investigó mucho tiempo, se analizaron diferentes formas en las que se programan normalmente los inventarios en 2D y se valoró que ventajas y desventajas tenía cada uno, después de tener las ventajas y desventajas se escribieron los requisitos que debía de tener nuestro sistema de inventario, cuando se tenía todos los factores se decidió programar un inventario con características de otros investigados pero que fuera un inventario desde 0 que se ajustara a nuestras necesidades y proyecto, el resultado fue un sistema de inventario simple con lo necesario para cubrir las necesidades del proyecto.

Hay una gran variedad de objetos en el juego, esta gran variedad se debe a la idea de que sea difícil hacer varias partidas iguales, por lo que el abanico de opciones que puedes obtener de objetos es amplio para cumplirlo.

Accesorios

Los accesorios (ver Imagen 19) son ítems que se pueden obtener de diversos cofres, estos accesorios dan pocas estadísticas.

Imagen 19: Accesorios del juego

Armaduras

Las armaduras (ver Imagen 20) son lo que más estadísticas proporciona en el videojuego, hasta un máximo de 3 efectos siendo la armadura del héroe sabio la que más estadísticas proporciona.

Imagen 20: Armaduras del juego

Espadas

Las espadas (ver Imagen 21) tienen una gran importancia en la jugabilidad ya que cada una aplica un efecto diferente con el ataque especial.

Imagen 21: Espadas del juego

Gemas

Las gemas (ver Imagen 22) son objetos que son dejados por los jefes al ser derrotados.

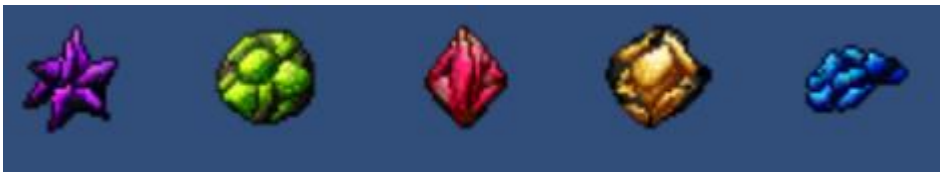

Imagen 22: Gemas del juego

Pociones

Las pociones (ver Imagen 23) son una parte importante de la jugabilidad ya que es la única forma de curarse si no se tiene cerca una de las torres los efectos de las pociones son las siguientes:

- Pociones rojas: Las pociones rojas curan un valor de vida predeterminado dependiendo de su nivel, el valor de curación es el siguiente: 10, 20, 30, 40, 50.
- Pociones azules: Las pociones azules regeneran un valor de mana predeterminado dependiendo de su nivel, el valor de regeneración es el siguiente: 10, 20, 30, 40, 50.
- Pociones amarillas: Las pociones amarillas regeneran un porcentaje de la vida máxima, este porcentaje depende de su nivel siendo el siguiente: 10%, 20%, 30%, 40% 50%.
- Pociones verdes: Las pociones verdes regeneran un porcentaje del mana máximo, este porcentaje depende de su nivel siendo el siguiente: 10%, 20%, 30%, 40% 50%.

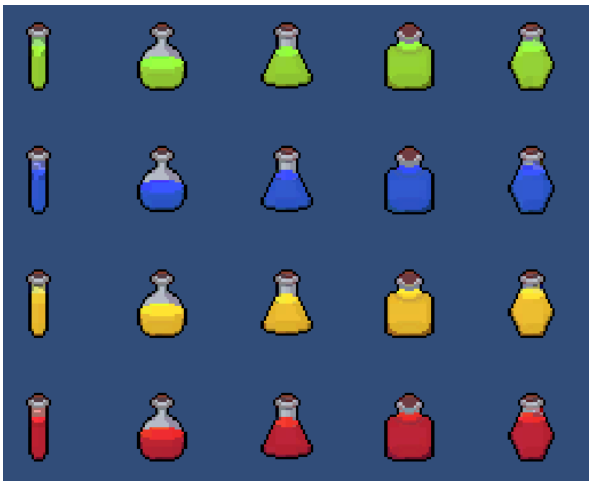

Imagen 23: Pociones del Juego

Manual del aventurero

El manual del aventurero (ver Imagen 24) es un elemento que ayuda al jugador a conocer los controles del juego, de esta manera se pueden consultar los controles en cualquier momento. El manual del aventurero es recibido después de terminar el tutorial en el patio de armas.

Imagen 24: Manual del Aventurero

5.5 Controles

En este apartado se muestran los controles que se usan en el juego, estos controles se pueden ver dentro del juego en el manual del aventurero.

- Moverse a la izquierda: Tecla A
- Moverse a la derecha: Tecla D
- Saltar: Tecla Espacio
- Enfundar/Desenfundar Arma: Tecla E
- Ataque: Click Izquierdo
- Ataque Especial: (En el suelo) Tecla Q
- Ataque Descendente: (En el aire) Tecla Q
- Escudo: Click Derecho
- Interactuar con NPCs: Tecla F
- Avanzar Diálogos: Tecla F
- Abrir/Cerrar Inventario: Tecla I
- Ver descripción: (En el inventario) Click derecho sobre el objeto
- Equipar/Usar objeto: (En el inventario) Click izquierdo sobre el objeto
- Ocultar/Mostrar Inventario: (En el inventario) Presionar Botón I
- Ocultar/Mostrar Subir Estadísticas: (En el inventario) Presionar Botón +
- Cerrar Menú Portales: Tecla P
- Cerrar Manual: Tecla I

5.6 Interfaz

La interfaz está diseñada para PC intenta tener lo mínimo en la pantalla. En esta sección se aprecia la interfaz del juego (ver Imagen 25, 26, 27) y se explica cada elemento de esta.

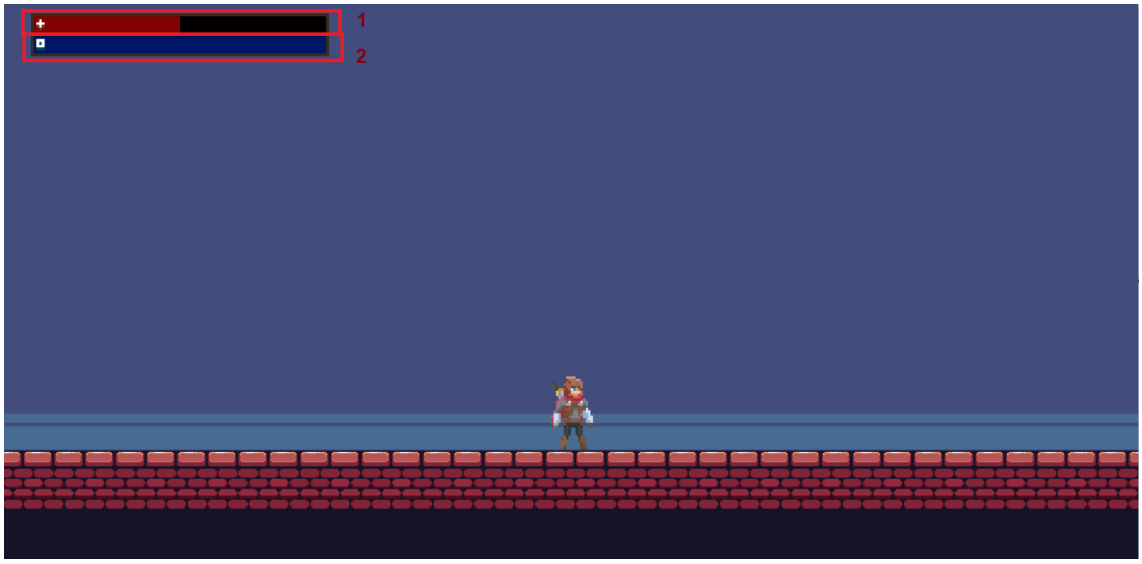

Imagen 25: Interfaz del juego

Barra de vida (1)

La barra de vida es el indicador de cuando daño puede aguantar el personaje.

Barra de mana (2)

La barra de mana muestra el mana actual que se tiene, se gasta conforme se utilice la habilidad especial.

Imagen 26: Inventario del Jugador

Inventario (3)

El inventario es donde se almacenan los objetos que has recogido

Menú Estadísticas (4)

El menú de estadísticas recoge todo lo relacionado con las estadísticas del personaje.

Barra de Experiencia (5)

La barra de experiencia nos muestra cuanto nos falta para subir al siguiente nivel.

Estadísticas Actuales del jugador (6)

Las estadísticas del jugador nos muestran los parámetros actuales de cada una.

Slot de Inventario (7)

El slot es lo que compone el inventario en cada uno se puede almacenar un tipo de objeto.

Inventario Equipamiento (8)

El inventario de equipamiento se encarga de mostrar los ítems equipados actualmente.

Botón Ocultar inventario (9)

El botón es una opción que se da por el inventario que está a la izquierda se desea ocultar

Botón Ventana Puntos (10)

El botón permite acceder a la ventada de gastos de puntos obtenidos por subir de nivel

Imagen 27: Menú Puntos

Menú Puntos Estadísticas (11)

El menú puntos estadísticas acoge todo lo referente a los gastos de puntos obtenidos por la subida de nivel en las diferentes estadísticas.

Botón Restar punto (12)

Si una estadística tiene puntos en ella se podrá quitar esos puntos gastados anterior mente en ella para utilizarlos en otra estadística, este botón solo aparece visible si la estadística asociada a él tiene puntos en gastados en ella.

Imagen Estadística (13)

La imagen hace referencia a la estadística que se verá afectada al sumar o quitar puntos

Botón Sumar punto (14)

El botón de sumar puntos solo aparece cuando se disponen de puntos de habilidad para gastar.

6 Implementación del videojuego

En este punto se muestran los aspectos importantes de la realización del videojuego, esto conlleva el personaje principal, enemigos, inventario, sistema de sonido, diversos sistemas y objetos utilizados.

Hemos optado en este apartado en dividir las partes por secciones en las que se muestren los componentes que lleva cada parte.

6.1 Personaje

El personaje es el protagonista de nuestro videojuego, mediante este mismo el usuario puede interactuar con el mundo. A continuación, se explican los componentes del personaje, su sistema de animaciones y los scripts asociados al personaje.

Componentes

El personaje lleva varios componentes (ver Imagen 28), cada uno tiene una propia finalidad por ello se explican brevemente cada componente a continuación:

- *Transform:* Proporciona una posición rotación y una escala en los ejes x, y, z. Este componente se modifica en ciertas circunstancias que se explican más adelante.
- *Sprite Renderer:* Proporciona una imagen del personaje mediante la cual se puede apreciar donde está el personaje.
- *Animator:* Proporciona las animaciones que realiza el personaje, este componente se explica en más profundidad más adelante.
- *Box Collider 2D:* Proporciona una hitbox al personaje mediante la cual puede interactuar con otros elementos del entorno.
- *Rigidbody 2D:* Proporciona gravedad al personaje.

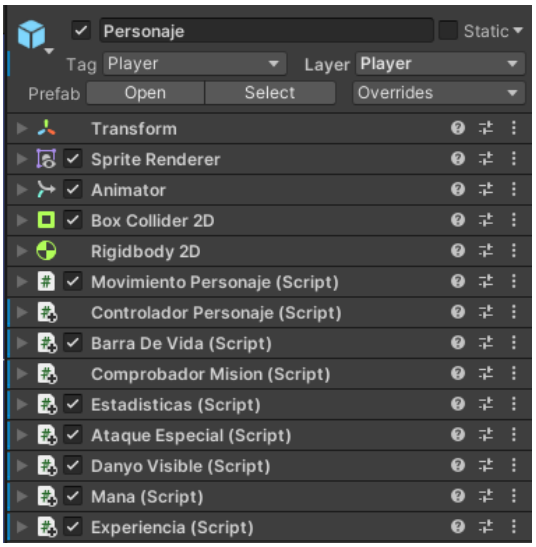

Imagen 28: Componentes del personaje

Animación

La animación se realiza por el componente *animator* el cual se puede ver en la imagen anterior. El animator recibe un parámetro que es el controlador de animaciones en este controlador de animaciones alberga las animaciones que realiza el personaje, su funcionalidad está basada en una máquina de estados.

Las animaciones del personaje se distribuyen en estados (ver Imagen 29), estos estados se relacionan entre sí por transiciones. Una transición puede contener parámetros que establezcan una condición para pasar a por ellos a los estados, estos parámetros pueden ser de tipo Int, float, bool o trigger.

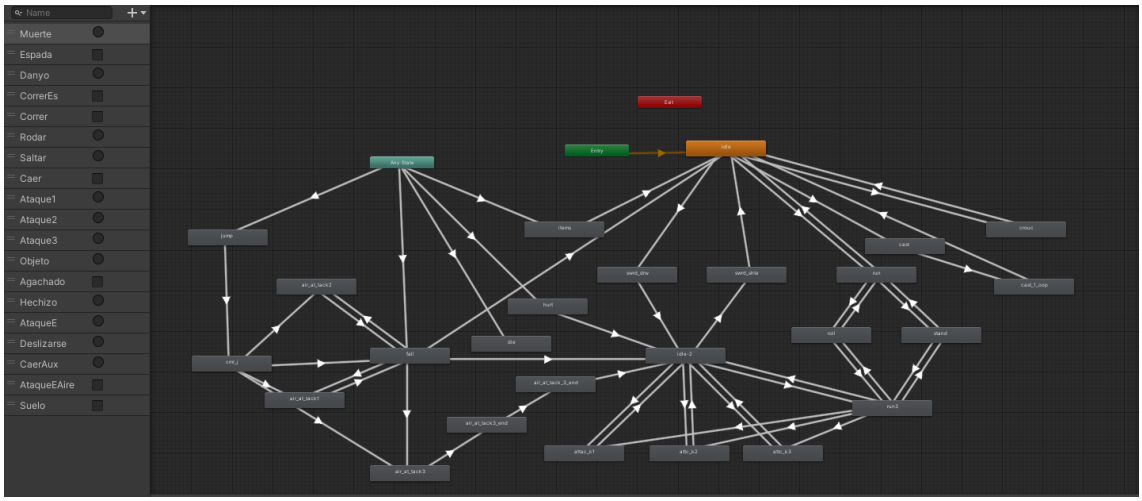

Imagen 29: Máquina de estados Controlador del personaje

El controlador del personaje está compuesto por 25 estados (sin contar los estados Entry, Any State y Exit) de los cuales se usan 20 de ellos, 5 de estos estados actualmente no se usan porque al final no se implementó la funcionalidad para los que estaban pensados o se cambió por otra idea diferente.

Los estados que podemos observar en el controlador del personaje contienen cada uno una animación, estos estados son los siguientes:

- *Entry:* Es el estado base de la máquina de estados cuando se inicia la máquina de estados siempre empieza en este estado.
- *Any State:* Es un estado de serie de la máquina de estados desde él se puede ir a los estados deseados desde cualquier estado siempre y cuando se cumplan las premisas asociadas a las transiciones.
- *Exit:* Un estado de serie de la máquina de estados que permite la salida de las animaciones.
- *Idle:* Es uno de los dos estados base de nuestro personaje, cuando no se tiene la espada sacada y no se realiza ninguna acción se permanece en este estado.
- *Run:* Este estado contiene la animación de correr sin la espada equipada.
- *Roll:* En este estado se realiza la acción de rodar del personaje.

- *run3:* Este estado contiene la animación de correr con la espada equipada.
- *swrd_drw:* Este estado contiene la animación de sacar la espada.
- *swrd_shite:* Contiene la animación de guardar la espada.
- *idle-2:* Estado base cuando se tiene la espada equipada.
- *attack_k1:* Contiene la animación de ataque normal 1 cuando se está en el suelo.
- *attack_k2:* Contiene la animación de ataque normal 2 cuando se está en el suelo.
- *attack_k3:* Contiene la animación de ataque especial cuando se está en el suelo.
- *jump:* Contiene la animación de inicio de salto del personaje.
- *cmr_j:* Contiene la animación de continuación del salto del personaje.
- *air_at_tack1:* Contiene la animación de ataque normal 1 cuando se está en el aire.
- *air_at_tack2:* Contiene la animación de ataque normal 2 cuando se está en el aire.
- *air_at_tack3:* Contiene el inicio animación de ataque especial cuando se está en el aire.
- *air_at_tack3_end:* Contiene la continuación del ataque especial en el aire.
- *air_at_tack_3_end:* Contiene el final del ataque especial en el aire.
- *fall:* Contiene la animación de caída del personaje.
- *die:* Contiene la animación de muerte del personaje.
- *hurt:* Contiene la animación de recibir daño del personaje.

Los estados que actualmente no son usados son los siguientes:

- *ítems:* Contiene la animación de usar ítem.
- *stand:* Contiene la animación de deslizarse.
- *cast:* Contiene la animación de cargar un hechizo.
- *cast_1_oop:* Contiene la animación de lanzar un hechizo.
- *Crouc:* Contiene la animación de caminar agachado.

Anteriormente hemos mencionado que los controladores tienen parámetros con los que se controlan las transiciones entre estados. Los parámetros utilizados en este controlador son los siguientes:

- *Muerte:* Parámetro de tipo *trigger* que acciona la acción de la animación de muerte
- *Espada:* Parámetro de tipo bool que controla si el personaje lleva equipada la espada.
- *Danyo:* Parámetro de tipo *trigger* que se activa al recibir daño.
- *CorrerEs:* Parámetro de tipo *bool* que controla si el personaje se está moviendo con la espada.
- *Correr:* Parámetro de tipo *bool* que controla si el personaje se mueve sin la espada.
- *Rodar:* Parámetro de tipo *trigger* que se activa cuando se realiza la acción de rodar.
- *Saltar:* Parámetro de tipo *trigger* que se activa cuando se salta.
- *Caer:* Parámetro de tipo *bool* que controla si el jugador está cayendo hacia abajo.
- *Ataque1:* Parámetro de tipo *trigger* que se activa al realizar el primer ataque.
- *Ataque2:* Parámetro de tipo *trigger* que se activa al realizar el segundo ataque.
- *Ataque3:* Parámetro de tipo *trigger* que se activa al realizar el ataque especial.
- *Objeto:* Parámetro de tipo *trigger* que se activa al usar un objeto.
- *Agachado:* Parámetro de tipo *bool* que controla si el jugador esta agachado.
- *Hechizo:* Parámetro de tipo *trigger* que se activa al usar un hechizo.

- *AtaqueE:* Parámetro de tipo *trigger* que se activa al realizar el ataque especial.
- *Deslizarse:* Parámetro de tipo *trigger* que se activa al deslizarse.
- *CaerAux:* Parámetro de tipo *trigger* que se usa para solucionar un error de animaciones que ocurría al caer.
- *AtaqueEaire:* Parámetro de tipo *bool* que controla si el jugador está en el aire.
- *Suelo:* Parámetro de tipo *bool* que controla si el jugador está en el suelo.

Estos parámetros mencionados son controlados y modificados mediante código, de forma que se realice la acción desea en el momento deseado.

Scripts

Todas las acciones que el personaje puede realizar se controlan mediante scripts que son ficheros de código escritos en C#

Como se puede observar en una de las imágenes anteriores nuestro personaje pose varios scripts que se explicaran ahora, pero antes añadir que los estados que se encuentran en los controladores de animaciones también pueden llevar *scripts* también hablaremos de estos, pero explicaremos los que están asociados al personaje en primer lugar.

MovimientoPersonaje

Este *script* es el encargado de controlar las acciones del personaje tiene varias funciones entre ellas algunas a destacar las siguientes: función para comprobar si el personaje está en el suelo o está en el aire, una función que se encarga de activar el escudo del personaje dependiendo de si está en enfriamiento o no, el tiempo de enfriamiento se puede cambiar externamente ya que es una variable publica, función para controlar el ataque del personaje dependiendo de que tipo de ataque sea y cuantos ataques normales se hayan realizado, la función encargada del movimiento del personaje, la función para poder rodar, la función para girar el personaje. Estas son algunas de las funciones que componen este *script* muchas de estas funciones hacen cambiar el estado en el árbol de estados para así que las animaciones del personaje hagan las acciones acordes al movimiento del personaje.

ControladorPersonaje

Este *script* se mediante la ayuda del patrón Singleton se encarga de asegurarse que no haya otro objeto de tipo Personaje en las escenas. También almacenar el último punto de guardado por el que paso el jugador y la última salida de escena por la que paso.

BarraDeVida

Este *script* se encarga de controlar la vida del jugador, la vida máxima llega del *script* Estadísticas, este *script* tiene funciones para los cálculos de armadura y resistencia mágica que se encargan de determinar el daño recibido dependiendo del daño del enemigo y de las estadísticas del personaje. La vida del personaje se refleja en la barra de vida del HUD.

ComprobadorMision

Este script se encarga de almacenar la información de misiones realizadas para así que no se repitan ciertos eventos.

Estadísticas

Este script se explica en el apartado 6.3.

AtaqueEspecial

Este script se encarga de seleccionar el ataque especial que se debe realizar dependiendo del arma equipada en el momento.

DanyoVisible

Al recibir daño el jugador el daño recibido es pasado a este script el cual crea un *GameObject* que muestra el daño recibido.

Mana

Se encarga de determinar cuanto mana le queda al jugador, el mana solo se puede regenerar mediante pociones, el mana máximo disponible se saca de las estadísticas del jugador.

Experiencia

El script es el encargado de controlar la experiencia del jugador, calcula la experiencia necesaria para subir nivel mediante la formula:

experiencia necesaria = exp_nivel + exp_varnivel *(act_nivel -1)

- exp_nivel: variable de tipo float que contiene el valor de la experiencia base en cada nivel, su valor está establecido a 125.
- exp_varnivel: variable de tipo float que controla la variación de experiencia de cada nivel, su valor está establecido a 35.
- act_nivel: el nivel actual del jugador.

De esta manera la experiencia necesaria para cada nivel varía dependiendo de en qué nivel estes por ejemplo a nivel 1 se requiere 125, a nivel 2 se requiere 160 y va sumiendo de forma constante.

Cuando se recibe experiencia se suma a una variable, el *script* actualiza la imagen de la barra de experiencia de acorde al porcentaje de experiencia obtenida frente a la necesaria para subir nivel, cuando la experiencia ganada es igual o superior a la experiencia necesaria se resta la experiencia necesaria para subir nivel a la obtenida y se sube un nivel al jugador, con esta subida de nivel el jugador gana un punto en la parte de estadísticas que se explica más adelante.

Scripts en las animaciones

Como se ha dicho antes algunos estados contienen scripts, esto es debido a que se activan cuando entra al estado, durante su estancia en el estado, al salir del estado.

AtacandoBool

Este *script* se encuentra en los estados *air_at_tank2, air_at_tank1, attac_k1, attack_k2, attack_k3*. Se encarga de cuando empieza la animación llamar al script *MovimientoPersonaje* para establecer la variable *e*_atacando a true, cuando se sale de la animación se cambia el valor de *e_atacando* a false.

AtacandoBoolEspecial

Este *script* se encuentra en el estado *air_at_tank3*. Cuando empieza la animación la variable *e_atancando* se cambia a true y el valor *bool AtaqueEAire* del árbol de estados se pone a true.

AtacandoBoolEspecialSalida

Este *script* se encuentra en el estado *air_at_tank_3_end*. Cuando termina la animación la variable *e_atancando* se cambia a false y el valor *bool AtaqueEAire* del árbol de estados se pone a false.

DesactivarColisiones

Este *script* se encuentra en el estado *roll*. Cuando empieza la animación desactiva la detección de colisiones entre las colisiones 7 y 8 que corresponden al jugador y a los enemigos, cuando sale del estado vuelve a activar las colisiones entre 7 y 8.

Objetos Hijo

El GameObject del personaje tiene objetos hijos (ver Imagen 30) necesarios para diversas acciones los objetos son los siguientes:

El objeto espada contiene los *colliders* correspondientes al ataque del jugador, estos *colliders* cambian de tamaño durante la animación para ajustarse a la *hitbox* del ataque

El objeto comprobadores contiene un objeto con un *collider* que sirve para detectar si el jugador está pisando el juego o está en el aire.

El objeto Efectos contiene las animaciones del escudo del jugador.

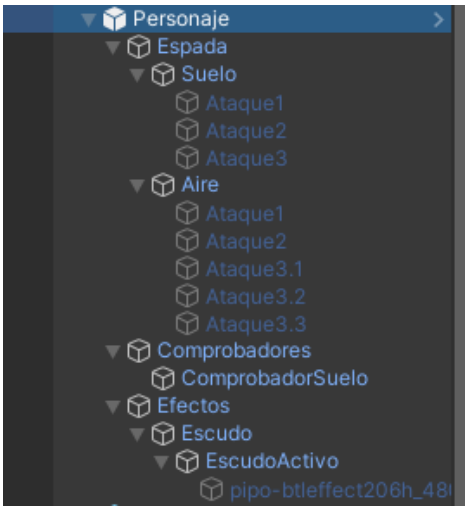

Imagen 30: Estructura de GameObject del Personaje

6.2 Enemigos

Los enemigos son el obstáculo principal del juego que se interponen en el camino, hay tres tipos de enemigos que se clasifican en pacíficos, ofensivos.

Componentes Comunes

Los enemigos al igual que el personaje tienen componente (ver Imagen 31 y 32), estos componentes son los mismos que se han explicado antes en el caso del personaje, es decir, *Tranform, Sprite Renderer, Animator, Box Collider2D, Rigibody2D.*

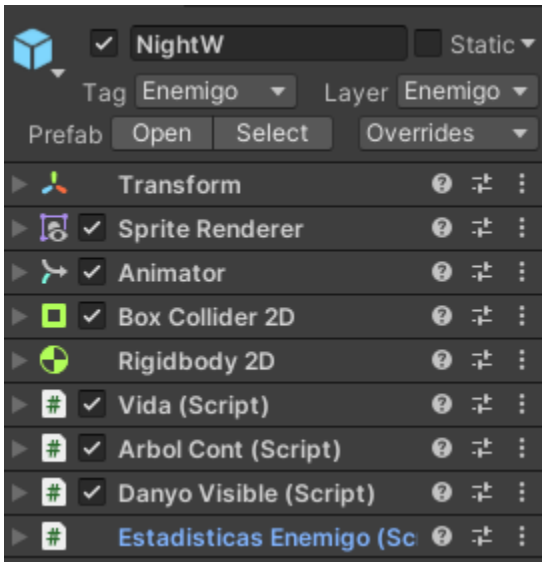

Imagen 31 Componentes de un enemigo normal

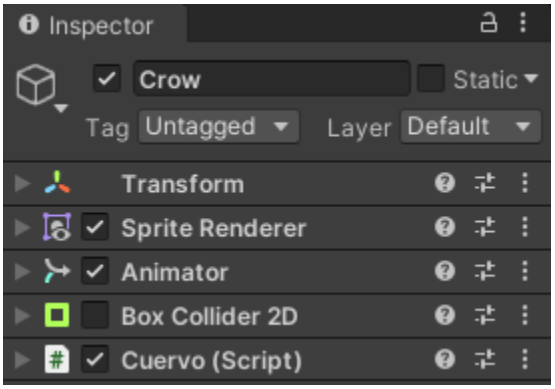

Imagen 32: Componentes de un enemigo pacífico

Los componentes referentes a los enemigos ofensivos solo varían los scripts asociados dependiendo si son enemigos normales o jefes.

6.2.1 Enemigos Pacíficos

Los enemigos pacíficos más que enemigos en si serían como decoración, su propósito es llenar un poco el entorno para que haya más vida, solo se encuentra en esta categoría un enemigo pacífico que fue creado más que nada para experimentar el funcionamiento de los scripts en los estados de la máquina de estados, al final se las pruebas y experimentos el cuervo se terminó y se dejó en el juego como un enemigo pacífico.

Animación

La máquina de estados que se ve en la imagen 33 pertenece al cuervo.

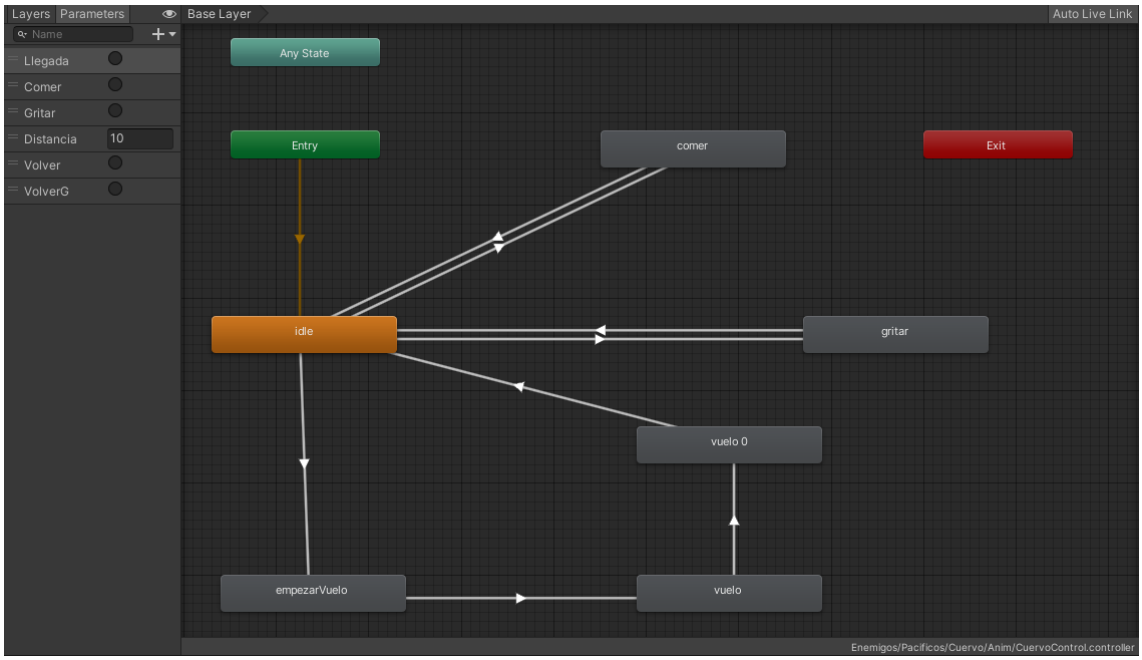

Imagen 33: Maquina de estados del cuervo

A continuación, se explican los diferentes estados de la máquina de estados asociada a al cuervo:

- *Idle:* Es el estado por defecto del cuervo, se encuentra en este estado cuando no está realizando ninguna acción.
- *Comer:* Es el estado que contiene la animación de comer del cuervo.
- *Gritar:* Es el estado que ejecuta la acción de gritar del cuervo.
- *EmpezarVuelo:* Es el estado que almacena el comienzo de la animación de vuelo
- *Vuelo:* Es el estado que almacena la animación de Vuelo del Cuervo
- *Vuelo 0:* Es el estado que almacena la animación de Vuelo del cuervo, pero este se utiliza para volver a su posición.

Los parámetros del animador se utilizan para llegar a cada estado dependiendo de que acción se deba realizar en el código, los parámetros son los siguientes:

- *Llegada:* Parámetro de tipo *trigger* que se activa cuando el cuervo vuelve a su posición inicial.
- *Comer:* Parámetro de tipo *trigger* que se activa cuando se activa cuando el cuervo debe realizar una animación de stay.

- *Gritar:* Parámetro de tipo *trigger* que se activa cuando el cuervo debe realizar la acción de gritar.
- *Distancia:* Parámetro de tipo *float* que almacena el valor de la distancia entre el cuervo y el jugador.
- *Volver:* Parámetro de tipo *trigger* que se activa cuando el tiempo de seguimiento se a agotado se usa para volver a la posición inicial.

Scripts

Cuervo

El *script* tes el principal controlador del cuervo, en cada frame actualiza la distancia entre el jugador y el cuervo para que siempre sea la correcta. Al estar un tiempo sin hacer nada se activa el *trigger* de Gritar para que realice una acción distinta al idle.

Scripts en Estados

CuervoGrito

Al entrar en el estado que lleva este *script* se genera un GameObject que muestra un texto en movimiento, este texto es la palabra “CAW”.

CuervoVuelo

Al entrar en el estado que lleva este script el cuervo empieza a volar hacia el jugador y se queda volando alrededor a cierta distancia establecida. Pasados unos segundos se pasa al siguiente estado.

CuervoVuelta

Al entrar en el estado que lleva este script el cuervo regresara al punto inicial el que se encontraba, al comprobar que ha llegado se pasara al estado de idle.

6.2.2 Enemigos Ofensivos

Los enemigos ofensivos se clasifican en dos tipos que son los normales y los jefes, Los componentes de estos son casi los mismos, lo único que varía son los scripts que tienen.

Enemigos Normales

Los enemigos normales son lo que te puedes encontrar con más frecuencia, están situados en diferentes zonas del mapa para obstaculizar el avance del jugador

Animación

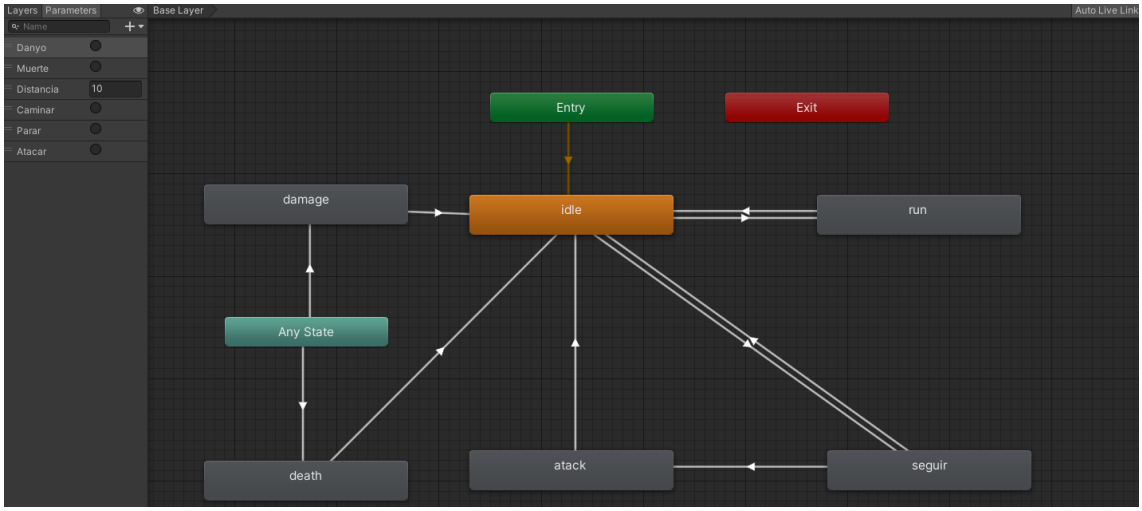

Imagen 34: Árbol de estados del Controlador de un enemigo normal

A continuación, se explican los diferentes estados de la máquina de estados (ver Imagen 34) asociada a los enemigos.

- *Idle:* Es el estado por defecto del enemigo, se usa cuando el personaje no se mueve ni ataca al jugador.
- *Mover:* Es el estado que se usa para cuando el enemigo está usando el patrón de movimiento de un punto a otro punto es decir que está en modo centinela.
- *Seguir:* Es el estado que se usa cuando detecta que está cerca del jugador, en este estado el enemigo se encuentra siguiendo al jugador.
- *Atacando:* Es el estado en el que el enemigo ejerce la animación de ataque hacia el jugador.
- *Damage:* Es el estado en el que el enemigo recibe daño se puede acceder en cualquier punto de la máquina de estados ya que está conectado a Any State.

- *Death:* Es el estado en que el enemigo a perdido toda su vida, al finalizar este estado el enemigo desaparece y el jugador recibe una recompensa.

Algo importante a destacar es que algunos estados tienen Scripts dentro de ellos, la razón de esto es que estos scripts se ejecutan durante la animación correspondiente por ello es más cómodo, estos scripts se explicaran más adelante.

Los parámetros del animador se utilizan para llegar a cada estado dependiendo de que acción se deba realizar en el código, los parámetros son los siguientes:

- *Danyo:* Parámetro de tipo *trigger* que se activa cuando el enemigo recibe daño.
- *Muerte:* Parámetro de tipo *trigger* que se activa cuando la vida del enemigo es igual o inferior a 0.
- *Atacar:* Parámetro de tipo *trigger* que se activa cuando la distancia entre el enemigo y el jugador es igual o inferior a la establecida.
- *Caminar:* Parámetro de tipo *trigger* que se activa cuando se determina que el enemigo ha de moverse hace un punto en modo centinela.
- *Parar:* Parámetro de tipo *trigger* que se activa cuando el enemigo debe de para de moverse.
- *Distancia:* Parámetro de tipo *float* que se guarda el valor de la distancia entre el enemigo y el jugador.

Scripts

Vida

El *Script* se encarga de controlar la vida que tiene el enemigo, al ser publica la vida máxima se le puede cambiar con facilidad dependiendo del enemigo, el *Script* tiene entre otras funciones los cálculos de armadura y resistencia magia que se encargan de calcular el daño total que se recibirá dependiendo del daño entrante y de las estadísticas de defensa almacenadas en el *Script* de Estadísticas. Una vez este el daño final que se recibe calculado se envía esa cantidad al Script de DanyoVisible para que se muestre el daño y se calcula la nueva vida del enemigo, esta vida se ve reflejada en la barra de vida que aparece encima del enemigo, en el caso de que la vida del enemigo sea inferior o igual a 0 se activara el Trigger de Muerte del Controlador mediante el SetTrigger().

ArbolCont

El *Script* tiene la finalidad de al iniciar guardar la posición inicial, crear una posición destino para el *script* movimientocenti, calcular la distancia del enemigo con el jugador y mandar ese valor al animator mediante un SetFloat(), también tiene diferentes métodos para girar el enemigo de dirección y resetear el movimiento del enemigo este reseteo es una medida de seguridad por si el enemigo quedara atascado en algún momento.

DanyoVisible

El *Script* se encarga de crear un GameObject con el daño recibido para así poder dar información al jugador de la cantidad de daño que ha ocasionado al enemigo.

Estadísticas

El *Script* se encarga de almacenar las estadísticas dadas al enemigo de esta manera se pueden cambiar con facilidad sin tener que entrar al código. En concreto se almacenan las estadísticas de armadura. Daño y resistencia mágica.

Scripts en estados

Muerte

El *script hace* que cuando entre en el estado asociado a este primero comprueba si el enemigo es un jefe, en el caso de serlo encuentra un objeto en la escena con tag Salida. Este script tiene tres variables publicas: Experiencia en la que se le da el valor de experiencia que dará el enemigo al jugador al morir, Drops en forma de Array donde se ponen los objetos que dejara caer el enemigo al morir y la variable jefe de tipo bool en la que se marcara si es un jefe o no. Una vez que se esté saliendo de la animación de muerte el script comprobara si ha de crear algún Objeto (estos son los objetos que están en la variable Drops) los objetos que se crean aparecen en un radio aleatorio en una esfera de 1 de diámetro alrededor del enemigo, comprobara si es un jefe en caso de serlo activara la salida encontrada anteriormente, dará la experiencia al jugador y por último hará desaparecer el enemigo de la escena.

MovimientoCenti

Este *script* está asociado a un estado en el momento de que se entra al estado se sacan información del *script* ArbolCont esta información es el destino adonde se dirige el enemigo cuando no está el jugador cerca, el enemigo se mueve a una velocidad declarada anteriormente (generalmente es 2) una vez llegado a su destino el *script* actica un método en el *script* ArbolCont que espera unos segundos y después establece un nuevo destino al que moverse.

EjecutarSonido

Este *script* tiene una variable publica de tipo String en la que se escribe el sonido que debe de realizar en el momento de entrar al estado, cuando esto sucede se llama al *script* ActivadorSonidos, se le pasa al método ActivarSonido() el String que se tiene.

SeguirJugador

En este *script* se tiene como variables publicas una velocidad y una distancia de ataque cuando se entra en el estado asociado a este el enemigo comienza a moverse hacia el jugador a la velocidad estipulada una vez que se eta a la distancia de ataque se establece el Trigger Atacar para que se pase a la acción de atacar al jugador.

Jefes

Los jefes son enemigos más fuetes que los normales y que tienen una habilidad especial, hay dos jefes que funcionan de forma similar.

Animación

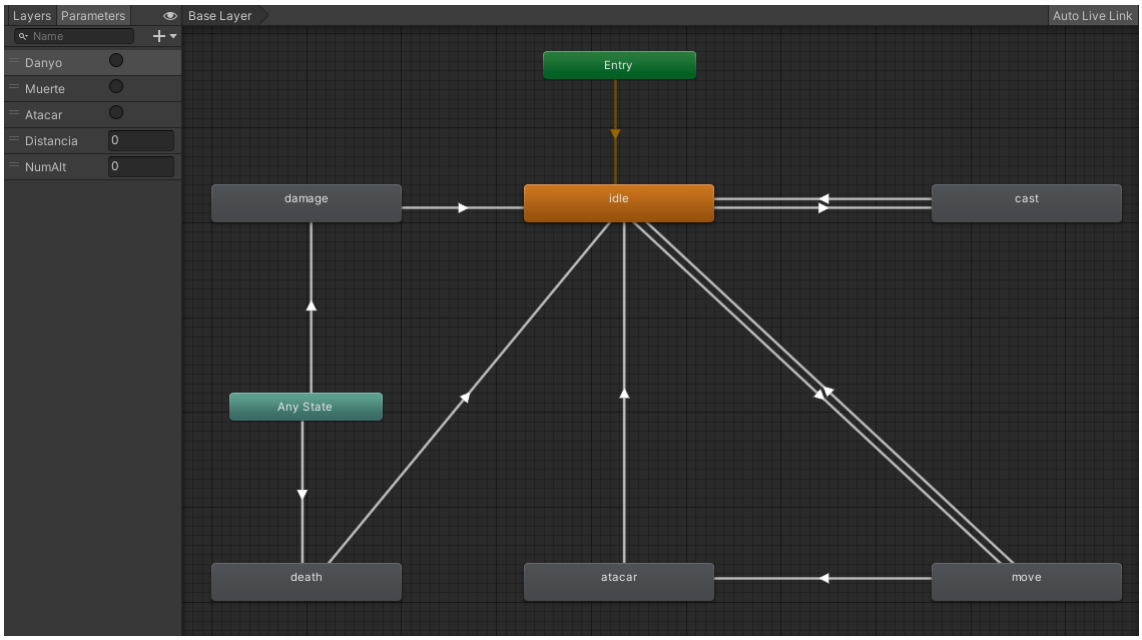

Imagen 35: Maquina de estados de los jefes

A continuación, se explican los diferentes estados de la máquina (ver Imagen 35) de estados asociada a los enemigos.

- *Idle:* Es el estado por defecto del enemigo, se usa cuando el personaje no se mueve ni ataca al jugador.
- *Cast:* Es el estado en el que el enemigo hace su habilidad especial.
- *Move:* Es el estado que se usa cuando detecta que está cerca del jugador, en este estado el enemigo se encuentra siguiendo al jugador.
- *Atacar:* Es el estado en el que el enemigo ejerce la animación de ataque hacia el jugador.
- *Damage:* Es el estado en el que el enemigo recibe daño se puede acceder en cualquier punto de la máquina de estados ya que está conectado a Any State.
- *Death:* Es el estado en que el enemigo a perdido toda su vida, al finalizar este estado el enemigo desaparece y el jugador recibe una recompensa.

Como en el caso de los enemigos normales en algunos estados se encuentran scripts, estos scripts se explicarán más adelante.

Los parámetros del animador se utilizan para llegar a cada estado dependiendo de que acción se deba realizar en el código, los parámetros son los siguientes:

- *Danyo:* Parámetro de tipo *trigger* que se activa cuando el enemigo recibe daño.
- *Muerte:* Parámetro de tipo *trigger* que se activa cuando la vida del enemigo es igual o inferior a 0.
- *Atacar:* Parámetro de tipo *trigger* que se activa cuando la distancia entre el enemigo y el jugador es igual o inferior a la establecida.
- *Distancia:* Parámetro de tipo *float* que se guarda el valor de la distancia entre el enemigo y el jugador.
- *NumAlt:* Parámetro de tipo *float* que guarda el valor asociado a la siguiente acción a realizar.

Scripts

Algunos *scripts* de los jefes funcionan de forma similar a los de los enemigos normales, también comparten algunos *scripts*.

JefeControl

Este *script* se encarga de determinar la distancia entre el enemigo y el jugador, cuando es determinada se actualiza la variable del animator Distancia con el valor correspondiente. También dispone de métodos para girar al enemigo para que mire hacia su objetivo y otro método que detecta si esta colisionando con el arma del jugador o con alguna habilidad.

VidaJefe

El *script* se encarga de controlar la vida del jefe funciona de la misma manera que el *script* de vida de un enemigo normal con la única diferencia de que la vida del jefe se ve en una barra de vida arriba en la interfaz y también muestra el valor de la vida en números de esta manera el jugador se puede hacer una idea de cuantos golpes o habilidades aguantara.

EstadisticasEnemigo

Mismo funcionamiento que el *script* explicado anteriormente de Estadísticas en los enemigos normales.

DanyoVisible

Mismo *script* explicado anteriormente en los enemigos normales

Scripts en estados

Muerte

Mismo *script* explicado anteriormente en los enemigos normales

CastHab

El script se encarga de invocar la habilidad del jefe en un radio determinado alrededor del jugador dependiendo del jefe que sea la habilidad invocada será diferente.

MovimientoJefe

El script se encarga del movimiento del jefe cuando detecta al jugador, este script siempre se moverá hacia el jugador y no ejecutará ningún patrón de movimiento.

Jefenumalt

El *script* se encuentra en el estado idle se encarga de generar un numero aleatorio entre unos valores determinados para así determinar qué acción realizara normalmente si el numero generado es 0 ira al estado que realiza habilidades y si es 1 ira al estado de movimiento.

EjecutarSonido

Cuando se entra en el estado asociado a este *script* se llama al método del script encargado de controlar los sonidos con el sonido que debe de ejecutarse. Este script se explica en más profundidad en el punto 6.7.

6.3 Estadísticas

Las estadísticas es una parte importante de nuestro juego ya que marcan la dificultad en pasar diversas zonas. Antes de empezar a desarrollar el proyecto se hizo una amplia investigación de posibles partes del proyecto una de estas investigaciones tenía como objetivo el pensar que estadísticas debería de tener nuestro sistema, al final las estadísticas que están en el sistema son defensa física, defensa mágica, daño físico, daño mágico, mana, vitalidad y suerte.

El sistema de Estadísticas cuenta con un *script* que controla todas las funcionalidades del sistema

Las estadísticas tienen una estadística base que nunca varia, una estadística variable que almacena el valor de la estadística obtenida a través de mejoras estas mejoras son los valores obtenidos por equipamiento y por gastos de puntos de estadística y una variable estadística total que almacena la suma de la estadística base y la estadística variable, esta estadística total es la que ve el jugador en la interfaz.

El sistema controla la información de la interfaz de estadísticas vista anteriormente en la imagen 26 con esto el jugador puede saber el valor de las estadísticas. También controla el gasto de puntos para ganar estadísticas como se ve en la imagen 27.

Al cambiar un objeto equipado en la zona de equipamiento se llama al *script* estadísticas pasando los bufos del objeto añadido o quitado, los dos métodos de añadir y quitar bufos funcionan de la misma manera que es recibe el objeto de tipo *ItemBuff* que contiene el tipo de estadística que modifica y su valor a modificar, una vez recibido el *ItemBuff* se consulta el tipo de estadística a modificar y se le suma el valor de la estadística en su correspondiente estadística variable en el caso que fuera quitar bufo se restaría lo sumado anteriormente en la estadística variable.

Los puntos que se utilizan en la parte de gasto de puntos son almacenados en este *script*, pero son recibidos del script de experiencia cuando se sube de nivel, se puede subir una estadística cuando se disponen de puntos disponibles, también se pueden quitar puntos utilizados en las estadísticas para cambiarlos a otra estadística, solo se pueden quitar puntos cuando se han gastado en la estadística anteriormente.

Otra parte por destacar del script es que cuenta con variables públicas (ver Imagen 36) para asociar los botones de los gastos de puntos, los textos donde se muestra los valores totales de las estadísticas, la interfaz de las estadísticas, la interfaz del inventario ya que desde este script se da la opción de ocultar el inventario y la interfaz de gasto de puntos para mostrarla u ocultarla.

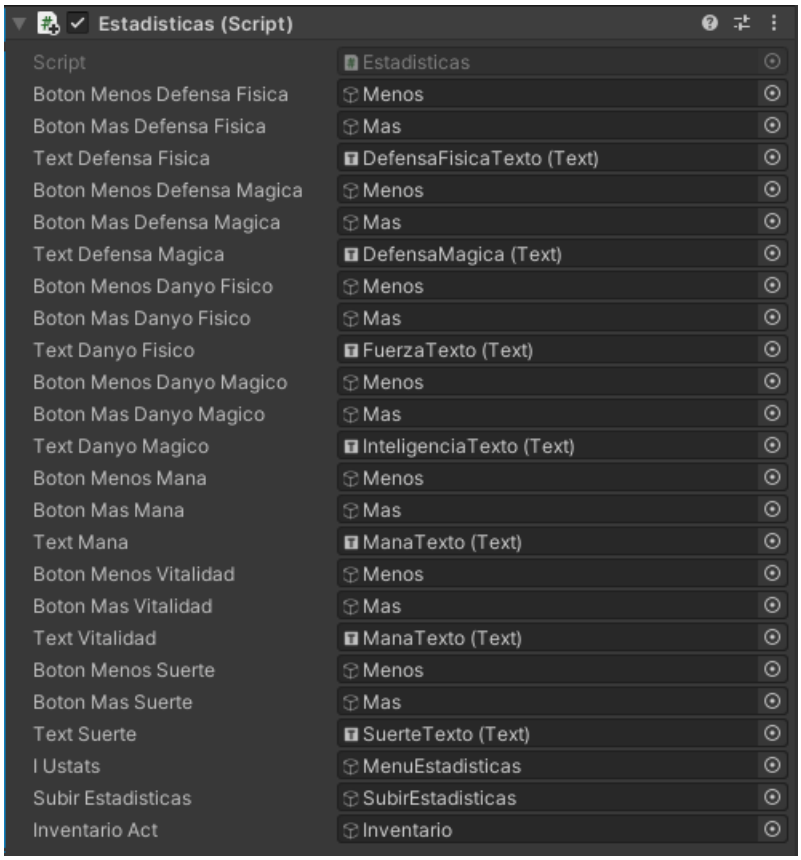

Imagen 36: Script Estadisticas

6.4 Inventario

Un inventario en un juego es un lugar donde poder almacenar los objetos/ítems que se obtienen con el paso del tiempo.

Nuestro inventario está dividido en dos, el primer inventario es donde se van almacenando los objetos al ser recogidos este tiene 30 slots, el segundo es el inventario de equipamiento en el que ve que equipamiento se lleva, este último inventario está compuesto por 11 slots cada uno puede almacenar un tipo de objeto.

6.4.1 Ítems

Para explicar en qué consisten los ítems primero se debe saber que es un *ScriptableObject*.

ScriptableObject

Es un contenedor de datos que se puede usar para guardar grandes cantidades de datos, independientemente de las instancias de clase. Redice el uso de la memoria del proyecto al evitar copias de valores.

Esto tiene muchas ventajas hace que sea más fácil el crear objetos determinados con mayor facilidad.

En nuestro caso se ha creado el script *ItemObject* para crear el *ScriptableObject.*

ItemObject

Consiste en un *script* que hace posible la creación de los ítems, al ser un *ScriptableObject* podemos asignarle un menú para la posterior creación de los objectos en nuestro caso la ruta es Assets/Inventory System/Items/item con ello podremos crear el ítem y asignarle los valores deseados.

Los valores del ItemObject son los siguientes:

- Sprite: Almacena la imagen que corresponde al ítem.
- Stackable: Una variable de tipo bool que establece si el ítem solo puede tener una unidad o las que quiera.
- Type: El tipo de ítem que es, se puede elegir de una lista que contiene los tipos: Comida, Pocion, Arma, Casco, Pechera, Pantalones, Zapatos, Anillos, Collar, Escudo, Patata, Gema.
- Descripción: Un script que almacena la descripción del ítem.
- Data: Es una clase que actúa como tipo Ítem que almacena la siguiente información:

- Name: Una variable String que almacena el nombre.
- Id: Una variable int que almacena el id del objeto, cada objeto tiene un id diferente reservando el id -1 para los objetos recién creados.
- Buffs: Un array que almacena los bufos por medio de una clase ItemBuff
  - Atribute: El tipo de atributo que es el bufo se elige mediante una lista que contiene las siguientes opciones: Fuerza, Inteligencia, DefensaFisica, DefensaMagica, Suerte, Mana, Vida.
  - Value: El valor del bufo que da, el valor se determina aleatoriamente dependiendo del mínimo y máximo.
  - Min: El valor mínimo que puede tener el bufo.
  - Max: El valor máximo que puede tener el bufo.

No hay un límite de bufos que puede tener un ítem, pero como hemos decidido que los ítems tendrían entre 0 o 3 bufos.

En la imagen 37 podemos ver un ejemplo de un *ItemObject* creado y completado.

ItemControlador

Es un script que contiene el *ItemObject* para así poder recogerlo y llevarlo al inventario, este script va asociado a un *GameObject* creando así los llamados ítems dentro del juego para poder interactuar con ellos, una vez que el ítem es recogido del suelo este es eliminado de la escena.

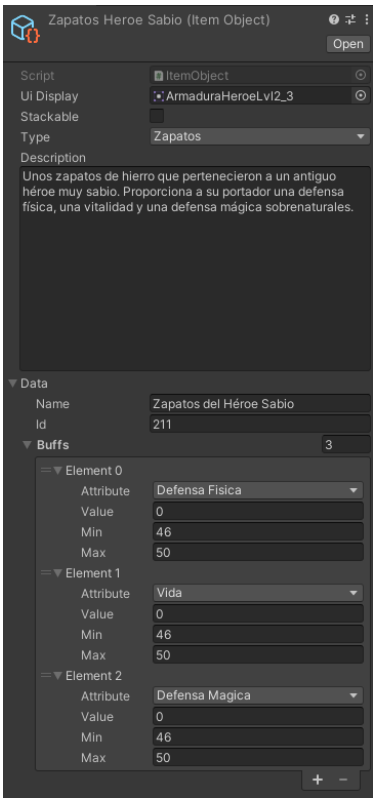

Imagen 37: ItemObject de un objeto con los valores ya determinados

6.4.2 Inventario

Primero hablaremos el Script más importante que se encarga de controlar el inventario:

Inventory

Este script se encarga de gestionar los espacios del inventario con arrays de lo métodos más importantes son:

- *SlotLibre():* Comprueba que el inventario tenga un espacio disponible de tener espacio devolverá el slot correspondiente, de no tener devolverá un null.
- *SlotInventerio(Item):* Comprueba si el objeto ya se encuentra en el inventario de ser así devolverá el slot donde se encuentra en caso no estar devolverá un null.
- *AnyadirItem(ItemObject):* Al recoger un ítem se llama a este método con el ítem deseado, este primero comprobara si es *Stackable* o no, de no serlo buscara un slot libre en el inventario y lo añadirá si es posible, de serlo primero buscara si ya está en el inventario de estar en el inventario sumara una unidad al slot de no estar intentara añadir lo en un nuevo slot.
- *SlotLibreEstadisticas(ItemType):* Comprueba si en el inventario de estadísticas se dispone de un espacio disponible para el tipo de ítem, de ser así devuelve del slot correspondiente no disponer de espacio devolverá un null.
- *AnyadirEquipo(ItemObject):* Comprueba si hay un slot disponible en el inventario de estadísticas, de ser así añadirá el ítem a dicho slot.

Slot

Este *script* se usa en un GameObject para hacerlo un slot de inventario en el que principal mente se almacena información del ItemObject correspondiente al slot y las interacciones con él.

Cuando se interactúa con el slot (ver Imagen 38) se detecta el tipo de click realizado, dependiendo de esta interacción se usará el objeto, se equipará de ser posible o se vera la información del objeto. Dependiendo del tipo de objeto que sea realizara un efecto otro.

Imagen 38: Slot inventario con un ItemObject

SlotEquipo

El *script* funciona de forma similar al script de slot pero con ciertos cambios, principalmente se encarga de almacenar el objeto equipado en dicho slot, está programado para elegir qué tipo de objeto puede almacenar en el slot estos tipos son: Arma, Casco, Pechera, Pantalones, Zapatos, Anillo, Collar, Escudo y Patata.

Cuando un objeto entra o sale del slot el *script* actualiza las estadísticas dependiendo de las que aporte el objeto (ver Imagen 39).

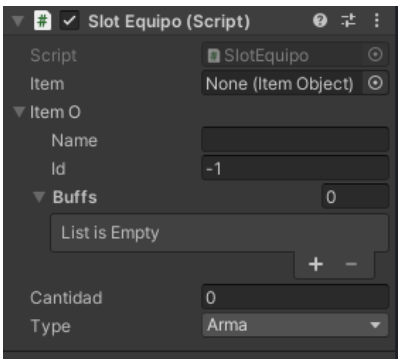

Imagen 39: SlotEquipo sin nada almacenado

InformacionInventario

Este script se encuentra dentro de un *GameObject* en la interfaz cuando en un slot se detecta un click derecho activa el *GameObject*. Este Script se encarga de mostrar en la interfaz creada la información del *ItemObject* deseado (ver Imagen 40).

Imagen 40: Información de un Ítem

6.5 Entorno

Hay varios objetos en el entorno que ayudan a complementar ciertas mecánicas o dar más vida al entorno en este punto veremos cómo funcionan estos elementos.

6.5.1 Cofres

Los cofres son elementos diseñados para poder obtener ítems, el ítem que obtienes depende del tipo de cofre y de la rareza el ítem, Cuando estas cerca de un cofre se puede abrir mediante la tecla F.

Componentes

Los componentes de los cofres son: *Transform, Spride Renderer, Animator, Box Collider2D* y su *script*, los componentes (ver Imagen 41) ya se han explicado antes ahora se explica el *script*:

Cofres

Es un script en el que se establece el tipo de cofre que es, contiene 6 arrays de rarezas, cada array contiene diferentes ítems. Cuando se interactúa con el cofre se consulta del tipo de cofre que es y se llama al método correspondiente al cofre para generar un numero aleatorio que decidirá que rareza soltara, dependiendo del cofre las probabilidades cambian, al elegir la rareza elegirá de forma aleatoria un ítem contenido en el array y lo instanciara en el mapa en la posición del cofre.

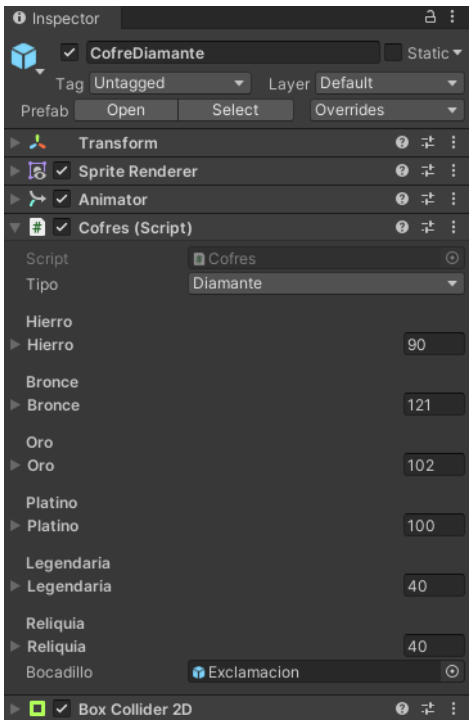

Imagen 41: Cofre de tipo Diamante

6.5.2 Torres

Las torres son elementos que ayudan al jugador, hay tres tipos de torres.

Las torres tienen los componentes de Collider 2D, Sprite Renderer

TorreTab

La torreTab es una forma de volver a la taberna.

PilarTP

El *script* hace que cuando el jugador este en la torre aparece el bocadillo de interacción si se interactúa con el objeto el jugador será enviado a la escena de la taberna.

PilarCuracion

Esta torre te permite curarte.

TorreCuracion

El *script* hace que cuando el jugador entre en contacto con la torre su vida es regenerada al máximo, estas torres dependiendo de la posición pueden curarte una vez o las veces que se quiera.

TorreSpawn

TorreCheck

El *script* hace que cuando el jugador entre en contacto con la torre se guarda la posición de la torre, cuando el jugador muera reaparecerá en la última torreSpawn visitada.

6.5.3 Elevador

El elevador es un elemento que se diseñó para ser utilizado como forma de conectar dos zonas de forma más sencilla.

Esta compuesto de varios objetos que ayudan a su funcionamiento los más destacables son: Dos Objetos que actúan como puntos destinos uno llamado StartedPoint y otro llamado FinalPoint, estos dos objetos delimitan de que punto a que otro ira el elevador al activarse. Otro Objeto es la palanca que es la que detecta la colisión para activar el ascensor.

Scripts

Elevador

Este *script* guarda como variables publicas el Objeto de la plataforma a mover, los puntos StartPoint, FinalPoint y una variable de tipo float que guarda la velocidad de movimiento de la

plataforma. Dependiendo del valor de una variable privada de tipo bool se determina hacía que punto debe de moverse la plataforma.

Palanca

Este *script* está situado en el objeto dentro del elevador que corresponde a la palanca su finalidad es detectar la colisión con la espada del personaje y cuando sucede esto llama a un método del script elevador que cambia el valor de la variable privada para así cambiar la dirección a la que se dirige el elevador.

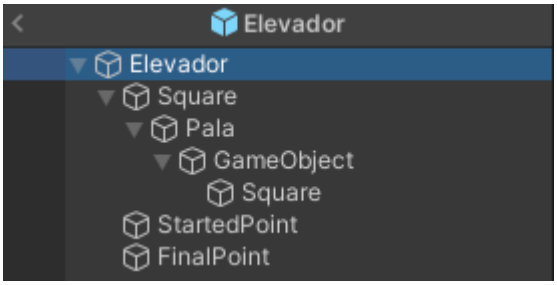

Imagen 42: Elevador distribución

Una cosa que recalcar es que según la imagen 42 el objeto Square corresponde a la parte del elevador que debe moverse y fuera de ese objeto se encuentran los objetos StartedPoint y FinalPoint estos dos objetos se encuentran fuera de la parte móvil porque si no los dos puntos se moverían con el elevador.

6.5.4 Zonas Ocultas

Las zonas ocultas son un clásico de este tipo de juegos, hacen que el jugador intente encontrar las máximas que pueda.

Scripts

OcultarZona

Este *script* se encuentra en el GameObject que oculta la zona, el funcionamiento de este *script* es simple, comprueba los triggersEnters que se encuentran dentro de una BoxCollider 2D si uno es estos objetos tiene el tag “Player” que es el correspondiente al jugador el script desactiva el componente Timemap Render de esta manera la zona que antes no se veía será visible.

6.6 Sonido

El sonido es una parte importante a la hora de meterse en el mundo del videojuego, el principal problema a la hora de pensar en cómo abordar la creación de sonido es de qué manera colocar el sonido y como activarlo. Al principio se pensó en poner una variable de sonido en donde hiciera falta, pero si se hacía de esta manera seria tedioso el encontrar cada sonido ya que estarían dispersados en diferentes *scripts*, la solución a la que se llego fue la creación de un *GameObject* que almacenara todos los sonidos y un script en este *GameObject* que almacenara los sonidos del *GameObject* listos para su ejecución, de esta manera solo se debía llamar al *script* donde se almacenan los sonidos con el sonido deseado a ejecutar (ver Imagen 43).

Imagen 43: Sonidos del juego

El sistema de sonidos está compuesto por 3 *scripts* que son los siguientes:

ActivadorSonidos

Este script es donde se almacenan todos sonidos listos para su ejecución, el script tiene un método que al recibir el nombre del sonido deseado a reproducirse entra en un switch para seleccionar el correcto. También se dispone de un método para parar los sonidos, este método pue creado ya que en algunos casos se debía de parar el sonido para que ajustara con lo deseado.

ControladorSonido

Este script hace posible que el *GameObject* donde se almacenan los objetos pase entre escenas.

ActivarSonido

Este *script* se añade en los estados que deben reproducir un sonido, tiene una variable publica de tipo *string* para poder determinar que sonido debe de ser ejecutado, de esta manera el script puede ser reutilizado. El *script* llama al método que activa los sonidos en ActivadorSonidos cuando entra en el estado.

6.7 Patrones de programación

En este punto se muestran los patrones de programación [10][11] que se han aplicado durante el desarrollo del proyecto y por qué de su uso.

6.7.1 Singleton

El patrón singleton [12] es uno de los dos patrones utilizados, el uso de este patrón se implementa para solucionar el problema de poder mantener algunos Objetos entre escenas y al mismo tiempo que no se repitieran.

La implementación de este patrón está en varios *scripts* que son: ControladorPersonaje, ControladorCMvcam, ControladorMainCamp, ControladorCanvas, ControladorEventos, ControladorInventario y ControladorSonido. El motivo de usarlo en varios scripts es el asegurarnos de mantener ciertos objetos entre escenas que son necesarios para el correcto funcionamiento del juego.

La implementación del patrón consiste en una variable static de la clase que se llama Instance, en el script en el método Awake() se comprueba si la instancia de la clase es null de ser así es la primera vez que se instancia y por ello have que la instancia sea igual a la instancia actual y emplea el metro DontDestroyOnLoad() para en el *GameObject*, si la instancia no fuera quiere decir que ya hay una instancia del objeto por ello la nueva instancia se destruye porque queremos evitar los repetidos de estas clases.

6.7.2 Prototype

El patrón prototype [13] es uno de los dos patrones utilizados, el uso de este patrón se implementa para solucionar el problema de la creación de Objetos que contienen *ItemObjects*, durante una de las pruebas que se realizaron en una de ellas nos percatamos que si tocaban dos objetos iguales las estadísticas eran las mismas en los dos, cosa que no estaba planeada la idea principal es que aunque tocaran dos objetos iguales las estadísticas fueran diferentes entre ellos por ello se decidió el aplicar el patón prototype para solucionar este problema.

La implementación de este patrón se encuentra en el script *ItemObject* ya que en este es donde se crea el objeto que se desea clonar.

La implementación del patrón consiste en la creación de un método llamado *getCopyItemObject()* en el que se crea un nuevo *ItemObject* que copia los valores del objeto que desea copiar, con la única diferencia que a la hora de copiar el array de *ItemBuff* este vuelve a generar los valores aleatorios para que sean diferentes al otro *ItemObject.*

6.8 Estándares de código

Los estándares de código son importantes a la hora de escribir un código ya que siguiendo ciertas pautas es mucho más fácil el entendimiento y el mantenimiento del código. En la asignatura de MES (mantenimiento y evolución del software) se muestra la importancia del seguimiento de estos estándares.

A continuación, se muestran ciertas pautas seguidas:

Los nombres son importantes ya que definen que función tiene la variable en el código, un nombre muy corto puede representar un problema a la hora de proporcionar información lo que hace más difícil entender el código. Por ejemplo, si se usa de nombre en una variable “co” hace que las personas que no han realizado esa parte del código no puedan saber a qué se refiere exactamente, si se utiliza como nombre de una variable “collider2D” hace que sea entendible en el contexto para las personas que analizan y mantienen el código.

El uso de las mayúsculas también es importante, en nuestros casos las variables comienzan en minúscula pero si disponen de otra palabra la segunda empieza en mayúscula un ejemplo es la variable “tiempoEntreAtaques”, en el caso de los métodos, nuestros métodos empiezan en mayúscula y las palabras siguientes también en mayúscula un ejemplo es el método “DesactivarEscudo()” esto es para ponerse en conjunto con los métodos de unity que empiezan en mayúscula.

En referencia a los comentarios del código, los comentarios del código son buenos a la hora de aclarar el funcionamiento de este, hay distintas opiniones en el mundo de la programación algunas personas opinan que un código bien estructurado no necesita comentarios y otras personas que opinan que los comentarios deben añadir claridad al código y por ello el código debería de tener comentarios simples que añadan información para que los demás entiendan el código. En el código del proyecto se pueden encontrar en algunos *scripts* comentarios que

ayudan al entendimiento del mismo, no en todas las funciones hay comentarios ya que hay funciones que se entienden lo que realizan por el nombre de la misma al ser funciones no complejas.

Un ejemplo de comentario en nuestro código es en el script *MovimientoPersonaje* en el que hay pequeños comentarios en partes del código para dar información, en la línea 162 de este script se encuentra el comienzo del método *DireccionMirando()* y en la línea 161 hay un comentario que da información del método que es: "Este método cambia el renderizado del Sprite del personaje para que mire hacia el lado en el que se mueve el personaje", no es un comentario corto ni muy largo es lo justo para dar a entender que realiza el método.

7 Pruebas

Durante la implementación del proyecto se han realizado diferentes pruebas a la hora de comprobar las funcionalidades de este, en este punto se ven algunas de las pruebas realizadas en el proyecto, no se explican todas las pruebas realizadas ya que para comprobar el funcionamiento de las partes se realizaron bastantes pruebas, en este punto se explican algunas de las más importantes.

Prueba General

La prueba general tiene como propósito el probar la funcionalidad del juego ya montado tratando de ver si se encuentra algún fallo durante el recorrido que hará el jugador. Esta prueba se realiza desde la página en la que esta subido el proyecto, es decir desde el lugar en el que se podrá jugar, de esta manera podemos encontrar fallos o problemas en el juego.

Fallos encontrados:

Diálogos

En el sistema de diálogos se requiere que estes en contacto con el collider del NPC para poder hablar con ellos, el fallo encontrado es que en los NPCs que tienen varias interacciones se debe salir del collider y volver a entrar para poder hablar con ellos de nuevo. Este fallo se ha intentado solucionar de diferentes maneras, pero cuando se solucionaba afectaba gravemente a la funcionalidad de los diálogos dejando inhabilitadas algunas partes, por ello se decidió que como era un fallo no grave que no afectaba en gran medida dejarlo por no disponer del tiempo suficiente para solucionarlo. Los NPCs que tienen todavía un dialogo por realizar les aparece un bocadillo sobre la cabeza.

Ítems

Los ítems deberían de poder tener diferentes estadísticas aun siendo el mismo tipo es decir dos espadas de luz aun siendo la misma deberían de tener diferentes valores de buffo. Durante la prueba realizada se encontró que este no está el caso y que tenían el mismo valor, para poder solucionar este problema se recurrió al patrón de programación Prototype, al usar este patrón solucionamos este error, los detalles del patrón han sido explicados en el punto 6.7.2.

Plataformas

Durante las pruebas realizada encontramos que el jugador no podía saltar correctamente entre plataformas, ya que se chocaba con la plataforma en lugar de atravesarla. Para solucionar este error se recurrió a la modificación del componente plataforma ya que dispone de una variable que establece el ángulo en el que se comporta como una plataforma, durante la prueba este valor estaba en 180 y después de la prueba se cambió a 140 solucionando así este error.

Otro error que se encontró durante la realización de varias pruebas es que en ocasiones puedes quedarte atascado en la plataforma sin poder moverte, no se ha podido descubrir porque sucede este fallo, sucede pocas veces y las condiciones del suceso no se han podido deducir.

Pruebas individuales

Una vez hablado de la prueba general que se llevó a cabo, se va a explicar algunas de las pruebas individuales más relevantes que se han realizado.

Pruebas realizadas a los enemigos

Como se ha explicado anteriormente los enemigos son una parte importante, por ello las pruebas realizadas a estos deben comprobar su correcto funcionamiento, también contamos con la experiencia de programar enemigos en otros proyectos por lo que algunos problemas que podrían surgir se evitaron antes como el caso de no poder una variable de tiempo entre ataques en el enemigo lo que provocaba que el enemigo atacara cada frame lo que hacía que atacara más de 30 veces por segundo, al contar con experiencia se evitó este fallo con una variable para controlar la repetición de la acción.

En las pruebas realizadas a los enemigos algunas se centraban en si el enemigo recibía el daño correctamente lo que concluyo que sí que recibía correctamente el daño, otra prueba se centraba en el movimiento del enemigo en modo centinela esta prueba sí que arrojo cierto fallos a la hora de realizarse, el motivo de estos fallos es que a la hora de inicializarse el enemigo se guardaba la posición como punto de retorno, al intentar volver al punto de inicio si el enemigo se había iniciado sobre el nivel del suelo este intentaba llegar al punto lo que resultaba imposible al estar en el aire, para solucionar este error los enemigos se posicionaron a nivel del suelo para así no tener que preocuparse por guardar un punto inaccesible.

Pruebas realizadas al personaje

Al personaje se le han aplicado varias pruebas durante el proyecto ya que al ir añadiendo más funcionalidades podían afectar a las acciones que realizaba el personaje. Algunas de estas pruebas son las siguientes:

- **Prueba de ataque:** Unas de las primeras pruebas que se realizaron fuero la de ataque del jugador, dando como resultado que se detectó un problema a la hora de atacar, el problema que surgió fue que al atacar a un enemigo y no moverse del lugar los siguientes ataques no afectaban al enemigo, la causa del problema se detectó en el collider del arma, aunque se pasaba a un estado de desactivado para el juego es como si no se moviera del lugar por ello no se actualizaba al entrar en contacto porque ya lo había hecho, para solucionarlo se modificó el collider para que una vez atacara el jugador este redujera su tamaño y con esto se solucionó el error.
- **Prueba del árbol de estados:** Esta prueba se realizó muchas veces, queríamos que el cambio de estados fuera lo más fluido y exacto posible, por ello la prueba se repitió muchas veces para ajustar los parámetros de las transiciones, las primeras pruebas son las que más se notaba la necesidad de modificar ciertos parámetros ya que las transiciones a estados tienen por defecto un tiempo de transición que ronda entre los 0,25 y 0,5 segundos, este tiempo en algunas transiciones se tenía que aumentar o quitar, por ejemplo en la transición a ataque se debía de quitar ya que requeríamos que la animación fuera instantánea.

Pruebas realizadas al inventario

El inventario es una funcionalidad importante del juego por lo que queríamos asegurarnos de que el inventario funcionara correctamente ya que aparte de guardar los ítems también interactuaba con las estadísticas mediante el inventario de equipo. Las pruebas del inventario fueron minuciosas para comprobar su correcto funcionamiento, de estas pruebas destacar las siguientes:

- **Prueba para mostrar estadísticas:** Dentro del inventario hay una funcionalidad para hacer aparecer una interfaz donde se ven los detalles del objeto, la finalidad de esta prueba es comprobar que las estadísticas del objeto se muestran correctamente al usuario, esto lo podemos comprobar mirando la interfaz los valores que nos muestra y compararlos con los valores que hay en los componentes del Ítem. La primera vez que se realizó esta prueba dio un fallo a la hora de mostrar las estadísticas, este fallo fue provocado por un problema con el array que se encarga de almacenar las estadísticas en los slots, la solución fue modificar ligeramente el método para su correcto funcionamiento después de la modificación no volvió a repetirse el error.
- **Prueba pasar objetos de inventario al inventario equipamiento:** Esta prueba tiene como objetivo el asegurarse que al pasarse los objetos permanecen sin cambios los mismos y que las estadísticas se suman correctamente al jugador, también probar si al desequipar un objeto se quitaban las estadísticas correctamente y el objeto pasaba al inventario normal. Para ello se anotaron los valores del objeto antes de pasar el objeto y los avalores de las estadísticas, al pasar el objeto al inventario de equipamiento se comprueba si los valores sumados concuerdan con los esperados, esta prueba paso los requisitos a la primera por lo que no fue necesario modificar nada, por si acaso la prueba se repitió varias veces, pero el resultado continúo siendo positivo.

Pruebas a Usuarios

Las pruebas con usuarios externos al desarrollo del proyecto son importantes ya que prueban el proyecto desde un punto de vista imparcial, esto hace que podamos obtener datos relevantes para el desarrollo e ideas para implementar, durante esta prueba el juego fue probado por 14 personas y estas dieron su opinion y también reportaron algunos bugs existentes.

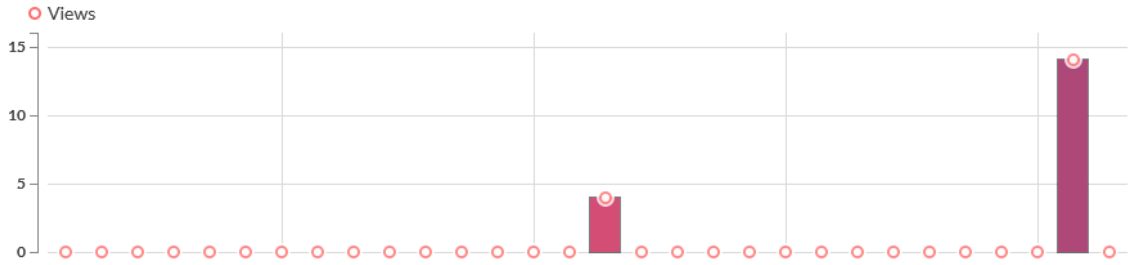

Imagen 44: Grafica de usuarios

Como se puede apreciar en la imagen 44 el juego fue probado por usuarios externos al proyecto 2 veces, la primera fue un grupo pequeño de 4 personas que para comprobar el funcionamiento en la página web, la segunda vez fue un grupo de 14 personas que probaron el juego en más profundidad, en estas dos pruebas se recogieron las opiniones de los usuarios y los reportes de bugs. A continuación, se explican los fallos encontrados, soluciones e ideas de los usuarios divididas en los dos grupos de pruebas.

Grupo 1 (4 personas)

Este grupo probaron el juego mínimamente para comprobar si funcionaba correctamente en la página web, durante esta prueba se reportó el siguiente fallo:

- **Problema con la interfaz:** la interfaz fue diseñada para una pantalla de 1920x1080 lo que provocaba que al jugar en otras resoluciones no se viera bien la pantalla, este problema se solucionó ajustando algunos ajustes de la interfaz para así poder permitir que se autoajuste dependiendo de la resolución del usuario.

Grupo 2 (14 personas)

Esta prueba fue más extensa que la anterior y en ella los usuarios tenían plena libertad para moverse por el mapa, durante esta prueba se reportó los siguientes fallos:

- **Vida del jugador:** Durante la prueba se detectó un problema con la vida del jugador este problema consistía que si no se abría el inventario/sistema de estadísticas antes de recibir daño no se actualizaba correctamente la vida, por lo que al recibir daño sin haber abierto las estadísticas el jugador se volvía inmortal, este problema tiene que ver con las actualizaciones de la interfaz al no estar visible, el problema se solucionó haciendo que durante la escena de carga se abriera y cerrara automáticamente para así obligar a que se actualice correctamente.

- **Vida jefes:** La vida de los jefes es un elemento de interfaz que aparece arriba de la misma, el problema detectado es que al jugar en pantalla minimizada o en una resolución inferior a la programada no se veía correctamente, la solución fue la misma que en la primera prueba de usuarios, ajustar unos parámetros de la interfaz para que se autoajustara, de esta manera se solucionó.
- **Fallo con una trampa:** La trampa de tipo bomba no funcionaba correctamente, no hacia daño el personaje, esto era debido a que en el último frame de animación el collider estaba como no trigger entonces se comportaba como un sólido, la solución fue activar este parámetro.
- **Fallo con las paredes:** Si el jugador estaba en el aire pegado a una pared y intentaba ir hacia la pared se quedaba quieto en el aire hasta que parara de caminar, como debería de funcionar es que no se quedara pillado con la pared y bajara hacia abajo como lo haría normalmente en el aire, para solucionarlo se pusieron dos detectores en el lateral del personaje para que detectara la pared y entonces no permitir que se desplazara con esto se solucionó este error, como se puede ver en la imagen 45, se aprecia en la parte derecha un rectángulo, en cada terminación del rectángulo se encuentra una esfera de 0,7 de radio para poder detectar la interacción con la pared.

Imagen 45: Personaje con detectores

Estos son algunos de los errores detectados por las personas que probaron el proyecto, ahora se explican las sugerencias y peticiones que dieron.

- **Regenerar mana:** La gran mayoría de personas que probaron el proyecto coincidieron en una sugerencia y fue que al morir no se regeneraba el mana y sugirieron que se regenerara el mana al morir porque si no sería muy injusto el no poder usar el mana, por ello se modificó y se puso que se regenerara el mana al morir.
- **Cofre de armas:** La aleatoriedad de los cofres hace que en algunas partidas hay la posibilidad de que no obtengas un arma en toda la partida, por ello sugirieron que sería buena idea que se añadiera un cofre que solo de armas, esta idea se tuvo en cuenta y se decidió que se añadiera como futuros trabajos a realizar.

8 Conclusiones

Al inicio del desarrollo del proyecto nos propusimos como objetivo principal el lograr una versión jugable de un videojuego 2D de tipo *metroidvania*, este objetivo se ha cumplido de cierta manera, hemos conseguido una versión del juego jugable con varios enemigos funcionales y 2 jefes de sala.

En lo que respecta a los otros objetivos hemos cumplido el aplicar una metodología ágil durante el desarrollo del proyecto, esto ha sido de mucha utilidad para poder llevar un plan de trabajo y seguimiento óptimos. Al mismo a la hora de realizar las pruebas pertinentes se ha comprobado la importancia de obtener una opinion externa al proyecto para poder realizar cambios y mejoras.

En el objetivo de aplicar patrones de programación, hemos podido aplicar dos patrones de programación, estos patrones han ayudado bastante a solucionar problemas que encontramos y a facilitar parte de la programación.

Los objetivos secundarios también se han podido cumplir en cierta manera. El objetivo secundario de crear una zona del mapa que obligue al jugador a explorar se ha conseguido con la zona del mapa de la mazmorra. El objetivo de que los enemigos sean diferentes entre ellos se ha cumplido, aunque si no fuera por la limitación de tiempo se abrían podido añadir mas variedad de enemigos y elementos que los diferencien significativamente entre ellos. Añadir elementos con los que se puedan interactuar con ellos se ha conseguido con la adición de cofres, torres y NPCs, estos elementos ofrecen un mayor incentivo para que el jugador explore. Para cumplir que los enemigos tengan diferentes interacciones se les añadió el que si no esta el jugador cerca de ellos se encuentran en un estado de centinela que hace que se muevan entre dos puntos.

El código fuente del proyecto se encuentra en el siguiente enlace: **https://github.com/diegoger222/TFG-Diego-Adrian**

8.1 Relación del trabajo desarrollado con los estudios cursados

Para el desarrollo de este proyecto se ha usado el conocimiento obtenido en los estudios cursados que en este caso es la rama de ingeniería del software, también a esto se le suma el conocimiento obtenido mediante las investigaciones para otros proyectos que se realizaron durante la carrera.

Para la aplicación de la metodología ágil hemos aplicado lo aprendido en DSW y en PIN, donde se nos enseñó en qué consistía la metodología ágil y su uso mediante proyectos de software. Para planificar las UTs en la metodología ágil se decidió aplicar lo aprendido en AER donde mediante casos de uso se analizaba un sistema para su correcto funcionamiento, gracias a pensar en los requisitos funcionales se ha podido planear correctamente lo que debe de cumplir.

A la hora de aplicar patrones de programación se aplica lo aprendido en la asignatura DDS donde se aprendió patrones de programación y la importancia de su aplicación en diversos casos.

Para los estándares de código se ha utilizado lo aprendido en MES donde se nos enseñó la importancia de una estructuración de código para que su entendimiento y mantenimiento fuera más sencillo de lograr.

También tener en cuenta las asignaturas optativas de Desarrollo de videojuegos 2D y Desarrollo de videojuegos 3D, de estas asignaturas es de donde surgió la idea de realizar este proyecto.

Como conclusión se han utilizado bastantes conocimientos aprendidos en los estudios cursados, también se han adquirido conocimientos a la hora de investigar para realizar el proyecto.

9 Trabajos futuros

Los trabajos a futuro que surgen apartir de este proyecto están orientados a mejorar el videojuego tanto en la ampliación de zonas y tanto en la jugabilidad. Estos trabajos son los siguientes:

- Desarrollar un mapa más amplio, el construir un mapa acorde a la jugabilidad y historia es una tarea complicad por lo que sería un buen punto el ampliar el mapa de acorde al primer concepto de mapa.
- Mejorar las mecánicas del personaje, el personaje tiene varias mecánicas, pero se pueden ampliar las mecánicas, al mismo tiempo que se amplía las mecánicas es importante el pulir las mecánicas ya existentes para hacerlas más simples y optimas.
- Mejorar el tiempo de carga entre pantallas, la carga entre pantallas no es un problema a no ser que la escena a la que se vaya tenga muchos objetos y mapeado, en el mundo de los videojuegos existen varios métodos de optimización de carga y de mapeado que no ha dado tiempo a investigar durante el desarrollo.
- Crear una IA enemiga amplia, la IA es una parte que atrae la atención de las personas, durante el desarrollo del proyecto se pensó en la idea de crear una IA mediante matrices de posibilidades ligadas a acciones, pero al ser un trabajo demasiado complejo para poder ser realizado en el periodo de tiempo disponible se decidió dejar las acciones de los enemigos que se decidieran de forma simple, pero la ida de crear una IA amplia de esta manera es un concepto interesante.
- Crear nuevas funcionalidades que sean puzles como activadores que se activan golpeándolos con una flecha de arco, esto es una cosa que aparece en algunos videojuegos de este tipo y que ayudan a mejorar el ambiente.
- Crear diversos cofres que suelten equipamiento especifico, como se ha dicho antes la aleatoriedad del juego hace ser posible que no se obtenga ningún arma en alguna partida, por ello se debería al menos añadir un cofre que suelte una espada asegurada.

10 Bibliografía

[1] Metroidvania | fandom https://castlevania.fandom.com/es/wiki/Metroidvania

[2] Hollow Knight https://www.hollowknight.com/

[3] Dead Cells https://dead-cells.com/

[4] Ori and the blind Forest | fandom https://oriandtheblindforest.fandom.com

[5] Soluciones | unity. https://unity.com/es/solutions

[6] Basese | unity. https://docs.unity3d.com/es/530/Manual/UnityBasics.html

[7] GameObject | gamedevtraum. https://gamedevtraum.com/es/desarrollo-de-videojuegos-y-aplicaciones-con-unity/serie-manejo-general-del-motor-unity/unity-que-es-un-gameobject-y-cuales-son-sus-propiedades/

[8] Flujo de ejecución | docs unity. https://docs.unity3d.com/es/530/Manual/ExecutionOrder.html

[9] Hollow Knight World Edit | GitHub https://github.com/nesrak1/HKWorldEdit2

[10] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software* Addison Wesley ISBN 10: 0201633612

[11] Alexander Shvetes, Patrones de diseño. *Sumérgete en los patrones de diseño https://refactoring.guru/es/design-patterns/book*

[12] Refactoring Guru | https://refactoring.guru/es

[13] Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates. *Head First Design Patterns* O'Reilly Media ISBN 10: 9780596007126

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**		X		
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**				X
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.				X
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**		X		

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Al tratarse de un proyecto relacionado con un videojuego 2D no se ha podido encontrar un grado de relación alto con ningún ODS, se han analizado las diferentes opciones y al final se ha concluido que el proyecto está relacionado en cierta medida con los siguientes ODS:

- El ODS 4 (educación de calidad) está relacionado de forma media ya que se piensa que el trabajo realizado podría usarse como ejemplo a la hora de enseñar dentro del ámbito de la programación de videojuegos, de forma que se mostrara los posibles pasos a seguir a la hora de la realización de un proyecto de videojuego para que las personas supieran en que aspectos enfocarse. También se puede utilizar la parte de código donde se han aplicado los patrones de diseño como ejemplos de uso de estos mismos en la creación de videojuegos.
- El ODS 17 (alianzas para lograr objetivos) esta relacionado de forma media ya que al estar el proyecto publicado de forma publica en Itch.io y en GitHub permite que cualquier persona del mundo pueda ver el programa para aprender conceptos o ideas. Al mismo tiempo en el proyecto se utiliza diversas informaciones de distintas partes del mundo para la creación de este. A parte de lo comentado anteriormente este proyecto en si mismo se ha realizado de forma colaborativa con otra persona para conseguir el resultado final, fomentando así el intercambio de información.

Como hemos visto el trabajo realizado esta relacionado con pocos ODS, la mayor relación que podría tener es en el ámbito del estudio aportando información y ejemplos a la hora de la realización de un videojuego o de un proyecto.