Universitat de Girona
Escola Politècnica Superior

Trabajo de fin de grado

Estudios: Grado en Diseño y Desarrollo de Videojuegos

Título: Desarrollo de un juego de aventuras con entornos procedurales

Documento: Memoria

Alumnos: Denís Maseda Ares
Miquel Comasòlivas Vellisco

Tutor: Gustavo Patow
Departament: Informàtica, Matemàtica Aplicada i Estadística
Àrea: Llenguatges i Sistemes Informàtics

Convocatoria (mes/any) Junio/ 2023

1. Introducción y objetivos

1.1 Introducción

Desde hace ya unos cuantos años, la industria de los videojuegos genera una cantidad considerable de dinero, haciendo que muchas empresas dediquen una cantidad abrumadora de capital para el desarrollo de los mismos. Como consecuencia, esto genera una carencia de innovación y un "seguimiento de tendencias" que acaba provocando una sensación de haber jugado al mismo juego, sólo que con una estética diferente, ya que no consideran viable arriesgarse a hacer un juego diferente y que no genere un número de ventas elevado.

Después de realizar un análisis de mercado, hemos podido comprobar que hay una escasez de juegos basados en la exploración de islas generadas proceduralmente combinado con un toque de estrategia y aventura. Un inconveniente muy importante a tener en cuenta es el problema de la generación procedural, ya que, si no está bien desarrollado, puede hacer que el jugador se acabe cansando de explorar escenarios "demasiado similares" por la falta de identidad de cada isla.

A consecuencia de todo lo explicado anteriormente, creemos conveniente la realización del proyecto enfocado a este sector, debido a que cuentan con un escaso número de títulos que hayan tenido un éxito relevante, añadiendo a todo esto la posibilidad de aprender sobre la generación de entornos y solucionar los posibles problemas que nos encontremos.

1.2 Motivaciones

Las motivaciones que nos han llevado a desarrollar este proyecto se pueden agrupar en la siguiente clasificación:

- Estudiar la cabida de este videojuego en el mercado.
- Aprender sobre la generación de entornos, concretamente nos interesa el desarrollo de islas procedurales.
- Investigar y desarrollar una IA capaz de defenderse y atacar cuando el jugador decida saquear un territorio.
- Diseñar el sistema de mejoras y gestión de recursos.
- Balancear la economía del juego.
- Incrementar nuestro conocimiento de Unity 3D.

1.3 Propósitos y objetivos del proyecto

El propósito de este proyecto es desarrollar un videojuego al completo, de una calidad por encima de la media de sus rivales, con la intención de obtener un nivel de desarrollo y unos conocimientos relevantes.

Estos objetivos se podrían agrupar en las siguientes categorías:

1. Dominar la utilización de Unity3D y Visual Studio.
2. Concentrar los esfuerzo de diseño y desarrollo en las partes importantes del mismo, sin intentar abarcar por encima de nuestras posibilidades.
3. Desarrollar la IA de los enemigos para que generen una sensación de satisfacción.
4. Ambientar los escenarios con los sonidos correspondientes.
5. Balancear el juego a nivel de Vida/Recursos/Enemigos.
6. Ampliar el contenido de un género poco desarrollado.
7. Aprender sobre las necesidades de eficiencia en un proyecto de unas dimensiones considerables.
8. Estructurar correctamente el proyecto para aumentar la eficiencia a la hora de trabajar.

1.4 Distribución de las tareas:

La distribución de tareas la hicimos en dos partes, ya que al principio del proyecto decidimos cuáles serían los objetivos de desarrollo de cada uno, y según íbamos avanzando en el proyecto hemos ido haciendo separación de tareas en función del tiempo invertido para que resultara más equitativo.

Denís Maseda Ares

Estética	20%
Narrativa	5%
Mecánicas	30%
Tecnología	45%

Denís se ha centrado desde un principio en el desarrollo de las mecánicas principales y el aspecto estético del juego, debido a que sus objetivos profesionales se centran en aprender sobre diseño de niveles e integración de mecánicas.

El apartado de la estética hace referencia al modelado y texturizado de la mayor parte de los recursos que el jugador puede recoger y a la búsqueda de la mayoría de los assets

implementados en el juego. También se incluye en este apartado el desarrollo del HUD y del inventario.
En cuanto al apartado de la narrativa, aquí se incluye todo lo relacionado a crear una historia que dé sentido al universo desarrollado y a las misiones propuestas.
El porcentaje referente a las mecánicas es uno de los más importantes, ya que aquí se agrupa el desarrollo del personaje casi en su totalidad, el desarrollo de las IA base junto a un enemigo y los npc, implementación del sistema de recursos, inventario y loot.
El apartado de tecnología se refiere a la implementación del sistema de sonidos, utilización de máscaras para las animaciones, y utilización del sistema de shaders.

Miquel Comasòlivas Vellisco

Estética	10%
Narrativa	5%
Mecánicas	60%
Tecnología	25%

Miquel se ha centrado especialmente en las mecánicas más interactivas con el entorno como atacar, talar o picar incluyendo posibles efectos visuales y las correspondientes animaciones. Además, también ayudó en el diseño de interfaces diseñando los iconos de los recursos.

Las tareas realizadas en común consisten en el planteamiento de todo el videojuego, ya que para poder llevar a cabo este proyecto, hicimos un proceso de investigación y un desarrollo teórico muy necesario. Una vez hecho este paso, procedemos a la repartición de tareas.

2. Estudio de viabilidad

Antes de empezar el desarrollo de un proyecto de esta índole, es necesario comprobar si realmente hay un posible mercado para él y si resulta viable de desarrollar, ya sea por costes de producción como por limitaciones del personal a cargo del desarrollo.
A lo largo de este apartado hablaremos sobre las necesidades de hardware y software para desarrollar el videojuego, el estudio de mercado, el marketing que sería necesario realizar y los jugadores objetivos de este proyecto.

2.1 Recursos necesarios y viabilidad del proyecto

En este apartado se explicarán las necesidades técnicas, a nivel de hardware y software, que nos ayudarán a llevar a cabo este proyecto.

2.1.1 Recursos técnicos

Para la realización de este proyecto necesitaremos un ordenador personal por cada integrante del equipo y un conjunto de programas de uso gratuito, por lo cual no hay necesidad de realizar una inversión de dinero durante el desarrollo.
Los recursos que hemos utilizado son:

- **Sobremesa con I7 , 32 gb de ram y una RTX 3070 ti**: utilizado como herramienta de desarrollo y de dispositivo de pruebas.
- **Portátil HP OMEN 17 con intel i7, 32 gb de RAM y una RTX 3070 laptop:** utilizado como herramienta de pruebas y desarrollo para la parte de Miquel
- **Mando de Xbox one:** utilizado para testear la integración del juego con controlador de consola.
- **Unity v2021.3**: motor de videojuegos donde desarrollaremos nuestro proyecto.
- **Inkscape:** programa de diseño vectorial utilizado para la creación de algunos elementos del HUD.
- **Photopea.com:** web de edición de imágenes online similar a photoshop y completamente gratuita.
- **App.diagrams.net:** página web gratuita que nos permite realizar diagramas.
- **Visual Studio 2019:** editor de código, utilizado para programar todos los scripts implementados.
- **Discord:** programa gratuito utilizado para la realización de todas las reuniones a lo largo del desarrollo.
- **Blender:** programa utilizado para el modelado de algunos gameobjects.
- **Substance Painter:** programa utilizado para la texturización de los recursos creados en blender.
- **GameDev.tv:** página web basada en la venta de tutoriales de desarrollo de videojuegos.
- **Github Desktop:** software que nos permite tener una copia en cloud del proyecto e ir trabajando al mismo tiempo.

- **Sketchfab:** página web donde muchos creadores suben sus modelos 3D, de donde pertenecen la mayoría de nuestros assets.
- **Mixamo:** página web donde se puede subir un personaje 3D y animarlo con las distintas animaciones que dispone.
- **Freesound:** página web para descargar sonidos para la implementación del proyecto.

Todos los programas expuestos anteriormente son gratuítos por lo que no nos ha supuesto ningúna inversión. En el caso de **GameDev**, ya contábamos con esos tutoriales antes de empezar el proyecto, por lo tanto no fue necesario invertir dinero.

2.1.2 Recursos humanos:

En todo proyecto hacen falta diversos perfiles de trabajadores para poder conseguir el mejor resultado posible en el menor tiempo necesario. Debido a esto podemos concretar que harán falta los siguientes profesionales:

- **Programador:** la tarea a desarrollar consistirá en desarrollar todo el código necesario para el funcionamiento del juego.
- **Diseñador del juego:** se encargará de implementar los diferentes escenarios y de darle vida a los diferentes personajes y enemigos, consiguiendo un apartado estético, visual y narrativo capaz de involucrar al jugador.
- **Diseñador 2D/3D:** su principal función se basará en desarrollar todos los assets necesarios para el juego, siempre y cuando no se puedan encontrar en internet de forma gratuita.
- **Artista de sonidos:** Su principal función consiste en implementar los sonidos necesarios en el videojuego.

Hay muchos otros perfiles que hacen falta para implementar este proyecto, pero que no tienen un peso tan relevante. Cabe destacar que, como desarrolladores, no nos hemos enfocado a hacer assets propios, si no que la mayoría son de páginas como Sketchfab, donde otros creadores los publican de forma gratuíta, aunque sí que hay objetos como los recursos que sí hemos diseñado nosotros.
A nivel de sonidos hemos tenido tres fuentes importantes para obtenerlos, ya que por un lado nos ha cedido un paquete de sonidos el productor musical “GALEA”, el cual disponía de otro proyecto. También los hemos obtenido de GameDev y por último de variadas páginas web.

2.1.3 Viabilidad económica:

Hablando del apartado económico, hemos visto que no necesitamos hacer ninguna inversión económica para poder llevar a cabo este proyecto, pero si hiciéramos un plan de los costes, saldría lo siguiente:

Recursos	Costes
Equipos de trabajo	4.000€
Controlador de juego	50€
TOTAL	4.050€

En cuanto a lo que se refiere al suelo de los profesionales, podemos estimar que serían profesionales junior, sin experiencia o casi nula, por lo que el esquema de los sueldos sería el siguiente:

- Desarrollador de videojuegos: 12€/hora
- Programador C++: 11€/h
- Diseñador 2D/3D: 8€/hora
- Artista de sonidos: 11€/h

Para conseguir la información de los sueldos hemos optado por la utilización de la página web https://www.jobted.es.

Se podría obtener el precio de desarrollar este proyecto multiplicando las horas invertidas por el coste de trabajo de cada profesional, pero como hemos sido nosotros los únicos participantes del mismo, el coste final es 0.

2.2 Estudio de mercado

Una vez que hemos llegado a este punto del desarrollo, contando ya con una idea de qué proyecto queremos desarrollar, tenemos que estudiar cómo se encuentra hoy en día el sector de los videojuegos, y más concretamente el sector donde nos queremos centrar con este proyecto, ya que hay que detectar los puntos fuertes de los rivales y hacerse un hueco aportando ideas innovadoras y refrescantes.

2.2.1 Estado del arte

Como hemos dicho anteriormente, para poder tener éxito con este proyecto, necesitamos hacer un análisis exhaustivo de los juegos más relevantes en el sector que nos queremos introducir.
El objetivo de este análisis es detectar cuales son los puntos claves que han llevado al éxito a todos estos juegos, detectar que comparten en común entre ellos y ya como colofón final, detectar qué aspectos son los más irrelevantes para no tenerlos en cuenta, e intentar cambiarlos por aspectos novedosos que puedan actualizar este sector.

Ahora hablaremos de los juegos que creemos que son referentes en los géneros que hemos elegido para desarrollar nuestro proyecto, explicando un poco sus características y destacando cuáles son los puntos fuertes que les han hecho destacar.

The Last Of Us (Aventura)

The Last Of Us es un juego de aventura 3D con una estética realista, donde el jugador tendrá que ir avanzando en la historia para descubrir las vivencias por las que han pasado los protagonistas en la pandemia de EE UU.
Lo que nos ha llamado la atención de este juego es la increíble integración de la historia con el escenario, haciendo que ambos sean uno.

Figura 2.2.1: Captura de la serie The Last Of Us

Patrician II (estrategia)

Patrician II es un videojuego de simulación comercial, donde el jugador tiene que aprender a gestionar sus recursos, ya que todos los aspectos del juego ocurren en tiempo real. De este juego hemos tomado como referencia el balance de los recursos y el sistema de comercio.

Figura 2.2.2: Captura del juego Patrician 2

No Man's Sky (generación procedural)

No Man's Sky es un juego de exploración y aventura donde el jugador tendrá que explorar todo el universo intentando categorizar el mayor número de especies. Todo el escenario es generado proceduralmente, de forma que es bastante complicado encontrarse al mismo tipo de animales.

De este juego nos quedamos con dos apartados importantes, ya que por un lado es impresionante la capacidad de generación del universo, pero por otro lado, en algunas ocasiones resulta demasiado similar un planeta al otro, ya que comparten muchas características.

Figura 2.2.3: Captura del juego No Man's Sky

The Legend of Zelda: Link's Awakening (estética y jugabilidad)

The Legend of Zelda: Link's Awakening es un videojuego de acción y aventura, donde el jugador tiene que ir combatiendo contra los distintos rivales, desbloqueando las distintas armas y objetos. Gracias a esta variedad, el jugador puede ir alternando la forma de combatir, ya que por lo general no hay una única forma de derrotar a los enemigos.

De este juego tomamos como referencias el aspecto estético, las mecánicas y el funcionamiento base de los enemigos.

Figura 2.2.3: Captura del juego The Legend of Zelda: Link's Awakening

Como conclusión del análisis de estos juegos, podemos comprobar que hay muchos elementos diferentes nunca antes utilizados en el género que queremos realizar. Además, hay determinados aspectos que necesitan una mejora, ya que nos parece que un poco de esfuerzo pueden obtener mejores sensaciones para el jugador.
Como aspectos innovadores, implementamos un sistema de mejoras de las estructuras y personaje para que el jugador pueda descubrir nuevas islas y tenga más facilidades a la hora de derrotar a los enemigos. Además, el escenario en sí es una innovación, ya que ninguno de los juegos anteriores se basa en islas, lo cual afecta directamente el sistema de recursos para aumentar la dificultad del juego.

2.3 Público objetivo

Para poder hacer un juego y que este tenga éxito comercialmente, es necesario tener identificado claramente cuál es el público objetivo del mismo. Dada la índole de nuestro proyecto, se podría decir que va enfocado tanto a jugadores casuales como a los más fanáticos del género de la exploración y aventuras.

Para definir con más precisión el público objetivo, podemos decir que el rango de edad se encuentra entre los 7 y los 30 años, que disfrutan de intentar descubrir todas las posibilidades que te brinda el juego, ya sea en forma de generación del entorno, como todas las combinaciones de mecánicas posibles para combatir contra los enemigos de todas las formas posibles que admite el juego.

3. Planificación

3.1 Diagrama de Gantt

Fue a principios de junio del 2022 cuando empezamos a hablar con nuestro tutor del tfg, el cual nos dió la libertad de empezar a plantear el proyecto en cuanto quisiéramos, por lo que no tardamos mucho en empezar el desarrollo.

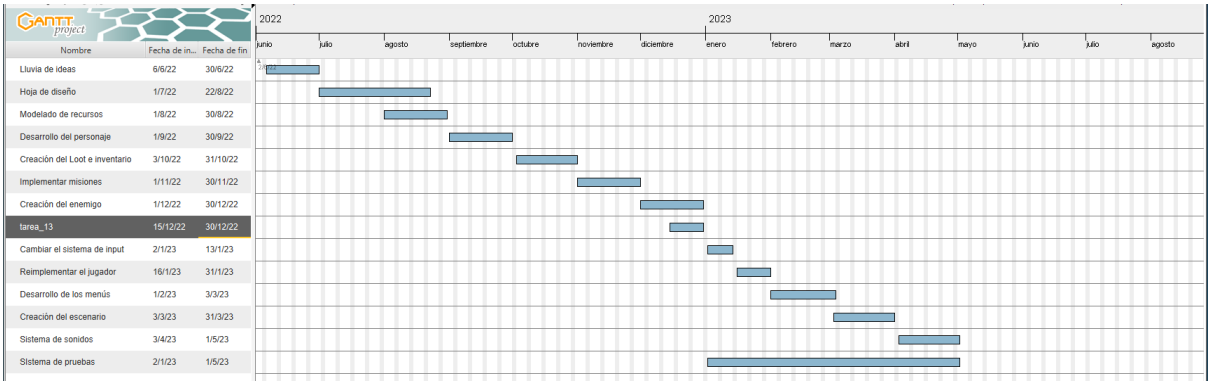


Figura 3.1: Diagrama de Gantt del proyecto.

3.2 Herramientas de trabajo y programario

A lo largo de este apartado hablaremos de todos los programas que hemos utilizado para la realización de este proyecto, explicando al mismo tiempo su utilidad.

Para poder desarrollar un videojuego hace falta un motor de videojuegos, el cual implementa todas las herramientas necesarias para el funcionamiento del mismo. Partiendo de esta base, nos encontramos con la posibilidad de crear uno por nuestra cuenta o utilizar los que están disponibles en el mercado.

Hoy en día nos podemos encontrar con dos motores principales, los cuales son Unity3D y Unreal Engine 5.

En cuanto a lo que Unreal se refiere, es un motor enfocado al desarrollo de grandes producciones, donde todas las partes de desarrollo están bastante diferenciadas para ayudar a que todos los trabajadores puedan trabajar de forma simultánea sin tener problemas de conflictos a la hora de subir los cambios. El lenguaje de desarrollo es C++ pero camuflado en bloques de código ya que utiliza un lenguaje de scripts visual.

La otra opción válida sería Unity3D, enfocado principalmente a estudios más pequeños, donde no hay tanta separación en las diferentes partes del desarrollo y que permite una grán versatilidad a la hora de realizar la implementación de los scripts. El lenguaje de desarrollo es C#.

En nuestro caso nos hemos decantado por Unity, ya que está muy orientado al desarrollo de videojuegos indie, facilitando mucho al usuario el desarrollo de todos los apartados. Esto es así porque todas las características están muy relacionadas. Un ejemplo de esto es la facilidad de utilizar un animator y controlar las animaciones por eventos.

Para editar el código hemos utilizado única y exclusivamente Visual Studio, que tiene integración completa con Unity3D y facilita mucho el trabajo a realizar gracias a todas las características de las que dispone.

4. Marco de trabajo y conceptos previos

Es necesario contextualizar que excepto el juego de Zelda, el resto no pertenecen al género en el que nos queremos centrar, por lo que a la hora de realizar la implementación nos encontraremos que obtendremos una gran variedad de conceptos novedosos para un género con unos estándares ya muy marcados.

En cuanto al sistema de mejoras de edificios y personaje, nos encontramos con que estará planteado de tal forma que obligue al jugador a realizar una progresión lineal sin que este se dé cuenta, con la intención de que sea capaz de disfrutar de ello sin la necesidad de saberlo. Según avance en la historia, se le irá pidiendo recursos más variados, que en un futuro le servirán para mejorar diversas estructuras o incluso a él mismo, lo cual no suele ser muy común en el género.

5. Diseño del videojuego

5.1 Mecánicas

5.1.1 El espacio del juego

Durante el proceso de planteamiento del juego nos dimos cuenta de que queríamos un juego de exploración, donde el jugador tuviera que ir visitando las distintas islas que estuvieran disponibles para ir consiguiendo los recursos necesarios para ir evolucionando el poblado donde se encuentra.

Espacio a nivel de isla:
Este será un espacio dividido, una subcategoría de open world, ya que el jugador puede desplazarse libremente por toda la isla para ir completando las misiones mediante la recolección de recursos/completando las mazmorras.

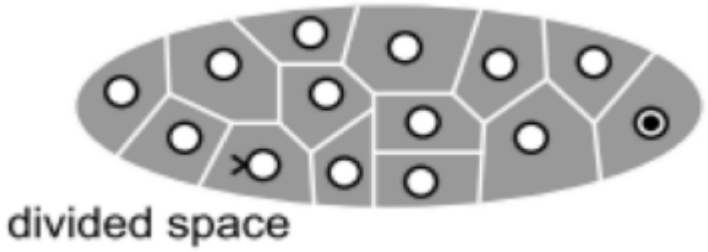


Figura 5.1.1: Captura del ejemplo “divided space”

Espacio entre islas:
La distribución de las islas consta de únicamente dos nodos conectados entre ellos bidireccionalmente, es decir, si se supone que el nodo izquierdo es la isla principal del juego y la de la derecha la isla a conquistar o saquear, solo se puede ir de una a otra y viceversa, con lo cual, obtenemos una distribución como la que se ve en la figura 5.1.2.

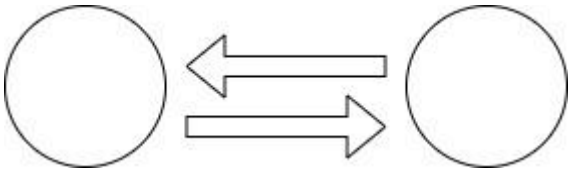


Figura 5.1.2: Imagen de la estructuras de las islas

El espacio dentro de la isla será como el de la isla principal, que hemos explicado anteriormente.

El sistema de exploración entre islas se implementará en la versión futura de este proyecto.

5.1.2 Recursos, atributos y objetos

Hay distintos tipos de recursos con los que cuenta este proyecto, con los cuales el jugador puede ir interactuando, ya sea para comercio, para consumo propio o incluso para mejorar las propias estructuras del pueblo.

Recursos	**Atributos**
Moneda	- Cantidad (nº unidades) - Valor
Minerales (Oro/Plata/Diamante)	- Cantidad (nº unidades) - Valor (precio) - Rareza (prob. aparición)
Comida y cultivos (Cacao / Patata / Tomate / Pimiento / Vainilla / Maíz / Trigo)	- Cantidad (nº unidades) - Valor (precio) - Valor energético - Prob. aparición
Recursos (Tela / Madera / Piedra)	- Cantidad (nº unidades) - Valor (precio) - Prob. aparición
Personaje	- Vida - Estamina - Capacidad de almac. - Ataque - Defensa
Enemigos	- Vida - Ataque - Defensa - Ratio de visión

La probabilidad de aparición de cada uno de estos elementos depende en función de cada recurso, ya que los minerales tienen una probabilidad menor frente a otros recursos como por ejemplo la madera. Estas probabilidades también dependen de la entidad que lo genere como recompensa; para poner un ejemplo podríamos decir que la probabilidad de que un enemigo suelte una unidad de diamante es nula.

Ahora vamos a hablar de la utilidad de los recursos descritos anteriormente y la forma de conseguirlos, ya que no todos se comportan de la misma manera.

Las monedas se pueden conseguir de dos formas distintas, ya que por un lado las podemos obtener de las mazmorras (tanto de cofres como de los enemigos) como de completar las misiones que nos proponen los habitantes del pueblo.

Los recursos de la categoría de minerales se pueden encontrar en los cofres que se encuentran en las mazmorras y en menor medida, a la hora de picar las piedras. Su utilidad se centra en la mejora de las diferentes construcciones.

En la categoría de Comida y cultivos, nos encontramos con recursos que se pueden conseguir mediante los cofres de las mazmorras. También hay la posibilidad de encontrarse con semillas, las que se pueden cultivar en los campos de cultivo del pueblo. Estos recursos se pueden consumir para recuperar vida.

Los recursos base, como son tela, madera y piedra; se pueden conseguir directamente en el pueblo inicial mediante la realización de mecánicas base como pueden ser interactuar, talar y picar.

Otros objetos como pueden ser el personaje y el enemigo tienen atributos propios para la realización de las mecánicas de cada uno.

Es importante remarcar un detalle de este apartado. Estos son los elementos que teníamos pensado utilizar durante el desarrollo del juego pero se verán implementados en una versión futura. Los elementos restantes son:

- Monedas
- Cacao
- Maíz
- Trigo
- Tela

5.1.3 Acciones del jugador

Al ser un juego de acción y aventura, podemos encontrarnos unas mecánicas similares a los juegos que hemos descrito anteriormente y las podemos ver enumeradas en la siguiente lista:

- **Caminar**: movimiento básico del jugador, que le permite desplazarse por todo el escenario e ir investigando / completando todas las misiones.
- **Correr:** mecánica igual a la de andar pero con una velocidad de desplazamiento mayor.
- **Atacar**: permite infringir daño a los enemigos; se puede hacer mediante una espada, que permite utilizar distintos combos, o con un ataque especial, que es un orbe mágico.

- **Talar**: mediante la utilización de un hacha, el jugador puede golpear árboles para cortarlos, los cuales al colisionar contra el suelo se destruyen y generan el recurso de la madera.
- **Picar**: el jugador puede golpear las rocas con el pico, con lo cual las destruirá y estas tendrán un recurso, ya sea piedra o un mineral.
- **Defenderse**: el jugador puede utilizar el escudo del que dispone para bloquear los ataques que realizan los enemigos, pero se consumirá estamina mientras que el escudo esté activo.
- **Interaccionar**: permite realizar acciones diferentes como coger objetos del suelo, abrir cofres, acceder al menú de las estructuras importantes, etc.
- **Intercambiar**: gracias a la interacción con determinados aldeanos de la isla, el jugador puede intercambiar los recursos que posee por otros que le hagan falta. Las ofertas de intercambio no tienen porqué ser las mismas todos los días.
- **Comer**: mediante la interacción con la comida, el jugador puede incrementar su salud de forma instantánea.
- **Guardar los recursos**: para poder guardar los recursos, el jugador tendrá que ir personalmente al ayuntamiento, siempre y cuando venga de otra isla. Si el jugador se encuentra en su isla, estos recursos se guardan directamente.

De las mecánicas iniciales que teníamos planteado implementar, no hemos implementado para esta versión el sistema de intercambio y la mecánica de comer, pero que se verán en futuras versiones del proyecto.

5.1.4 Objetos con mecánicas propias

A lo largo del planteamiento de este proyecto, nos hemos dado cuenta de que hay una serie de objetos que cuentan con una mecánica propia que no es influenciada por nada más. Uno de estos objetos son los recursos, ya que habrá una mecánica propia que se encarga de gestionar la probabilidad de que al jugador le toquen unos recursos u otros en función de determinados parámetros, como pueden ser la dificultad de una zona, la rareza de los recursos, la dificultad del enemigo, etc.

Otro objeto que tiene mecánica propia es el sistema de cultivos, ya que el sistema de crecimiento va ligado al tiempo físico que necesitan para crecer, el cual se implementará en el futuro.

5.1.5 Economía del juego

La economía del juego se puede dividir en 4 categorías distintas que son “Sources”, “Drains”, “Converters” y “Traders”. Estos ayudan a que el juego esté bien balanceado y ofrezca una experiencia lúdica redonda al jugador, donde se consideren justas todas las acciones económicas.

Sources

Las sources con las que se encontrará el jugador a lo largo del juego son las siguientes:

- **Cofres**: cuando el jugador interactúa con un cofre, este realiza la animación de abrirse y procede a proporcionar un recurso. El contenido de cada cofre puede variar en función de su ubicación y la dificultad de la zona. También se puede dar el caso de que el cofre no de ninguna recompensa.
- **Enemigos**: a la hora de que el jugador derrote a un enemigo, éste le proporcionará un recurso, determinado en función de la dificultad del mismo.
- **Cultivos**: una vez que el cultivo está maduro y listo para su recolección, el jugador puede proceder a su recolección simplemente acercándose a ellos.

Drains

- **Talar/Picar/Atacar:** a la hora de realizar estas acciones, se consumirá estamina. Cada acción tiene un coste distinto.
- **Enemigo:** cuando los enemigos ataquen al jugador y el ataque colisione contra este, la vida del jugador se reducirá.
- **Escudo**: cuando el jugador se esté protegiendo con el escudo, la estamina se irá decrementando en función del tiempo que esté bloqueando.

Converters

- **Talar/Picar:** al realizar esta acción, el jugador convierte los árboles y las rocas en madera y piedra, que son recursos utilizables.
- **Comer:** convierte los recursos categorizados como comida en salud para el jugador.

Traders

- **Aldeanos:** el jugador podrá realizar intercambios en los puestos de intercambio fijados en el pueblo. Los intercambios disponibles por día cambian en función del día.

5.1.6 Feedback al jugador:

A lo largo del juego, se presentarán distintos tipos de feedback en forma de sonidos para que entienda que la acción se ha completado correctamente y que sólo tiene que esperar a que se finalice la misma.

El feedback auditivo que escuchará será el siguiente:

- El jugador realiza una interacción.
- El jugador recoge un objeto.
- El jugador no puede recoger un objeto.
- El jugador realiza la interacción de picar con una piedra.

- El jugador realiza la interacción de talar con un árbol.
- El cofre finaliza la animación de abrirse.
- El jugador cambia de botón.
- El jugador selecciona un botón.

Todos estos sonidos serán únicos para cada tipo de feedback, por lo que cuando el jugador los escuche sabrá perfectamente en qué situación se encuentra.

5.1.7 Esquema de interfaces

A lo largo del juego nos podemos encontrar las siguientes interfaces, con las que el jugador podrá interactuar para entender lo que está pasando en el juego, además de saber cómo proseguir en el mismo.

La primera interfaz con la que nos encontramos es la interfaz de inicio del juego, donde el jugador puede elegir entre crear una nueva partida, cargar la partida anterior o salir del juego. Ver Figura 5.1.7.1.

Figura 5.1.7.1: Captura de pantalla del menú de inicio de nuestro juego

La siguiente interfaz con la que nos encontraremos es la pantalla de pausa, donde el jugador puede cargar/guardar la partida, volver al menú de pausa y salir del juego. Ver Figura 5.1.7.2.

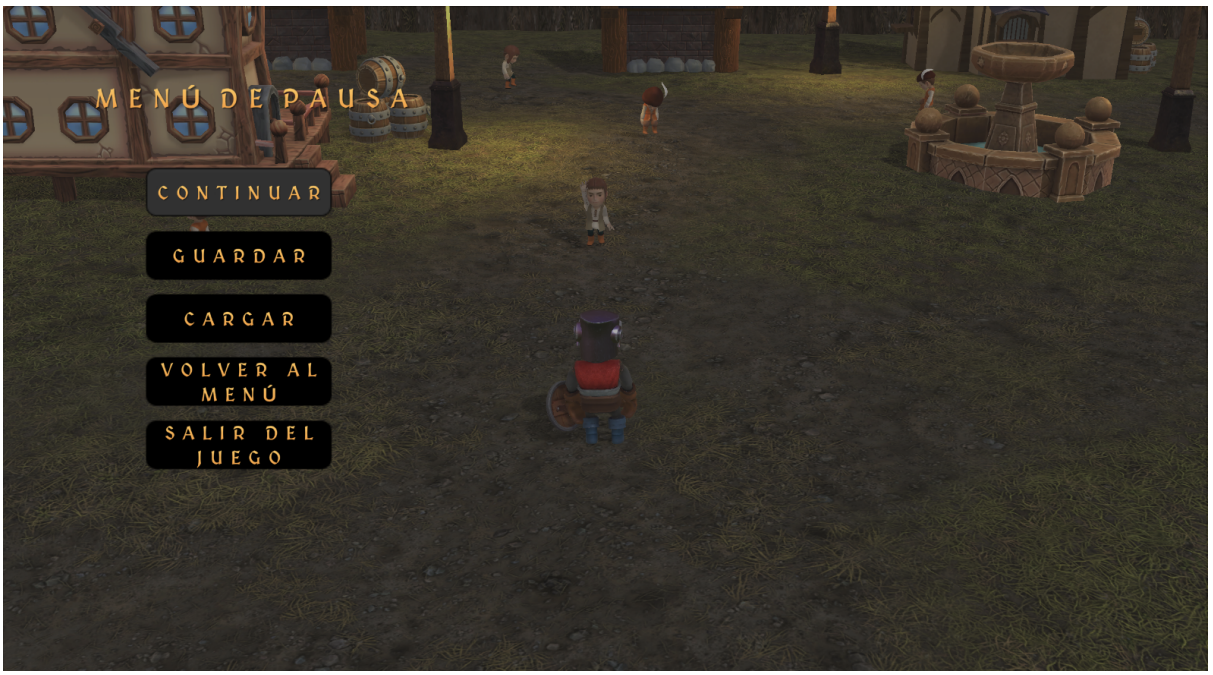


Figura 5.1.7.2: Captura de pantalla del menú de pausa de nuestro juego

Otra pantalla de información con la que nos encontraremos es al de nivel de las estructuras/jugador, donde se puede ver el nivel actual de la estructura según las estrellas que estén iluminadas, los niveles que se puede mejor la estructura junto a la recompensa que ofrece y los recursos que hacen falta para subir al siguiente nivel, aunque en la versión actual del juego no contamos con esta funcionalidad, se verá implementada en la siguiente versión. Ver Figura 5.1.7.3

Figura 5.1.7.3: Captura de pantalla del menú de mejora del juego"Littlewood"

Otra interfaz que resulta muy útil también, es la interfaz que muestra los niveles actuales de vida y estamina, los cuales se muestran siempre que los valores no estén al máximo.
Con la intención de hacerlos diferentes a todos los juegos con los que nos hemos comparado, optamos por realizarlos circulares. Ver Figura 5.1.7.4.

Figura 5.1.7.4: Captura de pantalla del HUD de salud/stamina

Para guiar al jugador hacia las misiones de una manera adecuada, hemos establecido un npc que cuenta con un cartel encima de la cabeza, donde se muestra un texto indicativo. Al principio se le hará una introducción para presentarle el contenido del pueblo, con el cual puede interactuar. Una vez hecho esto se le indicarán de forma ordenada los objetivos establecidos para superar las misiones del juego; además de guiarlo hacia el objetivo Ver Figura 5.1.7.5.

Figura 5.1.7.5: Captura de pantalla del HUD de salud/stamina

Para recordarle al jugador el objetivo a seguir hay un panel en la esquina izquierda superior de la pantalla, donde se pondrá la información de la misión actual. Estará siempre activa a excepción de cuando el jugador entre en un menú o cuando se acerque al NPC que le indica los objetivos. Ver Figura 5.7.1.6.

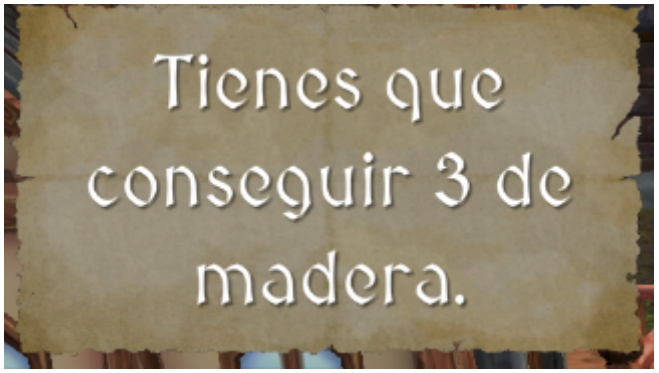


Figura 5.1.7.6: Captura de pantalla del recordatorio de misión

Dentro del poblado, el jugador puede acercarse al ayuntamiento e interactuar con él para comprobar los recursos con los que cuenta. Cuando el jugador está en el poblado, este será el único inverario con el que contará. Ver Figura 5.1.7.7.

Figura 5.1.7.7: Captura de pantalla del invetario del ayuntamiento

Cuando el jugador salga de esta isla para investigar otras, contará con un inventario de un tamaño limitado, donde se irá almacenando los objetos que encuentre. Esta característica estará disponible en una versión futura del proyecto. Ver Figura 5.1.7.8.

Figura 5.1.7.8: Captura de pantalla del inventario de “Stardew Valley”

Para facilitar la interacción del personaje con algunos objetos, se ha puesto un mensaje al lado de ellos indicando la tecla/botón a pulsar para poder realizarla. Ver Figura 5.1.7.9.

Figura 5.1.7.9: Captura de las guías de interacción

5.1.8 Visión global del juego

Para poder tener una mejor imagen de cómo funcionará este juego, podemos observar el diagrama de la Figura 5.1.8.1, donde se explican todas las partes de la isla principal y las acciones que se pueden realizar en la misma. Así mismo también se habla del sistema para visitar nuevas islas

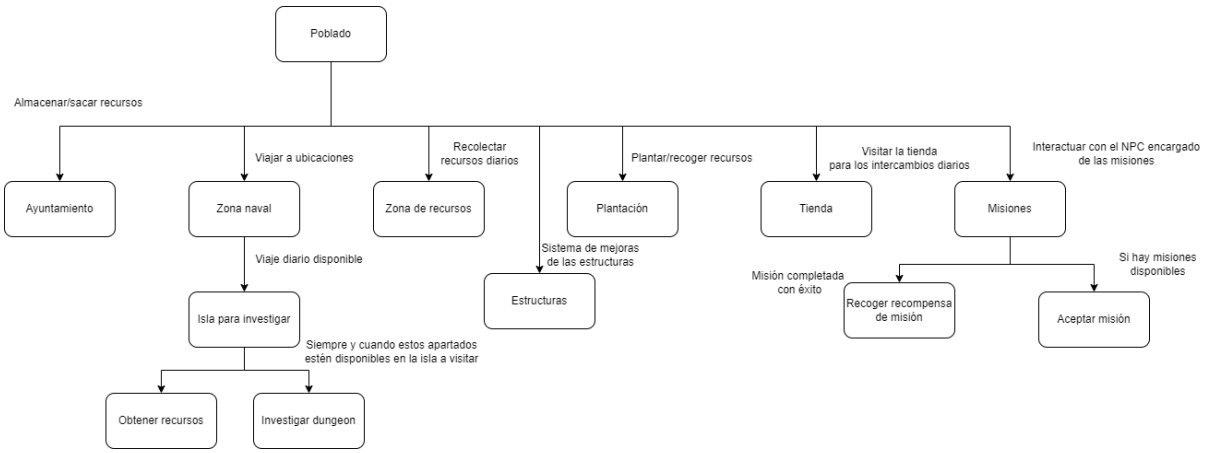


Figura 5.1.8.1: Gráfico explicarivo de la Visión Global del Juego

Es necesario remarcar que, para la versión actual del proyecto, sólo contamos con algunas estructuras, donde solo es funcional el ayuntamiento, un sistema de recursos en la propia isla y el sistema de misiones. Las restantes características serán implementadas en versiones futuras.

5.1.9 Niveles de dificultad

En cuanto a lo que niveles de dificultad se refiere, hemos establecido un sistema de niveles, el cual permite al jugador ir accediendo a zonas más difíciles donde los enemigos presentan un reto mayor, y será más sencillo encontrar objetos de mayor rareza.

El sistema de niveles está planteado para que el jugador pueda conseguir la suficiente habilidad en la zona anterior antes de pasar a la siguiente fase, ya que si pasara antes de lo recomendado, se puede dar la situación de que se aburra de jugar por el cansancio de repetir la misma parte de la misión hasta superarla o dejarlo. Ver Figura 5.1.9.1.

Este sistema de dificultad se verá implementado en la siguiente versión del proyecto.

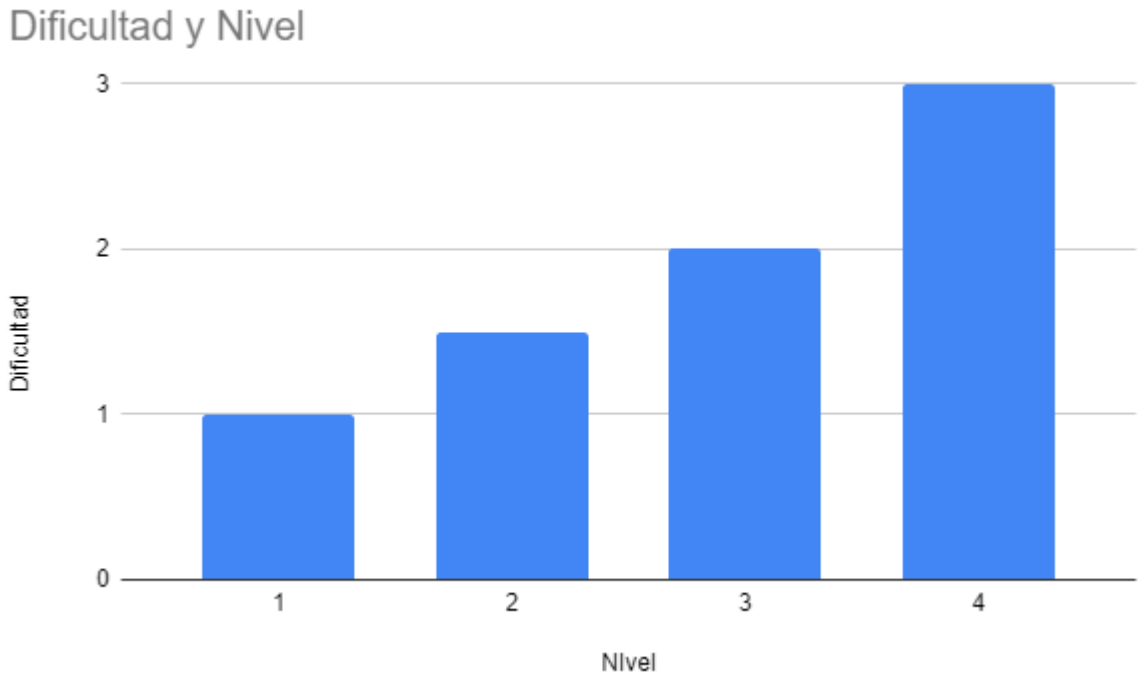


Figura 5.1.9.1: Gráfico de dificultad

5.1.10 Game balancing

El nivel de dificultad en este juego se basa en el nivel que dispongan las estructuras que hay en el poblado, ya que mejorar la estructura de navegación permite visitar zonas más complicadas con mejores recursos, pero con enemigos más complicados. Para hacer más sencillos los combates, el jugador tendrá que mejorar las estructuras del pueblo y sus estadísticas. Esto se puede ver en la Figura 5.1.10.1.

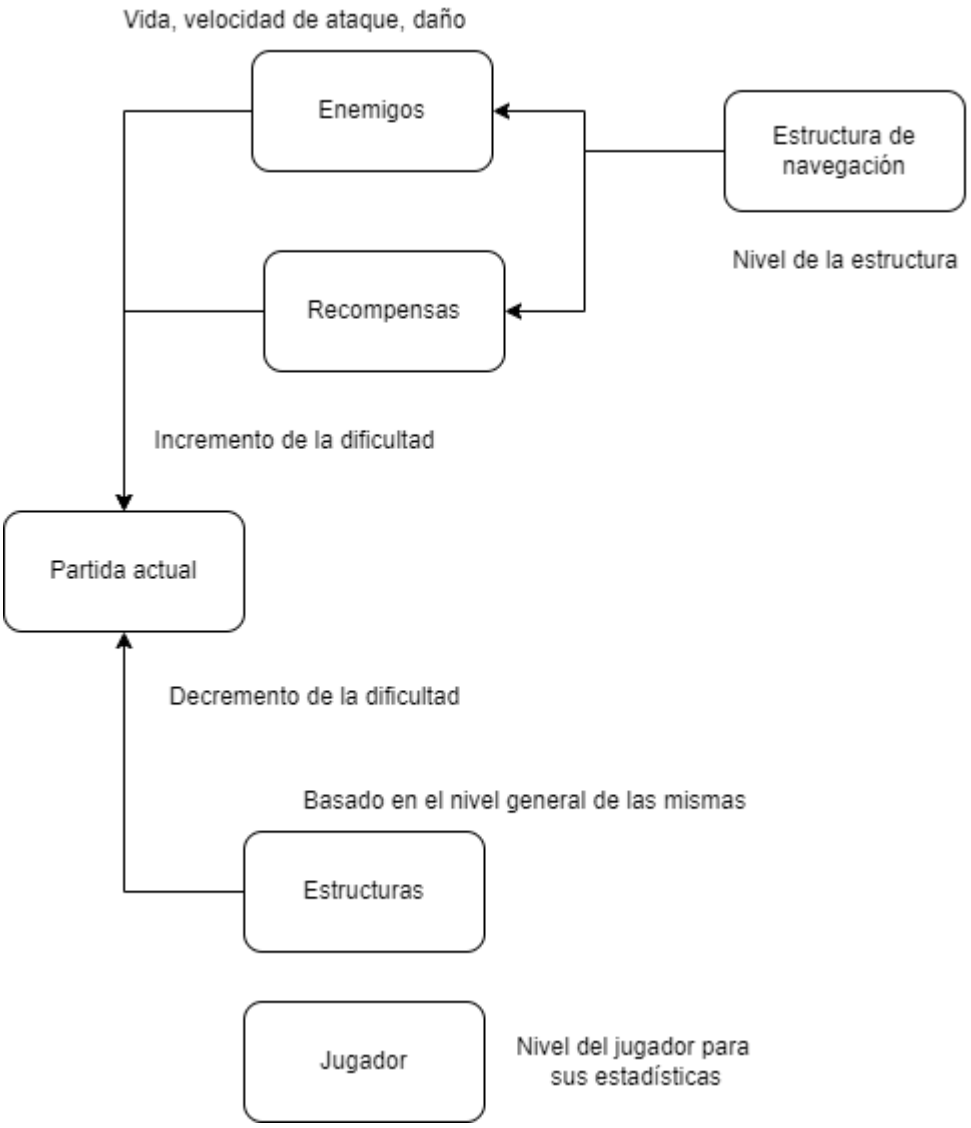


Figura 5.1.10.1: Gráfico explicarivo del Game Balancing

Es importante destacar que esta parte no ha sido implementada para la versión actual del juego.

5.1.11 Shadow Cost

Una forma indispensable de balancear el juego es con un Shadow Cost. En este caso hemos optado por utilizar la estamina, ya que resulta fundamental tener estamina para realizar cualquier acción que requiera un arma. Es decir, cuando el jugador quiera realizar la acción de atacar, talar, picar, correr o bloquear; se consumirá una cantidad determinada de estamina y esta no se va a regenerar hasta que se finalice la acción. De este modo, evitamos un abuso de un conjunto de ataques excesivos y balanceamos las interacciones entre atacar, esquivar y bloquear. En caso de que el jugador acabe con toda la estamina, esta se regenerará con mayor lentitud.

5.1.12 Estrategia piedra-papel-tijera

Dependiendo de la situación que el jugador deba enfrentar, se presentan distintas estrategias que pueden resultar o no exitosas. Un ejemplo podría ser el caso que deba enfrentarse a un enemigo poco veloz con el que pueda resultar más efectivo un ataque básico sin defensa, o incluso un ataque especial si se dispone de suficiente estamina, o si se presenta un enemigo al cual se le deba "romper" la defensa mediante el escudo para luego atacar. En esas situaciones, primero una defensa y luego un ataque, daría más resultado. Ver Figura 5.1.12.1.

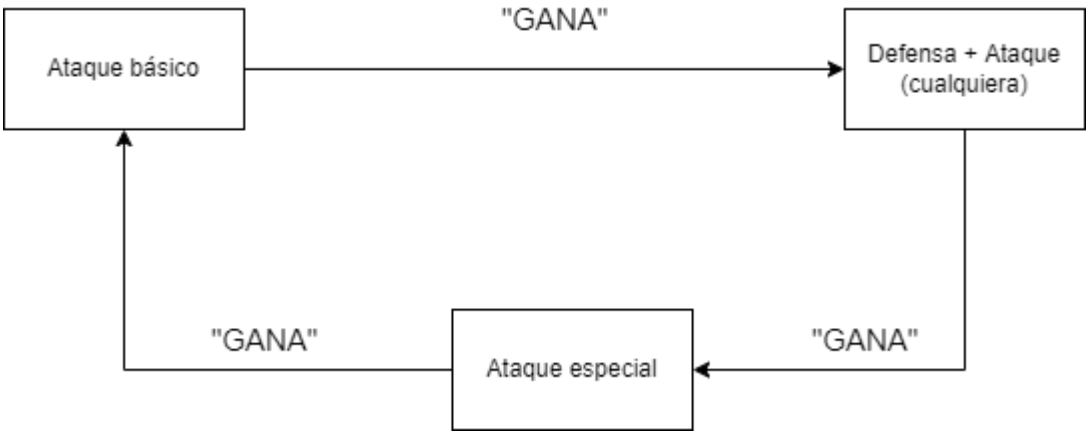


Figura 5.1.12.1: Gráfico explicarivo de la estrategia piedra - papel - tijera

Cabe destacar que esta gráfica es en función de la estamina, ya que en caso que el jugador se encuentre con baja estamina, un ataque básico es más eficaz que un ataque con defensa, ya que se quedaría sin estamina para poder escapar. Pasa lo mismo con el ataque + defensa y ataque especial, porque un ataque especial consume más que el otro. En el caso que el enemigo se encuentre con poca vida, un ataque especial le gana a ataque básico aunque se quede sin estamina si es capaz de matarlo de un golpe.

5.2 Narrativa del juego

La narrativa es un apartado importante en el desarrollo de cualquier videojuego, ya que ayuda al jugador a comprender la situación y a guiarlo hacia las misiones.

5.2.1 Sinopsis

El personaje principal despierta en la playa, sin saber quien es ni de dónde viene. Al rato de despertar lo encuentra el alcalde del pueblo. Al ver que no recuerda nada de su vida pasada, el alcalde le pide que lo siga al pueblo, donde podrán hablar tranquilamente.

Una vez que ambos están en el pueblo, el alcalde le presenta al jugador la situación del pueblo, ya que se encuentran asolados por las hordas de piratas; los cuales se dedican en cuerpo y alma a saquearlos cada vez que se consiguen recuperar. El alcalde le

comenta al jugador que ellos estarán encantados de acogerlo, pero que necesitarán un poco de ayuda para hacer prosperar el poblado. Ahora que el jugador ya sabe la situación del poblado, le enseña las ubicaciones más importantes del mismo, ya que estas serán imprescindibles para el desarrollo de la aventura.

Así mismo, le comenta que necesitan diversos materiales para ir mejorando las construcciones del pueblo, por lo que le pide al jugador que las vaya mejorando poco a poco, indicando en cada momento cual es la estructura más recomendable a mejorar, aunque se puede mejorar la que más necesaria considere el jugador, siempre y cuando tenga los recursos necesarios para ello.

Ahora que el jugador ya está en contexto, se le pide que se dirija a la herrería para conseguir unas herramientas en condiciones óptimas, con las que pueda recolectar recursos y defenderse de los enemigos en caso de que esto sea necesario.

Ahora que está armado y con los objetivos claros, el jugador se dirige a su primera expedición, donde conseguirá los recursos necesarios para hacer algunos avances en el desarrollo del poblado, pero lo que no sabe es que le quedan muchas expediciones por delante, cada una más complicada que la anterior, para poder mejorar el poblado al completo.

Para esta versión del proyecto contamos con una variación de la narrativa, en la que el alcalde se da cuenta que es la primera vez que el jugador visita el poblado y le hace de guía para que este pueda conocer todas las ubicaciones de importancia. Así mismo, le explica las ubicaciones de importancia que existen fuera de éste. Una vez que el jugador ya está puesto en contexto, le explica que necesita su ayuda para poder conseguir unos materiales, los cuales se encuentran en las ubicaciones explicadas anteriormente. Cuando el jugador consiga todos los recursos que necesita el alcalde, éste lo invita a explorar la mazmorra que se encuentra cerca del poblado, donde puede probar a encontrar los cofres para conseguir suculentas recompensas.

5.2.2 Dispositivos de narrativa

A la hora de plantear el desarrollo de este proyecto, teníamos en mente que necesitábamos integrar dentro del juego todos los elementos que fueran posibles.

Teniendo esto como premisa, consideramos oportuno que el alcalde del pueblo cuente toda la información que sea necesaria para entender el mundo que rodea al jugador. Además de esto, la información se muestra a través de un panel camuflado de papiro que está integrado directamente en el escenario. La información que se le facilitará al jugador es el contenido del pueblo (las ubicaciones de importancia) y los objetivos de cada misión.

De todas formas, el objetivo es intentar poner la menor cantidad de información en diálogos e incitar al jugador a investigar por su cuenta el pueblo, ya que es una forma más divertida de aprender donde están todas las ubicaciones de importancia.

5.2.3 Mecanismos para avanzar en la historia

La forma de proseguir en la historia es completando las misiones que va proporcionando el alcalde, ya que después de cada una explicará acontecimientos que sucedieron en el pueblo y que los han llevado a la situación actual. Cabe destacar que esta características estará presente en futuras versiones.

5.4 Mundos del juego

5.4.1 Dimensión física

El espacio de desarrollo del videojuego se basa en islas, las cuales pueden estar hechas de distintos materiales. La isla principal sirve a modo de tutorial, donde se enseña al jugador las funcionalidades básicas del juego, y se le presenta una pequeña introducción a las mecánicas con las que estará interactuando con el entorno constantemente.

A la hora de pensar cómo implementar la investigación de diferentes islas, se nos ocurrió la idea de establecer un puerto, donde el jugador pueda acercarse y solicitar un viaje para explorar más fronteras. Además, estas islas serán procedurales, por lo que no podrá visitar dos veces la misma isla, así que si se encuentra con cualquier recurso que pueda considerar importante, es mejor que lo recoja sin pensárselo dos veces.

5.4.2 Dimensión temporal

El juego depende de un ciclo de día y noche, ya que hay algunos eventos que dependen del paso del tiempo para que puedan ocurrir.

- Los viajes en barco solo se pueden realizar una vez al día; por lo que necesitamos que se acabe el día para poder realizar otro viaje.
- Al final de cada día, existe una probabilidad de que al siguiente día ocurra una invasión del poblado. Este porcentaje va en función de las acciones que realice el jugador.
- Al final de cada día, se inicia el proceso de generación de recursos de la isla principal, se actualiza el proceso de crecimiento de los recursos y se cambia los intercambios que ofrecen en la tienda.

5.4.3 Dimensión ambiental

Este aspecto tiene como objetivo servir de base para establecer la historia, ya que todas las acciones giran en torno a las consecuencias de la vida pirata.

Esto viene reflejado en las construcciones, ya que son construcciones de ambientación medieval, al igual que la estética de los personajes. Además, todo esto da paso a la introducción del juego y el desenlace de todas las aventuras que vivirá el jugador a lo largo de su travesía por estas tierras tan azotadas por esos piratas.

5.5 Game layout charts

El diagrama que rige toda la estructura del juego se puede ver en la figura 5.5.1:

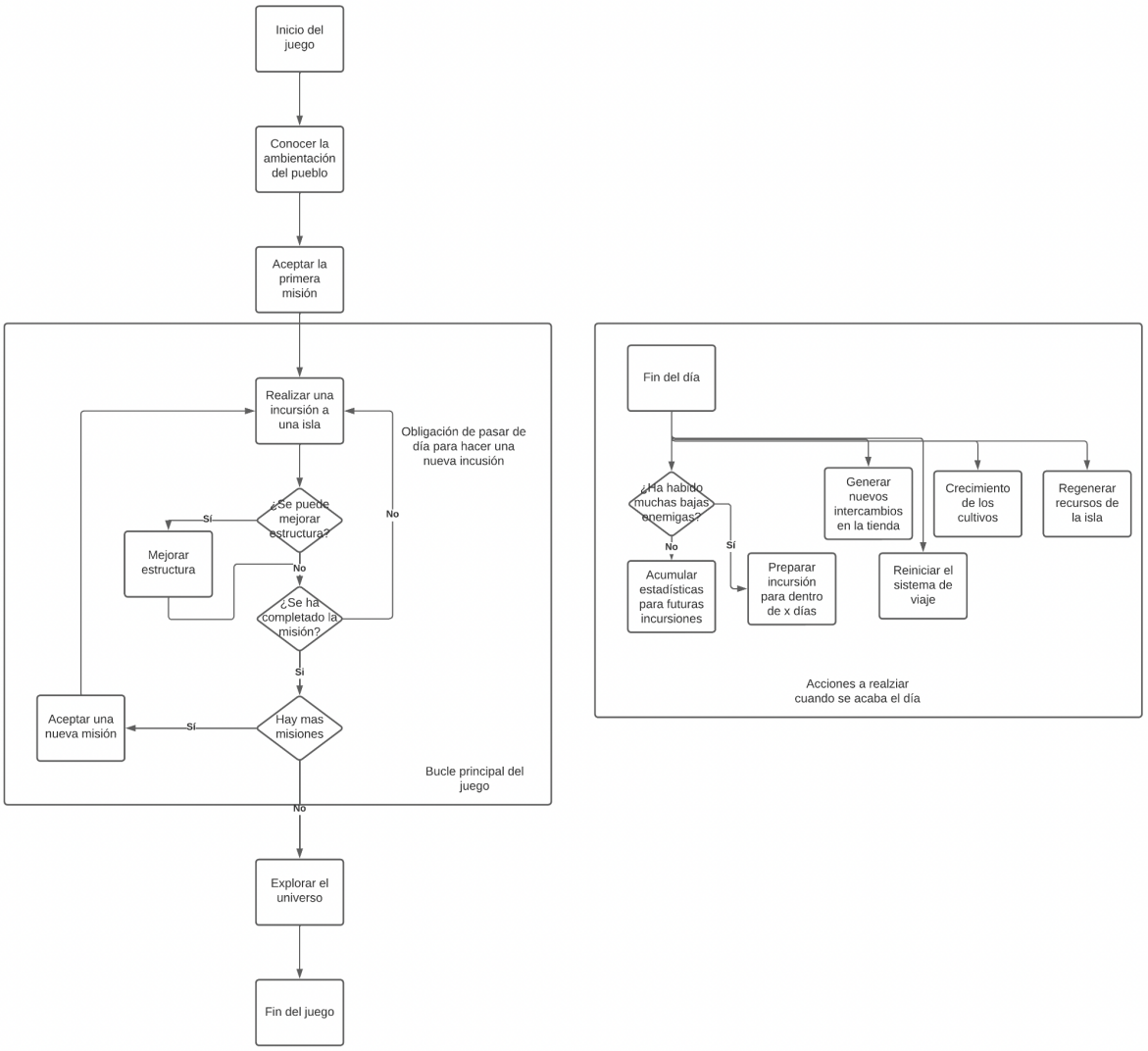


Figura 5.5.1: Esquema que resume el funcionamiento principal del juego

En este diagrama podemos ver el funcionamiento general de todo el juego, visto a alto nivel. Hay dos estructuras principales, donde una está designada a explicar el flujo

principal del juego, la otra se encarga de comentar las acciones que ocurren una vez que se ha finalizado el día.

Centrándonos en el diagrama de la izquierda, podemos ver como se repite una estructura importante, ya que forma la parte base del juego, que consiste en la exploración y la mejora de las estructuras. Cabe destacar que en este diagrama no se representa el hecho de tener que afrontar una incursión a la isla del jugador, los cuales son eventos esporádicos que no suceden constantemente.

En el diagrama de la derecha se explican las acciones que se llevan a cabo una vez que se ha acabado el día, las cuales son de vital importancia para que el jugador pueda seguir disfrutando de la experiencia.
Lo explicado anteriormente se verá implementado en versiones futuras, aunque actualmente nos podemos encontrar con el bucle de aceptar misiones y el de regeneración de recursos, solo que este último va en función del tiempo transcurrido y no del ciclo día-noche.

5.6 Player inputs

El juego está desarrollado de tal forma que sea posible jugar con cualquier tipo de mando o teclado, ya que con esto consigue llegar a un mayor público objetivo, aunque todo el sistema de feedback está pensado para la utilización de un mando de Xbox o un teclado.

A la hora de establecer qué tecla se encargaría de hacer cada función, hemos hecho un análisis del mercado de videojuegos que cuentan con esta funcionalidad e intentar replicar las que nos parecen más útiles para la comodidad del jugador.

Por convención, se utilizan las teclas WASD para el movimiento del mismo, ya que resulta cómodo para la disposición de la mano en reposo. Siguiendo con este razonamiento, se ha utilizado las teclas F y G para las acciones de talar y picar, ya que simplemente se necesita desplazar un dedo para realizarlas, al igual que con la tecla de interactuar, que se realiza con la tecla E. Para las acciones de atacar, bloquear y ataque especial nos encontramos con que se realizan con el ratón, lo cual es una convención ya en el sector de los videojuegos, utilizando que click izquierdo para atacar, el derecho para bloquear y el click central para el ataque especial.

6. Implementación y pruebas

A lo largo de este apartado hablaremos de la implementación en unity de nuestro vídeojuego, explicando las partes importantes, de tal forma que sea sencillo entender el funcionamiento de cada gameobject con sus respectivos scripts.

6.1 Implementación de los niveles

Un videojuego está formado por escenas, en las que acontecen todos los eventos de nuestro juego. De forma general se suelen ordenar según su acontecimiento cronológico.

6.1.1 Estructura de las escenas

Para implementar los distintos niveles hemos utilizado dos escenas, donde la primera nos sirve como menú principal del juego y la segunda como el escenario principal del proyecto. En la Figura 6.1.1 se ven estas escenas.

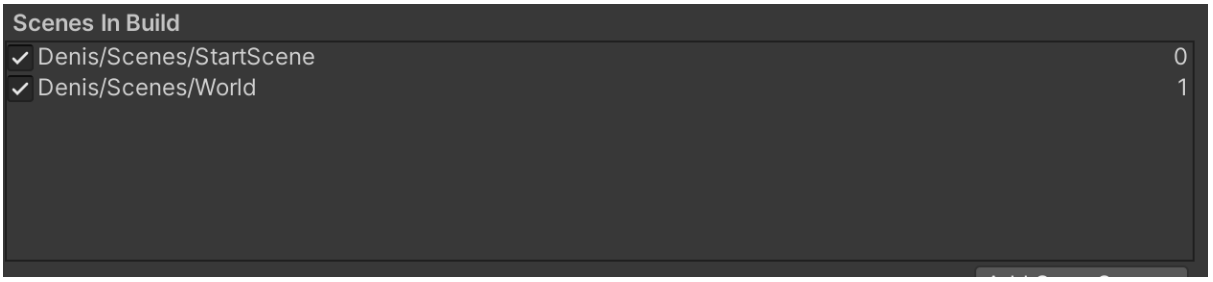


Figura 6.1.1: Captura de pantalla de las escenas que componen nuestro videojuego

6.1.2 Start Scene

La escena Start Scene es la primera que se ejecuta cuando abrimos el juego y que funciona como menú de inicio. En esta escena el jugador puede elegir entre empezar una nueva partida, cargar una partida o salir del juego. En esta escena hay un script principal que se encarga de controlarla, llamado "**Main Menu**", aunque para su correcto funcionamiento también cuenta con otros diversos scripts que explicaremos a continuación.

Clase Main Menu

Esta clase se encarga de controlar el funcionamiento base del menú de inicio, que cuenta con métodos públicos para que los diferentes botones puedan realizar las acciones necesarias.

- **Transition()**: método principal de esta clase, que recibe como parámetro un booleano, el cual nos permite saber si tenemos que cargar la información de la partida anterior, o si simplemente es una partida nueva.

  Lo primero que realiza esta clase es buscar el objeto fader en la escena y esperar a que acabe la animación de FadeOut. Una vez que ha acabado la animación, carga asíncronamente la escena del escenario donde transcurre todo el videojuego. Para poder cargar / empezar una nueva partida, necesitamos buscar el objeto SavingWrapper, que, en función de nuestras necesidades, borrará o cargará el archivo de guardado que se encuentra en nuestro sistema. Estas acciones se realizan con los métodos "**LoadSceneInfo**" y "**DestroyGame**".

  Una vez que todo este proceso se ha acabado, se realiza la animación de FadeIn.
  El método "**DontDestroyOnLoad**", que es propio de Unity, nos hace falta para que, a la hora de cargar la escena, no se elimine el gameobject actual y poder cargar toda la información de la partida, ya que de otra forma, se eliminaría el gameobject antes de llegar a esta parte del código.

```
private IEnumerator Transition(bool loadGame)
  {

      Cursor.visible = false;

      DontDestroyOnLoad(gameObject);

      Fader fader = FindObjectOfType<Fader>();

      yield return fader.FadeOut(fadeOutTime);

      yield return SceneManager.LoadSceneAsync(1);

      if (loadGame)
      {
         LoadSceneInfo();
      }
      else
      {
         DestroyGame();
      }

      yield return new WaitForSeconds(fadeWaitTime);
      Time.timeScale = 1f;
      yield return fader.FadeIn(fadeInTime);

      Destroy(gameObject);

      yield return null;

  }
```

- **StartNewGame()**: este método empieza una corrutina con la función Transition(), pasándole como parámetro un boolenao establecido a falso para que no cargue ninguna partida.
- **LoadGame()**: este método hace lo mismo que el anterior, pero con la diferencia que pasa un booleando establecido a verdadero para que cargue la partida guardada.
- **QuitGame()**: este método se encarga de cerrar el juego llamando al método Application.Quit() propio de Unity.

Clase Persistent Object Spawner

Esta clase se encarga de comprobar si se ha instanciado el gameObject llamado Fader, y en caso de que no haya sido instanciado, llama a un método llamado **"SpawnPersistentObject",** el cual lo instancia y establece que Unity no lo destruya cuando se cambie de escena.

```
[SerializeField] GameObject persistentObjectPrefab;

    static bool hasSpawned = false;

    private void Awake()
    {
        if (hasSpawned) return;

        SpawnPersistentObject();

        hasSpawned = true;
    }

    private void SpawnPersistentObject()
    {
        GameObject obj = Instantiate(persistentObjectPrefab);
        DontDestroyOnLoad(obj);
    }
```

Clase Input Mode:

Esta clase se encarga de controlar que sistema de inputs se está utilizando para poder cambiar la información del feedback que se le proporciona al jugador y otros cuantos elementos más del juego para el buen funcionamiento del mismo. Esta clase es común a todas las escenas.

- **Start()**: Este método nos permite establecer un primer sistema de inputs, que por lo general suele ser el teclado. Lo primero que hace es buscar el sistema de input actual, y le pasa esta información al método "**UpdateControlScheme"**, que notifica este cambio de sistema de inputs.

```
void Start()
  {
     PlayerInput input = FindObjectOfType<PlayerInput>();
     UpdateControlScheme(input.currentControlScheme);
     Time.timeScale = 1f;
  }
```

Los siguientes métodos a comentar de esta clase son los que nos permiten controlar cuando se tiene que estar observando los cambios de inputs:

- El método **OnEnable()** y **OnDisable()** se encargan de vincular / desvincular el sistema de alertas del InputUser al método propio llamado **onInputDeviceChange**.
- El método **onInputDeviceChange** se encarga de comprobar si efetivamente se ha cambiado el sistema de inputs del juego y en caso de que sea cierto, llama al método **UpdateControlScheme()** con el nombre del sistema de control actual.

```
void OnEnable()
  {
     InputUser.onChange += onInputDeviceChange;
  }

  void OnDisable()
  {
     InputUser.onChange -= onInputDeviceChange;
  }

  void onInputDeviceChange(InputUser user, InputUserChange change, InputDevice
device)
  {
     if (change == InputUserChange.ControlSchemeChanged)
     {
        UpdateControlScheme(user.controlScheme.Value.name);
     }
  }
```

- Como último método a comentar tenemos el **"UpdateControlScheme()"**, el cual recibe como parámetro el nombre del esquema de control que se está utilizando en ese momento, y notifica a los elementos con texto y botones este cambio para que ajusten su comportamiento a este nuevo sistema de inputs.

```
void UpdateControlScheme(string schemeName)
  {
     // assuming you have only 2 schemes: keyboard and gamepad
     if (schemeName.Equals("Gamepad"))
     {

        foreach (LookAtCamera item in FindObjectsOfType<LookAtCamera>())
        {
           item.SetGamepadText();
        }
        foreach(ButtonMusicEvents item in FindObjectsOfType<ButtonMusicEvents>())
        {
           item.UpdateInputmode(false);
        }
     }
     else
     {
        foreach (LookAtCamera item in FindObjectsOfType<LookAtCamera>())
        {
           item.SetKeyboardText();
        }
        foreach (ButtonMusicEvents item in FindObjectsOfType<ButtonMusicEvents>())
        {
           item.UpdateInputmode(true);
        }
     }
  }
```

Clase BlockFrameRate

Esta clase también es común a todas las escenas, ya que su finalidad es establecer el máximo número de fps a los que se ejecutará el videojuego.

- **Awake()**: este método desactiva el vSync y establece el targetFrameRate a 60 para que el juego se ejecute a 60FPS.

```
private void Awake()
   {
      QualitySettings.vSyncCount = 0;
      Application.targetFrameRate = 60;
   }
```

Clase Decrease Music Volume

Esta clase cuenta con un único método llamado "**DecreaseVolumeOnClick**", el cual se encarga de reducir el volumen de la música del menú cuando el jugador hace clic en cualquiera de los botones.

```
[SerializeField] AudioSource audioSource;
  [SerializeField] float speed;

  public void DecreaseVolumeOnClick()
  {
    while(audioSource.volume > 0)
    {
      audioSource.volume -= Time.deltaTime* speed;
    }
  }
```

Clase Button Music Events

Esta clase se encarga de reproducir los sonidos correspondientes según las acciones que ocurran en los botones. Además, esta clase es común en todas las escenas que incorporen botones. Cabe destacar que mediante el parámetro IsPauseMenu, nos permite reutilizar este script para el menú de pausa, ya que en este caso necesitamos una fuente de sonido que no está implícita en el botón.

- **Start()**: en este método se comprueba si el botón es del menú de pausa, y en caso de no serlo, se obtiene el componente AudioSource del objeto al que está enlazado.
- **IsButtonSelected() / IsButtonSelectedWithGamepad()**: este método se llama cuando el jugador pasa el ratón por encima del botón o selecciona el botón con el mando. Se encarga de reproducir el sonido identificativo de que el botón está enmarcado.
- **IsButtonClicked()**: en este método se reproduce el sonido identificativo conforme a que se ha hecho clic en el botón.

6.1.3 World Scene

En esta escena es donde sucede todo el desarrollo del juego. En este apartado hablaremos de los scripts que controlan esta escena, dejando para otros apartados los componentes que la integran, como pueden ser el jugador, enemigos, etc.

Clase Reload Game On Death

Esta clase se encarga de reiniciar el nivel cuando el jugador ha muerto.

- **Reload Scene()**: este método se encarga de llamar a una corrutina que realiza una transición mientras que se carga la información para reiniciar la partida. El código es idéntico al de "**Transition**", explicado para la clase Main Menu.

Clase Pause Menu Controller

El trabajo de esta clase consiste en mantener un control de cuándo se debe activar y desactivar el menú, además de contener todas las funciones necesarias para el funcionamiento del mismo.

- **OnEnable() / OnDisable()** : La primera función nos permite vincular la tecla escape desde el script y establecer que cada vez que se pulse, se inicie la función **PauseGame().** La función OnDisable se encarga de romper este vínculo.

```
private void OnEnable()
    {
        menu = controls.Menu.Escape;
        menu.Enable();

        menu.performed += PauseGame;

    }

    private void OnDisable()
    {
        menu.Disable();
    }
```

- **PauseGame()**: este método se encarga de activar y desactivar el menú de pausa en función de su estado anterior. También se encarga de controlar que no se pueda abrir si el jugador está visualizando el inventario.

```
private void PauseGame(InputAction.CallbackContext context)
  {
    if (!GameObject.FindGameObjectWithTag("CityHall").GetComponent<TownHallController>().IsInventoryOpen())
    {
      isPaused = !isPaused;

      if (isPaused)
      {
        ActivateMenu();
      }
      else
      {
        DeactivateMenu();
      }
    }
  }
```

- **ActivateMenu()** :Este método se encarga de ajustar todo para poder visualizar el menú de una manera óptima. Establece como visibles el cursor, la interfaz de pausa, cambia la escala de tiempo a 0, desactiva el recordatorio de misión, y cambia el esquema de control de jugador a menú.

```
private void ActivateMenu()
  {
    Cursor.visible = true;
    missionInfoHUD.SetActive(false);
    GameObject.FindGameObjectWithTag("Player").GetComponent<PlayerMovment>().ChangeInputMode(PlayerMovment.ControlSchemne.Menu);
    Time.timeScale = 0;
    pauseUI.SetActive(true);
  }
```

- **DeactivateMenu()**: Este método realiza lo mismo que el anterior, sólo que al contrario, para poder volver a la rutina de funcionamiento del juego.

```
public  void DeactivateMenu()
  {
    Cursor.visible = false;
    missionInfoHUD.SetActive(true);
    GameObject.FindGameObjectWithTag("Player").GetComponent<PlayerMovment>().ChangeInputMode(PlayerMovment.ControlSchemne.Player);
    Time.timeScale = 1;
    pauseUI.SetActive(false);
    isPaused = false;
  }
```

A partir de ahora hablaremos de los métodos propios que permiten el buen funcionamiento del menú de pausa.

- **SaveGame() / LoadGame()** : estos métodos se encargan de buscar el objeto llamado SavingWrapper y llamar al método Save() / Load() respectivamente. Una vez hecho esto, desactivan el menú de pausa.

```
public void SaveGame()
    {
        FindObjectOfType<SavingWrapper>().Save();
        DeactivateMenu();
    }

    public void LoadGame()
    {
        FindObjectOfType<SavingWrapper>().Load();
        DeactivateMenu();
    }
```

- **ReturnToMainMenu()**: este método llama a la corrutina que hemos explicado anteriormente llamada "Transition", sólo que cuenta con la diferencia de que esta vez carga la escena 0, que es la del menú principal.
- **QuitGame()**: Este método cierra el juego.

```
public void QuitGAme()
    {
        Application.Quit();
    }
```

6.2 Sistema de guardado

Un apartado que iremos viendo repetido a lo largo de este proyecto es el del sistema de guardado, ya que cada entidad implementará su interfaz de guardado para almacenar en disco toda la información necesaria. Para poder hacer esto, necesitaremos unos scripts determinados que explicaremos a continuación.

6.2.1 Saving wrapper

La clase SavingWrapper es la encargada de interactuar con el sistema de guardado a petición de entidades externas a este sistema.

- **Load()**: este método permite cargar la partida a través del SavingSystem, llamando al método Load, al cual le pasamos un nombre para identificar cual es el archivo a cargar.

```
public void Load()
    {
        //Call to saving system
        savingSystem.Load(defaultSaveFile);
    }
```

- **Save()**: este método permite realizar un guardado a través del SavingSystem, llamando al método Save, al cual le pasamos un nombre para guardar el archivo.

```
public void Save()
    {
        savingSystem.Save(defaultSaveFile);
    }
```

- **DestroySavedFile()**: con este método se nos permite borrar el archivo de guardado, pasándole el nombre del archivo al savingSystem a través del método Delete.

```
public void DestroySavedFile()
    {
        savingSystem.Delete(defaultSaveFile);
    }
```

6.2.2 Isaveable:

Esta interfaz establece los métodos a través de los cuales, las entidades que quieran guardar cualquier archivo, tienen que implementarlo.

Los métodos con los que cuenta son "**CaptureState()**", el cual regresa un objeto que es el cual se guarda, y el método **"RestoreState()",** a través del cual se le pasa al objeto su estado anterior; y el propio objeto se encarga de implementar el sistema de recuperación del estado anterior.

```
public interface ISaveable
{
    object CaptureState();
    void RestoreState(object state);
}
```

6.2.3 Saveable Entity

Este script se añade a todas los gameobjects que tengan información de importancia que quieran guardar, ya que se encarga de generar un identificador único al que asociar esta información, y busca todas las interfaces descritas anteriormente que se encuentren dentro de ese gameobject.

Lo primero que tenemos que hacer es generar un identificador único que pertenezca a un solo objeto, sin importar la escena en la que se encuentre.

- **IsUnique()**: este método recibe como parámetro un string que es el identificador de propuesta, al cual se le aplican unas cuantas comprobaciones para ver si realmente es válido.
    - Lo primero que comprobamos es si existe en el diccionario. Si no existe, es un identificador válido.
    - Lo siguiente a comprobar es que en caso de que exista, sea una referencia a sí mismo, por lo cual también sería válido.
    - También se puede considerar que la clave esté utilizada pero el contenido de la misma sea nulo, en cuyo caso se borrará la clave y se considerará válida.
    - Por último se comprueba que si en caso de que exista el identificador, se accede al gameObject y se comprueba que sea el mismo en ambos casos, para evitar que haya inconsistencia en la información.
    - En caso de que no se cumpliera ninguna de las condiciones anteriores, se considerará no válida.

```
private bool IsUnique(string candidate)
  {
    if (!globalLookup.ContainsKey(candidate)) return true;

    if (globalLookup[candidate] == this) return true;

    if (globalLookup[candidate] == null)
    {
      globalLookup.Remove(candidate);
      return true;
    }

    if (globalLookup[candidate].GetUniqueIdentifier() != candidate)
    {
      globalLookup.Remove(candidate);
      return true;
    }

    return false;
  }
```

La siguiente parte del proceso sólo se ejecuta cuando estamos en el modo editor, por lo cual no se incluirá cuando hagamos la build del proyecto.

- **Update()** : en esta parte del código es donde se asigna el identificador que tendrá el objeto, por lo cual es necesario que sólo se ejecute en unos momentos muy concretos.
    - La primera condición es que no se puede ejecutar si el gameobject se está ejecutando.
    - También es necesario que no se ejecute el código si el gameobject no está instanciado en alguna escena determinada.

Una vez que ya estamos en las condiciones óptimas de ejecución, accedemos al sistema de guardado de información serializada del script correspondiente al gameobject y nos guardamos el valor correspondiente al "uniqueIdentifier". Lo siguiente a realizar es comprobar si es válido, ya sea por motivos de ser nulo o no contener nada o por no ser único. Si se da el caso de que este id no sea válido, se genera uno aleatorio mediante las llamadas internas de Unity, se guarda en las propiedades del script y se asigna al diccionario. En caso de que este id no fuera válido, se comprobará en la siguiente iteración.

```
#if UNITY_EDITOR
    private void Update()
    {
        if (Application.IsPlaying(gameObject)) return;
        if (string.IsNullOrEmpty(gameObject.scene.path)) return;

        SerializedObject serializedObject = new SerializedObject(this);
        SerializedProperty property = serializedObject.FindProperty("uniqueIdentifier");

        if (string.IsNullOrEmpty(property.stringValue) || !IsUnique(property.stringValue))
        {
            property.stringValue = System.Guid.NewGuid().ToString();
            serializedObject.ApplyModifiedProperties();
        }

        globalLookup[property.stringValue] = this;
    }
#endif
```

Los métodos que explicaremos a continuación son los encargados de obtener y enviar la información del sistema de guardado que implementa cada script dentro del gameobject.

- **CaptureState()**: este método se encarga de crear un diccionario donde poder almacenar la información del gameobject. Una vez hecho esto, recorre todos los componentes del gameobject que implementan la interfaz explicada anteriormente y las guarda en el diccionario, donde la clave es el nombre del script que implementa la interfaz y el object es el contenido de la información a guardar.

```
public object CaptureState()
    {
        Dictionary<string, object> state = new Dictionary<string, object>();
        foreach (ISaveable saveable in GetComponents<ISaveable>())
        {
            state[saveable.GetType().ToString()] = saveable.CaptureState();
        }
        return state;
    }
```

- **RestoreState()**: este método recibe como parámetro un object, al cual le hacemos un cast a diccionario para poder trabajar con el. Una vez que ya tenemos este diccionario, buscamos todas los componentes que implementan la interfaz descrita en el apartado anterior y restauramos el estado que obtenemos del diccionario accediendo a él a través del nombre del componente.

```
public void RestoreState(object state)
    {
        Dictionary<string, object> stateDict = (Dictionary<string, object>)state;
        foreach (ISaveable saveable in GetComponents<ISaveable>())
        {
            string typeString = saveable.GetType().ToString();
            if (stateDict.ContainsKey(typeString))
            {
                saveable.RestoreState(stateDict[typeString]);
            }
        }
    }
```

6.2.4 Saving System:

Este script se encarga de gestionar todo el sistema de guardado, implementando todos los métodos necesarios para poder llevarlo a cabo.

Lo primero que necesitamos saber es donde se guardan los archivos, utilizando la propia gestión interna de Unity.

- **GetPathFromSaveFile():** este método recibe como parámetro el nombre del archivo que queremos recuperar, y nos regresa el path donde tenemos que buscar este archivo.

```
private string GetPathFromSaveFile(string saveFile)
    {
        return Path.Combine(Application.persistentDataPath, saveFile + ".sav");
    }
```

Una vez que ya contamos con el path del archivo que queremos buscar, podemos llamar a una función que se encargue de hacerlo.

- **LoadFile()**: esta función recibe como parámetro el nombre del archivo que queremos buscar, y consigue el path del mismo con la función que hemos explicado anteriormente. Mediante el método File.Exist() comprueba si existe el archivo al que queremos acceder. En caso de que este archivo no exista, retorna un diccionario vacío y en caso de que exista, lo abre con los métodos propios de Unity, y los deserializa del formato binario para poder trabajar con ellos a modo de diccionario, y retorna este archivo como resultado.

```
private Dictionary<string, object> LoadFile(string saveFile)
  {
    //Esta función se encarga de obtener el archivo guardado y en caso de que no exista, crea un diccionario nuevo
    string path = GetPathFromSaveFile(saveFile);
    if (!File.Exists(path))
    {
      return new Dictionary<string, object>();
    }
    using (FileStream stream = File.Open(path, FileMode.Open))
    {
      BinaryFormatter formatter = new BinaryFormatter();
      return (Dictionary<string, object>)formatter.Deserialize(stream);
    }
  }
```

Una vez que ya contamos con el diccionario, el cual contiene toda la información de la última partida guardada del jugador, es hora de asignar toda esta información a los gameobjects correspondientes.

- **RestoreState()**: este método recibe como parámetro un diccionario que contiene toda la información. Para poder cargar toda esta información de una forma ordenada, buscamos todas las entidades que contengan el componente "SaveableEntity" y si su identificador se encuentra dentro del diccionario, se le asigna la información correspondiente mediante el método RestoreState del script mencionado anteriormente.

```
private void RestoreState(Dictionary<string, object> state)
  {
    foreach (SaveableEntity saveable in FindObjectsOfType<SaveableEntity>())
    {
      string id = saveable.GetUniqueIdentifier();
      if (state.ContainsKey(id))
      {
        saveable.RestoreState(state[id]);
      }
    }
  }
```

El método público **"Load()"** se encarga de llamar a estos dos últimos métodos y para poder realizar la acción de cargar la última partida guardada.

```
public void Load(string saveFile)
  {
    RestoreState(LoadFile(saveFile));
  }
```

El siguiente conjunto de métodos a definir, explican cómo se realiza el sistema de guardado.

- **CaptureState()**: este método recibe como parámetro un diccionario que contiene toda la información del estado anterior. Se encarga de buscar todas las entidades que posean el script "SaveableEntiry" guardar su estado en el diccionario con la ayuda del método de la entidad llamado "CaptureState". También guarda la información de cuál fue la última escena que estaba abierta.

```
private void CaptureState(Dictionary<string, object> state)
    {
    foreach (SaveableEntity saveable in FindObjectsOfType<SaveableEntity>())
    {
        state[saveable.GetUniqueIdentifier()] = saveable.CaptureState();
    }
    state["lastSceneBuildIndex"] = SceneManager.GetActiveScene().buildIndex;
}
```

- **SaveFile()**: este método nos permite guardar en disco la información que hemos obtenido en el paso anterior. Este método recibe como parámetro el nombre del archivo que necesitamos editar/crear y el objeto, que es el diccionario que hemos creado con anterioridad. Lo primero que se hace en este método es obtener el path del archivo y abrirlo en modo create, para que en caso de que ya exista, se sobreescriba. Para poder guardar esta información necesitamos serializarla a un formato binario.

```
private void SaveFile(string saveFile, object state)
    {
        string path = GetPathFromSaveFile(saveFile);
        print("Saving to " + path);
        using (FileStream stream = File.Open(path, FileMode.Create))
        {
            BinaryFormatter formatter = new BinaryFormatter();
            formatter.Serialize(stream, state);
        }
    }
```

- **Save()**: este método público es el cual permite a las demás clases acceder al sistema de guardado, a través del cual se abre el archivo de guardado, se captura el estado actual y se guarda el archivo.

```
public void Save(string saveFile)
    {
        Dictionary<string, object> state = LoadFile(saveFile);
        CaptureState(state);
        SaveFile(saveFile, state);
    }
```

Como final de este apartado, tenemos dos apartados menos importantes pero también necesarios para el buen funcionamiento de este proyecto.

- **Delete()**: este método se encarga de buscar el path del archivo y eliminarlo.

```
public void Delete(string saveFile)
    {
        File.Delete(GetPathFromSaveFile(saveFile));
    }
```

- **LoadLastScene()**: este método se encarga de recuperar el archivo de guardado, buscar con la clave “lasSceneBuildIndex” y cargar la escena. Cabe destacar que en caso de no encontrar esta clave, se carga la escena actual. Además, recupera el estado actual de todos los elementos del juego.

```
public IEnumerator LoadLastScene(string saveFile)
    {
        Dictionary<string, object> state = LoadFile(saveFile);
        int buildIndex = SceneManager.GetActiveScene().buildIndex;
        if (state.ContainsKey("lastSceneBuildIndex"))
        {
            buildIndex = (int)state["lastSceneBuildIndex"];
        }
        yield return SceneManager.LoadSceneAsync(buildIndex);
        RestoreState(state);
    }
```

6.2.5 SerializableVector3

Por defecto no se puede guardar un Vector3, por lo cual necesitamos crear esta clase que nos permite hacer eso descomponiendo este Vector3 en 3 float, los cuales sí que se pueden guardar. Además cuenta con un método que nos permite obtener directamente el Vector3.

```
[System.Serializable]
public class SerializableVector3
{
    float x, y, z;

    public SerializableVector3(Vector3 vector)
    {
        x = vector.x;
        y = vector.y;
        z = vector.z;
    }

    public Vector3 ToVector()
    {
        return new Vector3(x, y, z);
    }
}
```

6.3 Sistema de input

Para la realización de este proyecto se ha optado por la utilización del nuevo sistema de input de Unity, ya que por un lado da una versatilidad mayor, y por otro nos permite aprender el funcionamiento del mismo para ser punteros en el sector de desarrollo.

6.3.1 Incorporación al proyecto

Debido a la novedad del mismo, es un paquete que no se encuentra por defecto, por lo que es necesario descargarlo desde el gestor de paquetes del proyecto. Una vez que ya contamos con este paquete, necesitamos crear un Input Action Asset, ya que se encargará de almacenar todos los eventos que se pueden hacer en nuestro proyecto y las teclas con las cuales se activa.

6.3.2 Utilización del paquete

Para poder utilizar este recurso, necesitamos añadir un componente a algún gameobject (que en nuestro caso es el propio personaje principal) llamado Player Input. En este caso lo hemos configurado para que pueda cambiar entre teclado y ratón automáticamente, y que el modo de utilización del mismo sea a través de eventos, lo cual nos favorecía para poder tener un plus de independencia de código. Ver Figura 6.3.2.1.

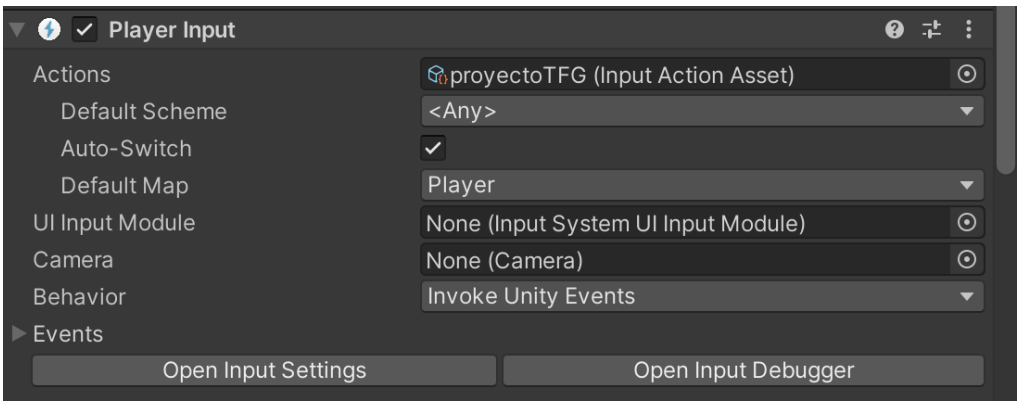


Figura 6.3.2.1: Captura de pantalla del sistema de inputs

6.3.3 Creación de los Action Maps

Para poder utilizar los controles que teníamos en mente, necesitamos crear un sistema de mapas que cumpla con nuestras necesidades en cada momento, ya sea para el movimiento del mismo jugador, como para el desplazamiento dentro de los menús. Para la realización de esto, hemos necesitado crear 3 mapas distintos, los cuales podemos definir como “Player”, “Menu”, “Inventory”. Ver Figura 6.3.3.1.

Figura 6.3.3.1: Captura de pantalla del sistema de Action

En la figura anterior podemos ver un action map llamado “UI”, el cual lo hemos utilizado como sistema de guía para la navegación de interfaces, y lo hemos preferido conservar para tener un ejemplo en el futuro, por si añadimos nuevos sistemas.

6.3.4 Creación de las Actions de cada mapa

Para vincular las acciones que realiza el jugador a la hora de pulsar las teclas, necesitamos crear estas acciones, que nos permite realizar justamente eso.

Player

El mapa de player es uno de los más extensos , ya que incorpora todas las acciones que tiene que realizar el jugador para poder interactuar con las mecánicas.

- Move: esta acción es del tipo valor numérico (Vector2), el cual se puede interactuar con el stick izquierdo de un mando, con las teclas WASD y las flechas. Ver Figura 6.3.4.1.

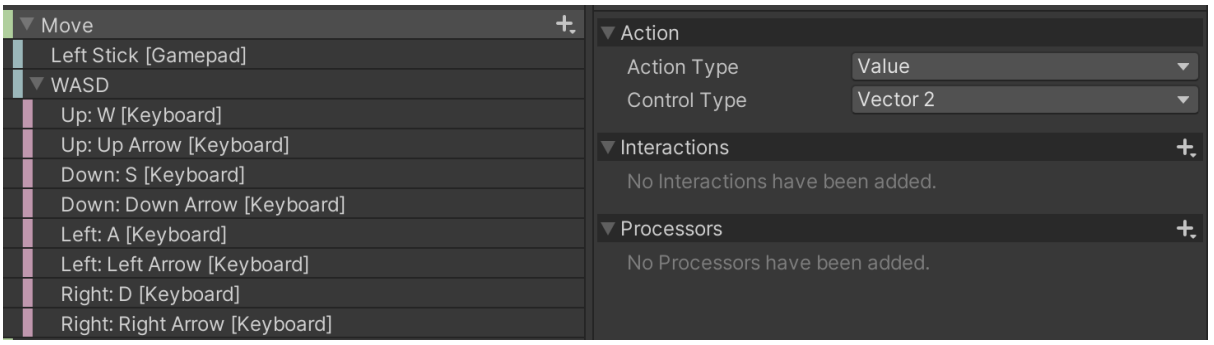


Figura 6.3.4.1: Captura de pantalla de la acción mover

- Run: esta acción es del tipo botón, que detecta cuando se ha pulsado. Se puede interactuar con la tecla “Shift” o pulsando el joystick izquierdo. Cuenta con valor inicial establecido a falso. Ver Figura 6.3.4.2.

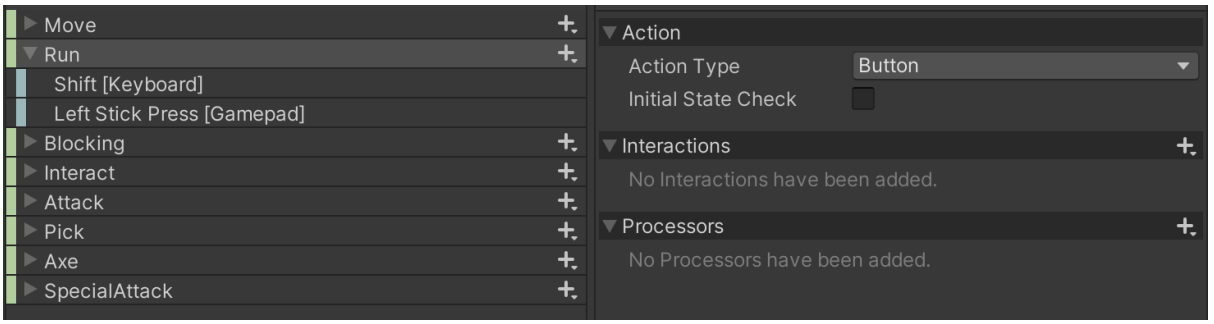


Figura 6.3.4.2: Captura de pantalla de la acción mover

- Blocking: esta acción es de tipo botón, con la peculiaridad de que para interactuar con ella, se pone la condición de que esta tiene que efectuarse por mantener pulsado el botón. El modo de interacción se efectúa por pulsar el clic derecho o el botón LB del mando. Ver Figura 6.3.4.3.

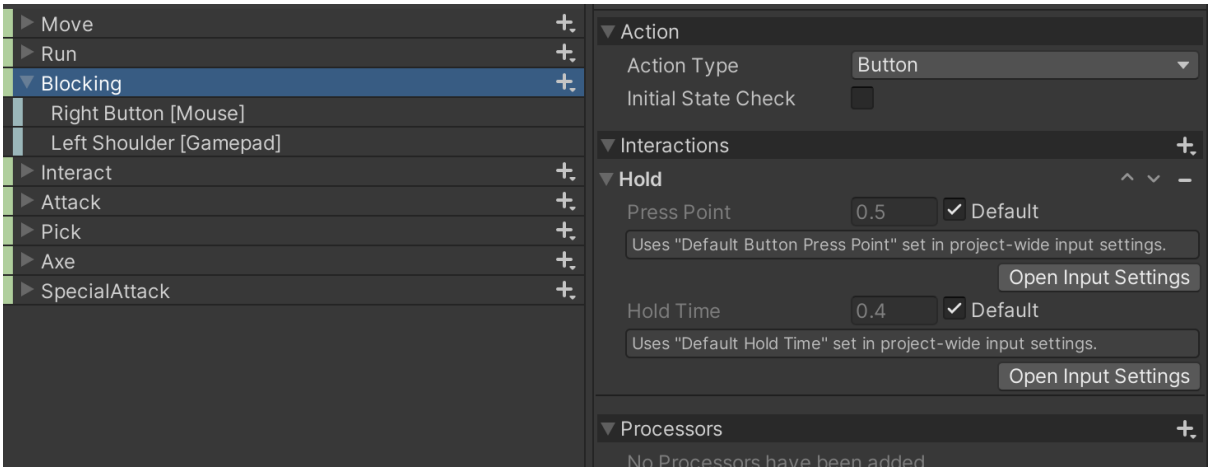


Figura 6.3.4.3: Captura de pantalla de la acción bloquear

- Interact: esta acción es del tipo botón, donde la condición de interacción es del tipo “press” con la finalidad de que solo se realice una vez. Las teclas con las que se puede interactuar con este evento son la “E” y el botón “Y” del mando. Ver Figura 6.3.4.4.

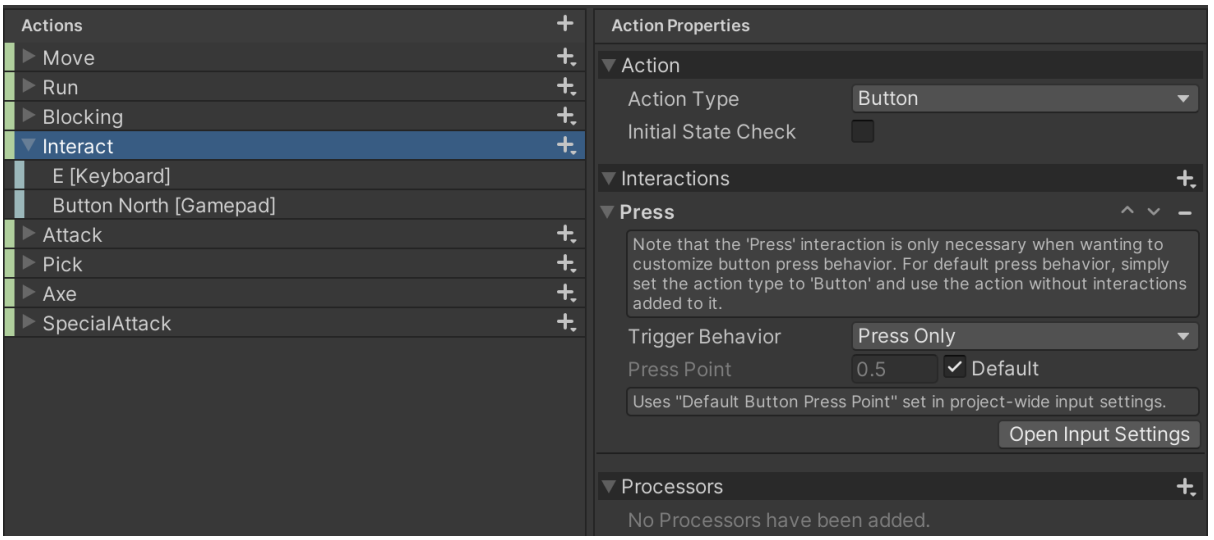
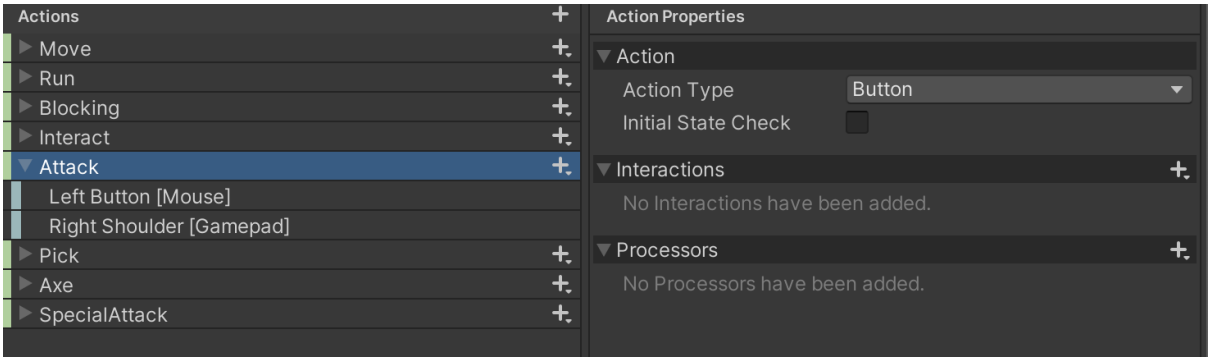


Figura 6.3.4.4: Captura de pantalla de la acción interactuar

- Attack: esta acción también es del tipo botón donde las maneras de interactuar son el clic izquierdo y el botón RB. Ver Figura 6.3.4.5.



- Figura 6.3.4.5: Captura de pantalla de la acción atacar

- Pick: acción de tipo botón, donde las teclas de interacción son la F en teclado y la A en mando. Ver Figura 6.3.4.6.

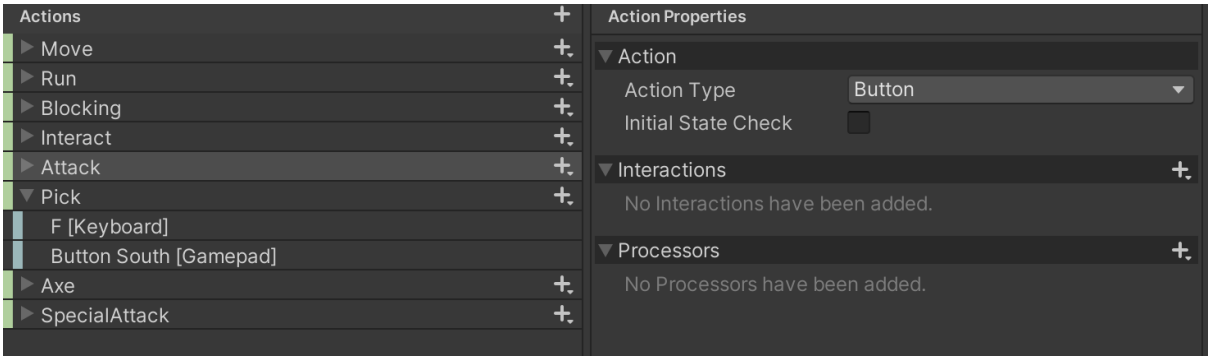


Figura 6.3.4.6: Captura de pantalla de la acción picar

- Axe: acción de tipo botón, donde las teclas de interacción son la G en teclado y la X en mando. Ver Figura 6.3.4.7.

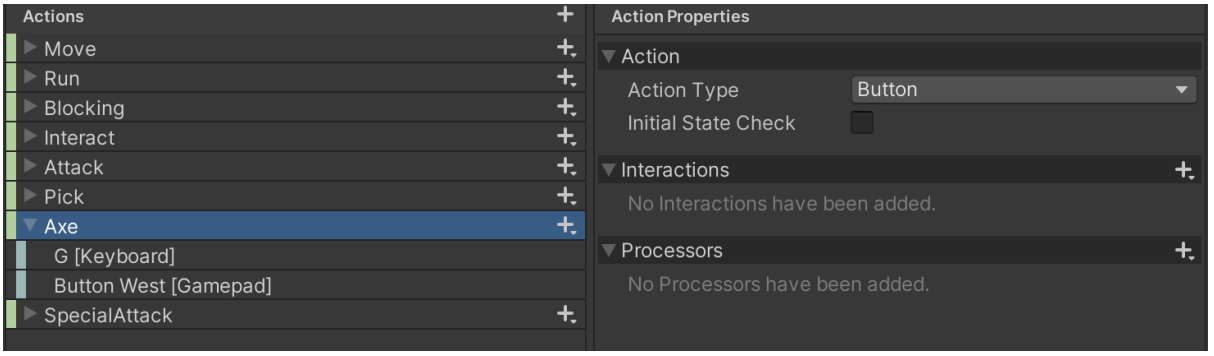


Figura 6.3.4.7: Captura de pantalla de la acción talar

- Special Attack: acción de tipo botón, donde las teclas de interacción son el botón central del rayón y RT en mando. Ver Figura 6.3.4.7.

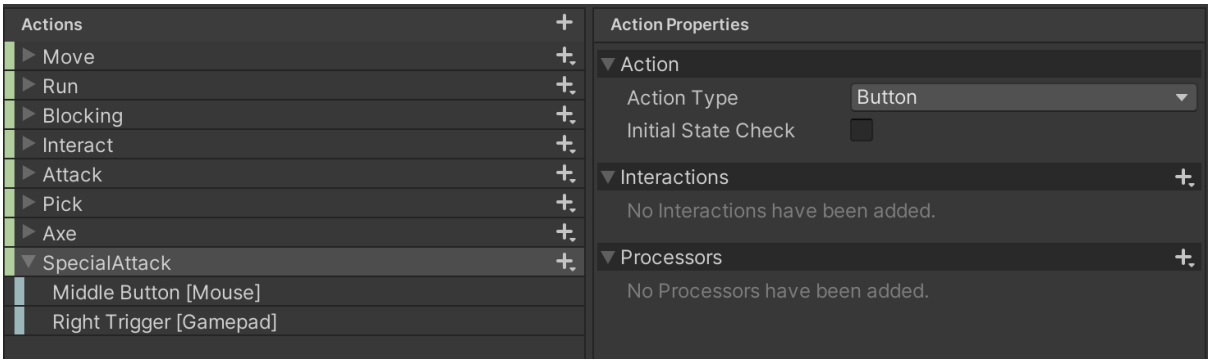


Figura 6.3.4.7: Captura de pantalla de la acción talar

Menu

Este mapa de acciones es bastante básico, ya que para esta versión del proyecto no necesitamos más acciones.

- Escape: acción de tipo botón, donde las teclas de interacción son la ESC en teclado y la Start en mando. Ver Figura 6.3.4.8.

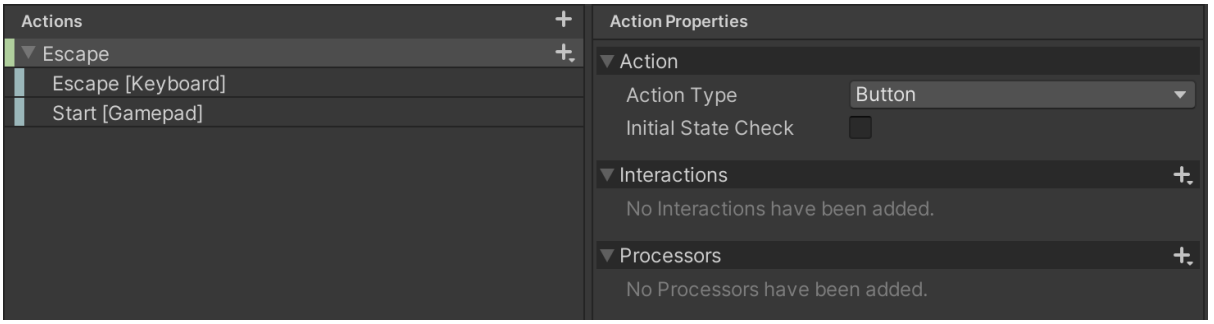


Figura 6.3.4.8: Captura de pantalla de la acción cerrar menú

Inventory

Este mapa de acciones es bastante básico, ya que para esta versión del proyecto no necesitamos más acciones.

- Close: acción de tipo botón, donde las teclas de interacción son la ESC en teclado y la B en mando. Ver figura 6.3.4.9.

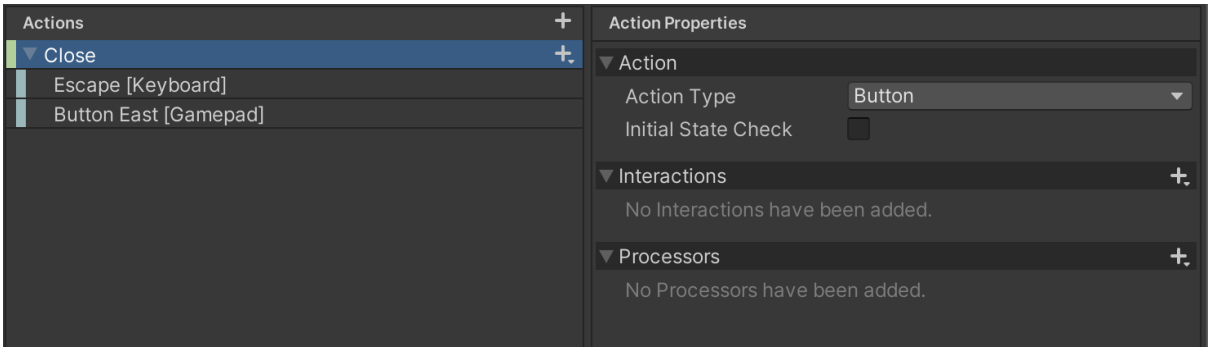


Figura 6.3.4.9: Captura de pantalla de la acción cerrar inventario

6.3.5 Sistema de bind

Para poder utilizar las acciones descritas anteriormente, necesitamos enlazarlas a las funciones de un script, lo cual se realiza mediante el componente player input.

- Player, ver Figura 6.3.5.1:

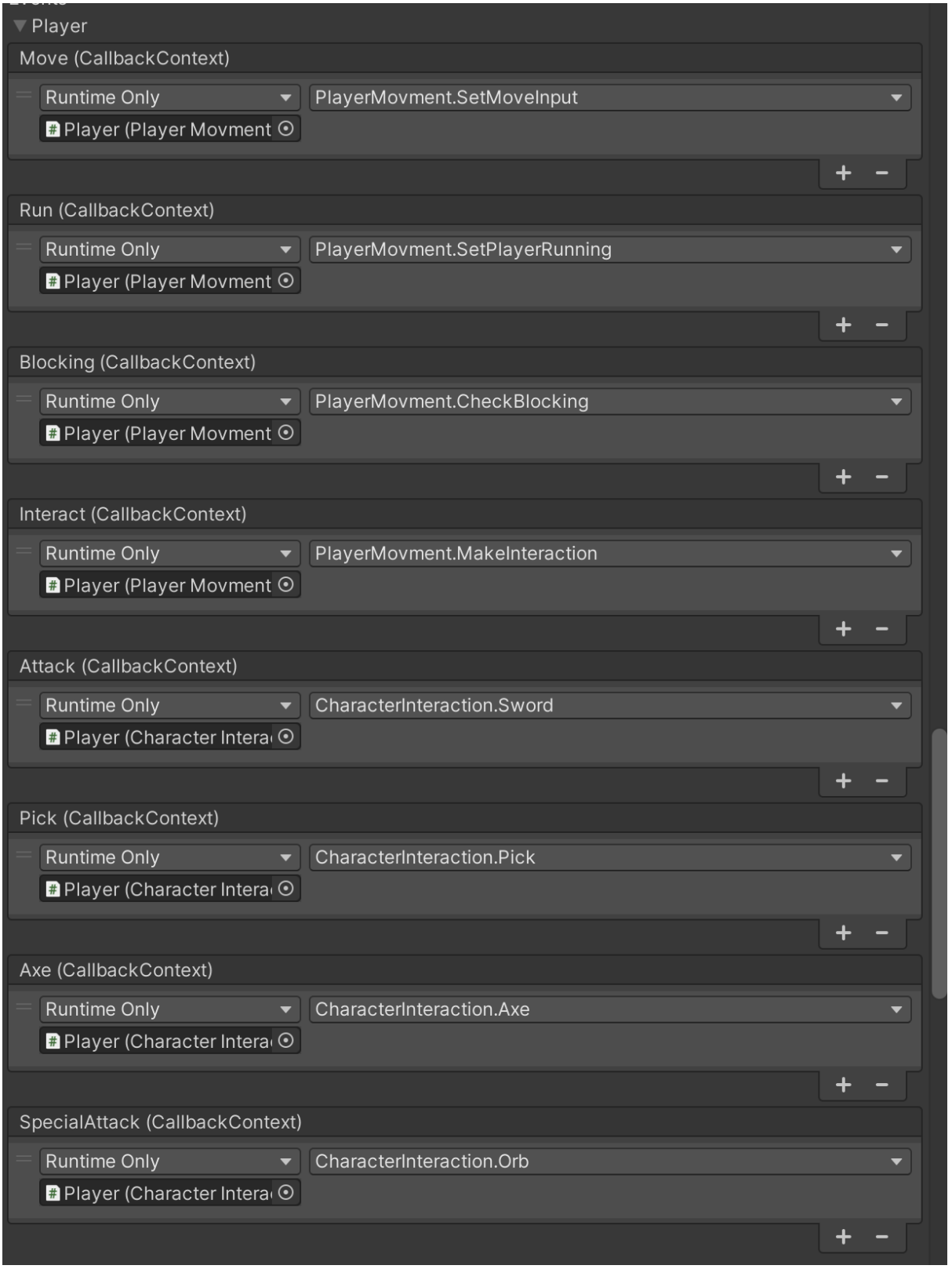


Figura 6.3.5.1: Captura de pantalla de las acciones del mapa Player

- Menú, ver Figura 6.3.5.2:

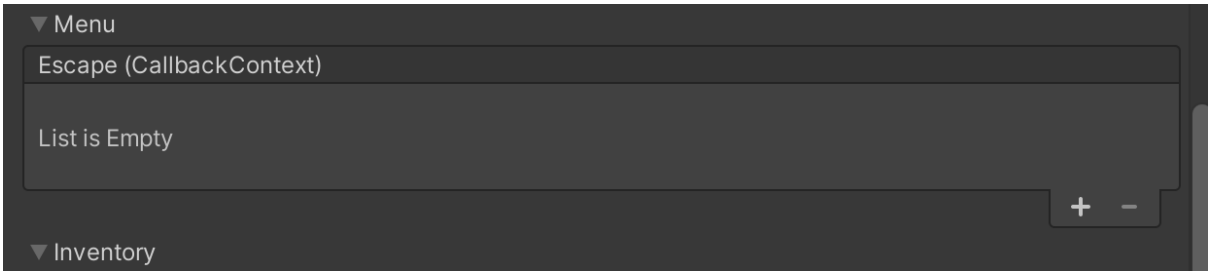


Figura 6.3.5.2: Captura de pantalla de la acciones del mapa Menú

Cabe destacar que en la figura 6.3.5.2 no hay accioens asociadas, ya que para el buen uso de este código tuvimos la necesidad de enlazarlo mediante script, puesto que si no se generaba una dependencia que podría dar fallos en el futuro, o por lo menos resultar un poco complicado de entender.

- Inventario, ver Figura 6.3.5.3:

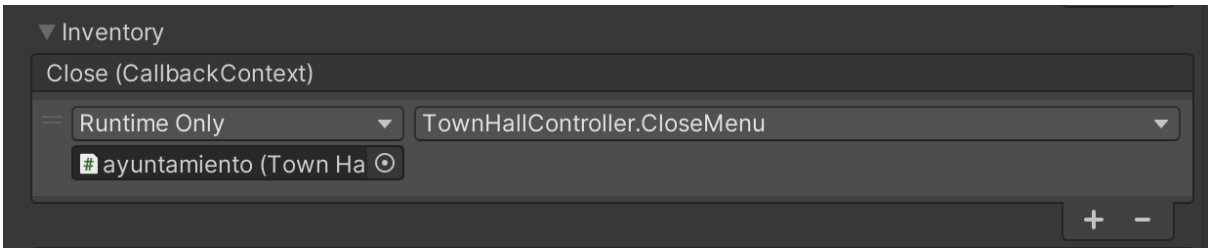


Figura 6.3.5.3: Captura de pantalla de la acciones del mapa Inventario

En las figuras anteriores podemos observar cómo se enlazan las acciones de cada mapa con los scripts correspondientes. Analizaremos que realiza cada acción más adelante cuando entremos en profundidad de la materia, analizando detenidamente el contenido de cada script en su apartado correspondiente.

6.4 Sistema de salud/stamina

Este sistema cuenta con dos partes diferenciadas, las cuales funcionan entre sí, ya que por un lado tenemos la lógica que se encarga de controlar los niveles, la regeneración y el desgaste, mientras que por otro se encuentra toda la representación visual de la misma. Es importante que la lógica que se encarga de gestionar la vida y la estamina sea la misma, por lo que no necesitamos tener código repetido.

6.4.1 Representación visual

Para la representación de estos elementos hemos optado por barras circulares, por lo cual conseguiremos un toque diferenciador frente a nuestros rivales. Ver Figura 6.4.1.1.

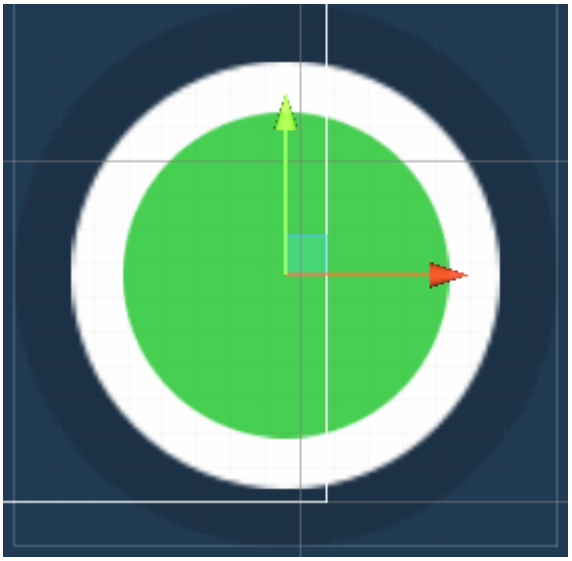


Figura 6.4.1.1: Captura de pantalla del sistema de representación circular

Estas Progress bar está compuesta de 4 elementos:

- Una sombra para resaltar la barra circular del resto de elementos en pantalla.
- La barra de progreso blanca.
- Un fondo de color para el logo (en este caso verde).
- Un logo (en este caso estamos en el prefab genérico, por lo que no cuenta con logo).

Para poder transformar una barra de progreso lineal a circular necesitamos aplicar los siguientes cambios en la imagen que se encarga de contener esta barra de progreso, la cual se puede ver en la figura 6.4.1.2:

Figura 6.4.1.2: Captura de pantalla de la configuración de la barra circular

Cambiamos el tipo de imagen a “Filled” para poder cambiar el método de llenado a “Radial 360”. Además, ponemos el origen de la barra en “Bottom” y establecemos el sentido de llenado en “Clockwise”. Para poder modificar el porcentaje de la barra de progreso lo podemos hacer con el “Fill Amount”.

6.4.2 Scripts de control

A partir de ahora, se comentará el sistema de scripts que se encargan de controlar el buen funcionamiento de estos componentes.

Radial Bar

Este script se encarga de controlar el valor de la barra de progreso e ir actualizando cuando es necesario.

- **UpdateValues()**: este método recibe como parámetro el nuevo valor que debe tomar la barra de progreso. Una vez que posee este valor, comprueba que tipo de velocidad de cambio necesita tomar, ya que si hay una gran diferencia entre el valor anterior y el actual, incrementa la velocidad del cambio. Para poder trabajar cómodamente, normalizamos los valores para trabajar en escalas de 1 a 0, que es la forma nativa de trabajar del sistema de la interfaz. Como último se llama al método "**CheckTypeOfChangeOfValue()**" el cual se encarga de comprobar si es un cambio que necesita incrementar o decrementar el valor.

```
public void UpdateValues(float value)
  {
    changing = true;
    if (Mathf.Abs(currentValue - value) >= 20)  changePerSecond = highChange;
    else changePerSecond = lowChange;

    currentValue = value;
    oldFillValue = fill.fillAmount;
    newFillValue = Normalise();
    CheckTypeOfChangeOfValue();
  }
```

- **ChangeFillValue()**: este método se encarga de actualizar el valor del campo "Fill amount" que podemos observar en la Figura 6.4.4.2 para establecer el nuevo valor. Además, también cambia el color de la barra de color ya que va variando entre rojo y blanco en función del valor anterior.

```
private void ChangeFillValue()
  {
    fill.fillAmount = oldFillValue;
    fill.color = Color.Lerp(lowColor, highColor, oldFillValue);
  }
```

- **CheckCanSleep()**: este método se encarga de comprobar si la diferencia de valor entre actual y máximo es despreciable, y se puede empezar la cuenta atrás para que se esconda la barra de progreso.

```
public bool CheckCanSleep()
  {
    float res = Mathf.Abs(currentValue - maxValue);
    return res < 0.1f;
  }
```

- **Update()**: este método se encarga de incrementar/decrementar gradualmente el valor de la barra de progreso, para ir ajustándose a las necesidades de cada momento. Dentro de cada incremento gradual, se comprueba que este incremento no sea mayor que el valor a conseguir y en caso de que lo sea, se establece directamente el valor objetivo.

```
private void Update()
  {
    if (oldFillValue > newFillValue && isDecreasing)
    {

      if (changePerSecond + Time.deltaTime + oldFillValue < newFillValue)
      {
        oldFillValue = newFillValue;
      }

      oldFillValue -= changePerSecond * Time.deltaTime;
      ChangeFillValue();

    }
    else if(oldFillValue < newFillValue && isIncreasing)
    {
      if (changePerSecond+Time.deltaTime + oldFillValue > newFillValue)
      {
        oldFillValue = newFillValue;
      }
      else
      oldFillValue += changePerSecond * Time.deltaTime;
      ChangeFillValue();
    }
    else
    {
      changing = false;
    }
  }
```

Animator

Como hemos comentado anteriormente, estas barras de salud y estamina no están visibles constantemente, por lo que necesitamos un animator que se encargue de controlar todos los estados en los que puede estar y las condiciones de cada uno.

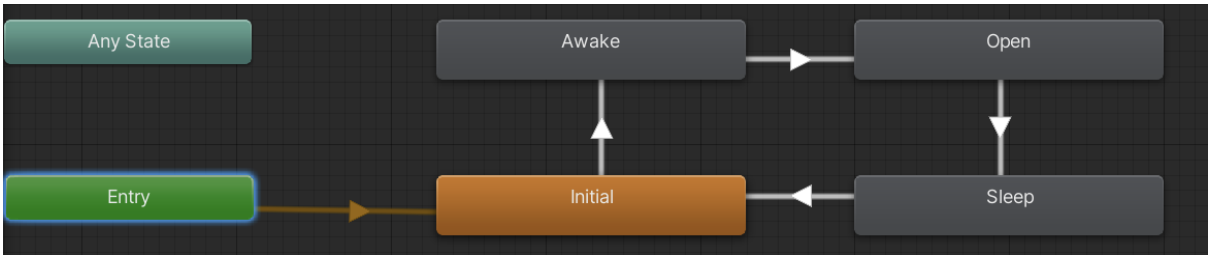


Figura 6.4.4.1: Captura de pantalla de la configuración de la barra circular

Como podemos ver en la Figura 6.4.2.1, contamos con un estado inicial, donde la barra se encuentra invisible (debido a que se hace minúscula) y cuando se activa el trigger "awake", se realiza la transición al estado "Awake", donde se realiza la animación de agrandarse, y una vez acabada la animación, se pasa al estado "Open" donde la barra de progreso es completamente visible. En cuanto se activa el trigger "sleep" se cambia al estado "Sleep", donde se realiza la animación de minimizar la barra de progreso, y cuando acaba pasa al estado "Initial". Ver Figura 6.4.4.2.

Figura 6.4.4.2: Captura de pantalla de los parametros del animator

RadialBar Animator Controller

Este script se encarga de gestionar el estado de las animaciones explicadas anteriormente, ya que cuenta con una lista donde se encuentran las barras de salud y estamina.

- **AwakeRadialBars()**: este método establece un contador a 0, y en caso de que las barras no estén en estado de "Open", llama a la corrutina que se encarga de establecer el trigger "awake" en cada una de las barras de progreso. Para evitar que se pueda dar un solapamiento de llamadas a la corrutina, se para la misma antes de volver a llamarla.

```
public void AwakeRadialBar()
  {

    timer = 0;

    if (!isOpen)
    {
      StopCoroutine(AwakeBars());
      StartCoroutine(AwakeBars());
      isOpen = true;
    }

  }
```

- **SleepRadialBar()**: este método se encarga de llamar a una corrutina que se encarga de establecer el trigger de "sleep" en cada una de las barras de progreso. Al igual que en el método anterior, se comprueba que no exista la corrutina por una llamada anterior, por lo que se para antes de volver a llamarla.

```
public void SleepRadialBar()
  {
    StopCoroutine(SleepBars());
    StartCoroutine(SleepBars());
    isOpen = false;
  }
```

- **CheckCanSleep()**: este método se encarga de recorrer todas las barras de progreso y comprobar si pueden dormir, es decir, desaparecer.

```
private bool CheckCanSleep()
  {
    bool sleep = true;
    foreach (GameObject rb in RadialBars)
    {
      if (!rb.GetComponent<RadialBar>().CheckCanSleep())
      {
        sleep = false;
      }

    }
    return sleep;
  }
```

- **Update()**: Este método se encarga de actualizar el valor del contador, y en caso de que supere el valor mínimo y se pueda hacer desaparecer a las barras, se llama al método “SleepRadialBar()”.

```
private void Update()
  {
    if (!isOpen) return;
    timer += Time.deltaTime;
    if (timer > 2f)
    {
      if (CheckCanSleep())
      {
        SleepRadialBar();
      }

    }
  }
```

RadialBar General Controller

Este método guarda una referencia a la barra de progreso con la que trabaja, y para poder identificar cual es el script que controla la estamina y cual la vida, se puede establecer por inspector el tipo de función que realiza (esta es solo orientativa para no confundirse a la hora de verlo). Los valores que se encarga de mantener y actualizar son el valor actual y el valor máximo.

Cabe destacar que la relación entre estos scripts se hace a través del jugador, porque como es necesario utilizar el “RadialBarGeneralController” para el enemigo, sería añadir lógica extra innecesaria.

- **IncreaseValue()**: este método recibe como parámetro el incremento actual, y comprueba que, al sumar el valor actual con el incremento, no se pase del valor máximo. En caso de que la barra de progreso asociada no sea nula, llama al método encargado de actualizar el valor.

```
public void IncreaseValue(float value)
  {
    currentValue = currentValue + value < maxValue ? (currentValue + value) : maxValue;
    if (radialBar != null) radialBar.UpdateValues(currentValue);
  }
```

- **DecreaseValue()**: este método recibe como parámetro el decremento actual, y comprueba que al sumar el valor actual con el decremento no sea menor que el valor mínimo. En caso de que la barra de progreso asociada no sea nula, llama al método encargado de actualizar el valor.

```
public void DecreaseValue(float value)
  {
    currentValue = currentValue - value > 0 ? (currentValue - value) : 0;
    if (radialBar != null) radialBar.UpdateValues(currentValue);
  }
```

- **CanRecharge()**: este método se encarga de comprobar si aún se están produciendo cambios en la barra, y en caso de que sea así no se puede recargar.

```
public bool CanRecharge()
  {
    if (radialBar != null) { return !radialBar.IsChanging(); }
    else return true;
  }
```

Este método también implementa el sistema de guardado, por lo que cuenta con los siguientes métodos:

- **RedialBarSaveData**: esta estructura se encarga de almacenar dos valores, el currentValue y el maxValue.

```
[System.Serializable]
   struct RadialBarSaveData
   {
      public float currentValue, maxValue;
   }
```

- **CaptureState()**: este método se encarga de crear una nueva instancia de la struct y guardar los valores que se encarga de mantener este código.

```
public object CaptureState()
   {
      RadialBarSaveData data = new RadialBarSaveData();
      data.currentValue = currentValue;
      data.maxValue = maxValue;
      return data;
   }
```

- **RestoreState()**: este método se encarga de restaurar este script a su estado anterior, recuperando la estructura donde se ha almacenando esta información y aplicándola con dos métodos, uno que se encarga de cambiar el valor máximo de él mismo y la barra de progreso, y otro que hace lo mismo, sólo que para el valor actual.

```
public void RestoreState(object state)
   {
      RadialBarSaveData data = (RadialBarSaveData)state;
      //UpdateValue(data.currentValue);
      UpdateMaxValue(data.maxValue);
      SetCurrentValue(data.currentValue);
   }
```

Figura 6.4.4.3: Captura de pantalla del resultado de las barras de progreso

6.5 Sistema de recompensas

El sistema de recompensas se ha desarrollado de tal forma que sea lo más sencillo posible para reutilizar el código y poder asignarlo de una forma eficiente y rápida, tanto a enemigos como a cofres.

6.5.1 Loot

Para mantener todo lo relacionado con un recurso junto hemos optado por utilizar "scriptable objects" junto a la opción de "[CreateAssetmenu]" para poder crear estos objetos como una opción del menú de assets. Este script también permite generar recursos por código, aunque no es el objetivo principal.

```
[CreateAssetMenu]
public class Loot : ScriptableObject
{
    public GameObject lootAsset;
    public Sprite lootSprite;
    public string lootName;
    public int dropChance;

    public Loot(string lootName, int dropChance)
    {
        this.lootName = lootName;
        this.dropChance = dropChance;
    }
}
```

6.5.2 LootBag

Este script se encarga de gestionar que loot tendrá que soltar cada entidad en el momento adecuado y mantener la lista de loot posible.

- **GetDroppedItem()**: este método se encarga de obtener un número aleatorio entre el 1 y el 100, después comprueba de entre todos los objetos de la lista, cuál tiene una probabilidad de aparición mayor al número que ha salido. De entre todas las opciones se elige una al azar aunque, se puede dar el caso de que no salga ninguna opción. El método retorna una de las opciones anteriores.

```
Loot GetDroppedItem()
  {
    int randomNumber = Random.Range(1, 101); // 1-100
    List<Loot> possibleItems = new List<Loot>();
    foreach (Loot item in lootList)
    {
      if (randomNumber <= item.dropChance)
      {
        possibleItems.Add(item);
      }
    }
    if(possibleItems.Count > 0)
    {
      return possibleItems[Random.Range(0, possibleItems.Count)];
    }
    Debug.Log("Not loot dropped");
    return null;
  }
```

- **InstantiateLoot()**: este método recibe como parámetro la posición de spawn y con el método anterior obtiene un recurso de la lista. En caso de que este recurso no sea nulo. Este método es llamado desde el gameobject que quiera hacer el spawn del recurso.

```
public void InstantiateLoot(Vector3 spawnPosition)
  {
    Loot droppedItem = GetDroppedItem();
    if(droppedItem != null)
    {
      GameObject asset = Instantiate(droppedItem.lootAsset,
spawnPosition, Quaternion.identity);
    }
  }
```

6.5.3 Pick Object

Este script tiene una función que sirve para controlar que cuando el jugador se acerque al recurso, este se añada correctamente al inventario del ayuntamiento, se reproduzca el sonido correspondiente y se destruya.

```
public void ObjectTriggered()
    {
        soundPlayer.PlayOneShot(grabSound);
        inventory.AddResource(resource_name);
        mesh.SetActive(false);
        Destroy(gameObject, 2f);
    }
```

6.5.4 Player Triggered Object

Este script sirve para detectar cuando el jugador entra dentro del trigger del recurso y llama a la función explicada en el punto anterior.

```
private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            gameObject.GetComponentInParent<PickObject>().ObjectTriggered();
        }
    }
```

6.5.5 Humo estilizado: efecto visual para la aparición de recursos

Con el objetivo de generar un efecto visual cuando aparecen los recursos, se usó la herramienta integrada en Unity llamada “Visual Effects Graph”, la cual sirve para crear efectos de partículas mediante el uso de un lenguaje más visual mediante nodos y arcos de unión. A continuación, se muestra cada uno de los bloques implementados para conseguir el efecto deseado. Ver Figura 6.5.5.1.

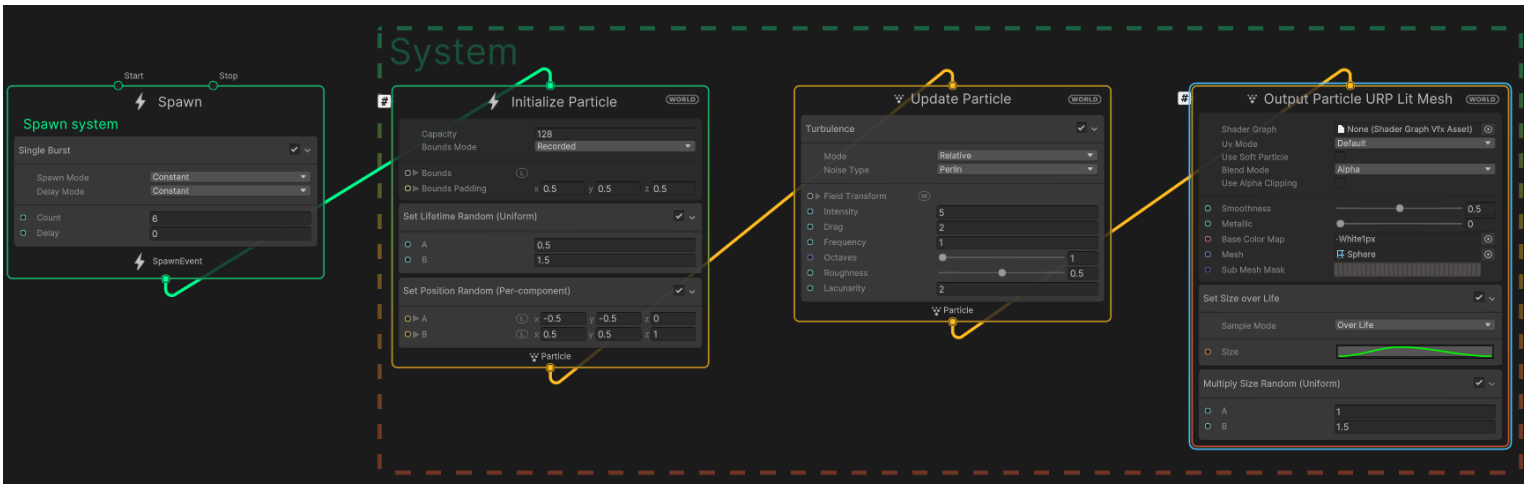


Figura 6.5.5.1: Captura de pantalla del "graph" entero del efecto del humo para la aparición de los recursos

El primer bloque consiste en la creación de las partículas. En concreto, se define que la creación sea constante y el número total de partículas presentes. Ver Figura 6.5.5.2.

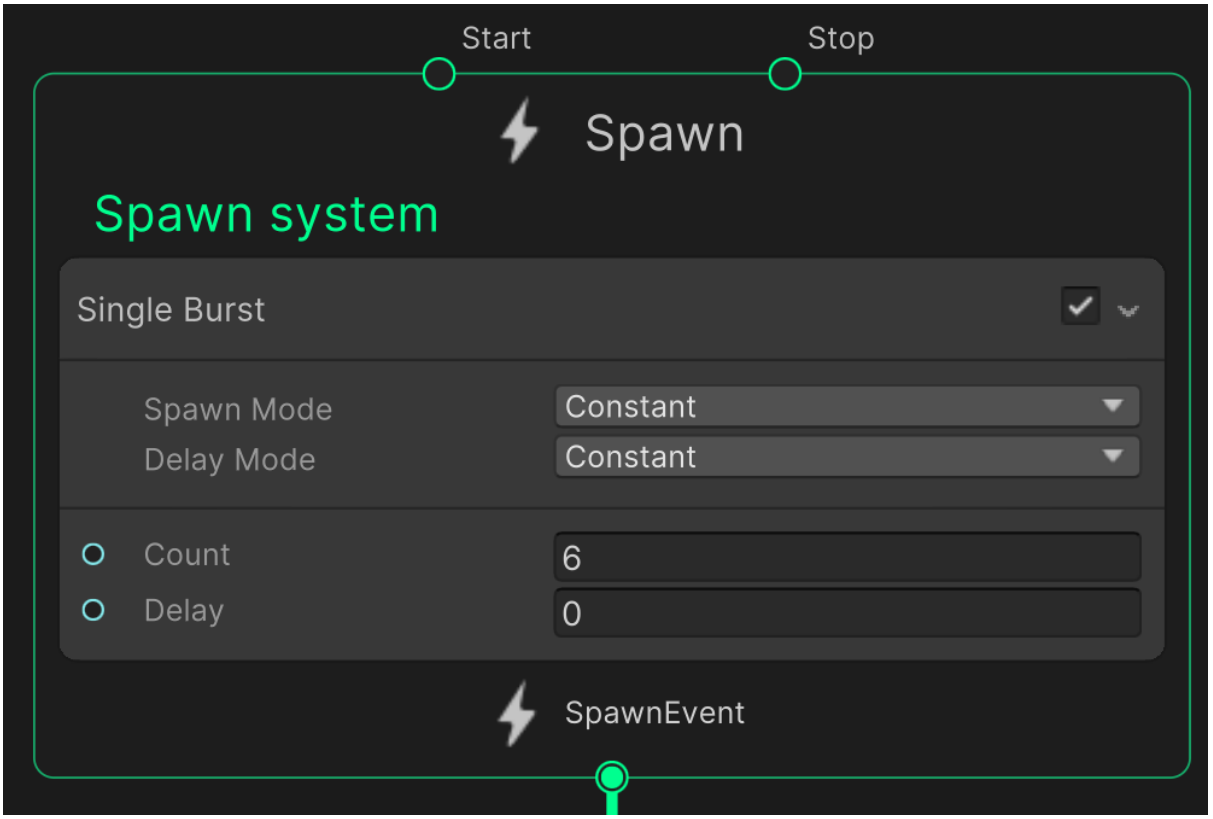


Figura 6.5.5.2: Captura de pantalla del primer bloque del efecto visual del humo de los recursos

El siguiente bloque define la máxima capacidad de partículas del sistema, además de definir la vida de cada una de las partículas mediante el nodo "Set Lifetime Random", el cual establece dos valores que representan el tiempo de vida mínimo y el máximo. En este caso, se optó por una duración por partícula entre 0.5 y 1.5 segundos. Por último, se usa el nodo "Set Position Random" con la finalidad de instanciar dichas partículas en

posiciones aleatorias dentro de una área o “bounding-box” definida por dos puntos (A y B). Nótese que para cada punto se muestra una “L” referente a que la posición debe ser en local para que el área sea alrededor del objeto correspondiente. Ver Figura 6.5.5.3.

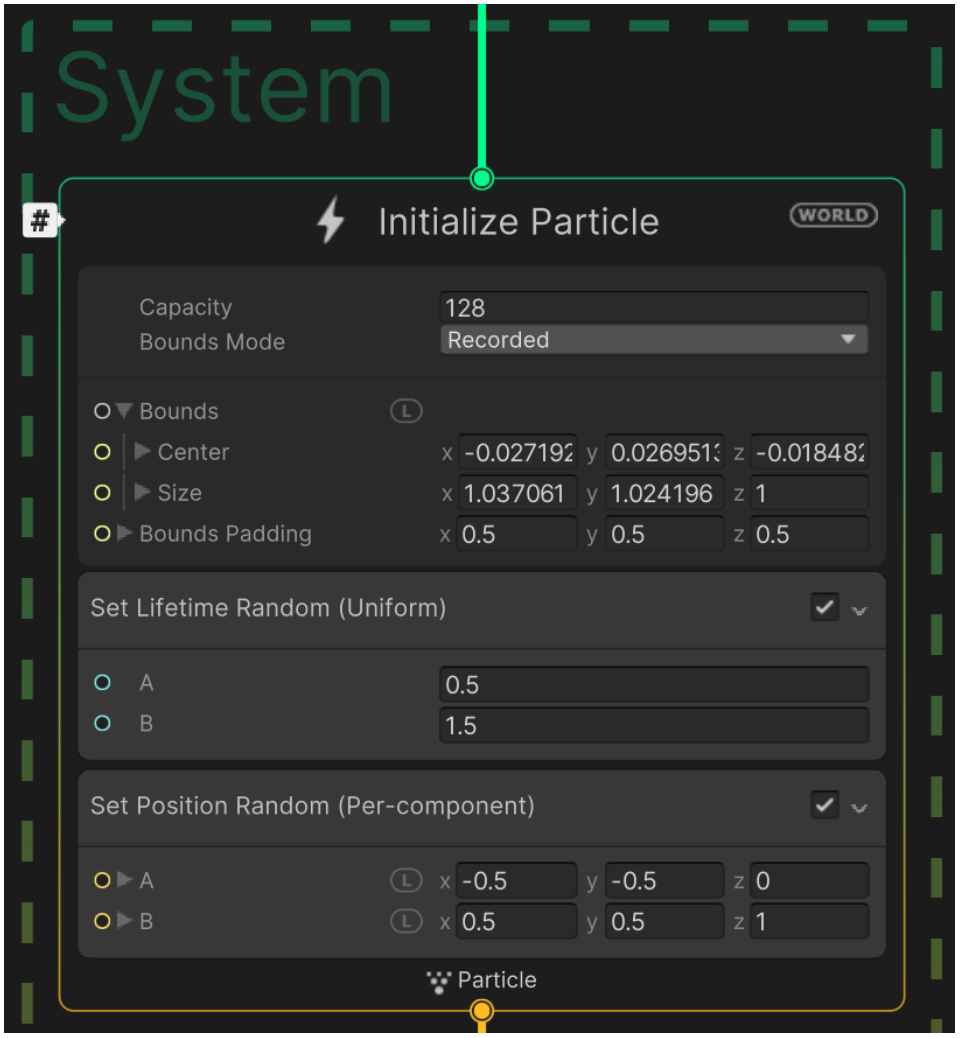


Figura 6.5.5.3: Captura de pantalla del segundo bloque del efecto visual del humo de los recursos

Seguidamente, para añadir cierto movimiento a las partículas, se añade el nodo “Turbulence” el cual genera un “campo de ruido” el cual se aplica a la velocidad de las partículas con el fin de obtener un efecto más natural. Después de ciertas comprobaciones, se usa el ruido de Perlin con una intensidad de 5 y un coeficiente de rozamiento a 2. Ver Figura 6.5.5.4.

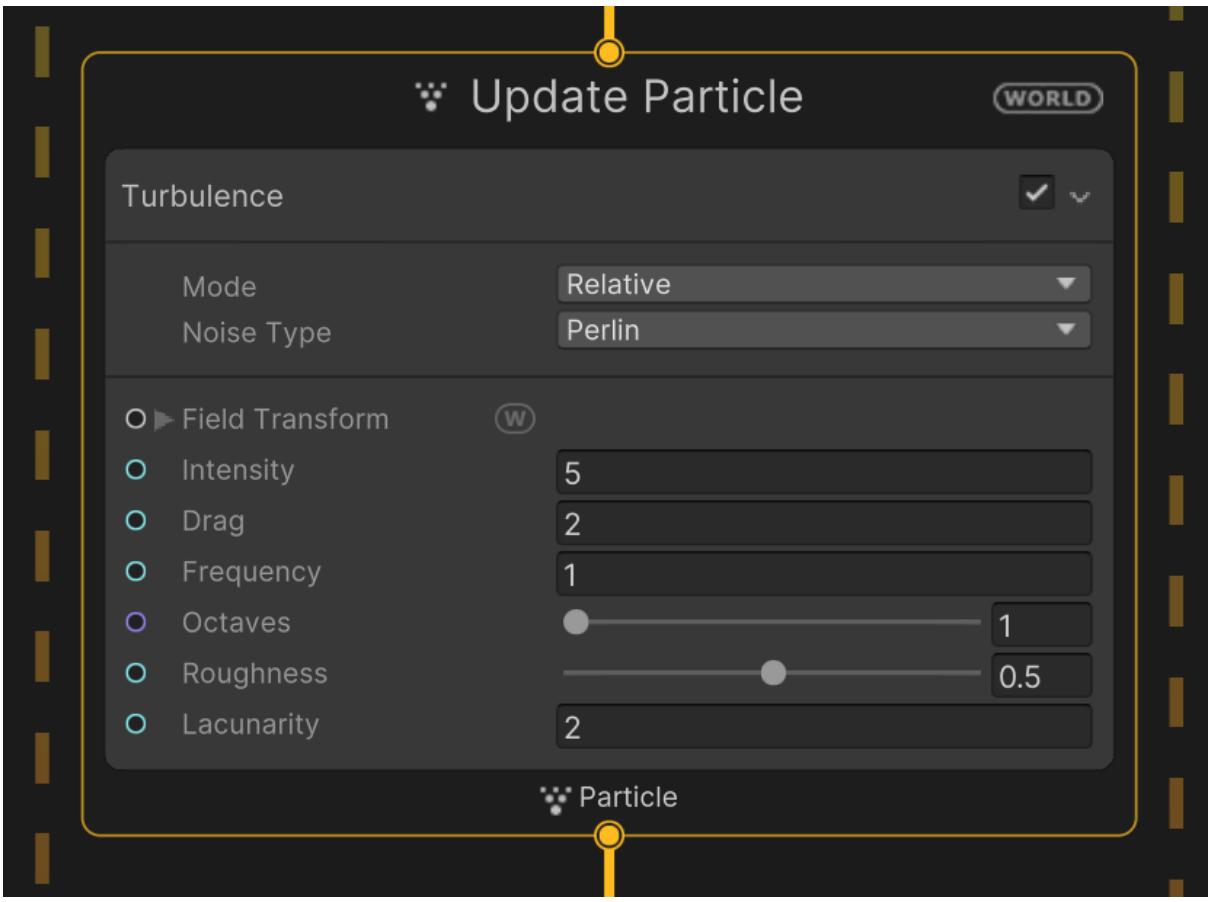


Figura 6.5.5.4: Captura de pantalla del tercer bloque del efecto visual del humo de los recursos

Por último, el bloque que genera la malla de las partículas considera un tipo de geometría esférica. Acto seguido, se añade el nodo “Set Size over Life” el cual define una curva para establecer el tamaño a lo largo de la vida de cada partícula, la cual va creciendo rápidamente y luego decrece más despacio. Finalmente, para tener tamaños distintos, se añade el nodo “Multiply Size Random”, el cual multiplica el tamaño de las partículas por un valor aleatorio entre 1 y 1.5. Ver Figura 6.5.5.5.

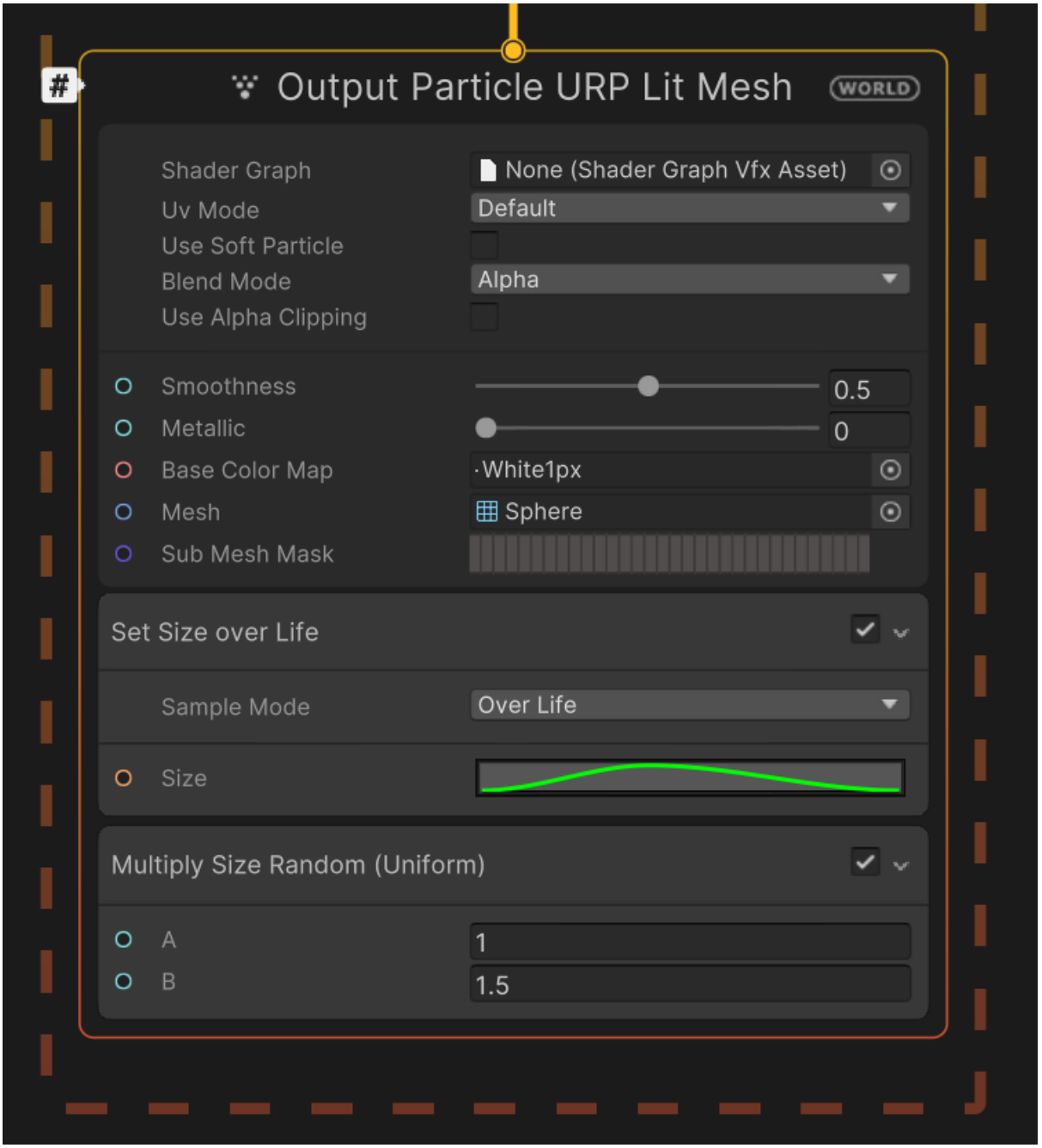


Figura 6.5.5.5: Captura de pantalla del cuarto bloque del efecto visual del humo de los recursos

6.5.6 Resumen

Esta estructura de scripts se ha diseñado de esta forma para permitir la opción de deshabilitar la mesh del recurso, y que se pudiera seguir reproduciendo el sonido de haberlo recogido, ya que este sonido es espacial.

6.6 Orientación de los textos

Un apartado importante es orientar correctamente los textos a través de los cuales el jugador entenderá como interactuar con los elementos del escenario. El script que se encarga de realizar esto se llama “LookAtCamera”. Además, este script se encarga de

mantener los textos adecuados, tanto para teclado como para mando, que se realiza a través de los métodos “SetGAmepadText() “y “SetKeyboardText()”, los cuales se llaman desde el script “InputMode”, explicado con anterioridad.

- **Start()**: este método se encarga de establecer por defecto el texto de teclado, y de mantener la posición de la cámara adecuada, ya que en caso de que el gameobject se encuentre en la dungeon, nos interesa la cámara vertical.

```
private void Start()
{
   SetKeyboardText();

   foreach(Camera item in FindObjectsOfType<Camera>(true))
   {
      if (isInDungeon && item.CompareTag("DungeonCamera"))
      {
         cameraPosition = item.transform;
      }
      else if(!isInDungeon && item.CompareTag("MainCamera"))
      {
         cameraPosition = item.transform;
      }
   }
}
```

- **Update()**: este método actualiza la orientación del texto mediante el método LookAt de Unity y la información de la cámara.

```
void Update()
{

    textObject.transform.LookAt(textObject.transform.position + cameraPosition.rotation *
(-Vector3.back), cameraPosition.transform.rotation * (-Vector3.down));

}
```

6.7 Sistema de respawn

Para asegurarnos de que el jugador no se quede sin recursos ni contenido que hacer durante su estancia en el juego, existe este escript que se encarga de controlar que

objeto es necesario instanciar y controlar los tiempos del mismo. Además, se guarda una referencia a los componentes que hay que instanciar, junto a los padres que deben de controlar su posición en la escena.

- **TimeSpawner()**: esta función recibe como parámetros toda la información necesaria para poder instanciar cada gameobject con sus propias características. En función de qué objeto sea necesario instanciar, se ejecutará una parte del switch u otra.

```
private IEnumerator TimerSpawn(Vector3 position, Quaternion rotation, Vector3 scale,SpawnObject
obj, float time , GameObject objDestroy, List<Loot> loot, PatrolPath patrulla)
{
    yield return new WaitForSeconds(time);
    GameObject aux;
    switch (obj)
    {
        case SpawnObject.Tree:
            aux = Instantiate(tree, position, rotation, bosque.transform);
            Destroy(objDestroy);
            aux.transform.localScale = scale;
            break;

        case SpawnObject.Rock:
            aux = Instantiate(rock, position, rotation, cantera.transform);
            aux.transform.localScale = scale;
            break;

        case SpawnObject.Chest:
            aux = Instantiate(chest, position, rotation, cofresDungeon.transform);
            aux.transform.localScale = scale;
            aux.GetComponent<LootBag>().SetLootList(loot);
            break;
        case SpawnObject.Enemy:
            aux = Instantiate(enemy, position, rotation, enemigos.transform);
            aux.GetComponent<AIController>().SetPatrol(patrulla);
            break;
    }

    yield return null;
}
```

- **HasToRespawnObject()**: este método recibe como parámetro toda la información necesaria para realizar el respawn, y llama a la corrutina que se encarga de ello.

```
public void HasToRespawnObject(Transform transform, SpawnObject obj, float time = 5f, GameObject objDestroy = null, List<Loot> loot = null, PatrolPath patrulla = null)
{
    Vector3 position = transform.position, scale = transform.localScale;
    Quaternion rotation = transform.rotation;
    StartCoroutine(TimerSpawn(position, rotation, scale,obj, time, objDestroy, loot, patrulla));
}
```

6.8 Sistema de interacción

Para realizar de una manera sencilla todo el sistema de interacción del jugador con los diferentes elementos del juego, se ha desarrollado un script que permite implementar a cada objeto su propia forma de interacción mediante un método común.

6.8.1 IInteractor

Esta interfaz consta de un método llamado DoInteraction(), el cual es necesario implementar a la hora de heredar de esta clase, donde la función de éste es que, cuando el jugador intente realizar una interacción con un elemento del juego, todos tengan el mismo método y no sea necesario aprenderse el método de cada uno, lo cual simplifica el código.

```
public interface IInteractor
{
    void DoInteraction();

}
```

6.9 Sistema de cofres

Una de las formas a través de las cuales el jugador podrá conseguir recompensas es mediante el sistema de cofres, los cuales le darán un loot aleatorio dentro de las posibilidades del mismo.

El prefab del cofre está formado por el esqueleto (necesario para la animación), el gameObject del cofre y un Text, para mostrar la información necesaria.

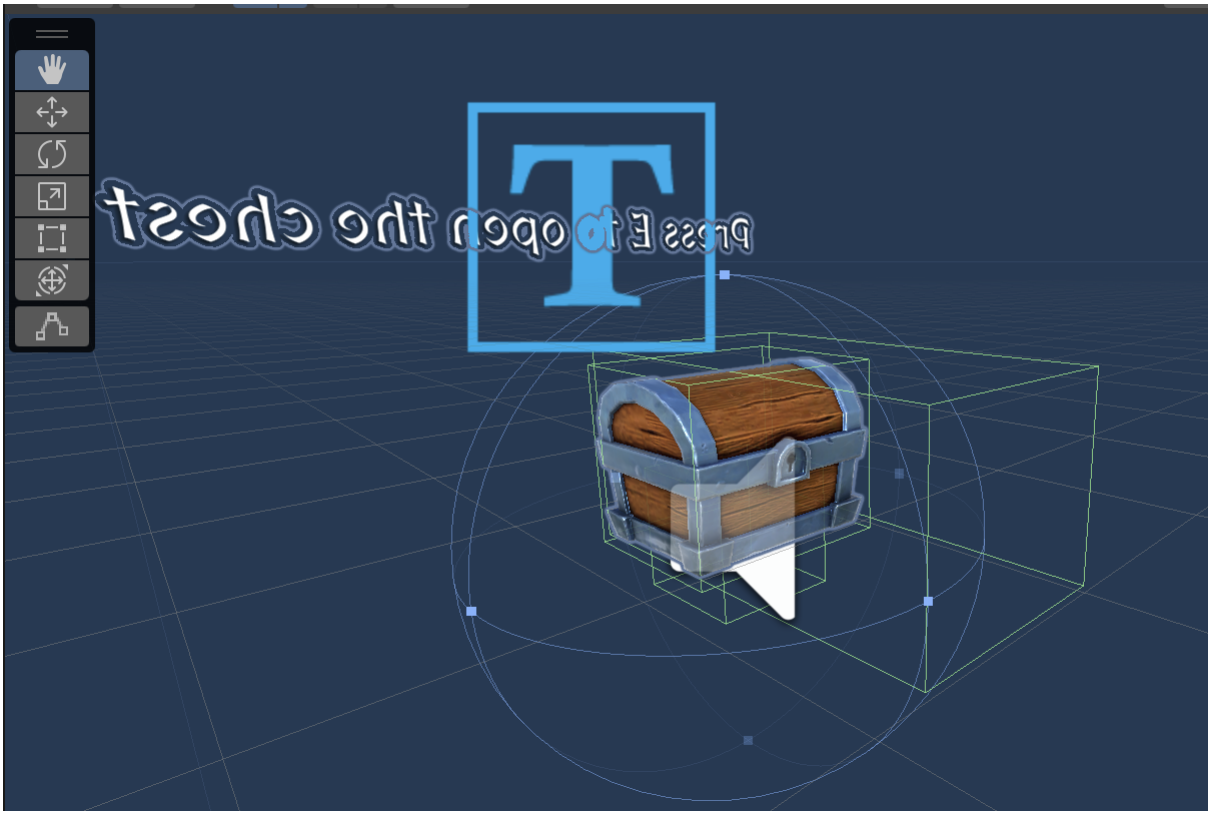


Figura 6.9.1: Captura de pantalla del prefab del cofre

6.9.1 Implementación del prefab

Este gameobject cuenta con dos box colliders, donde el más pequeño sirve para que el jugador colisione contra el cofre, y el más grande realiza la función de trigger, para saber cuando se acerca el jugador. También cuenta con animator para realizar las animaciones que hemos desarrollado para él. Otros componentes con los que cuenta son un texto para guiar al jugador con las interacciones y un audio Source.

Para el buen funcionamiento, utiliza los siguientes componentes:

- Look At Camera: script explicado con anterioridad para poder girar el texto a donde corresponda.
- NavMesh Obstacle, para que el enemigo lo detecte como un obstáculo y pueda moverse adecuadamente.
- LootBag: script que permite al cofre instanciar determinados recursos cuando el jugador interactúa con él.

6.9.2 Animator

Este componente con el que cuenta, se encarga de controlar todos los estados de la animación, teniendo la estructura de la Figura 6.9.2.1.

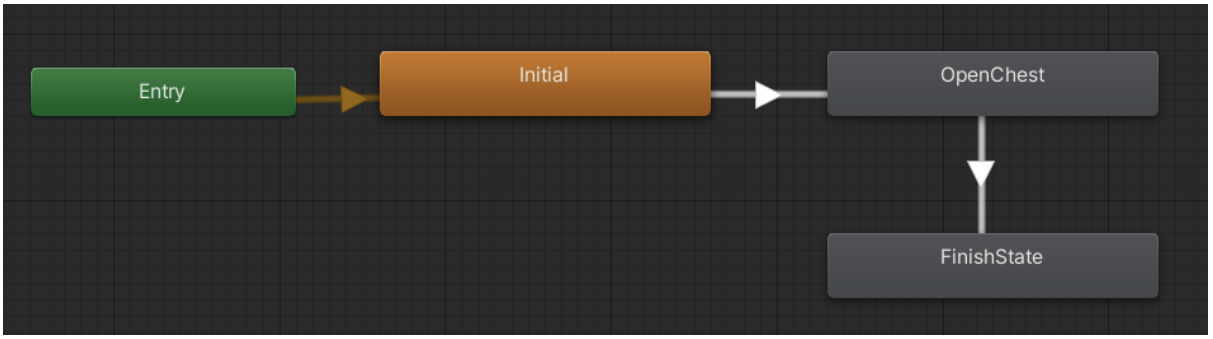


Figura 6.9.2.1: Captura de pantalla del sistema de animaciones

Cuenta con un trigger el cual se activa cuando tiene que realizar la animación de OpenChest. Una vez que se ha acabado la animación, se pasa directamente al estado FinishState.

6.9.3 Chest Controller

Este script se encarga de controlar todo el funcionamiento del cofre.

- **OnTriggerEnter()**: este método se encarga de detectar quién ha entrado dentro del trigger, y en caso de que fuera el jugador, se establece como objeto interactuable dentro del script del jugador y activa el cuadro de texto de información.

```
private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            playerMov = other.gameObject.GetComponent<PlayerMovment>();
            playerMov.SetInteractableObject(this);
            textInfo.gameObject.SetActive(true);
        }
    }
```

- **OnTriggerExit()**: este método se encarga de detectar quién ha salido del trigger, y en caso de que fuera el jugador, se borra la opción del interactuable dentro del script del jugador, y se desactiva el cuadro informativo del texto.

```
private void OnTriggerExit(Collider other)
{
    if (other.gameObject.CompareTag("Player"))
    {
        playerMov.RemoveInteractableObject();
        playerMov = null;
        textInfo.gameObject.SetActive(false);
    }
}
```

- **DoInteraction()**: este método pertenece a la interfaz que implementa este script para permitir al jugador realizar la interacción, y que se encarga de comprobar si el cofre está abierto; en caso de que no sea así se realiza la animación de abrir, y se quita de los objetos interactuables.

```
public void DoInteraction()
{
    if (!isOpen)
    {
        isOpen = true;
        chestAnimator.SetBool("Open", isOpen);
    }
    playerMov.RemoveInteractableObject();

}
```

- **DestroyChest():** esta corrutina se encarga de esperar tres segundos para llamar al script que se encarga de hacer el respawn del objeto, pasándole como parámetro su transform, el tipo de objeto que es y la lista de loot que posee. Como último paso se destruye a sí mismo.

```
private IEnumerator DestroyChest()
{
    yield return new WaitForSeconds(3.0f);
    respawn.HasToRespawnObject(transform, RespawnObjects.SpawnObject.Chest,
60f ,null, GetComponent<LootBag>().GetLootList());
    Destroy(gameObject, 0);

    yield return null;
}
```

- **CheckEndAnimation()**: este método se encarga de comprobar si se ha acabado la animación, es decir, está en el "Finish state", y en caso de ser así reproduce el sonido de abrirse, instancia el loot correspondiente, desactiva colisiones y la mesh, y por último, llama a la corrutina encargada de la destrucción. Este método se comprueba en el Update en caso de que se haya abierto el cofre.

```
private void CheckEndAnimation()
{
    if (chestAnimator.GetCurrentAnimatorStateInfo(0).IsName("FinishState"))
    {
        isOpen = false;
        audioPlayer.PlayOneShot(clip);
        GetComponent<LootBag>().InstantiateLoot(new Vector3(transform.position.x,
transform.position.y+1, transform.position.z));
        bCollider.enabled = false;
        mesh.SetActive(false);
        StartCoroutine(DestroyChest());
    }
}
```

6.10 Sistema de inventario

Este apartado es importante para el buen funcionamiento del juego, ya que cuando el jugador recoge un objeto es importante poder clasificarlo y guardarlo en su inventario, para que este lo pueda utilizar más adelante en mejoras de las estructuras o incrementar su salud.

6.10.1 Inventory System

Este script se encarga de controlar la cantidad de recursos que tiene el jugador e implementar todas las funciones necesarias para poder trabajar con ellos. Además, esta clase se extiende de la clase Subject, que forma parte del sistema de Observer e implementa la interfaz de guardado.

- **CustomNotify()**: la clase Subject tiene un método público llamado "CustomNotify", el cual se encarga de sobrescribir para poder adaptarlo a sus necesidades. Este método recibe como parámetro el nombre del recurso y la cantidad. por cada observer que se encuentre en la lista de observers, envía un mensaje con esta información.

```
public override void CustomNotify(string name, int count)
{
    _observers.ForEach((_observers) =>
    {
        _observers.OnNotify(name, count);
    });
}
```

- **AddResource()**: este método se encarga de recibir como parámetro el nombre del recurso y clasificarlo para añadirlo al inventario. Además, se encarga de llamar al método que se encarga de notificar a los observers el cambio en los recursos.

```
public void AddResource(string resource_name)
{

    if (resource_name == "diamond")
    {
        diamond++;
        CustomNotify(resource_name, diamond);
    }
    else if (resource_name == "gold")
    {
        gold++;
        CustomNotify(resource_name, gold);
    }
    else if (resource_name == "pepper")
    {
        pepper++;
        CustomNotify(resource_name, pepper);
    }
    else if (resource_name == "silver")
    {
        silver++;
        CustomNotify(resource_name, silver);
    }
    else if (resource_name == "tomatoe")
    {
        tomatoe++;
        CustomNotify(resource_name, tomatoe);
    }
    else if (resource_name == "vanilla")
    {
        vanilla++;
        CustomNotify(resource_name, vanilla);
    }
    else if (resource_name == "wood")
    {
        wood++;
        CustomNotify(resource_name, wood);
    }
    else if(resource_name == "stone")
    {
        stone++;
        CustomNotify(resource_name, stone);
    }
}
```

Para poder conservar toda la información que tiene el inventario para partidas futuras, necesitamos implementar el sistema de guardado.

- **InventorySaveData()**: esta struct se encarga de mantener tantas entidades como recursos hay.

```
[System.Serializable]
  struct InventorySaveData
  {
     public int diamond;
     public int gold;
     public int pepper;
     public int silver;
     public int tomatoe;
     public int vanilla;
     public int wood;
     public int stone;
  }
```

- **CaptureState()**: este método se encarga de crear una nueva instancia de la struct explicada antes, guardar en ella toda la información del inventario y retornarla con esta información para que se pueda guardar.

```
  public object CaptureState()
  {
     InventorySaveData data = new InventorySaveData();

     data.diamond = this.diamond;
     data.gold = this.gold;
     data.pepper = this.pepper;
     data.silver = this.silver;
     data.tomatoe = this.tomatoe;
     data.vanilla = this.vanilla;
     data.wood = this.wood;
     data.stone = this.stone;

     return data;
  }
```

- **RestoreState()**: este método recibe como parámetro un objeto que contiene toda esta información, y le aplica un cast a la struct declarada para poder manipular la información. Una vez que ya es accesible, carga toda la información de esta en las entidades propias del script.

```
public void RestoreState(object state)
{
    InventorySaveData data = (InventorySaveData)state;
    diamond = data.diamond;
    gold = data.gold;
    pepper = data.pepper;
    silver = data.silver;
    tomatoe = data.tomatoe;
    vanilla = data.vanilla;
    wood = data.wood;
    stone = data.stone;
}
```

6.10.2 General Inventory HUD Controller

Este script se encarga de controlar la información que se muestra en la pantalla, actualizándose siempre que sea necesario.

- **LoadInformation()**: este método se encarga de aplicar en los campos de texto toda la información correspondiente a cada uno de los recursos.

6.11 Ayuntamiento

Este prefab es donde se encuentra todo el sistema de inventario del jugador y toda la lógica que se encarga de gestionarlo. El modelo cuenta con dos puntos de acceso, a través de los cuales el jugador puede interactuar con la estructura y acceder al inventario. Además, cada punto de acceso tiene un sistema de guía de interacción para guiar al jugador, como se puede ver en la Figura 6.11.

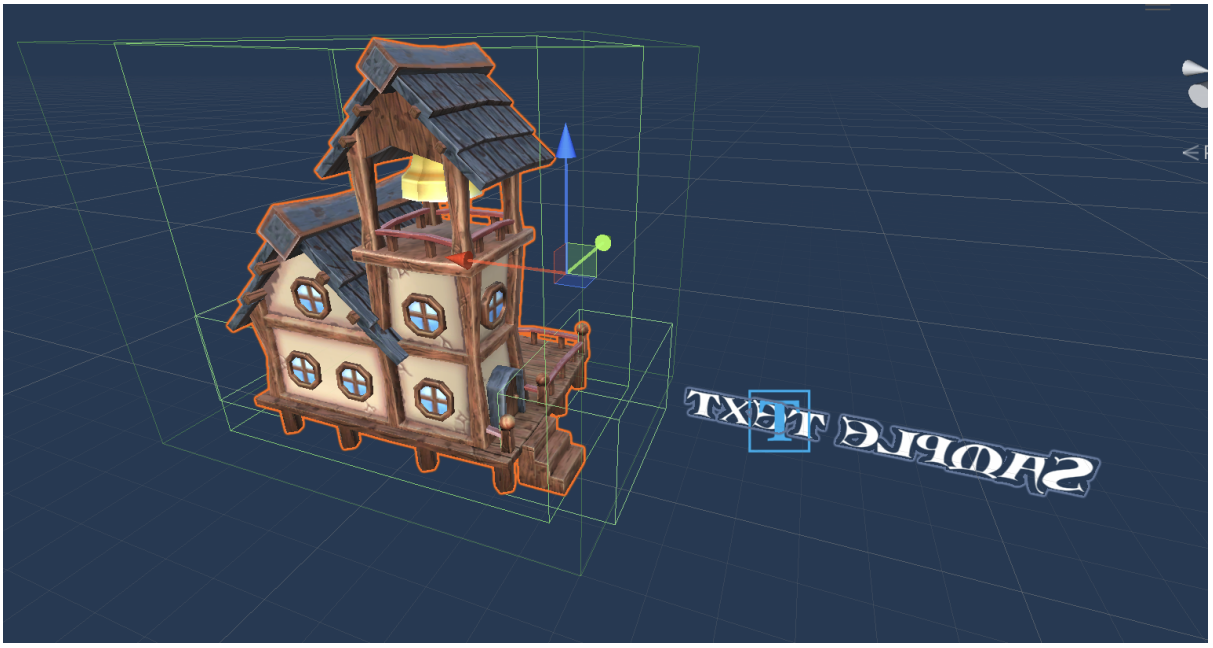


Figura 6.11: Captura de pantalla del prefab del ayuntamiento

Este gameobject cuenta con los siguientes componentes:

- NavMeshObstacle(): permite que los npc detecten su ubicación y tracen trayectorias eficientes.
- SaveableEntity(): permite establecer un número identificativo para que se pueda guardar la información necesaria.
- InventorySystem(): sistema encargado de almacenar la información del inventario.

6.11.1 Interact Trigger Ayuntamiento

Este script se encarga de detectar cuando entra el jugador en el trigger del ayuntamiento y establecer los datos necesarios.

- **OnTriggerEnter():** este método detecta si el que ha entrado es el jugador y si es así, establece a visible el texto informativo y llama al método que avisa al script que controla el ayuntamiento de que el jugador está dentro, y le pasa una referencia del jugador.

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        text.SetActive(true);
        controller.PlayerInside(other.gameObject);
    }
}
```

- **OnTriggerExit()**: este método detecta cuando sale un gameobject, y en caso de ser el jugador, desactiva el texto informativo y llama al método que se encarga de avisar al script de control del ayuntamiento de que el borre al jugador.

```
private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player"))
    {
        text.SetActive(false) ;
        controller.RemovePlayer();
    }
}
```

6.11.2 Town Hall Controller

Este script se encarga de gestionar la información del inventario y el acceso a él.

Los primeros métodos de los que vamos a hablar son los que se encargan de gestionar cuando el jugador está dentro o fuera del entorno de interacción.

- **PlayerInside()**: este método recibe como parámetro el jugador, y establece como objeto interactuable a sí mismo y se guarda una referencia del jugador.

```
public void PlayerInside(GameObject caballero)
{
    caballero.GetComponent<PlayerMovment>().SetInteractableObject(this);
    player = caballero;
}
```

- **RemovePlayer()**: este método se borra de objeto interactuable dentro del script del jugador, y elimina la referencia que contiene.

```
public void RemovePlayer()
{
    player.GetComponent<PlayerMovment>().RemoveInteractableObject();
    player = null;
}
```

Para poder realizar todo el sistema de interacciones, este script implementa la interfaz IInteractor, implementado el método DoInteraction.

- **DoInteraction**(): en caso de que el jugador no sea nulo, se llama al método ShowMenu(), el cual explicaremos más abajo.

```
public void DoInteraction()
{
    if (player)
    {
        ShowMenu();
    }
}
```

Para poder controlar el funcionamiento del HUD del menú, contamos con los siguientes métodos:

- **ShowMenu()**: este método se encarga de preparar todo lo necesario para mostrar el menú. Cambia el esquema de control al del inventario, desactiva el recordatorio de misión, hace visible el cursor, llama al método del inventario que regresa toda la información que contiene para pasársela a la función del hud, que se encarga de cargarla y muestra el HUD del inventario.

```
private void ShowMenu()
{
    inventoryOpen = true;

player.GetComponent<PlayerMovment>().ChangeInputMode(PlayerMovment.Control
Schemne.Inventory);
    actualMissionReminder.SetActive(false);
    Time.timeScale = 0f;
    Cursor.visible = true;
    controller.LoadInformation(inventory.GetInventoryInformation());
    HUDInfo.SetActive(true);
}
```

- **CodeToCloseInventory()**: este método realiza todo lo contrario al anterior, ya que se encarga de gestionar que el juego regrese a la normalidad. Lo más destacable es que establece el sistema de control del jugador.

```
private void CodeToCloseInventory()
{
    inventoryOpen = false;
    Cursor.visible = false;
    actualMissionReminder.SetActive(true);
    HUDInfo.SetActive(false);
    Time.timeScale = 1f;

player.GetComponent<PlayerMovment>().ChangeInputMode(PlayerMovment.Control
Schemne.Player);
}
```

- Los métodos **CloseMenu**() y **CloseByClicking**() llaman al método anterior, sólo que uno es para cuando el jugador interactúa por teclado y otro por el botón.

```
public void CloseMenu(CallbackContext ctx)
{
    if (ctx.started)
    {
        CodeToCloseInventory();
    }

}

public void CloseByClicking()
{
    CodeToCloseInventory();
}
```

6.12 Sistema de cámaras

Para nuestro juego contamos con dos sistemas distintos de cámaras, ya que una cámara la utilizamos para la vista en general del juego, y la otra para cuando el jugador entra en las mazmorras. El control de este sistema de cámaras se lleva a cabo mediante los "Region collider", que lo explicaremos en el siguiente apartado, mientras que en este nos centraremos exclusivamente en el código que gestiona el movimiento de las mismas.

6.12.1 Camera Follow

Este script cuenta con dos métodos:

- **Awake()**: se encarga de calcular el offset a partir de las posiciones iniciales del jugador y la cámara.

```
private void Awake()
{
    offset = transform.position - target.position;
}
```

- **LateUpdate()**: este método calcula la posición futura a través de la posición objetivo más la suma del offset. Aplica esta posición a la cámara a través de un SmoothDamp.

```
private void LateUpdate()
{
    Vector3 targetPosition = target.position + offset;
    transform.position = Vector3.SmoothDamp(transform.position, targetPosition,
ref currentVelocity, smoothTime);
}
```

6.13 Sistema de observer

Para poder relacionar de una forma correcta sin generar dependencias entre el sistema del inventario y el sistema de quest, hemos desarrollado este script que se encarga de gestionar los mensajes entre ambos de una manera sofisticada y elegante.

6.13.1 Subject

Este script mantiene una lista de las interfaces a las cuales es necesaria enviar información. Para gestionar esta lista, tenemos a los siguientes métodos:

- **AddObserver**(): recibe por parámetro la referencia al observer y la añade a la lista.

```
public void AddObserver(IObserver observer)
{
    _observers.Add(observer);
}
```

- **RemoveObserver**(): este método recibe por parámetro la interfaz de observer que es necesario eliminar y la elimina de la lista.

```
public void RemoveObserver(IObserver observer)
{
    _observers.Remove(observer);
}
```

Además, este método cuenta con un CustomNotify, el cual se encarga de implementar la clase que herede de esta, y lo usará para notificar los cambios cuando sea necesario.

```
public abstract void CustomNotify(string name, int count);
```

6.13.2 IObserver

Este script es una interfaz la cual tienen que implementar el script que quiera utilizar este sistema de notificación, debe implementar el siguiente método:

- **OnNOtify()**: este método recibe como parámetro el nombre del recurso y la cantidad.

```
public interface IObserver
{
    public void OnNotify(string name, int count);
}
```

En caso de querer reutilizar este sistema de observer, es necesario crear otros métodos para adaptarse a futuras necesidades.

6.14 Fuente de la vida

Este gameObject se encarga de regenerar la vida del jugador, siempre y cuando este se encuentre al lado de ella y realice la interacción. Para el buen funcionamiento, nos encontramos con los siguientes componentes:

- Un collider para detectar las colisiones con el jugador.
- Un collider con función de trigger para detectar cuando entra el jugador.
- LookAtCamera, para controlar el funcionamiento del texto explicativo de la interacción.
- Un componente de texto para explicar cómo se realiza la interacción.
- NavMeshObstacle: este componente permite a las IA detectar que es una zona no navegable.

Toda esta información la podemos ver en la Figura 6.14.

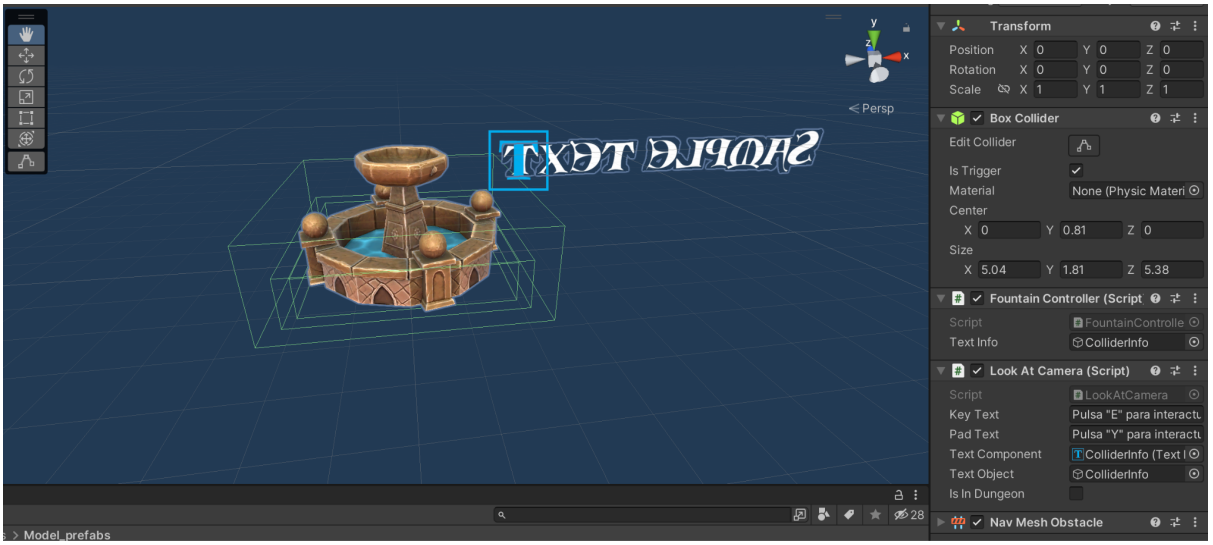


Figura 6.14: Captura de pantalla del prefab de la fuente

6.14.1 Fountain Controller

- **OnTriggerEnter()**: este método detecta que objeto entra en el trigger, y, en caso de ser el jugador, activa el texto explicativo, se establece como objeto interactivo dentro del script del jugador y se guarda una referencia.

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        textInfo.SetActive(true);

other.gameObject.GetComponent<PlayerMovment>().SetInteractableObject(this);
        player = other.gameObject;
    }
}
```

- **OnTriggerExit()**: este método detecta quien sale del trigger, y, en caso de que sea el jugador, desactiva el texto explicativo, quita la referencia del objeto interactuable y se borra la referencia del jugador.

```
private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player"))
    {
        textInfo.SetActive(false);

other.gameObject.GetComponent<PlayerMovment>().RemoveInteractableObject();
        player = null;
    }
}
```

- **DoInteraction()**; este método es el que implementa de la interfaz de la interacción, a través del cual se comprueba si el jugador se encuentra dentro, y en caso de ser así, se llama al método que se encarga de recuperar la salud del mismo.

```
public void DoInteraction()
{
    if(player != null)
    {
        player.GetComponent<PlayerMovment>().RestoreAllHealth();
    }
}
```

6.15. Personaje principal

Este apartado es uno de los más extensos y destacados del proyecto, ya que forma parte del objetivo base del desarrollo.

Para empezar a explicar el funcionamiento del mismo, podemos empezar por documentar la forma de controlar el personaje, ya que para eso contamos con un componente llamado “Character controller”, al que le podemos pasar la configuración de los controles por código. Esta parte la explicaremos más adelante. Los parámetros que hemos utilizado para este componente son los siguientes los cuales podemos ver en la Figura 6.15.0.1.

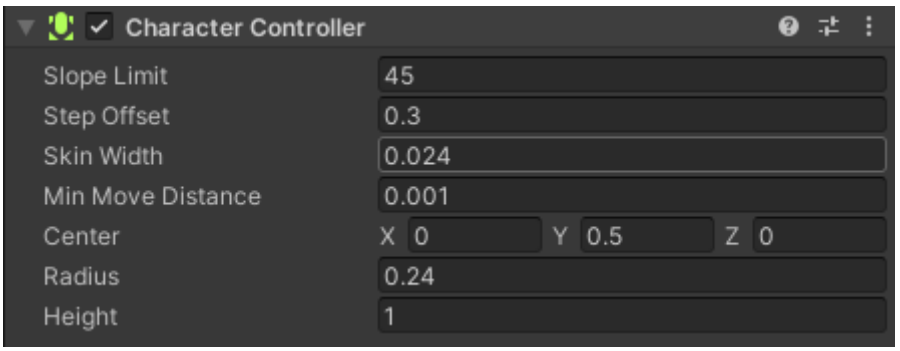


Figura 6.15.0.1: Captura de pantalla de los parámetros del Character Controller

Para poder bloquear el ataque de los enemigos, el prefab del jugador cuenta con un componente llamado “Box Collider”, el cual se sitúa delante de él y que se activa y desactiva mediante código. Esta funcionalidad se explicará más adelante. Se puede visualizar en la Figura 6.15.0.2.

Figura 6.15.0.2: Captura de pantalla del box collider del jugador

Además de estos componentes. también nos podemos encontrar con otros de los que ya hemos hablado con anterioridad:

- Player Input: que se encarga de gestionar todo el sistema de los inputs.
- Radial Bar General Controller: que en este caso contamos con 2, ya que uno nos sirve para controlar la estamina y el otro la salud.
- Saveable Entity: que nos permite establecer el identificador del personaje, que en este caso como queremos que sea el mismo en todas las escenas, cuenta con uno personalizado, llamado player.

- Audio Source: para poder reproducir todos los sonidos relacionados con el mismo.

A partir de ahora, nos vamos a encontrar con scripts únicos, que se encargan de gestionar el funcionamiento del jugador en gran parte de los aspectos más importantes.

6.15.1 Player Movement

Este script es el que se encarga de gestionar en gran medida todo el comportamiento del personaje, por lo que la extensión del mismo es bastante elevada.
Lo primero de todo, para tener un apartado por el que empezar, podemos hablar del apartado Update, el cual se puede considerar como el esqueleto de control del mismo.

- **Update()**: este método se encarga de realizar las siguientes acciones:
    - En caso de que el jugador esté muerto y sea necesario notificarlo, le envía un mensaje al script encargado de recargar la partida.
    - Si el jugador está muerto, bloquea el funcionamiento de las acciones.
    - **HandleMovement**(): este método se encarga de calcular la velocidad y dirección del movimiento.
    - En caso de que haya movimiento, calcula la dirección de la rotación a partir de este mediante **HandleRotation**().
    - El método **CheckBodyCollision**() se encarga de comprobar si alguna roca lanzada por los enemigos ha colisionado con el jugador y en caso de ser así, calcula el daño realizado.
    - **CalculateGravity()**: como el componente “Character Controller” no dispone de cálculo de la gravedad y no utilizamos root motion en cuanto a lo que animaciones se refiere, necesitamos calcularla por nuestra cuenta.
    - **CheckStaminaAmount()**: se encarga de comprobar cuando se está utilizando stamina y gestionarla adecuadamente.
    - **CheckPlayerSpeed()**: se utiliza para comprobar si el jugador está corriendo y las acciones necesarias relacionadas con lo mismo.

```
private void Update()
{
    if(isDead && hasToNotify)
    {
        hasToNotify = false;
        FindObjectOfType<ReloadGameOnDeath>().ReloadScene();
    }
    if (isDead) { return; }

    HandleMovement();

    if (targetVelocity != Vector3.zero) HandleRotation();

    CheckBodyCollision();

    CalculateGravity();

    CheckStaminaAmount();

    CheckPlayerSpeed();
}
```

Sistema de Input

Una de las primeras partes que necesitamos explicar para poder entender cómo funciona este código es la asignación de los valores de la entrada de teclado/mando. Las funciones que vamos a explicar a continuación están relacionadas con el input system, y se puede ver en la Figura 6.3.5.1.

- **SetMoveInput()**: este método se encarga de gestionar el movimiento del jugador. La primera parte de este se encarga de comprobar si la acción está en desarrollo (se refiere a si las teclas están siendo pulsadas, no al momento inicial de pulsarlas o al de soltarlas) y si el jugador no está haciendo acciones (acciones se refiere a talar, picar y atacar), y se almacena la información del movimiento en una variable. En caso de que se haya cancelado el movimiento o el jugador esté realizando acciones, el vector del movimiento se establece a nulo , se establece la animación de Idle y se establece la velocidad de andar.

```
public void SetMoveInput(CallbackContext ctx)
{
    if (ctx.performed && !doingActions)
    {
        MoveInput = ctx.ReadValue<Vector2>();

    }
    else if (ctx.canceled || doingActions)
    {
        MoveInput = Vector2.zero;
        basicActions.SetIdleAnimation();
        currentSpeed = walkSpeed;
    }

}
```

- **SetPlayerRunning()**: este método se encarga de controlar que se pueda realizar la acción de correr, ya que es necesario que el jugador no esté realizando acciones y que no esté bloqueando. Si el jugador está con la velocidad de caminar y aún tiene estamina, se establece la velocidad de correr. En caso contrario, se establece la velocidad de caminar.

```
public void SetPlayerRunning()
{
    if (doingActions || isBlocking ) { return; }
    if(currentSpeed == walkSpeed && !stamina.Empty())
    {
        currentSpeed = runningSpeed;
    }
    else
    {
        currentSpeed = walkSpeed;
    }
}
```

- **CheckBlocking()**: este método se puede dividir en dos partes principales, cuando empieza la acción y cuando finaliza. Si el jugador acaba de pulsar la tecla, no se están realizando acciones, se encuentra caminando y cuenta con suficiente estamina, y en caso de ser así, activa el collider frontal, establece un booleano de control a cierto y realiza la animación de bloquear. En caso de que se haya finalizado la acción de pulsar la tecla, se desactiva el collider, se establece a falso el booleano de control y se acaba la animación de bloquear.

```
public void CheckBlocking(CallbackContext ctx)
{

    if (ctx.started && !doingActions && currentSpeed == walkSpeed &&
stamina.GetCurrentValue() > 0)
    {
        frontCollider.enabled = true;
        isBlocking = true;
        animator.SetBool("Blocking", true);
    }
    else if (ctx.canceled)
    {
        frontCollider.enabled = false;
        isBlocking = false;
        animator.SetBool("Blocking", false);
    }
}
```

- Los métodos **SetInteractableObject**() y **RemoveInteractableObject**() se encargan de guardar y eliminar una referencia a través de la cual el jugador puede interactuar con ese objeto.

```
public void SetInteractableObject(IInteractor obj)
{
    interactable = obj;
}

public void RemoveInteractableObject()
{
    interactable = null;
}
```

- **MakeInteraction()**: este método se encarga de comprobar que nos encontremos al inicio de la acción y si el objeto para interactuar no es nulo, si es así, se realiza la interacción.

```
public void MakeInteraction(CallbackContext ctx)
{

    if (ctx.started && interactable != null)
    {
        sounds.Interactions();
        interactable.DoInteraction();
    }
}
```

Sistema de movimiento

Ahora que ya hemos explicado cómo se obtiene la información que es necesaria para la interacción del jugador, vamos a hablar de cómo se procesa todo el sistema de movimiento del mismo.

- **HandleMovement():** este método se encarga de calcular la velocidad actual. Partiendo de la información de movimiento se realiza una interpolación de velocidades entre la velocidad anterior y la futura, además de añadirle la velocidad de la gravedad, y realiza el desplazamiento del personaje mediante método Move.

```
private void HandleMovement()
{
    targetVelocity = new Vector3(MoveInput.x, 0, MoveInput.y).normalized *
currentSpeed;

    currentVelocity = Vector3.SmoothDamp(currentVelocity, targetVelocity, ref
velocitySmoothRef, velocitySmoothSpeed);

    currentVelocity = currentVelocity + gravityMovement;

    controller.Move(currentVelocity * Time.deltaTime);
}
```

- **HandleRotation()**: este método se encarga de calcular la rotación necesaria a partir de la velocidad que hemos calculado anteriormente gracias al método LookRotation.

```
private void HandleRotation()
{
    transform.rotation = Quaternion.LookRotation(targetVelocity, Vector3.up);
}
```

- **CheckPlayerSpeed()**: este método comprueba que en caso de que haya movimiento, el jugador esté corriendo y tenga estamina, se establece la animación de correr. En caso contrario, se comprueba si no queda estamina, que de ser cierto se establece un incremento menor (para castigar al jugador), se actualiza la velocidad a la de caminar y se establece la animación de caminar.

```
private void CheckPlayerSpeed()
{
    if (MoveInput == Vector2.zero) { return; }
    if (currentSpeed == runningSpeed && !stamina.Empty())
    {
        basicActions.SetRunAnimation();
    }
    else
    {

        if (stamina.Empty())
        {
            staminaIncrease = lowerIncrease;
            hasToRechargeStamina = true;
        }
        currentSpeed = walkSpeed;
        basicActions.SetWalkAnimation();
    }
}
```

Sistema de estamina

Ahora que ya sabemos cómo se gestiona el movimiento del personaje, necesitamos gestionar que se reduzca la estamina de una manera eficiente.

- **DecreaseStamina()**: este método recibe como parámetro la cantidad de stamina que hay que reducir y se la pasa al sistema de control de la stamina a través del método **DecreaseValue()**, del cual hemos hablado con anterioridad. Además, hace una llamada al sistema de control del hud para que se muestre la información relacionada con la estamina.

```
public void DecreaseStamina(float value)
{
    stamina.DecreaseValue(value);
    rbc.AwakeRadialBar();
}
```

- **CheckStaminaAmount()**: este método se encarga de comprobar si se está gastando estamina, y de ser así, gestiona las pérdidas correspondientes y cuando es necesario recargar estamina.
  Lo primero que comprueba es si está corriendo. En caso de ser así, reduce la estamina correspondiente. En caso de que el jugador se encuentre bloqueando y tenga estamina suficiente, reduce la estamina correspondiente a bloquear y establece a visibles el HUD. En caso de que no cuente con estamina y el método de incremento sea el elevado, se cambia al incremento menor, se cancela la

acción de bloquear si esta estaba activa y si no estaba activa, se establece la velocidad de caminar y la animación correspondiente. Si el jugador no está haciendo acciones, la cantidad de estamina no es la máxima y se puede recargar, se recarga la estamina. En caso de que se quedara sin estamina y ya se haya llenado el contador de la misma, se borra la penalización con la cual contaba el jugador.

```
private void CheckStaminaAmount()
{
    if (currentSpeed == runningSpeed)
    {

        stamina.DecreaseValue(actualStaminaLoss);
        rbc.AwakeRadialBar();
    }
    else if (isBlocking && stamina.GetCurrentValue() > 0)
    {
        stamina.DecreaseValue(lossBlocking);
        rbc.AwakeRadialBar();
    }
    else if (stamina.GetCurrentValue() <= 0 && staminaIncrease == higherIncrease)
    {
        staminaIncrease = lowerIncrease;
        hasToRechargeStamina = true;
        if (isBlocking)
        {
            isBlocking = false;
            animator.SetBool("Blocking", false);
        }
        else
        {
            currentSpeed = walkSpeed;
            basicActions.SetWalkAnimation();
        }
    }
    else if (!doingActions && stamina.GetCurrentValue() < 100  &&
stamina.CanRecharge())
    {
        actualStaminaLoss = staminaLoss;
        stamina.IncreaseValue(staminaIncrease);
    }
    if (hasToRechargeStamina && stamina.GetCurrentValue() == 100)
    {
        staminaIncrease = higherIncrease;
        hasToRechargeStamina = false;
    }
}
```

Sistema de colisiones

Para detectar si el jugador ha sido golpeado, necesitamos controlar el sistema de físicas, que en este caso, hemos hecho una implementación por código.

- **CheckBodyCollision()**: este método se encarga de comprobar si ha habido colisiones, utilizando el método propio de Unity llamado "Physics.OverlapCapsule", al cual le pasamos las dimensiones del personaje y la capa en la cual tiene que comprobar si ha habido colisiones. Esta información la guardamos en un vector y en caso de que la longitud sea mayor a 0, se reduce la vida del jugador y se elimina la capacidad de dañar del objeto que ha detectado.

```
void CheckBodyCollision()
{
    var layer = 8;
    var layermask = 1 << layer;

    Collider[] hitColliders = Physics.OverlapCapsule(this.transform.position, new
Vector3(this.transform.position.x, this.transform.position.y + 1.4f,
this.transform.position.z), 0.4f,layermask);

    if(hitColliders.Length > 0)
    {
        hitColliders[0].GetComponent<ThrowableRock>().CollidedWithEnemy();
        DecreaseHealth(10f);
    }
}
```

Sistema de gravedad

Como hemos comentado anteriormente, no poseemos gravedad directamente, por lo que necesitamos calcularla mediante código, y de eso se encarga la siguiente función:

- **CalculateGravity()**: este método comprueba si el jugador se encuentra en el suelo y en caso de ser así, reinicia la gravedad, pero si nó, calcula la gravedad en función del tiempo, es decir, la aceleración de la misma hasta un máximo establecido; y lo guarda en una variable para que se pueda utilizar en la parte del movimiento explicado anteriormente.

```
private void CalculateGravity()
{
    if (IsPlayerGrounded())
    {
        currentGravity = constantGravity;
    }
    else
    {
        if (currentGravity > maxGravity)
        {
            currentGravity -= gravity * Time.deltaTime;
        }
    }
    gravityMovement = gravityDirection * -currentGravity;
}
```

Sistema de acciones

Para poder integrar la parte que hemos desarrollado por independiente, contamos con un sistema de booleanos que nos permite detectar cuando el jugador está realizando acciones, y es necesario que se pause temporalmente el flujo de este script para que se puedan realizar.

```
public void SetDoingActions(bool state)
{
    doingActions = state;
}
public bool IsDoingActions()
{
    return doingActions;
}
```

Sistema de salud

Para controlar cómo se gestiona la salud, contamos con unos métodos que nos permiten reducir e incrementar los mismos.

- **RestoreAllHealth()**: este método se encarga de obtener el valor máximo de la salud que posee el jugador e incrementar la salud actual con esta, mostrando este cambio en el HUD.

```
public void RestoreAllHealth()
{
    health.IncreaseValue(health.GetMaxValue());
    rbc.AwakeRadialBar();
}
```

- **IncreaseHealth()**: este método recibe como parámetro el valor a incrementar de la salud, y se lo pasa al método "**IncreaseValue()"** del sistema de salud, que hemos explicado anteriormente y se muestra estos cambios en el HUD.

```
public void IncreaseHalth(float value)
{
    health.IncreaseValue(value);
    rbc.AwakeRadialBar();
}
```

- **DecreaseHealth()**: este método recibe como parámetro el daño recibido. Se encarga de realizar la animación de daño del jugador, reduce este valor pasándolo al método "**DecreaseValue()"** que hemos explicado anteriormente y muestra el HUD. En caso de que el jugador se quede sin vida, se establece a cierto el booleano que sirve para controlar si el jugador ha muerto y el booleano que sirve para reiniciar la partida. Además, se reproduce el sonido de daño del jugador y se reproduce la animación de muerte. En caso de que no se haya muerto el jugador se reproduce el sonido de daño. El script de los sonidos lo explicaremos más adelante.

```
public void DecreaseHealth(float value)
{
    animator.SetTrigger("Hit");
    health.DecreaseValue(value);
    rbc.AwakeRadialBar();
    if(health.GetCurrentValue() <= 0)
    {
        isDead = true;
        hasToNotify = true;
        sounds.PlayerKilled();
        animator.SetTrigger("Death");
    }
    else
    {
        sounds.DamagePlayer();
    }
}
```

Sistema de guardado

Para poder guardar la información que tiene este script, necesitamos implementar la interfaz ISaveable y la siguiente Struct formada por 2 SerializableVector3, la cual queda de la siguiente manera:

```
[System.Serializable]
struct PlayerSaveData
{
   public SerializableVector3 position;
   public SerializableVector3 rotation;
}
```

- **CaptureState()**: este método regresa una struct la cual contiene el valor de la posición y la rotación.

```
public object CaptureState()
{
   PlayerSaveData data = new PlayerSaveData();

   data.position = new SerializableVector3(this.transform.position);
   data.rotation = new SerializableVector3(transform.eulerAngles);
   return data;
}
```

- **RestoreState()**: este método recibe como parámetro un objeto, al cual le hacemos un cast para poder leer la información, y le asignamos la posición y rotación. Es importante destacar que para poder asignar esta información es necesario desactivar el gameobject, ya que de otra manera daría errores.

```
public void RestoreState(object state)
{
   PlayerSaveData data = (PlayerSaveData)state;
   controller.gameObject.SetActive(false);
   transform.position = data.position.ToVector();
   transform.eulerAngles = data.rotation.ToVector();
   controller.gameObject.SetActive(true);
}
```

Sistema de control del input

Para poder adaptarnos a las distintas situaciones que hay a lo largo del juego, necesitamos poder cambiar el sistema de input en runtime.

Para evitar errores ortográficos, contamos con un enun público que cuenta con los esquemas válidos:

```
public enum ControlSchemne
{
    Player,
    Menu,
    Inventory
}
```

- **ChangeInputMode()**: este método recibe el nombre del sistema de input que hace falta utilizar en cada momento, y lo cambia para cumplir con las necesidades.

```
public void ChangeInputMode(ControlSchemne mode)
{
    switch (mode)
    {
        case ControlSchemne.Player:
            playerInput.SwitchCurrentActionMap("Player");
            playerMode = true;
            break;
        case ControlSchemne.Menu:
            playerInput.SwitchCurrentActionMap("Menu");
            playerMode = false;
            break;
        case ControlSchemne.Inventory:
            playerInput.SwitchCurrentActionMap("Inventory");
            playerMode = false;
            break;
    }

}
```

6.15.2 Animator

Para poder controlar las animaciones con las que cuenta el personaje principal, necesitamos utilizar un animator. El animator del jugador cuenta con dos capas, ya que una se corresponde al cuerpo entero y otra solo a la parte superior del mismo. Para poder hacer esto necesitamos una máscara de cuerpo que solo afecte a la parte superior, como se puede ver en la Figura 6.15.2.1.

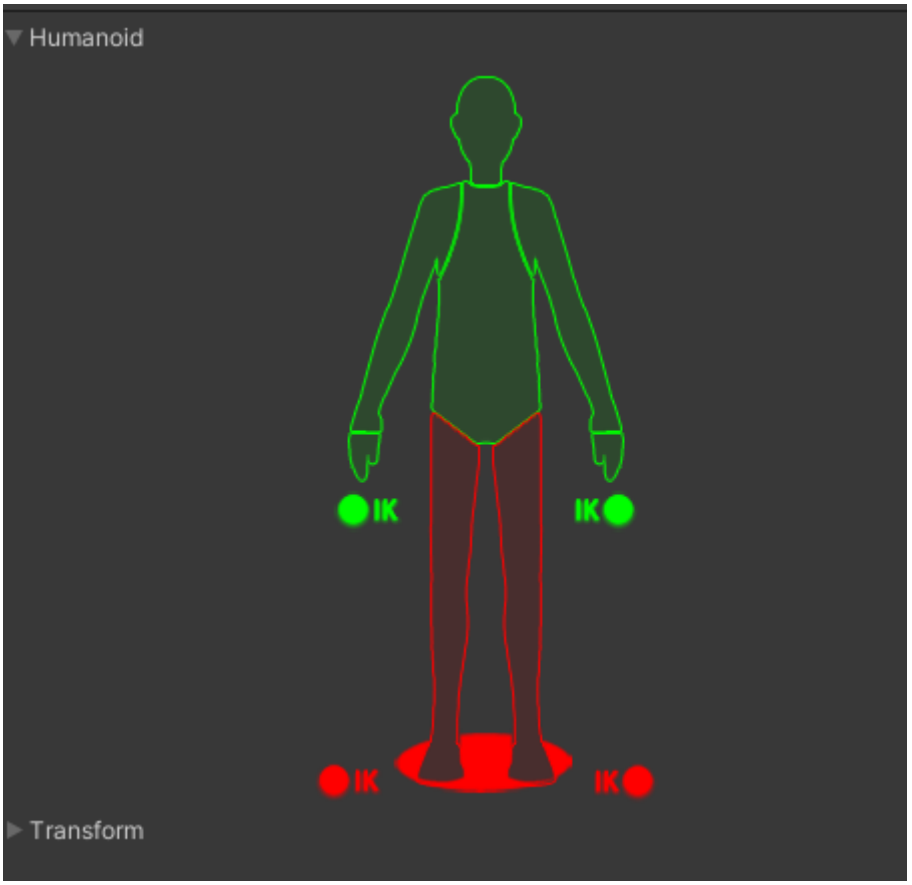


Figura 6.15.2.1: Captura de pantalla de la máscara del jugador

A la hora de crear esta capa en animator, establecemos los parámetros que se pueden apreciar en la Figura 6.15.2.2 para que se ajuste a nuestras necesidades.

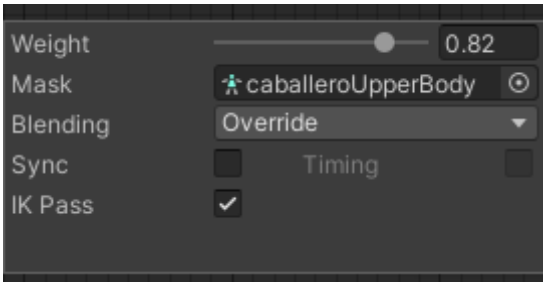


Figura 6.15.2.2: Captura de pantalla de los parámetros de la capa del animator

Para empezar, vamos a hablar de la capa base, a través de la cual se realizan la mayor parte de las animaciones:

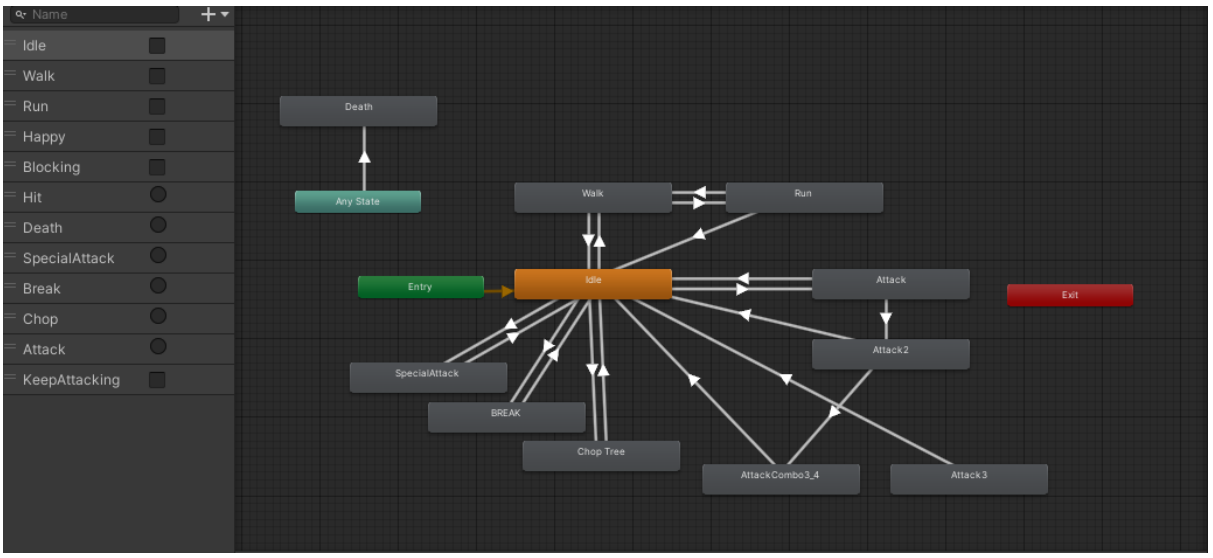


Figura 6.15.2.3: Captura de pantalla de los parámetros de la capa base del animator

Como podemos observar en la figura anterior, contamos con bastantes parámetros y estados de animación. Por defecto el personaje siempre está en estado de Idle, que es el central, a través del cual se puede navegar a los distintos estados. Cuando el jugador realiza el input de caminar, se pasa al estado de caminar, y sólo desde este se puede pasar al de correr, pero desde ambos se puede navegar al estado inicial de Idle. Para controlar estos cambios, contamos con los booleanos Idle, Walk y Run, que forman parte de las condiciones para pasar de un estado a otro y que estas transiciones no esperan a que se finalice la reproducción del estado para cambiar del mismo.

En cuanto al estado de muerte, va enlazado desde el estado propio de Unity, que es el any state, el cual permite acceder a los estados enlazados desde cualquier estado, donde se tiene que cumplir la condición de que el trigger “Death” sea cierto.

La siguiente parte a explicar es el UpperBody, donde nos encontramos con dos únicas animaciones, que son la animación de bloquear y recibir daño, a excepción de la animación inicial, que simplemente es un estado vacío para que no interfiera en las animaciones de la capa base. Todo esto lo podemos ver en la figura 6.15.2.3.

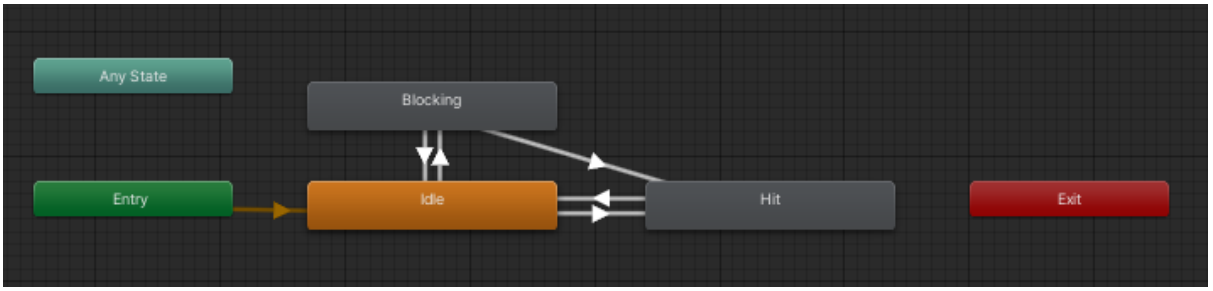


Figura 6.15.2.3: Captura de pantalla de los parámetros de la capa “UpperBody”del animator

6.15.2 Animaciones de las mecánicas y combinaciones

Para este apartado, resulta importante destacar que las animaciones referentes a las mecánicas de talar, picar o lanzar el orbe (o ataque especial) se implementan de una manera, mientras que las animaciones de la mecánica de ataque con espada se implementa de una manera ligeramente distinta debido a que consiste en una posible combinación de ataques con espada.

Por un lado están las animaciones de talar picar y el orbe (o ataque especial), las cuales consisten en una sola animación cada una con una serie de eventos que ejecutan unas pequeñas funciones de código en cierto punto de la animación. Primero de todo, se muestra el componente “Animator” en relación a dichas mecánicas.

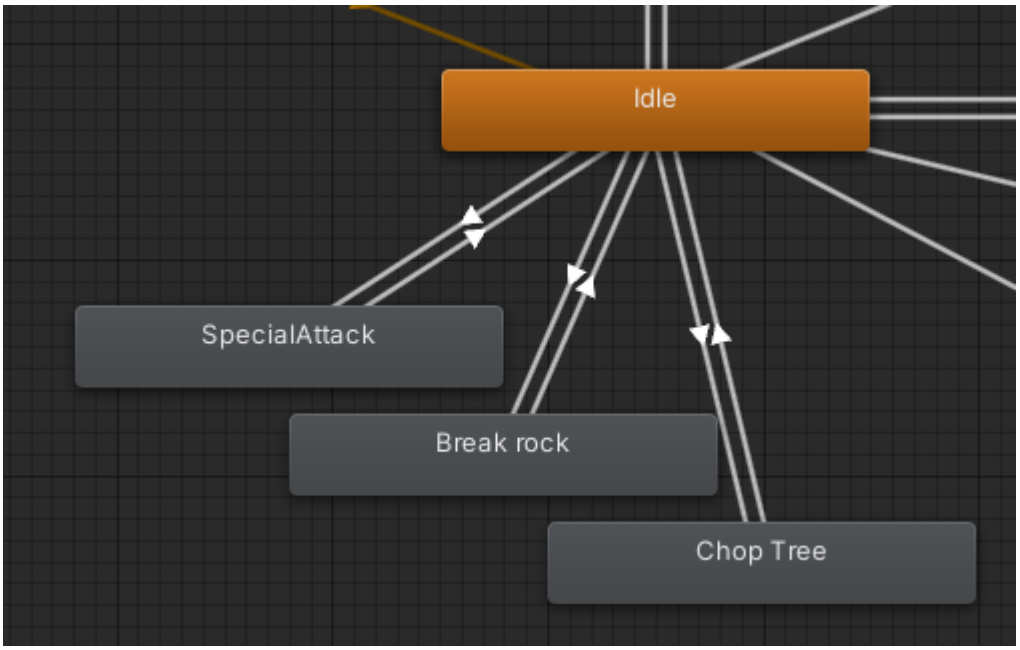


Figura 6.15.2.1.1: Captura de pantalla del componente Animator referente a las mecánicas de talar, picar y ataque especial

Todos los estados de la Figura 6.15.2.1.1 son muy similares entre ellos y, para evitar redundancia de imágenes, solo se mostraran capturas de pantalla de un solo estado (o mecánica) con los debidos comentarios y referencias a los demás estados cuando en caso necesario. Para ello, se toma como referencia el estado “Chop Tree” correspondiente a la animación de la mecánica de tala de árboles. Así pues, a continuación se muestra una figura que refleja la configuración del estado (válido para los dos restantes).

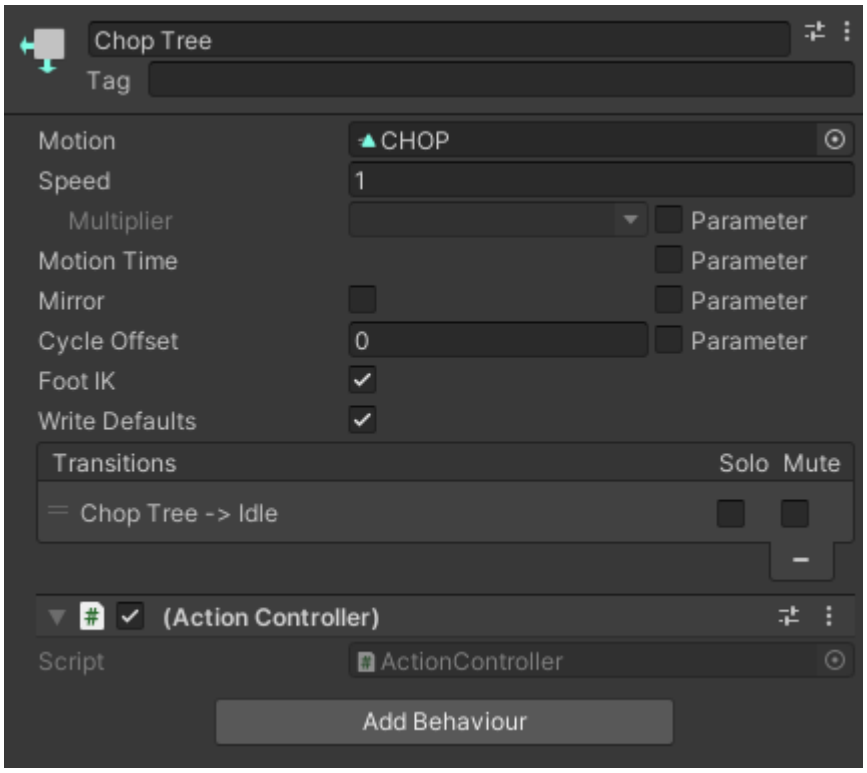


Figura 6.15.2.1.2: Captura de pantalla de la configuración del estado "Chop Tree" o tala de árboles

Lo único a destacar en la figura anterior es la presencia de un objeto de una clase derivada de la clase de Unity que gestiona las máquinas de estado de las animaciones. El siguiente código llama únicamente a un "callback" de Unity que se ejecuta cuando la máquina de estados de animaciones termina de ejecutar el estado que contiene un objeto de la clase que ejecuta el "callback" "**OnStateExit**", en otras palabras equivale a una función que se llama automáticamente cuando una animación se está terminando de ejecutar. Concretamente, lo que esta función ejecuta es otra función de un script del jugador (el jugador se identifica con el tag "Player") llamada "**SetDoingActions**", la cual restringe (o no) el movimiento del jugador. Es decir, el parámetro ajusta el movimiento del jugador y, en este caso, se le establece a "false", lo cual significa que ya no está realizando acciones y, por lo tanto, el movimiento se restablece con toda normalidad.

```
public class ActionController : StateMachineBehaviour
{
    // OnStateExit is called when a transition ends and the state machine finishes evaluating this state
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        GameObject.FindGameObjectWithTag("Player").GetComponent<PlayerMovment>().SetDoingActions(false);
    }
}
```

```
public void SetDoingActions(bool state)
{
    doingActions = state;
}
```

A continuación, en cuanto a la animación en sí misma, cabe destacar que tanto la animación de picar y talar resultan prácticamente idénticas ya que ambas llaman a los mismos eventos. La principal diferencia aparece en la animación del ataque especial donde se llaman solo a dos eventos totalmente distintos a los anteriores (más específicos para este).

Para las dos primeras animaciones, se usan los eventos de "**EnableTrigger**", el cual necesita de un parámetro de tipo enumerado con tres valores distintos (Axe, Pick, Sword) definido en la clase "**GunController**", y que es distinto si se trata de la animación de talar (Axe) o picar (Pick). Dicho evento sirve para activar el trigger del arma correspondiente para así poder detectar cualquier objeto con el que colisione. Después, se usa el evento "**DisableTrigger**" justamente para lo contrario que el anterior, es decir, para desactivar el trigger del arma correspondiente. Ambos eventos se complementan con el objetivo de establecer un rango de la animación dónde el trigger del arma esté activo con el fin de realizar la mecánica correspondiente. El último evento que llaman estas animaciones recibe el nombre de "**DisableActivatedGun**", el cual desactiva el arma al final del todo de la animación.

```
public void EnableTrigger(GunController.GunType type)
{
    if (type == activatedGun.gunType)
        activatedGun.EnableTrigger();
}

public void DisableTrigger()
{
    if (activatedGun != null)
        activatedGun.DisableTrigger();
}

public void DisableActivatedGun()
{
    //Desactiva el trigger
    this.DisableTrigger();
    //Oculta el arma
    if(activatedGun != null) activatedGun.gameObject.SetActive(false);

    //Establece el parámetro bool de seguir atacando a falso (por si el ataque es con espada)
    animator.SetBool("KeepAttacking", false);
    //Resetea el trigger de ataque para evitar reproducciones de las animaciones de ataque espontáneas
    animator.ResetTrigger("Attack");
}
```

Por otro lado, están también los métodos que realmente activan y desactivan los triggers de las herramientas y armas (lo que mantienen las referencia a dichos triggers).

```
public void EnableTrigger()
{
    trigger.enabled = true;
}

public void DisableTrigger()
{
    trigger.enabled = false;
}
```

Acto seguido, para la animación del ataque especial se usan eventos distintos como el evento llamado "**InstantiateSpecialAttack**", el cual se llama en el primer fotograma de la animación con la finalidad de instanciar el orbe encima del jugador y que posteriormente lanza. Por último, se llama al evento "**ThrowSpecialAttack**" que corresponde al momento justo en que se lanza dicho orbe.

```
public void InstantiateSpecialAttack()
{
    specialAttack_inst = Instantiate(specialAttackPrefab, specialAttackInitialPosition.position, Quaternion.identity) as GameObject;
    specialAttack_inst.GetComponent<SpecialAttackController>().SetReferenceDirection(specialAttackInitialPosition);

    playerMovement.DecreaseStamina(staminaFactor);
}

public void ThrowSpecialAttack()
{
    if (specialAttack_inst != null)
        specialAttack_inst.GetComponent<SpecialAttackController>().ThrowSpecialAttack();
}
```

De la clase "SpecialAttackController" se define el método "**SetReferenceDirection**", el cual toma una dirección como referencia a partir de la posición donde se ha inicializado el orbe (definida por un objeto vacío colocado a una cierta altura respecto al personaje) y el método "**ThrowSpecialAttack**" que lanza el orbe especial tomando como dirección la definida previamente.

```
public void SetReferenceDirection(Transform refDir)
{
    referenceDirection = refDir;
}

public void ThrowSpecialAttack()
{
    orbThrown = true;
    timeOrbThrown = Time.time;

    orbDirection = Quaternion.AngleAxis(25f, referenceDirection.right) * referenceDirection.forward;
}
```

Por otro lado y recuperando lo ya mencionado con anterioridad, también se presenta la combinación de ataques con espada, la cual es particularmente diferente al resto de

mecánicas. La siguiente figura muestra los estados correspondientes a la combinación de ataques con espada.

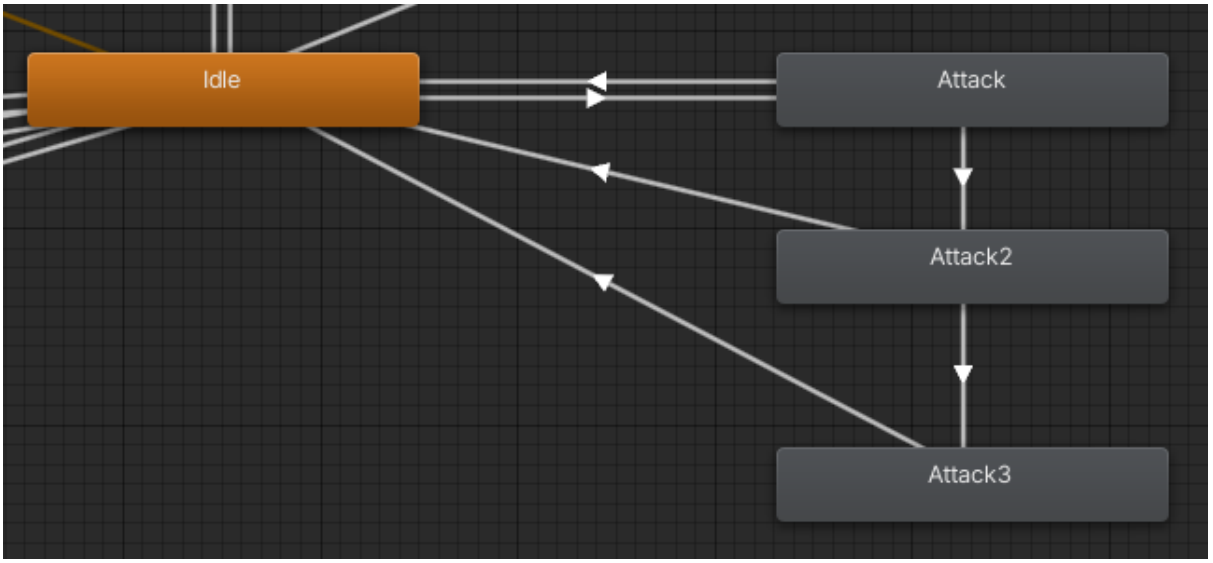


Figura 6.15.2.1.3: Captura de pantalla del componente Animator referente a la combinación de ataques con espada

La combinación de los ataques consiste en 3 posibles ataques distintos en cuanto a animación. Es decir, cuando el jugador inicia el primer ataque, debe presionar de nuevo la misma tecla antes de que termine la animación del primer estado para poder ejecutar el siguiente ataque (o estado) y así sucesivamente. Naturalmente, desde cada uno de los estados se puede volver al estado inicial ("Idle"). Se toma el estado "Attack" como referencia ya que el resto son prácticamente idénticos.

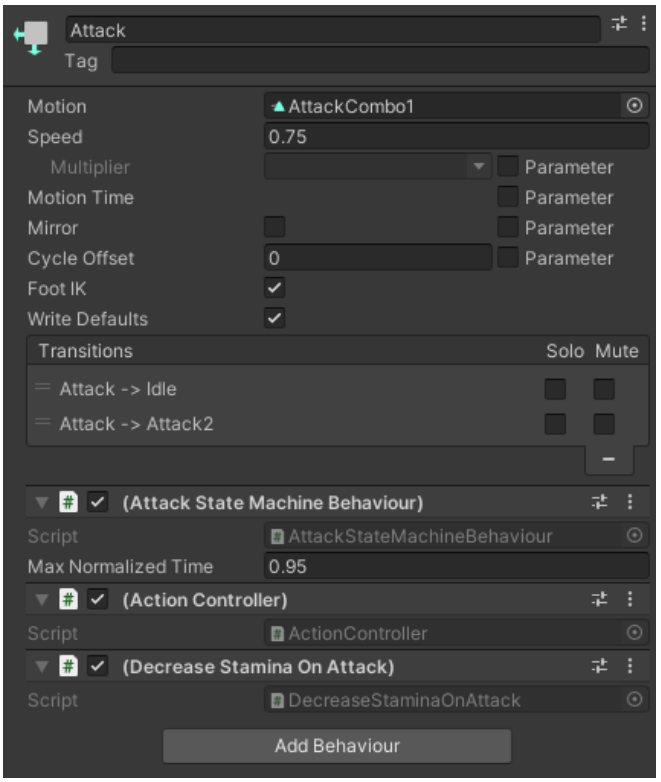


Figura 6.15.2.1.4: Captura de pantalla de la configuración del estado "Attack"

Como se puede ver en la figura anterior, los estados de ataque presentan tres objetos de clases derivadas de la ya mencionada clase de máquinas de estados de Unity. La primera función consiste en un "callback" que se llama automáticamente por Unity cuando se inicia el estado y resetea el trigger de ataque (el que define el ataque inicial), establece el atributo booleano "KeepAttacking" a false (usado para determinar si se ejecuta el siguiente estado de ataque), se establece que el jugador pueda llevar a cabo el siguiente ataque mediante la llamada a la función "**CanDoNextAttack**" y que, por defecto, no hay siguiente ataque.

```
// OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
   override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
   {
      animator.ResetTrigger("Attack");
      animator.SetBool("KeepAttacking", false);
      animator.GetComponent<CharacterInteraction>().CanDoNextAttack(true);
      nextAttack = false;
   }
```

```
public void CanDoNextAttack(bool state)
{
      if (state)
      {
         canDoNextAttack = state;
         keepAttacking = false;
      }
      else
      {
         keepAttacking = canDoNextAttack = state;
      }
}
```

La segunda consiste en un "callback" que se llama a cada fotograma de la animación y que va comprobando si el porcentaje normalizado de progreso de la animación es inferior a un valor previamente establecido como parámetro, si se puede seguir atacando, es decir, el jugador ya ha pulsado de nuevo la tecla de atacar y todavía no se había definido que atacaría de nuevo, se establece que habrá un próximo ataque (nextAttack) y se establece la variable "KeepAttacking" del "Animator" a cierto para proseguir al siguiente estado.

```
// OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
   override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
   {
      if (stateInfo.normalizedTime <= maxNormalizedTime &&
         animator.GetComponent<CharacterInteraction>().KeepAttacking() &&
         !nextAttack)
      {
         nextAttack = true;
         animator.SetBool("KeepAttacking", true);
      }
   }
```

La última función es el callback que ejecuta Unity cuando el estado está por finalizar. Se comprueba si hay próximo ataque y en caso que no haya, se desactiva el arma actual llamando a la función "**DisableActivatedGun**" (espada) y se establece que no pueda hacer un próximo ataque mediante la función "**CanDoNextAttack**".

```
// OnStateExit is called when a transition ends and the state machine finishes evaluating this state
   override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
   {
      if (!nextAttack)
      {
         animator.GetComponent<CharacterInteraction>().DisableActivatedGun();
         animator.GetComponent<CharacterInteraction>().CanDoNextAttack(false);
      }
   }
```

A continuación, también mantiene una referencia a un objeto de la clase "ActionController" vista anteriormente y también una referencia a un último objeto encargado de decrementar el nivel de estamina. Al inicio de la entrada al estado, Unity llama al "callback" "OnStateEnter" el cual decrementa la estamina correspondiente.

```
// OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
   override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
   {
      animator.GetComponent<CharacterInteraction>().DecreaseAttackStamina();
   }
```

6.15.3 PlayerSounds

Este script almacena todos los sonidos que necesita el jugador para dar una sensación redonda a la hora de jugar. Cabe destacar que este script es más bien de utilidades, por lo que la explicación que daremos va a ser superficial por su mera importancia. Para los sonidos de caminar, correr y daño se obtiene un clip aleatorio de la lista de sonidos y se reproduce, mientras que para los sonidos de interacción y muerte simplemente se reproduce el sonido disponible.

Eventos de animación de Unity

Lo interesante a destacar es cómo se hacen las llamadas de las funciones de andar y caminar, ya que utilizamos el sistema de eventos de Unity desde las animaciones.
Para las animaciones de caminar, hemos declarado unos eventos que buscan una función con el mismo nombre que la del evento. Para que resulte más realista, establecemos el evento un par de fotogramas antes que el personaje camine porque el sonido tiene un delay. Esto lo podemos ver en la Figura 6.15.3.1.

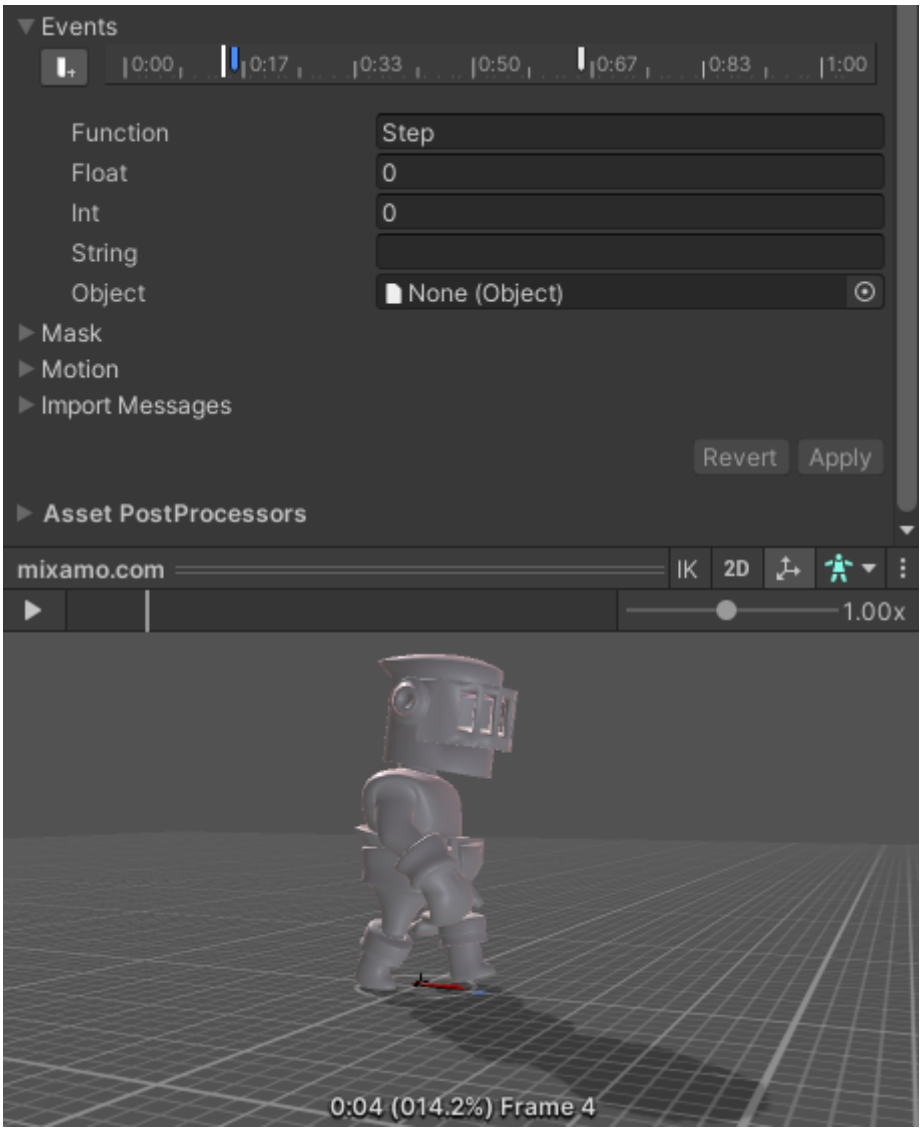


Figura 6.15.3.1: Captura de pantalla del sistema de eventos para la animación de andar

Para la animación de correr, nos encontramos con la misma situación, sólo que contamos con la diferencia de que la función a buscar se llama Run.

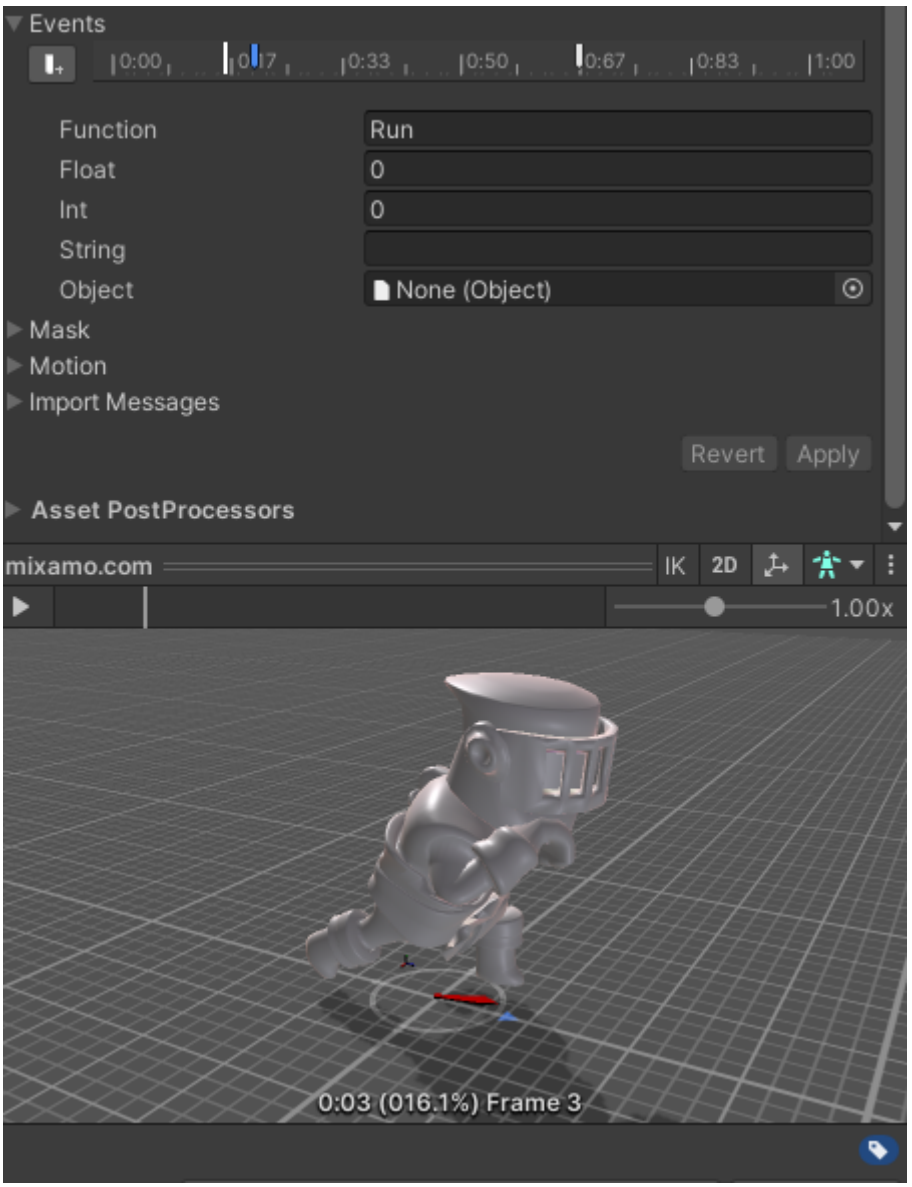


Figura 6.15.3.2: Captura de pantalla del sistema de eventos para la animación de correr

```
    #region - Damage -
    public void DamagePlayer()
    {
        audio.PlayOneShot(GetRandomClipDamage());
    }

    private AudioClip GetRandomClipDamage() =>
clipsDamage[UnityEngine.Random.Range(0, clipsDamage.Length)];

    #endregion

    #region - Walk -


    private void Step()
    {
        AudioClip clip = GetRandomClipWalk();
        audio.PlayOneShot(clip);
    }

    private AudioClip GetRandomClipWalk() =>
clipsWalk[UnityEngine.Random.Range(0, clipsWalk.Length)];

    #endregion

    #region - Run -

    private void Run()
    {
        AudioClip clip = GetRandomClipRun();
        audio.PlayOneShot(clip);
    }

    private AudioClip GetRandomClipRun() => clipsRun[UnityEngine.Random.Range(0,
clipsWalk.Length)];

    #endregion

    #region - Death -
    public void PlayerKilled()
    {
        audio.PlayOneShot(deathSound);
    }
    #endregion

    public void Interactions()
    {
        audio.PlayOneShot(interaction);
    }
```

6.15.4 Sistema de interacción por daño

En cuanto a las armas y herramientas que definen las mecánicas del juego, se procede a exponer su implementación y cómo funcionan. Primero de todo, destacar que el usuario dispone de un hacha para talar árboles, un pico para picar piedra y una espada para acabar con los enemigos. Por último y para completar las mecánicas, también se puede usar un ataque especial mediante el lanzamiento de un orbe.

Antes de nada, es necesario destacar que todas las armas y herramientas, a excepción del ataque especial, tienen asociado un script llamado "**GunController**" el cual controla y gestiona el comportamiento de cualquier arma o herramienta. Primero de todo, cabe mencionar que es este script quién guarda y crea el tipo enumerado para las armas anteriormente comentado en el apartado de animación (**GunType**). Cabe destacar que, a cada una de las armas y herramientas, se les asigna inicialmente el valor de **GunType** correspondiente.

```
public enum GunType
{
    Axe,
    Pick,
    Sword
}
```

A continuación, se define el evento que se llama automáticamente cuando el trigger del arma colisiona con cualquier otro collider, el cual comprueba si se trata de un objeto con el que un arma o herramienta pueda interaccionar mediante su tag y si ha interaccionando con el arma o herramienta correcta. También se comprueba si el objeto con el que ha colisionado implementa la interfaz **INteractableAssetInterface** y se llama al método de la interfaz (**OnDamage**).

```
private void OnTriggerEnter(Collider other)
  {
    if (other.CompareTag("GunInteractable") && this.CheckForCorrectInteractableGun(other.gameObject))
    {
      if (other.TryGetComponent<INteractableAssetsInterface>(out INteractableAssetsInterface element))
      {
        if(gunType != GunType.Sword)
        {
          audioPlayer.PlaySound();
        }
        element.OnDamage();
      }
      else
      {
        Debug.Log("Not interactable");
      }
    }
  }

  private bool CheckForCorrectInteractableGun(GameObject other)
  {
    List<GunType> interactableGuns = other.GetComponent<DestructibleElements>().GetGunToInteract();
    return interactableGuns.Contains(gunType);
  }
```

Acto seguido, se muestra la interfaz **INteractableAssetInterface** la cual declara una única función llamada OnDamage y que será implementada por cualquier otra clase que deba recibir daño o deba interaccionar con alguna de las armas y/o herramientas.

```
public interface INteractableAssetsInterface
{
    public void OnDamage() { }
}
```

Además, cada uno de dichos elementos interactuables heredan también de una clase llamada “DestructibleElements” la cual mantiene una lista pública de armas y herramientas (tipo enumerado “GunType”) con las que el objeto puede interaccionar y proporciona un método para obtener dicha lista.

```
public class DestructibleElements : MonoBehaviour
{
    [SerializeField] protected List<GunController.GunType> gunsToInteract;

    public List<GunController.GunType> GetGunToInteract()
    {
        return gunsToInteract;
    }
}
```

En cuanto al ataque especial, éste no necesita implementar ninguna interfaz ni heredar de ninguna otra clase ya que su núcleo principal reside en dos funciones. Una es un “**Update**” llamado a cada fotograma del juego y el otro es un “**OnTriggerEnter**” procedente del trigger esférico del orbe. Dentro del “Update” se va moviendo el orbe y, si ha pasado cierto tiempo, se destruye. Dentro del “OnTriggerEnter”, cuando choca con otro objeto físico, comprueba si es un elemento interaccionar a través del tag y si tiene implementada la interfaz INteractableAssetInterface. Si es así, llama a la función “OnDamage”. Por el contrario, si tan solo choca con cualquier objeto del entorno, reproduce una animación que lo termina por destruir.

```
private void Update()
{
      if (orbThrown)
      {
         transform.Translate(5f * Time.deltaTime * orbDirection);
         if (Time.time - timeOrbThrown > 20f)
            this.DestroySpecialAttack();
      }
   }

private void OnTriggerEnter(Collider other)
{
      if (other.CompareTag("GunInteractable"))
      {
         if (other.TryGetComponent<INteractableAssetsInterface>(out INteractableAssetsInterface element))
         {
            element.OnDamage();
         }
         else
         {
            Debug.Log("Not interactable");
         }

         audioSource.clip = destructionClip;
         audioSource.Play();
         orbThrown = false;
         GetComponent<Animator>().SetTrigger("OrbExplosion");
      }
}
```

Además, cada una de las armas y herramientas (en este caso también se incluye el ataque especial), tienen una función que se llama cuando se pulsa la tecla adecuada para activar la mecánica. Para el caso del hacha y el pico, son dos funciones distintas (ya que son dos teclas distintas), pero muy similares entre ellas. Se considera la función correspondiente al hacha como referencia. Cuando se pulsa la tecla adecuada, se comprueba que el jugador esté en el “estado inicial” (se considera estado inicial cuando está en Idle) y tiene suficiente estamina, en cuyo caso establece que el jugador está realizando acciones para restringir el movimiento, se resta estamina, se activa el arma o herramienta oportuna y se activa el trigger de animación correspondiente.

```
public void Axe()
{
    if (CheckPlayerIinitalState() && HasEnoughtStamina(staminaAxe))
    {
        playerMovement.SetDoingActions(true);
        playerMovement.DecreaseStamina(staminaAxe);
        this.SetActivatedGun(GunController.GunType.Axe);
        animator.SetTrigger("Chop");
    }
}

private bool CheckPlayerIinitalState()
{
    AnimatorStateInfo stateInfo = animator.GetCurrentAnimatorStateInfo(0);
    AnimatorStateInfo stateInfo2 = animator.GetCurrentAnimatorStateInfo(1);
    if (stateInfo.IsName("Idle") && stateInfo2.IsName("Idle") &&
        !playerMovement.IsDoingActions())
    {
        return true;
    }
    else return false;
}

private bool HasEnoughtStamina(float requiredStamina)
{
    return playerMovement.GetCurrentStamina() >= requiredStamina;
}
```

A continuación, la función cambia ligeramente para el caso de la espada. En este caso, al pulsar la tecla adecuada también se comprueba si está en el estado inicial y si tiene suficiente estamina. La principal diferencia está en que gestiona la variable "KeepAttacking" para comprobar desde el "Animator" si se debe pasar a la siguiente combinación o no.

```
public void Sword(CallbackContext ctx)
{
    if (ctx.started)
    {
        if (CheckPlayerIinitalState() && HasEnoughtStamina(staminaAttack))
        {
            playerMovement.SetDoingActions(true);

            this.SetActivatedGun(GunController.GunType.Sword);
            animator.SetTrigger("Attack");
            keepAttacking = false;
        }
        else if (canDoNextAttack && HasEnoughtStamina(staminaAttack) && !keepAttacking)
        {
            keepAttacking = true;
        }
    }
}
```

Finalmente, para el ataque especial la función tampoco varía mucho. Simplemente, comprueba igual si tiene estamina y si está en Idle y restringe los movimientos del jugador (SetDoingActions) y activa el trigger para reproducir la animación.

```
public void Orb()
{
    if(CheckPlayerIinitalState())
    {
        playerMovement.SetDoingActions(true);

        if (playerMovement.GetCurrentStamina() >= staminaFactor)
            animator.SetTrigger("SpecialAttack");
        else
            playerMovement.SetDoingActions(false);
    }
}
```

Acto seguido, se debe resaltar cómo ciertos objetos interactuables (árboles y rocas) reaccionan ante una interacción con un arma o herramienta. Para ello, cada uno debe implementar su propia función “OnDamage” de la interfaz anterior y mostrar ciertas particularidades. Primero, se muestran los tres modelos que se usan para mostrar e interactuar con los árboles.

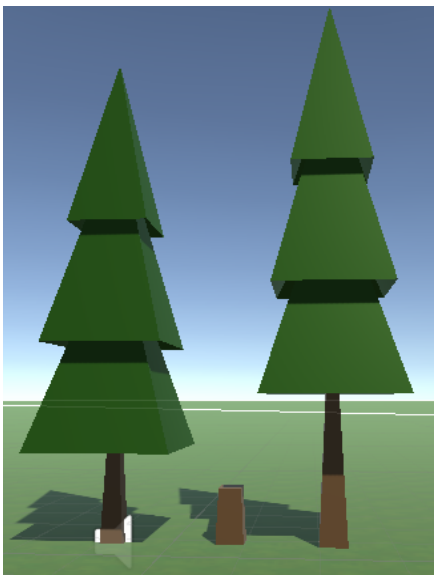


Figura 6.15.1.1: Captura de pantalla de los distintos modelos de árboles que se usan.

En primer lugar, se usan tres modelos distintos para diferenciar entre el árbol al completo y de pie (derecha), el tronco cuando es talado (en medio) y la parte que cae cuando se tala (izquierda). Para la parte entera y de pie, se usa el script “TreeController” el cual implementa la función “OnDamage” e instancia tanto el tronco inferior y la parte

del árbol que cae en sus posiciones correspondientes y se activa el sistema de partículas que simula las pequeñas cortezas de un árbol que salen disparadas.

```
public void OnDamage()
{
    GameObject go = Instantiate(fixedStump, transform.position, transform.rotation) as GameObject;
    go.transform.localScale = transform.localScale/1.3f;
    Bounds b = go.GetComponent<Collider>().bounds;

    GameObject aux = Instantiate(fallenStump, transform.position + (Vector3.up * b.size.y),  transform.rotation);
    aux.transform.localScale = transform.localScale;
    if (particleSys != null)
      particleSys.Play();

    gameObject.GetComponent<Collider>().enabled = false;
    respawn.HasToRespawnObject(transform, RespawnObjects.SpawnObject.Tree, 40f ,go);
    gameObject.SetActive(false);
    Destroy(gameObject);
}
```

Finalmente, la parte del árbol que cae contiene un collider, un trigger y un rigidbody para la parte de física, además de un script llamado “FallenStumpController” el cual únicamente implementa la interfaz INteractableAssetInterface. En cuanto al código, implementa el callback “OnEnable” que es llamado cuando se activa el objeto (en este caso al ser instanciado por la clase anterior) e inicializa todas las referencias necesarias, reproduce un sonido y genera una rotación del vector derecho del jugador en un ángulo aleatorio para aplicar un impulso en esa dirección con el objetivo de que caiga hacia la derecha del jugador (ya que empuña el hacha con la mano derecha) consiguiendo así el efecto de tala.

```
private void OnEnable()
{
    rb = GetComponent<Rigidbody>();
    col = GetComponent<Collider>();
    audioS = GetComponent<AudioSource>();

    audioS.PlayOneShot(fallingTree);

    Transform playerTransform = GameObject.FindGameObjectWithTag("Player").transform;

    //Creamos una rotacion del vector "right" del jugador pera generar la direccion en la que va a caer el arbol
    //Para que caiga siempre hacia la derecha del jugador (mas realista debido a que empuna el hacha con la derecha),
    //ajustamos el valor minimo del Random
    Vector3 vec = Quaternion.AngleAxis(Random.Range(-10f, -85f), Vector3.up) * playerTransform.right;
    rb.AddForce(vec * 500f, ForceMode.Impulse);
}
```

Acto seguido, también se implementa la función “OnDamage“ que instancia el recurso de la madera que podrá ser recogido por el jugador y le aplica un pequeño impulso en vertical enfatizando el efecto “cartoon” del videojuego.

```
public void OnDamage()
{
    GameObject woodAsset_inst = Instantiate(woodAsset, col.bounds.center, Quaternion.identity);

    woodAsset_inst.GetComponentInChildren<Rigidbody>().AddForce(Vector3.up * 1f, ForceMode.Impulse);

    hoja.SetActive(false);
    tronco.SetActive(false);

    audioS.PlayOneShot(breakTree);
    rb.Sleep();
    col.enabled = false;
    Destroy(gameObject, 3f);
}
```

Por último, también se implementan las funciones de “OnTriggerEnter” y “OnCollisionEnter” para cuando el árbol talado impacta con el suelo y éste llame a su **propia** función “OnDamage” y para cuando el árbol talado impacta con otro árbol (de pie) y llama a **su** función “OnDamage” respectivamente.

```
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.TryGetComponent<TreeController>(out TreeController collision_tree_controller))
    {
        collision_tree_controller.OnDamage();
    }
}

private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Floor") && !floorHit)
    {
        floorHit = true;
        this.OnDamage();
    }
}
```

Después, las rocas resultan elementos más sencillos ya que se componen sólo de un modelo y un script llamado “RockController” qué también hereda de la clase “DestructibleElements” e implementa la interfaz anteriormente vista.

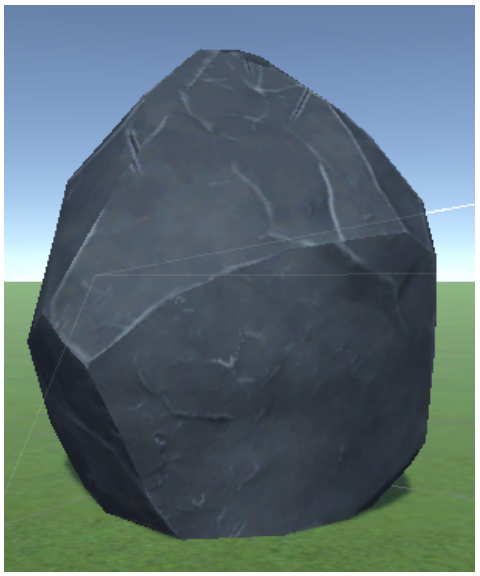


Figura 6.15.1.2: Captura de pantalla del modelo de roca usado

La función “OnDamage” de “RockController” se limita a instanciar el recurso de piedra para que el jugador pueda recogerlo y reproducir un sonido.

```
public void OnDamage()
{
    Instantiate(rock_resource_asset, transform.position, Quaternion.identity);
    respawn.HasToRespawnObject(this.transform, RespawnObjects.SpawnObject.Rock,40f);

    meshR.enabled = false;
    box.enabled = false;
    audioS.PlayOneShot(breakRock);

    Destroy(gameObject, 5f);
}
```

6.16 Sistema de regiones

Con la intención de poder optimizar el sistema de recursos del ordenador y poder implementar un sistema de música ambiental por regiones, contamos con este sistema de regiones, el cual cuenta con unos cuantos componentes de control.

- Colliders de control: para poder detectar en qué zona se encuentra el jugador.
- Region Controller: script que se encarga de controlar las acciones a realizar en función de la región que sea.
- Music Blender: este script se encarga de gestionar qué canciones deben sonar en función de la ubicación del jugador.

6.16.1 Region Colliders

Para saber qué ubicación está visitando el jugador necesitamos colocar unos box colliders, los cuales funcionan como triggers, a través de los cuales lo podamos controlar. Esto se puede ver en la Figura 6.16.1.1.

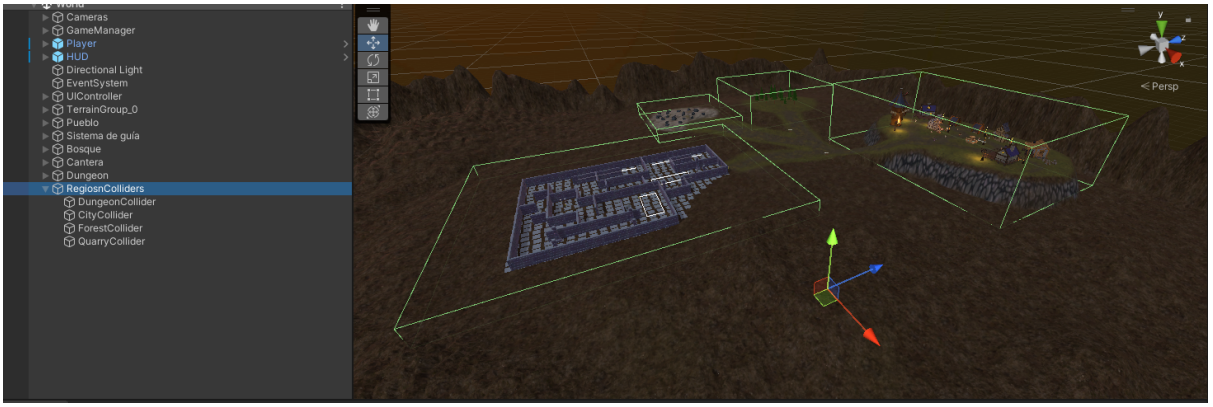


Figura 6.16.1.1: Captura de pantalla del sistema de colliders.

6.16.2 Region Controller

Además, cada gameobject que cuenta con este trigger, también tienen el script RegionController, del cual vamos a hablar ahora. Lo primero que necesitamos saber para el buen funcionamiento de este script, es conocer la región con la que estamos trabajando. Para esto cuenta con un Enum que nos da a elegir entre 4 zonas posibles: Ciudad, Bosque, Cantera o Mazmorras; y que se elige a través de inspector.

- **Start()**: en este método se busca al componente que se encarga de controlar el sistema de música, y en caso de ser necesario desactivar el terreno, se llama a la corrutina encargada de hacerlo.

```
private void Start()
{
    blender = FindObjectOfType<MusicBlender>();
    if (hasToDisable)
    {
        StartCoroutine(SetStateOfRegion(false));
    }
}
```

- **NotifyRegionStatus()**: este método recibe como parámetro la información de si la región actual tiene que estar activada o desactivada, y lo notifica al script que se encarga de la música a través de un método único para cada región.

```
private void NotifyRegionInformation(bool status)
{
    switch (regionType)
    {
        case RegionType.city:
            blender.CityRegion(status);
            break;
        case RegionType.dungeon:
            blender.DungeonRegion(status);
            break;
        case RegionType.forest:
            blender.ForestRegion(status);
            break;
        case RegionType.quarry:
            blender.QuarryRegion(status);
            break;
    }
}
```

- **OnTriggerEnter()**: este método se encarga de comprobar que gameObject entra en el collider, y en caso de ser el jugador, llama al método que se encarga de notificar este cambio en la región y llama a la corrutina que se encarga de activar los objetos del escenario.

```
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.CompareTag("Player"))
    {
        NotifyRegionInformation(true);
        StartCoroutine(SetStateOfRegion(true));
    }
}
```

- **OnTriggerExit()**: este método se encarga de detectar que gameObject sale del trigger, y en caso de que sea el jugador, desactiva los objetos pertenecientes a esta región y llama al método que se encarga de notificar que se ha salido de la región para cambiar la música ambiental.

```
private void OnTriggerExit(Collider other)
{
    if (other.gameObject.CompareTag("Player"))
    {
        NotifyRegionInformation(false);
        StartCoroutine(SetStateOfRegion(false));
    }
}
```

6.16.3 Music Blender

Este método se encarga de gestionar la canción que debe de sonar en cada momento. Para poder hacer esto, contamos con un método por región, el cual cambia el estado de la región, ya sea a cierto o falso, reinicia el contador para hacer la fusión de la música y se pone a cierto el booleano de control de cambiar música.

```
public void CityRegion(bool state)
{
   city = state;
   timeStartedBlending = Time.time;
   hasToBlendMusic = true;
}

public void ForestRegion(bool state)
{
   forest = state;
   timeStartedBlending = Time.time;
   hasToBlendMusic = true;
}

public void QuarryRegion(bool state)
{
   quarry = state;
   timeStartedBlending = Time.time;
   hasToBlendMusic = true;
}

public void DungeonRegion(bool state)
{
   dungeon = state;
   timeStartedBlending = Time.time;
   hasToBlendMusic = true;
}
```

- **Update()**: este método se encarga de comprobar si es necesario hacer la fusión de la música, y en caso de ser así, se obtiene el porcentaje actual a partir del tiempo que ha pasado desde que empezó. En función de que región sea la que esté activa, se activará un volumen u otro. Cuando el porcentaje llega a 1, se desactiva el booleano de control para mezclar la música.

```
    private void Update()
    {
        if (hasToBlendMusic)
        {
            float timeSinceStarted = Time.time - timeStartedBlending;
            float percentage = timeSinceStarted / blendTime;

            if (city)
            {
                cityMusic.volume = Mathf.Lerp(cityMusic.volume,maxVolume, percentage);
                worldMusic.volume = Mathf.Lerp(worldMusic.volume, 0f, percentage);
                forestMusic.volume = Mathf.Lerp(forestMusic.volume, 0f, percentage);
                quarryMusic.volume = Mathf.Lerp(quarryMusic.volume, 0f, percentage);
                dungeonMusic.volume = Mathf.Lerp(dungeonMusic.volume, 0f, percentage);
            }
            else if (forest)
            {
                cityMusic.volume = Mathf.Lerp(cityMusic.volume, 0f, percentage);
                worldMusic.volume = Mathf.Lerp(worldMusic.volume, 0f, percentage);
                forestMusic.volume = Mathf.Lerp(forestMusic.volume, maxVolume, percentage);
                quarryMusic.volume = Mathf.Lerp(quarryMusic.volume, 0f, percentage);
                dungeonMusic.volume = Mathf.Lerp(dungeonMusic.volume, 0f, percentage);
            }
            else if (quarry)
            {
                cityMusic.volume = Mathf.Lerp(cityMusic.volume, 0f, percentage);
                worldMusic.volume = Mathf.Lerp(worldMusic.volume, 0f, percentage);
                forestMusic.volume = Mathf.Lerp(forestMusic.volume, 0f, percentage);
                quarryMusic.volume = Mathf.Lerp(quarryMusic.volume, maxVolume, percentage);
                dungeonMusic.volume = Mathf.Lerp(dungeonMusic.volume, 0f, percentage);
            }
            else if (dungeon)
            {
                cityMusic.volume = Mathf.Lerp(cityMusic.volume, 0f, percentage);
                worldMusic.volume = Mathf.Lerp(worldMusic.volume, 0f, percentage);
                forestMusic.volume = Mathf.Lerp(forestMusic.volume, 0f, percentage);
                quarryMusic.volume = Mathf.Lerp(quarryMusic.volume, 0f, percentage);
                dungeonMusic.volume = Mathf.Lerp(dungeonMusic.volume, maxVolume,
percentage);
            }
            else
            {
                //Esto implica que es la música general del mundo
                cityMusic.volume = Mathf.Lerp(cityMusic.volume, 0f, percentage);
                worldMusic.volume = Mathf.Lerp(worldMusic.volume, maxVolume, percentage);
                forestMusic.volume = Mathf.Lerp(forestMusic.volume, 0f, percentage);
                quarryMusic.volume = Mathf.Lerp(quarryMusic.volume, 0f, percentage);
                dungeonMusic.volume = Mathf.Lerp(dungeonMusic.volume, 0f, percentage);
            }
            if (percentage == 1) hasToBlendMusic = false;
        }
    }
```

6.17 Sistema de Quest

Para darle un puntillo al juego, hemos desarrollado un sistema de quest, a través de las cuales el jugador puede ir avanzando en la historia y así, descubrir las mecánicas de una manera ordenada, aunque el jugador puede hacer caso omiso a estas indicaciones e investigar libremente.

6.17.1 Quest Controller

De la primera clase que hablaremos es QuestController, que se encarga de gestionar las misiones con las que cuenta el jugador. Este script funciona en conjunto a otro llamado **QuestInteractor**, a través del cual el jugador interactúa.

- **Start()**: este método se encarga de añadir el componente actual al sistema de observer del inventario, para que cuando el jugador consiga un recursos se pueda comprobar si ha completado la misión. Además, llama al método AddDialogs(), el cual carga los diálogos en una array para poder enseñarlos al jugador cuando sea necesario.

```
private void Start()
   {

GameObject.FindGameObjectWithTag("CityHall").GetComponent<Subject>().AddObs
erver(this);
      isObserving = true;
      AddDialogs();
   }
```

- **OnNotify()**: este método forma parte del sistema de Observer, a través del cual recibe la información de cuál es el material que ha conseguido el jugador, y en caso de que se hayan conseguido las unidades necesarias, se actualiza toda la información esencial para pasar a la siguiente misión.

```
public void OnNotify(string name, int count)
    {
        if(currentQuest != null)
        {
            if(currentQuest.GetResourceType() == name && count >= currentQuest.count)
            {
                FindObjectOfType<QuestInteractor>().HasToUpdateText();
                questCompletada = true;
                hudInfo.text = "Enhorabuena, has completado la misión. Habla con el aldeano.";
            }
        }
    }
```

- **SetNextDialog()**: este método se encarga de gestionar cuál es el próximo diálogo que se mostrará al jugador, en función de en qué parte de la historia se encuentre, es decir, si está en el tutorial inicial, si tiene una misión activa, si hay que cambiar de misión, o si ya ha acabado la historia del juego. En caso de que el jugador ya haya acabado las misiones, se elimina el sistema de observer para liberar recursos.

```
public string SetNextDialog()
    {
        string currentInfo = "";
        if(currentDialog < dialogLoader.Count)
        {
            currentInfo = dialogLoader[currentDialog];
            currentDialog++;
        }
        else if (questCompletada)
        {
            currentInfo = "Enhorabuena, has completado el encargo. Gracias por tu ayuda.";
            currentQuest = null;
            questCompletada = false;
        }
        else if (currentQuest != null)
        {
            currentInfo = "Tienes que conseguir " + currentQuest.count + " de " + TranslateWord( currentQuest.GetResourceType() )+ ".";
            hudInfo.text = "Tienes que conseguir " + currentQuest.count + " de " + TranslateWord(currentQuest.GetResourceType() )+ ".";
        }
        else if (questNumber < questList.Count)
        {
            currentQuest = questList[questNumber];
            currentInfo = "Tienes que conseguir " + currentQuest.count + " de " + TranslateWord( currentQuest.GetResourceType()) + ".";
            questNumber++;
        }
        else
        {
            currentInfo = "Es hora de que salgas a investigar los calabozos y descubras los tesoros que contienen.";
            if (isObserving)
            {
                GameObject.FindGameObjectWithTag("CityHall").GetComponent<Subject>().RemoveObserver(this);
                isObserving = false;
                hudInfo.text = "Investiga los calabozos.";
            }
        }
        return currentInfo;
    }
```

Para poder continuar en la misión con la que estaba el jugador, se ha implementado la interfaz del guardado en este script, que cuenta con tres métodos básicos para el funcionamiento:

- **Capture State()**: este método guarda una struct que contiene el número del diálogo actual y el número de la misión actual.

```
[System.Serializable]
   private struct QuestSaveData
   {
      public int questNumber, currentDialog;
   }

   public object CaptureState()
   {
      QuestSaveData data = new QuestSaveData();
      data.currentDialog = currentDialog;
      data.questNumber = questNumber;
      return data;

   }
```

- **Restore State()**: este método se encarga de recuperar la struct guardada, asignar los valores que contiene, y cargar todos los textos de feedback del jugador gracias al método **LoadCurrentDialog**().

```
public void RestoreState(object state)
   {
      QuestSaveData data = (QuestSaveData)state;
      currentDialog = data.currentDialog;
      questNumber = data.questNumber;
      LoadCurrentDialog();
   }
```

- **LoadCurrentDialog()**: este método realiza una función similar al **SetNextDialog**(), con la diferencia de que no avanza de diálogo.

```
private void LoadCurrentDialog()
   {
      string currentInfo = "";
      if (currentDialog < dialogLoader.Count)
      {
         currentInfo = dialogLoader[currentDialog];
      }
      else if(questNumber < questList.Count)
      {
         currentQuest = questList[questNumber];
         currentInfo = "Tienes que conseguir " + currentQuest.count + " de " +
currentQuest.GetResourceType();
      }
      else
      {
         currentInfo = "Es hora de que salgas a investigar los calabozos y descubras los
tesoros que contienen";
         if (isObserving)
         {

GameObject.FindGameObjectWithTag("CityHall").GetComponent<Subject>().Remove
Observer(this);
            isObserving = false;
         }
      }
      FindObjectOfType<QuestInteractor>().SetTextOnLoad(currentInfo);
   }
```

6.17.2 Quest Interactor

Esta clase se encarga de mantener un sistema de interacción entre el jugador y las quest. Además, también actualiza la información de las misiones según va interactuando el jugador. El componente es asignado a un NPC que se sitúa en la villa, en un lugar visible para facilitar su uso.

- **OnTriggerEnter():** este método se encarga de comprobar cuando entra alguien al collider, y en caso de ser el jugador, se pone a cierto un booleano de control, se establece el objeto como interactuable dentro del script del jugador, se modifica el trigger del NPC para que reproduzca la animación de hablar, se activa el gameObject que muestra la información del NPC y se desactiva el recordatorio de misión en el HUD.

```
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.CompareTag("Player"))
    {
        playerInside = true;

other.gameObject.GetComponent<PlayerMovment>().SetInteractableObject(this);
        animator.SetTrigger("Talking");
        textBackground.SetActive(true);
        hudInfo.SetActive(false);
    }
}
```

- **OnTriggerExit()**: este método detecta si un objeto sale del trigger y en caso de que fuera el jugador, se pone a falso la variable de control, se quita el objeto de interactuable, se establece el trigger de saludar, se desactiva el texto donde se muestra la información del NPC y se activa el recordatorio de misión.

```
private void OnTriggerExit(Collider other)
{
    if (other.gameObject.CompareTag("Player"))
    {
        playerInside = false;

other.gameObject.GetComponent<PlayerMovment>().RemoveInteractableObject();
        animator.SetTrigger("Salute");
        textBackground.SetActive(false);
        hudInfo.SetActive(true);
    }
}
```

- **DoInteraction()**: este método se ejecuta por llamada del jugador gracias a la interfaz que implementa, y en caso de que el jugador esté dentro del trigger, se llama a la función que se encarga de cargar este texto mediatne el método del QuestController.

```
public void DoInteraction()
{
    if (playerInside)
    {
        ShowNextDialog();
    }
}

private void ShowNextDialog()
{
    textComponent.text = quest.HasToShowNewDialog();
}
```

6.17.3 Scriptable Quest

Para poder crear las quest desde inspector, hemos desarrollado un sistema de ScriptableObject que funciona de la siguiente manera. Lo primero con lo que nos encontramos es con un enum, donde están todos los tipos de recursos posibles.

```
enum Resource
{
   diamond,
   gold,
   pepper,
   silver,
   tomatoe,
   vanilla,
   wood,
   stone
}
```

Cuenta con unos parámetros públicos a través de los cuales se puede establecer el nombre de la quest, la descripción del tipo de recurso (que es uno del enum) y la cantidad del mismo. Además tiene un parámetro para convertir a texto el nombre del recurso.

```
public string questName, questDescription;

[SerializeField] Resource resource;

public int count;

public string GetResourceType()
{
   return resource.ToString();
}
```

6.17.4 NPC de interacción

Este NPC cuenta con un capsule collider para detectar las colisiones, un box collider que funciona de trigger, y el script questInteractor. Esto lo podemos observar en la Figura 6.17.4.1.

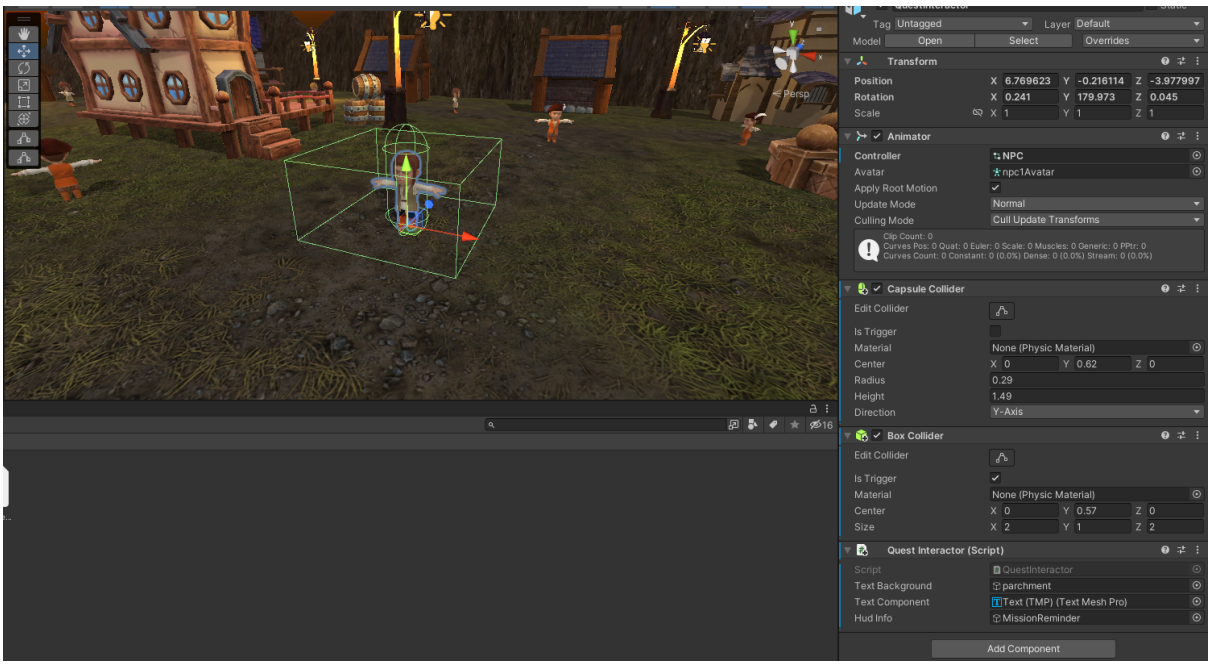


Figura 6.17.4.1: Captura de pantalla del NPC.

Además, este NPC también cuenta con un animator para representar dos estados, saludando y hablando. Es importante destacar que este NPC comparte animator con todos los demás por lo que, en la Figura 6.17.4.2 vamos a ver una distribución diferente que explicaremos más adelante.

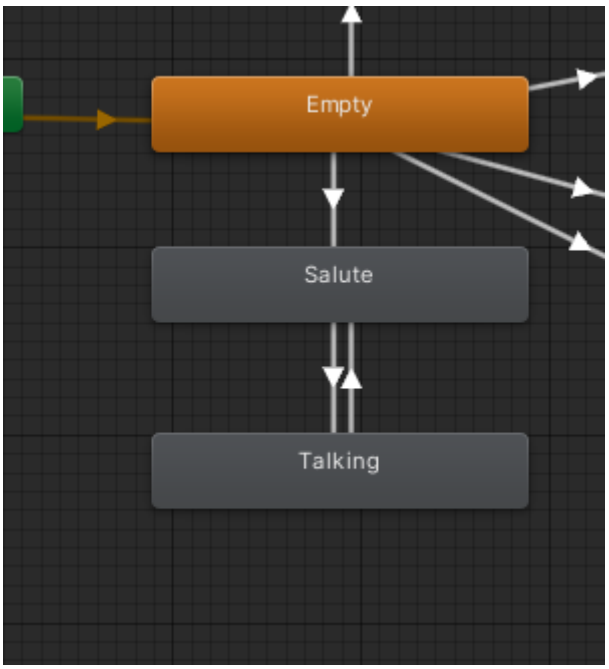


Figura 6.17.4.2: Captura de pantalla del animator.

6.18 Sistema de guías

A la hora de plantear el escenario, nos dimos cuenta de que podría ser un poco complicado encontrar la ruta a seguir para llegar a las ubicaciones necesarias según va

avanzando el jugador, por lo que hemos desarrollado un sistema de guías, ubicadas a la salida del pueblo.

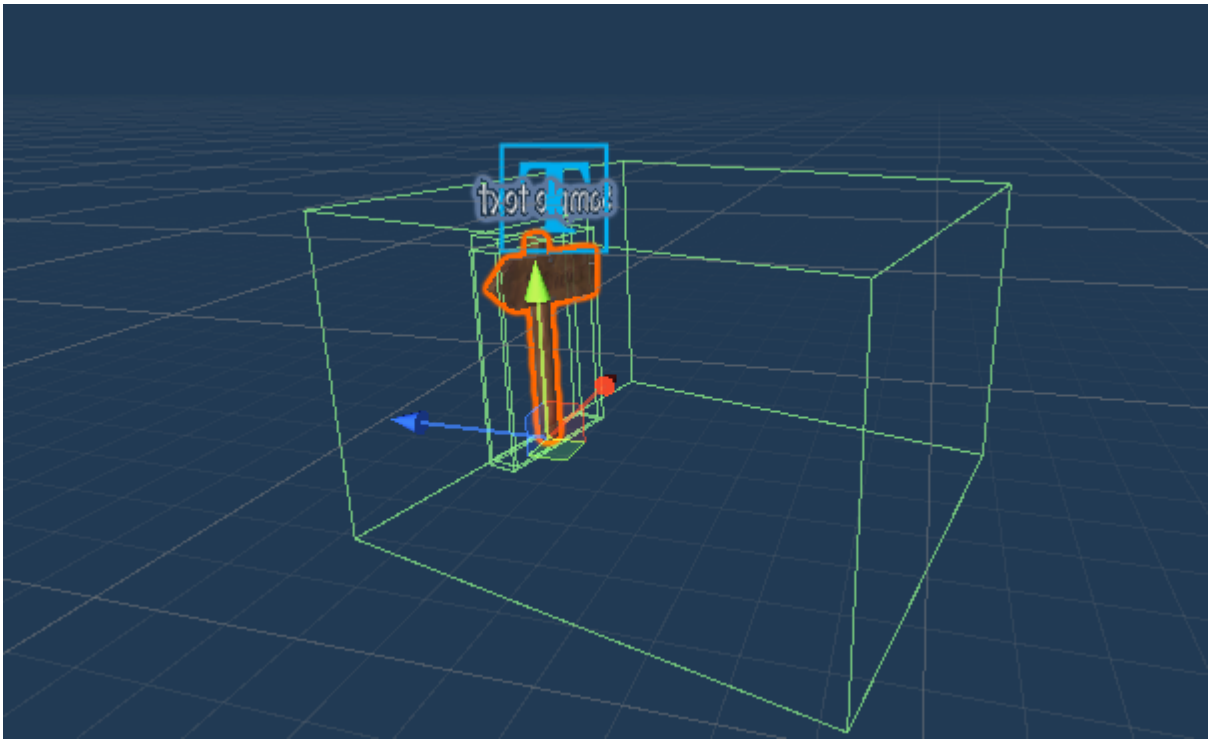


Figura 6.18.0.1: Captura de pantalla de la guía.

Como podemos ver en la figura anterior, contamos con un poste de madera, el cual tiene dos box colliders, donde el pequeño funciona como colisión y el grande como trigger para detectar cuando el jugador está dentro. Además, cuenta con el sistema que hemos visto anteriormente para orientar el texto mediante el script Look “At Camera”, y utiliza el componente navMesh Obstacle para evitar que las IAs se queden atrancadas con el. La dirección la marca la punta afilada de madera.

6.18.1 Label Collider Controller

Este script cuenta con dos métodos importantes, los cuales se encargan de detectar si el jugador entra o sale del collider y activar o desactivar el texto respectivamente.

```
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            textInfo.SetActive(true);
        }
    }

    private void OnTriggerExit(Collider other)
{
        if (other.gameObject.CompareTag("Player"))
        {
            textInfo.SetActive(false);
        }
    }
```

6.19 NPCs

Para conseguir un poco más de realismo, hemos implementado una variedad de tres NPC que se encuentran en el pueblo, con los cuales no puede interactuar el jugador.
Cada NPC comparte esqueleto, animaciones y animator, por lo que facilita el trabajo, ya que nos evitamos tener que hacer lo mismo tres veces.

Cada NPC cuenta con los mismos componentes donde nos encontramos el animator, para poder controlar las animaciones, un capsule collider para poder controlar las colisiones con el jugador, un rigidbody con el cual podemos desplazar el NPC, el Nav Mesh Agent para poder controlar el desplazamiento del mismo y el script “NPC Controller” para poder gestionar el movimiento

6.19.1 Animator

En cuanto a la hora de ver el comportamiento de los NPCs, nos parecía un poco aburrido que todos tuvieran la misma animación, por lo que adaptamos el animator para que cada uno tuviera un comportamiento aleatorio al inicio de cada partida.
En la lógica del animator nos encontramos con un estado inicial vacío, del cual parten cuatro estados de comportamiento que son:

- Normal
- Feliz
- Enfadado
- Triste

Estos cuatro estados cuentan con su propia animación de idle y caminar, los cuales se asignan a cada npc en el momento de empezar la partida mediante script. Una vez que el jugador ya tiene asignado un tipo de comportamiento de los anteriores, no puede cambiar de comportamiento a no ser que se active/desactive el gameobject, cosa que

pasa cada vez que el jugador sale del poblado, pero no resulta molesto ya que no es un cambio visible. Todo esto lo podemos ver en la Figura 6.19.1.1.

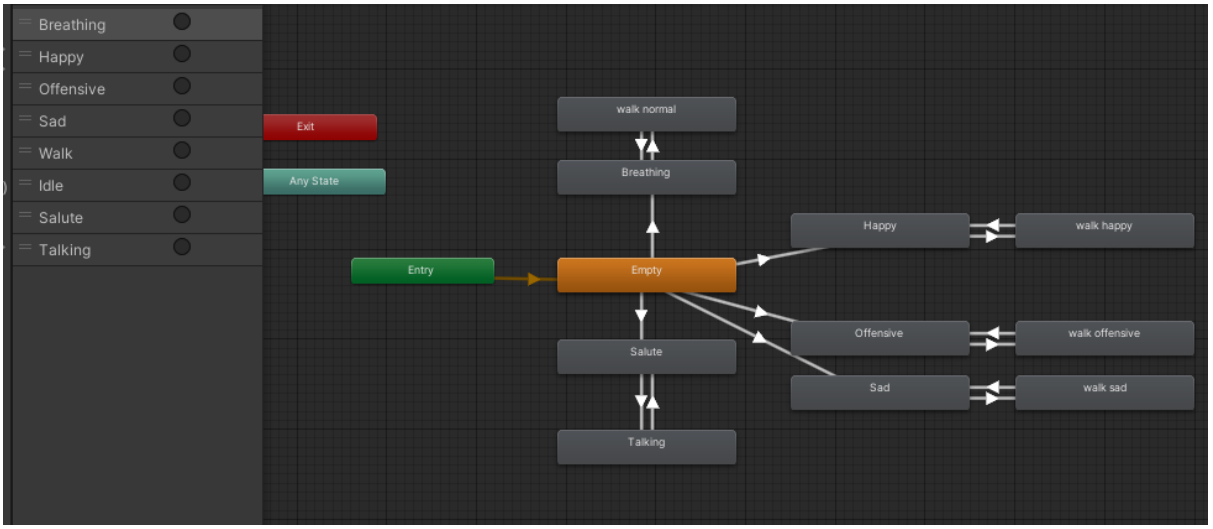


Figura 6.19.0.1: Captura de pantalla del animator de los NPC.

6.19.2 NPC Controller

Este script es el encargado de gestionar todo el comportamiento de los NPCs, desde el movimiento hasta el tipo de comportamiento para las animaciones. Para decidir el tipo de comportamiento contamos con un enum que los almacena y facilita su elección.

```
enum Behaviour
{
    Breathing,
    Happy,
    Offensive,
    Sad
}
```

- **TakeBehaviour()**: este método se encarga de obtener un número aleatorio del 0 al 4, y en función del número que salga tendrá un comportamiento u otro y se asigna en la variable behaviour.

```
private void TakeBehaviour()
{
    int value = Random.Range(0, 4);

    switch (value)
    {
        case 0:
            behaviour = Behaviour.Breathing;
            break;
        case 1:
            behaviour = Behaviour.Happy;
            break;
        case 2:
            behaviour = Behaviour.Offensive;
            break;
        case 3:
            behaviour = Behaviour.Sad;
            break;
    }
}
```

- **Awake()**: este método se encarga de adjudicar el comportamiento inicial y guardarse todas las referencias necesarias para el buen funcionamiento del script.

```
private void Awake()
{
    TakeBehaviour();
    animator = GetComponent<Animator>();
    navAgent = GetComponent<NavMeshAgent>();
    targetPosition = transform.position;
}
```

- **SetInitialIdleState()**: a partir del contenido de la variable behaviour, se establece el estado de Idle correspondiente a cada estado anímico.

```
private void SetInitialIdleState()
{
    if (behaviour == Behaviour.Breathing)
    {
        animator.SetTrigger("Breathing");
    }
    else if (behaviour == Behaviour.Happy)
    {
        animator.SetTrigger("Happy");
    }
    else if (behaviour == Behaviour.Offensive)
    {
        animator.SetTrigger("Offensive");
    }
    else
    {
        animator.SetTrigger("Sad");
    }
}
```

- **OnEnable()**: este método se llama a la hora de activar de nuevo el gameObject, y se encarga llamar al método SetInitialIdleState().

```
private void OnEnable()
{
    SetInitialIdleState();
}
```

- **Update()**: este método es el corazón central del script, ya que se encarga de gestionar el comportamiento de los NPC. Lo primero que realiza es comprobar si el NPC ha llegado a su destino, y en caso de que ya se haya acabado el tiempo de espera, llama a una corrutina que se encarga de buscar una nueva posición y realizar el desplazamiento. En caso de que el NPC tarde más de lo permitido en llegar a esa posición (se estima que se ha quedado atrancado con algún edificio o que es imposible llegar a esa ubicación), se llama a una corrutina que se encarga de realizar un desplazamiento en la dirección contraria para que se pueda mover el NPC. En caso de que el NPC no esté en su destino, se actualiza el contador encargado de controlar cuando es necesario realizar el paso descrito anteriormente.

```
private void Update()
{

   if (IsInPosition(wayPointTolerance) && !isWaiting)
   {
      StartCoroutine(SearchNewPosition());
      timeArrivingThisPosition = 0f;
   }

   if(timeArrivingThisPosition >= maxTimeToSearchPosition)
   {
      StartCoroutine(SerInversePosition());
      timeArrivingThisPosition = 0f;
   }

   if (!isWaiting)
   {
      timeArrivingThisPosition += Time.deltaTime;
   }

}
```

- **IsInPosition()**: este método recibe como parámetro la tolerancia admitida a la hora de calcular la distancia que hay entre el NPC y la posición objetivo. Regresa una comparación que es cierta si la distancia es menor que la tolerancia.

```
private bool IsInPosition( float wayPointTolerance)
{
   float distance = Vector3.Distance(targetPosition, transform.position);
   return distance <= wayPointTolerance;
}
```

- **GetPositionToSearch()**: este método recibe como parámetro una posición y una tolerancia, con los cuales se encarga de obtener un valor aleatorio comprendido entre ellos.

```
private float GetPositionToSearch(float position, float tolerance)
{
   return Random.Range(position - tolerance, position + tolerance);
}
```

- **SearchNewPosition()**: esta corrutina se encarga de buscar la nueva posición. Para hacer esto se encarga de establecer el booleano de control a cierto, en caso de

que nos encontremos en la primera vez de la búsqueda de posiciones, no nos interesa que se ponga a cierto el trigger "Idle", ya que es su estado inicial. Después obligamos al NPC a esperar 7 segundos en el sitio. Cuando se acabe esta cuenta atrás buscará posiciones aleatorias a partir de su posición actual hasta que encuentre una posición navegable, se actualiza el animator al trigger de caminar y se establece la variable de control a falsa.

```
private IEnumerator SearchNewPosition()
{
    isWaiting = true;

    if (!firstLoop)
    {
        animator.SetTrigger("Idle");
    }
    firstLoop = false;
    yield return new WaitForSeconds(7);

    do
    {
        float x = GetPositionToSearch(transform.position.x ,  searchDistance);
        float z = GetPositionToSearch(transform.position.z, searchDistance);
        targetPosition = new Vector3(x, transform.position.y, z);
    } while (!navAgent.SetDestination(targetPosition));
    animator.SetTrigger("Walk");

    isWaiting = false;
    yield return null;
}
```

- **SetInversePosition()**: esta corrutina se encarga de obtener un valor aleatorio entre el -1 y el 1 y se suma a la posición actual del NPC. Una vez hecho esto, se realiza la navegación hacia esta dirección.

```
private IEnumerator SerInversePosition()
{

    float dir = Random.Range(-1, 1);
    targetPosition = new Vector3(transform.position.x + dir, targetPosition.y,
transform.position.z + dir);
    navAgent.SetDestination(targetPosition);
    yield return null;
}
```

Figura 6.19.2.1: Captura de pantalla del animator de los NPC.

Como podemos observar en la figura anterior, este es el resultado de los NPC.

6.20 Enemigo

Otro pilar fundamental para el desarrollo de este proyecto es el enemigo, ya que va a ser el principal reto con el cual se va a enfrentar el jugador. Una característica necesaria para que el jugador pueda detectar fácilmente si es un enemigo es que este tenga una mal aspecto.

Para el prefab del enemigo contamos con unos componentes muy necesarios que son un capsule Collider para calcular las distintas colisiones, animator para reproducir todas los estados de animación con los que cuenta este personaje, un Nav Mesh Agent que nos permite realizar todo el desplazamiento del personaje. Además, también cuenta con el script Loot Bag que permite al personaje guardar unas recompensas, las cuales puede soltar cuando muera, el Script Destructible Elements, que permite decidir con qué elemento se puede interactuar para destruirlo y en este caso es la espada, y el script Radial Bar General Controller, que sirve para mantener el contador de la vida. También tiene el script Compute Movement, que nos permite trazar todas las trayectorias que seguirá el personaje para su desplazamiento y el script AI Controller el cual es el cerebro principal de este componente.

6.20.1 Animator

Este componente se encarga de mantener todos los estados de la animación y realizar los cambios entre unos y otros en el momento adecuado.

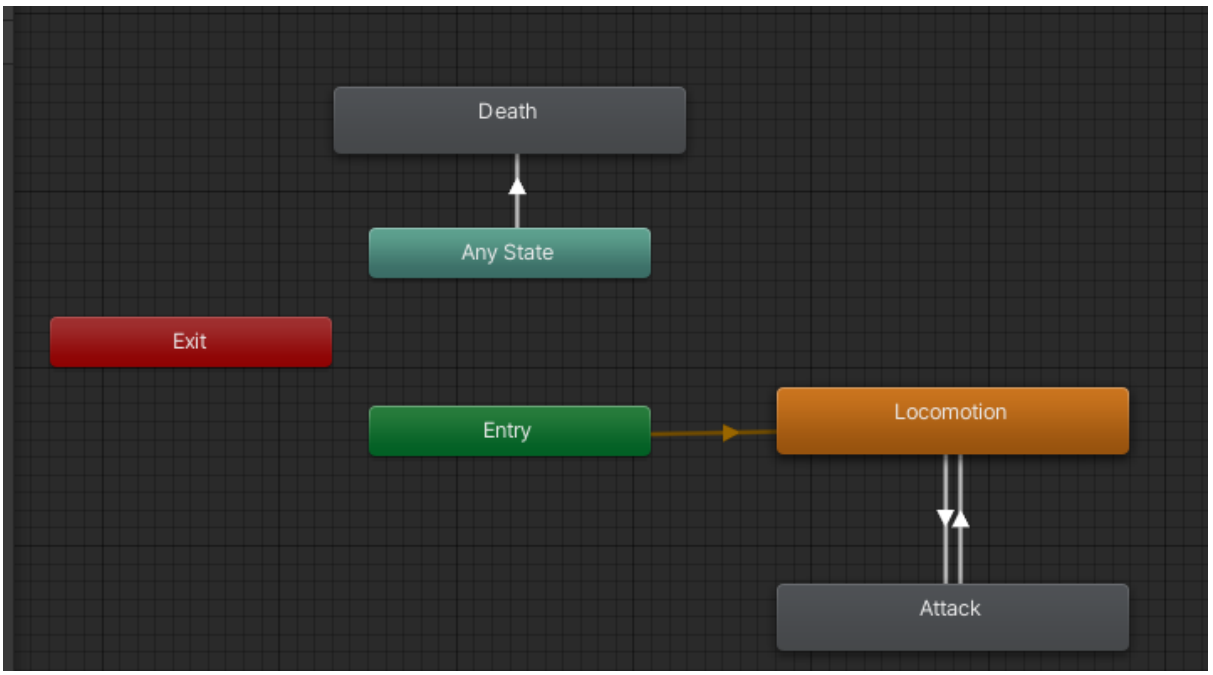


Figura 6.20.1.1: Captura de pantalla del animator del enemigo

Como podemos observar en la figura anterior, tenemos 3 estados iniciales. Por un lado nos encontramos con el estado de muerte, el cual se puede activar desde cualquier estado de ejecución, ya que está conectado al estado inicial Any State y se hace mediante el trigger de control llamado "Death". Por otro lado nos encontramos con los estados Locomotion y Attack, donde el jugador se encuentra siempre en Locomotion y cuando se activa el trigger Attack, se cambia de estado y realiza el ataque; cuando acaba el estado, regresa directamente al estado de Locomotion. El estado de locomotion cuenta con un Blend Tree donde contamos con tres animaciones, las cuales son Idle, Walking y Run, y se va pasando de un estado a otro en función de la velocidad de desplazamiento del enemigo. Esto se puede observar en la Figura 6.20.1.2.

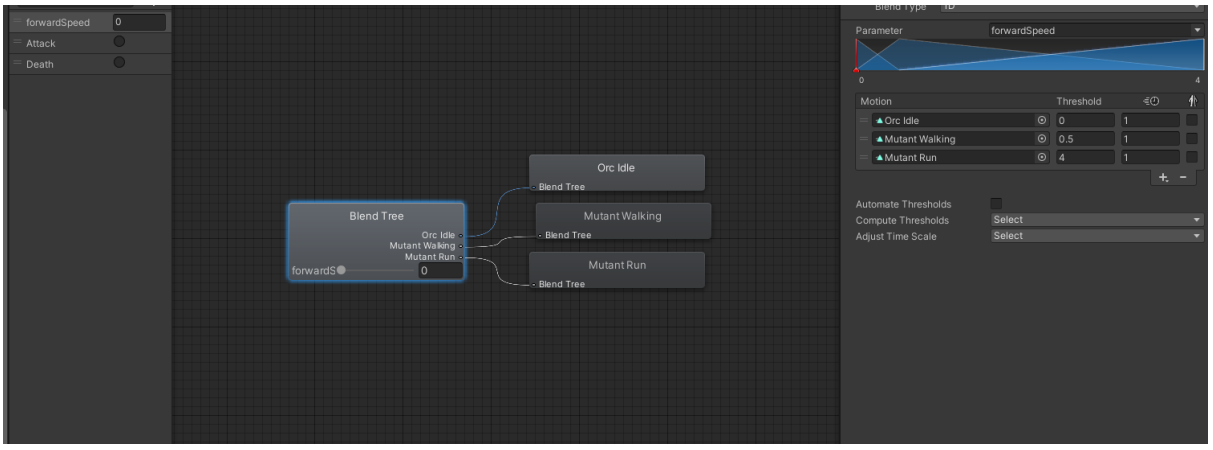


Figura 6.20.1.: Captura de pantalla del estado Locomotion

6.20.1.1 Check End Attack

Retomando el tema del estado de Atacar, este estado cuenta con un script propio llamado CheckEndAttack, el cual implementa el método "**OnStateExit**", donde obtiene el

Script AIController para hacer una llamada al método "**AttackStateFinished**" el cual sirve para avisarlo de que el estado de ataque ha acabado y el script puede seguir con su funcionamiento normal.

```
override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo,
int layerIndex)
{
    animator.GetComponent<AIController>().AttackStateFinished();
}
```

6.20.2 Compute Movement

Este método es el encargado de realizar todo el cálculo del desplazamiento de una ubicación a otra con una velocidad determinada y pasársela al animator para que reproduzca la animación correspondiente.

- **MoveTo()**: este método se encarga de establecer el destino en el NavMeshAgent, asignarle la velocidad correspondiente y establecer a falso el booleano de control de "isStopped". La velocidad se establece a partir de una fracción, ya que se mantiene en una variable la velocidad máxima a la que puede ir el enemigo y mediante un valor (como puede ser 0.5) se multiplica por esta velocidad máxima para obtener el valor correspondiente.

```
private void MoveTo(Vector3 destination, float speedFraction)
{
    navMeshAgent.destination = destination;
    navMeshAgent.speed = maxSpeed * Mathf.Clamp01(speedFraction);
    navMeshAgent.isStopped = false;
}
```

- **StartMoveAction()**: este método público recibe como parámetro el destino y la fracción de velocidad, y llama al método privado "MoveTo()" al cual le pasa esta información.

```
public void StartMoveAction(Vector3 destination, float speedFraction)
{
    MoveTo(destination, speedFraction);
}
```

- **Cancel()**: este método se encarga de cancelar la acción de desplazamiento, estableciendo a cierto el booleano que se encarga de controlar esto, llamado isStopped.

```
public void Cancel()
{
    navMeshAgent.isStopped = true;
}
```

- **Update()**: este método se encarga de obtener la información de la velocidad de desplazamiento local y pasárselo al animator a través de la variable float "foorwardSpeed".

```
private void Update()
{
    Vector3 velocity = navMeshAgent.velocity;
    Vector3 localVelocity = transform.InverseTransformDirection(velocity);
    float speed = localVelocity.z;
    animator.SetFloat("forwardSpeed", speed);
}
```

6.20.3 Sistema de patrulla

Para poder indicarle al enemigo el camino a seguir necesitamos un componente que se llama Patrol Path. Este gameObject cuenta con unos determinados hijos, los cuales indican el camino a seguir por parte del enemigo que tenga asociado este patrol path. Este gameObject cuenta con un script llamado "Patrol Path", el cual analizaremos a continuación.

6.20.3.1 Patrol Path

Este script cuenta con tres métodos muy sencillos para controlar su funcionamiento.

- **GetNextIndex()**: este método recibe como parámetro el índice actual, al cual tenemos que sumarle 1 para ir al siguiente hijo, pero en caso de que se supere el número hijos disponibles, se regresa el hijo número 0 como siguiente hijo.

```
public int GetNextIndex(int i)
{
    return i + 1 < transform.childCount ? i + 1 : 0;
}
```

- **GetWaypoint()**: este método recibe como parámetro el número del hijo que queremos visitar y nos regresa su posición.

```
public Vector3 GetWaypoint(int id)
{
    return transform.GetChild(id).transform.position;
}
```

El otro método con el que contamos se llama OnDrawGizmos, el cual simplemente se utiliza como medio de debug para poder visualizar constantemente la posición de estos puntos constantemente.

6.20.4 AI Controller

Este es el script que se encarga de controlar todo el funcionamiento del enemigo y gestionar su lógica correspondiente.

Para poder entender cómo funciona este script, vamos empezar por explicar el método update, el cual se puede considerar como el cerebro central.

- **Update()**: este método está dividido en distintas secciones y en todas se llama al método llamado CanChase(), que se encarga de comprobar si el jugador está con vida. La primera parte se encarga de controlar si el enemigo sigue con vida y en caso de no ser así, bloquea el funcionamiento del enemigo. La siguiente parte se encarga de controlar si el enemigo está demasiado cerca del jugador, y en caso de no estar atacando, cancela el movimiento actual y llama al método encargado de escapar. La siguiente parte se encarga de controlar si el jugador se encuentra dentro del rango de ataque, cancela la acción de movimiento y llama al método de atacar. También se encarga de controlar si el jugador está en rango de perseguir y el enemigo no se encuentra atacando, empieza a perseguirlo. Como última región, si el jugador no está atacando, se encarga de realizar el sistema de patrulla. Además, siempre se incrementa el tiempo que ha pasado desde que llegó al punto de destino de la patrulla para que no pierda el tiempo transcurrido.

```
private void Update()
{

    if(health.Empty() ) { return; }


    float distance = DistanceToPlayer();
    //Par?metro de escapar
     if (distance < 2f && CanChase() && !isAttacking)
     {
        movement.Cancel();
        ScapeFromPlayer();

     }

     //Par?metro de atacar

     else if(distance < attackRange && CanChase() )
     {
        movement.Cancel();
        Attack();
     }

     //Par?metro de perseguir
     else if(distance < chaseRange && CanChase() && !isAttacking)
     {
        ChasePlayer(distance);
    }

    //Par?metro de patrullar
    else if(!isAttacking)
    {
        PatrolBehaviour();
    }

    timeSinceArriveAtWaypoint += Time.deltaTime;

}
```

- **DistanceToPlayer()**: este método se encarga de calcular el valor de la distancia y de regresarlo como parámetro. En caso de que la referencia al jugador sea nula, regresa infinito y si no lo es, calcula la distancia entre los vectores de posición del enemigo y del jugador.

```
private float DistanceToPlayer()
{
    float distance;
    if (player == null)
    {
        distance = Mathf.Infinity;
    }
    else
    {
        distance = Vector3.Distance(player.transform.position, transform.position);
    }
    return distance;
}
```

- **CanChase()**: este método comprueba si el jugador está vivo a través del método que permite obtener la salud del mismo.

```
private bool CanChase()
{
    if (player == null) return false;

    return player.GetComponent<PlayerMovment>().GetCurrentHealth() > 0f;
}
```

- **ScapeFromPlayer()**: este método se encarga de calcular el vector director entre el jugador y el enemigo y normalizarlo. Le cambia la dirección para poder escapar del jugador y le suma la posición del enemigo para poder tener una posición. Una vez hecho esto, llama al método que se encarga de realizar la acción de desplazamiento.

```
private void ScapeFromPlayer()
{
    Vector3 direction = (player.transform.position - transform.position).normalized;
    direction = -direction;
    direction = transform.position + direction;
    movement.StartMoveAction(direction, 1);

}
```

- **Attack()**: en caso de que el enemigo no esté atacando, se pone a cierto el booleano de control que se encarga de controlar el ataque, se orienta el enemigo al jugador y se llama al trigger que se encarga de realizar la animación de atacar.

```
private void Attack()
{
    if (!isAttacking)
    {
        isAttacking = true;
        transform.LookAt(player.transform.position);
        animator.SetTrigger("Attack");
    }
}
```

- **ChasePlayer()**: este método se encarga de desplazar el enemigo hasta la posición del jugador. Hay un par de parámetros más que ahora mismo no se utilizan, pero servirán en otras iteraciones del proyecto.

```
private void ChasePlayer(float distance)
{
    seenPlayer = true;
    actuaSuspiciouslIterations = 0;

    movement.StartMoveAction(player.transform.position, 1f);
}
```

A partir de ahora vamos a explicar el funcionamiento del sistema de patrulla, por lo que primero explicaremos los métodos base del funcionamiento y después el método que los agrupa y gestiona todo el funcionamiento.

- **IsInPosition()**: este método se encarga de calcular la distancia entre la posición actual y la posición objetivo; si esta distancia es menor a una tolerancia permitida se regresa un booleano cierto.

```
private bool IsInPosition(Vector3 targetPosition, float tolerance)
{
    float distance = Vector3.Distance(targetPosition, transform.position);
    return distance <= tolerance;
}
```

- **GetCurrentWaypoint()**: este método se encarga de obtener la posición del punto que tiene que visitar el enemigo.

```
private Vector3 GetCurrentWaypoint()
{
    return patrolPath.GetWaypoint(currentWaypointIndex);
}
```

- **AtWaypoint()**: este método se encarga de calcular si el enemigo se encuentra en el punto de patrulla teniendo en cuenta una tolerancia.

```
private bool AtWaypoint()
{
    return IsInPosition(GetCurrentWaypoint(), waypointTolerance);
}
```

- **CicleWaypoint()**: este método se encarga de de obtener el número del siguiente punto a visitar y lo almacena en la variable "currentWaypointIndex".

```
private void CicleWaypoint()
{
    currentWaypointIndex = patrolPath.GetNextIndex(currentWaypointIndex);
}
```

- **PatrolBehaviour()**: este método se guarda como próxima posición la posición actual para evitar posibles problemas. En caso de que el patrol path no sea nulo, se comprueba si el jugador ya está en la posición y en caso de estarlo, se cambia de posición objetivo y se reinicia el contador del tiempo en la posición actual. Se establece como posición objetivo la posición del nuevo punto que hemos establecido anteriormente, y en caso de que el tiempo de espera sea superior al esperado se llama al método que se encarga de realizar el desplazamiento del enemigo pasándole como velocidad de desplazamiento la utilizada para patrullar, que se establece por inspector.

```
private void PatrolBehaviour()
{
    Vector3 nextPosition = guardPosition;

    if(patrolPath != null)
    {
        if (AtWaypoint())
        {
            timeSinceArriveAtWaypoint = 0;
            CicleWaypoint();
        }
        nextPosition = GetCurrentWaypoint();
    }
    if(timeSinceArriveAtWaypoint > waypointDwellTime)
    {
        movement.StartMoveAction(nextPosition, patrolSpeedFraction);
    }
}
```

A partir de ahora vamos a explicar el funcionamiento del sistema de daño del enemigo, el funcionamiento de cómo reaparece el enemigo y diversas funciones del script que sirven para darle más realismo a este personaje.

Para poder recibir daño, este script implementa la interfaz **"INteractableAssetsInterface"**, la cual nos obliga a completar el método llamado **"OnDamage"** donde se llama al método "**TakeDamage()**" el cual recibe por parámetro el daño realizado.

```
public void OnDamage()
{
    TakeDamage(50f);
}
```

- **TakeDamage()**: este método recibe como parámetro el daño recibido y llama al método encargado de reducir la vida. En caso de que el enemigo se quede sin vida, se llama al trigger "Death" para que se realice la animación de muerte, se desactivan los componentes de navMesh y el capsule collider. Además, se llama a la corrutina encargada de realizar el spawn de las recompensas, se llama al método encargado de realizar la reaparición del enemigo donde pasamos el tipo de gameobject necesario, el tiempo necesario a esperar para realizar esta acción y como último se le pasa el patrolPath del enemigo actual. Como últimos pasos se reproduce el sonido de muerte y se destruye el gameObject después de 5 segundos.

```
public void TakeDamage(float damage)
{
    health.DecreaseValue(damage);

    if (health.Empty())
    {
        animator.SetTrigger("Death");
        this.GetComponent<NavMeshAgent>().enabled = false;
        this.GetComponent<CapsuleCollider>().enabled = false;
        StartCoroutine(SpawnLoot());
        respawn.HasToRespawnObject(transform,
RespawnObjects.SpawnObject.Enemy, respawnTime, null, null, patrolPath);
        audioS.PlayOneShot(deathAudio);
        Destroy(gameObject, 5f);
    }
    else
    {
        audioS.PlayOneShot(damageAudio);
    }
}
```

- **SpawnLoot()**: este método se encarga de esperar 3 segundos y después instancia en la escena el recurso que corresponda.

```
private IEnumerator SpawnLoot()
{
    yield return new WaitForSeconds(3);
    GetComponent<LootBag>().InstantiateLoot(new Vector3(transform.position.x,
transform.position.y + 1, transform.position.z));
    yield return null;
}
```

- **SetPatrol()**: este método se encarga de recibir como parámetro la patrulla que tiene que utilizar el enemigo y se utiliza para cuando se hace la reaparición del gameobject.

```
public void SetPatrol(PatrolPath patrulla)
{
    patrolPath = patrulla;
}
```

Cuando el enemigo está realizando la animación de atacar, nos podemos encontrar con dos eventos, los cuales sirven para controlar cuando se tiene que mostrar la piedra que tiene el enemigo en la mano y cuando se tiene que instanciar la piedra que lanza el

enemigo. Es importante aclarar que este componente se explicará en el siguiente apartado, pero que para lo que necesitamos explicar aquí no es de vital importancia conocer su funcionamiento.

- Mostrar la piedra: Para mostrar la piedra tenemos un evento llamado "**InstantiateRock**", donde se activa el gameObject de la piedra, como se puede ver en la Figura 6.20.3.1.

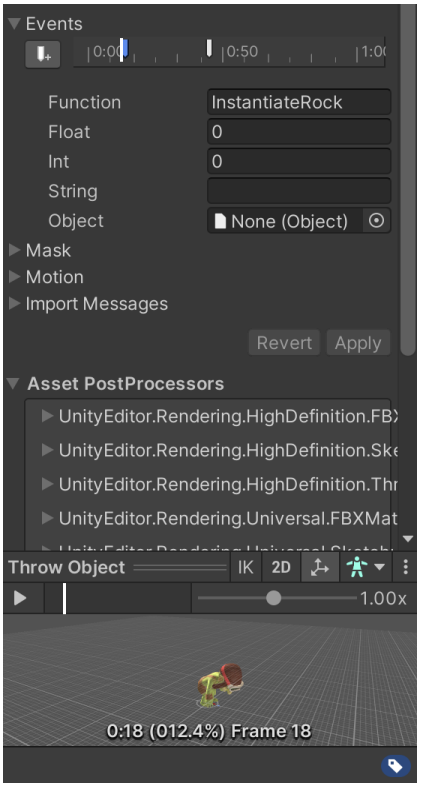


Figura 6.20.4.1.: Captura de pantalla del evento de instanciar una roca

```
private void InstantiateRock()
{
    visualRock.SetActive(true);
}
```

- Lanzar la piedra: para poder lanzar la piedra contamos con otro evento en esta animación, el cual llama al método "**ThrowRock**", que desactiva la piedra visual, instancia la piedra con las características necesarias para que funcione correctamente, añade un impulso a la misma y reproduce un sonido de lanzar. Esto lo podemos visualizar en la Figura 6.20.3.2.

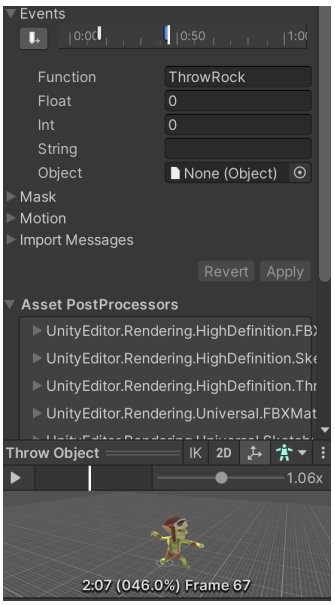


Figura 6.20.4.2.: Captura de pantalla del evento de lanzar la roca

```
private void ThrowRock()
{
   visualRock.SetActive(false);

   GameObject asset = Instantiate(rock, rockPosition.position, Quaternion.identity);
   Vector3 impulse = transform.forward;
   impulse.y += 0.2f;
   asset.GetComponent<Rigidbody>().AddForce(impulse * 100);
   audioS.PlayOneShot(throwRockAudio);
}
```

- **AttackStateFinished()**: este método sirve para desactivar la variable de control de cuando el enemigo se encuentra atacando para que este pueda seguir con su ciclo de funcionamiento.

```
public void AttackStateFinished()
{
   isAttacking = false;
}
```

El enemigo también cuenta con el sonido de los pasos implementados, los cuales se reproducen a través de los eventos de la animación, y se llama a los métodos encargados de buscar un clip aleatorio y reproducirlo. Los eventos se establecen un par de fotogramas antes de que se realice la acción de pisar, para que así los sonidos estén compenetrados.

Figura 6.20.4.4.: Captura de pantalla del evento de reproducir el sonido de caminar

```
public void WalkSound()
{
    audioS.PlayOneShot(GetRandomClipWalk()); ;
}

private AudioClip GetRandomClipWalk() => walkClips[UnityEngine.Random.Range(0,
walkClips.Length)];
```

En el caso de los sonidos, en la animación de correr nos encontramos con el mismo tipo de eventos.

6.20.5 Piedra del enemigo

El enemigo tiene una piedra con la cual puede hacer daño, y que analizaremos a continuación. Este gameObject cuenta con varios componentes los cuales son un

rigidbody para calcular las físicas, dos Box Colliders donde uno funciona como trigger y el otro sirve para realizar las colisiones, un AudioSource para reproducir el sonido de colisionar y por último, el script que se encarga de controlar las funciones de la roca.

6.20.5.1 Throwable Rock

- **OnTriggerEnter**(): este método se encarga de detectar contra qué ha colisionado la roca, donde en caso de que colisionara con algún elemento del escenario (que es el caso del primer if), reproduce el sonido de la colisión y llama al método CollideWithEnemy(), para que no pueda herir al jugador otra vez (ya que entendemos de que en caso de que la roca ya esté en el suelo, no tiene sentido que hiera al jugador); en caso de simplemente colisionar con el jugador se reproduce el sonido de colisión.

```
private void OnTriggerEnter(Collider other)
{
    if (!other.gameObject.CompareTag("Player") &&
!other.gameObject.CompareTag("GunInteractable") &&
!other.gameObject.CompareTag("Ignore"))
    {
        audioS.PlayOneShot(rockCollision);
        CollidedWithEnemy();
    }
    if (other.gameObject.CompareTag("Player"))
    {
        audioS.PlayOneShot(rockCollision);
    }

}
```

- **CollideWithEnemy**(): este método cambia la capa del objeto para que no pueda herir al jugador y destruye el gameObject. Este método es público ya que se llama desde el jugador.

```
public void CollidedWithEnemy()
{

    int LayerIgnoreRaycast = LayerMask.NameToLayer("Default");
    gameObject.layer = LayerIgnoreRaycast;

    Destroy(gameObject, timeToDestroy);
}
```

6.21 Implementación de los recursos

Para implementar el sistema de recursos, hemos optado por crear una estructura donde todos cuentan con los mismos componentes y lo explicaremos a continuación.

Todos los recursos cuentan con un gameObject padre, el cual cuenta con los componentes AudioSource para poder reproducir sonidos, el script ResourceReleased que se encarga de añadir un impulso vertical y reproducir el efecto del humo y por último tenemos el script PickObject que hemos explicado anteriormente y que se encarga de gestionar los efectos de que se recoja el recurso. Además, contamos con dos hijos, donde uno es el recurso como tal y el otro es el humo para el efecto visual.

Hablando del primer hijo, el cual es el gameobject del recurso en cuestión, nos encontramos con que tenemos dos boxColliders, donde uno funciona de trigger y el otro sirve para detectar colisiones, un rigidbody para poder aplicar físicas y el componente PlayerTriggeredObject, del cual hablaremos más adelante.

En cuanto a lo que el segundo objeto se refiere, este cuenta con el componente VisualEffect que se encarga de reproducir el objeto StylizedSmoke para que haya humo cada vez que se instancia un recurso en el escenario.

6.21.1 Player Triggered Object

Este script se encarga de detectar cuando entra un jugador en el trigger y llamar al método "**ObjectTriggered**" del script PickObject.

```
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.CompareTag("Player"))
    {
        gameObject.GetComponentInParent<PickObject>().ObjectTriggered();
    }
}
```

6.21.2 Recursos Implementados

En este apartado iremos mencionando los recursos que hemos implementado con los que puede interactuar el jugador. Además, es importante destacar que estos recursos los hemos modelado nosotros. Ver Figuras 6.21.2.1 a 6.21.2.8

Diamante

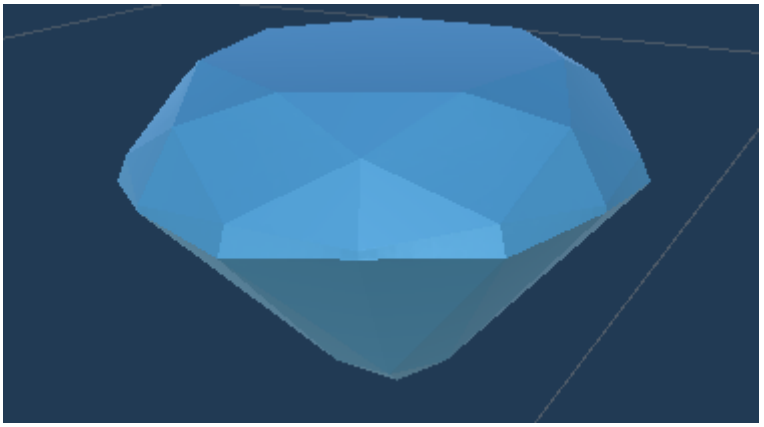


Figura 6.21.2.1.: Captura de pantalla del recurso diamante

Oro

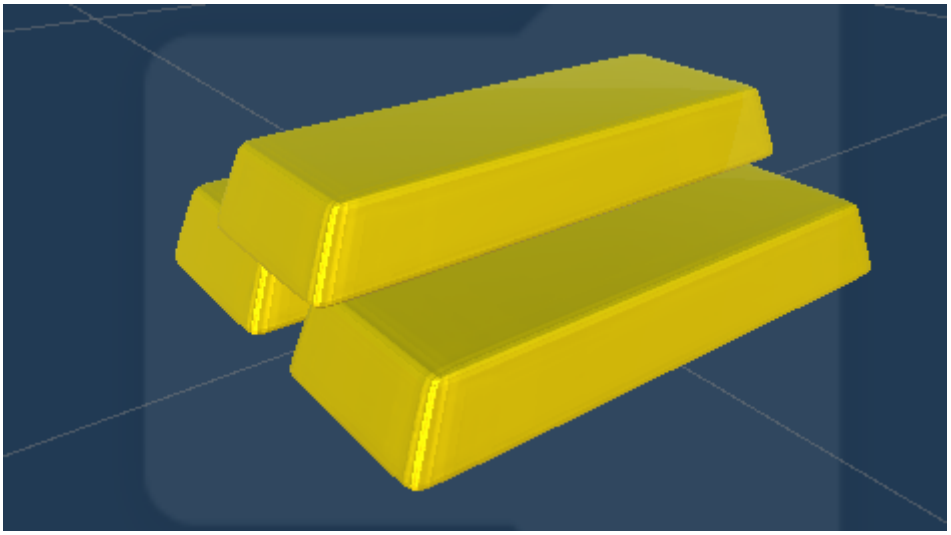


Figura 6.21.2.2.: Captura de pantalla del recurso oro

Madera

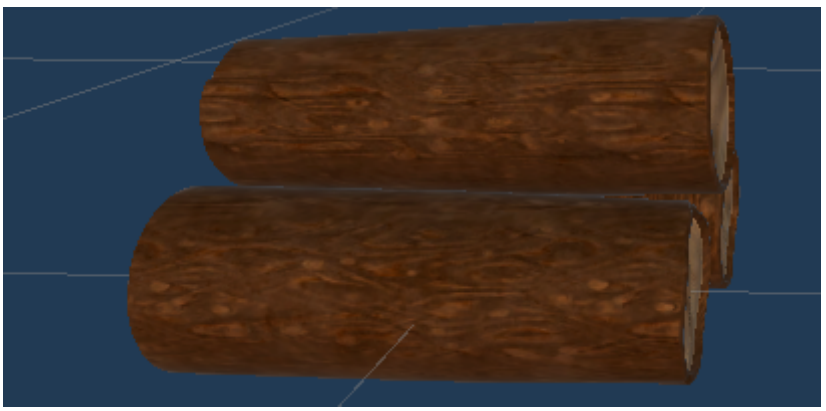


Figura 6.21.2.3.: Captura de pantalla del recurso madera

Pimiento

Figura 6.21.2.4.: Captura de pantalla del recurso pimiento

Piedra

Figura 6.21.2.5.: Captura de pantalla del recurso piedra

Hierro

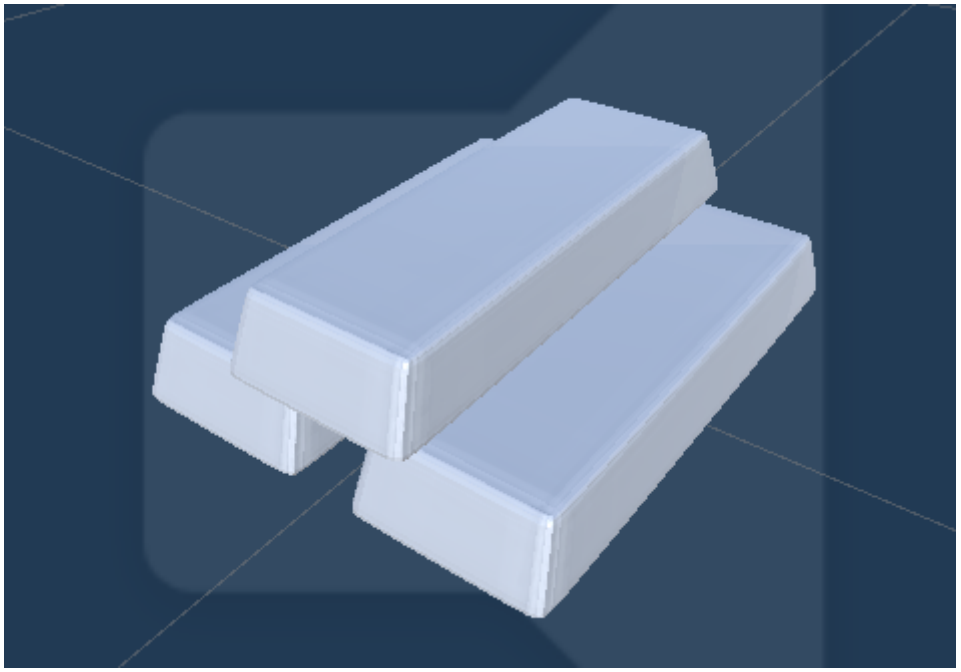


Figura 6.21.2.6.: Captura de pantalla del recurso hierro

Tomate

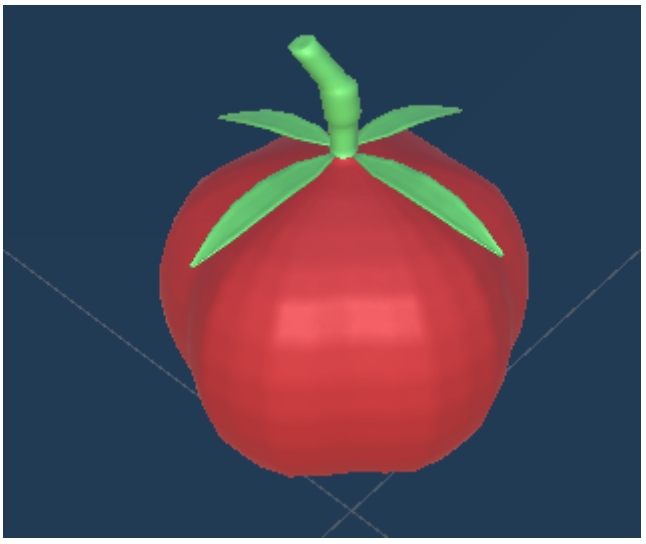


Figura 6.21.2.7.: Captura de pantalla del recurso tomate

Vainilla

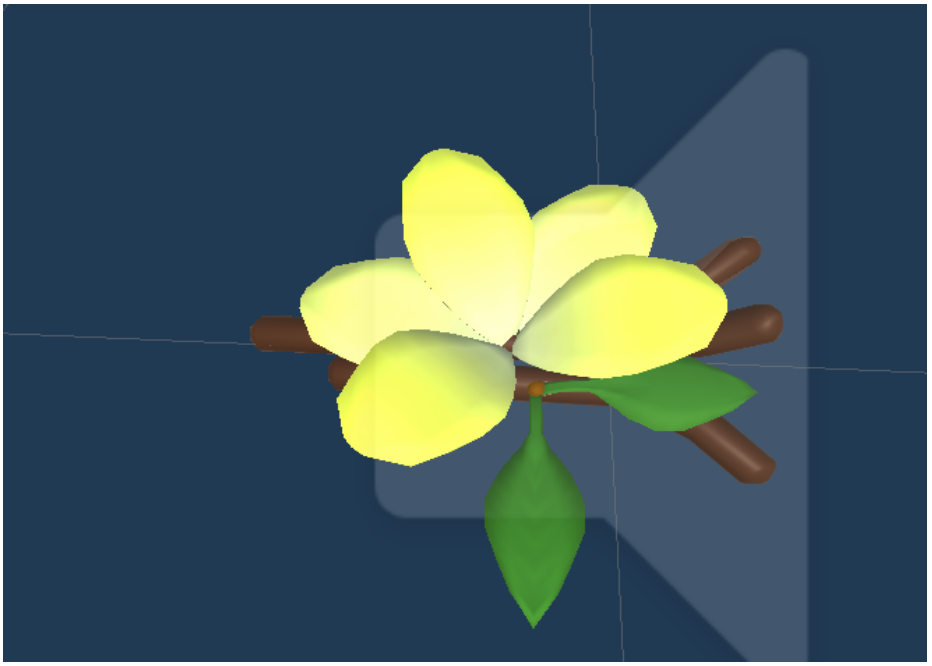


Figura 6.21.2.8.: Captura de pantalla del recurso vainilla

6.22 Shader de agua

Para poder crear el efecto de este shader, necesitamos un objeto al cual podemos aplicarlo y utilizarlo para ver los cambios que irá sufriendo. El primer paso que necesitamos es conocer la profundidad del agua en cada parte del plano y para ello creamos el efecto "**Depth Fade**" el cual es un subshader.

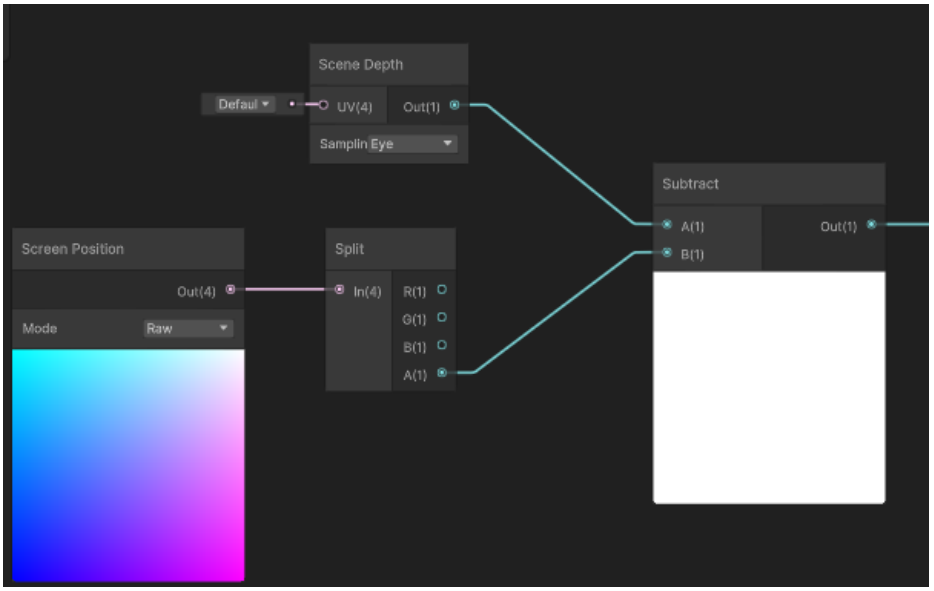


Figura 6.22.1.: Captura de pantalla del shader Depth Fade

En la Figura 6.22.1 podemos observar tres nodos, donde el nodo Scene Depth que nos permite conocer la distancia a la que se encuentra un objeto de la cámara, mientras que el nodo Screen Position nos devuelve la posición de un objeto en función de la posición de la cámara. Una vez que tenemos esta información, necesitamos restar estas distancias, por lo que utilizamos el nodo Subtract, pero en caso de la Screen Position solo nos interesa el valor alpha y utilizamos el método split para obtener ese valor.

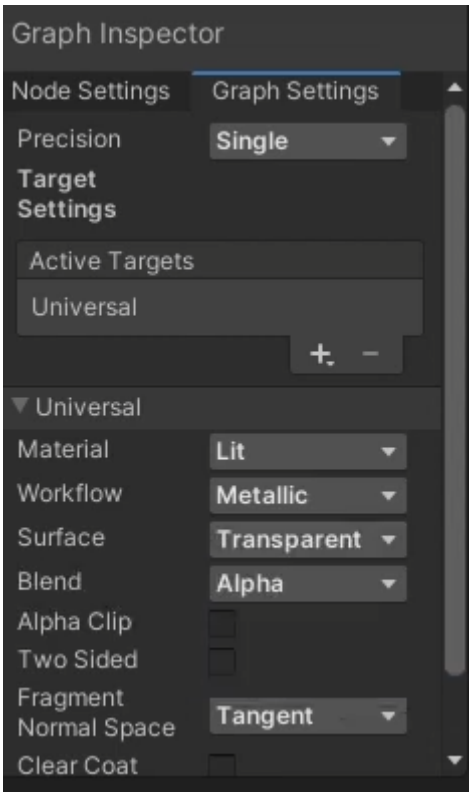


Figura 6.22.2.: Captura de pantalla de los ajustes del shader

Como vemos en la Figura 6.22.2, necesitamos cambiar los ajustes del shader para que sea capaz de demostrar la transparencia del mismo.

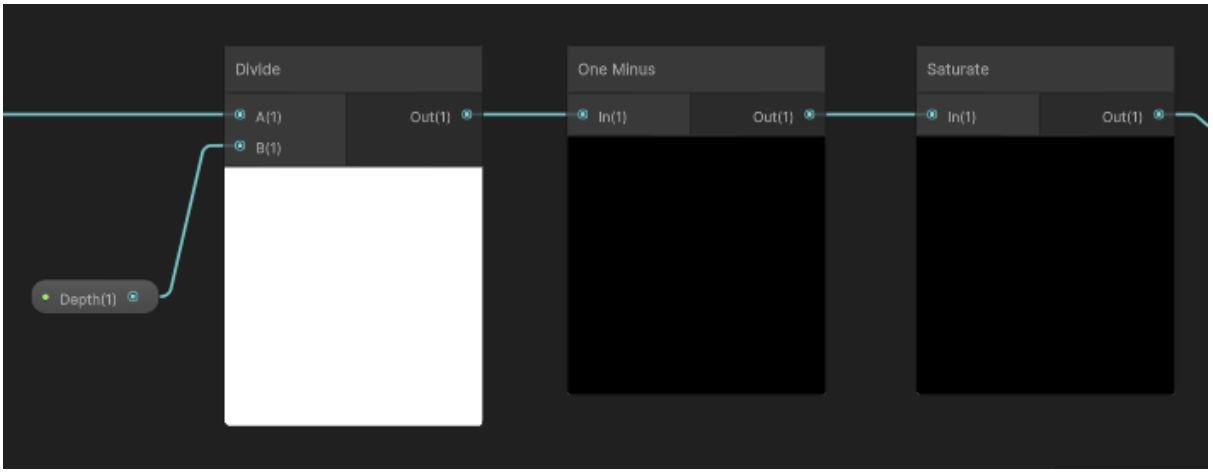


Figura 6.22.3.: Captura de pantalla de los nodos intermedios

El resultado del nodo Subtract lo pasamos al nodo Divide, el cual sirve para controlar el lugar donde ocurre el degradado entre las zonas negras y blancas, que lo hacemos mediante la variable Depth. El nodo One Minus permite invertir el resultado del nodo anterior, para que de esta forma la parte transparente se encuentre en las orillas y no por el centro del agua. Ahora tenemos que añadir el nodo Saturate, que permite delimitar los valores entre 1 y 0.

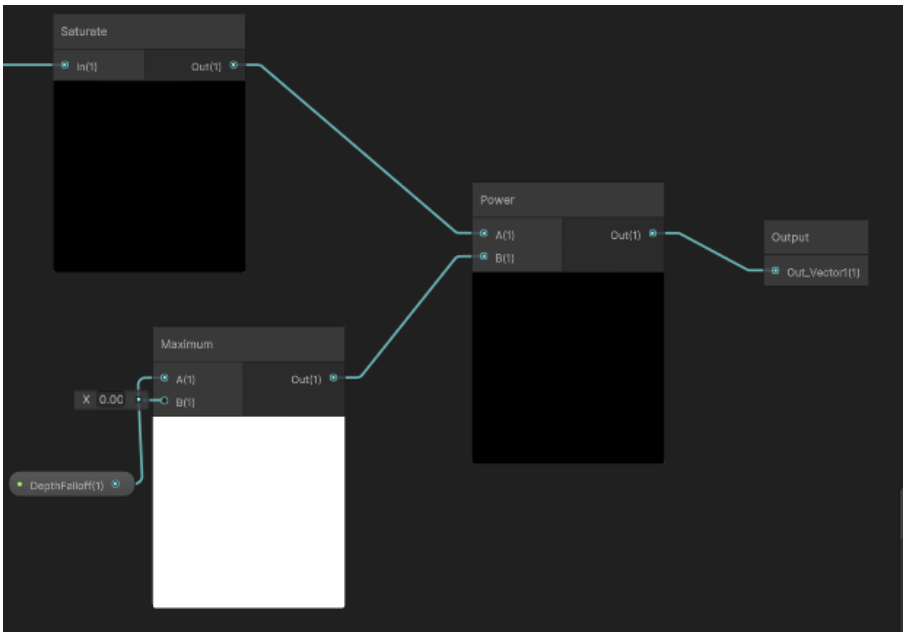


Figura 6.22.4.: Captura de pantalla de los nodos finales

Para poder hacerlo más realista, necesitamos poder controlar el degradado en sí, por lo que añadimos el nodo potencia, donde la base es el resultado de las operaciones y la potencia es la variable “DepthFalloff”, pero si la variable fuera 0 o negativa, tendríamos un serio problema por lo que necesitamos un nodo llamado Maximum donde un valor es la variable anterior y otro es 0.001. Esto se puede ver en la Figura 6.22.4.

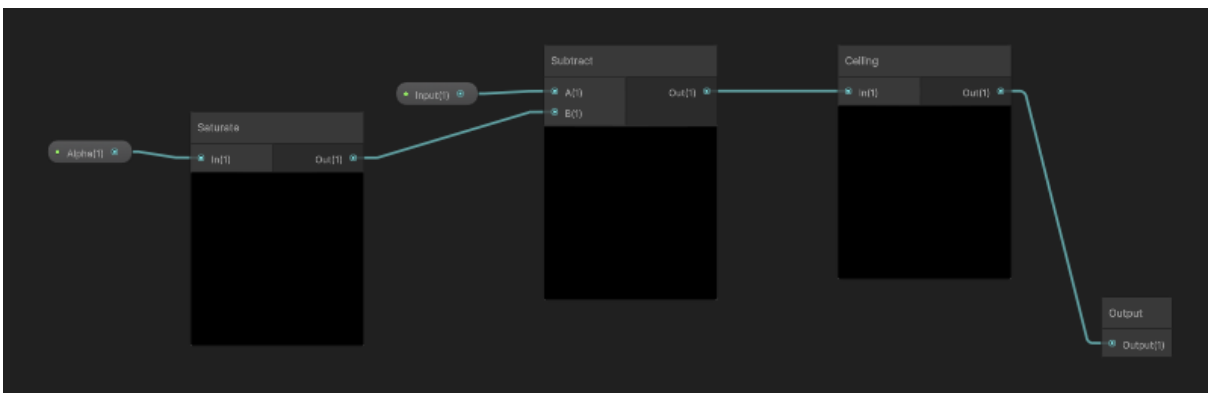


Figura 6.22.5.: Captura de pantalla de del subshader CutOut

Ahora hablaremos del subshader "**Cutout**" el cual recibe como parámetro un valor llamado Alpha, y le aplicamos el nodo Saturate para obtener valores comprendidos entre 1 y 0. Este valor se lo restamos a la variable input mediante el nodo subtract. Como último paso este valor lo añadimos a la variable Ceiling, que se encarga de realizar un redondeo a la alta.

A partir de ahora hablaremos del Shader "shaderAgua" que utiliza los subshader mencionados anteriormente.

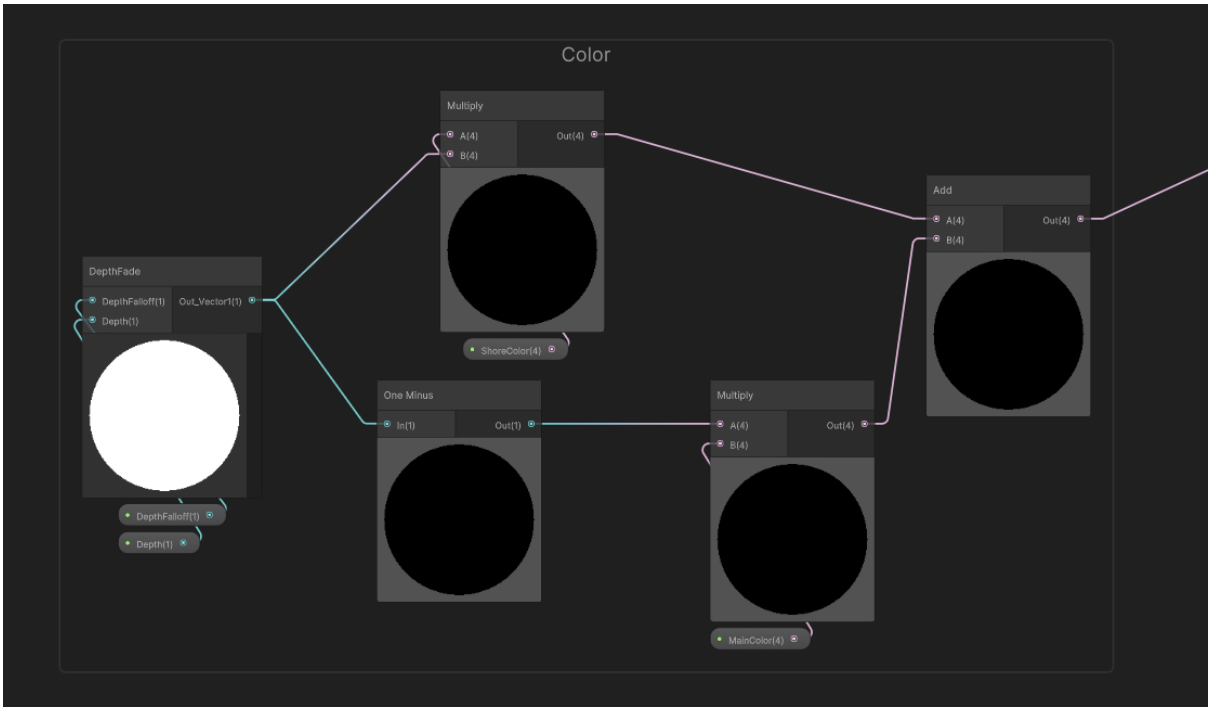


Figura 6.22.6.: Captura de pantalla de aplicar color al agua

Como podemos ver en la figura anterior, necesitamos añadir color para que no se vea en blanco y nego, por lo que para consegir el color de la orilla utilizamos directamente el DepthFade y lo multiplicamos por la variable del color, pero para el color del mar necesitamos hacer la inversa del DepthFade y multiplicarlo por la variable de color correspondiente. Como parte final nos encontramos con que necesitamos unir ambos valores por lo que utilizamos el nodo Add.

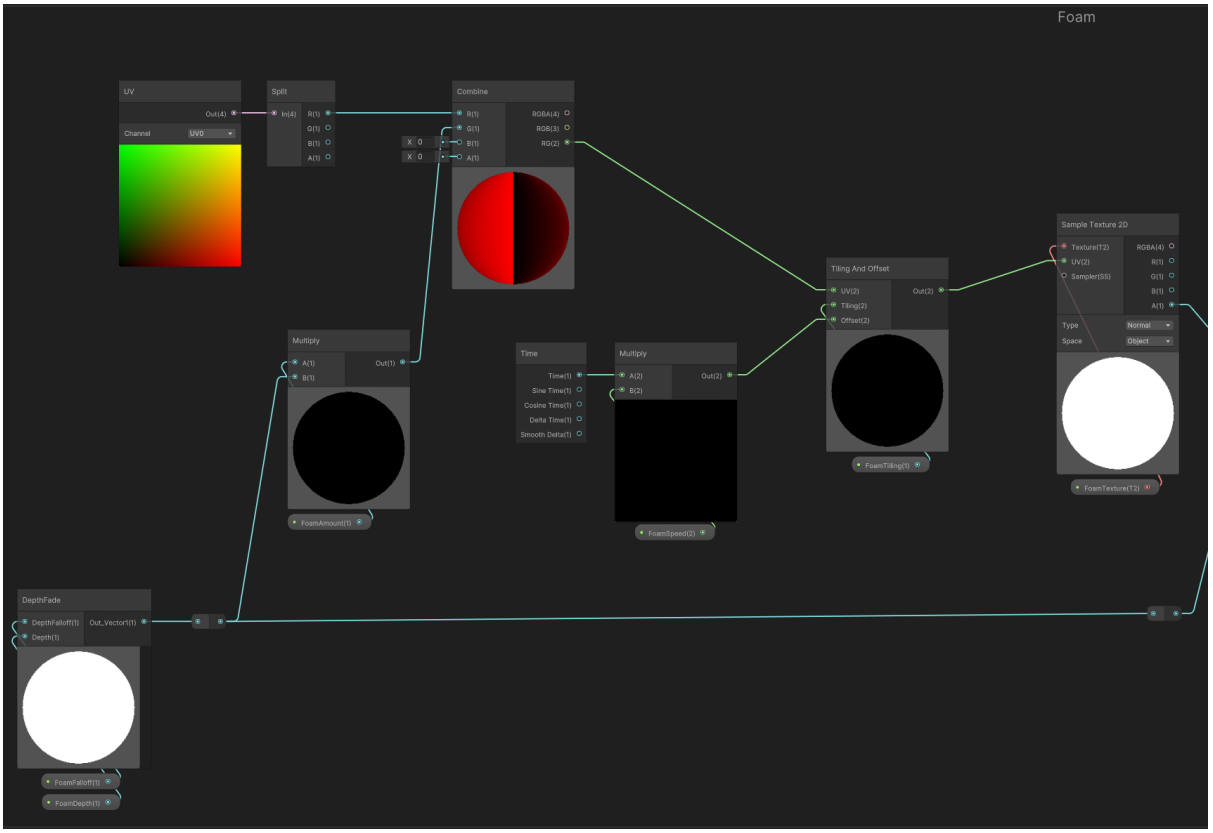


Figura 6.22.6.: Captura de pantalla de la primera parte de crear espuma

Ahora queremos añadir una textura a la espuma por lo que lo primero que hacemos es crear el nodo UV, donde está guardada toda la información de los nodos UV de nuestro modelo y utilizamos el nodo Split para separar las componentes que tienen y utilizar el nodo X, que en este caso es la R. Una vez que ya tenemos esto, creamos el nodo Combine para poder juntar la componente X de la UV y la componente Y, que en este caso proviene del DepthFade multiplicada por el factor de la cantidad de espuma que queramos. Para poder añadir esta textura lo que hacemos es crear el nodo Tilling And Offset, donde el parámetro UV le pasamos la información proveniente del parámetro RG del nodo anterior. Para darle movimiento a esta espuma necesitamos crear el nodo Time, al cual lo multiplicamos por el factor de velocidad llamado FoamSpeed y unimos este valor al apartado de offset, y por último, lo que hacemos es modificar el Tilling de esta textura mediante el parámetro de tilling que está modificado por una variable llamada FoamTilling. Hasta ahora tenemos toda la información necesaria para poder modificar esta textura, pero nos falta la parte más importante que es poder añadir esta textura por lo que necesitamos crear el nodo Sample Texture 2D, al cual le pasamos la textura mediante la variable FoamTillin y le pasamos toda la información almacenada en el nodo Tilling And Offset mediante el parámetro UV. Fijándonos en la Figura 6.22.7, vemos como se multiplica la textura por el DepthFade para obtener la región donde es necesario que actúe esta textura, y utilizamos un Cutout para delimitar esta región.

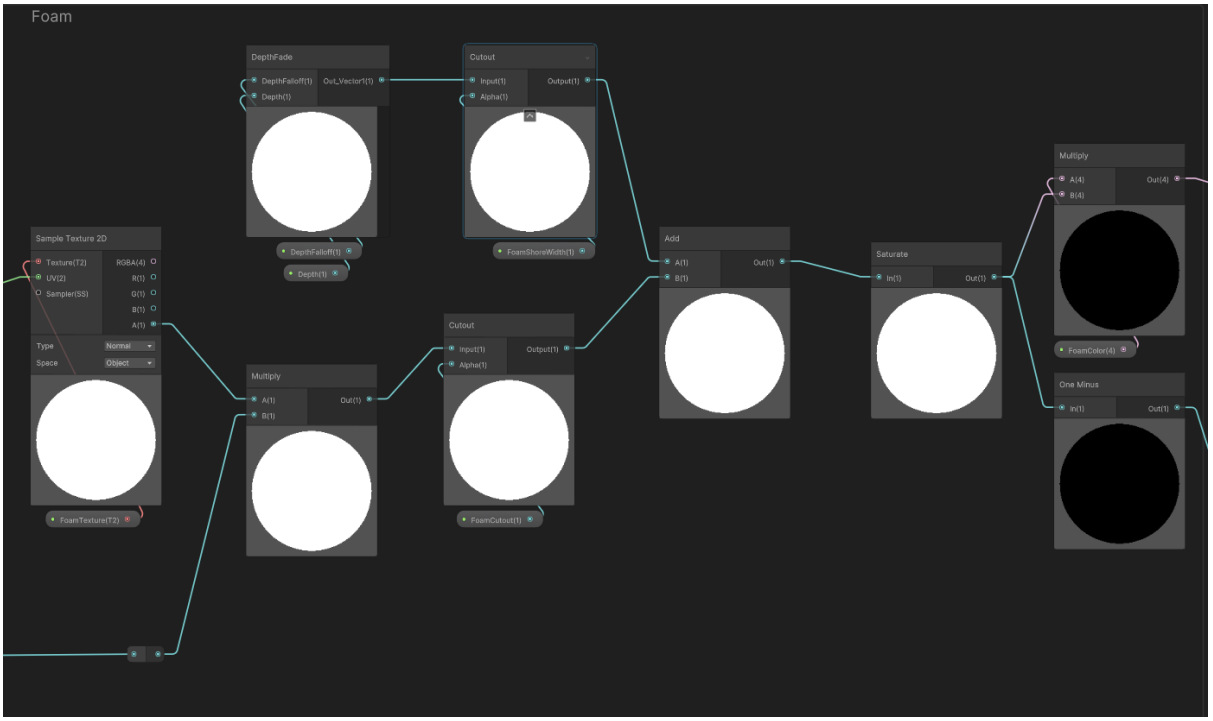


Figura 6.22.7.: Captura de pantalla de la segunda parte de crear espuma

Lo primero que hacemos es añadir el nodo DepthFade el cual conectamos con el nodo Cutout para crear la zona de la orilla. Estos valores los sumamos a los explicados en el párrafo anterior mediante el nodo Add, y estos valores los pasamos al nodo Sature para que estén comprendidos entre 1 y 0. Como queremos que la espuma y el color del agua no se mezclen, necesitaremos por un lado multiplicar el resultado del nodo Sature por el color, y por otro lado, negarlo para que se multiplique por el color del agua. Explicaremos cómo se hace esta unión más adelante, ya que nos hace falta explicar otros apartados antes. Lo descrito en este párrafo se puede visualizar en la Figura 6.22.7.

El siguiente apartado del que vamos a hablar es la creación de cáusticas a lo largo del agua, para conseguir darle un efecto más realista.

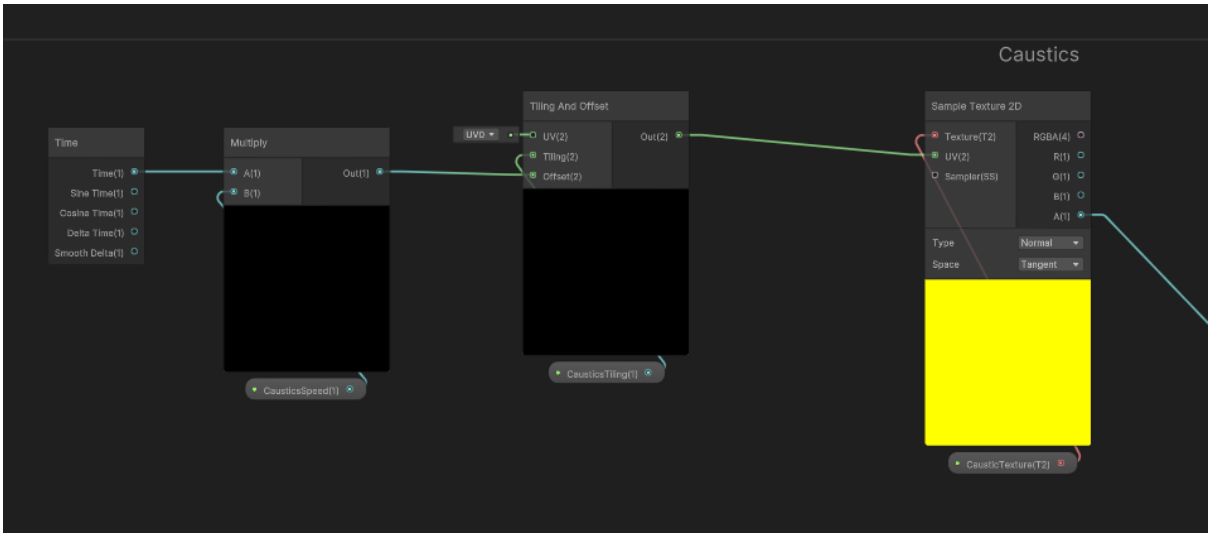


Figura 6.22.8.: Captura de pantalla de la primera parte de creación de las cáusticas

Para crear las cáusticas, como podemos ver en la figura anterior, lo que hacemos es crear una textura al que le aplicamos un desplazamiento y tilling, al mismo ejemplo que en el apartado anterior. El problema de hacer esto es que nos quedarán unas cáusticas como si fueran espuma, lo cual no nos interesa, ya que las cáusticas son simplemente brillos puntuales.

Figura 6.22.9.: Captura de pantalla de la segunda parte de creación de las cáusticas

Como se observa en la Figura 6.22.9, es el mismo código que en la Figura 6.22.8, sólo que observamos la diferencia de que se desplaza a la inversa, ya que multiplicamos la velocidad por -1.

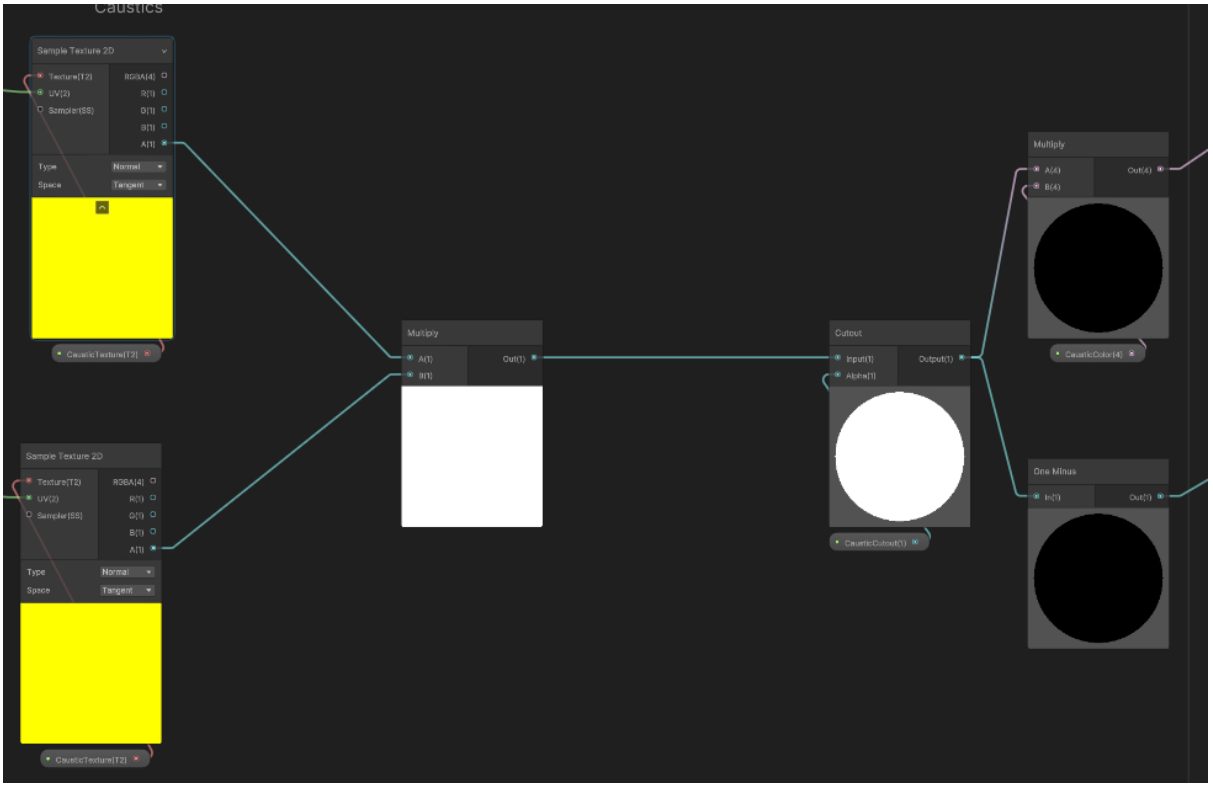


Figura 6.22.10.: Captura de pantalla de la tercera parte de creación de las cáusticas

En esta figura lo que hacemos es multiplicar ambas texturas de cáusticas para que solo haya cáusticas en las zonas que ambas coincidan, y así obtener el resultado deseado. Para limitar la zona de actuación de estas cáusticas tenemos el nodo Cutout, del cual salen dos nodos, uno es el multiply que le asignará el color a estas cáusticas, mientras que el nodo one minus sirve para añadir a las zonas donde no queremos que haya cáusticas.

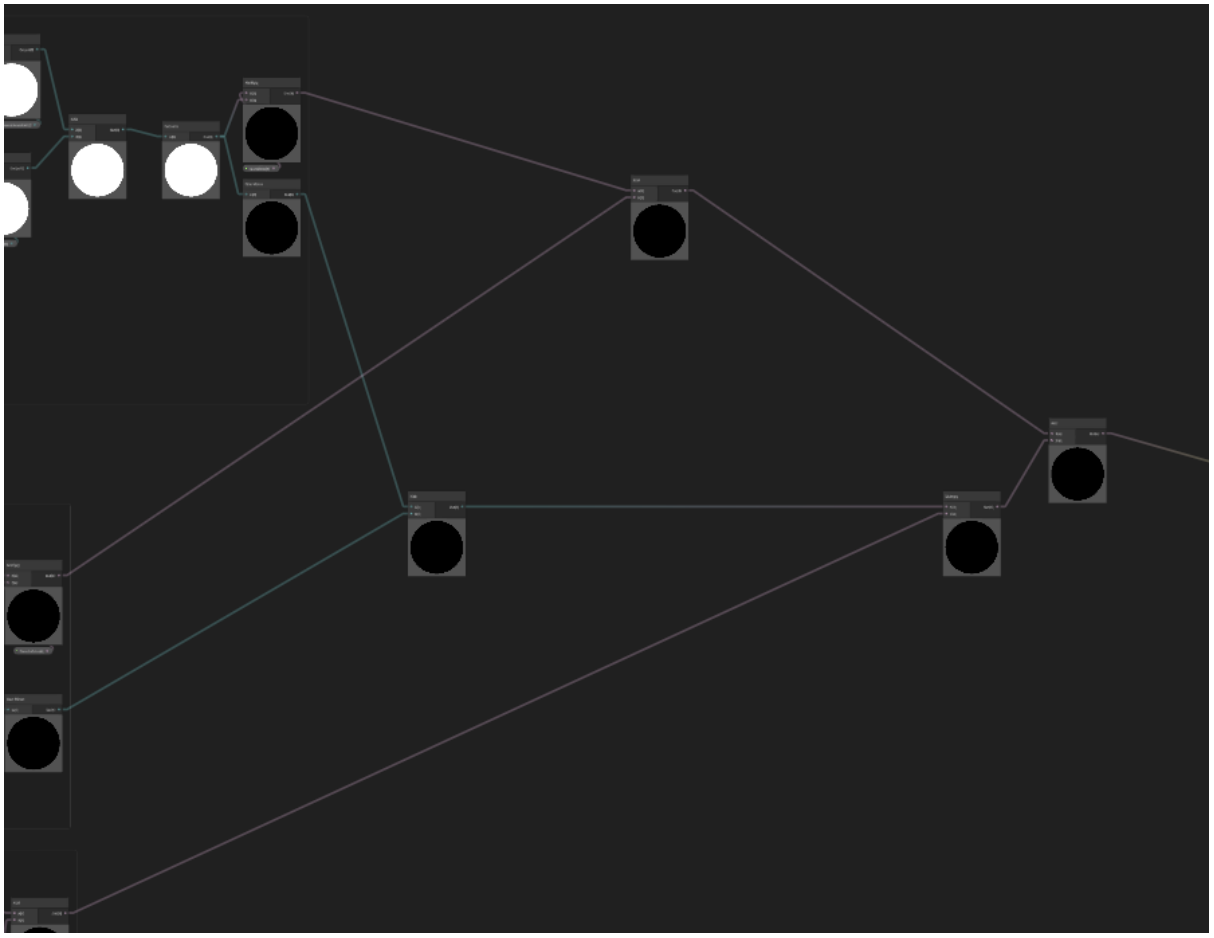


Figura 6.22.11.: Captura de pantalla de unir los elementos

Para unir todo de una forma correcta, nos encontramos con que necesitamos unir con un nodo Add los valores de los nodos One Minus de las cáusticas y de la espuma, porque de esta forma tenemos juntos los nodos donde no se tiene que mostrar color de la espuma, por lo que nos viene muy bien para multiplicarlo con el nodo de color mediante el nodo Multiply. Además, sumamos los valores de los nodos Multiply de la espuma de las cáusticas a través del nodo Add. Como resultado final sumamos el valor de los nodos Add y Multiply mediante el nodo Add. El resultado de este nodo se añade a la propiedad Base Color del shader. Todo esto lo podemos visualizar en la Figura 6.22.11.

La última parte de este proceso consiste en crear el movimiento de las olas, que explicaremos a continuación.

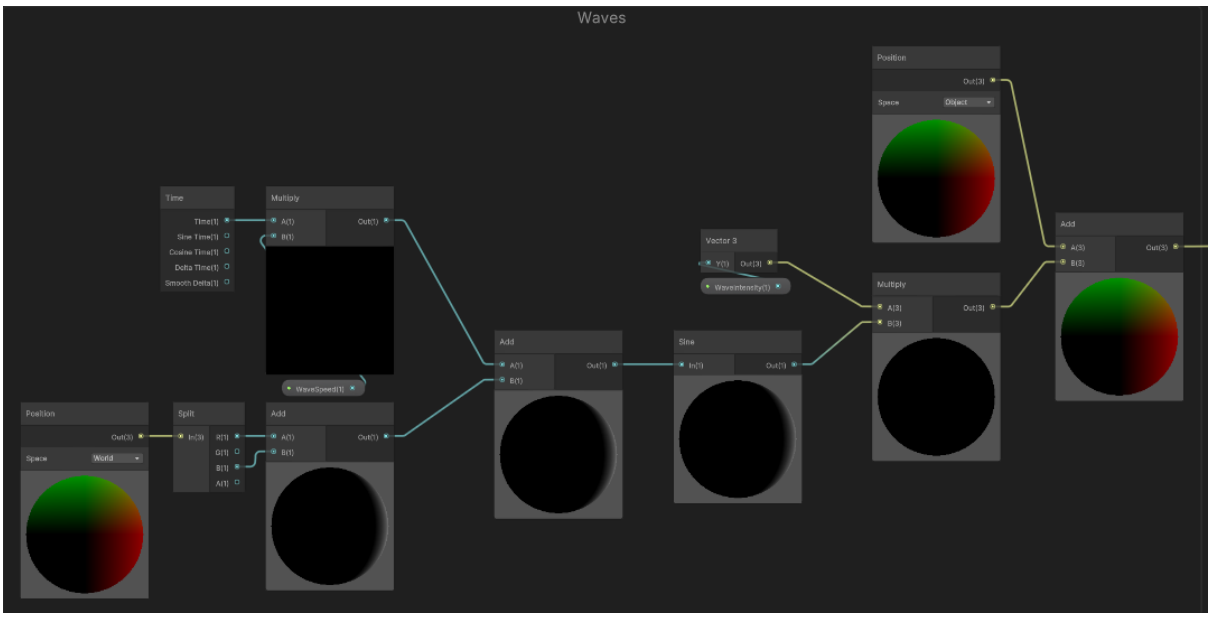


Figura 6.22.12.: Captura de pantalla de unir los elementos

Lo primero que hacemos es añadir el nodo position, el cual nos da la posición del plano en el mundo, al cual le hacemos un Split para poder obtener los componentes por independiente y con el nodo Add sumamos el valor de X y Z. Para poder tener variaciones en el tiempo, multiplicamos el valor del nodo Time por la velocidad de las ondas mediante el nodo Multiply. Una vez que ya contamos con estos dos valores, se los pasamos al nodo Add y el resultado de este al nodo Sine, que se encarga de generar oscilaciones verticales a partir de esta información. Por si nos interesa exagerar estas deformaciones, utilizamos el nodo Multiply para multiplicar el valor del nodo seno al valor de la variable WaveIntesity, pero solo en el eje Y, ya que el resto de ejes no nos interesan. Como paso final, creamos un nodo Position, el cual lo ponemos en espacio local para que funcione correctamente, y le añadimos la información que hemos calculado anteriormente mediante el nodo Add. El resultado de este nodo se añade a la propiedad Position del shader. Todo esto lo podemos ver en la Figura 6.22.12.

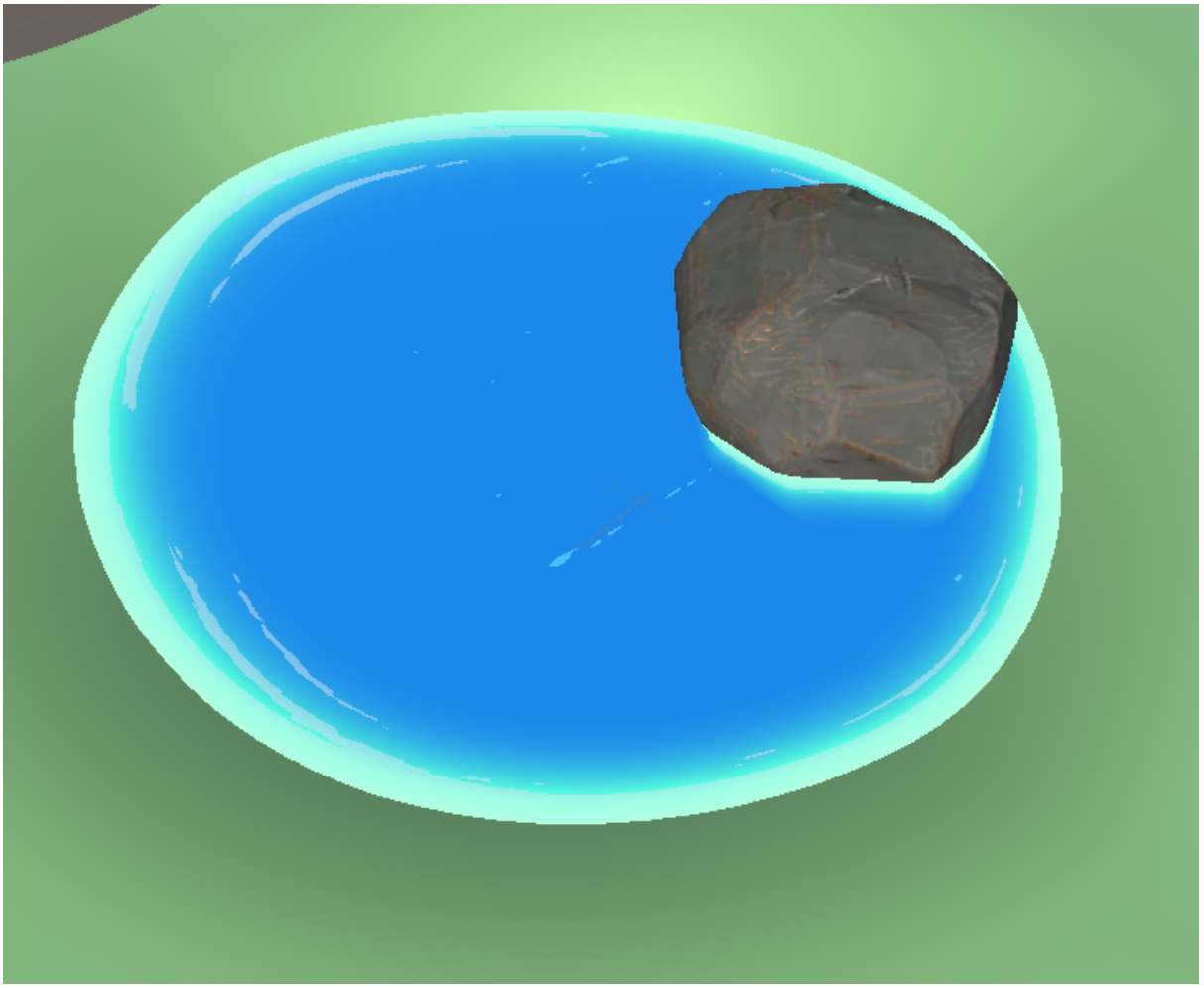


Figura 6.22.13.: Captura de pantalla de unir los elementos

En la Figura 6.22.13 podemos observar el resultado del shader explicado anteriormente.

7. Resultados

7.1 Legislación y normativa vigente

EL juego que hemos desarrollado no presenta ningún problema a nivel legislativo, ya que no se necesita guardar ninguna información del jugador de carácter personal, por lo que no se aplica en ninguna situación la LOPD (Ley Orgánica de Protección de Datos) ni la LSSICE (Ley de Servicios de la Sociedad de la Información y Comercio Electrónico) debido a que el proyecto no constituye una actividad económica.

En cuanto a lo que a problemas de Copyright se refiere, parte de los assets que tenemos son desarrollados por nosotros mismos, y la mayoría de ellos son de uso gratuíto; donde solo es necesario mencionar al autor. Solo una pequeña cantidad de ellos no permiten la comercialización de los mismos, por lo que para poder hacer esto último necesitaríamos hablar con los desarrolladores de estos assets e intentar llegar a un acuerdo o cambiarlos. También es importante destacar que la comercialización no es el objetivo final de este proyecto.

7.2 PEGI

El sistema de Pan European Game Information, más comúnmente conocido como PEGI, es el sistema europeo que nos permite clasificar los juegos mediante iconos. Estos iconos describen la edad recomendada para jugar al juego y para describir el contenido del juego.

Figura 7.2.1: Captura de pantalla de las distintas PEGI

En el caso de nuestro juego, podemos ver que como nos centramos en el sistema de combates, tendríamos que poner la etiqueta violencia. Es importante tener en cuenta que como es un juego de fantasía y no se ve en ningún momento sangre ni

destripamientos ni contamos con violencia verbal, podríamos decir que la edad mínima recomendada es 7 años.

7.3 Resultado final

En las Figuras 7.3.1 a 7.3.16 podemos ver los resultados de este proyecto.

Figura 7.3.1: Captura de la pantalla inicial del juego

Figura 7.3.2: Captura de la zona inicial del juego

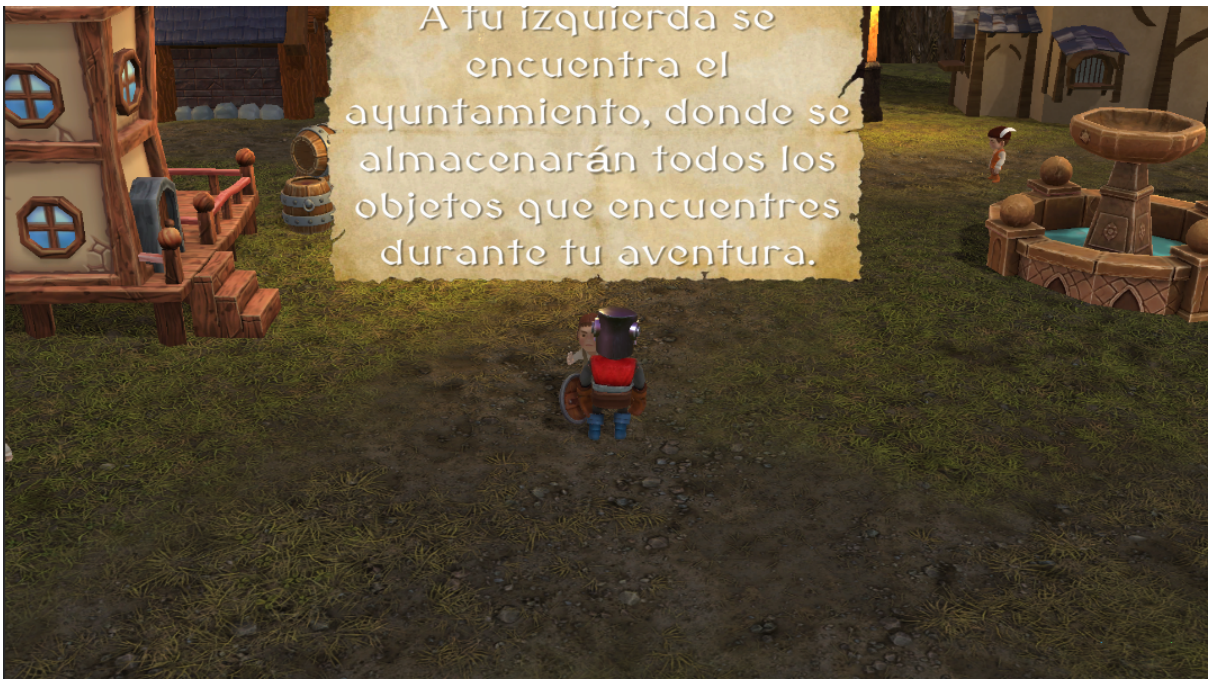


Figura 7.3.3: Captura de la interacción con el NPC de las misiones

Figura 7.3.4: Captura de la interacción inicial con el ayuntamiento

Figura 7.3.5: Captura de pantalla del ayuntamiento

Figura 7.3.6: Captura de pantalla de la interacción con la fuente

Figura 7.3.7: Captura de pantalla de la acción de talar

Figura 7.3.8: Captura de pantalla del resultado de talar + feedback de completar la misión

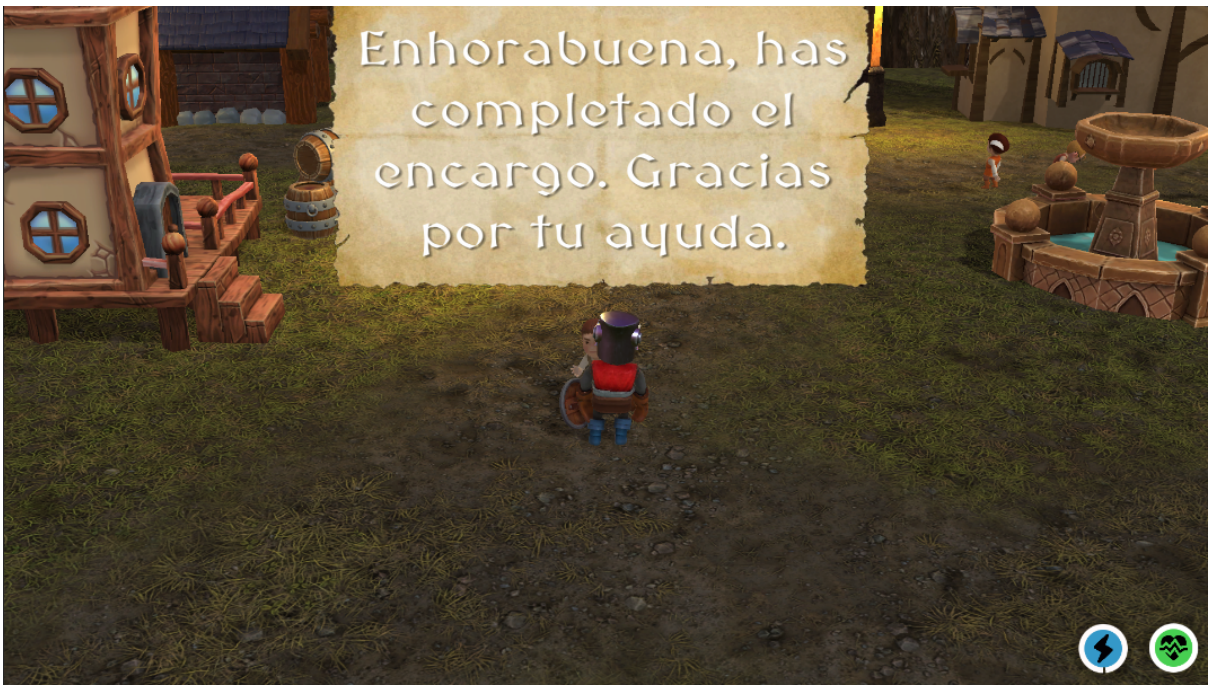


Figura 7.3.9: Captura de pantalla del feedback de completar la misión

Figura 7.3.10: Captura de pantalla de la acción picar

Figura 7.3.11: Captura de pantalla del resultado de picar

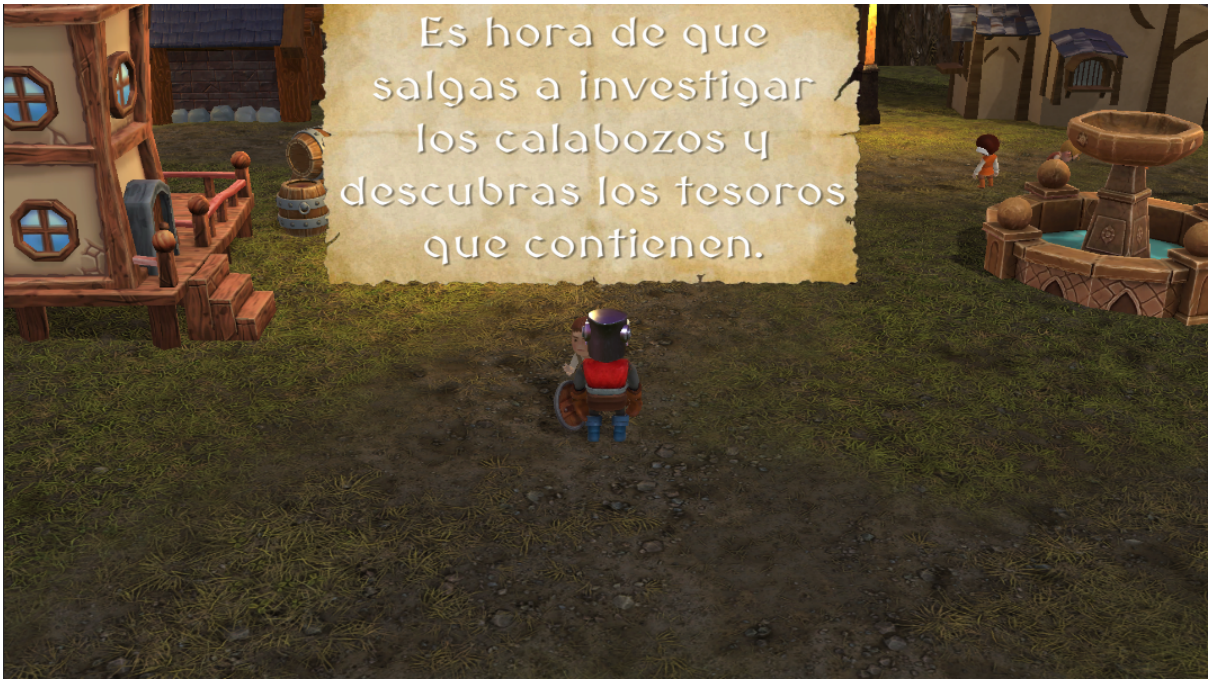


Figura 7.3.12: Captura de pantalla de la misión final

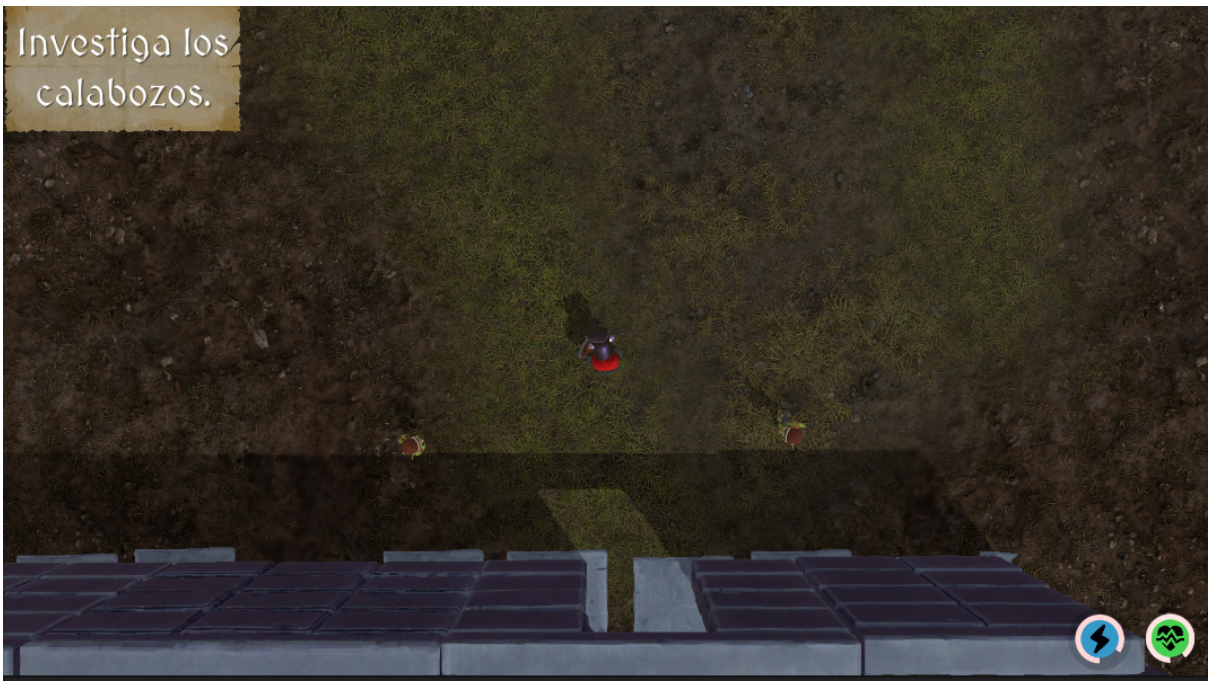


Figura 7.3.13: Captura de pantalla de la acción de recibir daño del jugador

Figura 7.3.14: Captura de pantalla de la acción de morir del enemigo

Figura 7.3.15: Captura de pantalla de la recompensa de los cofres

Figura 7.3.16: Captura de pantalla del menú de pausa

8. Conclusiones

8.1 Valoración del trabajo

Pensando en todo el tiempo que hemos invertido para llevar a cabo este proyecto, nos damos cuenta que realmente hemos conseguido superar todos nuestros objetivos y transformar en realidad esos pensamientos que eran simplemente un sueño un año atrás.

Gracias a todo esto, hemos aprendido a ir superando todas las fases del desarrollo de un proyecto serio, que parte de la necesidad de crear un juego que sea capaz de destacar en un sector donde día a día la gente lucha por crear el próximo bombazo, hasta la parte de buscar la manera más sencilla de implementar cada mecánica, y que dicho código sea lo más reutilizable posible.

Un ejemplo claro de esto es la necesidad de aprender a desarrollar las mecánicas con un toque personal, ya que es muy sencillo seguir los pasos que dictan todos los gigantes del sector, pero no sentarse a pensar si realmente estas rutas encajan con la forma de ver este videojuego y de qué manera se puede adaptar para que tengan el sentido que se necesita. Esta situación nos la encontramos a la hora de desarrollar el movimiento del personaje principal, ya que no nos acababa de gustar la forma de la cual estaba implementado, y preferimos implementar todo a partir de nuestro criterio y la influencia de varios desarrolladores.

Como resumen de lo expresado anteriormente, estamos muy contentos con el resultado obtenido y todo el trabajo realizado. Además, estamos muy felices de que a la hora de ver el videojuego, desde un punto externo al desarrollo, tenemos la sensación de que es realmente un videojuego y no una demo como si nos ha pasado con otros proyectos.

8.2 Desviación de la planificación actual

Durante el desarrollo de este proyecto han ocurrido bastante inconvenientes, pero gracias a las horas de trabajo que hemos invertido, hemos conseguido cumplir con el plan inicial y llegar a todas las fases del desarrollo a tiempo.

9. Trabajo futuro

Como ya se ha explicado anteriormente, este es un trabajo realizado entre dos personas, pero para esta versión está implementada al 100% la parte de Denís, mientras que la parte de Miquel se encuentra a un 20%, por lo que en la siguiente versión se debería de encontrar las siguientes mejoras:

- **Generación procedural de entornos**: Para cumplir con los objetivos de desarrollo del videojuego, tendríamos que desarrollar un sistema capaz de generar un terreno que pueda investigar el jugador, y en el cual se le presente un reto que sea interesante. Junto a esto también se encuentra la creación de un sistema que permita al jugador navegar entre estas islas.
- **Sistema de intercambios:** Para ser un poco permisivos y no castigar tanto al jugador con el hecho de tener que realizar visitas a otras islas, se desarrollará un sistema de intercambios para que pueda conseguir los recursos que necesite, siempre y cuando estos estén disponibles para el intercambio.
- **Comer + inventario**: Para que el jugador sea capaz de poder regenerar vida cuando se encuentre en un viaje a otra isla, desarrollaremos un sistema de inventario donde pueda almacenar contenido temporalmente y poder ingerir comida para regenerar la salud.
- **Cultivar:** Como no queremos que el jugador dependa exclusivamente de los alimentos que encuentre a lo largo de sus visitas a otras islas, hemos pensado en desarrollar un sistema de cultivos donde pueda plantar y recoger los alimentos que consigan crecer en su isla.
- **Sistema de mejoras de estructuras**: Esto nos sirve para ir controlando el progreso del jugador en el juego, y comprender los ritmos necesarios para ir incrementando la dificultad del mismo.
- **Enemigo cuerpo a cuerpo**: Actualmente sólo contamos con un enemigo, el cual es de largo alcance y que es bastante sencillo de debilitar. Para complicar las cosas al jugador, se nos ha ocurrido desarrollar un enemigo de corto alcance, ya que de esta forma el jugador tenga que utilizar más la mecánica de bloquear y no tanto la de atacar.
- **Mejora del sistema de UI**: En esta versión del proyecto nos encontramos con unas UI que son funcionales, pero un poco básicas, por lo que estaría bien que fuéramos un paso más lejos y desarrollar unas con las que el jugador se sienta agusto.
- **Implementar un sistema de dificultad**: como hemos explicado anteriormente en el punto del sistema de mejoras, queremos desarrollar un sistema de dificultad a través del cual los enemigos supongan un reto mayor.

Es importante destacar que, por motivos de desarrollo, puede ser que algunas de estas características no acaben viendo la luz o si lo hacen, estas se encuentren alteradas para adecuarse a las circunstancias necesarias.

10. Bibliografía

En la bibliografía nos encontramos con distintos apartados, ya que preferimos clasificar los distintos tipos de enlaces.

Tutoriales

Gius Caminiti. (9 de febrero de 2021). Shader de agua estilizada con Unity Shader Graph [Video]. Recuperado de https://www.youtube.com/watch?v=jRuCQnp78gk&t=1309s

Pixel icaG. (14 de julio de 2021). How to Create Diamond Gem in Blender in 90 sec - Quickie Tuts #02 [Video].
Recuperado de https://www.youtube.com/watch?v=2HTPvklPVe8

Game Dev Bits. (12 de abril de 2022). Unity3D Fast Tips - Placing a weapon in a character's hand [Video].
Recuperado de https://www.youtube.com/watch?v=IYI_bP_Wjpg

Comp-3 Interactive. (18 de diciembre de 2020). How to Make Radial Progress Bars [Unity Tutorial] [Video]. Recuperado de https://www.youtube.com/watch?v=emYjAOylbho

Flarvain. (6 de febrero de 2021). Shop Tutorial Unity - [2021] [Vídeo]. Recuperado de https://www.youtube.com/watch?v=EEtOt0Jf7PQ&t=632s

Redmatter20. (4 de noviembre de 2021). How to Blend Music into Different Biomes in Unity | Unity Tutorial [Video].
Recuperado de https://www.youtube.com/watch?v=H6ghJweZpnU&t=190s

Assets

Filter Forge. Normal Map [Mapa de normales].
https://www.filterforge.com/filters/12527-normal.html

dagos32. (21 de julio de 2018). Stylized Rock [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/stylized-rock-0dde25b1b7024a68aa4c6c68261e991b

tomanski. (18 de diciembre de 2020). Stylized Shield [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/stylized-shield-61e6ea4f60e3405aa9aca5305bc71352

Icon Icons. Icon Ray [Sprite]. Recuperado de
https://icon-icons.com/es/icono/ray/80903

Icon Icons. Icono Salud, corazon, pulso [Sprite]. Recuperado de https://icon-icons.com/es/icono/salud-corazon-pulso/48576

Pasindu.Anjana. (13 de abril de 2022). Stylized Chest [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/stylized-chest-a074f2d107df43f58f5744a3e6999d9f

Bradobrey. (21 de diciembre de 2015). town hall [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/town-hall-3128ee34c658466ba00699959459eb2e

zawboo. (8 de octubre de 2019). GB2 [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/gb2-2d405e9c08694c7f9e4717e0e54b54c9

zawboo. (7 de octubre de 2019). SD [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/sd-12aba43b896246ce8f936b0f40ab8a35

zawboo. (8 de octubre de 2019). SD2 [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/sd2-6ff04b70dbc2499ba324b03652df6bed

zawboo. (8 de octubre de 2019). WT[Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/wt-5d9fd96b31054f13b15c2f7099ba3dae

catzer. (6 de marzo de 2020). Small Stylized House [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/small-stylized-house-be5c115c4dca494881b87409e6246289

sashagallie. (2 de marzo de 2020). Medieval House [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/medieval-house-34c9f5b15230473285911500ede10dfc

Pranav_1603. (2 de enero de 2023). Stylized House [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/stylized-house-d515e7457d814d2dae4e0ccdd3dc4828

Misa. (16 de diciembre de 2020). -Game ready- Stylized Fox Village Windmill [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/game-ready-stylized-fox-village-windmill-29d7de4a599047b9b9c85ae64153f916

Aartee. (14 de noviembre de 2019). The Blacksmith's [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/the-blacksmiths-d93444656b204ddd8270b9a9be0d99ec

RipeR. (8 de enero de 2022). Barrels [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/barrels-71b622c6e4aa4a848d9b6fcdbfc27ad7

SK Milanova. (12 de abril de 2018). Simple Medieval Lamp [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/simple-medieval-lamp-c1f1184dc3274720b707dc6293279c31

HenryBoadle. (3 de abril de 2019). Modular Dungeon Kit [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/modular-dungeon-kit-ecbeefd3212f447ba2f10f1986ad5769

Hermi Marske. (16 de marzo de 2021). Stylized fountain 3DEX tutorial [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/stylized-fountain-3dex-tutorial-1b88af909c6d4862a268823d0b0f1eed

ValaBanana. (24 de noviembre de 2020). Letrero V2 [Archivo 3D]. Recuperado de https://sketchfab.com/3d-models/letrero-v2-30bd5f2c9f604ab9b9d2b5d1882ef6eb

Wojciech Kalinowski. (s.f.). Medieval Sharp [Fuente de texto]. Recuperado de https://www.dafont.com/es/wojciech-kalinowski.d10671

Pixaby. Página de descarga de sonidos. [Sonidos]. Recuperado de https://pixabay.com

IENBA. (2 de noviembre de 2022). Game Reward [Sonido]. Recuperado de https://freesound.org/people/IENBA/sounds/656643/

Inspector J. (6 de marzo de 2018), Ice » Impact, Ice, Moderate, A.wav [Sonido]. Recuperado de https://freesound.org/people/InspectorJ/sounds/420882/

spookymodem. (25 de septiembre de 2014). Goblin Death.wav [Sonido]. Recuperado de https://freesound.org/people/spookymodem/sounds/249813/

Fenodyrie. (2 de abril de 2021). Goblins » goblin dying in fight.wav [Sonido]. Recuperado de https://freesound.org/people/Fenodyrie/sounds/565928/

newlocknew. (22 de abril de 2022). Rocks.Stones » ROCKBrk_Stone hit.Shattering Into Fragments_EM_(7lrs).wav [Sonido]. Recuperado de https://freesound.org/people/newlocknew/sounds/631485/

11. Anexos

Como anexos hemos adjuntado el proyecto entero de unity, la build del proyecto y un vídeo de una partida entera; a través de la cual podemos ver el uso de las diferentes mecánicas de las que disponemos, los cuales se encuentran disponibles en el pdf llamado Anexos, que contiene un enlace.

En caso de que se quiera acceder a cualquier recurso que hayamos implementado, simplemente es necesario acceder a la carpeta de assets del proyecto e ir navegando a través de las distintas categorías. Para esto necesitamos descargar el archivo correspondiente del pdf mencionado en el apartado anterior.

12. Manual de usuario

12.1 Iniciar el juego

Para descargar el juego tenemos que acceder al pdf llamado anexos, entrar en el enlace , descargar el archivo de la build y descomprimirlo.
Para poder ejecutar el juego, necesitamos acceder a la carpeta llamada Build y dentro de esa carpeta buscamos el archivo llamado “TheLastIsland.exe”, el cual se encargará de lanzar el juego. Este juego cuenta con la función de guardado, pero esta es manual.

12.2 Controles

Este juego ha sido diseñado para jugarse tanto con teclado como con mando, por lo que contamos con el siguiente esquema de control:

- **Movimiento:** teclas WASD y el joystick izquierdo del mando.
- **Correr**: Shift o pulsar el joystick izquierdo.
- **Atacar**: click izquierdo o botón RB del mando.
- **Bloquear**: Click derecho o botón LB del mando.
- **Interactuar**: tecla E o botón Y del mando.
- **Picar**: tecla F o botón A del mando.
- **Talar**: tecla G o botón X del mando.
- **Ataque especial**: botón central del ratón