

DOCUMENTACIÓN TFG



ÍNDICE

RESUMEN.....	3
ABSTRACT.....	3
PLANTEAMIENTO DEL PROBLEMA Y JUSTIFICACIÓN.....	3
PLANIFICACIÓN.....	4
Primeros pasos.....	4
Organización.....	5
DATOS TÉCNICOS.....	6
DISEÑO.....	6
Controles multiplataforma.....	6
Prefabs.....	6
Diseño modular.....	7
Menús.....	7
Sistema vida.....	8
Sistema daño.....	8
Armas incorporadas al jugador.....	9
GRÁFICOS.....	10
Sistema de renderizado Pipeline.....	10
Cell shading.....	10
FX cartoon.....	11
FX gore.....	12
Animaciones.....	12
Animaciones aditivas.....	12
Skybox.....	14
SONIDO.....	14
Sistema de sonidos aleatorios para diversidad.....	14
Música.....	15
Sonido 3d de los enemigos.....	15
Sistema de sonido del jugador.....	17
GAMEPLAY.....	18
Sistema de movimiento old.....	18
Sistema de movimiento new.....	18
Enemigos old.....	19
Enemigos new.....	19
Detección jugador por enemigos.....	20
Pathfinding nuevo.....	20
Cámara que sigue al jugador, fov, etc.....	21
PickUps.....	22
HUD dinámico.....	22
Jugador con habilidades modulares (Revisar).....	23
Skins.....	24
Animaciones del jugador adaptables a lo que suceda.....	25

Mapa que hace cosas(world turn).....	26
Inventario jugador.....	27
Físicas old.....	28
Físicas new.....	28
Ataque melee old.....	28
Game controller.....	28
Boss fight.....	28
RENDIMIENTO.....	30
CONCLUSIONES.....	31
REFERENCIAS BIBLIOGRÁFICAS:.....	31
DICCIONARIO.....	33

RESUMEN

Nuestro videojuego, “Doom 2punto5”, es un juego *shooter* (juego de disparos) de avance lateral (2D) en un entorno 3D inspirado en la franquicia de “Doom”.

Este juego ha sido desarrollado en el motor de videojuegos “Unity”, por lo que hemos usando C#, un lenguaje de programación que no habíamos estudiado previamente en el grado.

En este juego se implementan sistemas de amplio movimiento, variedad de enemigos y variedad de armas para crear una sensación de juego dinámica y entretenida, además de hacer uso de modelados 3D y sus animaciones para tanto el jugador y enemigos como para el propio mapa y sus elementos. También en el apartado 2D hemos tenido que aprender como utilizar y modificar sprites.

Además cabe destacar el sistema de partículas y el sonido dinámico del juego para mejorar la inmersión y la experiencia de juego.

En conclusión, hemos desarrollado un videojuego que manteniendo una jugabilidad 2D explora cómo aprovechar el entorno 3D y se centra en crear una experiencia inmersiva e interesante para el jugador.

ABSTRACT

Our videogame, “Doom 2punto5”, is a lateral scroll (2D) in a 3D environment shooter videogame inspired in the Doom franchise.

This game has been developed in the game engine “Unity”, which means we have used the programming language C# for its development, a programming language we didn’t study at the vocational training.

In this game we have implemented ample movement systems, various enemies and different weapons to create a dynamic and entertaining gaming feeling while using 3D models of the player, the enemies, the map and its elements and their animations. On top of that, on the 2D side we have learnt how to use and edit sprites.

In addition, it’s worth mentioning the use of particle systems and dynamic sound FX to improve the player’s immersion and the gaming experience as a whole.

In conclusion, we have developed a videogame that, while keeping a 2D gameplay, explores how to take advantage of 3D assets and focuses on making an interesting and immersive experience for the player.

PLANTEAMIENTO DEL PROBLEMA Y JUSTIFICACIÓN

A la hora de decidir el proyecto pensamos que hacer un videojuego podría ser un desafío interesante y nos pusimos a pensar que era algo a lo que quisiesemos jugar. Optamos entonces por un juego de disparos de desplazamiento lateral, pero que en vez de tener gráficos 2D tuviese gráficos 3D, algo que no es muy común en el mercado.

Pensamos que un juego de estas características podría ser un reto interesante para un TFG, además de ser un tema que nos gusta, del que podemos aprender mucho y que si salía bien, en un futuro cambiándolo un poco se pudiese llegar a comercializar.

PLANIFICACIÓN

Primeros pasos

La primera parte a la hora de decidir el proyecto, fue elegir que queríamos hacer. Rápidamente decidimos hacer un videojuego y terminamos decantandonos por el género *Run and gun*, es decir un juego de scroll lateral con disparos.

Tras esto necesitábamos saber qué queríamos hacer exactamente por lo cual comenzamos a probar diferentes juegos del género para aprender qué cosas eran necesarias, que mecánicas, nos parecían interesantes, que cosas no nos gustaban, etc. Los videojuegos probados fueron:

- 2Doom
- Doom 2
- Metal Slug
- Kirby
- Valfaris
- Castlevania
- Metroid
- Dead cells

Mientras probamos los diferentes videojuegos íbamos apuntando cada uno de los aspectos que consideramos importantes.

Los apuntes que tomamos fueron:

3D—>

- Usar Mixamo para las animaciones
- Animaciones independientes de un mismo modelo

General—>

- Controlar altura salto
- Apuntado independiente al movimiento
- Que el personaje parpadee al recibir daño
- Identificar daño
- Animar respawn

- Dash
- Doble salto
- Arma cuerpo a cuerpo
- La cámara sigue al jugador
- Vibración de pantalla al disparar
- Dar sensación de inercia
- Immortality frames
- Collision box grande
- Respuesta inmediata a acción
- Cuando un enemigo recibe daño avanzara más lento
- Reducir al mínimo la exposición
- Tutorial que no da la sensación de tutorial
- La posición de los enemigos será relativa al rango de las armas usadas

Tras probar videojuegos empezamos a buscar tutoriales de como se hacen para decidir en qué motor gráfico podríamos trabajar. Tras mirar varias opciones, llegamos a la conclusión de que desarrollar en *Unity* sería la mejor opción.

Después de esto empezamos a buscar tutoriales de juegos *Run and Gun* en *Unity*. Esto nos llevó a cuestionarnos si queríamos que fuese 2D o 2.5D, por lo que miramos qué materiales podemos usar para hacer el videojuego y, tras ver que teníamos más recursos 3D para realizar el juego, optamos por la versión 2.5D.

A continuación de esta decisión volvimos a probar algún videojuego de *Scroll lateral* en 2.5D para ver cómo podría quedar el videojuego que teníamos en mente, también miramos y descargamos recursos 3D y animaciones que podríamos necesitar para hacer el videojuego.

Por último para la etapa inicial del proyecto, comenzó la búsqueda de tutoriales y guías para aprender a crear videojuegos en unity que fuesen *Scroll lateral* y en 2.5D.

Organización

A la hora de desarrollar el proyecto, una vez tuvimos claro que era lo que queríamos hacer, empezamos a organizarnos para programar siguiendo un tutorial todas las noches. Para la organización del proyecto usamos la herramienta GitHub Desktop junto con LFS para gestionar archivos de gran tamaño, y mientras seguíamos los tutoriales, programábamos de forma alterna para que ambos entendiésemos y aprendiésemos bien sobre lo que estábamos haciendo.

Terminados los tutoriales continuamos con esta dinámica de trabajo en la que uno se dedicaba a programar mientras que el otro estaba de apoyo aportando ideas, viendo foros buscando soluciones o revisando que todo esté avanzando bien.

Esta decisión la tomamos en parte porque nos había ido bien y estábamos aprendiendo mucho, pero también porque no conocíamos bien *Unity* y veíamos que solo con abrir el

proyecto tras bajarlo del repositorio se podían llegar a producir 20.000 cambios y nos daba la sensación de que habría muchos problemas al intentar sincronizar versiones. Ahora sabemos que era porque los proyectos los teníamos guardados en rutas distintas en cada ordenador.

Cuando el proyecto ya alcanzó una etapa más avanzada, el rol de la persona que no estaba programando pasó de tener un rol de apoyo a encargarse de la documentación, así ambos podríamos avanzar en distintas partes del proyecto y si había alguna duda nos podríamos preguntar en el momento.

Cabe destacar un problema que tuvimos con *GitHub* y *LFS*. En las fases finales del proyecto tuvimos un problema en el que *GitHub* no nos permitía actualizar el repositorio porque habíamos excedido el bando de ancha del *LFS*, entonces si queríamos continuar teníamos que pagar 60€ o esperar un mes para que nos diesen más bando de ancha de forma gratuita.

Como ninguna opción era viable nos paramos a ver qué archivos eran los que nos estaban dando problemas de tamaño y vimos que ambos eran archivos autogenerados por Unity. Decidimos entonces crear un nuevo repositorio y añadir todos los archivos del proyecto menos esos dos y deshabilitar el *LFS* para no volver a tener este problema.

DATOS TÉCNICOS

- Nombre de la aplicación: Doom 2punto5
- Plataforma: PC
- Lenguaje de programación: C#
- Requisitos mínimos del Sistema:
- Manual de instalación: Descargar el juego y ejecutar el .exe
- Plataformas: PC

DISEÑO

Controles multiplataforma

Con el objetivo de que el juego pudiese ser jugado con diferentes *input* hardwares se busco la manera mas sencilla y eficiente de lograr esto, tras una búsqueda se descubrió que hay una extensión oficial de *Unity* llamada *Input System* que permite establecer los botones de diferentes mandos a la misma acción dentro del videojuego permitiendo que este sin importar el mando que use detecte el mando y cada botón haga las acciones esperadas.

Prefabs

Unity tiene filosofía de trabajo basada en *prefabs* que son básicamente objetos o elementos que tu creas y guardas para en el futuro copiarlos y ponerlos de nuevo donde necesites en el juego, de manera que habiendo creado un solo enemigo, una vez creado

el *prefab*, puedes copiar y pegar este enemigo donde quieras y será independiente a los demás sin necesidad de programar o crear nada nuevo.

Además, son muy útiles ya que a partir del *prefab* los objetos nuevos que crees serán independientes de él, permitiendo así partir de una base que se podrá modificar completamente para una máxima flexibilidad y adaptabilidad a la hora de trabajar con ellos.

Diseño modular

Con el objetivo de que el desarrollo y expansión del videojuego fuese más fácil y adaptable, se decidió que no solo se usasen y creasen la máxima cantidad de *prefabs*, también que estos tuvieran los diferentes comportamientos y funciones de manera separada e independiente, de forma que si queríamos añadir o quitar más funciones o comportamientos, los objetos no necesiten un rediseño más allá de quitar aquello que no se vaya a utilizar y añadir la nueva función, que una vez terminada estará disponible para añadirse en los demás objetos.

Menús

Para los menús se han creado objetos que se muestran en la capa superior del elemento *Canvas* de la escena. En este elemento, en el que se muestra la interfaz del juego, se encuentran ocultos los menús que, cuando se cumplan las condiciones, pararán el juego y se volverán visibles.

Los menús tienen botones con los que el jugador puede interactuar.

El juego tiene tres menús:

- Principal: El menú principal está en su propia escena y contiene una imagen con dos botones, uno para comenzar el juego y otro para salir.



- Pausa: Este menú se encuentra dentro de las escenas de nivel y cuando el jugador pulsa el botón de pausa se activa, parando el resto del juego. Tiene tres botones, uno para continuar, otro para salir al menú principal y otro para salir del juego.

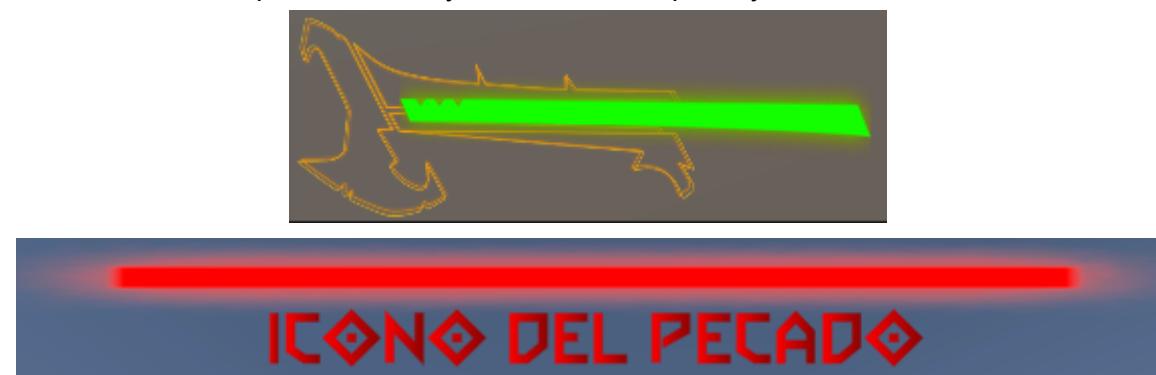


- Muerte: Este menú se encuentra dentro de las escenas de nivel y cuando la vida del jugador llegue a 0, tras un corto periodo de tiempo se activa, parando el resto del juego. Tiene tres botones, uno para reiniciar el nivel, otro para salir al menú principal y otro para salir del juego.



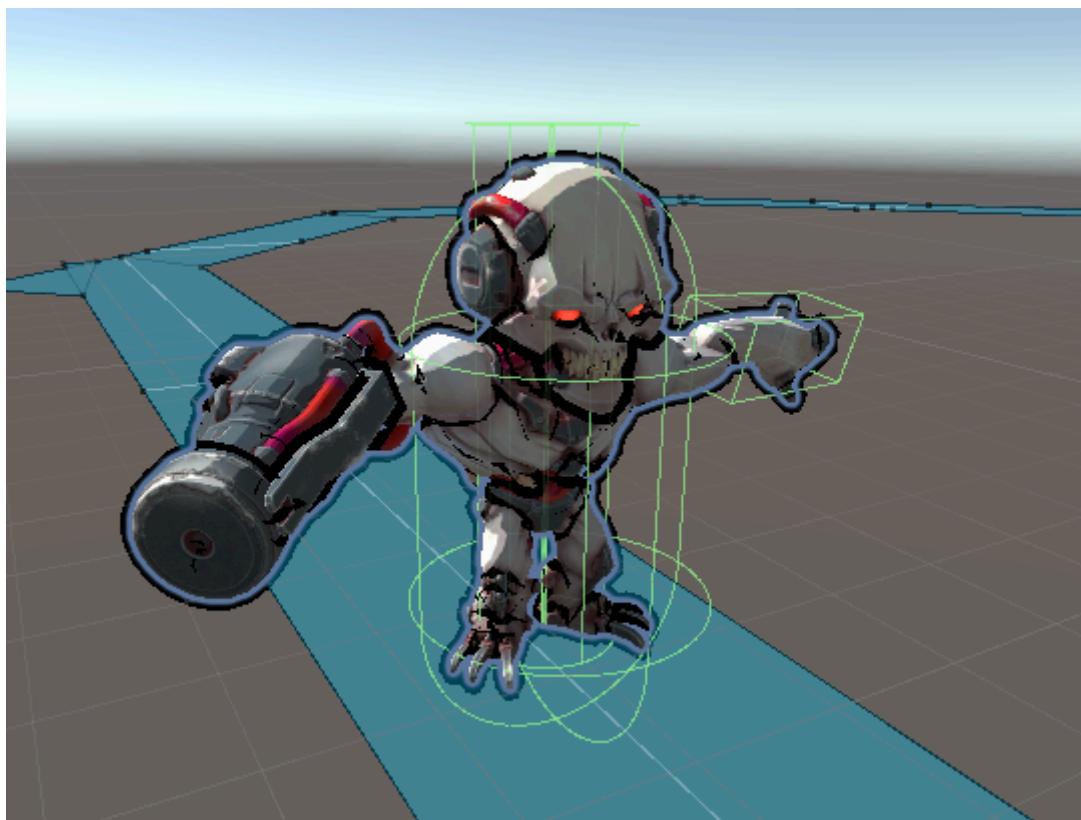
Sistema vida

Para el sistema de vida optamos por un sistema en el que cada elemento de Jugador o Enemigo tuviese un valor numérico que se pudiese modificar desde *Unity*. Cuando este valor llegue a 0 el objeto dejará de estar visible y ejecutará una animación y un sonido de muerte. Para el jugador y el enemigo final estos valores se representarán en pantalla con una barra de vida, que irá disminuyendo a medida que vayan recibiendo daño.



Sistema daño

Para el sistema de daño utilizamos *Colliders*, que con la propiedad “isTrigger” pueden atravesar otros objetos del mapa. Gracias a esto, podemos poner *Colliders* en objetos y detectar qué tipo de objetos están atravesando. Para el caso del jugador, el solo puede hacer daño con balas, así que las balas tienen un *Collider* y un *script* que les permite hacer daño si chocan contra un enemigo o destruirse si chocan contra una superficie. En el caso de los enemigos pueden hacer daño con proyectiles y con ataques a melee, así que tienen este tipo de *Collider* en los proyectiles y en las partes del cuerpo con las que ataque (Mayormente los brazos).



Armas incorporadas al jugador

Con el objetivo de facilitar el uso y obtención de armas en el videojuego, se vio en diferentes tutoriales que lo que se hacía es que todas las armas están en el lugar final (la mano del personaje) y a través de código activamos una y desactivas todas las demás de manera que no necesitas cargar nuevos elementos en el escenario y cambiar de una arma a otra es instantáneo, para lograr esto primero se pusieron todas las armas en la posición final y se le asignó a cada arma un número para identificar cual estará activa y cuales están desbloqueadas, gracias a esto permitimos no solo limitar las armas a las que tiene acceso y cual esta usando en todo momento, también permitimos que el jugador pueda desbloquear armas con *PickUps* y así lograr una progresión y variedad de opciones al jugador.



GRÁFICOS

Sistema de renderizado Pipeline

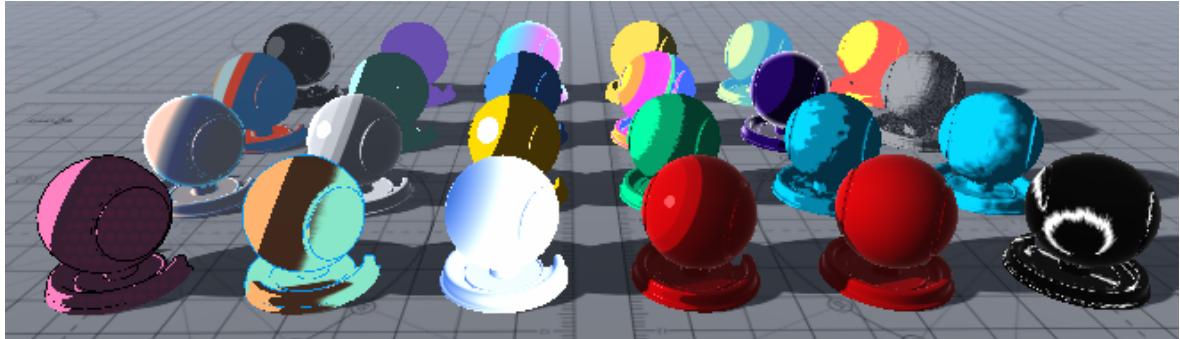
Se estuvo usando el sistema de renderizado por defecto de *Unity*, pero tras ver diferentes Assets y leer la documentación de *Unity* en relación a *Universal Render Pipeline* se decidió cambiar a este sistema de renderizado ya que nos proporcionaba un mejor rendimiento, más versatilidad en todo lo relacionado al aspecto visual del videojuego y que en principio en un tiempo este sistema de renderizado va a ser el por defecto de *Unity* eliminando el sistema *Classic*.

La implementación de este nuevo sistema de renderizado supuso la modificación de todos los materiales del proyecto adaptándolos al nuevo sistema de renderizado.

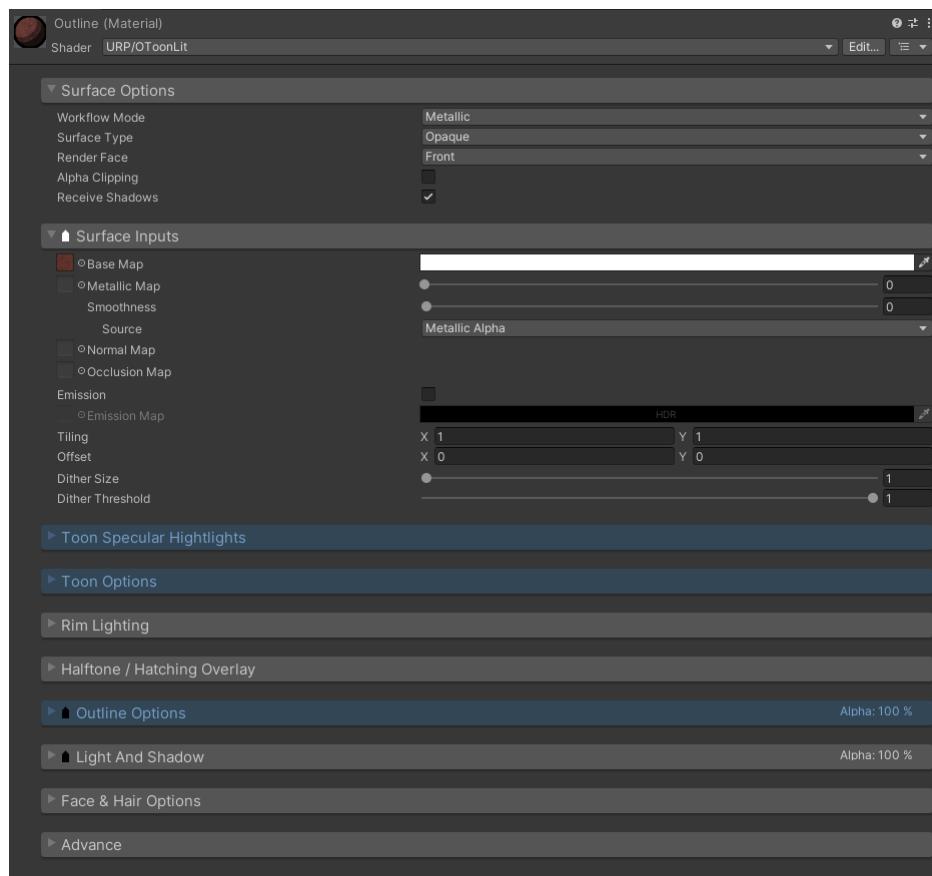
Entre las nuevas posibilidades de usar este sistema de renderizado es el utilizar un sistema de *Cell Shading* (efecto caricaturesco, sobre materiales, texturas, sombras, etc.) más eficiente y bonito.

Cell shading

Para mantener una estética constante entre los diferentes elementos del videojuego se tomó la decisión de implementar un estilo artístico que se pudiese aplicar a los diferentes elementos 3D del videojuego, optamos por el *Cell Shading* ya que permitía mantener una estética caricaturesca, y unificar como se veían todos los elementos 3D, para lograr esto se instaló un *Asset* que daba varios ejemplos.

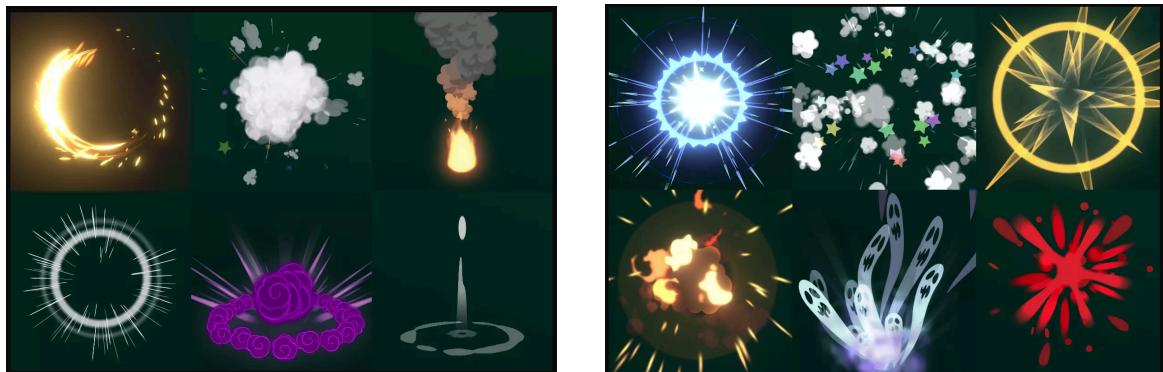


Una vez decidido qué estilo de *Cell Shading* queríamos implementar, creamos un nuevo material con las características deseadas y lo implementamos sobre todos los Modelos 3D cambiando solo la textura.



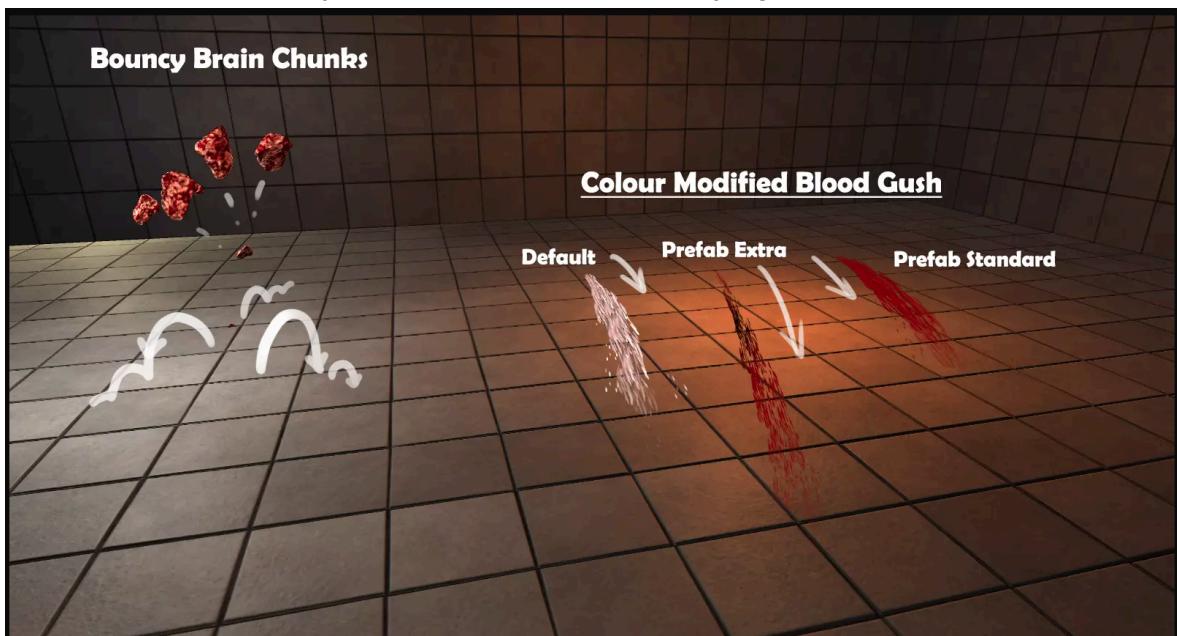
FX cartoon

Para los efectos visuales generales del videojuego se buscaron diferentes *assets* que encajasen con la estética del videojuego hasta encontrar un paquete de efectos con estilo caricaturesco que nos podría servir, este paquete fue modificado y adaptado para que encajasen aún más con la estética del videojuego. Para lograr esto se modificaron los Scripts de varios efectos para eliminar partes que no nos interesaban o modificarlos a nivel de código para que encajasen mejor.



FX gore

Aunque la estética del videojuego es bastante caricaturesca se quería implementar algo de gore para lograr esto se buscaron *assets* de efectos visuales que cumpliesen el objetivo deseado y se modificaron a nivel de código, texturas, modelos y efectos de partículas para que encajasen con la estética del videojuego.



Animaciones

Para poder asignar diferentes animaciones se ha usado un objeto dado por *Unity* llamado *animator* que nos permite manejar y controlar las diferentes animaciones de un elemento y la transición y cambio entre estas.

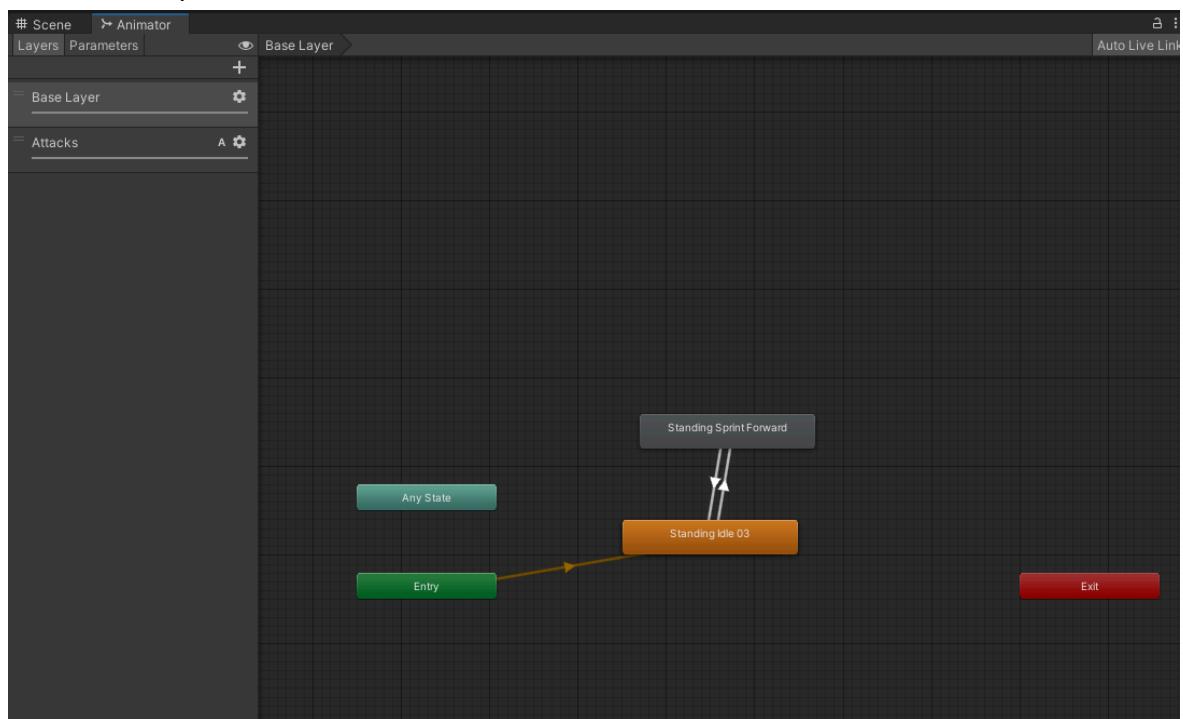
Para el uso e iniciación de las diferentes animaciones en el videojuego se han usado dos maneras principales de iniciar estas animaciones:

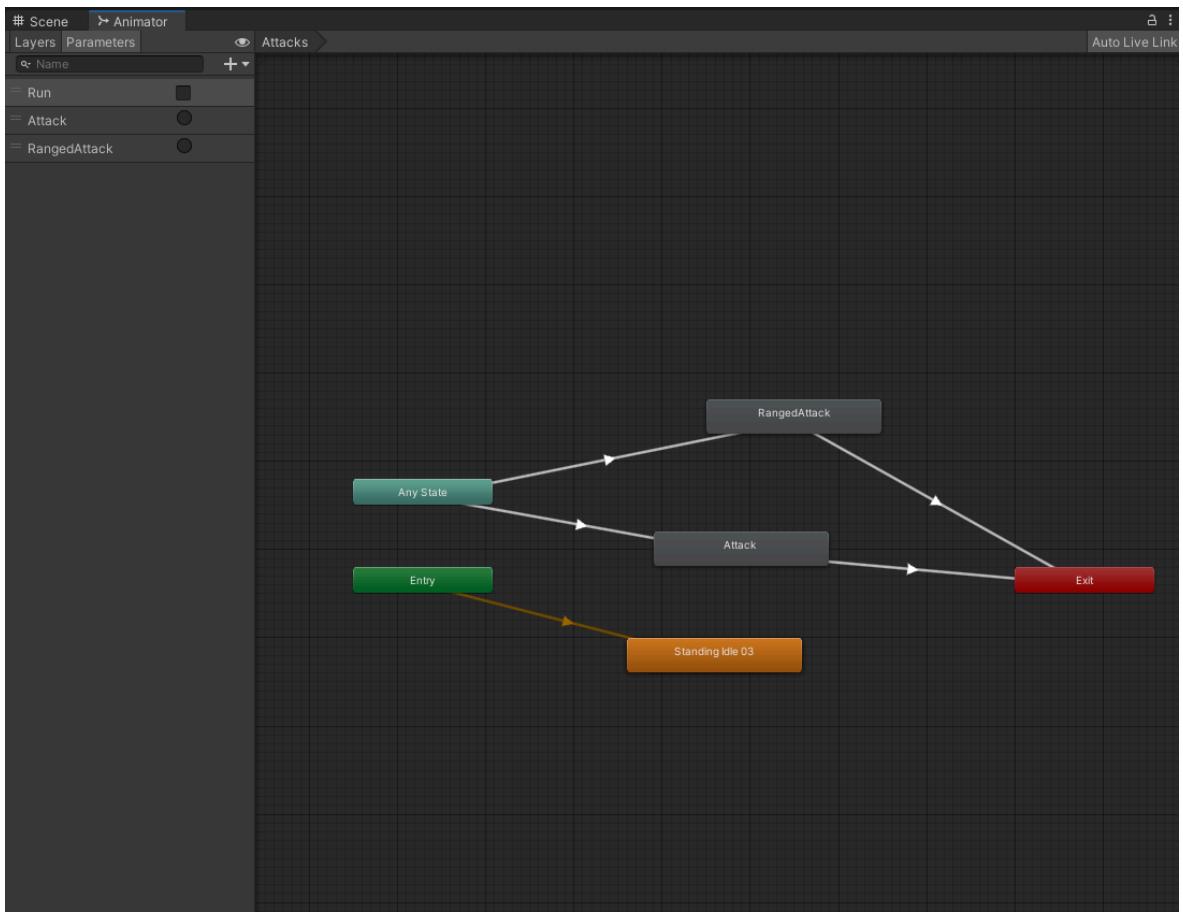
Por código: Invocando directamente la animación desde el código sobre un modelo 3D ayudándose del animator para identificar la animación.

Por variables en el *animator*: Cambiando el valor de las variables en el *animator* para indicar que tiene que haber un cambio entre una animación y otra, dejando que el propio *animator* haga el cambio y transición.

Animaciones aditivas

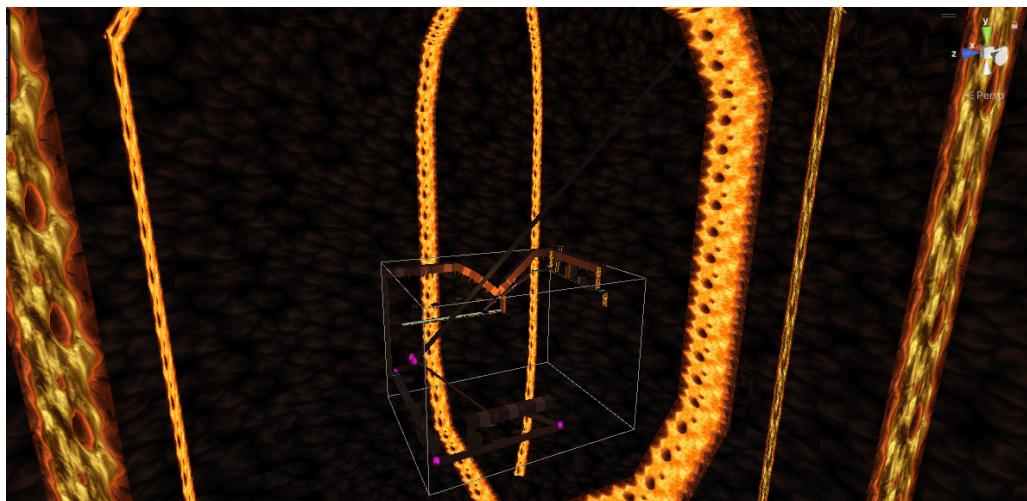
Se quiso que ciertas animaciones solo afectasen a cierta parte de un modelo 3D o que se fusionasen con otras animaciones para lograr una implementación más natural, por lo cual en el *animator* se tuvo que añadir una capa de animación extra que funcionase por encima de las animaciones que estuviesen activas en ese momento, para lograr que estas animaciones modificadas que solo afectaban a cierta parte del modelo, sobre escriban esa parte del modelo a la hora de iniciar dos animaciones a la vez.





Skybox

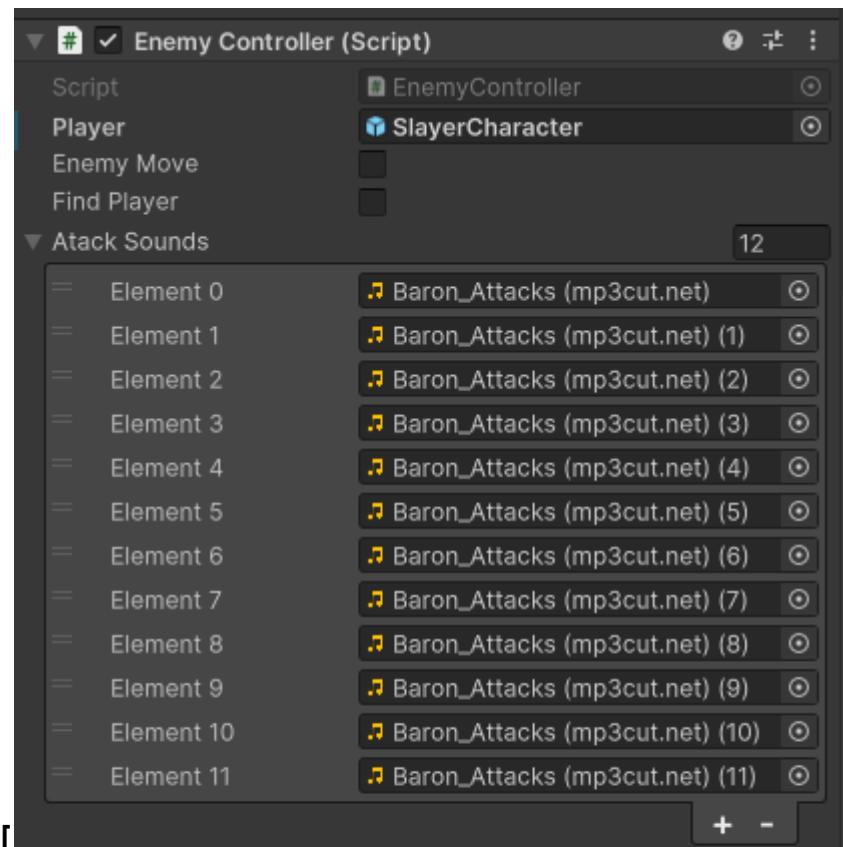
Para implementar un fondo al nivel se valoraron diferentes opciones entre ellas, una simple imagen con elementos 3D que seguía al jugador, pero un fondo estático no encajaba con la temática 3D, se probó a poner una *SkyBox* pero tampoco encajaba con la temática del juego ya que estéticamente era demasiado realista o demasiado caricaturesco, por lo que al final se optó por descargar una cueva grande, y personalizarla a nuestro gusto, esta cueva es un modelo 3D al cual se tuvieron que hacer modificaciones en *Blender* para cambiar las *Normales* del modelo 3D de esta manera las texturas aplicadas al modelo 3D se verían desde dentro en vez de por afuera.



SONIDO

Sistema de sonidos aleatorios para diversidad

A la hora de implementar los sonidos, nos dimos cuenta de que algunos sonidos (como los del jugador o los enemigos) se podían hacer muy repetitivos. Para evitar esto decidimos que a la hora de que se fuese a reproducir un sonido, el sonido a reproducir se escogiese de forma aleatoria dentro de un grupo de sonidos, evitando así la repetitividad que nos estaba dando problemas y mejorando la inmersión del juego.



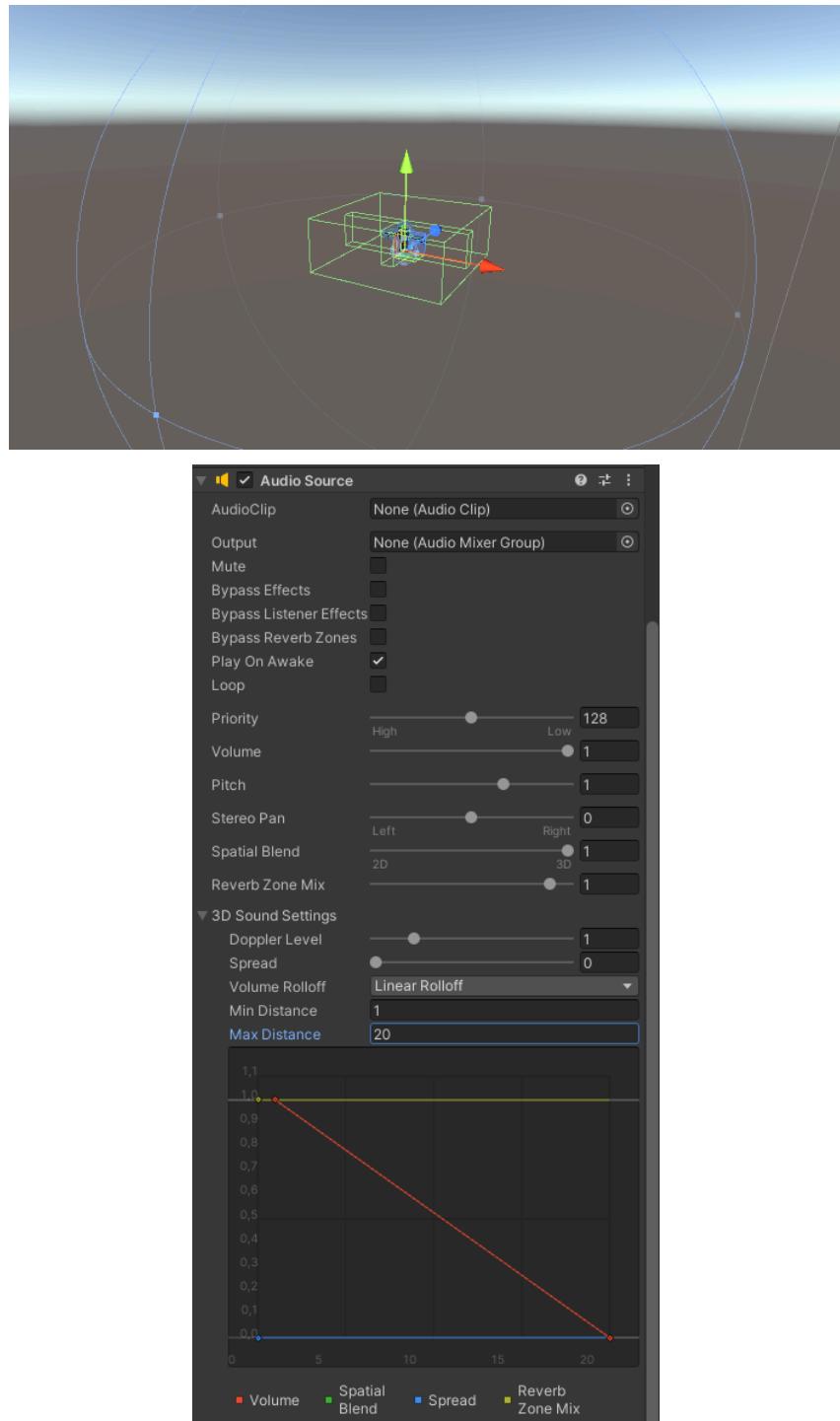
Música

La música implementada en el videojuego depende de donde se encuentre el jugador es decir dependiendo del menú sonara diferente música, dependiendo de qué parte del juega también cambiará esta, esto se logra midiendo la duración de las canciones y con *triggers* que cambian la musica que esta sonando en ese momento.

Sonido 3d de los enemigos

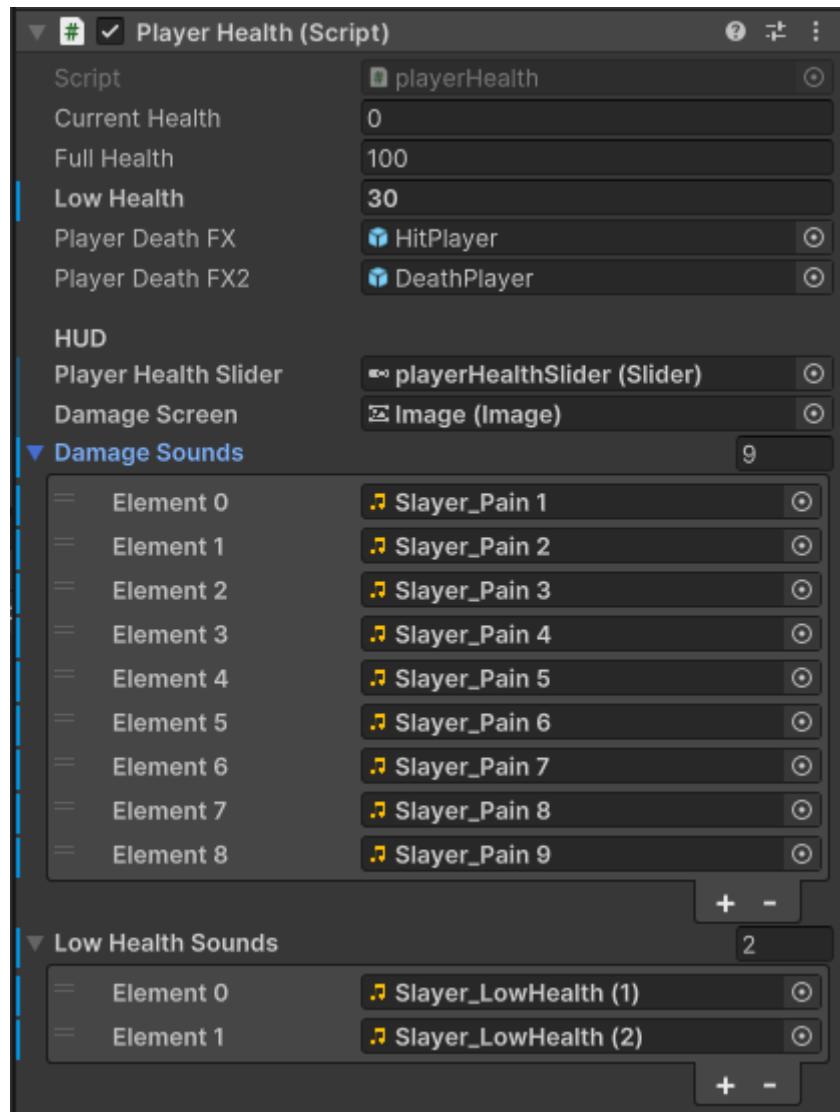
Para la creación de un entorno más realista se decidió que los sonidos de los enemigos sean 3D de manera que si estas a la derecha de un enemigo escuches el audio por el

altavoz izquierda y viceversa, también el usar audio 3D nos permite que dependiendo de la distancia del jugador a un enemigo los sonidos que este haga se escuchen más o menos altos, para poder implementar esto se tuvo que cambiar los *Audio Source* de los enemigos y crear una curva de sonido para decidir la deformación del audio en relación a la posición del personaje.



Sistema de sonido del jugador

Como tal el personaje del videojuego no tiene muchos sonidos a si que se hizo que estos sean reconocible y den información al jugador de lo que está sucediendo, para lograr esto se hizo que cada acción o estado importante del personaje fuese reflejada con diferentes sonidos, tales como al cambiar de arma, recibir daño, estar bajo en salud, recoger un arma, etc.



GAMEPLAY

Sistema de movimiento old

En el sistema de movimiento antiguo del videojuego, el personaje solo tenia las opciones de moverse de horizontalmente, saltar y realizar un salto en el aire, esto se lograba empujando el personajes en la dirección que queríamos y dejando que la propia

gravedad simulada por *unity* lo cual no daba apenas control de como reaccionaba el personaje a las acciones del jugador.

Sistema de movimiento new

Buscando un sistema de movimiento 2d para un juego de plataformas estilo *metroidvania* se ha implementado un *asset* que permite implementar módulos de movimiento a un personaje para que esta pueda realizar diferentes acciones(Deslizarse, Salto en la pared, nadar, agarrarse en los bordes, correr, salto doble, etc.), se modificó el core de este asset para que funcionase de manera correcta para un entorno 3D ya que estaba pensado para un entorno 2D con elementos 3D, entre los cambios que se tuvieron que hacer, fue adaptar todo el proyecto a un mundo sin gravedad ya que ahora el videojuego tiene gravedad de manera programática y no simulada, otro açambio en el asset fue el uso de animaciones para modelos 3D, estos cambios permiten que el jugador tenga un control más extenso del movimiento del personaje y pueda ir desbloqueando nuevos módulos para aumentar las capacidades de movimiento del personaje.

Gracias a la implementación de este asset ahora se tiene mucho más control sobre como se mueve el personaje y las opciones de movimiento son mayores.

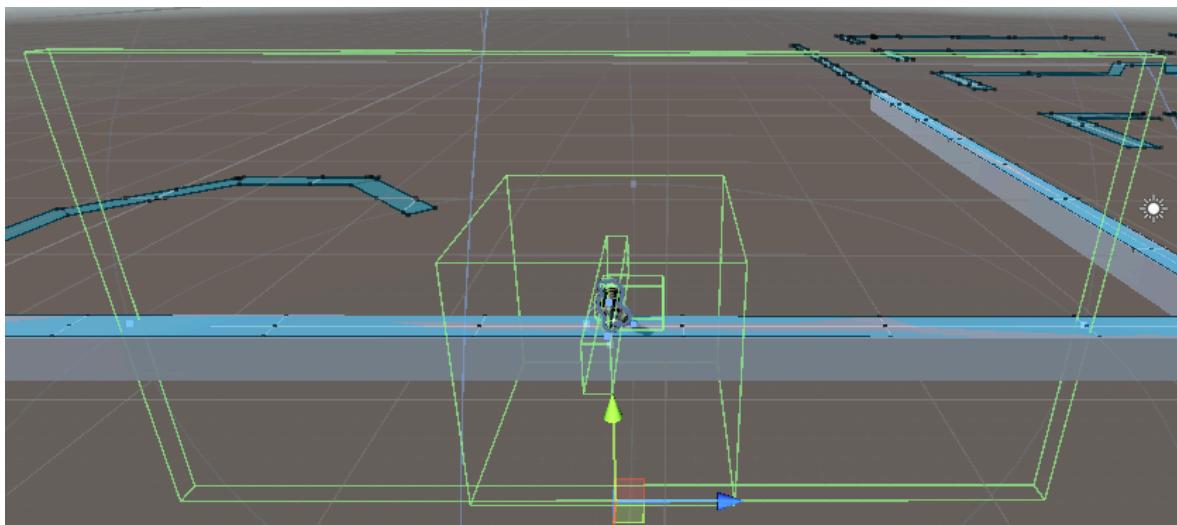
Enemigos old

La inteligencia de los enemigos en la versión anterior se basaba en dos *colliders*, uno se encarga de detectar al jugador y empujar el enemigo en el mismo eje en el que se encuentra el jugador y el otro simplemente que cuando el jugador entre en este sufra daño, de esta manera los enemigos solo se desplazan horizontalmente y hacen daño al jugador por contacto, lo cual no permite muchas variaciones de enemigos.

Enemigos new

Con la implementación del nuevo *pathfinding* se decidió cambiar completamente como funcionaba la inteligencia de los enemigos, ya que ahora estos tienen un *NavMeshAgent* que les dice por donde pueden moverse del mapa y seguir al jugador, también ahora tienen diferentes módulos para los ataques y acciones de estos(ataques cuerpo a cuerpo, ataques a distancia, habilidades, etc.) de esta manera la creación de un nuevo enemigo solo requiere colocar los valores deseados(vida, velocidad, daño,etc), animaciones, modelados correctos y los módulos deseados.

La creación de estos módulos de comportamiento se basan en un *collider* que detecta al jugador cuando entra en este y esto activa una animación con un comportamiento tras esta para ya sea hacer daño al jugador, lanzarle un proyectil, etc.

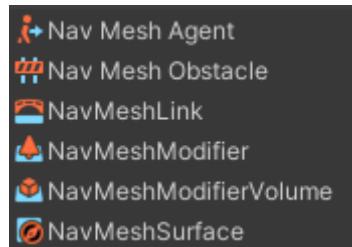


Detección jugador por enemigos

Se basa en un *collider* que al entrar en contacto con el jugador activará el *meshAgent* del enemigo y este comenzará a seguirlo siguiendo la lógica del *meshAgent*, de manera que una vez el enemigo haya detectado al jugador pueda seguirlo a donde su pathfinding le permita, brindando la posibilidad de que si el jugador se sale de este *collider* el enemigo tenga la opción de dejar de seguirlo si así se desea.

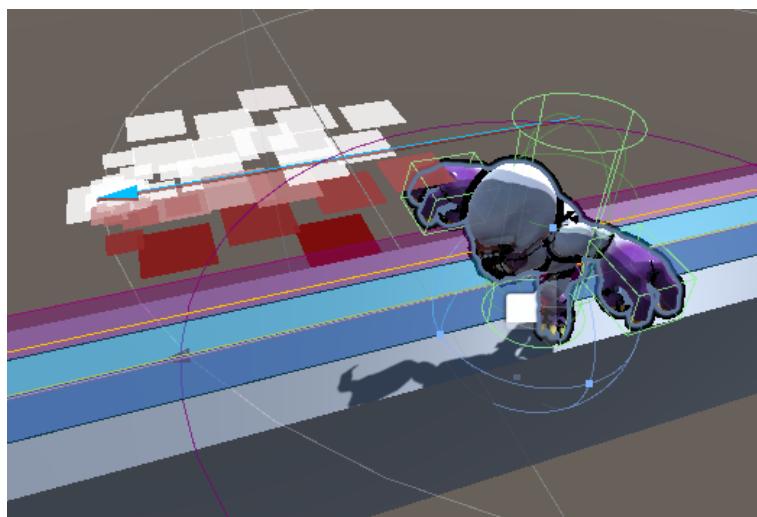
Pathfinding nuevo

Para el nuevo sistema encargado de que los enemigos localicen y sigan al jugador se encontró una extensión de Unity que permite trabajar con nuevos componentes, estos componentes son:



- **Nav Mesh Agent**

Encargado de controlar y definir un elemento, dándole una estructura para que sepa por qué partes del mapa puede moverse, la velocidad a la que lo hace, etc. También ejecuta el movimiento de este elemento por el mapa, su objetivo y la eficiencia a la que se mueve por el entorno.



- **Nav Mesh Obstacle**

Encargado de limitar las partes del mapa por las cuales puede moverse un *Mesh Agent*.

- **NavMeshLink**

Permite de manera manual unir dos partes del mapa que no están unidas, básicamente habilitar nuevas opciones de movimiento a un *Mesh Agent*.

- **NavMeshModifier**

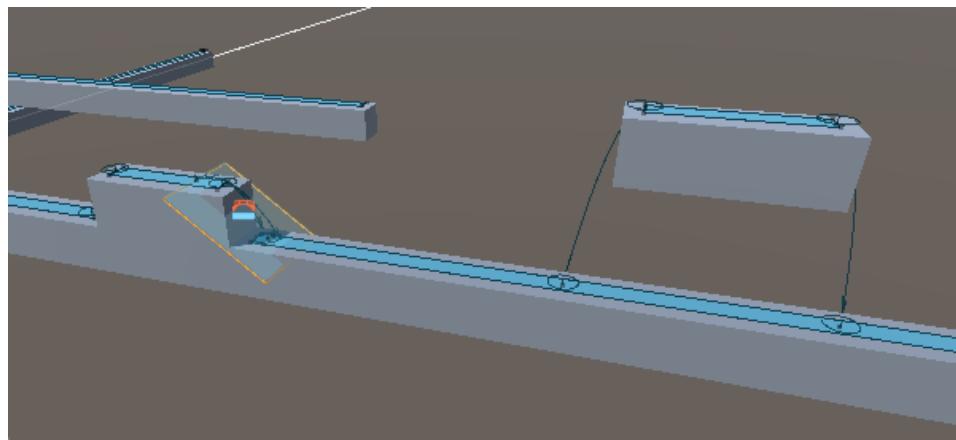
Permite que a la hora de generar las diferentes partes por las que un *Mesh Agent* puede moverse estas sean tratadas de manera diferente.

- **NavMeshModifierVolume**

Permite definir ciertas áreas del mapa de manera diferente para que el *Mesh Agent* interactúe de diferente manera con estas.

- **NavMeshSurface**

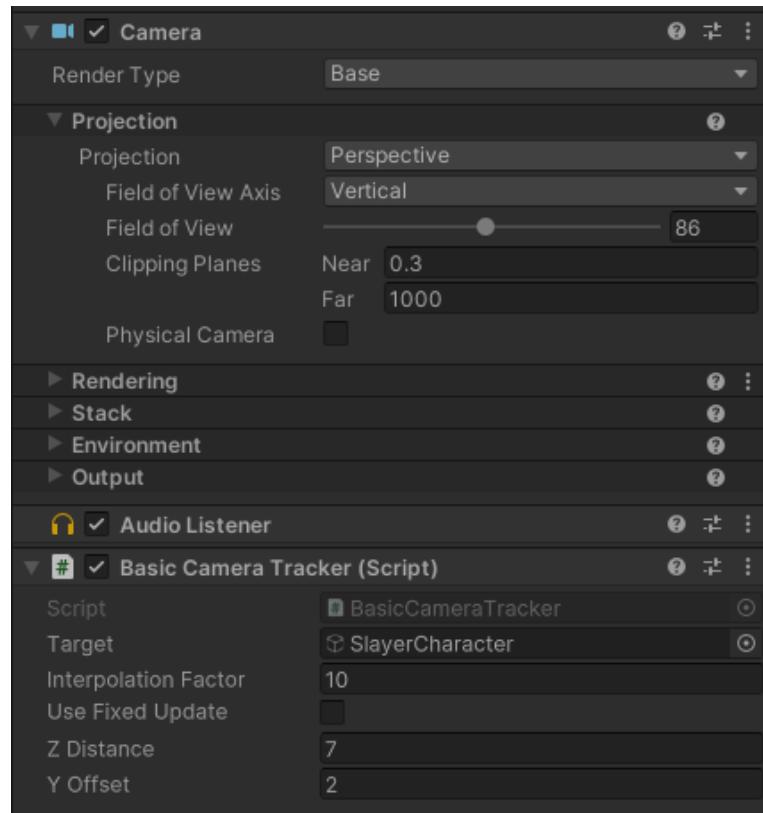
Se encarga de generar las diferentes partes por las que un *Mesh Agent* puede moverse basándose en parámetros del propio *Mesh Agent* y diferentes variables del mapa.



Con estos componentes podemos crear “perfils” (*Mesh Agents*) para definir que enemigos pueden moverse por las diferentes partes del mapa, su velocidad, cuanto espacio ocupan, con que eficiencia se mueven por este, etc.

Cámara que sigue al jugador, fov, etc

Para la creación de una cámara dinámica, se optó por usar una que contase con múltiples opciones y que siguiese al personaje por el escenario, para lograr movimientos más naturales. Para lograr esto se hizo que la cámara no estuviese dentro de ningún componente del personaje y que en cambio fuese registrando constantemente la posición del personaje para después ir a esta localización con una velocidad dada, también la cámara tiene la opción de elegir el *fov* para tener más opciones a la hora de como jugador personalizar la experiencia de juego.



PickUps

Para los PickUps (Objetos del mapa recolectables), hemos decidido crear un tipo de objeto del mapa que tenga un *Collider* que detecte si el jugador está dentro del objeto. Cuando el jugador esté dentro del objeto se ejecutará el script correspondiente a lo que se quiera hacer. Nosotros tenemos 2 tipos de PickUps:

- Vida: Cuando se atraviesa este objeto el jugador gana una cantidad fija de vida
- Arma: Cuando se atraviesa este objeto se detecta que arma es y se cambia en el inventario la propiedad booleana para que la pueda usar



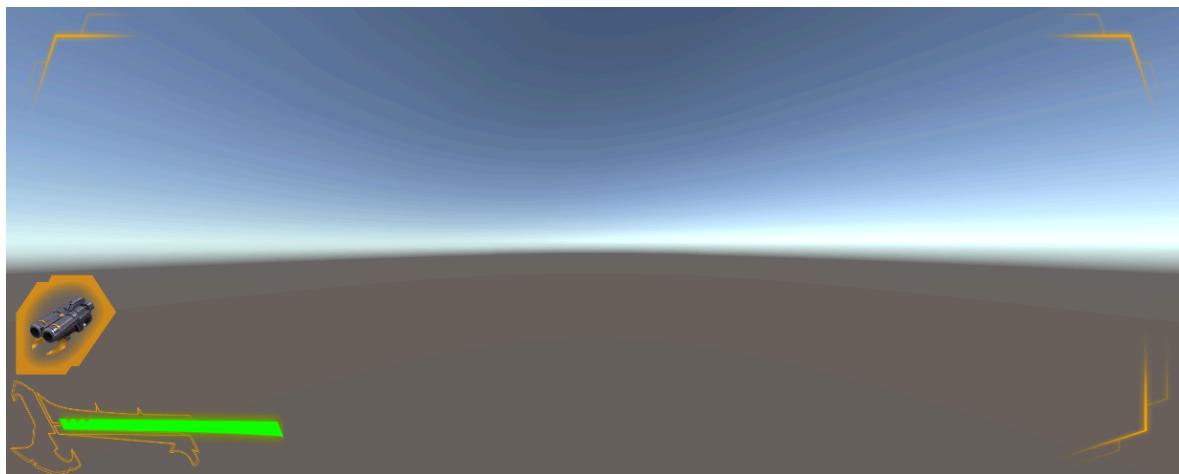
HUD dinámico

Para el HUD (Head-Up Display) queríamos hacer algo útil que diese buena información en tiempo real al jugador sin que fuese demasiado cargante o invasiva.

Para ello decidimos implementar los siguientes elementos:

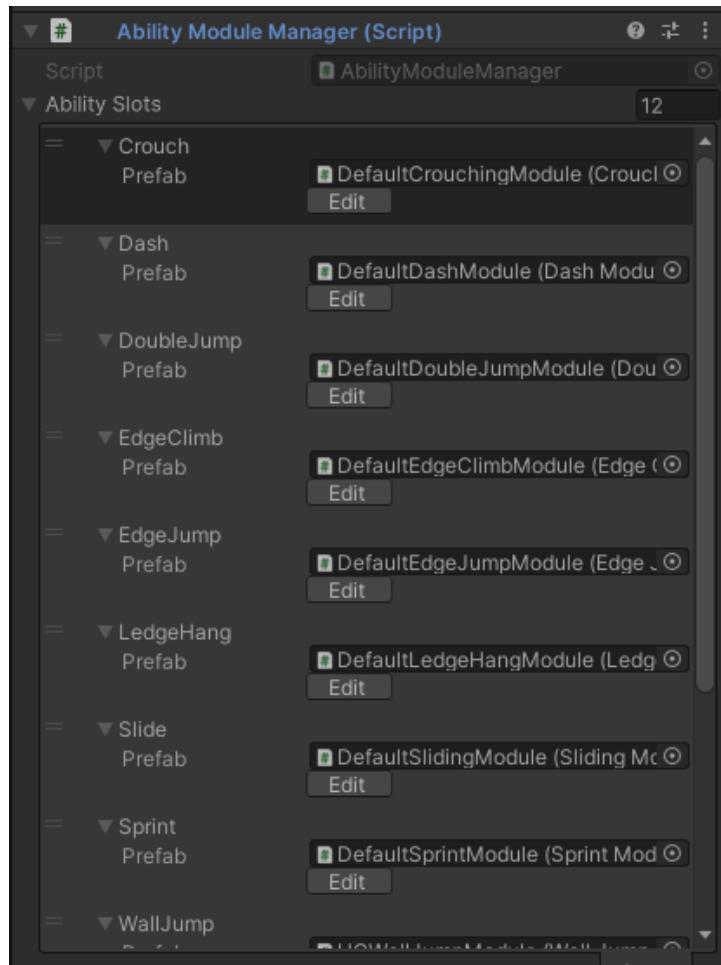
- Barra de vida: Una barra simple que muestra la cantidad de vida del jugador de forma porcentual. La sección verde muestra la vida actual mientras que la sección roja muestra la vida que le falta.
También está la barra de vida del jefe que muestra su nombre y muestra la vida actual de color rojo y la que le falte de blanco.
- Arma actual: Es un recuadro con una imagen del arma que se esté usando actualmente. Si el arma cambia esta imagen también lo hace.
- Detalles estéticos: Pequeños detalles que dan estilo y ayudan a guiar al jugador a los elementos de interés.





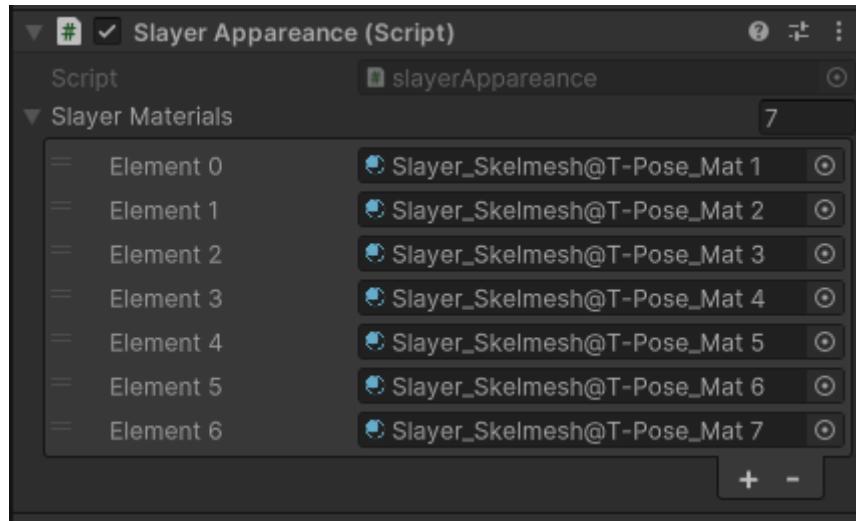
Jugador con habilidades modulares (Revisar)

En el mismo asset que el de movimiento venían opciones para incorporar distintas habilidades al jugador. Tras adaptarlo para que funcionase en nuestro entorno y con nuestro personaje pasamos a tener un jugador que podía tener distintas habilidades (Salto, doble salto, deslizarse, aceleron...) que se pueden activar y desactivar de forma independiente. Esto nos permitió, tras implementar las animaciones, conseguir ese grado de fluidez en el movimiento que estábamos buscando lo que definió como iba ser el diseño del mapa y la interacción con el entorno.



Skins

Para el modelo del jugador encontramos un paquete con varias texturas para el jugador. En principio pensamos que podrían ser desbloqueables al conseguir ciertos retos en el juego pero como sabíamos que no íbamos a tener tiempo se selecciona una de forma aleatoria cada vez que se inicia el juego.



Animaciones del jugador adaptables a lo que suceda

El sistema de animaciones que tenemos se basa en el elemento *Animator* de Unity, en el que se aplican las animaciones que se tengan de un modelado 3D en función a las condiciones que se establezcan, que pueden ser de distintos tipos.

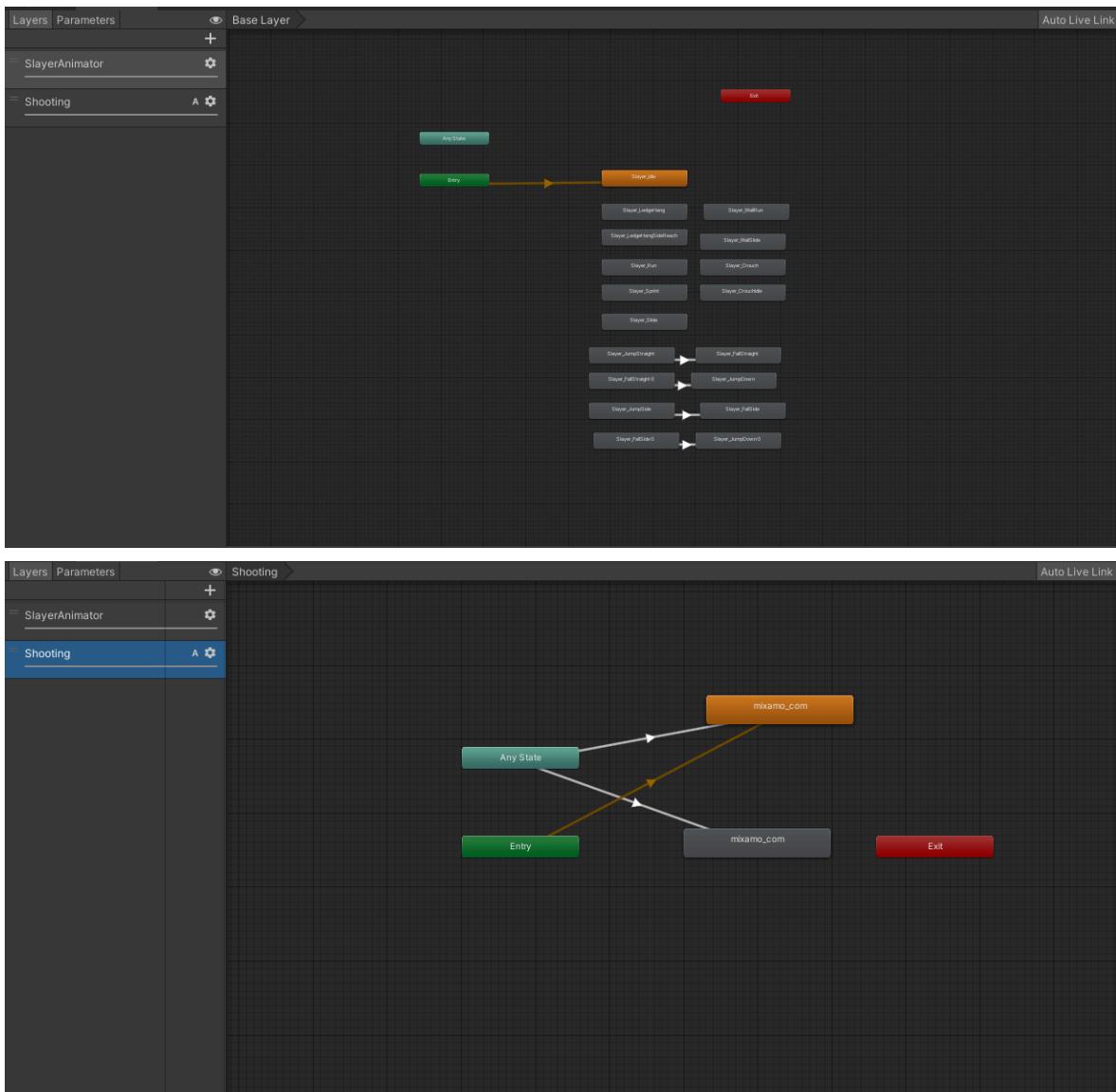
El *Animator* parte de una animación base, en nuestro caso la de reposo, y en función a las condiciones que se cumplan la animación cambia de una a otra. Normalmente un cambio así sería muy brusco, pero Unity tiene una herramienta ya implementada que hace transiciones entre animaciones de forma automática.

El cambio también se puede producir desde la parte de código, desde donde se pueden llamar a las distintas animaciones.

El problema llegaba cuando el modelado tenía que hacer varias cosas a la vez, como moverse y recibir daño. El *Animator* cuenta con herramientas para solucionar eso.

Para solucionar este problema, Unity cuenta con la posibilidad de crear varias capas de animaciones, a las que se pueden asignar propiedades específicas además de poder animar parcialmente los modelados.

De esta forma podemos darle a una capa una propiedad aditiva para que se anime por encima de otra animación y seleccionar la parte que nos interese animar del modelado.



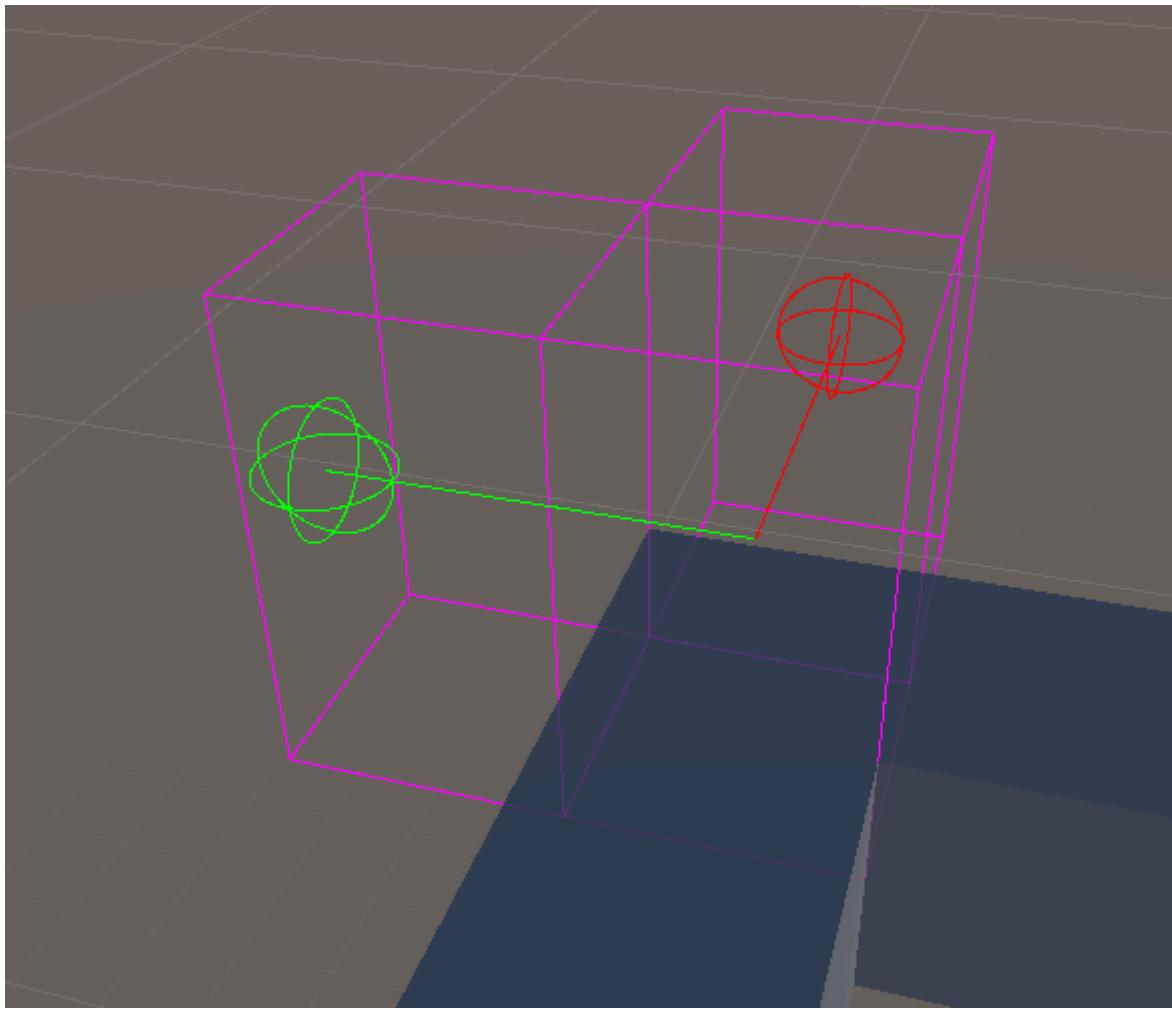
Mapa que hace cosas(world turn)

Para el diseño del mapa, como estábamos trabajando en un entorno 3D pero el movimiento del personaje solo le permitía avanzar a la izquierda o a la derecha, pensamos en hacer algo para que el nivel fuese un poco más interesante.

Para ello, ya que la ambientación del juego iba a ser un volcán, se decidió que el jugador pudiese girar por esquinas, aprovechando el entorno 3D y aumentando la sensación de profundidad del mapa.

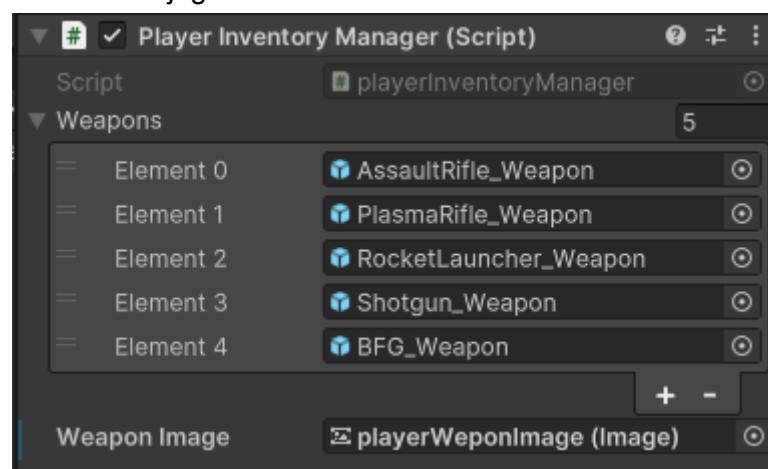
Para hacer esto, como el personaje no puede girar como tal debido a como está diseñado el juego en Unity, lo que se hace es girar el mapa manteniendo al jugador quieto.

Esto se hace poniendo una zona de detección en la esquina para que el mundo gire y se ponen otras dos los límites de los caminos para que, si el jugador se pasa de la zona de detección, este sea reubicado mirando a la dirección que le corresponda



Inventario jugador

El funcionamiento del inventario es una lista llena de los elementos que el jugador puede obtener, que tienen una propiedad booleana que determina si el jugador puede utilizarlo o no. Los únicos objetos que se pueden obtener son armas así que el inventario determina qué armas puede utilizar el jugador.



Físicas old

En la anterior versión del videojuego se quería mantener la simulación de gravedad proporcionada por *Unity* por lo que todos los elementos del videojuego la tenían por lo que en diferentes situaciones tenemos que contrarrestar esta gravedad simulada para conseguir los efectos deseados en los diferentes elementos.

Físicas new

Con el cambio de sistema de movimiento se decidió que la mayoría de elementos del videojuego careciesen de gravedad simulada para hacerla de manera programática, de esta manera podíamos controlar mejor como responden a las acciones del jugador.

Ataque melee old

El anterior sistema de ataque a melee funcionaba de tal forma que el personaje del jugador tenía un *collider* en un brazo que al realizar la animación asignada al ataque hacía daño a los enemigos por contacto y los empujaba en la dirección opuesta.

En la primera iteración de la mecánica había un bug en el que solo empujaba hacia delante si el enemigo se encontraba en un valor positivo del eje X, de lo contrario lo empujaba en la dirección del jugador. Esto se debía a que al impactar, se aplicaba una fuerza para empujar al enemigo, pero si la posición era negativa, la impulsaba en la dirección opuesta.

Game controller

El game controller es un elemento del mapa encargado de cargar y controlar ciertos elementos del mapa, tales como el audio de fondo, pudiendo cambiar la música de fondo a gusto, eliminar objetos que se salgan del mapa jugable para ahorrar recursos, de esta manera si el jugador se cae del mapa a este se le reducirán los puntos de vida a cero forzando la aparición y en el caso de cualquier otro elemento éste será directamente destruido.

Boss fight

Para el enemigo final se pensó en un principio que se cambiase el género del juego para hacerlo más interesante como que el boss fuese como un juego de lucha o un JRPG. Al final por falta de tiempo se ha optado por mantener la jugabilidad y utilizar para el boss al Icon of Sin (Icono del Pecado) que fue el jefe final del videojuego Doom 2. Para ello nos hemos inspirado en el primer jefe del NieR:Automata en el que el boss es muy grande y ataca a la plataforma en la que nos encontramos. Para poder derrotarlo se tendrán que esquivar los ataques y atacar de vuelta, ya que solo será vulnerable mientras está atacando.

Además, para hacerlo más interesante, se ha optado por utilizar la tecnología de WorldTurning explicada anteriormente para que el jugador de vueltas al boss y de una mejor sensación de dinamismo así como facilitar que el jugador pueda esquivar con más fluidez.



El funcionamiento de la pelea se basa en los siguientes puntos:

Detección:

Se ha delimitado un *Collider* con la propiedad *Trigger* para que el jefe pueda detectar si el jugador se encuentra en un perímetro determinado alrededor suyo. Si el jugador está fuera permanecerá en reposo, si se encuentra dentro empezará a atacar. También se impedirá que el jugador pueda volver fuera de la sala del jefe eliminando el camino por el que ha venido. Junto con todo esto también aparecerá una barra de vida horizontal en el *HUD* del jugador para poder ver en todo momento los puntos de vida restantes del jefe.

Ataques:

Cuando el jefe ha detectado al jugador empezará a llamar a las animaciones de ataque que hemos definido de forma aleatoria.

Para que hagan daño le hemos puesto *Colliders* en los brazos que tienen la propiedad *Trigger*. Cuando detectan que el jugador está dentro le reduce los puntos de vida y lo empuja en la dirección opuesta

Vida:

Para su vida, como con los enemigos normales, tiene una cantidad fija de vida que se reducirá cuando las balas del jugador impacten con *Colliders* específicos que tengan la propiedad de recibir daño. En este caso el jefe tiene *Colliders* en la cabeza y en ambos brazos, el resto del cuerpo al no estar expuesto no los necesita.

Además cuando recibe daño se llama a un sonido aleatorio dentro de una array de sonidos de daño.

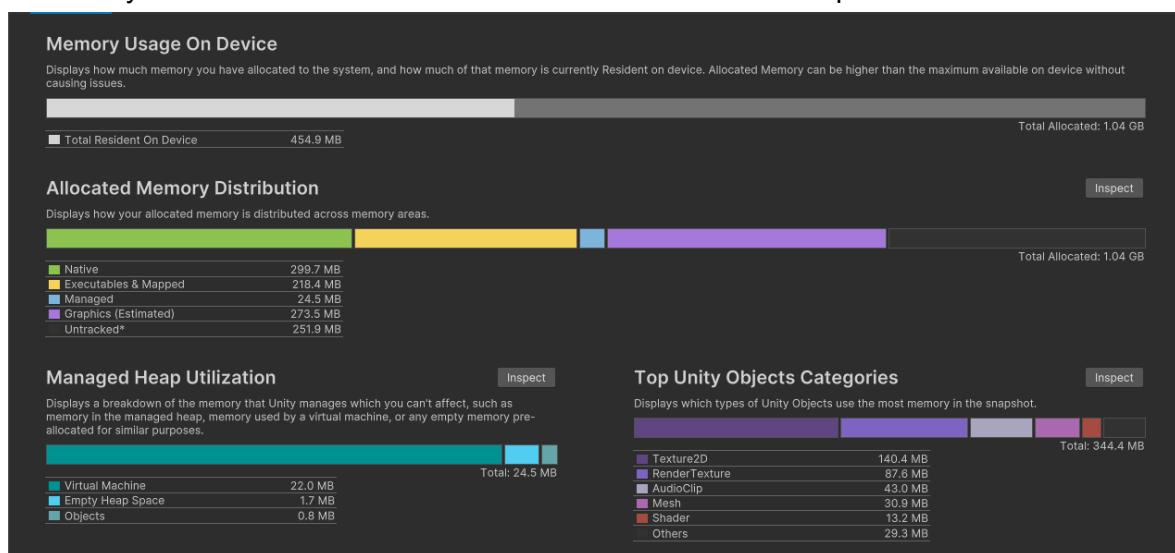
Muerte:

La vida se reduce a cero, ocultamos al jefe, se llama a un sonido de muerte del jefe, se dispara una animación de trozos de carne y sangre y se habilita una rampa para que el jugador pueda continuar.

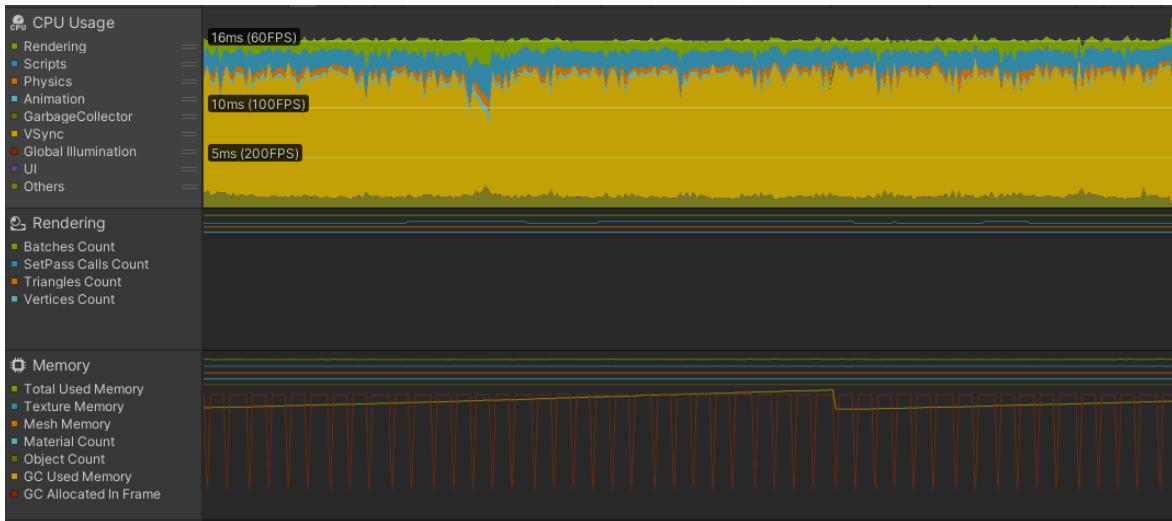
RENDIMIENTO

Para evaluar el rendimiento del juego hemos utilizado un paquete de Unity llamado “Memory Profiler” que se encarga de calcular el uso de recursos que está consumiendo el juego en un momento específico.

Con esta información se nos muestra un desglose de todo lo que esté consumiendo recursos y cuántos recursos consume además de los recursos que tiene reservados.



Esto junto con la herramienta predeterminada de Unity llamada “Profiler” nos permiten ver a fondo desde un ejecutable de desarrollo toda estadística relacionada con los recursos que nos pueda interesar en tiempo real.



CONCLUSIONES

Este proyecto ha representado un desafío no solo a nivel de programación, sino también de diseño, aprendizaje, gestión y organización, lo que nos ha obligado a salir de nuestra zona de confort y aprender a utilizar una variedad de herramientas de ámbitos diferentes.

Esto sobre todo se ha visto en la complejidad de abordar los diversos aspectos de un videojuego (video, audio, animaciones, efectos visuales, etc.) que ha requerido no solo aprender nuevas habilidades, sino también buscar métodos eficientes y prácticos para adquirir estos nuevos conocimientos y maximizar el tiempo disponible.

También nos hemos dado cuenta de que es crucial mantener la constancia y la diligencia al enfrentar proyectos de esta magnitud, ya que son altamente complejos y requieren una gran inversión de tiempo. Afortunadamente, hay abundantes recursos en internet que nos han sido de gran ayuda para establecer una base sólida sobre la cual desarrollar nuestros objetivos.

A raíz de esto nos hemos dado cuenta de la importancia de una distribución eficiente del trabajo para que sea agradable para todas las partes involucradas en la creación de un gran proyecto. Dado que se dedican muchas horas a lo mismo, es necesario que cada vez que uno se pone a trabajar en el proyecto, lo haga con entusiasmo y en algo que le resulte interesante para evitar la monotonía.

Finalmente, es importante saber cuándo detener la implementación de nuevas ideas en un proyecto de este calibre. Un gran número de funcionalidades en un mismo proyecto puede ser problemático si se intentan implementar todas a la vez. Siempre surgen nuevas ideas, pero no todas pueden ser implementadas, ya que el tiempo es un factor limitante y si no se dedica el tiempo necesario a cada idea, no solo la calidad de esta se verá afectada sino también la del resto del proyecto.

REFERENCIAS BIBLIOGRÁFICAS:

Modelados 3D. (25 de Diciembre 2023). En Sketchfab.

<https://sketchfab.com/feed>

Animaciones 3D. (25 de Diciembre 2023). En Mixamo.

<https://www.mixamo.com/#/>

Assets Unity. (10 de Enero 2024). En Unity.

<https://assetstore.unity.com>

Guia básica de Unity. (28 de Diciembre 2023). En Youtube.

<https://www.youtube.com/@unity/search>

<https://www.youtube.com/@mixandjam>

Como crear un videojuego en 2.5. (28 de Diciembre 2023). En youtube.

https://www.youtube.com/watch?v=EF_MPhDTFcE&list=PL2cNFQAw_ndx1ID-DuvFoeGR_nE4jOFTq&index=2

https://www.youtube.com/watch?v=5OkJ7Mb6QXo&list=PLWYGofN_jX5BupV2xLjU1HUVujl_yDIN6&index=27

https://www.youtube.com/watch?v=amXOLa0ZXcg&list=PLgPc_vfSLFYVVKxqUn-HrkHAS-8P8S-Wk&index=21

https://www.youtube.com/watch?v=4Blt4_U2rlU&list=PLYmJZ5mcI7OT3wUwLuUzp5Cjq32tU8V4a

<https://www.youtube.com/watch?v=sNp7sN7bhWA&list=PLLH3mUGkfFCWwekOW1OMxyylgc-Qm1OhI&index=11>

Tips generales. (5 de Enero 2024). En youtube.

<https://www.youtube.com/watch?v=PECZS26lgtM&list=PLTdG95U6i2G4Ls2kvI5ho0bUIN5zJeE1G&index=1>

<https://www.youtube.com/watch?v=zpISVReNg-A&list=PLTUZMVrECom107SzgvrQkJUAcm6pUVq8t>

Audios de los enemigos. (10 de Febrero 2024). En youtube.

<https://www.youtube.com/watch?v=ljoYXX0V09U&list=PL4sG3ZyWkBjAxptz4DOfcJSzUQUTfoBH&index=21>

Audios de las armas. (20 de Febrero 2024). En youtube.

<https://www.youtube.com/watch?v=rHNch6NtEys>

<https://www.youtube.com/watch?v=srvWTkHQyNw>

https://www.youtube.com/watch?v=u_Ni0BvADFo

Audios del protagonista. (24 de Febrero 2024). En youtube.

<https://www.youtube.com/watch?v=8jPPHxAYWwQ>

<https://www.youtube.com/watch?v=PdcHICqBHQc>

Elementos del HUD. (25 de Febrero 2024). En Sprites-Resource.

https://www.sprites-resource.com/pc_computer/doometernal/sheet/159871/

Otros modelos 3D e imágenes. (10 de Marzo de 2024). En Deviantart.

<https://www.deviantart.com/wargrey-sama/art/Mighty-DOOM-MODELS-DOWNLOAD-EVERYTHING-SO-FAR-888663301>

Diferentes tutoriales. (15 de Marzo de 2024) En youtube.

<https://www.youtube.com/watch?v=U9Ah7wnLhYE&list=PLaYjZPP4Xm6GclWDhwuJJuCG1mwkMK9uX&index=6>

https://www.youtube.com/watch?v=zFe77GJs4EQ&list=PLNEAWvYbJJ9ltV9VYRjrX0_E098SZny7k

https://www.youtube.com/watch?v=peskzx_5x7A&list=PL0c9rHNEUHlyryuY0PvipHTXyL2mBij9-&index=5

[https://www.youtube.com/watch?v=y96ADRAmixc&list=PLNEAWvYbJJ9lapw_S_qn9v42gtনaf11TI&index=1](https://www.youtube.com/watch?v=y96ADRAmixc&list=PLNEAWvYbJJ9lapw_S_qn9v42gt�af11TI&index=1)

https://www.youtube.com/watch?v=3GtgsTCbVqE&list=PLC7nmYI-cbT04xcL_VY6QkB67h9xgrKTx

https://www.youtube.com/watch?v=-rpo-RLRs0E&list=PLgPc_vfSLFYVVKxqUn-HrkHAS-8P8S-Wk&index=8

DICCIONARIO

Unity: Es una plataforma de desarrollo de videojuegos y motor gráfico líder en la industria, utilizada para crear experiencias interactivas en 2D y 3D. Ofrece un conjunto de herramientas y servicios que permiten a los desarrolladores diseñar, construir y desplegar juegos y aplicaciones interactivas en múltiples plataformas, incluidas consolas, computadoras, dispositivos móviles y realidad virtual/aumentada.

Run and Gun: Es un subgénero de los videojuegos de acción en el que los jugadores deben avanzar continuamente a través de niveles mientras se enfrentan a oleadas de enemigos usando armas de fuego. Este estilo de juego combina elementos de plataformas y disparos, requiriendo habilidades tanto de movimiento ágil como de puntería precisa.

Scroll lateral: Es un estilo de juego en el que la acción principal se desplaza horizontalmente a través de la pantalla, generalmente de izquierda a derecha. En este tipo de juegos, el jugador controla un personaje o un vehículo que se mueve a través de niveles lineales o semi-lineales mientras la cámara sigue su movimiento, proporcionando una sensación de progreso continuo a lo largo del escenario.

GitHub: Es una plataforma de desarrollo colaborativo que utiliza el sistema de control de versiones Git. Permite a los desarrolladores gestionar y almacenar su código fuente, colaborar con otros desarrolladores, y mantener un historial detallado de los cambios realizados en sus proyectos. GitHub proporciona herramientas para la revisión de código, seguimiento de errores y gestión de proyectos, facilitando así la colaboración eficiente en equipos de desarrollo.

LFS: (Large File Storage) es una extensión para Git diseñada para manejar archivos grandes y binarios de manera eficiente. En lugar de almacenar los archivos grandes directamente en el repositorio Git, Git LFS guarda referencias a esos archivos y los almacena en un servidor separado. Esto permite mantener el tamaño del repositorio más manejable y mejora el rendimiento de las operaciones de Git.

Prefabs: Es una plantilla o prototipo de un GameObject que puede ser reutilizado en múltiples lugares a lo largo del proyecto. Un Prefab guarda una copia completa del estado de un GameObject y sus componentes, permitiendo que los desarrolladores instancien copias de ese GameObject en la escena con todas sus propiedades y configuraciones predefinidas.

Canvas: Es un componente fundamental del sistema de interfaz de usuario (UI). Actúa como el contenedor principal para todos los elementos de la interfaz de usuario, como botones, imágenes, textos y otros controles interactivos. El Canvas asegura que estos elementos se rendericen y se comporten de manera coherente en la pantalla.

Colliders: Es un componente que define la forma y el área de un objeto para las interacciones físicas y de colisión. Los Colliders se utilizan para detectar colisiones y desencadenar eventos físicos en el juego.

Trigger: Es una propiedad de un componente Collider que permite que dicho Collider detecte cuando otros objetos entran, permanecen o salen de su volumen, sin que ocurra una colisión física. Los Triggers se utilizan principalmente para activar eventos o interacciones en el juego cuando los objetos interactúan con ellos.

Script: Es un archivo de código que define el comportamiento de los GameObjects en una escena. Los scripts se escriben utilizando lenguajes de programación como C# y permiten a los desarrolladores implementar la lógica del juego, manipular componentes, responder a eventos y controlar las interacciones del juego.

PickUps: Son objetos dentro de un videojuego que el jugador puede recoger para obtener ciertos beneficios, como puntos, salud, municiones, habilidades especiales, o cualquier otra mejora que contribuya al progreso en el juego.

Assets: Son los elementos fundamentales que componen un proyecto de desarrollo de videojuegos. Estos pueden incluir recursos como modelos 3D, texturas, audio, scripts, animaciones, materiales, prefabricados y más.

Universal Render Pipeline: Es un sistema de renderizado altamente personalizable y optimizado diseñado para ofrecer gráficos de alta calidad en una amplia variedad de plataformas, desde dispositivos móviles hasta consolas y PC de gama alta.

Classic: Es el sistema de renderizado original de Unity que ha sido utilizado durante muchos años. Aunque sigue siendo compatible, se considera obsoleto en comparación con las opciones más modernas y optimizadas como el Universal Render Pipeline (URP) y el High Definition Render Pipeline (HDRP).

Cell Shading: También conocido como Toon Shading, es una técnica de renderizado utilizada en gráficos por computadora para simular un estilo visual

similar al de las ilustraciones de dibujos animados o cómics. En lugar de producir sombras suaves y graduales como en el renderizado tradicional, el Cell Shading produce sombras y luces con bordes definidos y colores planos, dando al objeto renderizado una apariencia estilizada y simplificada.

Animator: Es una herramienta poderosa que permite controlar y gestionar las animaciones de los GameObjects en tiempo real. Utilizando un sistema de estados y transiciones, el Animator permite crear y controlar la reproducción de animaciones de manera dinámica, respondiendo a las acciones del jugador, eventos del juego y variables de estado.

SkyBox: Es una técnica utilizada en gráficos por computadora para simular el fondo del cielo y el horizonte en entornos tridimensionales, como videojuegos y aplicaciones de realidad virtual. Consiste en una textura envolvente o un conjunto de texturas que rodean al escenario del juego desde todos los lados, creando la ilusión de un horizonte lejano y un cielo infinito.

Blender: Es una potente suite de creación 3D de código abierto y multiplataforma que ofrece una amplia gama de herramientas para modelado, animación, renderizado, simulación, composición y edición de video. Con una comunidad activa de desarrolladores y usuarios, Blender se ha convertido en una de las herramientas más populares y versátiles para la creación de contenido digital en 3D.

Normales: Son vectores utilizados para definir la orientación de las superficies de los polígonos que componen el modelo. Cada vértice de un polígono tiene asociado un vector normal que apunta perpendicularmente desde la superficie del polígono en la dirección de su cara. Las normales se utilizan en renderizado 3D para calcular cómo la luz interactúa con la superficie del modelo, lo que afecta la apariencia visual de la misma.

Audio Source: Es un componente utilizado para reproducir sonidos en un juego. Puede reproducir una variedad de tipos de archivos de audio, como archivos WAV, MP3, OGG, y más, permitiendo a los desarrolladores incorporar efectos de sonido, música de fondo, diálogos de personajes y otros elementos auditivos en sus juegos.

Pathfinding: Es un concepto fundamental en el desarrollo de videojuegos y aplicaciones interactivas que implica encontrar el camino más eficiente entre dos puntos en un entorno tridimensional o bidimensional.

Fov: Es un parámetro utilizado en gráficos por computadora para definir el ángulo de visión que una cámara o un jugador tiene en un entorno virtual. Representa el área angular en la que los objetos y elementos del entorno son visibles desde el punto de vista de la cámara o del jugador, determinando qué tan amplio o estrecho es el rango de visión.

HUD: Es una interfaz gráfica superpuesta en la pantalla del jugador para proporcionar información relevante durante el juego sin obstruir la vista principal del mundo virtual.