Universitat de Girona
Escola Politècnica Superior

Trabajo final de grado

Estudio: Grado en Diseño y Desarrollo de Videojuegos

Título: Diseño y desarrollo de un videojuego comercial

Documento: Memoria

Alumnos: Aaron Miranda Gómez
Aitor López Alonso
Lluís Palerm Tur
Xabier Goenaga Urkiola

Tutor: Gustavo Patow
Departamento: Informática, Matemática Aplicada y Estadística
Área: Lenguajes y Sistemas Informáticos

Convocatoria Junio / 2021

Índice

1. Introducción y objetivos 6
1.1. Introducción 6
1.2. Motivaciones 6
1.3. Propósito y objetivos del proyecto 6
1.4. Distribución de tareas 7
2. Estudio de viabilidad 9
2.1. Recursos necesarios i viabilidad 9
2.1.1. Recursos técnicos 9
2.1.2. Recursos humanos 9
2.1.3. Viabilidad económica 10
2.2. Estudio de Mercado 11
2.2.1. Estado del arte 11
2.2.2. Modelo de negocio 13
2.2.3. Matriz de competencia 13
2.3. Público objetivo i tipología de jugador 14
3. Planificación 15
3.1 Planificación previa 15
3.2 Desarrollo de las tareas 15
3.3 Explicación de las herramientas utilizadas 16
3.3.1 Motor del videojuego – Unity 16
3.3.2 Editores de código 17
3.3.3 Software artístico 17
3.3.4 Software para documentación 17
3.3.5 Otros 18
4. Marco de trabajo y conceptos previos 19
5. Diseño del videojuego 19
5.1 Mecánicas 19
5.1.1. Espacio del juego 19
5.1.2. La mecánica del juego, los retos que se tienen que afrontar y las acciones posibles 20
5.1.3. Objetos, recursos e interacciones que puede hacer el jugador 21
5.1.4. Economía interna del juego 22
5.1.5. Estudio de los diferentes niveles del juego 22
5.1.6. Interfaces 22
5.2 Estudio y diseño de personajes 23
5.2.1. Peso narrativo de los personajes 23

5.2.2. Los personajes como base de la jugabilidad 23
5.2.3. Estilo artístico (condicionado por la jugabilidad) 23
5.2.4. Coherencia, características físicas y psicología 24
5.2.5. Lenguaje corporal y gestual, movimiento 25
5.2.6. Objetos y vestimenta que caracterizan a los personajes 25
5.3 Narrativa 25
5.3.1. Sinopsis 26
5.3.2. Backstory 26
5.3.4. Narrative devices 26
5.3.5. Tensión dramática y tensión de jugabilidad 27
5.3.6. Plot points y estructura narrativa 27
5.3.7. Mecanismos para hacer avanzar la trama 28
5.3.8. Granularidad 28
5.3.9. Intercalar los elementos de la narración en la estructura del juego 28
5.3.10. Disparadores (triggers) 28
5.4. Mundos de juego 28
5.4.1. Dimensión física 28
5.4.2. Dimensión temporal 35
5.4.3. Dimensión ambiental 36
5.4.4. Referentes estéticos 37
5.4.5. Dimensión emocional 39
5.4.6. Aspectos Éticos 40
5.5. Interfaces 40
5.5.1 Mapa 40
5.5.2 Inventario del carro 41
5.5.3 Comercio 41
5.5.4 Misiones 42
5.5.5 Inventario 43
5.5.6 Diálogos 43
5.5.7 Creador de diálogos 44
5.6 Game Layout Charts 46
5.7 Diseño de objetos 47
5.8 Producción externa 47
5.9 Entorno de Unity y diseño del proyecto 47
5.10. Elementos de feedback 48
5.11 Diseño de los efectos y partículas 48

5.12. Diseño de sonido 51
5.12.1. Efectos de sonido 51
5.12.2. Voces 52
5.12.3. Música 52
6. Implementación y pruebas 53
6.1 Carro 53
6.1.1 Movimiento 53
6.1.2 Limitadores 55
6.1.3 Inventario carro 57
6.2 Misiones 62
6.2.1 Funcionamiento 62
6.2.2 Creación de misiones 65
6.2.3 Guardar progreso de misión 71
6.3 NPC 73
6.3.1 Waypoint System 73
6.3.2 Movimiento de los NPCs 76
6.4 Optimización 77
6.4.1 Mesh combiner 77
6.4.2 LOD propio 80
6.5 Personaje 83
6.5.1 Movimiento 83
6.5.2. Interacción 83
6.6.1 Dibujar 85
6.6.2 Profundidad 87
6.6.3 Borrar 89
6.6.4 Guardar en disco 91
6.7 Hierba 92
6.8 Mecánica compraventa 98
6.8.1 Pizarra 98
6.8.2 Gestión de clases 98
6.8.3 Implementación compraventa 98
6.8.4 Editor de economía 103
6.9 Editor de materiales 106
6.10 Cascada 110
6.11 Rio 114
6.11.1 Generar mesh 114

6.11.2 Shader agua 120
6.12. Diálogo 123
6.12.1 Creador de ficheros de diálogo (*Dialogue Maker*) 123
6.12.2. Sistema de diálogo 127
6.13 Generar ciudades 132
6.13.1 Los planos 132
6.13.2 Interpretar el JSON en Unity 133
6.13.3 Las paredes 135
6.13.4 Los tejados 137
6.13.5 Murallas 144
6.13.6 Adaptar al terreno 145
6.13.7 Guardar ciudades 146
6.13.8 Resultados 147
7. Resultados 148
7.1 Legislación y normativa vigente 148
7.2 Pegi 148
7.3 Resultado final 149
8. Conclusiones 157
8.1 Valoración del trabajo 157
8.2 Desviaciones del trabajo original 157
9. Trabajo futuro 158
10. Bibliografía 159
11. Manual de usuario 159
11.1 Iniciar el proyecto 159
11.2 Objetivo del videojuego 159
11.3 Controles 159
11.4 Menú de opciones 160

1. Introducción y objetivos

1.1. Introducción

La industria de los videojuegos es actualmente una de las industrias de entretenimiento más populares. Hoy en día es más fácil para cualquier persona desarrollar un videojuego, independientemente de su nivel de conocimiento en la materia, hecho que hace cada día salgan nuevos videojuegos, llegando al punto de que, en 2020, sólo en la plataforma Steam, salieron a la venta 9852 videojuegos[1]. Pero pese a la cantidad de juegos que hay en el mercado, una mayoría son videojuegos que tienen algún aspecto o mecánica basada en combate, siendo los juegos que no implementan ningún tipo de combate, inusuales.

Por esa misma razón consideramos que puede ser muy interesante el desarrollo de un videojuego que no incorpore ninguna mecánica de combate, que simplemente el jugador sienta la tranquilidad de recorrer el mundo acompañado de una gran historia.

1.2. Motivaciones

Creemos que todos los miembros de este grupo compartimos las siguientes motivaciones con el proyecto:

- Desarrollar un proyecto grande des de 0.
- Trabajar en grupo compartiendo conocimientos.
- Gestionar las responsabilidades de cada miembro del grupo.
- Ir más allá de lo que hemos aprendido durante el grado.
- La creación de un proyecto del cual los 4 estemos orgullosos.

1.3. Propósito y objetivos del proyecto

El propósito del proyecto es el desarrollo de un videojuego de género *Strand Game,* inventado por el pionero de la industria Hideo Kojima. Este consiste en el desplazamiento entre diferentes espacios para cumplir las misiones que van entrelazando las diferentes historias de los personajes del juego. Para cumplir estas misiones habrá que negociar con NPC's, transportar objetos, buscar materiales.

Para ser más concretos queremos desarrollar-lo usando el motor de videojuegos Unity3D y los objetivos del proyecto son:

- Desarrollar las mecánicas básicas de un videojuego (movimiento, cámara, etc.…).
- Mapa interactivo para que el jugador pueda dibujar-lo de manera que lo entienda.
- Sistema de comercio.
- Desarrollar una herramienta de creación de misiones vía nodos gráficos para facilitar el desarrollo.
- Generador procedural de ciudades medievales.
- Desarrollar una herramienta de diálogo con la posibilidad de añadirle ramificaciones.
- Crear una historia que dé motivos al jugador para interactuar con los sistemas de comercio y el movimiento por el mundo.
- Desarrollar una economía que vaya acorde con la evolución de la historia, para transmitir, a través de las mecánicas, el estado del mundo.
- Creación de un tiempo de juego de 7 días en el cual, dependiendo del día la historia, evoluciona de una u otra manera.

Formando todos los objetivos anteriores parte de un objetivo único y mayor que es el de desarrollar un videojuego 3D de gran envergadura desarrollando sistemas que faciliten su desarrollo.

1.4. Distribución de tareas

Todos los miembros del proyecto tenemos un perfil similar ya que venimos de los mismos estudios, pero como cada uno de nosotros tenemos preferencias, desde el principio del proyecto marcamos una distribución de tareas que ha ido evolucionando durante el avance del proyecto. A continuación, analizamos la distribución de tareas de cada miembro del equipo.

Aitor López Alonso

A Aitor le gusta mucho la unión entre el mundo del arte y el de la tecnología, por eso estaba muy interesado en realizar tareas que se enfocaran al perfil de artista técnico y ha distribuido, principalmente, sus tareas entre estética y tecnología, como se puede apreciar en la Figura 1.1

Estética	30%
Narrativa	5%
Mecánicas	5%
Tecnología	60%

Figura 1.1 Distribución tareas Aitor López

Aaron Miranda Gómez

Aaron siempre ha sido un apasionado de las historias y por eso decidimos que él era el más indicado para darle sentido a nuestros personajes y mundo. Además de pensar, y escribir, la historia también se ha encargado de introducirla y hacer la interacción con ella en el juego. En la Figura 1.2 podemos ver la dedicación de Aaron en cada uno de los apartados.

Estética	3%
Narrativa	50%
Mecánicas	7%
Tecnología	40%

Figura 1.2 Distribución tareas Aaron Miranda

Xabier Goenaga Urkiola

Xabier tiene un perfil muy polivalente, como se puede ver en la Figura 1.3, pero sobre todo le gusta explorar nuevas tecnologías y siempre está intentando aprender cosas nuevas. Por eso, durante el proyecto, se ha ido adaptando y ha ido aprendiendo maneras de desarrollar nuestra visión.

Estética	25%
Narrativa	5%
Mecánicas	25%
Tecnología	45%

Figura 1.3 Distribución tareas Xabier Goenaga

Lluís Palerm Tur

Lluís es el integrante del equipo que más disfruta de los juegos por su jugabilidad y eso es lo que siempre intenta transmitir cuando los crea, que un juego sea divertido y agradable de jugar. Como se puede apreciar en la Figura 1.4, Lluís se ha centrado principalmente en las tareas que afectan a este aspecto.

Estética	5%
Narrativa	5%
Mecánicas	35%
Tecnología	55%

Figura 1.4 Distribución de tareas Lluís Palerm

Finalmente, en la Figura 1.5 recopilamos la dedicación global de las tareas para tener una visión más general los aspectos más trabajados del proyecto.

Estética	30 + 3 + 25 + 5 = 63	**15,75%**
Narrativa	5 + 50 + 5 + 5 = 65	**16,25%**
Mecánicas	5 + 7 + 25 + 35 = 72	**18%**
Tecnología	60 + 40 + 45 + 55 = 200	**50%**

Figura 1.5 Distribución de tareas del proyecto

2. Estudio de viabilidad

2.1. Recursos necesarios i viabilidad

A la hora de definir la viabilidad del proyecto, nos centramos en los aspectos mínimos necesarios para considerar a nuestro videojuego viable, teniendo en cuenta que somos un equipo formado por sólo cuatro estudiantes. Estos recursos necesarios se encuentran divididos en los apartados que se muestran a continuación.

2.1.1. Recursos técnicos

En temas de hardware, utilizamos los ordenadores tanto de sobremesa como portátiles que ya teníamos disponibles anteriormente:

- Las características del **ordenador de sobremesa de Aitor** son: CPU Ryzen 7 3700X 8-core 3.6 GHz, GPU NVIDIA GeForce RTX 2060 8GB, RAM DDR4 16GB 3200 MHz, SSD Samsung 860 QVO 1TB, placa base ASUS TUF B450-PLUS GAMING
- Las características del **ordenador de sobremesa de Lluís** son: CPU Intel i5-9600K 3.7GHz 6c 6t, GPU: GTX 1050ti 4GB GDDR5 1500MHz, RAM: DDR4 16GB 3200MHz, SSD Main: Kingston SATA 222GB 2,5", SSD: Intel Optane Nvme 32GB, HDD 1: Segate SATA 960GB 3,5", HDD 2: WD SATA 960GB 2,5", Placa base: Gigabyte z390 GAMING X).
- Las características del **ordenador portátil de Lluís** son CPU: Intel i5-1035G7 1.2GHz 4c 8t, RAM: DDR4 8GB 3733 MHz, SSD Main: Toshiba Nvme 128GB, Placa base: Placa personalizada Microsoft Surface Laptop 3.
- Las características del **ordenador de sobremesa de Xabier** son: CPU Intel i5-6500 3.2GHz, GPU Radeon RX 480, RAM 16 GB, SSD 860 EVO 500 GB, HDD Toshiba DT01ACA100, Placa base MSI H110M PRO-D
- Las características del **ordenador de sobremesa de Aaron** son: CPU Intel(R) Core(TM) i5-4460 3.20Ghz, GPU NVIDIA GeForce GTX 470, RAM 8GB, SSD SATAIII 128 GB, HDD WDC WD10EZEX-08M2NA0 932 GB, Placa base H81MHV3 BIOSTAR Group
- Las características del **ordenador portátil de Aaron** son: CPU Intel(R) Core(TM) i7-8750H 2.20GHz, GPU GTX 1060 6GB GDDR5, RAM 16 GB, SSD NVMe PCIe Gen3 298GB, SSD SATA 158GB, Placa base MS-16Q3 Micro-Star International.

También utilizamos un **micrófono de alta calidad** (Audio-Technica AT2020), que nos permitió grabar ficheros de audio para el proyecto.

Para el software, nos centramos en programas gratuitos, para así reducir costes lo máximo posible:

- El motor de videojuegos **Unity** (versión 2019.4.10f1)
- El editor de audio **Audacity**.
- **Visual Studio Code** para programar, editar y revisar el código.
- **Discord** para poder mantener comunicaciones a distancia.
- **Github** y su versión de escritorio, **Github Desktop**, para poder mantener versiones de control del proyecto y permitirnos trabajar simultáneamente.
- **Blender** para la creación y modificación de los modelos 3D utilizados.
- **GIMP 2.0** para la edición y creación de imágenes.

2.1.2. Recursos humanos

A la hora de hacer videojuegos, hay una diferencia muy importante entre los equipos de pequeño tamaño y aquellos formados por centenares o incluso miles de personas; mientras que en los estudios grandes cada persona tiene un rol específico y no suelen trabajar fuera de ellos, en los equipos

pequeños raramente una persona está dedicada a una sola área de trabajo, sino que suelen participar en diferentes aspectos de la creación del videojuego, ya sea por pasión o por falta de miembros.

Nuestro caso es del segundo tipo: al ser un equipo de cuatro personas, todos participamos en las áreas más básicas del proyecto a la vez que trabajábamos en un área específica.

Con esto en mente, los roles que desempeñamos durante el desarrollo de este proyecto son los siguientes:

- Un **diseñador de juego** define las bases del videojuego, desde la más amplia visión hasta los más pequeños detalles: que podrá hacer el jugador, que elementos existirán en el videojuego, y como serán las interacciones entre ellos y con el jugador. Se le puede entender como el que diseña los planos y límites del mundo de juego, que luego el resto del equipo usará para trabajar.
- El **diseñador de audio** se dedica a obtener audio (por medios propios, de bases de datos o comprándolo), y modificándolos para que se puedan aplicar al juego.
- Un **programador** implementa las mecánicas y las reglas que regirán el mundo de juego, utilizando algún lenguaje de programación, como puede ser C++ o Python.
- Un **guionista** se dedica a escribir los diálogos y historia del videojuego. Pueden trabajar en detalles como *flavour text* (libros dentro del universo de juego, por ejemplo), en los chascarrillos que gritan los personajes dentro del mundo del juego ("¡Tiro una granada!", "¡Voy a flanquearle!"), en los diálogos de las cinemáticas y otros elementos narrativos.
- Un **artista** crea todo el arte que haría falta para el videojuego. Esto incluye, entre otros, modelos de personajes, arte conceptual y el estilo visual de los menús. En nuestro caso, sería un artista de modelado 3D.

En la creación de videojuegos pueden participar muchos más roles, como **márquetin** o **testeo**, e incluso estos roles definidos arriba se pueden dividir aún más, con el diseñador de juego pudiendo ser un **diseñador de combate** o **diseñador de niveles**, como ejemplo. Pero, en nuestro caso los cinco roles definidos arriba fueron los que nos hicieron falta para poder llevar a cabo el proyecto.

2.1.3. Viabilidad económica

Al ser un Trabajo de Final de Grado, la viabilidad económica del proyecto no es un elemento principal, ya que el objetivo es poder presentar el proyecto acabado a tiempo.

Mirando los recursos técnicos, ya disponíamos de todo el hardware de antemano, mientras que en temas de software y recursos utilizamos aquellos que serían gratuitos:

- **Todos los archivos de audio**, exceptuando las voces de los personajes que hicimos nosotros y dos casos específicos, fueron obtenidos a través de la página **freesound.org**.
- **La mayoría de modelos**, como los de los personajes, fueron obtenidos de paquetes de recursos disponibles en **itch.io.**

El único gasto que hicimos en nuestro caso fue la compra de los modelos 3D utilizados en la creación de las ciudades. Este gasto fue 80 euros, que además de ser asequible, nos permitió poder centrarnos en otras áreas del videojuego, ya que ninguno de nosotros tenemos un perfil puramente artístico.

Los recursos humanos, de nuevo, son gratuitos ya que estamos haciendo el trabajo con la mentalidad de terminar el proyecto, no para obtener compensación económica. Aun así, como ejemplo, podemos observar los salarios medios de nuestros puestos, acorde a la página web Payscale[2]:

- Diseñador de videojuego: 2.413€/mes.

- Diseñador de audio: 2.916€/mes.
- Programador: 1.666 €/mes.
- Guionista: 1.915 €/mes.
- Artista 3D: 1.996 €/mes.

2.2. Estudio de Mercado

Habiendo decidido las bases de nuestro videojuego, el estudio de mercado nos ayudó a poder ver si tendríamos un hueco en el mismo, observando otros juegos similares al nuestro y planteando un modelo de negocio que nos funcione.

2.2.1. Estado del arte

Como ya se ha definido anteriormente, nuestro proyecto intentaría pertenecer al género del *Strand Gam*e, reado por Hideo Kojima y el videojuego Death Stranding (Kojima Productions, 2019).

Figura 2.1 Captura de pantalla de Death Stranding

Death Stranding es un videojuego en tercera persona de mundo abierto, y aunque tiene combate, es una parte muy mínima de la experiencia, siendo el foco de esta que el jugador transporte paquetes a diferentes puntos del mapa para así ayudar a unir una américa post apocalíptica.

Como bien dice Hideo Kojima, "Death Stranding es un juego de acción completamente nuevo, donde el objetivo del jugador es volver a conectar ciudades aisladas y a una sociedad fragmentada. Se crea de manera que todos los elementos, incluidos la historia y la jugabilidad, estén vinculados entre sí por el mismo hilo o conexión."[3]

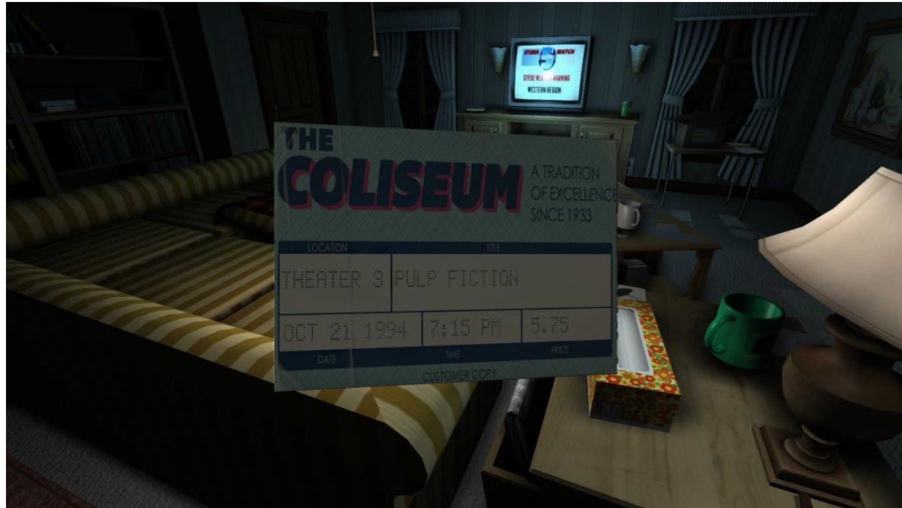

Figura 2.2 Captura de pantalla de Gone Home

Como solo hay un género del tipo *Strand Game*, también miramos hacia los *Walking Simulators*, juegos donde la narrativa toma peso central de la obra y toda su experiencia se basa en ella, como sería nuestro caso. Como ejemplos destacables, decidimos centrarnos en Gone Home (Fullbright, 2016) y What Remains of Edith Finch (Giant Sparrow, 2017).

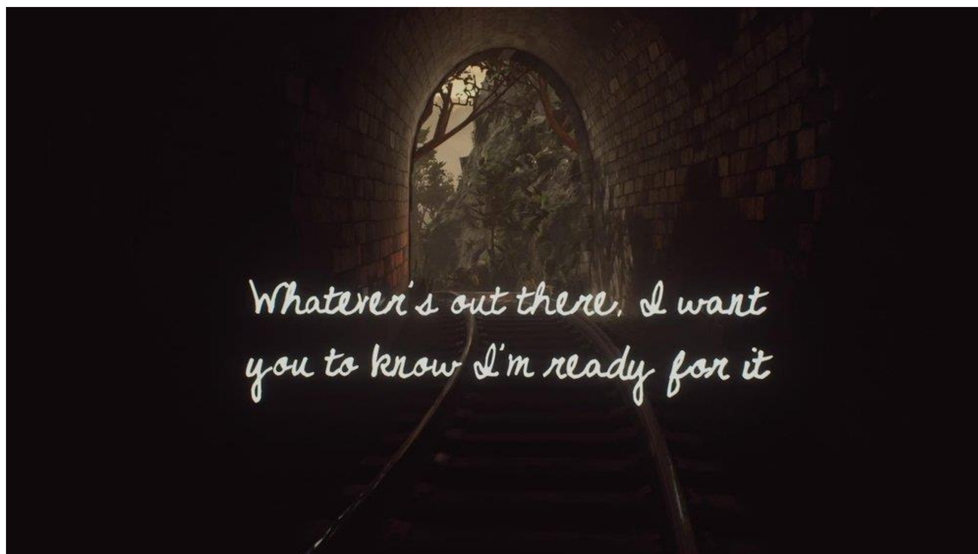

Figura 2.3 Captura de pantalla de What Remains of Edith Finch

Los dos son experiencias en primera persona de corta duración, pero mientras que Gone Home fue la consideraba obra pionera del género (antes hubo otras, pero las bases del género fueron marcadas por esta obra), What Remains of Edith Finch elevó el nivel del género con una obra que no es simplemente “avanza para saber más”, sino que hacía que cada pequeña historia dentro de su narrativa tuviera unas mecánicas diferentes, haciendo que el jugador tuviera una mejor conexión emocional con el juego.

2.2.2. Modelo de negocio

Dentro del mundo de los videojuegos hay diferentes métodos que los videojuegos pueden emplear para ganar dinero:

- Los juegos **free-to-play** son aquellos que no tienen ningún tipo de pago esencial que bloquee al jugador a la hora de disfrutar, pero como es el caso con League of Legends (Riot Games, 2009). Muchos de ellos dan la opción al jugador de gastar dinero en recursos que no afectan a la jugabilidad (por ejemplo, con elementos cosméticos).
- Una variante de los juegos free-to-play son los **freemium**: en estos tipos de juegos, que suelen ser de móvil, se utilizan diferentes tácticas para incentivar al jugador a gastar pequeñas cantidades de dinero. Un ejemplo sería Genshin Impact (MiHoYo, 2020), que permite comprar packs de materiales para acelerar el proceso de tener que jugar horas o incluso días para obtener la misma cantidad de manera gratuita.
- Muchos MMO (Massive Multiplayer Online o Multijugador Online Masivo) utilizan un método de **suscripción**, donde los jugadores pagan una vez al mes, similar a Netflix. Uno de los juegos más famosos con este método es el videojuego Final Fantasy XIV (Square Enix, 2010).
- Finalmente, el método más clásico es el **pay-to-play**, que consiste en que, al comprar el juego, se tiene acceso directo a todo su contenido base sin restricciones. Cualquier típico juego de consola, como por ejemplo Uncharted 4 (Naughty Dog, 2016), es de este tipo.

En nuestro caso, aunque es cierto que la compensación monetaria no es el factor principal, consideramos que es importante respetar nuestro trabajo y dedicación, y por eso decidimos hacer el videojuego pay-to-play, poniéndole un precio inicial que consideremos justo. Para hacer que los jugadores tengan ganas de jugar a nuestro proyecto, haremos disponible una demo que incluirá una pequeña porción del videojuego, permitiendo que la gente pueda probar la experiencia de primera mano.

2.2.3. Matriz de competencia

Con toda la información presentada anteriormente, decidimos hacer una tabla comparando nuestro proyecto, Debt Trading, con los proyectos mencionados anteriormente en una serie de categorías, para tener una visión más clara de nuestra situación, como se muestra en la Figura 2.4.

	Tipo de cámara	**¿Mundo abierto?**	**Duración (horas)**	**Precio (euros)**
Death Stranding	3ª persona	Sí	40 ½	59.99
Gone Home	1ª persona	No	2	12.49
What Remains of Edith Finch	1ª persona	No	2	6.99
Debt Trading	3ª persona	Sí	5*	9.99*

Figura 2.4 Matriz de competencia de nuestro proyecto con la competencia

**Valores provisionales que podrían cambiar en la fecha final.*

Como se puede observar, Debt Trading tiene una mezcla de elementos de los otros videojuegos: es un videojuego de mundo abierto en tercera persona, pero su duración sería corta, para así no aburrir al jugador alargando la trama de forma innecesaria. Con esto en mente, su precio sería más bajo, dando así más ganas al jugador de probarlo.

2.3. Público objetivo i tipología de jugador

Dentro del mundo de los videojuegos los jugadores se suelen clasificar en cuatro tipos, siguiendo la conocida Taxonomía de Bartle (añadir link correctamente). Esta taxonomía define a los jugadores con dos ejes: actuar o interactuar, y jugadores o mundo.

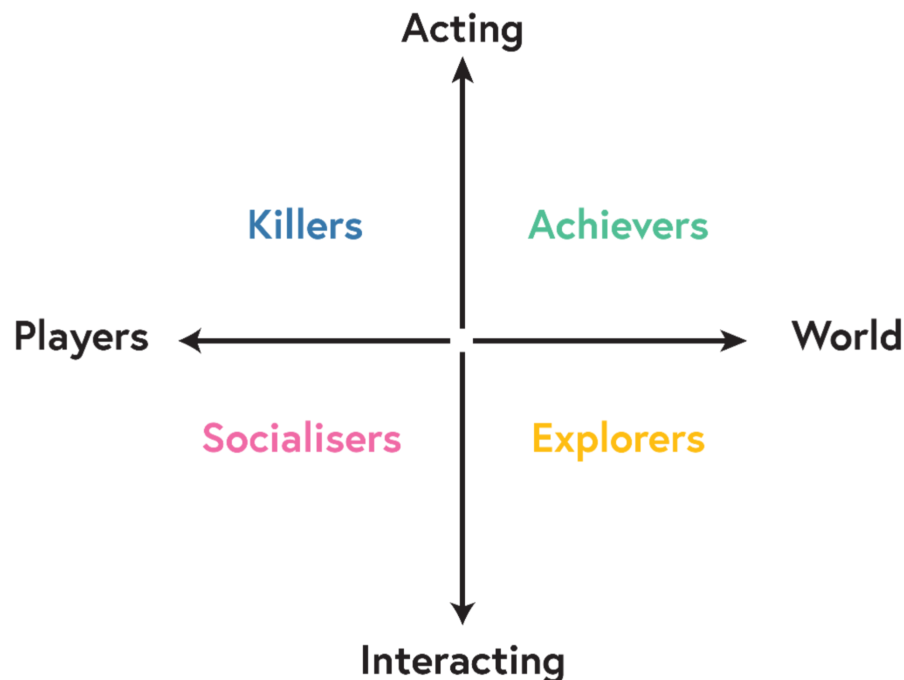

Figura 2.5 Ejemplo visual de los tipos de jugador según Bartle

Aunque esto se definió para los juegos MMO (como el mencionado anteriormente Final Fantasy XIV), también nos sirve para los juegos singleplayer: los **completadores** disfrutan, como su nombre indica, de completar todos los desafíos en el mundo, mientras que los **asesinos** disfrutan más al competir con otros jugadores, aunque sea de manera indirecta; en el lado opuesto, los **socializadores** disfrutan compartiendo historias y hablando del juego con otros jugadores, mientras que los **exploradores** simplemente disfrutan de explorar el mundo y descubrir sus secretos.

Nosotros nos enfocaremos en los exploradores; nuestro juego no tendrá gran complejidad mecánica comparado con otros, pero si crearemos un mundo interesante de explorar, con personajes diversos que harán que el jugador quiera saber más de ellos, ya sea completando sus misiones o explorando el mundo.

3. Planificación

3.1 Planificación previa

La planificación del proyecto empezó desde el primer día de su concepción, ya que sabíamos que la tarea que nos habíamos embarcado era de un calibre considerable y no podríamos improvisar por el camino. En la primera reunión del proyecto concretamos cuales serían los puntos claves a desarrollar para así centrar la mayoría de nuestros recursos en cumplirlos.

En la siguiente figura podemos observar el diagrama de Gantt que realizamos para este proyecto. Cada columna representa una semana de trabajo.

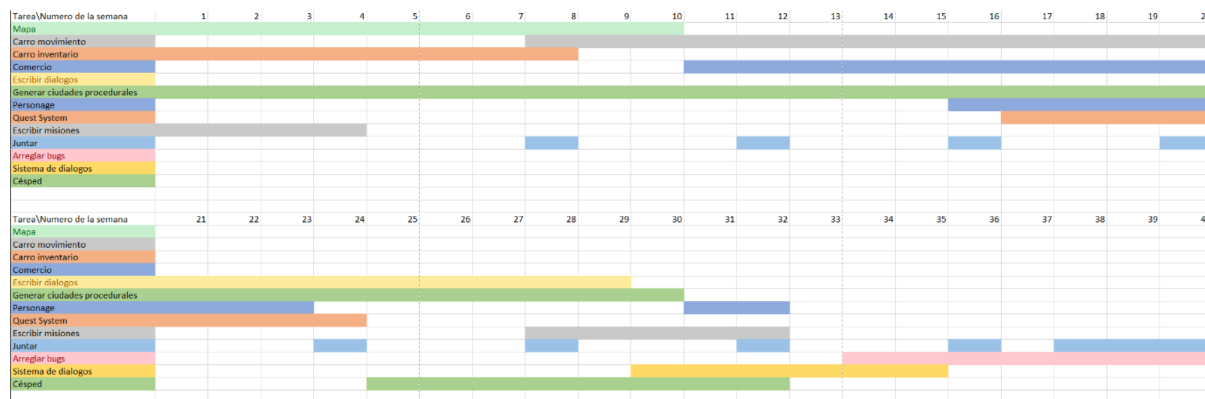

Figura 3.1 Diagrama de Gantt

3.2 Desarrollo de las tareas

Para el desarrollo de las tareas del TFG tuvimos dos controles de desarrollo: uno interno y otro externo.

El externo consistía en las reuniones bisemanales con nuestro tutor. Estas reuniones duraban entre una hora y hora y media. En ellas se formaba una especie de ágora virtual, por la COVID-19. En este espacio se compartía el progreso realizado por cada uno de los miembros del grupo, se explicaban trucos aprendidos y se resolvían dudas. Pero, más importante el tutor, nos transmitía si íbamos bien o no y en qué tarea deberíamos centrarnos después.

El interno consistía en repartirnos las tareas y registrar el progreso de éstas en la página *Hack&Plan*[4], ver Figura 3.2. En esta página pusimos todas las tareas a realizar y quien tenía que hacerlas, ya que no queríamos que en ningún momento nadie estuviera sin nada que hacer.

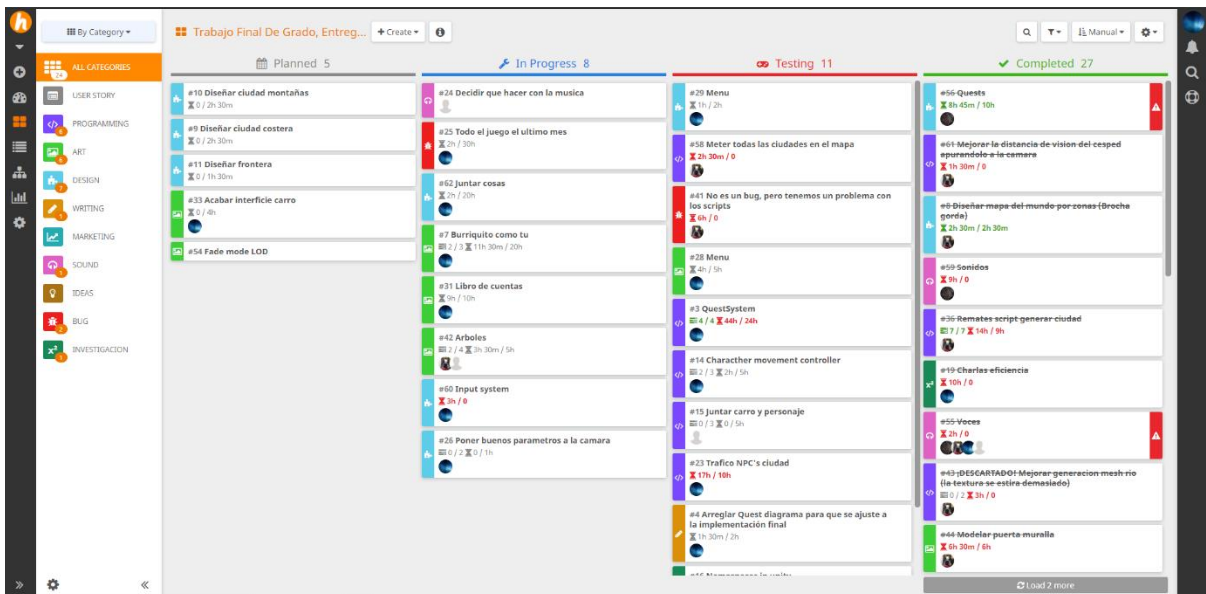

Figura 3.2 Hack&Plan del proyecto

A nivel individual cada miembro del grupo usó otras herramientas, como Trello o GoogleKeep, para llevar el control de los subpuntos de cada tarea principal.

3.3 Explicación de las herramientas utilizadas

3.3.1 Motor del videojuego – Unity

Hacer un videojuego con gráficos tridimensionales desde cero requiere una gran cantidad de trabajo, y hacerlo en menos de un año con un equipo de 4 personas es prácticamente imposible. Para eso elegimos usar Unity versión 2019.4.10f1 LTS, un motor de videojuegos con interfaz gráfica y motor de física incorporado. Gracias a algunos de los componentes que incluye Unity, como la propia física, el paquete de cámara Cinemachine o el render de texto TextMeshPro, pudimos centrarnos en desarrollar el videojuego y no invertir horas en crear un motor, cuando todo eso ya está hecho.

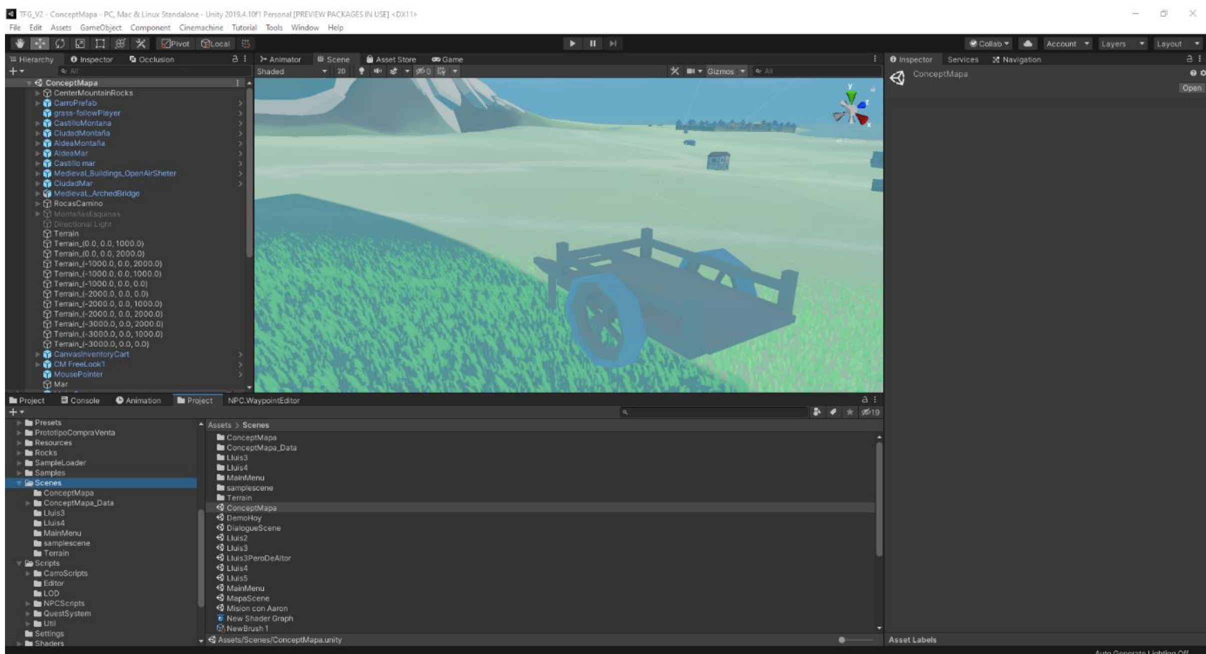

Figura 3.3 Interfaz de Unity

3.3.2 Editores de código

Aunque Unity tiene un sistema scripting visual de la misma manera lo tiene Unreal Engine, llamado Bolt, decidimos usar el sistema de scripting tradicional en C#, ya que consideramos que es más fácil de mantener a la larga. Al ser C# el lenguaje de programación de Unity, los editores más adecuados para este lenguaje de Microsoft son los propios editores de Microsoft: Visual Studio Comunity 2019 y Visual Studio Code. También, se trabajó con muchos archivos para guardar diferentes datos: archivos binarizados, objetos serializados, archivos JSON, … Para comprobarlos de manera rápida sin tener que abrir un IDE entero para ellos, muchas veces usamos el Notepat ++.

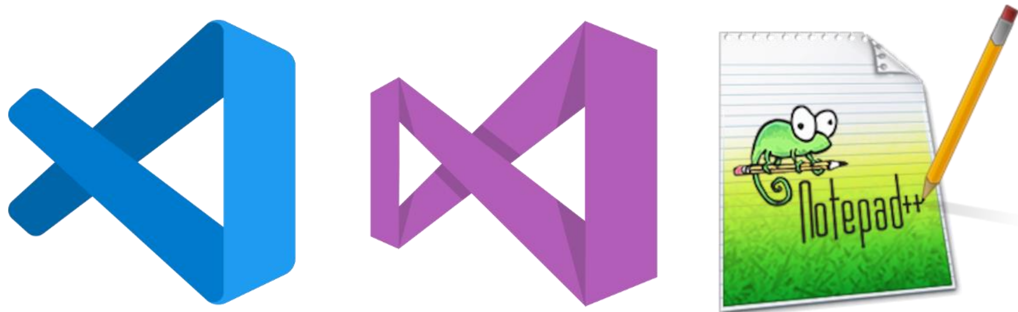

Figura 3.4 Editores usados

3.3.3 Software artístico

Incluso encontrando muchos de los assets que necesitábamos por internet, tanto 2D como 3D, nos hizo falta crear algunos originales o modificar existentes. Los principales programas que usamos para esta parte del proyecto son: Blender, Gimp y Inkscape.

Blender lo usamos principalmente para crear y modificar assets en 3D. Este es el programa con el que más assets propios hemos creado, al necesitar muchos elementos específicos, como por ejemplo las tejas de los tejados, que al ser creados proceduralmente, creamos un modelo propio de teja para que fuera más flexible.

Inkscape es nuestro editor de imágenes vectoriales predilecto. Es una de las herramientas artísticas con las que más cómodos nos sentimos, aunque en este proyecto, debido a su naturaleza tridimensional, no fue muy usado aparte de para la interfaz.

Gimp fue usado más para retocar imágenes que para crearlas. Esos retoques que hicimos no solían ser para assets del juego, la mayoría eran para material externo al juego (logos, banner, ícono de escritorio, ...). Pero para algun que otro asset del juego sí que se usó, por ejemplo, en la pantalla del menú. En ella el fondo representa que es la ciudad del mar, pero usar todo el modelo 3D para el menú nos parecía un coste computacional demasiado alto, así que hicimos una captura de la ciudad desde un punto lejano. La recortamos y retocamos un poco y la pusimos de fondo del menú principal del juego.

No todo el apartado artístico es visual, también hay audio. Para eso usamos la herramienta open source Audacity. En este proyecto, aparte de ser usada para diferentes archivos de audio del juego, fue más usada para realizar el “doblaje” de voz de los personajes.

3.3.4 Software para documentación

Para realizar este documento de texto decidimos utilizar Microsoft Word, tanto su versión de escritorio de Windows 10 como su versión para navegador, ya que contábamos con licencia de estudiante proporcionada por la EPS (Escuela politécnica superior) de la UdG (Universidad de Girona).

Ya que disponíamos de ella, también aprovechamos el Microsoft Exel. Esta herramienta de editor de CSV lo usamos para guardar datos que recolectamos durante las pruebas de implementación de algunas partes del juego, generar gráficas a partir de éstos, y realizar algunas planificaciones.

Por último, pero no menos importante, para realizar la documentación usamos el programa de diagramas vectoriales Draw.io. Esta herramienta fue usada principalmente en el apartado 6 de implementación.

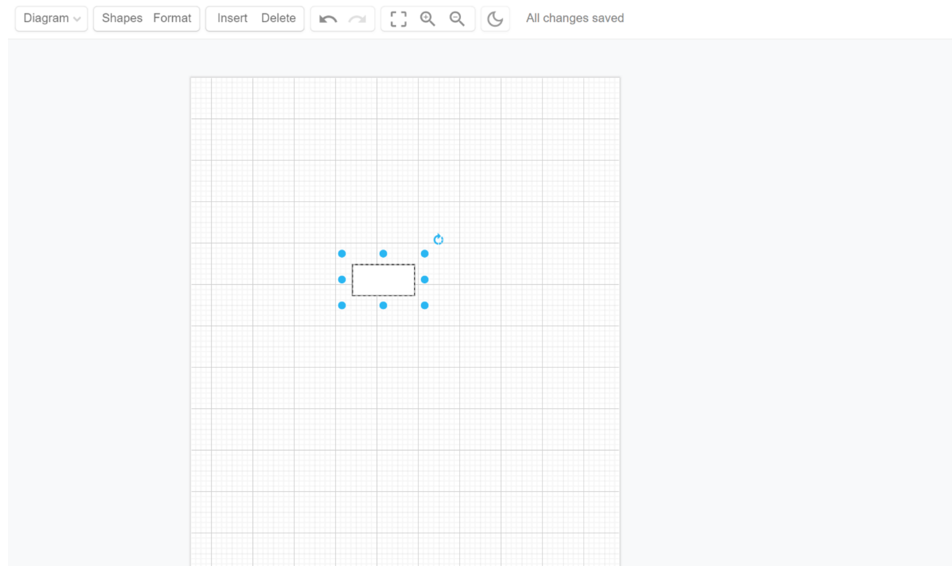

Figura 3.5 Captura de draw.io

3.3.5 Otros

En este apartado se explicarán otras herramientas que, debido a su condición, versatilidad o su uso nos es muy difícil de categorizarlas en otros grupos.

Al ser este un proyecto grande en tamaño y con bastante gente trabajando en el a la vez (para ser un trabajo de final de grado), no podíamos ir trabajando de cualquier manera y luego juntarlo todo rápido y mal. Necesitábamos mantener en cierto control y registro de los cambios realizados. Para eso decidimos usar el estándar de la industria que es Git junto a otras herramientas que trabajan con él. Esas otras herramientas fueron: GitHub para poder alojar el proyecto y acceder a él de forma remota, GitHub desktop fue muy usado por algunos miembros del equipo para poder solucionar conflictos a la hora de juntar ramas de Git y ver los diferentes cambios realizados de forma más visual. Por último, al final del proyecto también se usó GitKraken para, mediante el gráfico propio de ese programa, saber cuál es la rama de Git más avanzada y coger el progreso de esa.

Durante el desarollo del proyecto, para compartir diferentes archivos y más importante, que esté todo en el mismo sitio, usamos la aplicación de almacenage en la nuve Google Drive. Tambien, para hacer ciertos drafts rápidos de las mecánicas, o para realizar los dialogos, usamos el editor de texto Google docs.

En este trabajo, para comunicarnos entre nosotros, usamos Discord y WhatsApp. Para realizar las reunions con nuestro tutor del TFG, usamos el programa de Google de videollamadas llamado Google Meet.

4. Marco de trabajo y conceptos previos

A pesar de que lo hayamos mencionado antes, todavía no hemos explicado en qué se basa un juego *Strand Game*. Como dice Kojima en su entrevista con Gamereactor[5]: *"Social Strand System* (Strand Game) no es un tanto un género por si mismo, si no una manera más precisa de describir Death Stranding como una nueva experiencia".

También explica como sus trabajadores dicen que "sólo moverse por el mundo ya es divertido". Esto es porque quiere que el jugador "escoja su camino por el juego".

Cuando hablamos de que Debt Trading es un Strand Game, es porque busca fomentar la exploración y no poner ningún tipo de camino preestablecido al jugador; que este pueda escoger incluso que misiones cumplir y cuáles no, y en base a esto que la experiencia de juego sea diferente.

5. Diseño del videojuego

5.1 Mecánicas

5.1.1. Espacio del juego

A la hora de diseñar el mapa, sabíamos bien que este tenía que estar diseñado acorde a las necesidades del juego. Debía ser un mapa fácil de navegar, sin riesgo de perderse, un espacio abierto que invitase a la exploración por parte del jugador, un lugar donde poder perderse para luego volver a encontrar el camino.

El mapa está dividido en dos regiones o reinos, cómo se puede ver en la Figura 5.1.1. El reino del norte o el de la montaña, y el reino del sur o del mar. Cada uno de estos reinos tiene sus propias características para diferenciarlas entre ellas. El reino de la montaña es más rocoso y escarpado, con piedras por el suelo y montañas delimitando su perímetro. El reino del mar, en cambio, está delimitado por mar, como su nombre indica; es un terreno más llano, con un extenso prado al oeste de la región, y un colorido bosque al norte de la capital.

Figura 5.1.1 Mapa del mundo con leyendas

Cada uno de estos reinos tiene una ciudad y un pueblo, con sus diferencias territoriales.

La región de la costa tiene tejados de color azul, a juego con el mar que los rodea. También cuenta con casas de todo tipo, de madera y/o roca, de no más de dos pisos. Al ser una ciudad donde hay mucho tránsito de mercaderes, en el centro existe un mercado hecho únicamente de madera. La ciudad entera está protegida por robustas murallas de piedra.

Por el contrario, el norte sólo tiene casas de piedra, con tejados marrones y rojizos, que pueden llegar a los 3 pisos. La única madera que encontramos es en la ciudad, en las picas que forman la muralla de inspiración vikinga que la rodea. La muralla es de madera debido a que, al estar en la cima de una colina, y ser más fácil defenderse por la propia naturaleza del terreno, decidimos que no tenía sentido poner murallas tan gruesas, usando así el espacio reducido que deja la cuenca de la montaña en construir las casas y el castillo.

En el centro de los dos reinos se encuentra una gran montaña que sirve tanto para separar a las dos ciudades cómo para servir de guía al jugador, pues es un hito natural que se puede ver desde todos los puntos del mapa. Esta montaña, a diferencia de las perimetrales, esta recubierta de roca, pues queríamos darle un toque distintivo teniendo en cuenta su importancia.

Al pie de esta montaña se juntan dos ríos, uno que viene de la cordillera del norte y otro que viene de la del oeste. Estos afluentes, más la zona rocosa al sur del pueblo de la montaña, sirven de frontera natural entre los dos reinos (la línea discontinua en la Figura 5.1.1); delimitando así el territorio de cada reino de manera orgánica y sin mucho tiempo de desarrollo.

La zona rocosa antes mencionada es uno de los varios biomas que tenemos en nuestro juego. Decidimos separar el terreno en zonas características para evitar así la monotonía e incentivar la exploración. En total existen 4 biomas: el rocoso, la pradera, el bosque y la zona montañosa.

El bioma rocoso es la zona que se encuentra al sur del pueblo de la montaña. Es un prado que está cubierto por peñascos y piedras de gran tamaño, imposibilitando el paso al jugador. Al sur de este se encuentra el bosque. Está repleto de árboles de varios colores, predominados por tonos naranjas y amarillos, pero con toques de verde o azul. Al sur de este bosque encontramos la capital del mar, y a su este un prado masivo que alcanza hasta el final del terreno y se extiende por el costado oeste hasta la capital de la montaña. Este prado tiene unas ruinas antiguas, las cuales están a la puerta de la zona montañosa. Esta es una zona donde hay un camino abandonado que recorre las montañas, uniendo los dos reinos. Hay un poco de hierba y árboles rosas que van disminuyendo a medida que se gana altura para dar paso a rocas varias.

5.1.2. La mecánica del juego, los retos que se tienen que afrontar y las acciones posibles

Al ser nuestro juego un strand game, género inventado por el diseñador Hideo Kojima, la mecánica principal se basa en mover objetos y paquetes entre distintas localizaciones, relegando mayor importancia al mundo del juego. Desde un comienzo sabíamos que no nos era posible hacer un sistema de exploración tan grande como el de Death Stranding, con escaleras y puentes a construir; por lo que decidimos cambiar un poco el enfoque. Nuestros caminos serían fijos e inamovibles, pero estarían indocumentados. Sería el trabajo del jugador dibujar su propio mapa y saber manejarse con él.

Para darle un reto al hecho de conseguir objetos, decidimos que el jugador tuviese que comerciar para conseguirlos. Con esto no sólo conseguimos añadir un reto al hecho de conseguir los objetos, si no que darle también una capa de profundidad al tener que gestionar la economía del propio jugador.

Con el objetivo de darle complejidad a mover recursos por el mapa, decidimos que se tuviesen que colocar los objetos a mano en el carro; mecánica inspirada en los juegos Resident Evil. De esta manera forzamos al jugador a que no pueda llevar demasiados objetos consigo, fomentando así la compraventa.

A todo esto, se le suman las acciones básicas, tales como moverse, conducir el carro, interactuar con NPCs y recoger objetos.

5.1.3. Objetos, recursos e interacciones que puede hacer el jugador

Al ser el nuestro un juego basado en el intercambio de bienes, hay una considerable cantidad de objetos, cada uno con sus varias instancias.

Personaje

Este es el personaje del juego, el que el jugador controla. Puede moverse e interactuar con entidades, por lo que guarda su posición y orientación. También guarda un estado, el cual representa si está corriendo, andando, cogiendo algo…

Carro

Es un carro que puede controlar el jugador para desplazarse por el mapa y llevar objetos en él. Este tiene una posición y rotación, una velocidad a la que se esta desplazando y las diferentes velocidades que puede ir dependiendo de su estado. A parte de esto, también tiene un inventario de objetos.

NPC

Los NPC (Non Player Character o Personaje No Jugador) son las entidades que deambulan por las ciudades para darles vida. Al ser su función seguir una ruta, guardan esta misma y la velocidad a la que se desplazan, aparte de la posición y rotación, que gestiona el propio Unity. La gestión de evitar obstáculos se hace en runtime con el componente NavMeshAgent de Unity.

Mercader Personaje

Un componente que permite el intercambio de objetos al jugador. Este tiene el dinero y el inventario del protagonista.

Mercader NPC

Los mercaderes con los que podemos interactuar. Cada uno tiene un dinero y un inventario, además de una lista con los objetos en los que está interesado.

Item

Los objetos con los que se intercambia en la compraventa. Cada uno tiene un precio mínimo, máximo, un sprite y un nombre.

Objeto

Estos son los objetos que encuentras en el mapa y se pueden cargar en el carro. Pueden representar los ítems. También pueden ser cogidos para su transporte. Los objetos guardan la posición, rotación, nombre y estado.

Puerta

Las puertas para entrar en las ciudades. Guardan una posición y una rotación; además de un estado indicando si están abiertas o cerradas. Estas se abren automáticamente cuando el jugador se acerca.

Sticker mapa

Los sticker del mapa son dibujos en los cuales se guarda únicamente la posición a la hora de colocarlo en el mapa. Entre ellos están las montañas y los árboles. Estos guardan únicamente el tipo y la posición.

Points mapa

Estos son los dibujos del mapa que se generan en runtime partiendo de una serie de puntos. Estos guardan un material para su reconstrucción, además de los puntos que definen el trazo del jugador. Estos son ríos y lagos.

5.1.4. Economía interna del juego

En nuestro juego, la economía va muy ligada al gameplay, pues al ser un juego sobre el comercio, tenía sentido que este fuese la principal manera de interactuar con los recursos.

Sources

Coger objetos

Esta es la forma más básica de obtener recursos u objetos con los cuales después hacer negocios. Son objetos sin un gran valor, pero dado su fácil obtención, son idóneos para empezar a conseguir dinero para luego hacer intercambios, o incluso superar misiones.

En las misiones más sencillas, los NPC le dirán al jugador que necesitan un objeto que se encuentra en cierta localización. El jugador podrá ir ahí y recogerlo para superar la misión.

Recibir objetos

Esto sucede cómo recompensa de misiones. Una vez se haya cumplido una, el NPC puede regalar al jugador algo cómo muestra de gratitud.

Trader

Comerciar

Comerciar es la manera predilecta del juego de obtener recursos. El jugador puede comprar o vender los objetos adquiridos. El precio de cada transacción no es fijo, está basado en función a un minijuego que tiene que superar el jugador. Dependiendo del resultado de éste, el jugador tendrá beneficios o pérdidas a la hora de finalizar el comercio.

5.1.5. Estudio de los diferentes niveles del juego

Al ser nuestro juego un mundo abierto, carece de niveles. Una vez se acaba el tutorial, ya se puede desplazar por todo el mapa sin limitación alguna.

A pesar de que el juego esté dividido en dos reinos, y que en cierto punto de la historia las fronteras entre estos se cierren, siempre se puede mover entre ellos: cuando no es por los caminos ortodoxos, por caminos abandonados.

5.1.6. Interfaces

Debt Trading es un juego de mecánicas muy simples, por lo que decidimos mantener las interfaces de este modo, haciendo estas diegéticas siempre que era posible.

Al ser el escenario uno medieval, todas las interfaces diegéticas están inspiradas por objetos de esta época. El listado de misiones es una libreta (libro de cuentas), por ejemplo, y el mapa es un trozo de papel en el cual el jugador dibuja.

5.2 Estudio y diseño de personajes

En este apartado explicaremos a fondo el uso de los personajes creados para el prototipo de Debt Trading, centrándonos en nuestro planteamiento y mentalidad a la hora de diseñarlos y darles una función en el mundo de juego.

5.2.1. Peso narrativo de los personajes

Desde el principio del desarrollo los personajes de Debt Trading han sido un aspecto muy importante para nosotros, ya que son ellos los que aportarán la mayoría del peso narrativo durante el transcurso de la historia.

Los personajes mostrarán, a través de su aspecto visual, las misiones que ofrecen y su forma de hablar, diferentes aspectos de las sociedades en las que se encuentran y viven, haciendo que el jugador pueda ver diferentes puntos de vista y como estos evolucionan a lo largo de la semana que tiene lugar el juego.

Con esta información, clasificaremos a los personajes en tres grupos diferentes, mostrando los niveles de peso narrativo:

- Los **personajes principales** son Mena y los dos reyes, Noel y Azariel. Tienen el mayor peso narrativo y, quitando algún otro personaje, además de ser obligatorio interactuar con ellos para ir avanzando en la historia.
- Los **personajes secundarios** son habitantes de los dos reinos: de Esea tenemos a Rolan, Ian, Donald, Julian y Leona; de Lice tenemos a Koel y Diana. La mayoría son opcionales (exceptuando a Julian) y sus misiones son secundarias, pero gracias a su personalidad podremos saber más de este mundo y cómo se siente la gente a diferentes niveles de la escala social en cada reino.
- Finalmente, tenemos un tercer grupo de personajes que incluye a los vendedores, a los viandantes de cada reino y a aquellos personajes que aparecen en una misión, pero que no consideramos importantes (por ejemplo, un guarda que sólo aparece en un punto específico de la historia). Esta gente no tiene una personalidad marcada y simplemente existen, pero sirven para dar la ilusión de que el mundo se encuentra vivo, y de que los personajes con los que interactuamos son una pequeña parte de este mundo.

5.2.2. Los personajes como base de la jugabilidad

Los personajes funcionan como uno de los ejes centrales de nuestro proyecto. En términos de jugabilidad, son ellos los que inician y avanzan las misiones, ya sean de tipo principal o secundario. Es gracias a estas interacciones que el jugador tiene motivos para explorar el mundo y aprender más del mundo a través del diálogo.

No sólo eso, sino que aquellos personajes que podríamos considerar poco importantes también tienen peso en la jugabilidad: los vendedores, uno por ciudad, permiten que el jugador pueda interactuar con el sistema de compra y venta, y la cantidad de viandantes que se encuentran en los pueblos servirán para indicar de manera visual cual es la situación según avance el tiempo.

5.2.3. Estilo artístico (condicionado por la jugabilidad)

El estilo artístico de los personajes era un aspecto importante para nosotros: no sólo tenía que casar con el resto del aspecto visual del proyecto, sino que además tenía que hacerlo cuando ningún miembro del equipo de desarrollo nos veíamos capaces de poder llevar a cabo esta tarea.

Después de barajar diferentes opciones, encontramos en la página web itch.io unos *assets* gratuitos que pudimos comprar, que traían unos personajes diseñados de base: estos personajes, además,

tenían un estilo artístico lowpoly con texturizado simple, lo que casaba con el arte de los edificios y de nuestro juego hasta el momento.

Con estos assets como base, diseñamos diseño unos personajes básicos con diferentes aspectos principales, con versión hombre y mujer (persona mayor, guardia, viandante básico, etc.), y después se creó un script llamado *MaterialEditor*, que permitió que pudiéramos alterar el color de los diferentes aspectos de cada personaje.

Gracias a esto, pudimos crear personajes que casaran con el estilo artístico del proyecto y tuvieran suficiente variedad.Gracias a esto, pudimos crear personajes que casaran con el estilo artístico del proyecto y tuvieran suficiente variedad, como se puede observar en la Figura 5.2.1.

Figura 5.2.1 Ejemplo de dos personajes con colores diferentes

5.2.4. Coherencia, características físicas y psicología

Siendo un equipo de desarrollo pequeño, hacer una gran cantidad de personajes era una tarea compleja, así que nos centramos en hacer un grupo más reducido, pero dotando a cada uno de una personalidad marcada. Con el tiempo limitado, en esta demo del proyecto conseguimos hacer que tanto los personajes principales como los secundarios tuvieran un estilo de diálogo característico y "gruñidos" que sonarían diferente según su estado emocional, creados por nosotros mismos.

A continuación, tenemos una descripción corta de los personajes principales y secundarios.

- **Mena** es el protagonista de nuestra historia: un joven adolescente que tiene buenas intenciones, y que quiere rescatar a su padre. Estos elementos básicos permiten que sea un personaje a la vez que se mantiene mayoritariamente neutro, permitiendo que ninguna opción de diálogo suene fuera de lugar.
- **Azariel** es el rey de Esea, la ciudad del mar. Es directo, actúa tanto para conseguir sus objetivos como para molestar al resto, y no tiene preocupaciones de lo que piense el pueblo, pero intenta hacer de Esea un lugar digno de vivir.

- **Noel** es el rey de Lice, la ciudad de las montañas; aunque amable a primera vista, las personas que le conocen de verdad saben que para Noel todo detalle que conozca es información que puede usar a su favor. Sabe mucho más de lo que insinúa.
- **Rolan** es un hombre mayor que trabaja en Esea en todo tipo de trabajos que requieran habilidades manuales, como reparar carros y crear armaduras; se le nota cansado, pero es un buen hombre con un gran corazón.
- **Ian** es un hombre joven que el jugador encuentra encerrado, a punto de ser ejecutado. En la situación que se encuentra está asustado, pero su característica habla, llena de insultos y gritar a los cuatro vientos, se mantiene se mantiene.
- **Donald** es el padre de Ian, un borracho que se preocupa por poco más que el alcohol que tenga en la mano. No duda en amenazar a todo y todos, y sólo les tiene miedo a las consecuencias de sus actos.
- **Leona** es una soldado que vive en Esea. Es seria y pensativa, y desde que conoció a Diana, intenta tener detalles románticos con ella siempre que puede, ya que apenas pueden verse. Es seria y pensativa, y, desde que conoció a Diana, intenta tener detalles románticos con ella siempre que puede, ya que apenas pueden verse.
- **Julian** es un misterioso hombre que pertenece a un grupo de insurgentes de Esea que quiere acabar con Rolan. Habla en mensajes cortos y deja muchas cosas sin decir, pero suena seguro de sí mismo y que conoce la situación al pie de la letra.
- **Diana** es una mujer que vive en Lice. Es alegre y no duda en echar una mano siempre que puede, además de estar preocupada por la gran carga de trabajo que Leona tiene, haciendo escapadas siempre que puede para verla.
- **Koel** es un viejo vendedor de Lice que se dedica a coleccionar monedas. Es un erudito con mucha información sobre el mundo y como ha ido cambiando a lo largo de los años, pero que tiene tendencia a irse por las ramas y mostrar una clara falta de interés en temas que no le conciernen.

5.2.5. Lenguaje corporal y gestual, movimiento

Todos los personajes de Debt Trading son humanos, y por lo tanto ninguno de ellos tiene movimientos destacables, sino que reutilizamos las mismas animaciones entre ellos. No solo hacer animaciones únicas hubiera sido una cantidad de trabajo inasequible para este prototipo, sino que queríamos que todos los personajes, incluso aquellos que consideramos importantes, tuvieran el mismo nivel que los viandantes, para hacer que nadie parezca superior para el jugador destaque por encima.

5.2.6. Objetos y vestimenta que caracterizan a los personajes

Como se ha explicado en el apartado anterior, la falta de recursos ha sido una consideración importante a la hora de diseñar y crear el proyecto. Es por eso que todos los personajes tienen la misma vestimenta, simplemente con diferentes colores. La única excepción son los guardias de las dos ciudades, que llevan armadura para indicar su trabajo.

Además, tanto Noel y Azariel tienen un trono ordenado con cristales preciosos que indican su estatus social de manera directa; en el caso de Azariel, los cristales preciosos forman parte de la trama principal, cuando Julian pide a Mena que robe uno en la misión *The City's Hidden Side*.

5.3 Narrativa

La narrativa ha sido una parte importante de Debt Trading desde el principio, queriendo hacer una historia que tratara la necesidad del pueblo de unirse y la tiranía de aquellos en el poder. En esta sección destacaremos las bases como la sinopsis del proyecto y el trasfondo del mundo de juego, la mezcla de narrativa con jugabilidad y el diseño de misiones.

5.3.1. Sinopsis

En un mundo fantástico de estilo medieval, un padre mercader y su hijo, llamado Mena, llegan a una nueva ciudad después de un largo viaje en barco en busca de una mejor vida.

Por desgracia, poco después de bajar del barco, los dos hombres se cruzan con el rey de la ciudad, Azariel, quien abusa de su estatus y decide hacer al padre su esclavo propio sin ningún motivo aparente. Mena discute con el rey, y eventualmente este decide hacerle una oferta: si consigue una grandiosa suma de dinero, liberará a su padre.

Con esta gran tarea sobre sus hombros, Mena se embarcará en una pequeña aventura, sin ser consciente de que sus acciones a lo largo de la semana afectarán no solo su destino, sino el de todas las personas de dos reinos a punto de enfrentarse en una guerra.

5.3.2. Backstory

Cien años antes del comienzo del juego, hubo una gran guerra que afectó a muchísimas partes del mundo. Gente de diversos países decidieron irse a la zona donde tiene lugar el juego, ya que estaba inhabitada, haciendo que con el paso del tiempo se fueran creando dos comunidades principales, una en las montañas y otra cerca del mar, que se encontraban en buenos términos.

Cincuenta años antes de la llegada de Mena a la ciudad, dos reinos intentaron hacer que esta pequeña zona formara parte oficial de sus reinos de manera diplomática, pero ninguna llego a controlar las dos ciudades: el reino que se encontraba ya en el continente consiguió controlar la ciudad de las montañas, mientras que un reino de otro continente consiguió añadirse la ciudad cerca del mar.

Para seguir con las negociaciones mientras se recuperaban de la guerra, unos diez años antes del comienzo del juego, cada reino envió a un noble, coronándolos reyes de sus respectivas ciudades y dándoles plenos poderes sobre estas, con la idea que funcionaran como agentes diplomáticos y convencieran a la otra ciudad de unirse a ellos. Los dos “reyes” cambiaron los nombres de las ciudades a **Lice** (la ciudad de las montañas) y **Esea** (la ciudad del mar).

Pero en el tiempo que llevan la situación no ha mejorado, sino que ha empeorado: Noel y Azariel, los dos nobles, no se soportan entre ellos, y cuando llega el jugador las tensiones están tan altas que la idea de una guerra está muy presente en las mentes de los dos.

5.3.4. Narrative devices

Al ser un mundo tan extenso y con tanta historia, decidimos que Debt Trading contara su historia a cada momento posible, utilizando todos los recursos que nos fueran disponibles. De esta manera, podemos hablar de *active devices* y *pasive devices*.

- Los **active devices** (o elementos activos) son aquellos que tienen lugar en tiempo real mientras el jugador avanza en el juego. Estos son principalmente las misiones, que incluyen no sólo los diálogos que tienen lugar, sino los efectos que causan sus resoluciones (por ejemplo, elegir a quién salvar en una misión).
- Los **pasive devices** (o elementos pasivos), en cambio, no tienen lugar en el tiempo de juego, sino que en el mundo ocurrieron hace tiempo y el jugador puede encontrarlos o saber más de ellos. Objetos como monedas antiguas, libros que leer, la posición de ruinas y restos de casas explican de manera indirecta lo que fue ocurriendo en esa zona.

5.3.5. Tensión dramática y tensión de jugabilidad

En Debt Trading la trama tiene lugar en un período de una semana, donde el último día tiene lugar una guerra entre los dos reinos. Este *setting* hizo que pensáramos muy bien en cómo transmitir la situación no solo de Mena, sino de los residentes de la zona.

En primer lugar, la mayoría de las misiones que recibirá el jugador al comienzo de la trama (*Mother's Favorites*, *A Lover's Words*, etc.) serán de un aire más calmado, y los personajes no tendrán mucha urgencia. Esto irá cambiando conforme pasan los días, ya que todo el mundo podrá notar que se avecina algo peligroso, y mientras el jugador intenta salvar a su padre, los habitantes de los dos reinos le pedirán ayuda con situaciones más complejas y peligrosas (*Sail Me to the Moon*).

En segundo lugar, el propio mundo de juego cambiará conforme avance la historia: al principio las ciudades estarán llenas de vida y aunque habrá presencia de guardias de los dos bandos, no será algo extremo; conforme pasen los días habrá menos gente en las calles y ambos reinos bloquearán más las entradas y tendrán más guardias patrullando tanto dentro de los muros como a los alrededores.

Estos dos elementos nos ayudarán a ir mostrando en tiempo real la situación cambiante del mundo, y como cada vez más los habitantes temen el futuro incierto en el que se encuentran, dándoles una sensación de estar vivos y de que no son simplemente "los personajes que te dan misiones".

5.3.6. Plot points y estructura narrativa

Aunque pueda parecer muy compleja a primera vista, la trama de nuestro proyecto es bastante simple, con pocos puntos definitivos que realmente indiquen un cambio en la historia: la **llegada y encarcelamiento del padre de Mena**, la **revelación de que el reino de Lice va a invadir Esea**, y finalmente el resultado de la última misión, que será cuando estalle la guerra.

Estos plot points los utilizamos a la hora de crear una típica estructura en tres actos:

- El **planteamiento** de la historia es la llegada Mena y su padre a Esea, cuando el rey Azariel convierte a este último en su esclavo. Este evento es el que da verdadero comienzo al juego, y hace que Mena empiece a explorar el mundo y conocer a los habitantes de los dos reinos, incluyendo a Noel, el rey de Lice.
- El **nudo** ocurre con el segundo plot point. Pasados unos días, Noel implica que ya ha llegado al límite de su paciencia, y que va a unir Esea por la fuerza a su reino, con una invasión directa si hace falta. A partir de aquí el mundo se volverá más tenso, y aún a falta de declaración de guerra directa, el resto de personajes podrán intuir que algo va a ocurrir. Mientras van pasando los días, los personajes irán actuando conforme la situación, con peticiones de ayuda más serias y con menos ganas de perder el tiempo.
- Finalmente, el **desenlace** tendrá lugar el último día de la semana, cuando tenga lugar la invasión de Lice a Esea. Será en este punto cuando las acciones del jugador desencadenen en un resultado u otro, cerrando la historia de Mena, su padre, y de los dos reinos.

La trama de Debt Trading tiene lugar a lo largo de siete días; en cada uno de ellos, exceptuando el primero y el último, el jugador tendrá una serie de misiones principales y secundarias que hacer. Las misiones principales serán obligatorias; las secundarias no lo serán y, generalmente, solo se podrán completar durante un día o dos.

Todas las tramas seguirán una estructura lineal clara, permitiéndonos crear una narrativa interesante y enfocada en los elementos más interesantes, con la única excepción de la misión "*A Worthy Collection*", que permitirá que, según la moneda que el jugador encuentre por el mundo y devuelva al personaje, obtenga un diálogo diferente explicando las peculiaridades de la moneda.

5.3.7. Mecanismos para hacer avanzar la trama

Para mantener cierto control sobre la historia y cómo avanza, solamente las misiones principales avanzarán la historia principal de manera exclusiva, mientras que las secundarias avanzarán su historia particular si dura más de una misión (por ejemplo, si ayudas a dos chicas en el primer día, al tercero te volverán a pedir ayuda).

5.3.8. Granularidad

La frecuencia de los elementos narrativos se basa principalmente en los diálogos que el jugador puede entablar, así que será constante mientras el jugador complete misiones. En los momentos de exploración serán puntos específicos del mapa o la localización de objetos los que den cierta carga narrativa, pero a menor medida, para permitir al jugador absorber la información del mundo de juego de manera pasiva.

5.3.9. Intercalar los elementos de la narración en la estructura del juego

Los elementos de narración están estrechamente intercalados con la estructura del juego para así mantener una cohesión completa del proyecto. Las misiones que hemos creado fuerzan al jugador a explorar el mundo y moverse por él con el carro, a interactuar con los diferentes personajes y hablar con ellos, a utilizar los sistemas de compra de objetos y a manejar los sistemas de inventario.

5.3.10. Disparadores (triggers)

Los únicos *triggers* que impactan la narración serán los del diálogo: el jugador tendrá que elegir con quien habla, y si tiene la opción, podrá responder de una lista de opciones, haciendo que la misión avance de diferente manera o que consiga información extra que le permita sumergirse más en el mundo de juego.

5.4. Mundos de juego

5.4.1. Dimensión física

La dimensión física de nuestro videojuego es muy importante debido a la importancia de la exploración. Por ello el juego dispone de una dimensión física que sería el equivalente a 12 kilómetros cuadrados.

El terreno está delimitado por montañas en norte y el mar en el sur, delimitando así la dimensión física. Esta dimensión física se podría dividir en subdimensiones, entornos geométricos definidos en el juego que tienen cierta importancia. Estas subdivisiones serían:

- **Ciudad del mar:** Ciudad medieval con más de 150 casas y un castillo, rodeada por una muralla que la protege. Hay un rio que atraviesa la ciudad con un único puente que conecta las dos mitades de la ciudad. También hay un foro/mercado de madera en el centro de la ciudad en donde los comerciantes se dedican a vender sus productos. La muralla exterior tiene dos puertas con barreras de metal que impiden acceso a quien no tenga el permiso del rey. Ver Figura 5.4.1.

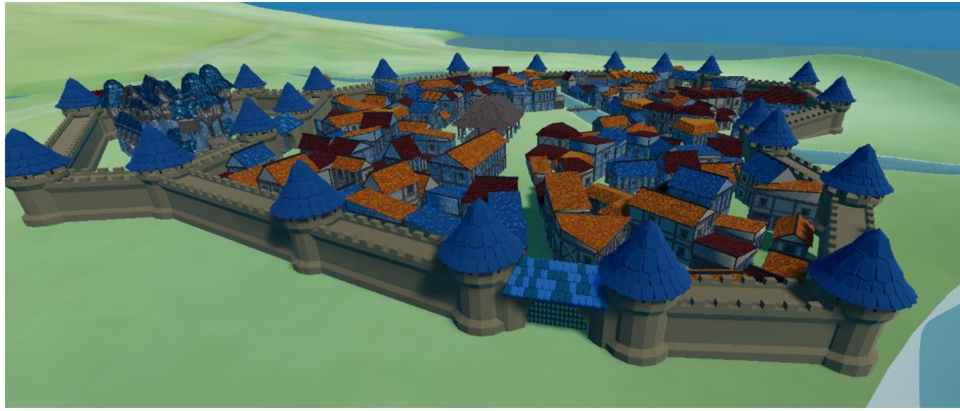

Figura 5.4.1 Ciudad del mar (Esea)

- **Aldea del mar**: Pequeño pueblo de menos de 30 casas, cercano a la ciudad del mar y a las ruinas, donde todas las casas son de madera. Esta aldea no tiene ninguna estructura de protección. Ver Figura 5.4.2.

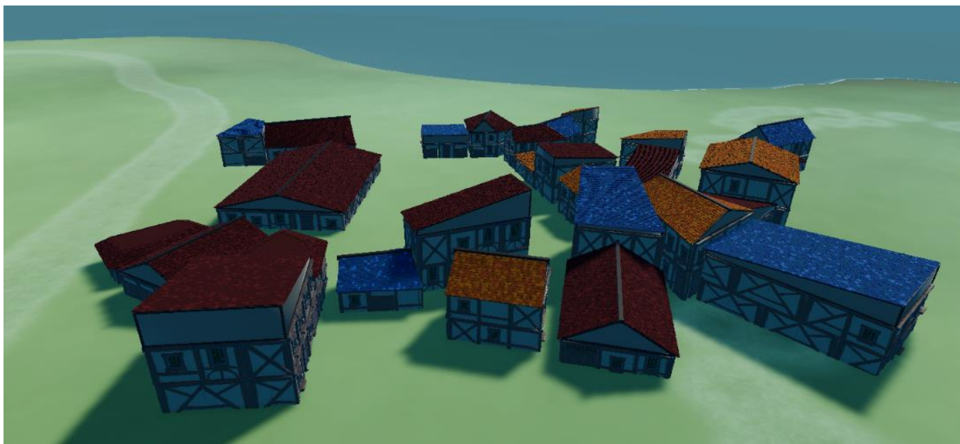

Figura 5.4.2 Aldea del mar

- **Ruinas:** Al oeste del reino del mar se encuentran unas ruinas de piedra que tienen un monolito en el centro y está rodeado de pilares de piedra. Estas ruinas hacen referencia a una cultura ya inexistente. Ver Figura 5.4.3.

Figura 5.4.3 Ruinas antiguas

- **Mar:** Casi todo el reino del mar está rodeado por el mar, con playas de césped que se extienden por todo el sur. Ver Figura 5.4.4.

Figura 5.4.4 Vista del mar desde arriba

- **Monte central:** Entre los dos reinos hay un gran monte de piedra escarpado que los divide y hace más difícil la visión e interacción entre ellos, también servirá como el principal punto de referencia del jugador para orientarse. Ver Figura 5.4.5

Figura 5.4.5 Mone central

- **Ríos:** Para hacer más clara la división entre los reinos hay dos ríos que dividen sus dominios, ambos ríos se unen en uno que es el que acaba atravesando a la ciudad del mar. Los ríos también sirven para hacer más difícil la exploración ya que el jugador tendrá la obligación de cruzar por los puentes, que estarán custodiados. Ver Figura 5.4.6.

Figura 5.4.6 Vista des de arriba de los ríos

- **Camino antiguo**: Al este hay una cordillera con un antiguo camino que une los dos reinos. Es un camino lento de subir debido a su pendiente inclinada. Este camino es indicado para cuando las fronteras están tensas y debes cruzar los reinos pasando desapercibido. Ver figuras 5.4.7 y 5.4.8.

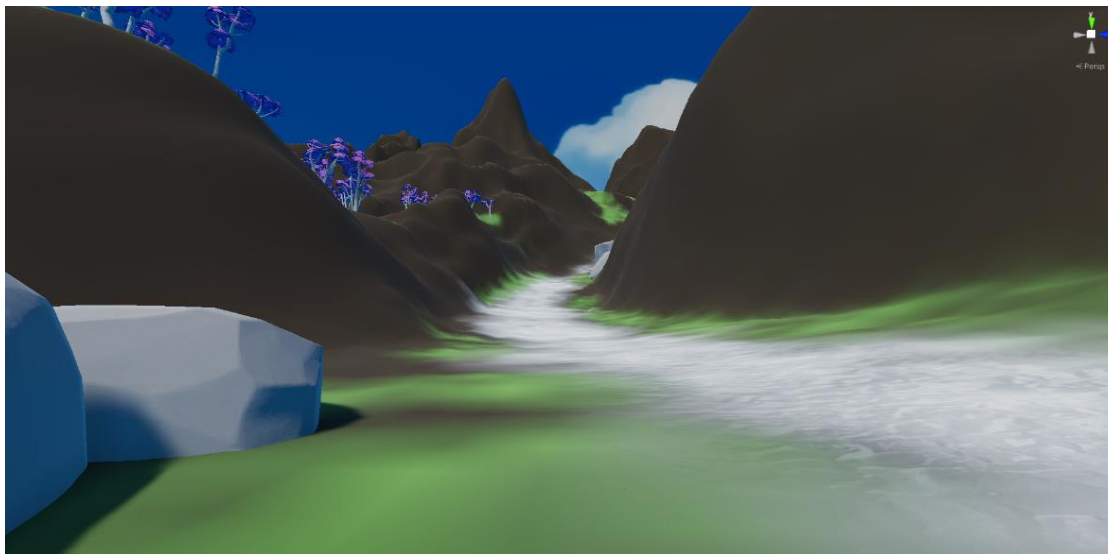

Figura 5.4.7 Camino antiguo

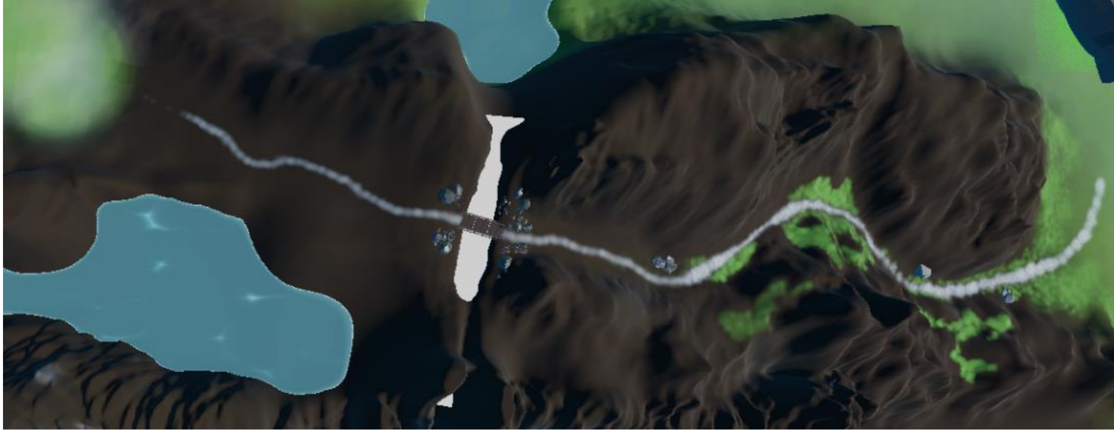

Figura 5.4.8 Camino antiguo visto desde arriba

- **Cascada:** En la frontera por el camino antiguo nace una gran cascada que llena el rio principal de agua. Ver Figura 5.4.9.

Figura 5.4.9 Cascada

- **Bosque:** Al oeste hay un frondoso bosque con árboles de distintos colores que sirve de escondite para los rebeldes. Ver Figura 5.4.10.

Figura 5.4.10 Bosque

- **Aldea montaña:** Al norte al lado de un pequeño lago está la aldea de la montaña con casas hechas únicamente de piedra. Al igual que la aldea del mar, ésta no tiene ninguna estructura de protección. Ver Figura 5.4.11.

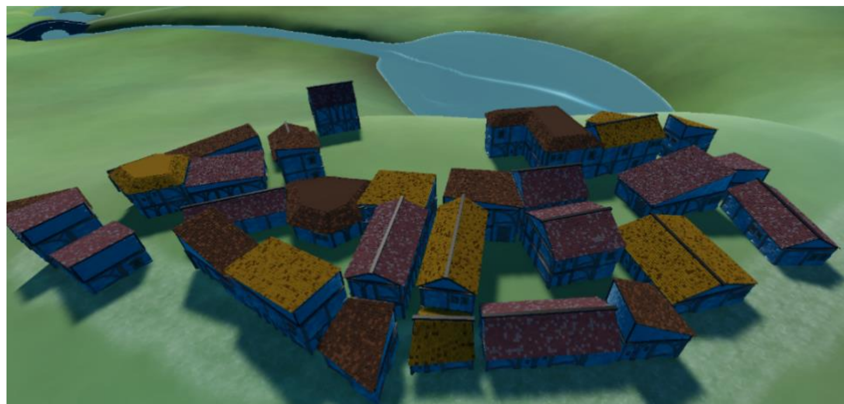

Figura 5.4.11 Aldea de montaña

- **Ciudad montaña:** La ciudad en la que vive el rey del reino de la montaña se encuentra al Noroeste y, pese a tener todas las casas construidas de piedra, su muralla está levantada únicamente de picas de madera. En el centro de la ciudad se encuentra un gran castillo en el cual vive el rey. Ver Figura 5.4.12.

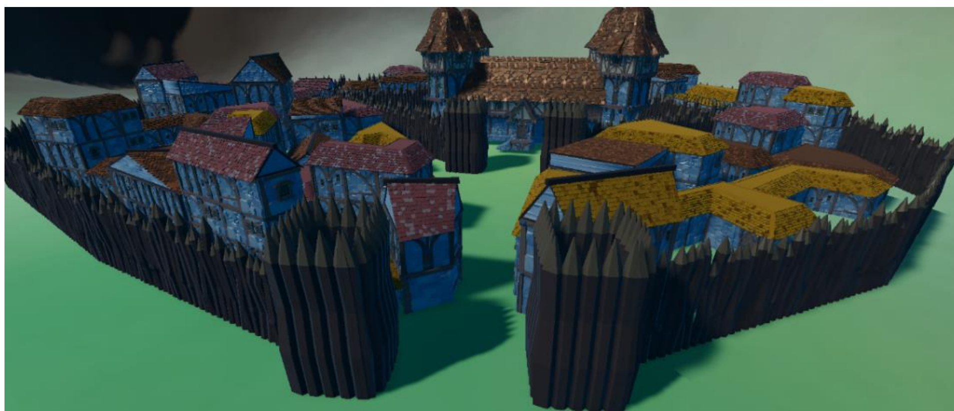

Figura 5.4.12 Ciudad de montaña (Lice)

Utilizamos diferentes técnicas para diferenciar las poblaciones de los distintos reinos. El elemento diferenciador principal es la muralla de las ciudades, ya que es lo primero que se ve al encontrarse con una ciudad. En el caso de la ciudad de montaña, está rodeada por unas picas de madera, un estilo más vikingo, mientras que la ciudad del mar está protegida por una muralla de piedra con torreones, acercándose más al estilo medieval. De manera más sutil también hemos utilizado la paleta de colores de los tejados para indicar a que reino pertenece la aldea / ciudad, siendo la paleta de la Figura 5.4.13 la representativa de la ciudad del mar i la paleta de la Figura 5.4.14 la que representa la ciudad de la montaña.

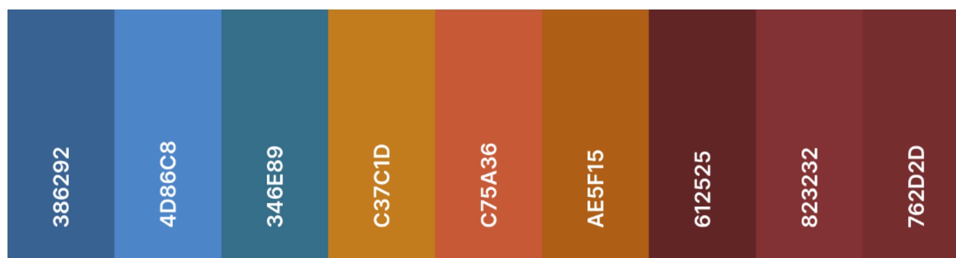

Figura 5.4.13 Paleta de colores tejados reino del mar

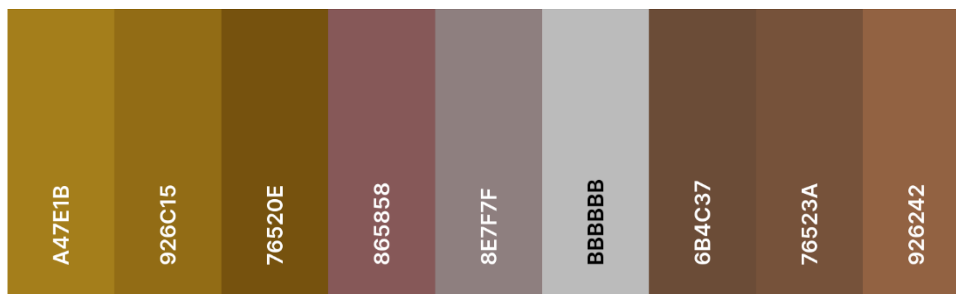

Figura 5.4.14 Paleta de colores tejados reino de montaña

Finalmente, en la Figura 5.4.15 podemos ver un esquema del espacio de juego en el cual a nivel de boceto se pueden apreciar todos los subespacios comentados durante todo el apartado.

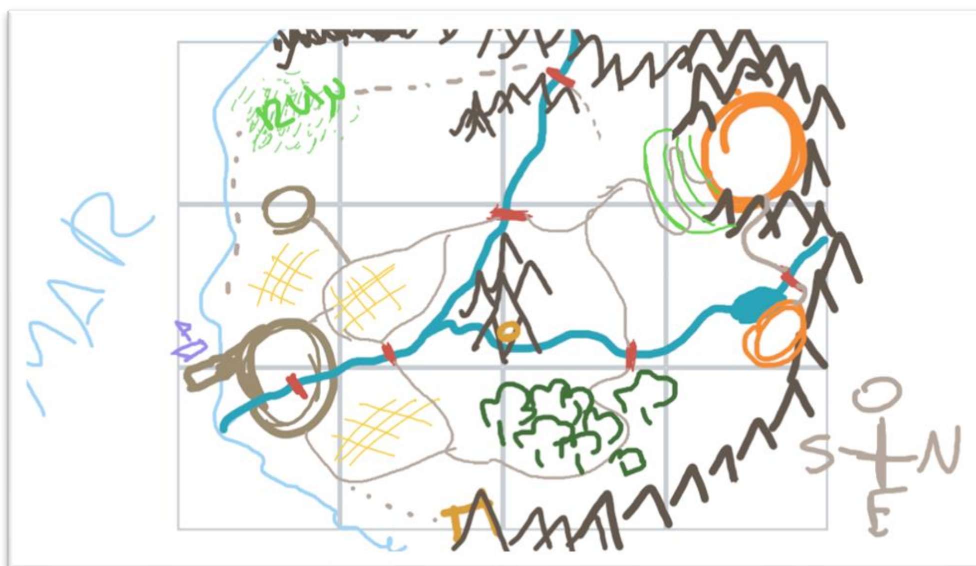

Figura 5.4.15 Esquema / boceto espacio de juego global

5.4.2. Dimensión temporal

La dimensión temporal es muy importante en nuestro videojuego ya que la historia, y la inmersión en ella, dependen de ésta. No tenemos una fecha definida, ya que no es realmente importante, y como el mundo es fantástico, no necesitamos que sea parezca a ninguna época real.

El factor importante es que el juego se distribuye en 7 días. Para avanzar al siguiente día, el jugador tiene que cumplir todas las misiones principales del día actual. Aquí es donde entra en juego nuestro diseño de misiones e historia, ya que todas las misiones secundarias que no se hayan finalizado antes del día correspondiente no se podrán avanzar.

En la Figura 5.4.16 podemos observar un diagrama en el cual podemos ver las misiones asignadas al primer día y al segundo, en rojo las misiones principales y en verde las secundarias.

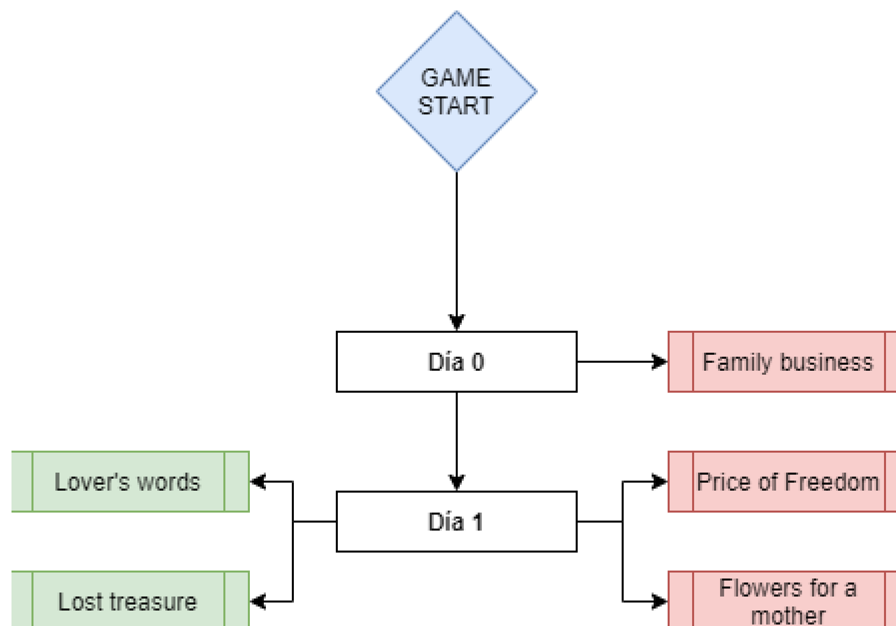

Figura 5.4.16 Diagrama de misiones

El punto está en que, si completamos las dos misiones principales del día 1, *Price of Freedom* y *Flowers for a mother*, sin hacer las misiones secundarias, perderemos la oportunidad de completarlas o, por el contrario, podremos completarlas con alguna penalización en la historia, el jugador habrá perdido la oportunidad de conocer la historia de ese personaje ya que puede ser que esté muerto, mutilado o simplemente esté enfadado...

5.4.3. Dimensión ambiental

El mundo que hemos creado es un mundo de fantasía, que no está ubicado en ninguna época ni región de la civilización, a pesar de ello mezcla elementos de diferentes culturas y épocas. Las ruinas que hemos ubicado cerca del mar están inspiradas en *Stonehenge,* el monumento megalítico del sur del Reino Unido construido entre finales del Neolítico y principios de la Edad de Bronce, que podemos ver en la Figura 5.4.17.

Figura 5.4.17 Stonehenge

Por otro lado, tanto los personajes como los edificios y la muralla del sur vienen de la cultura de la Alta Edad Media. La ciudad de la montaña sería la más multicultural, ya que sus edificios vienen, como hemos dicho antes, de la Alta Edad Media, pero su muralla viene de la cultura vikinga, donde usaban troncos afilados para las estructuras defensivas. Para darle el toque final la entrada a esta está inspirada en la antigua cultura inca, más concretamente en la ciudad de Machu Picchu, que, al estar en la montaña con terrenos tan irregulares, construyeron terrazas para crear pequeñas zonas de terreno plano, como se puede apreciar en la Figura 5.4.18.

Figura 5.4.18 Machu Picchu

5.4.4. Referentes estéticos

Nuestro principal referente estético sería el *The Legend of Zelda: Breath of the wild,* ya no sólo por sus gráficos con toon shading sino por sus vastos terrenos visibles desde largas distancias, que llevan la exploración del terreno a otro nivel. Ver Figura 5.4.18.

Figura 5.4.18 Captura de The Legend of Zelda: Breath of the wild

Por otro lado, decidimos que el juego tendría un arte Low Poly. Para esto dos de las principales referencias fueron *That Dragon, Cancer* para los personajes (Figura 5.4.19) y el Lonely Mountains: Downhill (Figura 5.4.20) para el entorno.

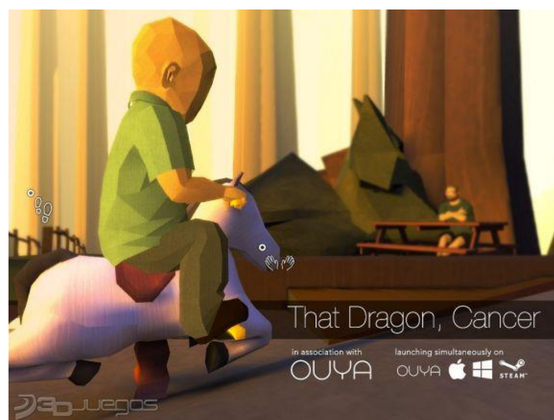

Figura 5.4.19 Captura de That Dragon, Cancer

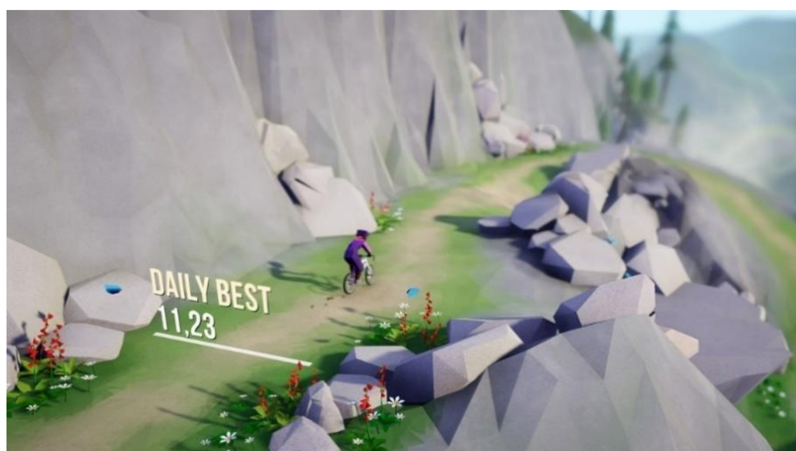

Figura 5.4.20 Captura de Lonely Mountains: Downhill

Para las ciudades no hemos usado un referente claro de un producto de los medios digitales, sino que hemos tomado muchas referencias de antiguas construcciones de la edad media y maquetas e ilustraciones de pinterest.

Figura 5.4.21 Maqueta casa medieval

Figura 5.4.22 Renderizado puente medieval

Figura 5.4.23 Renderizados de casas medievales

Figura 5.4.24 Fotografía de una casa medieval

5.4.5. Dimensión emocional

En cuanto a la dimensión emocional de nuestro videojuego, hemos utilizado la injusticia como motor que puede llegar a provocar frustración y/o rabia. Pero más allá del motor de la historia, cada personaje con cada diálogo intenta transmitir una emoción diferente al jugador. No todo son emociones malas, también hay momentos emotivos que causan gran felicidad, como, por ejemplo, el reencuentro de Leona y Diana.

Para ayudar a transmitir estas emociones de los personajes, cada línea de diálogo está ligada a una emoción. Al ejecutarse este diálogo en el juego se reproducirá un sonido que ayudará al jugador a entender la emoción que está expresando el personaje, ya que a veces es difícil transmitir una emoción sólo con un diálogo de texto. A continuación, podemos ver una lista de todas las emociones que en algún momento sienten los personajes y de las cuales tenemos grabados sonidos que las representan:

- Enfadado
- Indiferente
- Feliz
- Cansado
- Risueño
- Dubitativo
- Triste
- Confundido
- Asustado

Las emociones de Mena, el protagonista, están grabadas también y asignadas a sus correspondientes líneas de diálogo, pero será decisión del jugador decidir porque camino llevar la conversación según la emoción que la situación le transmita al jugador.

El juego no va de una historia feliz, todo lo contrario, hay muchas situaciones injustas y crueles, pero lo que sí que intentamos transmitir con este videojuego es que siempre hay luz en la oscuridad.

5.4.6. Aspectos Éticos

El mundo que hemos creado es duro e injusto, el jugador tendrá que lidiar con diferentes situaciones en las cual tendrá que ser partícipe y decidir qué es lo que él considera más justo o centrarse en sus necesidades y pasar por alto diferentes injusticias que se encontrará.

La primera situación con la que se va a encontrar el jugador es que al llegar al reino del mar y al ser extranjeros, el rey decide que el padre del protagonista será su esclavo. Si el protagonista quiere liberarlo, tendrá que pagarle una elevada cantidad de dinero que tendrá que conseguir trabajando.

Estás injusticias no son únicas del reino del mar ya que todo el mundo está repleto de diferentes injusticias que van desde estafas a maltratos y discriminaciones.

La mayoría de los habitantes de estos reinos están acostumbrados a estas situaciones y lo tienen normalizado, pero habrá diferentes grupos de personajes que denunciaran ciertas injusticias mientras que otros lucharan contra ellas activamente.

La ética del protagonista se basará en la ética del propio jugador, ya que será éste el que tomará las decisiones que harán evolucionar la historia.

5.5. Interfaces

En este apartado explicaremos las diferentes interfaces que hemos diseñado e implementado para el videojuego y para las herramientas que hemos desarrollado para facilitar el desarrollo.

5.5.1 Mapa

El mapa es importante al jugador, ya que es su manera de interpretar el mundo del juego, en él podrá hacer sus esbozos y delimitar las zonas para poder orientarse mejor. Para la zona de dibujo hemos utilizado una textura de papel viejo para darle un toque más diegético. Por otro lado, tenemos una botonera al lado izquierdo de la zona de dibujo donde el jugador puede escoger qué herramienta de dibujo quiere utilizar, es una botonera sencilla con espacio para añadir más herramientas de dibujo. En la Figura 5.5.1 podemos la interfaz descrita.

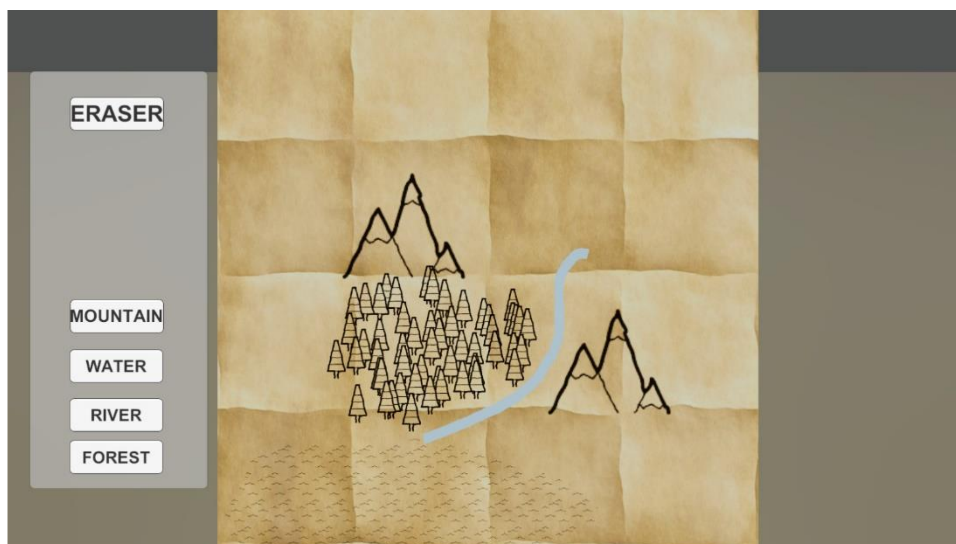

Figura 5.5.1 Interfaz mapa

5.5.2 Inventario del carro

El inventario del carro se define en una rejilla 3D. Es un inventario diegético, el jugador podrá colocar los objetos que recoja encajándolos en la rejilla según el volumen 3D que ocupe. En blanco están indicadas las casillas donde hay espacio disponible, en verde la casilla que se está seleccionando actualmente y, en caso de intentar colocar un objeto en un espacio ocupado, se indicaría en rojo.

En el lateral izquierdo tenemos una lista de los objetos que están disponibles para colocar y un botón que permite al jugador vaciar el carro para poder colocar los objetos de nuevo y así, organizar mejor el inventario. Ver Figura 5.5.2.

Figura 5.5.2 Interfaz inventario carro

5.5.3 Comercio

El comercio es el centro de la economía de nuestro videojuego, ya que es gracias a ello que el jugador podrá conseguir el dinero necesario para poder pagar al rey y comprar diferentes mejoras. En el lateral izquierdo tenemos un panel semi transparente, para que nos permita ver el entorno, en donde tenemos indicado el dinero actual del jugador. También hay dos botones, uno para abrir el panel de compra y otro para abrir el panel de venta.

En el panel de compra estarán listados los objetos que tiene el comerciante a la venta, junto con una imagen del objeto, su nombre y un rango de precios por el cual se puede obtener, a la izquierda de cada objeto en venta hay un botón para comprar el objeto.

Figura 5.5.3 Interfaz compra

Al comprar el objeto se cerrará la interfaz de la Figura 5.5.3 y se abrirá la interfaz que podemos ver en la Figura 5.5.4. En esta interfaz el jugador verá los controles del minijuego de regateo para decidir el precio final del producto. En la Figura 5.5.4 podemos ver que la interfaz indica que tecla tiene qué pulsar el jugador, en este caso la flecha izquierda.

Figura 5.5.4 Interfaz minijuego

En el caso del menú de venta de artículos, es prácticamente igual al de compra con la única diferencia que el botón de comprar pasa a ser un botón para vender. En ambos menús encontraremos un botón en la esquina inferior derecha para cerrarlo.

5.5.4 Misiones

El principal punto de información del jugador es el diario, este se representa en una interfaz diegética dónde podemos encontrar la información de las misiones actuales. En la página de la izquierda tenemos el nombre de las misiones. Al seleccionar una de ellas veremos en la página de la derecha la información ampliada de la misión. En la Figura 5.5.5 podemos ver que el jugador sólo tiene una misión principal y, en la página derecha, se ve el progreso de esta e información más detallada de los objetivos.

Figura 5.5.5 Interfaz misiones principales

Si pasamos de página, en el diario, veremos una interfaz muy similar, pero para las misiones secundarias en vez de las principales. Ver Figura 5.5.6.

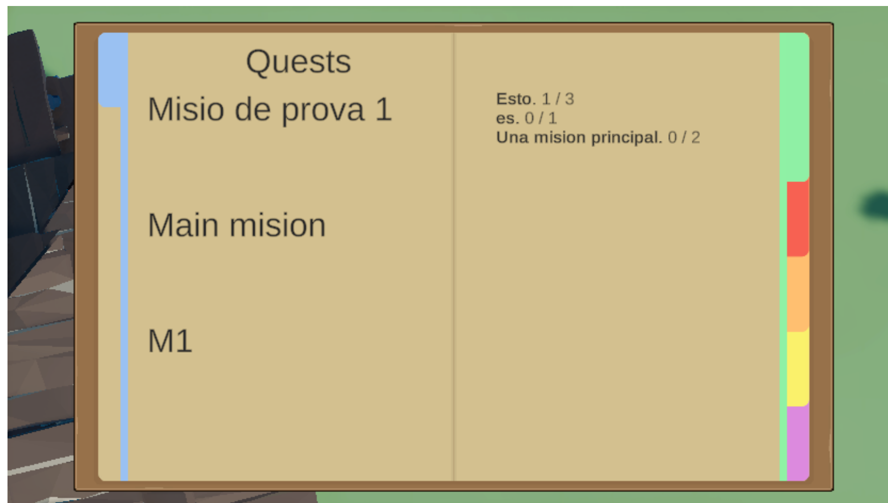

Figura 5.5.6 Interfaz Misiones secundarias

5.5.5 Inventario

Si avanzamos a la tercera página del diario, veremos un apartado en donde podremos consultar los objetos que tiene el jugador y la cantidad de dinero de la que dispone, como podemos ver en la Figura 5.5.7.

Figura 5.5.7 Interfaz Inventario

5.5.6 Diálogos

Cuando el jugador está manteniendo una conversación con un NPC, se mostrará en la parte inferior central un recuadro, en la parte superior de este se podrá ver el personaje que está hablando en la parte superior, en el centro veremos que está diciendo ese personaje y en la parte inferior derecha habrá un botón que al pulsarlo hará avanzar la conversación. Ver Figura 5.5.8.

Figura 5.5.8 Interfaz diálogo

Durante una conversación puede ser que un NPC le haga una pregunta a Mena. En este caso el jugador tendrá que escoger la respuesta. En el recuadro que hemos visto antes, veremos una lista de respuestas. Al pasar el ratón por encima de una de estas, quedará remarcada, como se puede apreciar en la Figura 5.5.9. Al hacer clic en una de las respuestas la conversación avanzará.

Figura 5.5.9 Interfaz preguntas

5.5.7 Creador de diálogos

Al contrario que todas las interfaces anteriores, en este apartado explicaremos una interfaz que no es del videojuego. Esta interfaz pertenece a una herramienta que hemos creado para facilitar el desarrollo del videojuego, DialogueMaker o herramienta de creación de diálogos.

Esta herramienta está compuesta de dos interfaces, una para crear texto y otra para crear respuestas a una pregunta. En la Figura 5.5.10 podemos ver la primera de ellas, en donde el guionista encontrará diferentes campos de texto para introducir la información referente a: nombre fichero, versión, número línea diálogo, emoción, que personaje dice la línea de diálogo y la línea de diálogo per se. En la parte inferior de la interfaz hay dos botones, uno que permite comprobar si los datos introducidos son válidos y otro que sirve para guardar el diálogo.

Figura 5.5.10 Interfaz crear dialogo

En la interfaz podemos ver que hay el botón *Dialogue.* Este botón sirve para cambiar la herramienta de la versión de diálogos a la versión de respuestas, que veremos en la Figura 5.5.11. En esta versión tenemos los campos de texto para introducir la información referente a: nombre fichero, versión, número de respuesta, texto respuesta. Al igual que en la anterior, tenemos los botones de validar y guardar en la parte inferior. Si pulsamos el botón *Question,* volveríamos a la interfaz de Diálogo.

Figura 5.5.11 Interfaz crear pregunta

5.6 Game Layout Charts

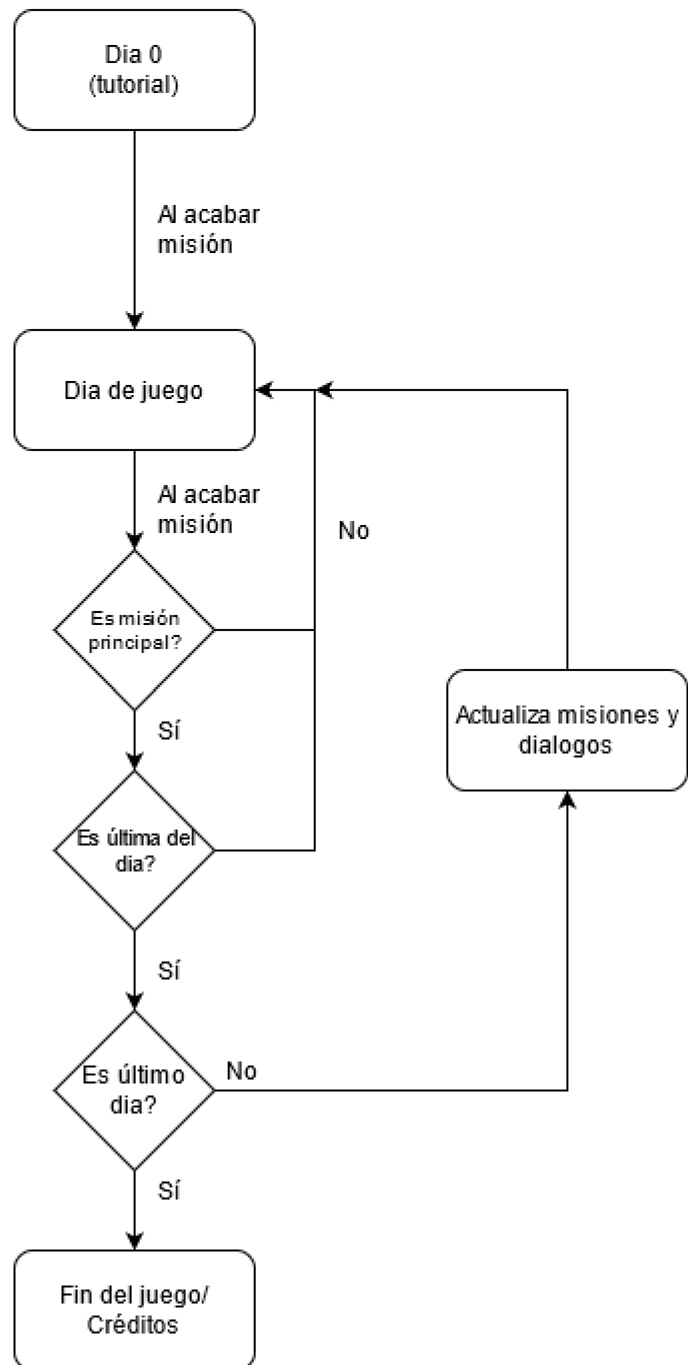

Figura 5.6.1 Flowchart del videojuego

Cómo hemos mencionado antes, y se puede observar en la Figura 5.6.1; nuestro juego está separado en días. Cada una de estos con sus misiones y diálogos. Al acabar todas las misiones principales de un día, se pasa al siguiente, pudiendo llegar a perder la oportunidad de hacer ciertas misiones opcionales.

5.7 Diseño de objetos

La mayoría de los objetos de este juego se pueden dividir en dos categorías. Interactuables, o con función, y los meramente decorativos. Estos dos tipos de objetos se suelen dividir en el propio juego, dando una paleta de colores más clara a los interactuables y una más oscura para los de decoración. Por ejemplo, las rocas que se encuentran en algunos caminos, como las que podemos encontrar en el camino de montaña, tiene una tonalidad de colores mucho más oscura que un objeto con el que se interactúa, como es el carro de este nuestro videojuego.

5.8 Producción externa

Los elementos que no hemos creado nosotros en el videojuego principalmente han sido assets visuales y sonoros, aunque también se han usado algunas librerías de código libre que no pertenecen a ninguno de los integrantes del grupo.

Los elementos sonoros fueron especialmente los efectos de sonido, los que obtuvimos de freesounds.org y que están especificados más adelante, en el punto 5.12.1. Los elementos visuales fueron principalmente los personajes y la base de construcción de las casas (Paredes, puertas, ventanas, ...).

Las librerías de código de creación externa de nuestro proyecto serian: Cinemachine, NavmeshNavigator, TextMeshPro, Triangulator y otros paquetes menores de Unity.

5.9 Entorno de Unity y diseño del proyecto

A la hora de distribuir los archivos del proyecto usamos un sistema considerado estándar en la comunidad de Unity: Dividir los assets por categorías y, dentro de esas categorías, ir especificando por carpetas a qué elemento pertenecen. Es decir, dentro de la carpeta assets, que es la principal del proyecto en el editor, lo scripts se encuentran en la carpeta scripts, los prefabs en la carpeta prefabs, las escenas en la carpeta Scenes, …. Por ejemplo, el script QuestLog.cs se encuentra en la dirección del proyecto Assets/Scripts/QuestSystem/QuestLog.cs.

En el juego, el desarrollo del mismo funcionó con la siguiente estructura de escenas: Para desarrollar una nueva mecánica o hacer pruebas de ciertos sistemas, cada uno de los desarrolladores de este proyecto disponían de una o varias escenas para dejar volar su imaginación. Después, para el juego final se juntó todo en una escena principal donde se realiza todo el transcurso de nuestro videojuego.

5.10. Elementos de feedback

Nuestro videojuego tiene pocos elementos de feedback, ya que no es un juego mecánicamente complejo que necesite respuestas sonoras o visuales al realizar acciones, como podría ser un combate o un juego de plataformas. Tampoco tiene puntuaciones como podría ser un juego arcade. Así, nos hemos limitado a poner los elementos justos para que sea agradable para los jugadores.

Nuestro recurso predominante para añadir feedback han sido los efectos de sonido. A pesar de que hay jugadores que juegan sin sonido, o con el volumen casi al mínimo, creemos que es un recurso con mucha fuerza para favorecer la inmersión en el videojuego.

Como hemos comentado previamente, en el apartado 5.4.5, hemos utilizado sonidos para dar feedback de las emociones de los personajes que están dialogando. Siguiendo con el sonido, también hemos añadido ruido de pasos cuando anda el personaje que, dependiendo del terreno que esté pisando, sonaran unos efectos de sonidos u otros.

Fuera del feedback que sería in-game, tenemos el feedback auditivo de los menús, para los cuales también hemos grabado nuestros propios sonidos para cuando el jugador pulsa algún botón de cualquiera de los menús.

Por otro lado, tenemos los elementos de feedback visual que se centran básicamente en las interfaces de usuario, las cuales hemos explicado en el apartado 5.5.

5.11 Diseño de los efectos y partículas

A pesar de buscar un estilo simplista para el juego, queríamos que este fuese lo más visualmente atractivo posible, por lo que usamos varias técnicas para embellecerlo.

HDRP

Primero de todo, decidimos usar el HDRP (High Definition Render Pipeline). Este es un asset propio de Unity, el cual permite usar su tubería de renderizado en alta definición. Esto, a pesar de tener un coste notorio en el rendimiento, nos permite tener la iluminación y sombreado de mayor calidad posible, además de assets únicos para esta tubería de renderizado. Entre estos se encuentra el VFX graph, por ejemplo; el que usamos para hacer la hierba, y que explicaremos en el apartado 6.

Luces

Para las luces, decidimos usar tres tipos. El skybox, una luz direccional y varios *Point Light.*

Gracias al skybox, podemos simular una luz constante, la cual no afecta a las sombras, pero ayuda en la iluminación. Al usar la tubería de renderizado de alta definición, teníamos varias opciones a escoger. Finalmente, decidimos usar las que se ven en la Figura 5.11.1.

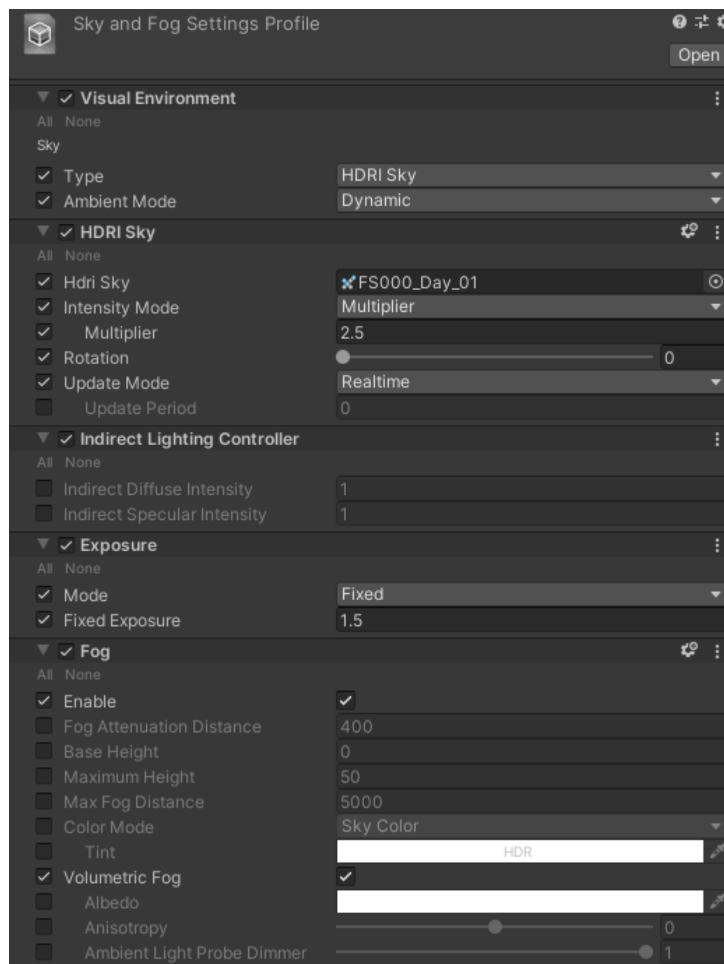

Figura 5.11.1 Ajustes del skybox

A pesar de haber opciones de renderizado más realistas, decidimos usar el tipo de cielo HDRI Sky porque era el que sabíamos usar y nos gusta los resultados que da. También pusimos parámetros de exposición y un poco de niebla para que las zonas lejanas no se renderizasen.

La luz direccional la hemos usado para emular el sol. Es una luz que emula rayos de luz que vienen de una única dirección, y que afectan en todo el mapa por igual. Como podemos observar en la Figura 5.11.2, esto proyecta sombras uniformes, dando la sensación de haber un sol.

Figura 5.11.2 Demostración de las sombras en las ruinas

Por último, hemos agregado luces de tipo *Point Light.* Estos son, como la documentación de Unity indica, luces que "se encuentran en un punto en el espacio y emiten luz en todas las direcciones por igual". Hemos usado este tipo de luces para simular hogueras o farolas que están dispersadas por varios lugares del mapa; dando así dinamismo a ciertos paisajes, pues las sombras que proyectan estas luces no son uniformes. Las que se pueden observar en la Figura 5.11.3 por ejemplo, son de las luces del mercado.

Figura 5.11.3 Luces en el mercado

Sombras

Para las sombras usamos la propia tubería de renderizado de Unity. Decidimos usar sombras suaves. Estas tienen el contorno degradado, como se puede ver en la Figura 5.11.4. Usamos este tipo de sombras porque, al ser más parecidas a las que nos encontramos en el día a día, el resultado es más vistoso y realista.

Figura 5.11.4 Sombra de la muralla

LOD

A pesar de no ser un efecto visual, creemos que el LOD (Level Of Detail) es digno de mención. Este es un componente de Unity que nos permite cambiar entre diferentes *renderers* a medida que cambia la distancia con la cámara. Con esto hemos podemos usar geometrías complejas sin perder mucho rendimiento, pues cuando el jugador se aleje, podemos cambiarla por una con menos polígonos, o quitarla por completo.

Partículas

Para dotar a nuestro juego de mayor realismo y estilo artístico, usamos los sistemas de partículas para realzar o rellenar ciertos sitios del mapa. La cascada, por ejemplo, es la combinación de dos sistemas de partículas. Cada uno de estos renderiza un material propio sobre un modelo con forma de cascada. El primero está encargado de hacer el movimiento del agua y el segundo de la espuma.

Y la hierba también usa un sistema de partículas, el VFXGraph (que es un sistema de partículas que funciona en GPU).

5.12. Diseño de sonido

Desarrollar un videojuego siempre es una tarea complicada, pero dentro del propio desarrollo el apartado sonoro siempre es extremadamente complicado. Todo el mundo puede, en mayor o menor medida, crear o editar elementos visuales con programas tan básicos como Paint, pero crear elementos sonoros requiere programas más complejos, e incluso aquellos que se pueden encontrar de uso gratuito muchas veces no son exactamente lo que se busca.

Con esto en mente, en este apartado explicaremos no sólo todo el aspecto sonoro de Debt Trading, sino también las complicaciones que encontramos y soluciones que utilizamos.

5.12.1. Efectos de sonido

Todo universo que se sienta vivo tiene sonidos, y desde el principio del desarrollo supimos que para nuestro proyecto tendríamos que depender de librerías de sonidos de uso gratuito. Como hemos explicado en el apartado de Viabilidad Económica, obtuvimos la mayoría de los archivos de sonidos de la página freesound.org. Estos sonidos son:

- steps_grass_gravel_rocks, por *Mystikuum*

- Rockburst8, por *Cheddababy*
- Typewriter19, por *tams_kp*
- Donkey braying, por *LeandiViljoen*
- R13-53-Horse Trotting on Stone, por *craigsmith*
- Cash Register, por *kiddpark*
- Walking in Long Grass, por *Leafs67*
- walk-grass, por *JanKoehl*
- Footsteps on Stone, por *Fission9*
- ezel gesnuif, por *eliasheuninck*
- FloorCracking, por *lennyboy*
- 56-Ruedas De Carrito-consolidated, por *DSA98*

Además de los definidos arriba, nosotros grabamos por nuestra cuenta diferentes sonidos interactuando con un papel y un libro. A continuación, estos archivos fueron modificados en Audacity con diferentes procesos, como la eliminación de ruido y añadiendo reverberación, y en algunos casos, juntar dos o más para obtener todos los sonidos para nuestro proyecto.

La lista de estos sonidos es la siguiente:

- Para el menú principal, AbrirMenu y CerrarMenu.
- Para el mapa, AbrirMapa y CerrarMapa.
- Del burro, que incluyen BurroAndarHierba, BurroAndarPiedra, BurroNoContinuar y AcariciarBurro.
- Dos sonidos para los botones de los menús, Boton1 y Boton2.
- Para los botones de los diálogos, BotonNext y BotonOpcionDialogo.
- MenaPasoPiedra, MenaPasoHierba y MenaSubirACarro para la interacción del personaje.
- También creamos sonidos para los objetos, ColocarObjetoCarro y ObjetoCaerSuelo.
- ComprarVenderObjeto lo utilizamos en el sistema de compra y venta.
- PasarPagina, que lo utilizamos a la hora de inspeccionar el menú de misiones.
- Finalmente, RuedaCarro, por si requerimos que la rueda del carro haga un sonido específico.

5.12.2. Voces

Siendo el diálogo uno de los aspectos más importantes del proyecto, queríamos tener algún tipo de *feedback* sonoro desde el principio. Pensamos en tres sonidos que aplicar durante el diálogo: el tecleo de una máquina de escribir, una voz amortiguada constante (similar al estilo de Animal Crossing) y, finalmente, un gruñido corto al comenzar cada frase. Después de varias pruebas, nos decantamos por la última opción, y planificamos como desarrollarla.

De esta manera, realizamos diferentes versiones de los gruñidos que nos harían falta para cada personaje, con la ayuda de amigas para los personajes femeninos y el de Mena. Obtuvimos más de 40 clips de audio que editamos de manera sencilla (recortando y eliminando el ruido de fondo), para así obtener clips de las emociones para cada personaje principal y secundario.

Este método fue sencillo y fácil de implementar, y nos permitió obtener un gran efecto que daba una sensación de “estar vivo” al diálogo que no habríamos conseguido de otra manera.

5.12.3. Música

La banda sonora está influenciada muy fuertemente por juegos cómo Minecraft y Zelda: Breath of the Wild. Para eso buscamos hacer canciones simples y ambientales, que reflejen una emoción concreta, como puede ser tranquilidad, alegría o tensión. Estas canciones se irán reproduciendo aleatoriamente

en ciertos momentos del juego, dependiendo de la emoción de ese instante. Por ejemplo, mientras el jugador da un largo paseo por la pradera buscando recursos, podría empezar a sonar una canción tranquila de fondo, que hiciese más amena la experiencia.

6. Implementación y pruebas

6.1 Carro

Como strand game que es nuestro juego, uno de los puntos más importantes es cómo moverse y cómo transportar mercancías por el mapa. Para eso decidimos de forma unánime usar un carro de tracción animal.

Figura 6.1.1 Carro con el jugador montado

6.1.1 Movimiento

El movimiento del carro esta divido en dos partes de código. Todo está en el mismo script llamado CarroMovement.cs. El primer bloque se encuentra dentro del unity event Update(). En éste básicamente se recogen los inputs del jugador y sé calculan ciertos parámetros necesarios que se usaran en la segunda parte del código de movimiento.

En la función GetInput, la variable steer representa el modificador de dirección Horizontal del vehículo, que puede oscilan entre 1 y -1. La variable accelerate se encargará de multiplicar el valor de la velocidad máxima para obtener la velocidad actual. En caso de uso de mando, para que no se esté moviendo constantemente, aunque el jugador no toque el joystick, le pusimos un umbral de 0.3 tanto positivo como negativo. Por último, la variable momentum funciona de tal manera que no pueda pasar de 0% a 100% de velocidad en un frame y así aumente de forma gradual.

```
if (movable)
{
    steer = Input.GetAxis("Horizontal");
    accelerate = Input.GetAxis("Vertical");
    if (accelerate < 0.3f && accelerate > -0.3f)
    {
        accelerate = 1 * momentum;
        currentMaxVelocity = 0;
    }
    else
    {
        momentum = 1;
    }
    if (accelerate < 0)
    {
        momentum = -1;
        accelerate = accelerate / 5;
        steer = steer / 5;
    }
}
```

Figura 6.1.2 Función GetInput

La segunda parte del código de movimiento se encuentra en la función FixedUpdate() de Unity. Esa función hace de “motor” del propio carro, es decir, aplica el movimiento correspondiente con los parámetros obtenidos de la función GetInput. Todos los movimientos del carro se realizan manipulando el componente rigidbody de Unity con la función MovePosition(). Esta función realiza las transformaciones pertinentes teniendo en cuenta todas las colisiones en esa transformación.

```
void FixedUpdate()//El motor como tal
{
    if (gonaEstamparse && accelerate >= 0 || !movable)
    {
        return;
    }

    char costat = 'M';
    if (checkeIfBalanced(ref costat))
    {
        //Debug.Log("Esta desbalançejat per el costat " + costat);
        estabilizar(costat);
    }

    velocity = Mathf.LerpUnclamped(velocity, currentMaxVelocity, (currentMaxVelocity != 0 ? timeToStop : timeToMaxVelocity) * Time.deltaTime);
    burroAnimation.SetFloat("Movement", accelerate);


    float finalAngle = maxAngle * steer;
    wheelColliders[0].steerAngle = finalAngle;
    wheelColliders[1].steerAngle = finalAngle;

    burroWheel.steerAngle = finalAngle;
    directionRotation.Set(0, 0.2f * steer, 0, 1);

    for (int i = 0; i < wheelColliders.Length; i++)
        wheelColliders[i].brakeTorque = breakTorque;

    burroWheel.brakeTorque = breakTorque;
    direction.localRotation = directionRotation;

    rb.MovePosition(tr.localPosition + direction.forward * Time.deltaTime * velocity * accelerate);
    if (currentMaxVelocity != 0) rb.MoveRotation(Quaternion.Lerp(tr.localRotation, direction.rotation, giradorVelocity * Time.deltaTime));
    updateWheelMeshes();
}
```

Figura 6.1.3 FixedUpdate del CarroMovement.cs

Todas estas trasformaciones no pueden ser 100% libres porque si no rompería la inmersión de este mundo. Para evitar eso, todos los movimientos están limitados por nosotros. Estos limitadores se explicarán más adelante.

6.1.2 Limitadores

Todo este movimiento está limitado por 3 componentes principales. Un limitador físico que detiene completamente el avance, un limitador horizontal que hace que el carro no esté nunca desestabilizado y pueda volcar y, un limitador de velocidad dependiente de la pendiente actual donde se encuentra el carro.

El limitador físico consiste en un boxCollider que se encuentra en el propio burro, que al colisionar, cambia la variable booleana gonaCollide que hace que cuando se llege al FixedUpdate haga un return automáticamente

```
0 references
private void OnCollisionEnter(Collision collision)
{
    if(collision.transform.tag != "Player") gonaCollide = true;
}

0 references
private void OnCollisionExit(Collision collision)
{
    if (collision.transform.tag != "Player") gonaCollide = false;
}
```

Figura 6.1.4 Funciones de colisión

El limitador horizontal se ejecuta cada vez que se ejecuta el FixedUpdate llamando a la función checkeIfBalanced(). Esta función devuelve un booleano que es true si el carro está debalanceado. Si devuelve true también actualizara la variable costat con el lado correspondiente que este desbalanceado ("L" para izquierda y "R" para la derecha). Se considera que está desbalanceado cuando los raycast que se lanzan desde las ruedas en dirección hacia abajo. Si todas las ruedas de un lado no están tocando el suelo, se considera que el carro estará desbalanceado de ese lado.

```
1 reference
private bool checkeIfBalanced(ref char unbalancedSide)
{
    for (int i = 0; i < groundCheckers.Length; i++)
    {
        Debug.DrawRay(groundCheckers[i].position, -transform.up * maxDistanceAcceptable, Color.yellow);
    }

    if (NoTouch())
        return false;

    // Comprovar costat esquerra
    if (!Physics.Raycast(groundCheckers[0].position, -transform.up, maxDistanceAcceptable))
    {
        if (!Physics.Raycast(groundCheckers[3].position, -transform.up, maxDistanceAcceptable))
        {
            unbalancedSide = 'L';
            return true;
        }
    }

    //Comprovar costat dreta
    if (!Physics.Raycast(groundCheckers[1].position, -transform.up, maxDistanceAcceptable))
    {
        if (!Physics.Raycast(groundCheckers[2].position, -transform.up, maxDistanceAcceptable))
        {
            unbalancedSide = 'R';
            return true;
        }
    }

    return false;
}
```

Figura 6.1.5 Función CheckeIfBalanced

Al recibir un true de esta función, se ejecutará la función estabilizar, que ejecutará la estabilización como tal.

```
1 reference
private void estabilizar(char costat)
{
    float sentitGir = costat == 'R' ? -1f : 1f;

    rb.constraints = RigidbodyConstraints.FreezeRotation;
    tr.RotateAround(tr.position, sentitGir * tr.forward, powerGir * Time.deltaTime);
    rb.constraints = RigidbodyConstraints.None;
}
```

Figura 6.1.6 Función estabilizar

Por último, el limtador de velocidad según la pendiente consiste en 3 possibles estados (Subiendo, bajando y estable) con sus 3 velocidades distintas para cada uno de esos momentos. Para calcular en qué punto de la pendiente se encuentra, se usaron principios básicos de la trigonometría. Usamos dos puntos que van de punta a punta del carro como distancia que luego normalizaremos a 1. Como la distancia del punto la calculamos sobre un plano XZ en el mundo de Unity, que al inclinarse la distancia será siempre menor o igual a la original.

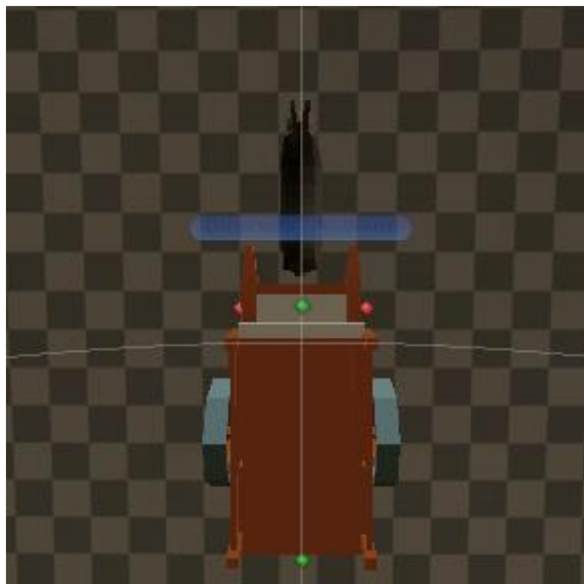

Figura 6.1.7 Los puntos verdes representan los puntos para la distancia del carro

Una vez tenemos estas distancias, podemos aplicar esos principios trigonométricos para sacar el ángulo actual del carro. Una vez tenemos este ángulo, podemos ver si sobreapasa el limite definido por nosotros y cambiar su velocidad.

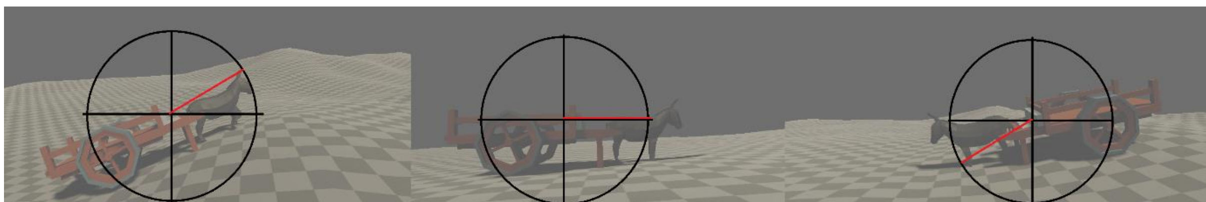

Figura 6.1.8 Esquematización de los cálculos trigonométricos

```
1 reference
private void CheckInclinacio()
{
    float altura = comprovantsDinclinacio[0].position.y - comprovantsDinclinacio[1].position.y;

    if (altura >= changeHeigth) currentMaxVelocity = maxSpeedUpwards;
    else if (altura <= -changeHeigth) currentMaxVelocity = maxSpeedDownwards;
    else currentMaxVelocity = maxSpeed;
}
```

Figura 6.1.9 Función CheckInclinacio

6.1.3 Inventario carro

El carro dispone de un inventario tridimensional diegético. El inventario consiste en un sistema de grid inspirado en los inventarios del Residen Evil, pero teniendo en cuenta un eje extra. Tiene dos partes, la visual que ve el jugador, y la interna que controla el código.

La parte interna se prepara en la función GenerarGrid() que se ejecuta en el Start del script IGrid.cs. En esté se genera un array tridimensional de INodes, que es una clase que sólo guarda información para el grid sobre el objeto contenido (como su posición en el grid, su posición en el mundo, si está ocupado y, cual es el objeto que lo ocupa).

```
///@pre Totes les cordenades del vector mesures son > 0
///@post Omple la variable grid amb el conjunts de nodes corresponents segon en
2 references
public void GenerarGrid()
{
    grid = new INode[mesures.x, mesures.y, mesures.z];
    Vector3 esquerraAbaixMon = transform.position - transform.right * (mesures.x) / 2 - transform.up * (mesures.y) / 2 - transform.forward * (mesures.z) / 2;//Agafar referencia al món físic

    for (int i = 0; i < mesures.y; i++)
    {//Altura - Y
        for (int j = 0; j < mesures.x; j++)
        {//Dreta - X
            for (int k = 0; k < mesures.z; k++)
            {//Frontal - Z
                Vector3 puntMon = esquerraAbaixMon + transform.right * (j * radiCub * 2 + radiCub) + transform.up * (i * radiCub * 2 + radiCub) + transform.forward * (k * radiCub * 2 + radiCub);

                grid[j, i, k] = new INode(j, i, k, puntMon);
            }
        }
    }

}
```

Figura 6.1.10 Función GenerarGrid

La parte visual se genera mediante la función GenerarGridVisual() que también se ejecuta en el Start del script IGrid.cs. Esta función lo que hace es instanciar unos cubos con la opacidad bajada que hacen de casilla del grid. Solo se instancian una vez, todas las demás se activan y desactivan.

```
1 reference
private void GenerarGridVisual()
{
    visualGrid = new GameObject[mesures.x, mesures.y, mesures.z];
    //Vector3 esquerraAbaixMon = transform.position - transform.right * (mesures.x)/2 - transform.up * (mesures.y)/2 - transform.forward * (mesures.z)/2;//Agafar referencia al món físic
    Vector3 esquerraAbaixMon = transform.position - transform.right * (mesures.x) / 2 - transform.up * (mesures.y) / 2 - transform.forward * (mesures.z) / 2;//Agafar referencia al món físic

    for (int i = 0; i < mesures.y; i++)
    {//Altura - Y
        for (int j = 0; j < mesures.x; j++)
        {//Dreta - X
            for (int k = 0; k < mesures.z; k++)
            {//Frontal - Z
                Vector3 puntMon = esquerraAbaixMon + transform.right * (j * radiCub * 2 + radiCub) + transform.up * (i * radiCub * 2 + radiCub) + transform.forward * (k * radiCub * 2 + radiCub);

                visualGrid[j, i, k] = Instantiate(gridCasellaPrefab, puntMon, gameObject.transform.rotation, gameObject.transform);
            }
        }
    }

}
```

Figura 6.1.11 Función GenerarGrid

Una vez activado el inventario del carro, se ejecuta automáticamente la secuencia que aparece en la siguiente figura

```
// Update is called once per frame
0 references
void Update()
{
    colocarMouse();

    if (selecionat != null)
    {
        goDownObjecte();

        selecionat.transform.position = _mouse.position;
        ObjSeleccionat = selecionat.GetComponent<IObjecte>();

        if (Input.GetKeyDown("r"))
            ObjSeleccionat.rotar();

        if (Input.GetMouseButtonDown(1))
            colocarObjecte();
    }
    else
    {
        goDownMouse();

        if (Input.GetMouseButtonDown(1))
            agafarObjecte();

    }

    UpdateGridVisual();

}
```

Figura 6.1.12 Función Update

La primera parte del Update, la función colocarMouse consiste en mover un transform invisible en el mundo del juego que hace de referencia al punto donde esta apuntado el jugador.

```
///@pre --
///@post La posicio del _mouse s'actualitza respecte la posicio del ratoli de veritat
1 reference
private void colocarMouse()
{
    Vector3 _mousePosition;
    Ray ray = _camera.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;

    if (Physics.Raycast(ray, out hit, 100f))
    {
        Debug.DrawLine(ray.origin, hit.point);
        _mousePosition = hit.point;
        _mousePosition.y = getAlturaGrid();
        _mouse.position = _mousePosition;
    }
}
```

Figura 6.1.13 Función colocarMouse

Apartir de aquí, el código se divide en dos partes: La que se ejecuta cuando el jugador tiene un objeto seleccionado o cuando el jugador no tiene ningún objeto seleccionado. Si no tiene ningún objeto selecionado, ejecutara la función goDownMouse. Esta función usa la función de nodeAlMon, que coge la posición actual del ratón del jugador para saber a dónde está apuntando, para coger la posición donde se encuentra del inventario y bajar hasta el final mientras este no esté ocupado por un objeto.

```
///@pre --
///@post La posicio del _mouse s'actualitza en la mateixa posicio pero cap abaix
1 reference
private void goDownMouse()
{
    INode node = NodeAlMon(_mouse.position);
    int alturaActual = node.posY;
    bool continuar = true;

    while (continuar && alturaActual > 0)
    {
        if (grid[node.posX, alturaActual, node.posZ].ocupat)
            continuar = false;
        else
            alturaActual--;
    }

    _mouse.position = grid[node.posX, alturaActual, node.posZ].posAlMon;
    //_mouse.position = new Vector3(_mouse.position.x, transform.position.y + alturaActual * radiCub*2 + radiCub, _mouse.position.z);
}
```

Figura 6.1.14 Función goDownMouse

```
///@pre pos != null
///@post Les cordenades obtingudes esntan dins del rang del grid
5 references
private INode NodeAlMon(Vector3 pos)
{

    nodeTransformador.position = pos;
    nodeTransformador.RotateAround(transform.position, transform.up, -transform.rotation.eulerAngles.y);
    pos = nodeTransformador.position;

    float percentX = getPercent(pos.x, (float)mesures.x, transform.position.x);
    float percentY = getPercent(pos.y, (float)mesures.y, transform.position.y);
    float percentZ = getPercent(pos.z, (float)mesures.z, transform.position.z);

    percentX = Mathf.Clamp01(percentX);
    percentY = Mathf.Clamp01(percentY);
    percentZ = Mathf.Clamp01(percentZ);

    // Debug.Log(percentX + " " + percentY + " " + percentZ);
    //Castear a int
    int X = Mathf.RoundToInt((mesures.x - 1) * percentX);
    int Y = Mathf.RoundToInt((mesures.y - 1) * percentY);
    int Z = Mathf.RoundToInt((mesures.z - 1) * percentZ);
    //int Y = mesures.y - 1;

    // Debug.Log(X + " " + Y + " " + Z);

    return grid[X, Y, Z];
}
```

Figura 6.1.15 Función NodeAlMon

Si en proceso el jugador hace clic izquierdo con el ratón, se ejecutará la función agafarObjecte. Esta función consta de varias partes. Primero mira que el nodo pulsado esté ocupado por un objeto, si no lo está, termina y no hace nada más. Después comprueba que ese objeto seleccionado no tenga ningún objeto encima en ninguna otra parte de este. Esta comprobación se hace en la función DaltLliure que coge la posición origen del objeto seleccionado y sus medidas para comprobar si no está completamente libre por arriba.

```
///@Pre --
///@Post Retorna True si el objecte no te cap altre objecte al damunt, fals altrament
1 reference
private bool DaltLliure(INode og, IObjecte objecteAComprovar)
{
    if ((og.posY + objecteAComprovar.mesures.y) >= mesures.y)
        return true;// Si no hi mes altura al comprovar s'acaba.

    int Y = og.posY + objecteAComprovar.mesures.y;

    //Elements incrementadors
    int incrementX = definirIncrement(ObjSeleccionat.mesures.x);
    int incrementZ = definirIncrement(ObjSeleccionat.mesures.z);

    for (int X = og.posX; continuarRecorregut(X, og.posX, ObjSeleccionat.mesures.x, incrementX); X += incrementX)
    {
        for (int Z = og.posZ; continuarRecorregut(Z, og.posZ, ObjSeleccionat.mesures.z, incrementZ); Z += incrementZ)
        {
            if (grid[X, Y, Z].ocupat)
            {
                return false;
            }
        }
    }

    return true;
}
```

Figura 6.1.16 Función DaltLliure

A continuación, se realiza un recorrido por las casillas correspondientes del Grid delimitado por la función continuarRecorregut. Esta función no sólo se usa aquí, también se usa en la función anterior de DaltLliure y en prácticamente todas las que hacen un recorrido limitado por el Grid del inventario. Esta función consiste en una comprobación booleana de varios parámetros, pasados a número para que, en caso de partición de ifs el compilador, pueda realizar la ejecución sin tener que recurrir a ramificar el codigo.

```
///@pre --
///@Post Retorna si amb les dades actuals ha de continuar amb el loop o no
14 references
private bool continuarRecorregut(int valorActual, int posActual, int mesura, int sentit)
{
    return Convert.ToBoolean(Convert.ToInt32((sentit == 1)) * Convert.ToInt32(valorActual < (posActual + mesura)) + Convert.ToInt32(sentit == -1) * Convert.ToInt32(valorActual > (posActual + mesura)));
}
```

Figura 6.1.17 Función continuarRecorregut

Si volvemos a al código que aparece en el Update del script, veremos el otro posible camino, el camino donde sí tenemos un objeto seleccionado. Aquí se empieza ejecutando la función goDownObject, que es muy similar a la función goDownMouse, pero se coloca una casilla por encima de la posición ocupada (Si es posible) en lugar de la ocupada. El objeto seleccionado se moverá a la posición definida por esta función. Una vez en este punto el jugador tiene dos posibilidades de interacción: Pulsar la tecla R para rotar el objeto en el inventario o, pulsar el botón izquierda del raton para colocar el objecto en el inventario del carro. La función para colocar el objeto se llama colocarObjete. Esta función es la inversa de agafarObjecte, donde hace prácticamente lo mismo, pero marcando el valor del objeto de la casilla por el objeto seleccionado por el jugador en lugar de definirlo como null.

```
///@pre --
///@Post El objecte seleccionat pasa a ser null i si i cap omple les caselles del grid i coloca el objecte al carro
1 reference
private void colocarObjecte()
{
    INode aux = NodeAlMon(selecionat.transform.position);

    if (cap(aux))
    {
        //Elements incrementadors
        int incrementX = definirIncrement(ObjSeleccionat.mesures.x);
        int incrementY = definirIncrement(ObjSeleccionat.mesures.y);
        int incrementZ = definirIncrement(ObjSeleccionat.mesures.z);

        //Bucle
        for (int X = aux.posX; continuarRecorregut(X, aux.posX, ObjSeleccionat.mesures.x, incrementX); X += incrementX)
        {
            for (int Y = aux.posY; continuarRecorregut(Y, aux.posY, ObjSeleccionat.mesures.y, incrementY); Y += incrementY)
            {
                for (int Z = aux.posZ; continuarRecorregut(Z, aux.posZ, ObjSeleccionat.mesures.z, incrementZ); Z += incrementZ)
                {
                    grid[X, Y, Z].ocupat = true;
                    grid[X, Y, Z].objecte = selecionat;
                }
            }
        }

        ObjSeleccionat.origen = aux;
        ObjSeleccionat.placeObject(transform.parent);
        //selecionat.transform.parent = transform.parent;


        selecionat = null;
        ObjSeleccionat = null;

    }

}
```

Figura 6.1.18 Función colocarObjecte

6.2 Misiones

Nuestro videojuego no requiere que el jugador aprenda mecánicas complejas, y es por eso que uno de los factores motivacionales principales pare que continúe jugando es la propia historia. Para hacer que esta sea interesante, debíamos desarrollar un complejo sistema de eventos, que a su vez fuera modulable, escalable y fácil de implementar para todos los miembros del equipo. Con esta idea en mente se desarrolló este sistema de misiones, que se muestra en la Figura 6.2.1

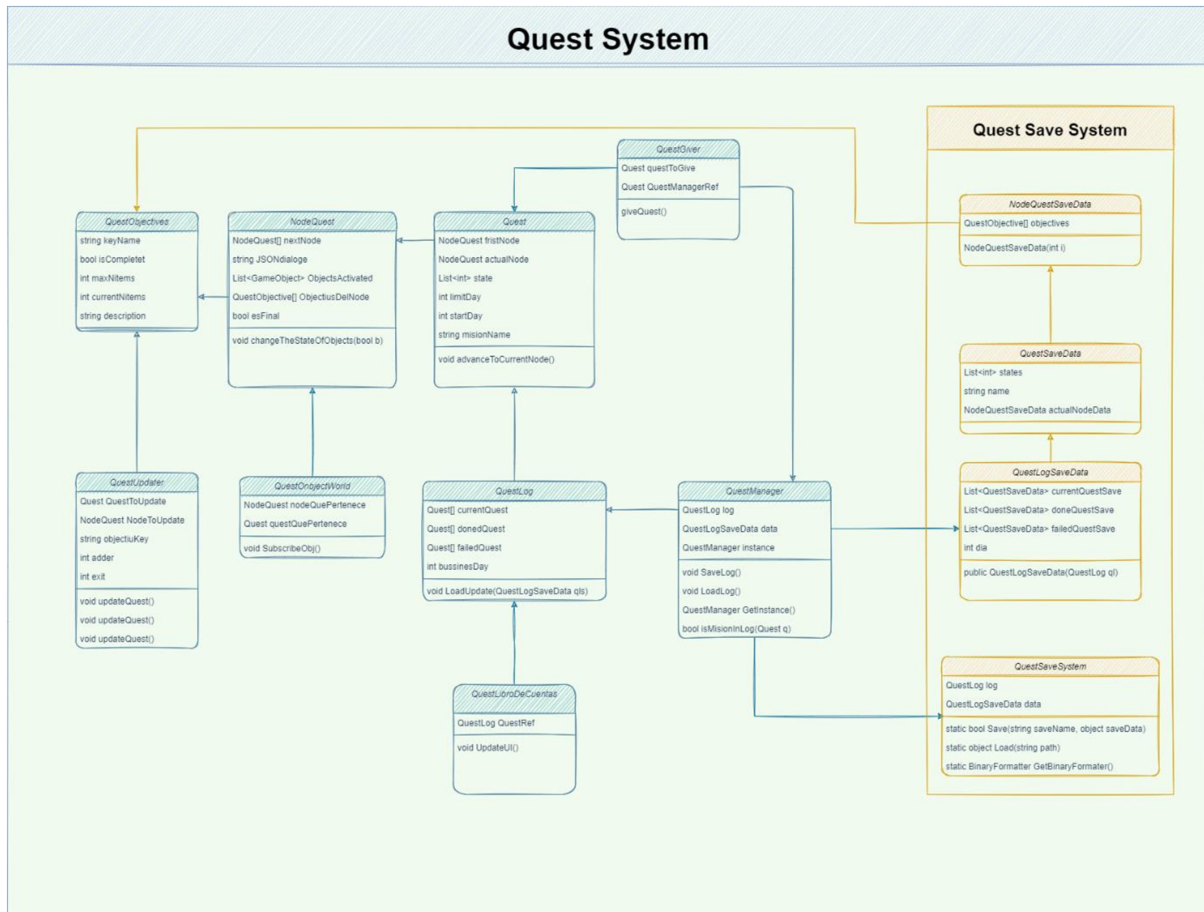

Figura 6.2.1 Esquema de clases de Quest System.

6.2.1 Funcionamiento

En este sistema, hay dos tipos de contenidos para los scripts: Los scripts que son contenedores de datos y los scripts que controlan la interacción con el jugador.

Como contenedor primario, empezamos con la clase *QuestObjective*. Esta clase representa un objetivo de la propia misión. Cada uno de los objetivos contiene:

- Un *string* representando la **key** identificadora del propio objetivo.
- Una *booleana* que dice si esta **completado**.
- Un *string* que hace de **descripción** para el jugador.
- Un *entero* que guarda el **máximo número** de veces que se tiene que completar este objetivo.
- Por último, un *entero* que lleva el **conteo** de las veces que se ha completado este objetivo.

Con esta clase creada, podemos crear el siguiente contenedor de datos, la clase **NodeQuest**. Esta clase ya es un *scriptableObject* de Unity, y contiene:

- una lista de QuestObjective.

- una lista de **posibles nodos siguientes**.
- Una lista de **objetos** donde se podrán suscribir los objetos de la misión.
- Para acabar, una *booleana* que determina si este nodo se considera un **final de la misión**.

Nos queda el resultado en el editor que se puede ver en la Figura 6.2.2.

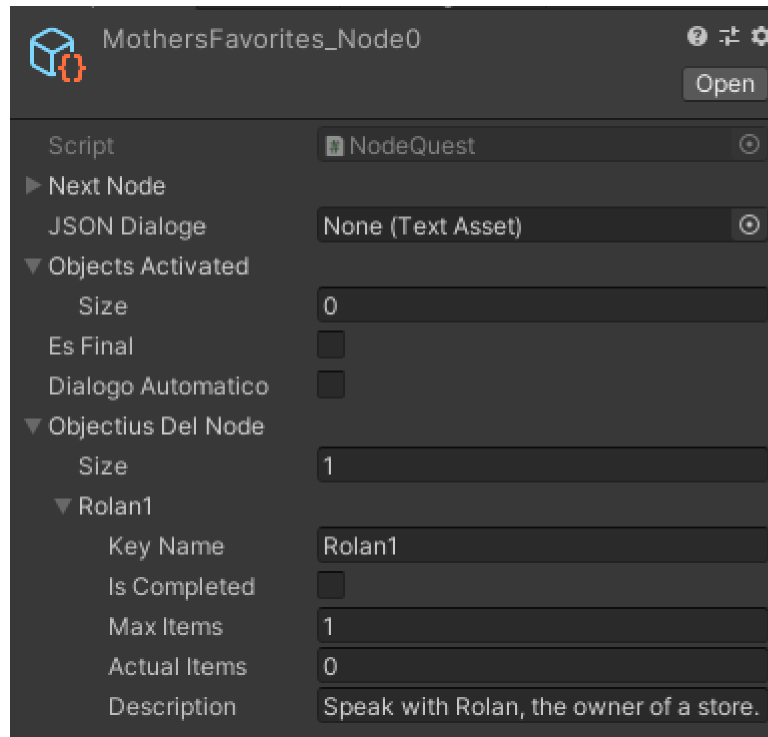

Figura 6.2.2 Visual de NodeQuest

Por último, lo juntamos todo en otro *ScriptableObject,* de la clase *Quest*. En esta clase guardamos:

- El **primer NodeQuest** de la misión.
- El **nodo actual** de la misión.
- Una *booleana* que determina si la misión es **principal**.
- Finalmente, el **nombre de la misión**.

El resultado de un *asset* misión se ve en la Figura 6.2.3.

Figura 6.2.3 Visual de Quest

Para agrupar todas estas misiones, permitiendo que el jugador las pueda observar, las guardamos en un *ScriptableObject* llamado *QuestLog*, como muestra la Figura 6.2.4.

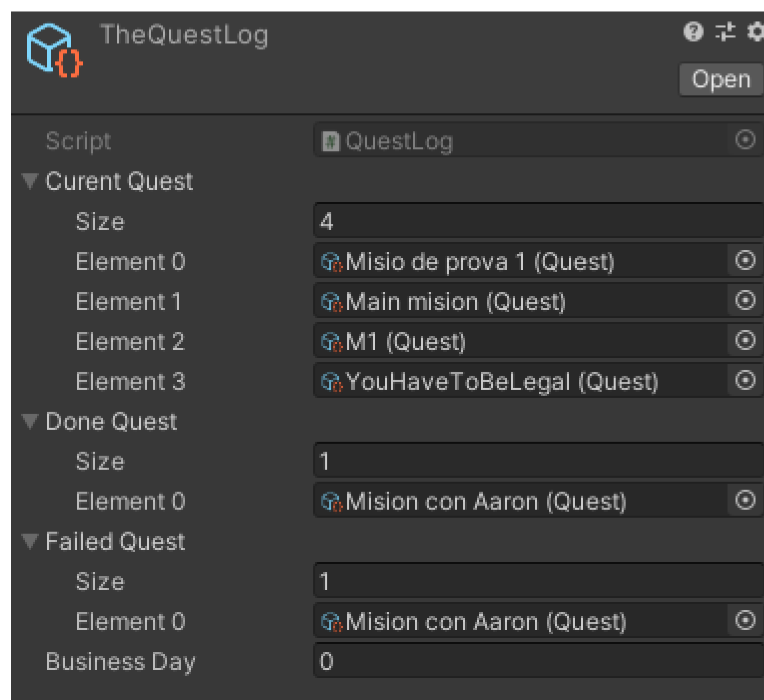

Figura 6.2.4 Visual de QuestLog

Todos estos contenedores de datos se actualizarán durante la partida, según los eventos e interacciones que tengan lugar en el mundo del juego.

6.2.2 Creación de misiones

Con los elementos definidos en el punto anterior, las misiones ya funcionaban y ya podían ser creadas, pero el proceso era bastante confuso, incluso para alguien que sabía cómo funcionaba. Teníamos que recordar la posición correcta de los nodos, la *key* de los objetivos de esos nodos, y otra serie de acciones que son susceptibles al error humano.

Para facilitar la creación de estas misiones, ideamos una herramienta visual, usando la librería *GraphView* de Unity (una librería que está en beta, y todavía no hay mucha documentación al respecto). Esta librería es la que se usa Unity para el *shaderGraph*.

Para empezar, creamos una ventana de Unity que contenga el *GraphView*, la cual podremos abrir en el editor pulsando Tools>QuestGraph. La idea es recrear los nodos pensados en código y sus conexiones como nodos visuales, conectados por como el usuario quiera. Para eso, creamos un componente *NodoQuestGraph* y un componente *QuestObjectiveGraph*.

Al crear un nodo, se empieza con una función que recrea las partes de ese nodo en el editor como los objetivos, las conexiones de nodo, etc.

Todos estos elementos nos darán el resultado que se muestra en la Figura 6.2.5.

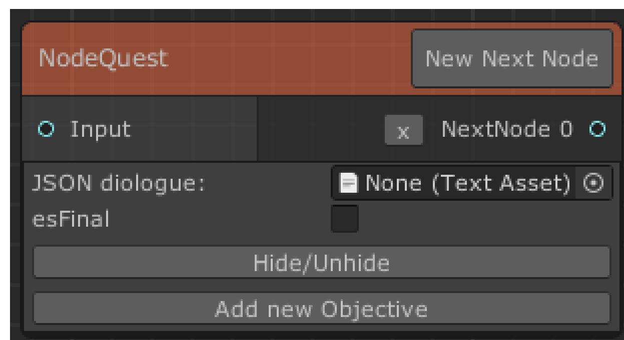

Figura 6.2.5 Visual de un NodeQuest en el editor

Para crear este nodo, se ejecuta la función que aparece en la Figura 6.2.6, que se encarga de instanciar cada uno de los elementos a mano.

```
public NodeQuestGraph CreateNodeQuest(string nodeName, Vector2 position, TextAsset ta = null, bool end = false)
{
    var node = new NodeQuestGraph
    {
        title = nodeName,
        GUID = Guid.NewGuid().ToString(),
        questObjectives = new List<QuestObjectiveGraph>(),
    };

    //Add Input port
    var generatetPortIn = GeneratePort(node, Direction.Input, Port.Capacity.Multi);
    generatetPortIn.portName = "Input";
    node.inputContainer.Add(generatetPortIn);

    node.styleSheets.Add(Resources.Load<StyleSheet>("Node"));

    //Add button to add ouput
    var button = new Button(clickEvent: () =>
    {
        AddNextNodePort(node);
    });
    button.text = "New Next Node";
    node.titleContainer.Add(button);

    //Button to add more objectives
    var button2 = new Button(clickEvent: () =>
    {
        AddNextQuestObjective(node);
    });
    button2.text = "Add new Objective";

    //Hide/Unhide elements
    var hideButton = new Button(clickEvent: () =>
    {
        HideUnhide(node, button2);
    });
    hideButton.text = "Hide/Unhide";



    //JSON Dialogue
    var JSON = new ObjectField("JSON diologue:");
    JSON.objectType = typeof(TextAsset);

    JSON.RegisterValueChangedCallback(evt =>
    {
        node.JSON = evt.newValue as TextAsset;
    });
    JSON.SetValueWithoutNotify(ta);

    //Bool es final
    var togle = new Toggle();
    togle.label = "esFinal";

    togle.RegisterValueChangedCallback(evt =>
    {
        node.esFinal = evt.newValue;
    });
    togle.SetValueWithoutNotify(end);

    var container = new Box();
    node.mainContainer.Add(container);// Container per a tenir fons solid

    container.Add(JSON);
    container.Add(togle);
    container.Add(hideButton);
    container.Add(button2);

    node.objectivesRef = new Box();
    container.Add(node.objectivesRef);

    //Refresh la part Visual
    node.RefreshExpandedState();
    node.RefreshPorts();
    node.SetPosition(new Rect(position.x, position.y, 400, 450));

    return node;
}
```

Figura 6.2.6 Código de la función CreateNodeQuest

Si pulsamos el botón de añadir un nuevo objetivo, se añade un objetivo al nodo, como se ve en la Figura 6.2.7. Podemos definir los parámetros del objetivo, que ya han sido especificados en el punto anterior. Si se pulsa el botón "x" del objetivo, este se elimina del nodo.

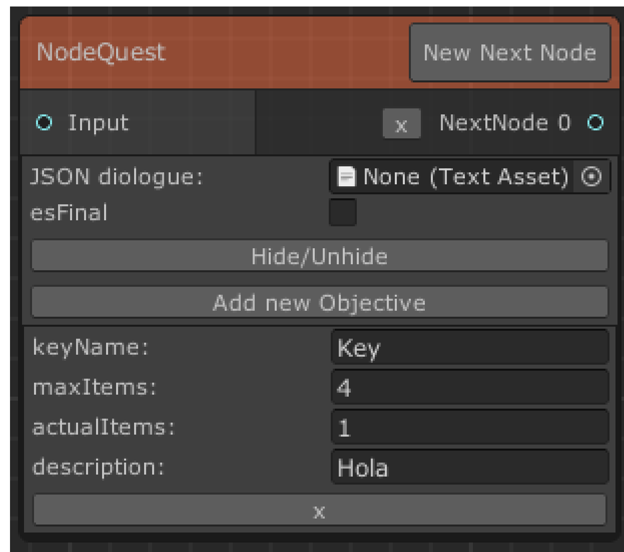

Figura 6.2.7 Visual de NodeQuest con un nuevo objetivo

Como se puede ver en la Figura 6.2.8, una vez creados los nodos, es cuestión de ir rellenado los campos y realizar las conexiones pertinentes para tener una misión completa y funcional.

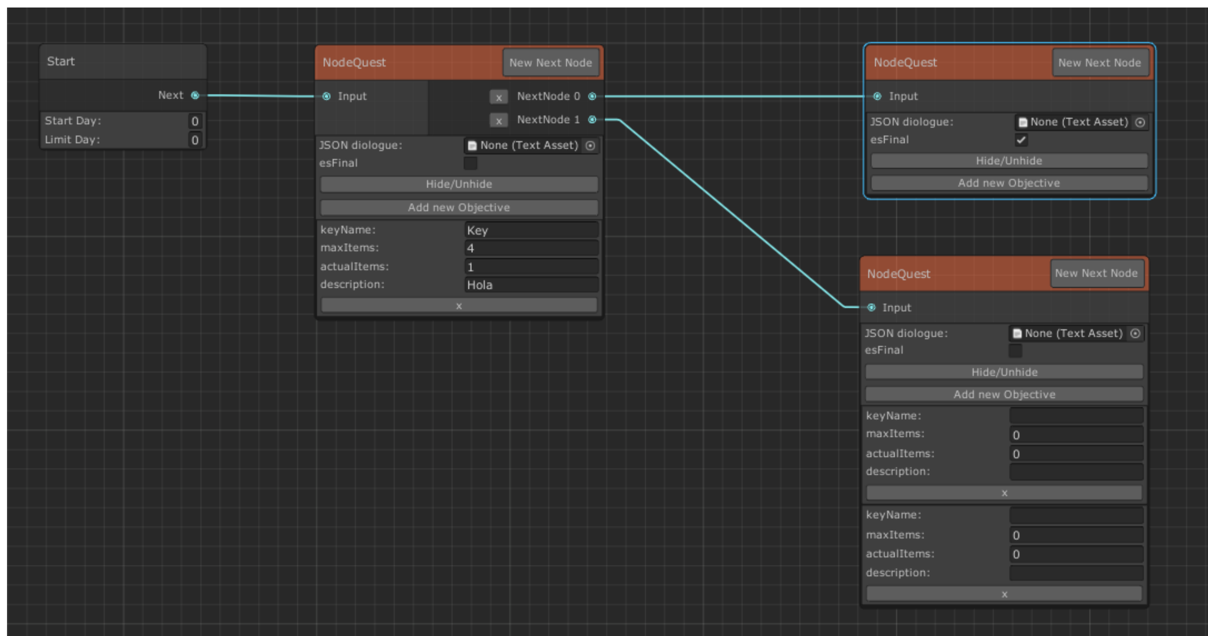

Figura 6.2.8 Ejemplo de una misión sencilla en el editor de misiones

Con la parte de creación acabada, no tendría sentido poder hacer todo esto si no se pudieran guardar los nodos creados en forma de misión. Para llevar esto a cabo, seguimos el siguiente proceso:

1. Coger un *ScriptableObject* vació de *Quest*.
2. Coger todos los nodos del gráfico.
3. Crear los *assets* de estos nodos si no existen; si existen, son modificados.
4. Guardar todas las conexiones realizadas.

5. Marcar el objeto *Quest* como "*dirty*" para que se guarde en disco los cambios realizados.

Podemos ver la función de guardar el grafico de la misión y de guardar las conexiones hechas en la Figuras 6.2.9 y 6.2.10.

```
public void SaveGraph(Quest Q)
{
    if (!Edges.Any()) return;


    List<NodeQuest> NodesInGraph = new List<NodeQuest>();
    // Nodes
    creteNodeQuestAssets(Q, ref NodesInGraph);

    // Conections
    saveConections(Q, NodesInGraph);

    //Last Quest parameters

    var startNode = Nodes.Find(node => node.entryPoint); //Find the first node Graph
    Q.startDay = startNode.startDay;
    Q.limitDay = startNode.limitDay;

    var firstMisionNode = Edges.Find(x => x.output.portName == "Next");
    var firstMisionNode2 = firstMisionNode.input.node as NodeQuestGraph;
    string GUIDfirst = firstMisionNode2.GUID;
    Q.firtsNode = NodesInGraph.Find(n => n.GUID == GUIDfirst);

    EditorUtility.SetDirty(Q);


}
```

Figura 6.2.9 Código de la función SaveGraph

```
private void saveConections(Quest Q, List<NodeQuest> NodesInGraph)
{
    var connectedPorts = Edges.Where(x => x.input.node != null).ToArray();
    Q.nodeLinkData.Clear();

    for (int i = 0; i < connectedPorts.Length; i++)
    {
        var outputNode = connectedPorts[i].output.node as NodeQuestGraph;
        var inputNode = connectedPorts[i].input.node as NodeQuestGraph;

        Q.nodeLinkData.Add(new Quest.NodeLinksGraph
        {
            baseNodeGUID = outputNode.GUID,
            portName = connectedPorts[i].output.portName,
            targetNodeGUID = inputNode.GUID
        });

        //Add to next node list
        NodeQuest baseNode = NodesInGraph.Find(n => n.GUID == outputNode.GUID);
        NodeQuest targetNode = NodesInGraph.Find(n => n.GUID == inputNode.GUID);

        if (targetNode != null && baseNode != null)
            baseNode.nextNode.Add(targetNode);
    }
}
```

Figura 6.2.10 Código de la función SaveConections

Para cargar estas misiones ya creadas y poder editarlas, utilizamos una función propia que nos permitía cargar los datos de los *ScriptableObjects* y pasarlos al *graph*. La carga empieza relativamente sencilla con la función que aparece en la Figura 6.2.11.

```
public void LoadGraph(Quest Q)
{
    if (Q == null)
    {
        EditorUtility.DisplayDialog("Error!!", "Quest aprece como null, revisa el scriptable object", "OK");
        return;
    }

    //var getNodes = AssetDatabase.LoadAllAssetsAtPath($"Assets/Resources/Misiones/{Q.misionName}/Nodes/{Q.misionName}_Node0.asset") as NodeQuest[];
    NodeQuest[] getNodes = Resources.LoadAll<NodeQuest>($"Misiones/{ Q.misionName}/Nodes");
    _cacheNodes = new List<NodeQuest>(getNodes);

    clearGraph(Q);
    LoadNodes(Q);
    ConectNodes(Q);
}
```

Figura 6.2.12 Código de la función LoadGraph

Pero, como se puede observar en la Figura 6.2.13, tenemos que reconstruir todos los elementes que existían previamente, creando los nodos 1 a 1.

```
private void LoadNodes(Quest Q)
{
    foreach (var node in _cacheNodes)
    {
        var tempNode = _targetGraphView.CreateNodeQuest(node.name, Vector2.zero, node.JSONDialoge, node.esFinal);
        //Load node variables
        tempNode.GUID = node.GUID;
        tempNode.JSON = node.JSONDialoge;
        tempNode.esFinal = node.esFinal;
        tempNode.RefreshPorts();
        foreach (QuestObjective obl in node.ObjectiusDelNode)
        {
            //CreateObjectives
            QuestObjectiveGraph objtemp = new QuestObjectiveGraph(obl.keyName, obl.maxItems, obl.actualItems, obl.description);


            var deleteButton = new Button(clickEvent: () => _targetGraphView.removeQuestObjective(tempNode, objtemp))
            {
                text = "x"
            };
            objtemp.Add(deleteButton);

            var newBox = new Box();
            objtemp.Add(newBox);


            objtemp.actualItems = obl.actualItems;
            objtemp.description = obl.description;
            objtemp.maxItems = obl.maxItems;
            objtemp.keyName = obl.keyName;

            tempNode.objectivesRef.Add(objtemp);
            tempNode.questObjectives.Add(objtemp);
        }

        _targetGraphView.AddElement(tempNode);

        var nodePorts = Q.nodeLinkData.Where(x => x.baseNodeGUID == node.GUID).ToList();
        nodePorts.ForEach(x => _targetGraphView.AddNextNodePort(tempNode));


    }
}
```

Figura 6.2.13 Código de la función LoadNodes

El aspecto más costoso de esta función es juntar las conexiones que existían previamente, ya que están involucradas muchas búsquedas en el conjunto de datos que tenemos, como se puede apreciar en la Figura 6.2.14, debajo de este texto.

```
private void ConectNodes(Quest Q)
{
    for (int i = 0; i < Nodes.Count; i++)
    {
        var conections = Q.nodeLinkData.Where(x => x.baseNodeGUID == Nodes[i].GUID).ToList();

        for (int j = 0; j < conections.Count(); j++)
        {
            string targetNodeGUID = conections[j].targetNodeGUID;
            var targetNode = Nodes.Find(x => x.GUID == targetNodeGUID);
            LinkNodes(Nodes[i].outputContainer[j].Q<Port>(), (Port)targetNode.inputContainer[0]);

            targetNode.SetPosition(new Rect(_cacheNodes.First(x => x.GUID == targetNodeGUID).position, new Vector2(150, 200)));
        }
    }
}

private void LinkNodes(Port outpor, Port inport)
{
    var tempEdge = new Edge
    {
        output = outpor,
        input = inport
    };

    tempEdge.input.Connect(tempEdge);
    tempEdge.output.Connect(tempEdge);
    _targetGraphView.Add(tempEdge);


}
```

Figura 6.2.14 Código de las funciones ConectNodes y LinkNodes

6.2.3 Guardar progreso de misión

Para guardar los datos, pasamos la información de las misiones por unas clases específicas, que son una versión *serializable* de los ScriptableObjects definidos anteriormente. Las propias clases tienen un constructor que cojen el scriptableObject para pasarlo a la clase serializable, que se ve en la Figura 6.2.15.

```
public QuestLogSaveData(QuestLog ql)
{
    //Manage current quest
    currentQuestSave = new List<QuestSaveData>();
    doneQuestSave = new List<QuestSaveData>();
    failedQuestSave = new List<QuestSaveData>();

    foreach (Quest q in ql.curentQuest)
    {
        QuestSaveData aux = new QuestSaveData();
        aux.name = q.misionName;
        aux.states = q.state;

        aux.actualNodeData = new NodeQuestSaveData(q.nodeActual.ObjectiusDelNode.Length);

        for (int i = 0; i < q.nodeActual.ObjectiusDelNode.Length; i++)
            aux.actualNodeData.objectives[i] = q.nodeActual.ObjectiusDelNode[i];

        currentQuestSave.Add(aux);
    }

    foreach (Quest q in ql.doneQuest)
    {
        QuestSaveData aux = new QuestSaveData();
        aux.name = q.misionName;

        currentQuestSave.Add(aux);
    }

    foreach (Quest q in ql.failedQuest)
    {
        QuestSaveData aux = new QuestSaveData();
        aux.name = q.misionName;

        currentQuestSave.Add(aux);
    }

    dia = ql.businessDay;

}
```

Figura 6.2.15 Código de la función QuestLogSaveData

Una vez estos datos se han serializado, llamamos a la función *Save,* que guarda estos datos en formato binario. El código de esta función se encuentra en la Figura 6.2.16.

```
public static bool Save(string saveName, object saveData)
{
    BinaryFormatter formatter = GetBinaryFormater();

    if (!Directory.Exists(Application.persistentDataPath + "/saves"))
    {
        Directory.CreateDirectory(Application.persistentDataPath + "/saves");
    }

    string path = GetPath(saveName);
    Debug.Log(path);
    FileStream file = File.Create(path);

    formatter.Serialize(file, saveData);
    Debug.Log("Saveado");
    file.Close();

    return true;
}
```

Figura 6.2.16 Código de la función Save

6.3 NPC

Después de desarrollar ciudades tan grandes mediante nuestro sistema procedural, debíamos llenarlo de alguna manera para se sientan con vida. Para eso desarrollamos un sistema de tráfico para NPCs que simule el movimiento típico de una ciudad.

6.3.1 Waypoint System

Para determinar los caminos que realizarían los NPC's ideamos un sencillo sistema de puntos definidos por el usuario con un funcionamiento similar a una lista doblemente encadenada.

6.3.1.1 Funcionamiento

El funcionamiento de este sistema es relativamente sencillo. Por parte de los puntos definidos del mundo, simplemente son unos transforms con una conexión con el nodo siguiente y el nodo anterior. También en caso de disponer de diferentes caminos posibles tiene una lista de diferentes puntos posibles.

Por parte del propio NPC, se calcula si se ha llegado al punto correspondiente en el Fixed Update.

```
// Update is called once per frame
0 references
void FixedUpdate()
{


    if (cachedTransform.position != objective)
    {
        destinationDirection = objective - cachedTransform.position;
        destinationDirection.y = 0;

        float destinationDistance = destinationDirection.magnitude;

        if(destinationDistance >= stopMargin)
        {
            done = false;
        }
        else
        {
            done = true;
            executeDone();
        }
    }
}
```

Figura 6.3.1 FixedUpdate

6.3.1.2 Creación en el mundo

Para poder crear y modificar estas rutas de los NPCs se desarrolló una ventana en el editor para facilitar su manipulación. El sistema es relativamente sencillo, sólo se tiene que arrastrar un transform que hará de padre del camino de puntos creados para empezar con la construcción de éste. Con esto aparecerán los botones que representan las opciones disponibles de creación del camino.

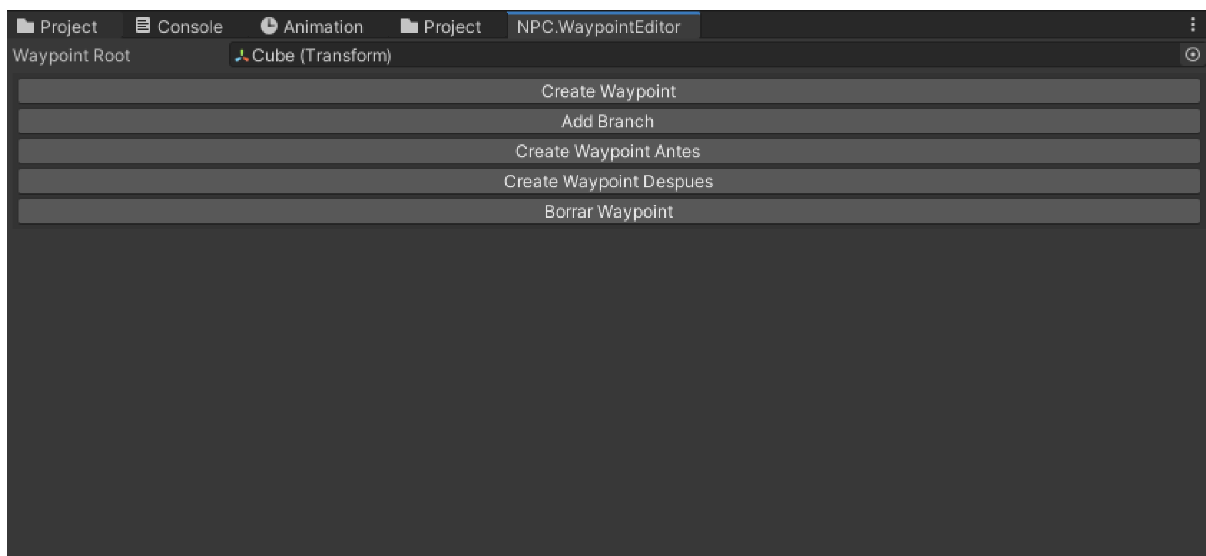

Figura 6.3.2 Ventana WaypointEditor en Unity

Todas las funciones de los botones de este editor son similares a la creación de un elemento en una lista doblemente encadenada. La primera función, la de CreateWaypoint, se encarga de crear un nuevo objeto waypoint y meterlo de hijo en el transform que hace de parent. Como se puede ver en la Figura 6.3.3, esta función crea el objeto y lo coloca al final de la lista de hijos. También, en caso de que se pueda, asigna los puntos anteriores y siguientes del mismo punto y les pone la misma orientación que el punto anterior.

```
private void CreateWaypoint()
{
    GameObject waypointObj = new GameObject("Waypoint " + waypointRoot.childCount, typeof(Waypoint));
    waypointObj.transform.SetParent(waypointRoot, false);

    Waypoint waypoint = waypointObj.GetComponent<Waypoint>();
    if(waypointRoot.childCount > 1)
    {
        waypoint.previousWaypoint = waypointRoot.GetChild(waypointRoot.childCount - 2).GetComponent<Waypoint>();
        waypoint.previousWaypoint.nextWaypoint = waypoint;

        waypoint.transform.position = waypoint.previousWaypoint.transform.position;
        waypoint.transform.forward = waypoint.previousWaypoint.transform.forward;
    }

    Selection.activeGameObject = waypoint.gameObject;
}
```

Figura 6.3.3 Función CreateWaypoint

Los botones de CreateWaypointBefore y CreteWaypointAfter realizan funciones muy similares, con la diferencia que la primera coloca el nuevo waypoint antes del punto seleccionado actualmente y la segunda después del seleccionado en la lista de hijos del parent.

```
private void CreateWaypointBefore()
{
    GameObject waypointObj = new GameObject("Waypoint " + waypointRoot.childCount, typeof(Waypoint));
    waypointObj.transform.SetParent(waypointRoot, false);

    Waypoint waypoint = waypointObj.GetComponent<Waypoint>();

    Waypoint selectedWaypoint = Selection.activeGameObject.GetComponent<Waypoint>();

    waypointObj.transform.position = selectedWaypoint.transform.position;
    waypointObj.transform.forward = selectedWaypoint.transform.forward;

    if(selectedWaypoint.previousWaypoint != null)
    {
        waypoint.previousWaypoint = selectedWaypoint.previousWaypoint;
        selectedWaypoint.previousWaypoint.nextWaypoint = waypoint;
    }

    waypoint.nextWaypoint = selectedWaypoint;
    selectedWaypoint.previousWaypoint = waypoint;

    waypoint.transform.SetSiblingIndex(selectedWaypoint.transform.GetSiblingIndex());

    Selection.activeGameObject = waypoint.gameObject;

}
```

Figura 6.3.4 Función CreateWaypointBefore

```
private void CreteWaypointAfter()
{
    GameObject waypointObj = new GameObject("Waypoint " + waypointRoot.childCount, typeof(Waypoint));
    waypointObj.transform.SetParent(waypointRoot, false);

    Waypoint waypoint = waypointObj.GetComponent<Waypoint>();

    Waypoint selectedWaypoint = Selection.activeGameObject.GetComponent<Waypoint>();

    waypointObj.transform.position = selectedWaypoint.transform.position;
    waypointObj.transform.forward = selectedWaypoint.transform.forward;

    waypoint.previousWaypoint = selectedWaypoint;

    if(selectedWaypoint.nextWaypoint != null)
    {
        selectedWaypoint.nextWaypoint.previousWaypoint = waypoint;
        waypoint.nextWaypoint = selectedWaypoint.nextWaypoint;
    }

    selectedWaypoint.nextWaypoint = waypoint;
    waypoint.transform.SetSiblingIndex(selectedWaypoint.transform.GetSiblingIndex() + 1);

    Selection.activeGameObject = waypoint.gameObject;
}
```

Figura 6.3.5 Función CreteWaypointAfter

Para borrar los puntos, es la misma tónica de hacer una función muy similar a una lista doblemente encadenada. Se redefinen las conexiones con el punto seleccionado y después, se elimina del mundo.

El ultimo botón, el de CreateBranch, crea posibles ramificaciones para avanzar por diferentes puntos, y así dar más variedad al recorrido de los NPCs. Esta función no tiene en cuenta el orden de la lista anterior de hijos del parent ya que se introduce en su propia lista que tiene el waypoint, haciendo así la ejecución de ésta mucho más sencilla.

```
private void CreateBranch(){
    GameObject waypointObj = new GameObject("Branch Waypoint " + waypointRoot.childCount, typeof(Waypoint));
    waypointObj.transform.SetParent(waypointRoot, false);

    Waypoint waypoint = waypointObj.GetComponent<Waypoint>();

    Waypoint branchedFrom = Selection.activeGameObject.GetComponent<Waypoint>();
    branchedFrom.branches.Add(waypoint);

    waypoint.transform.position = branchedFrom.transform.position;
    waypoint.transform.forward = branchedFrom.transform.forward;

    Selection.activeGameObject = waypoint.gameObject;
}
```

Figura 6.3.6 Función CreateBranch

Luego, una vez el punto esta creado, puede ser movido por la escena con las herramientas típicas de manipulación de transforms de Unity.

6.3.2 Movimiento de los NPCs

El movimiento de estos personajes es realiza gracias a la IA de Unity y su componente NavMeshAgent, que se encarga del movimiento y de los cálculos del pathfinding. Una vez han llegado al punto que se calcula en la función FixedUpdete, se ejecutará la función executeDone, que decidirá qual será el siguiente punto a objetivo para el NPC.

```
public void executeDone()
{
    bool doBranch = false;

    if (currentWaypoint.branches != null && currentWaypoint.branches.Count > 0)
    {
        doBranch = Random.Range(0f, 1f) <= currentWaypoint.branchRatio ? true : false;
    }

    if (doBranch)
    {
        currentWaypoint = currentWaypoint.branches[Random.Range(0, currentWaypoint.branches.Count - 1)];
    }
    else
    {
        if (direction == 0)
        {
            if (currentWaypoint.nextWaypoint != null)
            {
                currentWaypoint = currentWaypoint.nextWaypoint;
            }
            else
            {
                currentWaypoint = currentWaypoint.previousWaypoint;
                direction = 1;
            }
        }
        if (direction == 1)
        {
            if (currentWaypoint.previousWaypoint != null)
            {
                currentWaypoint = currentWaypoint.previousWaypoint;
            }
            else
            {
                currentWaypoint = currentWaypoint.nextWaypoint;
                direction = 0;
            }
        }
    }


    objective = currentWaypoint.GetPosition();
    agent.destination = objective;
    done = false;
}
```

Figura 6.3.7 Funcion ecxecuteDone

6.4 Optimización

En este apartado se hablarán de algunos pequeños scripts que, sin ellos, el juego no llegaría al objetivo de los 60FPS (Frames por segundo).

6.4.1 Mesh combiner

Una de las partes de este juego que estamos más orgullosos, con diferencia, ha sido la creación procedural de ciudades (explicado más adelante en el punto 6.13). Pero la primera vez que ejecutamos es script de generar, una vez añadidas las tejas pieza a pieza, con suerte el juego llegaba a los 10FPS. De la necesidad de arreglar este problema nació esta pequeña librería. Esta librería tiene dos funciones principales, combinar mesh por material o combinar mesh con varios materiales.

Las limitaciones de la función de combinar son dos: La primera, el total de vértices de la mesh resultante de la combinación no puede ser mayor de 65535. Este número es así debido a que, por defecto en Unity, el buffer gráfico tiene 16 bits. La segunda es que la mesh combinada sólo puede

tener un tipo de material. La función como tal es relativamente sencilla, coge todos los hijos del padre i combina toda la geometría en el padre. Al acabar, elimina a los hijos para liberar memoria.

```
MeshFilter meshF = parentToCombine.GetComponent<MeshFilter>();
if (meshF == null)
{
    meshF = parentToCombine.gameObject.AddComponent(typeof(MeshFilter)) as MeshFilter;
}

List<MeshFilter> ChildrenRemovePArent = new List<MeshFilter>(parentToCombine.GetComponentsInChildren<MeshFilter>());
ChildrenRemovePArent.RemoveAt(0);
MeshFilter[] meshFiltersChildren = ChildrenRemovePArent.ToArray();
CombineInstance[] combineMesh = new CombineInstance[meshFiltersChildren.Length];


if (parentToCombine.GetComponent<MeshRenderer>() == null)
{
    MeshRenderer mr = parentToCombine.gameObject.AddComponent(typeof(MeshRenderer)) as MeshRenderer;
    mr.sharedMaterial = meshFiltersChildren[0].transform.GetComponent<MeshRenderer>().sharedMaterial;
}

for (int i = 0; i < meshFiltersChildren.Length; i++)
{
    combineMesh[i].mesh = meshFiltersChildren[i].sharedMesh;
    combineMesh[i].transform = meshFiltersChildren[i].transform.localToWorldMatrix;
    //GameObject.Destroy(meshFiltersChildren[i].gameObject, 1f);
    meshFiltersChildren[i].gameObject.SetActive(false);
    if (Application.isEditor)
        GameObject.DestroyImmediate(meshFiltersChildren[i].gameObject);
    else
        GameObject.Destroy(meshFiltersChildren[i].gameObject);
}

Mesh mesh = new Mesh();
mesh.CombineMeshes(combineMesh);
meshF.sharedMesh = mesh;
```

Figura 6.4.1 Función combine

Como las tejas tienen varios materiales, se idéo la función CombineWithDiferentsMaterials. Esta función básicamente divide los hijos del padre por materiales, introducido en un diccionario, y hace que, por cada grupo de materiales, se llame a la función Combine explicada anteriormente.

```csharp
/// <summary>
/// Gets all the diferents materials of the children and sorts them in diferent children.
/// Each one of the children will recive a call of the MeshCombiner.Combine(Transform) function.
/// All the elements can only have one material per each. And all the diferents materials must
/// have a diferent name.
/// </summary>
/// <param name="parentToCombine"> Trasform tha will have the children that will combine the diferent materials</param>
public static void CombineWithDiferentsMaterials(Transform parentToCombine)
{
    //Remove components del pare
    if (parentToCombine.GetComponent<MeshRenderer>() != null)
    {
        if (Application.isEditor)
            Object.DestroyImmediate(parentToCombine.GetComponent<MeshRenderer>());
        else
            Object.Destroy(parentToCombine.GetComponent<MeshRenderer>());
    }
    if (parentToCombine.GetComponent<MeshFilter>() != null)
    {
        if (Application.isEditor)
            Object.DestroyImmediate(parentToCombine.GetComponent<MeshFilter>());
        else
            Object.Destroy(parentToCombine.GetComponent<MeshFilter>());
    }

    Dictionary<string, List<GameObject>> RepartimentDeFills = new Dictionary<string, List<GameObject>>();
    MeshRenderer[] ogChildren = parentToCombine.GetComponentsInChildren<MeshRenderer>();

    //Repartir els fills per material
    for (int i = 0; i < ogChildren.Length; i++)
    {
        if (RepartimentDeFills.ContainsKey(ogChildren[i].sharedMaterial.name))
        {
            RepartimentDeFills[ogChildren[i].sharedMaterial.name].Add(ogChildren[i].gameObject);
        }
        else
        {
            RepartimentDeFills.Add(ogChildren[i].sharedMaterial.name, new List<GameObject>());
            RepartimentDeFills[ogChildren[i].sharedMaterial.name].Add(ogChildren[i].gameObject);
        }
    }

    foreach (KeyValuePair<string, List<GameObject>> lg in RepartimentDeFills)
    {
        GameObject nouFill = new GameObject(lg.Key);

        foreach (GameObject element in lg.Value)
        {
            element.transform.parent = nouFill.transform;
        }

        nouFill.transform.parent = parentToCombine;

        Combine(nouFill);
    }

}
```

Figura 6.4.2 Función CombineWithDiferentsMaterials

Para hacer la prueba original creamos una escena con más de 2000000 vértices divididos entre objetos de 4 vértices. Sin combinar la escena estaba a 10 FPS. Una vez combinados en 4 meshes distintas en lugar de las 48000 existentes los FPS eran de 140.

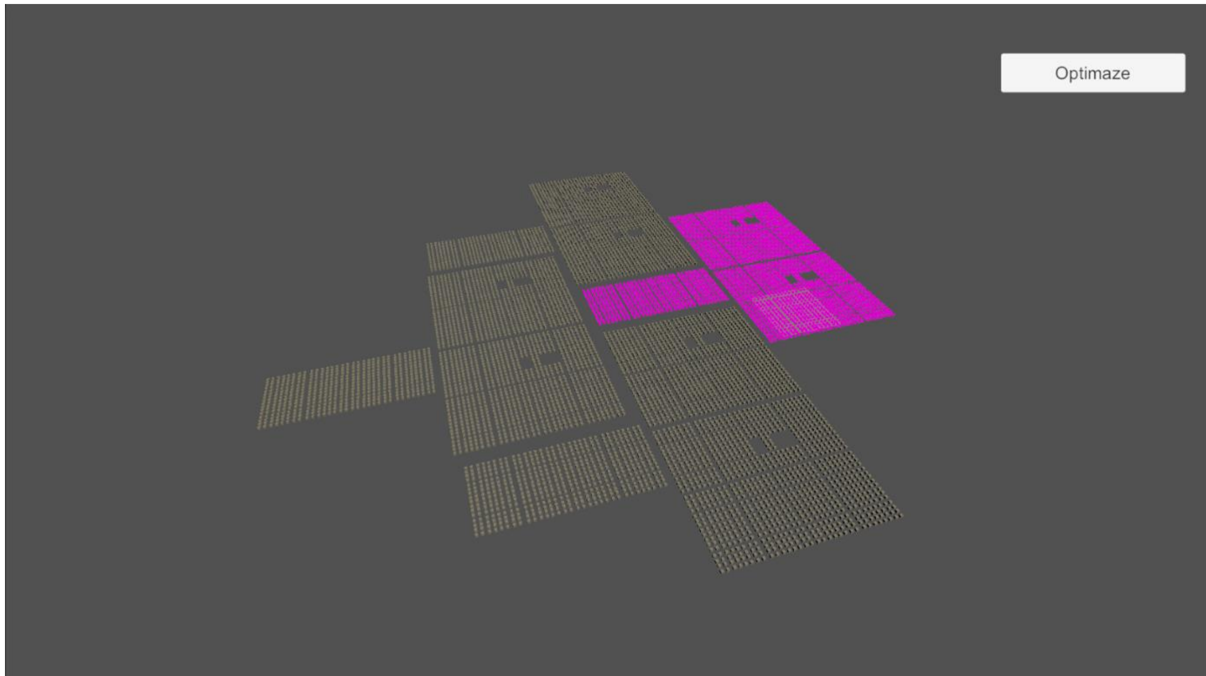

Figura 6.4.3 Escena de pruebas

6.4.2 LOD propio

Debido al gran número de agentes que están en movimiento en las ciudades continuamente, nos vimos obligados a implementar un sistema propio para gestionar los diferentes elementos que se ejecutan por cada uno de ellos, ya que, con las herramientas que nos proporciona Unity, sólo podíamos gestionar el nivel de detalle de las animaciones y de la geometría.

Para solucionar este problema de consumo que nos causaban los NPC's de las ciudades, hemos creado un script que nos permite de manera muy personalizable configurar LOD's o niveles de detalle. Los elementos que podemos configurar son: la distancia a la cual queremos que el LOD se active, los objetos, componentes y colliders que queremos que se deshabiliten y una lista de funciones que queremos que se ejecuten.

```
3 references
public float distance;
4 references
public GameObject[] objectsToDisable;
4 references
public Behaviour[] componentsToDisable;
4 references
public Collider[] collidersToDisable;
1 reference
public UnityEvent eventsToExecute;
```

Figura 6.4.4 Propiedades publicas LOD Propio

El único cálculo que necesita realizar nuestro script de forma continua es la distancia entre el jugador y cada uno de los NPC que tienen configurado nuestro script de LOD.

```
void Update()
{
    currentDistance = Vector3.Distance(originTransform.position, transform.position);
```

Figura 6.4.5 Calcular la distancia con el jugador

Si la distancia actual es mayor que la distancia que hemos configurado para que se active el LOD entonces se deshabilitaran los objetos, los componentes y los colliders indicados y se ejecutaran los eventos que hemos indicado en el inspector.

```
if (active && currentDistance > distance)
{
    active = false;
    for (int i = 0; i < objectsToDisable.Length; i++)
    {
        objectsToDisable[i].SetActive(false);
    }
    for (int i = 0; i < componentsToDisable.Length; i++)
    {
        componentsToDisable[i].enabled = false;
    }
    for (int i = 0; i < collidersToDisable.Length; i++)
    {
        collidersToDisable[i].enabled = false;
    }

    eventsToExecute.Invoke();
}
```

Figura 6.4.6 Ejecutar eventos distancia superada

Adicionalmente marcamos un booleano que hace de semáforo para no estar todo el rato desactivando los mismos elementos, sino que sólo se desactivaran la primera vez que el jugador supere la distancia. Si la distancia vuelve a ser menor a la indicada, se vuelven a activar todos los elementos y apagamos el semáforo, para que se pueda volver a desactivar en caso de que el jugador se aleje de nuevo.

```
if (!active && currentDistance < distance)
{
    active = true;
    for (int i = 0; i < objectsToDisable.Length; i++)
    {
        objectsToDisable[i].SetActive(true);
    }
    for (int i = 0; i < componentsToDisable.Length; i++)
    {
        componentsToDisable[i].enabled = true;
    }
    for (int i = 0; i < collidersToDisable.Length; i++)
    {
        collidersToDisable[i].enabled = true;
    }
}
```

Figura 6.4.7 Reactivar eventos por proximidad

A continuación, podemos ver las diferentes configuraciones que hemos probado y cuál ha sido la ganancia de rendimiento. Las configuraciones que usamos para testear este sistema de LOD propio fueron las siguiente: una resolución de 573X323, con 1000 NPCs caminando, con una distancia de activación de 50 unidades de Unity en un portátil Surface Laptop 3 en el editor de Unity. Los datos base que tomamos, es decir si usar este LOD personalizado, fue de 65 FPS de media con unos FPS mínimos de 30.

Los sets de componentes a desactivar eran los siguientes: El set 1 que contenía el componente mesh, el set 2 que contenía el mesh y animator, el set 3 que contenía el mesh, el animator y el collider; y el último set 4 con el mesh y el collider solo. Este último era para ver el impacto individual que causaba el componente collider. También probamos como afectaría tener varios niveles de distancia, para que los componentes se fueran desactivando uno a uno dependiendo de diferentes distancias, en lugar que todos a la vez. Tambien comparamos este Script con el propio LOD de Unity. El resultado de estas pruebas se puede comprobar en la Figura 6.4.8.

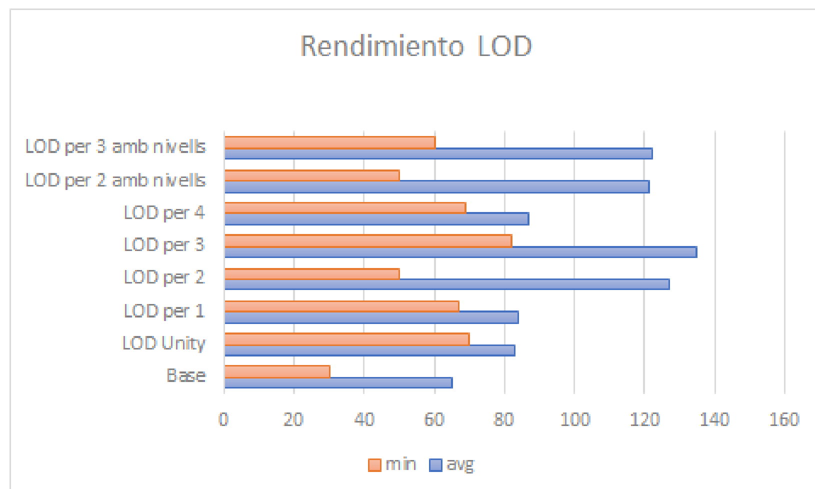

Figura 6.4.8 Grafico de barros

La conclusión que podemos sacar es que nuestro LOD para nuestro caso es mejor que el de Unity, que la idea de dividir la desactivación por diferentes niveles no es beneficiosa, y que nuestro punto óptimo es usar el set 3.

6.5 Personaje

6.5.1 Movimiento

El movimiento del personaje es un sistema sencillo de movimiento en 3r persona dependiente de la cámara. Una vez el jugador ha usado el input correspondiente para mover al personaje, se empieza calculando al ángulo objetivo con la función Mathf.Atan2(), multiplicándolo por el ángulo del eje y de la cámara. Para que el giro no se realice de golpe, haremos el giro más suave con la función Mathf.SmoothDampAngle(). Después, cogemos el ángulo objetivo, la versión que no es suave, y lo usamos para calcular la dirección del movimiento. Con esta dirección aplicamos la función move del character controller de Unity y el personaje realizará el movimiento.

```
if (direction.magnitude >= treshHoold)
{
    //Smothear el gir
    float targetAngle = Mathf.Atan2(direction.x, direction.z) * Mathf.Rad2Deg + _camera.eulerAngles.y;
    float angle = Mathf.SmoothDampAngle(transform.eulerAngles.y, targetAngle, ref turnSmothVelocity, turnSmothTime);
    transform.rotation = Quaternion.Euler(0f, angle, 0f);

    //Aplicar direcio de la camera
    Vector3 moveDir = Quaternion.Euler(0f, targetAngle, 0f) * Vector3.forward;

    //Moviment + gir
    controller.Move(moveDir.normalized * speed * Time.deltaTime);
}

animator.SetBool("Move", direction.magnitude >= treshHoold);
```

Figura 6.5.1 movimiento del personaje

6.5.2. Interacción

Para la interacción con el personaje existen dos tipos: La interacción por trigger y la interacción por raycast.

La interacción por trigger es la más simple. Simplemente consiste en acercarse al objeto y pulsar la tecla E. Esta interacción se utiliza por ejemplo en el sistema de diálogos o en el sistema de misiones para avanzar la trama.

El otro tipo de interacción posible consiste en crear un rayo desde el jugador y, dependiendo con que objeto colisione, realizar una acción diferente.

```
void Interact()
{
    if (State == CharacterState.Default)
    {
        RaycastHit hit;
        Debug.DrawRay(Camera.main.transform.position, Camera.main.transform.forward * 100, Color.red, 5f);

        if (Physics.Raycast(Camera.main.transform.position, Camera.main.transform.forward, out hit, Mathf.Infinity, interactableLayer))
        {
            CarroMovement carro = hit.collider.GetComponentInParent<CarroMovement>();

            if (carro)
            {
                carro.Mounted = true;
                animator.SetBool("Sit", true);
                carroObject = carro;

                State = CharacterState.Mounted;
                CanMove = false;

                return;
            }

            MercaderNPC npc = hit.collider.GetComponentInParent<MercaderNPC>();

            if (npc)
            {
                GetComponent<MercaderPlayer>().StartComerciar(npc);
                State = CharacterState.Trading;
                CanMove = false;
                return;
            }

            Pickable pickable = hit.collider.GetComponentInParent<Pickable>();

            if (pickable)
            {
                _carryObject = pickable;
                _carryObject.Pick();

                State = CharacterState.Carry;
                animator.SetBool("Carry", true);
                return;
            }
        }
    }
```

Figura 6.5.2 Fución Interact

6.6.1 Dibujar

La idea de que el jugador dibujara su propio mapa nació a comienzos del proyecto. La primera propuesta era que el jugador, con el trazo de su ratón, dibujara todos los elementos y luego pasar un algoritmo que estilizara estos dibujos.

Esta idea fue descartada bastante rápido, pues no sabíamos cómo hacer un algoritmo así, y tampoco queríamos confiar demasiado en las dotes artísticas del jugador. Además, que esto era un gran impedimento para jugadores con problemas a la hora de mover manos o brazos.

Al final nos decidimos en tener varios pinceles a escoger, cada uno para representar un elemento concreto del mapa. En principio, estos elementos serían montañas, arboles, ríos y masas de agua (lagos o mares). Y expandiríamos estos a medida que fuésemos necesitando más.

Los elementos cómo montañas o árboles fueron los más sencillos de implementar. Descargamos un pack de texturas varias de cartografía, los cuales usamos para representar estos elementos. Cuando el jugador clica sobre el mapa con uno de estos pinceles, primero se comprueba que no haya elementos ya creados en ese lugar, y luego se instancia el elemento en la posición del ratón. A este tipo de elementos los denominamos *stickers* por el funcionamiento parecido al de las pegatinas.

Los ríos y mares fueron un poco más complejos, puesto que estos no tienen ni forma ni tamaño definido. Para este tipo de elementos, decidimos recurrir a la entrada de formas por parte del jugador; pero estilizamos estos por código, para que el jugador no tenga que hacerlo. Para el mar, el jugador dibuja una forma con el ratón, y este será el perímetro. Cuando el jugador clica por primera vez se guarda la posición del ratón. Mientras lo va arrastrando, si la diferencia entre el último punto guardado y la posición actual es mayor a un umbral, se guarda esta posición al final de una lista. Cuando el jugador levanta el ratón, se crea un plano con estos puntos. Una vez el plano se ha creado, se le asigna el material correspondiente (en este caso son dos, uno encargado de estilizarlo para que parezca agua, y el otro encargado de la parte de borrar).

A la hora de estilizarlo, usamos la imagen, que se puede ver en la Figura 6.6.1, del pack de texturas antes mencionado. Usamos las coordenadas del plano en el mundo como coordenadas UV, y luego las usamos para poner la textura repetida sobre el plano.

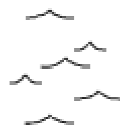

Figura 6.6.1 Textura de olas

Para que la repetición no se note tanto, desplazamos un poco cada hilera de textura en el eje X. Para darle dinamismo y efecto de agua, movemos la textura en el eje horizontal en el tiempo. El resultado final es el que se observa en la Figura 6.6.2.

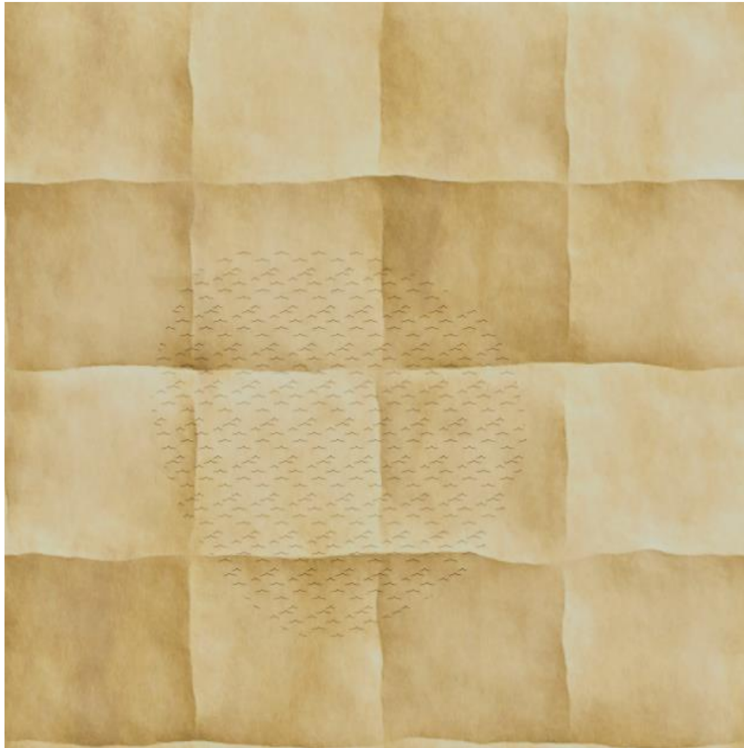

Figura 6.6.2 Ejemplo de agua en el mapa

Finalmente, para programar este efecto usamos el shadergraph de Unity, que se puede ver en la Figura 6.6.3.

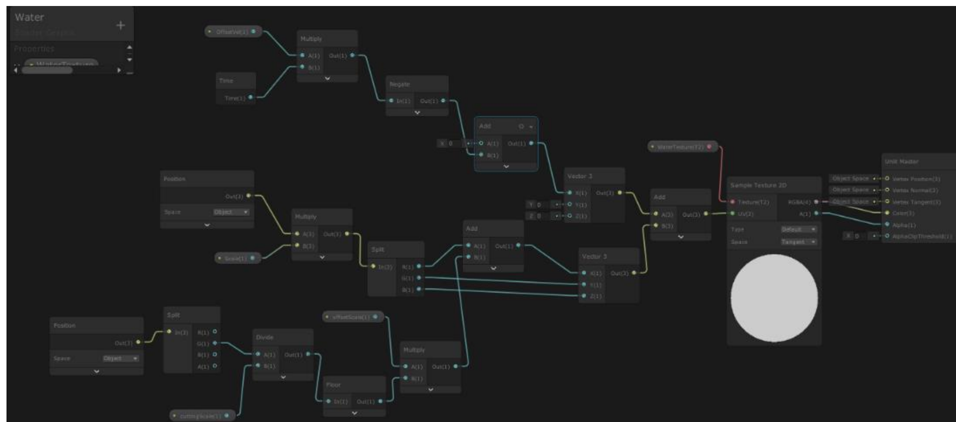

Figura 6.6.3 ShadeGraph shader mapa

A la hora de hacer los ríos, comenzamos igual, guardando el input del ratón del jugador en una lista, pero con un line-renderer (como se ve en la Figura 6.6.4) que dibuje la línea en pantalla mientras el jugador la va marcando.

```
if (_gO == null)
{
    _gO = new GameObject();
    _line = _gO.AddComponent<LineRenderer>();

    _line.startWidth = _line.endWidth = LINE_WIDTH;
    _line.positionCount = 0;
    _line.sharedMaterial = GetComponent<TexturePainter>().riverContorno;
}

_line.positionCount++;
Vector3 p = FindObjectOfType<TexturePainter>().sceneCamera.ScreenToWorldPoint(Input.mousePosition);
p.z = 0;
_points.Add(p);

_line.SetPositions(_points.ToArray());
```

Figura 6.6.4 Función para dibujar rio

Cuando este levante el ratón, convertiremos la línea en un mesh con la función LineRenderer.BakeMesh de Unity.

En relación al material, decidimos usar un color azul claro, tal y como se puede observar en la Figura 6.6.5.

Figura 6.6.5 Ejemplo de dibujar rio

6.6.2 Profundidad

De la forma en la que estaba el mapa en ese momento, no había ningún sentido de profundidad, y si dos elementos se solapaban, los dos dibujos se combinaban, creando un resultado poco claro y difícil de entender, como se observa en la Figura 6.6.6.

Figura 6.6.6 Problemas de profundidad y solapamiento

Con el objetivo de arreglar esto, modificamos las imágenes, rellenando el dibujo con blanco en la zona que tendría que haber color. Luego, dibujábamos esa zona con el color del fondo, mapeando las coordenadas del objeto en escala global a las coordenadas UV de este.

Para que el mapa se viese bien, queríamos que tuviera un cierto orden a la hora de ser dibujado, para que los elementos no se viesen sobrepuestos ni sucediesen cosas imposibles, cómo las que se pueden ver en la Figura 6.6.7.

Figura 6.6.7 Problemas de profundidad

Decidimos inventar una jerarquía que seguirían los elementos del mapa; y en caso de ser de la misma, tendría prioridad el *sticker* que más abajo estuviese, pues este estaría más cerca del jugador. Esta jerarquía es **montañas=arboles>ríos>lagos**. Con este orden buscábamos imitar cómo se ven los objetos a medida que se alejen en la distancia, haciendo que la parte baja del mapa fuese las más cercana y, de esta forma, se daría más profundidad al mapa.

Para ello, creamos un componente que ordenase los objetos en el eje z, que es el eje que apunta desde la cámara al plano, según una prioridad y la altura a la que estuviesen. Al no tener todos la misma altura, tuvimos que ordenarlos desde la parte de abajo en vez del centro, como se puede observar en la Figura 6.6.8.

```
_startPoint = cam.transform.position.z + cam.nearClipPlane + .5f;

float halfHeight = GetComponent<MeshFilter>().mesh.bounds.extents.y;
float yOffset = (transform.position.y + halfHeight) / 20;

transform.position = new Vector3(transform.position.x, transform.position.y,
    _startPoint + priority + yOffset);
```

Figura 6.6.8 Función para alinear por la parte baja

Añadimos este componente a los objetos en el momento que se crean, con la función AddComponent, por lo que sólo hace falta moverlos de lugar una vez. Siendo el resultado final el que se puede observar en la Figura 6.6.9

Figura 6.6.9 Ejemplo de profundidad

6.6.3 Borrar

Una de las funciones que también queríamos implementar era la creación de un sistema de borrado que permitiese al jugador equivocarse. La primera idea fue hacer un botón de deshacer, pero nos pareció demasiado técnico, muy parecido a un programa informático, por lo que decidimos implementar un borrador convencional.

Para conseguir este efecto, modificamos los materiales. Los shader tienen dos arrays de floats, uno con los valores de los puntos en el eje X y otro con los del eje Y, junto con un valor que guarda cuantos puntos hay, para poder recorrer los vectores, además de dos radios (uno interior y otro exterior), para que, a la hora de borrar, haya un borde suavizado.

Los puntos que guarda el shader están en coordenadas del mundo, y por cada pixel, calculamos la distancia a ese punto. Si el valor es menor al radio interior, el valor de la opacidad no se ve afectado. En cambio, si el valor es mayor al radio exterior, el valor de la opacidad es zero, y, en caso de que el valor este en este rango, reasignamos este valor entre 0 y 1, cómo se puede ver en la Figura 6.6.10.

```
fixed4 frag (v2f input) : SV_Target
{
    fixed2 worldUV = input.vWorld.xy;

    fixed4 col1 = tex2D(_MainTex, input.uv);
    fixed4 col2 = tex2D(_BackgroundTex, worldUV);

    fixed4 col = col1 * col2;

    float alpha = col1.a;

    for (int i = 0; i < _PointCount; i++)
    {
        fixed2 aux = fixed2(_PointsX[i], _PointsY[i]);
        float d = distance(aux, input.vWorld);

        d = clamp(d, _IRadius, _ORadius);

        d = remap(_IRadius, _ORadius, 0, 1, d);

        alpha *= d;
    }

    col.a = alpha;
    return col;
}
```

Figura 6.6.10 Vertex shader para borrar

Cómo los shaders no pueden modificar sus propiedades en tiempo de ejecución por sí solos, creamos un componente que les asignase los valores.

Este componente se encarga de encontrar el material encargado de borrar (cómo se ve en la Figura 6.6.11), pues los objetos pueden tener más de uno.

```
Material GetMaterial(Material[] materials)
{
    Material material = null;
    foreach (Material m in materials)
    {
        if (m.HasProperty(countName))
        {
            material = m;
            break;
        }
    }
    return material;
}
```

Figura 6.6.11 Función para obtener los materiales

Una vez tenemos el material, por cada frame en el que el jugador clique sobre un objeto con este componente, si la distancia entre este punto y el último guardado es mayor a un umbral, se añade este nuevo punto al vector del shader. Esto nos deja el resultado que se puede observar en la Figura 6.6.12.

Figura 6.6.12 Demostración de borrado

6.6.4 Guardar en disco

Para que los jugadores no puedan acceder al archivo del mapa y modificarlo o compartirlo, los datos se guardan binarizados. Cómo hemos mencionado antes, hay dos tipos de elemento a guardar: los *sticker* y los procedurales. Para los elementos de tipo sticker, con guardar la posición y el tipo de sticker, ya es suficiente. Además de esto, tambíen hay que guardar los puntos en los cuales el jugador ha borrado y el número de puntos de borrado, junto con el tipo de objeto que es, como se puede observar en la Figura 6.6.13.

```
public StickerElement(float x, float y, StickerType st, float[] xP, float[] yP, int np)
{
    this.positionX = x;
    this.positionY = y;
    this.stickerType = (int)st;
    this.xPoints = xP;
    this.yPoints = yP;
    this.eraserPoint = np;
}
```

Figura 6.6.13 Clase para guardado de Sticker

Sin embargo, para los elementos procedurales es un poco más complicado. El serializador de Unity no puede binarizar clases propias de Unity, como es el mesh, por lo que tenemos que pasar esto a un tipo de datos que si se puedan. Para eso, transformamos los datos a un array que guarde los vértices y otro con los triángulos, cómo se puede ver en la Figura 6.6.14.

```
public static void SerializableMeshInfo(Mesh m, out float[] vertices, out int[] triangles)
{
    vertices = new float[m.vertexCount * 3];
    for (int i = 0; i < m.vertexCount; i++)
    {
        vertices[i * 3] = m.vertices[i].x;
        vertices[i * 3 + 1] = m.vertices[i].y;
        vertices[i * 3 + 2] = m.vertices[i].z;
    }
    triangles = new int[m.triangles.Length];
    for (int i = 0; i < m.triangles.Length; i++)
    {
        triangles[i] = m.triangles[i];
    }
}
```

Figura 6.6.14 Función para serializar mesh

A la hora de cargar los datos, hacemos la conversión opuesta, con la función que se puede ver en la Figura 6.6.15. Una vez tenemos el mesh, se lo asignamos al objeto, y le añadimos el material dependiendo del tipo de objeto que es, el cual también guardamos.

```
public static Mesh GetMesh(float[] vertices, int[] triangles)
{
    Mesh m = new Mesh();
    List<Vector3> verticesList = new List<Vector3>();
    for (int i = 0; i < vertices.Length / 3; i++)
    {
        verticesList.Add(new Vector3(
                vertices[i * 3], vertices[i * 3 + 1], vertices[i * 3 + 2]
            ));
    }
    m.SetVertices(verticesList);
    m.triangles = triangles;

    m.RecalculateNormals();

    return m;
}
```

Figura 6.6.15 Función para reconstruir mesh

Junto con estos dos arrays, también guardamos los dos vectores con los puntos de borrado, el número de puntos de borrado y, cómo hemos mencionado antes, el tipo de objeto que es, cómo se observa en la Figura 6.6.16.

```
public ProceduralElement(UnityEngine.Mesh m, StickerType st, float[] xP, float[] yP, int np)
{
    Utility.Utility.SerializableMeshInfo(m, out this.vertices, out this.triangles);
    this.stickerType = (int)st;
    this.xPoints = xP;
    this.yPoints = yP;
    this.eraserPoint = np;
}
```

Figura 6.6.16 Clase para guardado de Procedural

6.7 Hierba

Al ser nuestro mundo de dimensiones tan grandes, y estar repleto de praderas, era necesario tener un sistema de hierba escalable y eficiente. Nuestra primera opción fue usar los sistemas de terreno del propio Unity.

La función de detalles (la predilecta para elementos pequeños) no está disponible en la tubería de renderizado en alta definición. Podíamos solventar este imprevisto usando la función de colocar árboles, con un par de cambios, para la hierba; pero no encontramos ningún asset que nos gustase ni que encajase con la estética del juego.

Tras buscar por internet, no encontramos con el asset que se puede ver en la Figura 6.7.1. Éste usa el sistema de partículas de GPU de Unity, el VFXGraph, por lo que no daría muchos problemas de rendimiento, y encajaba perfectamente en el estilo artístico del juego. El único contratiempo que había era que no es escalable, pero decidimos usar este asset igualmente, modificándolo.

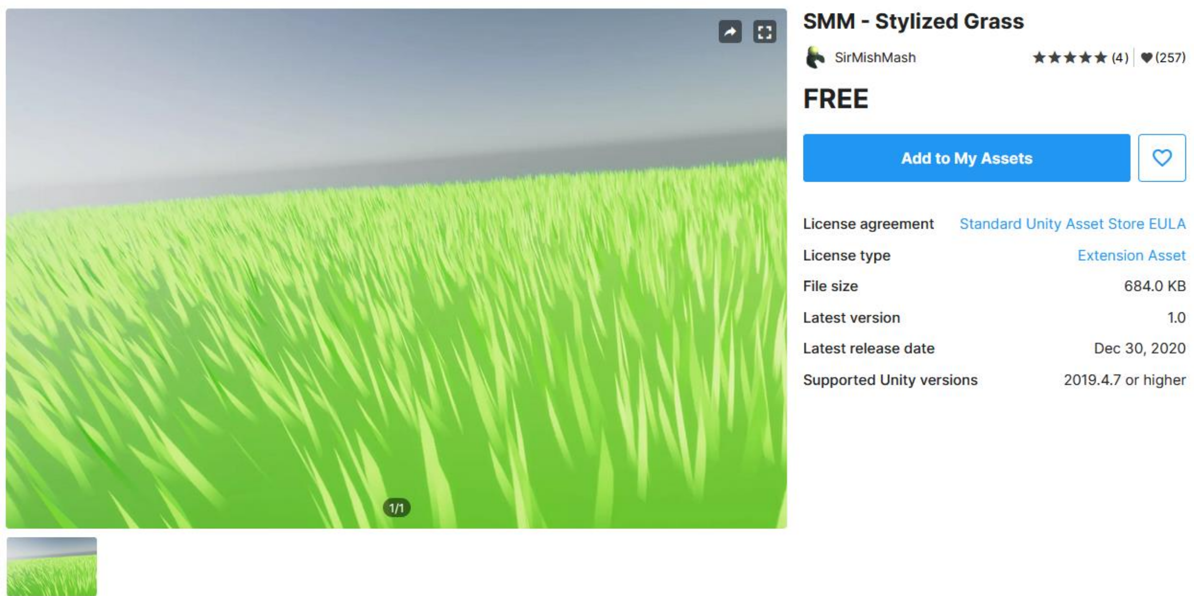

Figura 6.7.1 Asset de la hierba usada

Los problemas a solventar eran que la hierba se adaptase a la altura del terreno, y que tuviese un sistema de LOD integrado.

Altura del terreno

Para que la hierba se adaptase al terreno, usamos un heightmap (el que se puede ver en la Figura 6.7.2) para saber la altura a la cual tiene que colocarse.

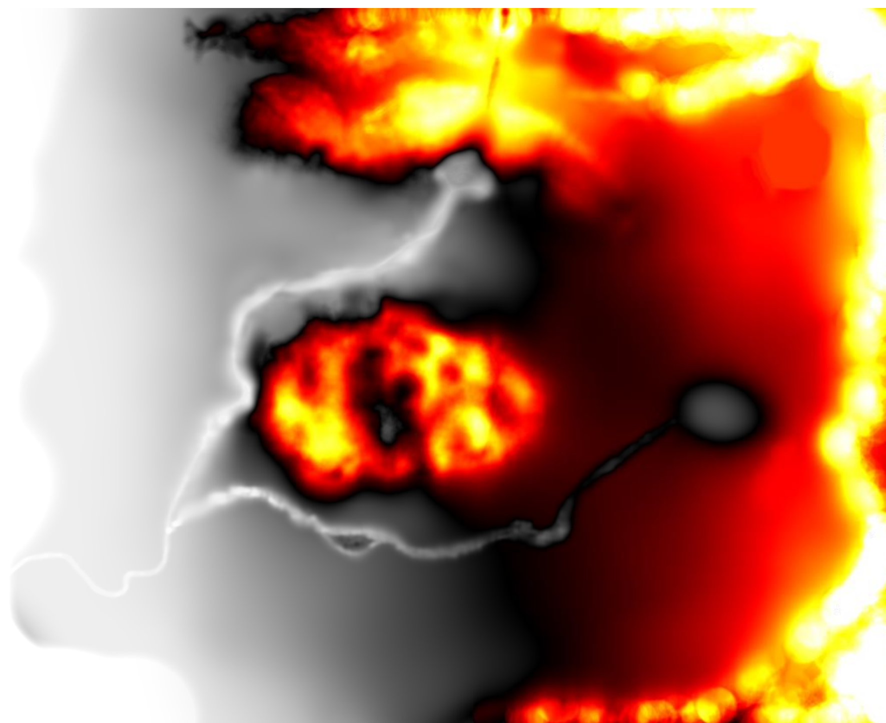

Figura 6.7.2 Mapa de altura

Para tener mayor rango, y con esto más precisión, el heightmap usa todos los canales de la imagen. El primer cuarto del rango de alturas va al canal alpha, el segundo al canal rojo, tercero al verde y el

último al azul, como se puede observar en la Figura 6.7.3. Con esto evitamos que la hierba quedase escalonada, y que siguiese una rampa ascendente recta, pues el valor máximo de cada píxel es 255 y la altura máxima del terreno es de 600.

```
float height = rawHeights[x, y];

float a = Mathf.Clamp(height       , 0, .25f);
float r = Mathf.Clamp(height - .25f, 0, .25f);
float g = Mathf.Clamp(height - .50f, 0, .25f);
float b = Mathf.Clamp(height - .75f, 0, .25f);

var color = new Vector4(r * 4, g * 4, b * 4, a * 4);
duplicateHeightMap.SetPixel(x, y, color);
```

Figura 6.7.3 Función para guardar mapa de altura usando los 4 canales

Una vez tenemos el heightmap, calculamos, a partir de la posición de una hebra de hierba en el mundo, su posición relativa en el mapa, sacamos el valor del color y extrapolamos la altura, como se puede observar en la Figura 6.7.4. Con esto actualizamos la posición de la hierba en el eje Y.

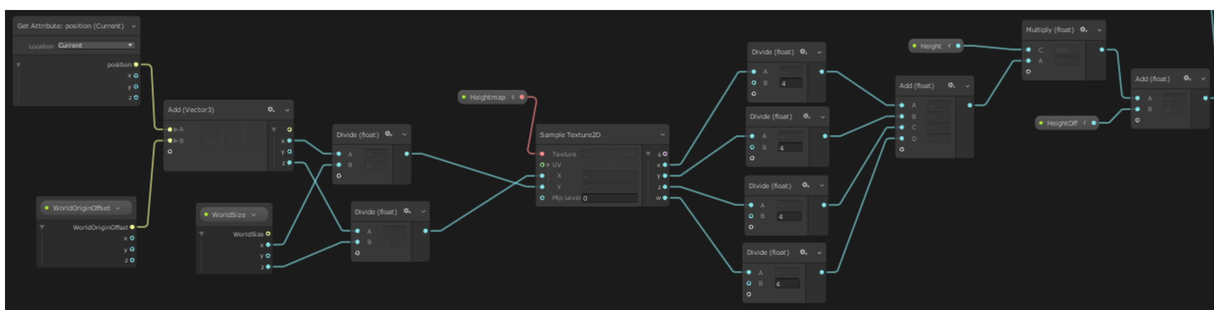

Figura 6.7.4 Shadergraph para calcular la altura de la hierba

El siguiente paso era implementar un sistema que nos permitiese decidir donde había hierba y donde no, pues queríamos que hubiese caminos, laderas y zonas sin hierba. Decidimos usar un planteamiento parecido al anterior. Por cada hebra, sacamos sus coordenadas en el mundo, y comprobamos esa coordenada en una textura que hace función de máscara, la que está en la Figura 6.7.5. Esta la dibujamos nosotros a mano, con referencia del mapa, para decidir por donde crece la hierba y donde no. Una vez tenemos el valor de la máscara, ponemos la escala de la hierba en función a dicho valor.

Figura 6.7.5 Máscara para la hierba

Con estas dos funciones, nos queda una hierba ajustable al terreno (cómo se ve en la Figura 6.7.6), bien en altura bien e inclinación en el terreno, bien en los lugares en donde ésta crece.

Figura 6.7.6 Demostración del césped

Sistema de LOD

Nuestro objetivo con la hierba era tener praderas densas, repletas de plantas allá donde se viese. Pero esto, teniendo en consideración el tamaño de nuestro mapa, es inviable, por lo que tuvimos que pensar un sistema de optimización, parecido al *Level Of Detail.*

Nuestro primer planteamiento fue separarlo en trozos más pequeños, que se fuesen cargando y descargando a medida que el jugador se acercase o se alejase. Esto no daba resultados óptimos, pues a la hora de cargarlos, el juego bajaba considerablemente la cantidad de fotogramas por segundo.

Al final nos decantamos por un sistema de hierba que "siguiese" al jugador. El objetivo era que cada hebra de hierba se mantuviese dentro de respecto una distancia del jugador, es decir, una vez la distancia supera un umbral, se pone al lado opuesto del jugador, simulando el efecto de hierba infinita.

Cómo este código funcionaría en la tarjeta gráfica, no queríamos usar *if statements*, pues no es lo más optimo en cuanto a eficiencia, además de lo poco estético que queda el nodo *branch* en programación visual. Teniendo esto en cuenta, nuestro algoritmo para reposicionar la hierba quedaría como se ve en la Figura 6.7.7. Restamos la coordenada X de la hebra con la del jugador y lo dividimos con la distancia máxima de renderizado de la hierba. Redondeamos este valor para que sea 1, 0 o –1, y lo negamos, puesto que queremos que la hierba cambie su posición a la opuesta a la que se encuentra. Convertimos este valor en entero, para evitar posibles errores, y obtenemos el signo. Multiplicamos este signo por la distancia máxima de renderizado y se la sumamos al valor actual.

Figura 6.7.7 Shadergraph para calcular la escala de la hierba

Podemos observar el funcionamiento en las dos Figuras 6.7.8 y 6.7.9.

Figura 6.7.8 Hebra siguiendo al jugador

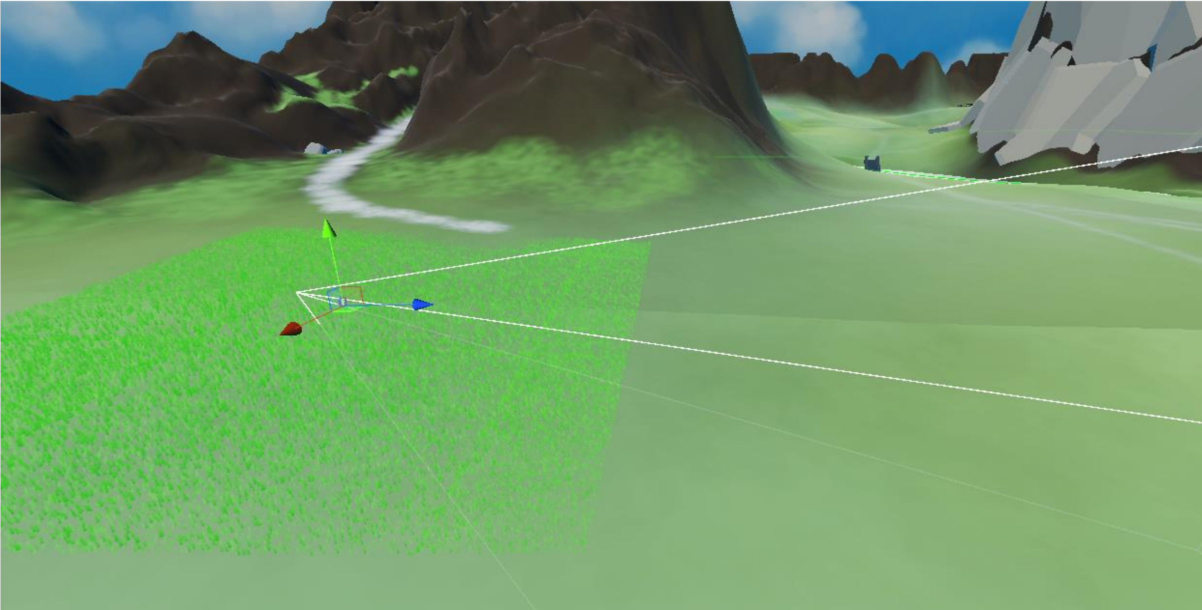
Figura 6.7.9 Hebra siguiendo al jugador (2)

Sin embargo, haciéndolo de esta manera nos dimos cuenta que un cuadrado con el jugador en el centro no era la mejor opción, ya que estamos renderizando mucha hierba en puntos no visibles para la cámara, así que buscamos la manera de optimizar el rango de visión de la hierba. Para ello, en vez de hacer que siguiera al jugador, hicimos que siguiera a un objeto, hijo del jugador, que tuviera un offset relativo a éste. La posición de este objeto se adapta en tiempo real para obtener el máximo rango de visión. Lo que hacemos es buscar el punto del terreno más cercano que ve la cámara con un raycast, y a esto le sumamos un offset definido por el diseñador en dirección del jugador al punto calculado con el raycast.

```
Vector3 GetCameraPosInGround()
{
    RaycastHit hit;

    Vector3 pos = cameraRef.transform.position;
    Vector3 dir = cameraRef.ScreenToWorldPoint(new Vector3(cameraRef.scaledPixelWidth / 2, 0, 10)) - pos;
    dir = dir.normalized;

    if (Physics.Raycast(pos - dir * 10, dir, out hit, 500, groundLayer, QueryTriggerInteraction.Ignore))
    {
        pos = hit.point;

        dir = (playerTransform.position - pos).normalized;
        pos += dir * offset.z + Vector3.Cross(hit.normal, dir) * offset.x;
    }

    return pos + Vector3.up * 0.1f;
}
```

Figura 6.7.10 Calcular posición visible más próxima

Como podemos ver en la Figura 6.7.11, con este sistema conseguimos aumentar el rango de visión de la hierba. Está hecho de esta manera y no haciendo un offset directamente desde el jugador porque así nos aseguramos de que siempre hay césped en el rango de visión de la cámara, ya que, aunque se mueva o varíe mucho la inclinación de la cámara, siempre será visible el césped.

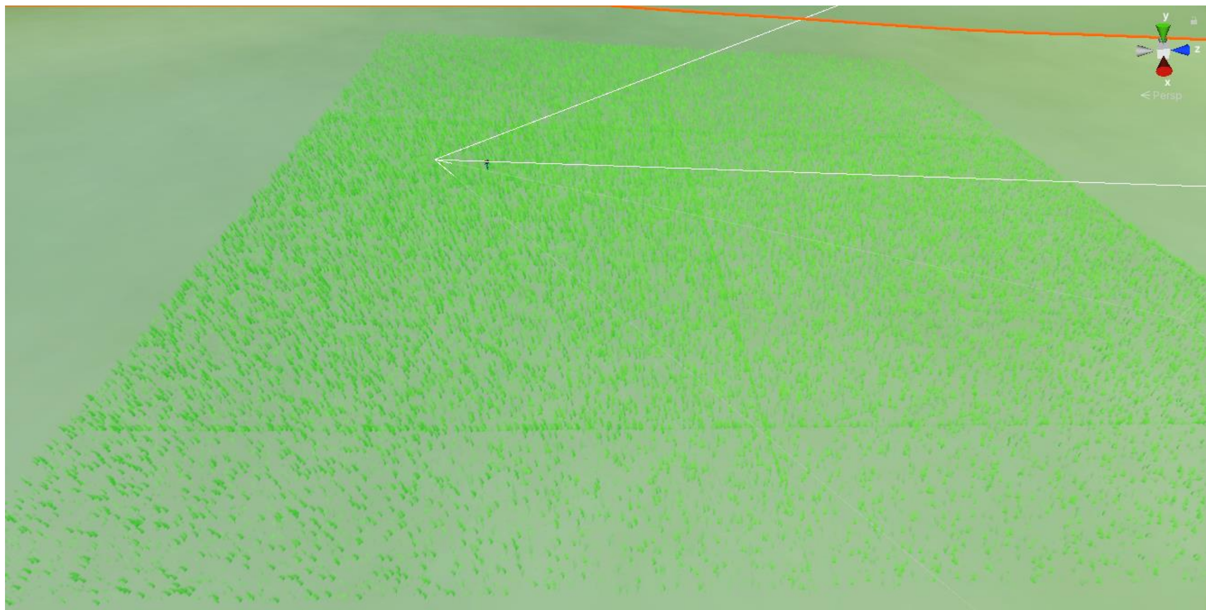

Figura 6.7.11 Hebra siguiendo al jugador con visibilidad mejorada

6.8 Mecánica compraventa

Nuestro objetivo al implementar el sistema de compraventa, no era sólo añadir un reto al jugador para la obtención de recursos, sino que también dar realismo y sensación de vida al juego. Para ello, pensamos que cada mercader fuese único, con un sistema de comercio preferido, un inventario propio, y unas preferencias de compra.

Con esto en mente, organizamos el código en la siguiente estructura: los mercaderes y el jugador heredan de una misma clase base, y existe una pizarra que guarda la información de todos estos. Así no hace falta que todos los mercaderes estén cargados al mismo tiempo, y tenemos que guardar en disco una única cosa.

6.8.1 Pizarra

Para la pizarra, tenemos una clase de instancia única (patrón *singleton*) con una lista que guarda todas las instancias de mercaderes. Aprovechando que todo está en una misma clase, esta es la que usamos para guardar y cargar de disco.

6.8.2 Gestión de clases

Cómo hemos mencionado antes, los mercaderes y el jugador heredan de una clase base MercaderObject. Esta se encarga de guardar los datos del mercader, tales como dinero, que objetos tiene, en cuales está interesado, que manera de comerciar prefiere y su nombre. Además de esto, también guarda con que mercader está comerciando y que objeto quiere comprar. Esta información la guardamos en esta clase, puesto que al principio queríamos que los mercaderes comerciasen entre ellos, para que el mundo se sintiese más vivo. Una vez implementado, no nos acabó de convencer la idea, por lo que acabos descartándola.

Los Mercaderes tienen el componente MercaderNPC. Este, cómo hemos mencionado antes, puede comerciar con otros Mercaderes. Para esto, detecta todos los mercaderes en un radio a la redonda, y encuentra el primero que tenga un objeto que le interese y que se pueda permitir, y lo compra.

En cuanto al jugador, este tiene el componente MercaderPlayer. La gran parte de la gestión de compraventa se hace en esta clase.

6.8.3 Implementación compraventa

La mecánica de compraventa está dividida en dos fases: la de escoger si el jugador comprará o venderá, y cual objeto será; y la del minijuego que determina a qué precio se efectúa el intercambio.

El proceso para todos los minijuegos es el mismo: una vez el jugador apriete el botón de interactuar y se detecte un mercader en frente, se obtienen las técnicas (minijuegos) de comercio que usa este comerciante y se activan para que estén listas a la hora de usarlas. Después se activa la interfaz de compraventa, que explicaremos más adelante, y esperamos a que el jugador decida qué acción quiere hacer.

Si decide que quiere comprar o vender un objeto, se activan los minijuegos. Todos estos, al finalizar devuelven un valor entre 0 y 1, con los cuales se saca una media, dado que puede haber más de un minijuego simultáneamente, que dicta cuanto benefició obtiene el jugador.

Dado que no todos los minijuegos duran el mismo tiempo, cada uno de ellos notifica al objeto MercaderPlayer cuando ha acabado, y cuando el número de minijuegos acabados sea el mismo a los minijuegos disponibles, si finaliza la compra o venta (cómo observamos en la Figura 6.8.1), dependiendo de cada caso, con la transacción de dinero correspondiente y se reinician todos los valores y desactivan objetos correspondientes para devolver el control al jugador.

```
public void Notificar(float ratio)
{
    _notifiedCount++;

    _tradeFactor = (_tradeFactor + ratio) / _notifiedCount;

    if (_notifiedCount == _tecnicas.Count)
    {
        _notifiedCount = 0;
        if (TradeCanvas.buy)
        {
            FinalizarCompra(_tradeFactor);
        }
        else
        {
            FinalizarVenta(_tradeFactor);
        }
        _tecnicas.Clear();
        _interestedItem = null;
        _tradingWith = null;

        foreach (Transform t in transform)
        {
            if (t.GetComponent<ICapitalismo>() == null)
                continue;

            t.gameObject.SetActive(false);
        }

        GetComponent<ThirdPersonMovement>().SetMove(true);
    }
}
```

Figura 6.8.1 Función para notificar fin de compra

Implementación minijuegos

Los minijuegos representan la manera de regatear del jugador, y su habilidad para sacar buenos precios por los productos. Hay tres tipos, el de simón dice, el de reflejos y el de control de altura. Todos estos están gestionados por hijos al objeto del jugador, en la jerarquía que se puede ver en la Figura 6.8.2, para que el componente de MercaderPlayer pueda acceder fácilmente a ellos.

Figura 6.8.2 Jerarquía minijuegos

El minijuego de simón es una implementación del clásico juego simón dice. En este se le aparecerán una secuencia de inptus al jugador, y este tendrá que volver a meterlos siguiendo el mismo orden. A la hora de comenzar el juego, se genera la secuencia aleatoria recursivamente, usando la función Invoke, la cual permite llamar a funciones tras un periodo de tiempo, cómo podemos observar en la Figura 6.8.3. Las opciones para esta secuencia son direcciones (arriba, abajo, izquierda o derecha) que serán las teclas que tendrá que pulsar el jugador, y se guardan en una lista para comprobar que la respuesta del jugador sea la correcta. Para dar a saber que opción a tocado, se muestra una flecha en pantalla que se rota para indicar la dirección al jugador.

```
void SelectOption()
{
    int i = Random.Range(0, options.Length);

    simon.Add(options[i]);
    arrowT.transform.rotation = Quaternion.Euler(0, 0, -90 * i);

    idx++;
    if (idx > max)
    {
        state = State.answering;
        idx = 0;
        arrowT.SetActive(false);
        return;
    }

    Invoke("SelectOption", 2);
}
```

Figura 6.8.3 Función para escoger opción de Simon

Una vez se ha seleccionado toda la secuencia, se cambia a la fase de respuesta. En esta, el jugador tiene que meter la respuesta correcta. Para confirmar la respuesta del jugador, tenemos una variable que sirve de índice para comprobar si el valor actual es el mismo al entrado por el jugador. En caso de ser así, se avanza el índice; por lo contrario, se le notifica a MercaderPlayer que el minijuego ya ha sido acabado con el resultado del minijuego. Si el minijuego se completa se le notifica al jugador.

```
bool correct = Input.GetKeyDown(simon[idx]);

if (correct)
{
    idx++;
}
else
{
    state = State.iddle;
    NotifyParent();
}

if (idx >= max)
{
    simon.Clear();
    state = State.selecting;

    idx = 0;
    NotifyParent();
}
```

Figura 6.8.4 Función para comprobar input

El juego de los reflejos es parecido al de simón, puesto que en este también hay que meter una secuencia de direcciones, pero se premia hacerlo lo más rápido posible. Al comienzo se escoge una dirección aleatoria, y se rota una flecha para que el jugador sepa el botón que tiene que pulsar, siempre y cuando queden intentos para hacer esto, como se ve en la Figura 6.8.5. En caso contrario se acaba el minijuego y se notifica a MercaderPlayer.

```
void ChangeArrow()
{
    _idx = Random.Range(0, options.Length);

    arrowT.transform.rotation = Quaternion.Euler(0, 0, 90 * _idx);

    if (_try >= max)
    {
        _waitin = false;
        arrowT.SetActive(false);
        NotifyParent();
        _try = _correct = 0;
    }
}
```

Figura 6.8.5 Función para cambiar figura de la flecha

A la hora de comprobar el resultado, sólo se tiene en cuenta si el jugador a pulsado la tecla correcta dentro de un tiempo límite, sin tener cuantos intentos haya hecho dentro de ese umbral. Cada fotograma se comprueba si el jugador ha entrado la respuesta, en cuyo caso se reinicia el tiempo por intento, se suma un correcto, se añade un intento y se escoge una nueva dirección. En caso contrario, si el tiempo se ha acabado, se aumenta el número de intentos, se reinicia el tiempo y se cambia de dirección, como se ve en la Figuras 6.8.6.

```
if (Input.GetKeyDown(options[_idx]))
{
    _dT = 0;
    _correct++;
    _try++;
    ChangeArrow();
}
else
{
    _dT += Time.deltaTime;

    if (_dT > MAX)
    {
        _dT = 0;
        _try++;
        ChangeArrow();
    }
}
```

Figura 6.8.6 Comprobación de input

El minijuego de control de altura es el más diferente de los tres. En este el jugador no tiene que repetir unas instrucciones que le han dado, sino que tiene que mantener un cuadrado dentro de otro que se mueve aleatoriamente. Al comienzo de este minijuego se instancian dos cuadrados, uno más grande que el otro; el grande se mueve aleatoriamente en el eje vertical (al comienzo hacia una posición que se le asigna aleatoriamente), mientras que el jugador controla el segundo intentando mantenerlo dentro, y a los diez segundos el minijuego se acaba. El objetivo es intentar que los dos cuadrados se solapen el mayor tiempo posible.

Cada fotograma se mueve el cuadrado grande hacia la dirección, y si ha llegado a él se le asigna una nueva posición a la que ir. En cuanto al cuadrado más pequeño, se mueve en la dirección que ha

indicado el jugador en el eje Y, y se comprueba si está dentro del área del cuadrado objetivo, como se ve en la Figura 6.8.7. En caso de ser así, se suma el tiempo.

```
customT.transform.localPosition = Vector3.Lerp(customT.transform.localPosition,
    new Vector3(customT.transform.localPosition.x, targetH, customT.transform.localPosition.z), Time.deltaTime);

if (Mathf.Abs(customT.transform.localPosition.y - targetH) < .5f)
{
    targetH = Random.Range(-heghtOffset, heghtOffset);
}

if (Input.GetKey(KeyCode.UpArrow))
{
    pVel = Mathf.Min(30, pVel + .1f);
}

if (Input.GetKey(KeyCode.DownArrow))
{
    pVel = Mathf.Max(-30, pVel - .1f);
}

customP.transform.position += new Vector3(0, pVel * Time.deltaTime, 0) ;

if (customT.GetComponent<Collider2D>().OverlapPoint(new Vector2(customP.transform.position.x, customP.transform.position.y)))
{
    okiTime += Time.deltaTime;
}
```

Figura 6.8.7 Función para posicionar cuadrado

Implementación interfaz

Como hemos mencionado antes, el jugador escoge que objeto quiere comprar o cual quiere vender desde una interfaz de venta. Esta permite visualizar una imagen, el nombre y el rango de precio del objeto al jugador.

La interfaz está dividida en dos ventanas, el panel de opciones, en el cual se puede escoger si se quiere comprar o vender objetos; y la lista de objetos a intercambiar.

El panel de opciones es una lista de botones *Toggle* (una casilla de verificación que le permite al usuario cambiar una opción a prendido o apagado). Cuando cualquiera de estos cambia de valor, llama a una función (dependiendo la opción que sea, como se ve en la Figura 6.8.8) y configura la tienda en valor a los objetos propios o del mercader con quien se está intercambiando.

```
public void BuyItems(Toggle t)
{
    if (!t.isOn)
        return;
    buy = true;
    AddItems(_npc.MercaderData.items);
}

public void SellItems(Toggle t)
{
    if (!t.isOn)
        return;
    buy = false;
    AddItems(_player.MercaderData.items);
}
```

Figura 6.8.8 Funciones para cambiar funcionalidad compraventa

Una vez en la función AddItems, primero vacía la interfaz de las opciones viejas y después, por cada elemento en la lista se instancia un objeto genérico de interfaz, tal cual se ve en la Figuras 6.8.9, al

cual se le modifican los valores para que este acorde con la información del objeto en cuestión. Este objeto se hace hijo de un componente de Unity que se encarga de ordenar y colocarlos.

```
foreach (Item i in items)
{
    GameObject aux = Instantiate(itemPrefab);

    ItemUI iu = aux.GetComponent<ItemUI>();
    iu.SetValues(i);

    aux.transform.SetParent(grid.transform);
    aux.transform.localScale = new Vector3(1, 1, 1);
    aux.transform.position = new Vector3(aux.transform.position.x, aux.transform.position.y, 0);
    aux.transform.localPosition = new Vector3();
}
```

Figura 6.8.9 Función para colocar los objetos

Para finalizar la compra, los elementos de la interfaz tienen un botón de compra, la cual llama a una función que encuentra al jugador, comprueba si este lo puede comprar y comienza la operación de compra o venta, dependiendo el modo, cómo se puede ver en la Figura 6.8.10.

```
public void Buy()
{
    MercaderPlayer mp = FindObjectOfType<MercaderPlayer>();

    if (mp == null)
    {
        Debug.LogError("Player not found");
        return;
    }

    if (TradeCanvas.buy && _reference.CanBeBought(mp.MercaderData.money))
    {
        Debug.Log("No enough money, es el mercado amigo");
        return;
    }

    if (TradeCanvas.buy)
    {
        mp.BuyItem(_reference);
    }
    else
    {
        mp.SellItem(_reference);
    }
}
```

Figura 6.8.10 Función para ejecutar compraventa

6.8.4 Editor de economía

Desde un comienzo queríamos tener un sistema de economía complejo, con varios objetos y varios mercaderes diferentes. El problema que nos causaba esto es que había que crear cada asset a mano y esto daba mucho margen a error y era muy difícil arreglarlos, por lo que decidimos hacer un editor dentro de Unity para tener todo en un mismo lugar.

Con este fin, creamos un objeto que muestra todos los objetos de tipo Mercader. Para eso buscamos todos los assets de tipo Mercader en los archivos cona la función FindAssets, y los guardamos en una

lista, cómo observamos en la Figura 6.8.11. El parámetro de FindAssets es un filtro para ayudar con la búsqueda, y le pasamos el tipo con la letra t.

```
private void RefreshAssets()
{
    string[] guids = AssetDatabase.FindAssets("t: Mercader");
    _reference.mercaders.Clear();
    foreach (string guid in guids)
    {
        string path = AssetDatabase.GUIDToAssetPath(guid);

        Mercader m = AssetDatabase.LoadAssetAtPath<Mercader>(path);

        if (m != null)
        {
            _reference.mercaders.Add(m);
            EconomyBlackboard.Get.AddMercader(m);

            if (m.items == null)
                m.items = new List<Item>();

            if (m.interestedIn == null)
                m.interestedIn = new List<Item>();

            if (!_toggleValues.ContainsKey(m))
                _toggleValues[m] = false;
        }
    }
}
```

Figura 6.8.11 Función para encontrar Assets de tipo Mercader

Recorremos esta lista mostrando el nombre de los mercaderes, un botón que nos permita ver más detalles expandiendo la información y otro botón para editar el mercader en cuestión. Para llevar la cuenta de que mercaderes hay que mostrar exactamente, usamos un Diccionario de clave Mercader y valor bool, cómo se puede ver en la Figuras 6.8.12.

```
GUILayout.BeginHorizontal();
_toggleValues[m] = EditorGUILayout.Foldout(_toggleValues[m], m.id);

if (GUILayout.Button("Edit"))
{
    Debug.Log("editting " + m.id);
    MercaderEditor.ShowEditWindow(m);
}

GUILayout.EndHorizontal();

if (_toggleValues[m])
    ShowMercader(m);
```

Figura 6.8.12 Expandir y editar mercaderes

En caso de querer mostrar más información, enseñamos al usuario cuanto a dinero tiene el mercader, que técnicas de comercio usa, que objetos tiene y cuales quiere comprar, pero no le dejamos editarlos al usuario. Para eso tiene que pulsar el botón de editar, el cual abre una ventana aparte para ello. Al

final de todo también hay un botón que permite crear un mercader, abriendo la misma ventana para poder editarlo.

La ventana de editar mercaderes recibe la referencia al mercader que se está editando, para así poder guardar los cambios en el disco duro. Al mercader se le puede modificar el nombre, la cantidad de dinero que tiene, los minijuegos, objetos y objetos en los que está interesado, y cómo antes, podemos expandir o minimizar estos campos para centrarnos únicamente en lo que nos interesa.

Para modificar nombre y dinero cambiamos las variables con las funciones de editor de Unity mismo. Con los demás atributos modificamos la lista que tiene el mercader guardando esta información.

En el caso de los minijuegos, es una lista de enum, en el cual podemos modificar cada valor en ella. También podemos eliminar el último elemento añadido o añadir uno nuevo, siempre y cuando no estén todas las opciones ya asignadas, cómo se puede ver en la Figuras 6.8.13.

```
for (int i = 0; i < _reference.tradeMinigames.Count; i++)
{
    EditorGUILayout.BeginHorizontal();
    TradeMinigame m = _reference.tradeMinigames[i];
    EditorGUILayout.LabelField(m.ToString());

    _reference.tradeMinigames[i] = (TradeMinigame)EditorGUILayout.EnumPopup(m);
    EditorGUILayout.EndHorizontal();
}

if (UnityEngine.GUILayout.Button("Add"))
{
    TradeMinigame aux = new TradeMinigame();
    TradeMinigame last = System.Enum.GetValues(typeof(TradeMinigame)).Cast<TradeMinigame>().Last();

    while (_reference.tradeMinigames.Contains(aux) && aux < last)
    {
        aux++;
    }
    _reference.tradeMinigames.Add(aux);
}
if (UnityEngine.GUILayout.Button("Remove"))
{
    EditorUtility.SetDirty(_reference);

    _reference.tradeMinigames.RemoveAt(_reference.tradeMinigames.Count - 1);
}
```

Figura 6.8.13 Función para añadir, modificar o quitar minijuegos

Para modificar las listas con los objetos y objetos en los cuales está interesado, recorremos la lista, mostrando un botón de modificar, el cual abre una nueva ventana cómo con los mercaderes. Al final también se muestra un botón de añadir, el cual abre la misma ventana.

Esta nueva ventana nos permite modificar el nombre y el rango de precio de los objetos, además de guardarlo estos en disco.

6.9 Editor de materiales

Como hemos mencionado antes, para los personajes descargamos un paquete de modelos ya creado, el que se puede ver en la Figura 6.9.1. Había un par de problemas, el primero era que no nos acababa de encajar la paleta de colores, y segundo, cada personaje sólo tenía una versión; no había variedad de colores, y eso no acababa de encajar en el gran mundo que queríamos crear. Decidimos hacer un editor de materiales dentro de Unity para solventar esto.

Figura 6.9.1 Paquete de personajes de Quaternion

Al ser unos personajes *low poly*, es decir, de bajos polígonos, decidimos que poder cambiar el color de cada material sería suficiente para hacer varias variaciones. Para esto, programamos dos maneras de poder escoger color: la primera es escoger el color exacto que tendrá el material, y la segunda es escoger un gradiente y que el color se escoja aleatoriamente al empezar el juego. Esta segunda sirve para ahorrar tiempo de trabajo con los personajes secundarios, pues no hace falta hacer una permutación por cada color, sino que es suficiente seleccionar un gradiente con el rango de colores deseados.

Para ambos casos usamos los *MateriaPropertyBlock* de Unity. Estos son assets que se ponen al *renderer* que sirven para modificar los valores de una propiedad de un material sin hacer una copia de este, ahorrando así espacio. A pesar de que esto complique el código, ayuda mucho en la optimización, pues modificar literalmente un material instancia una copia de este para poder mantener el original, creando esto un gran gasto en memoria.

Al cargar el componente, este recorre todos los materiales que tenga el modelo, creando una lista de MaterialData, una clase que guarda un color y un identificador del material, cómo se puede observar en la Figura 6.9.2. Cómo el propio componente es quien guarda esta lista, y esta se puede serializar, nos ahorramos el trabajo de tener que guardar esta información en disco.

```
public class MaterialData
{
    public int materialHashCode;
    public Color color;
}
```

Figura 6.9.2 Clase de MaterialData

A la hora de modificar los colores, recorremos todos los materiales del modelo, cómo antes hemos mencionado. Por cada material podemos escoger si queremos que el color sea el seleccionado, o uno aleatorio escogido de un gradiente.

En caso de que sea un fijo, obtenemos el *MaterialPropertyBlock* del material en cuestión con la función GetPropertyBlock. Si no hay ninguno asignado, este devuelve uno vacío, en cuyo caso le añadimos el color actual del material. Una vez tenemos el *PropertyBlock* listo, dejamos escoger color al usuario, se lo ponemos al *MaterialPropertyBlock* y le asignamos este al renderer. Después guardamos los cambios en disco. Todo este proceso se puede observar en la Figura 6.9.3.

```
void SelectColor(Renderer renderer, Material[] materials, Material mat, int index)
{
    Color newColor = Color.white;

    MaterialPropertyBlock mpb = new MaterialPropertyBlock();

    renderer.GetPropertyBlock(mpb, index);

    if (mpb.isEmpty)
    {
        mpb.SetColor(MaterialEditor.ColorName, mat.GetColor(MaterialEditor.ColorName));
        colors[index] = mat.GetColor(MaterialEditor.ColorName);
    }

    newColor = EditorGUILayout.ColorField(mpb.GetColor(MaterialEditor.ColorName));

    mpb.SetColor(MaterialEditor.ColorName, newColor);

    colors[index] = newColor;

    renderer.SetPropertyBlock(mpb, index);

    Save(_target);
}
```

Figura 6.9.3 Función para escoger color para material

Por el contrario, en caso de escoger gradientes, el usuario tendrá la opción de escoger más de un gradiente, puesto que así hay más colores posibles.

Para modificar estos, recorremos los gradientes posibles de este material, y en caso de que el jugador decida modifica alguno, guardamos estos cambios, como se puede observar en la Figura 6.9.4

```
for (int i = 0; i < gradients[index].Count; i++)
{
    Gradient old = gradients[index][i];
    gradients[index][i] = EditorGUILayout.GradientField(gradients[index][i]);

    if (old != gradients[index][i])
        Save(_target);
}
```

Figura 6.9.4 Función para escoger gradiente para material

A la hora de escoger un color dentro de las posibilidades, por cada material se obtiene un gradiente aleatorio dentro de los posibles (en caso de que no haya ninguno disponible mostramos un mensaje de error y salimos de la función), y de este obtenemos un color aleatorio. Creamos un *MaterialPropertyBlock,* le asignamos el color escogido y asignamos este al *renderer* antes de guardar los cambios, como se puede ver en la Figura 6.9.5.

```
if (GUILayout.Button("Select new color"))
{
    if (gradients != null && gradients[index].Count > 0)
    {
        newColor = gradients[index][Random.Range(0, gradients[index].Count)].Evaluate(Random.Range(0f, 1f));

        MaterialPropertyBlock mpb = new MaterialPropertyBlock();

        mpb.SetColor(MaterialEditor.ColorName, newColor);

        colors[index] = newColor;

        renderer.SetPropertyBlock(mpb, index);

        Save(_target);
    }
```

Figura 6.9.5 Función para asignar color de gradiente

Como hemos mencionado antes, de guardar los colores de cada material pues están en una lista en el propio componente, pero para guardar los gradientes, al no poder Unity serializarlos, tuvimos que hacer una clase propia que se encargara de ello.

Primero de todo, como Unity no puede serializar una lista anidada, tuvimos que crear una clase que sirviese de envoltorio a la lista para que no hubiese problemas. Con esto solucionado, decidimos usar la función Gradient.SetKeys de Unity. Esta recibe dos listas, una de *GradientColorKey* y otra de *GradientAlphaKey*. La primera tiene un color y un valor entre 0 y 1 que hace referencia a la posición en el gradiente de dicho color. La segunda también tiene un valor entre 0 y 1, pero en lugar de un color tiene un valor numérico representando el valor alpha de dicho color.

Para mantener el código lo más sencillo posible, decidimos que nuestros gradientes sólo pudiesen guardar dos colores, uno en cada esquina. Con esto en cuenta, recorremos todos los gradientes, y por cada uno obtenemos el color del comienzo y del final. Guardamos estos en dos listas, como se muestra en la Figuras 6.9.6, la primera encargada de guardar el comienzo de los gradientes, y las segunda el final.

```
public void SetGradientData(List<List<Gradient>> data)
{
    gradients1 = new List<ListWrapper>();
    gradients2 = new List<ListWrapper>();

    for (int i = 0; i < data.Count; i++)
    {
        List<Gradient> lg = data[i];
        gradients1.Add(new ListWrapper());
        gradients2.Add(new ListWrapper());

        for (int j = 0; j < lg.Count; j++)
        {
            Gradient g = lg[j];

            CustomGradient g1 = new CustomGradient();
            g1.color = g.colorKeys[0].color;
            g1.time = g.colorKeys[0].time;

            CustomGradient g2 = new CustomGradient();
            g2.color = g.colorKeys[1].color;
            g2.time = g.colorKeys[1].time;

            gradients1[i].Add(g1);
            gradients2[i].Add(g2);
        }
    }
}
```

Figura 6.9.6 Función para guardar gradiente

Teniendo en cuenta que, a la hora de guardar el objeto, guardamos los valores en un formato que Unity no entiende, tenemos que volver a hacer la conversión al inicializar el objeto. Para esto, por cada lista dentro de la lista, recorremos los gradientes que nosotros hemos guardado en nuestro formato. De estos obtenemos el color y la posición en el gradiente, bien en la posición inicial bien en la final, y se la asignamos al gradiente en cuestión. Añadimos estas en una lista para que el usuario pueda modificarlas, como se observa en la Figuras 6.9.7.

```
for (int i = 0; i < _target.gradients1.Count; i++)
{
    gradients.Add(new List<Gradient>());

    for (int j = 0; j < _target.gradients1[i].list.Count; j++)
    {
        Gradient g = new Gradient();

        GradientColorKey[] gk = new GradientColorKey[2];
        gk[0].color = _target.gradients1[i][j].color;
        gk[0].time = _target.gradients1[i][j].time;

        gk[1].color = _target.gradients2[i][j].color;
        gk[1].time = _target.gradients2[i][j].time;

        GradientAlphaKey[] ak = new GradientAlphaKey[2];
        ak[0].alpha = 1;
        ak[0].time = 0;

        ak[1].alpha = 1;
        ak[1].time = 1;

        g.SetKeys(gk, ak);
        gradients[i].Add(g);
    }
}
```

Figura 6.9.7 Función para reconstruir gradiente

6.10 Cascada

Para crear la cascada de nuestro videojuego lo primero que hicimos es modelar en Blender una cascada, como la de la Figura 6.10.1.

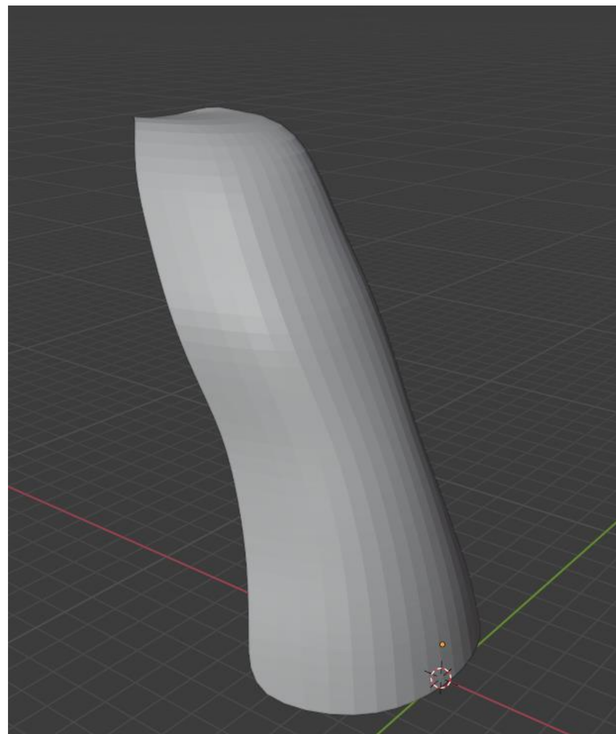

Figura 6.10.1 Modelo cascada

Después de modelar la cascada, creamos el shader con el sistema Shader Graph de Unity. Para hacer el movimiento de agua de la cascada, lo primero que hicimos es crear un ruido de Voronoi con movimiento, aplicarle una potencia y luego hacer un remap para eliminar los valores negros y ponerles un mínimo de blanco. Ver Figura 6.10.2.

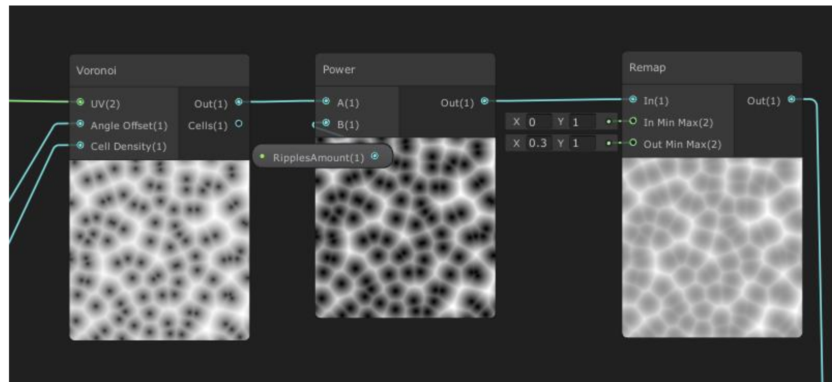

Figura 6.10.2 Nodos movimiento agua cascada

Seguidamente, utilizamos el canal verde del UV y lo invertimos. A este le aplicamos una potencia para aislar los valores blancos a la parte inferior. Después le multiplicamos un ruido simple para darle un poco de textura y le hicimos un remap para darle más brillo en los tonos blancos. El resultado de este proceso lo sumamos al resultado del proceso anterior, del ruido de Voronoi. Ver Figura 6.10.3.

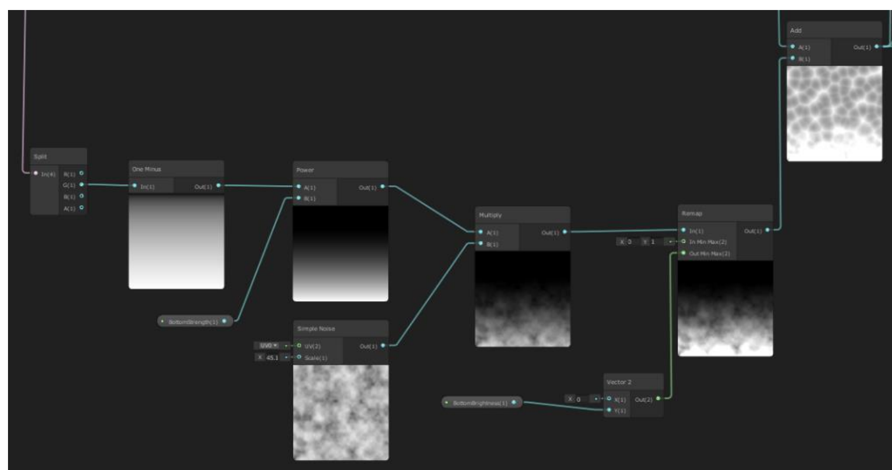

Figura 6.10.3 Nodos brillo parte inferior cascada

Al resultado de este cálculo le multiplicamos el color indicado por el diseñador: lo multiplicamos por el color del vértice y lo sacamos por el canal de color. Ver Figura 6.10.4.

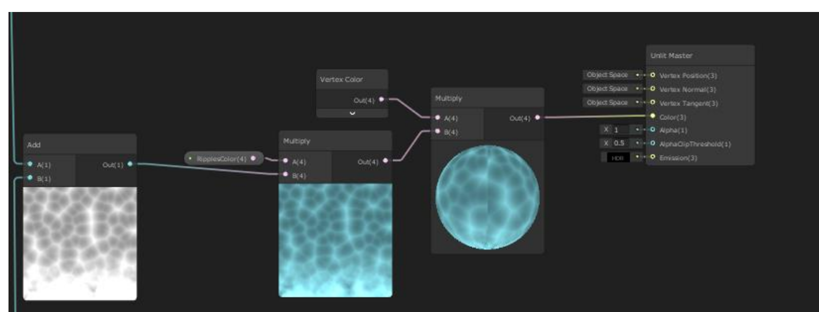

Figura 6.10.4 Nodos agregar color agua

En la Figura 6.10.5 podemos ver el resultado final de aplicar este shader a la cascada y en la Figura 6.10.6 podemos ver la configuración utilizada para obtener ese resultado.

Figura 6.10.5 Cascada con el material aplicado

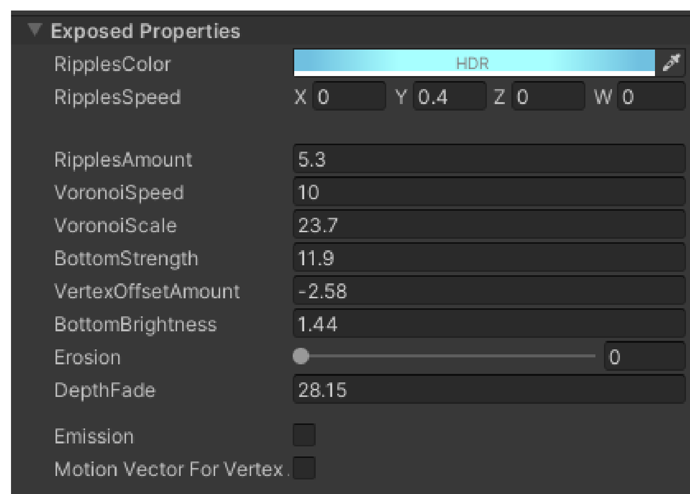

Figura 6.10.6 Propiedades material cascada

Luego le añadimos espuma para darle más brillo y movimiento. Para ello duplicamos el shader y a éste le añadimos tres variaciones: le aplicamos un valor de erosión que le dará valor al *alpha* de salida, sacamos por el canal de posición de vértices del nodo de salida la posición de los vértices sumándole el ruido de Voronoi en movimiento, y sacamos el ruido de Voronoi invertido por el valor *alpha clip threshold*. Ver variación en la Figura 6.10.7.

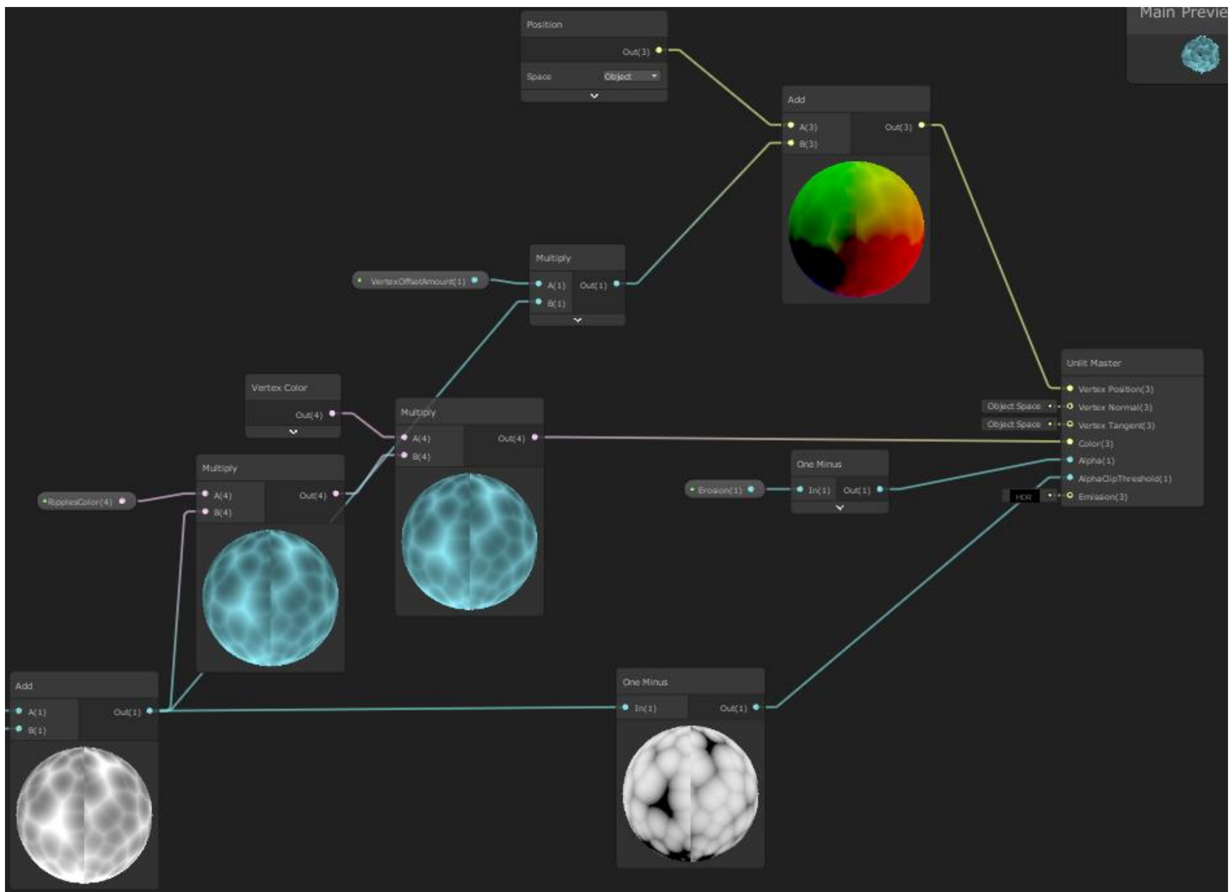

Figura 6.10.7 Variación nodos para añadir espuma

El resultado final de la cascada lo podemos ver en la Figura 6.10.8.

Figura 6.10.8 Cascada con espuma

6.11 Rio

En esta sección explicaremos la implementación del sistema que hemos utilizado para crear los dos ríos que atraviesan el mapa de nuestro videojuego.

6.11.1 Generar mesh

Para poder crear la mesh de los ríos adaptamos una librería gratuita, hecha por el youtuber y divulgador Sebastian Lagüe, publicada en el Asset Store de Unity con el nombre Bézier Path Creator[6]. Ver Figura 6.11.1.

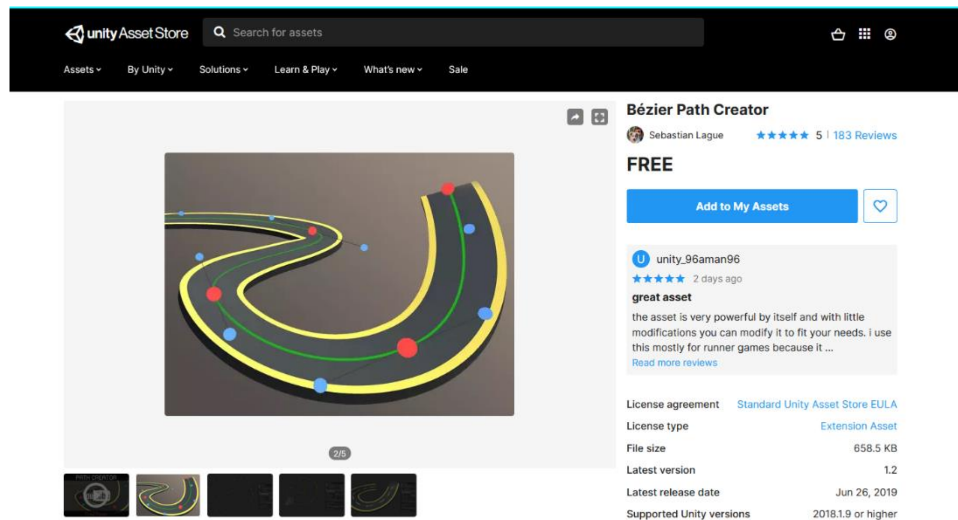

Figura 6.11.1 Página asset store de Bézier Path Creator

Esta librería la creó Sebastian Lague para explicar en su canal de Youtube como programar en Unity una herramienta para crear carreteras utilizando una curva de Bézier y así poder hacer diseño de niveles rápidamente. Ver figura 6.11.2.

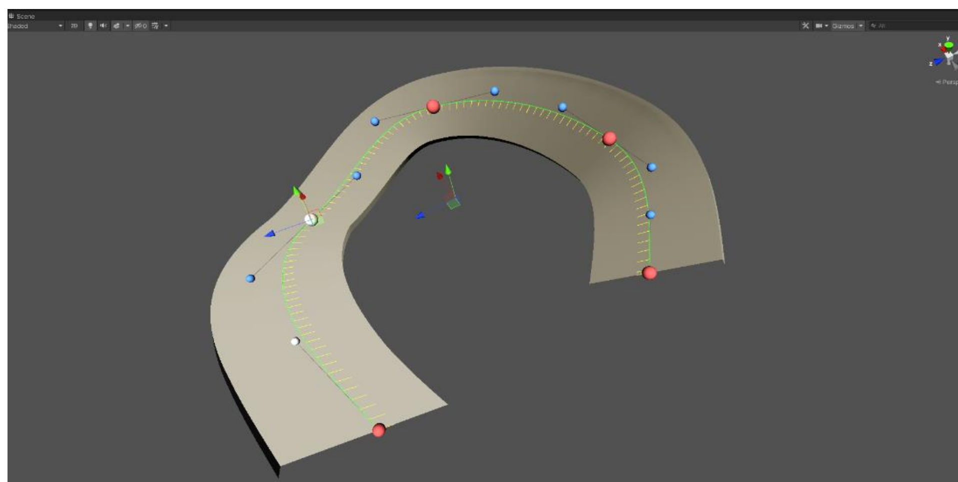

Figura 6.11.2 Ejemplo Bézier path creator

A nosotros nos interesaba este sistema de poder diseñar un camino a partir de una curva de bezier, para hacer así los ríos de forma que se adaptaran perfectamente al terreno sin tener que modelarlos en un programa de modelado y poder trabajar de manera más ágil y adaptable. La librería de Sebastian

está pensada para que toda la mesh tenga el mismo grosor, pero como podemos ver en la Figura 6.11.3, para nosotros eso supone un problema, ya que el río tiene anchos variables a lo largo de su camino y, al ser un terreno irregular, hacía que sobresaliera el río sobre el terreno.

Figura 6.11.3 Mesh creado sobresaliendo por el terreno

Para solucionar esto hemos optado por adaptar la librería respetando al máximo la estructura e interfaces del autor original. Para que el generador se adapte a nuestras necesidades, hemos hecho que, por cada punto del camino, podamos definir un grosor específico, haciendo una interpolación en los segmentos que une con los puntos vecinos para que no quede un corte muy brusco.

Para ello, lo primero que hemos hecho es añadir una lista de grosores que va a tener un valor por cada punto del camino, 0 si es el valor por defecto del camino o un valor definido si queremos modificar el grosor del punto, y hemos modificado el inspector para añadir un campo que nos permita modificar esos valores.

```
var currentWidth = creator.bezierPath.GetAnchorWidth(anchorIndex);
var newWidth = EditorGUILayout.FloatField("Width", currentWidth);
if (newWidth != currentWidth) {
    Undo.RecordObject(creator, "Set Width");
    creator.bezierPath.SetAnchorWidth (anchorIndex, newWidth);
}
```

Figura 6.11.4 Añadir campo para modificar ancho mesh

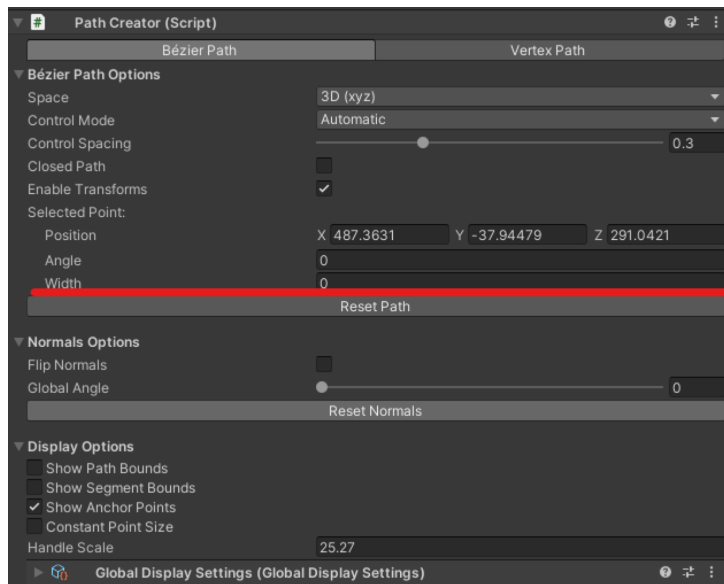

Figura 6.11.5 Inspector con el campo de ancho añadido

A la hora de crear el mesh, el script recoge la información de cada punto y divide el segmento en múltiples puntos con la finalidad de suavizar las curvas. Como hemos comentado anteriormente, nosotros interpolamos el grosor de esos puntos que genera para que cree una variación del grosor del mesh más suave, como podemos ver en las figuras 6.11.6 y 6.11.7.

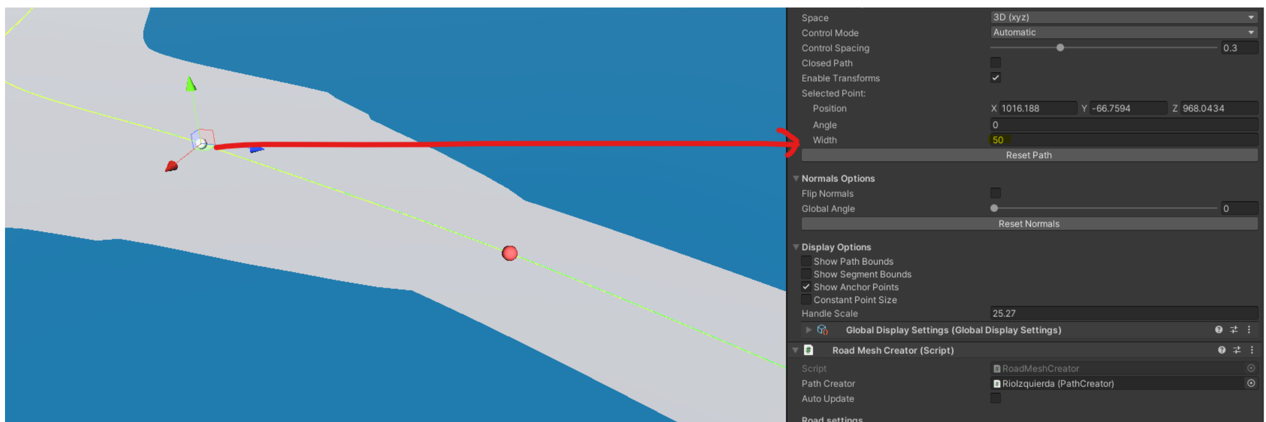

Figura 6.11.6 Configurar ancho mesh variable (1)

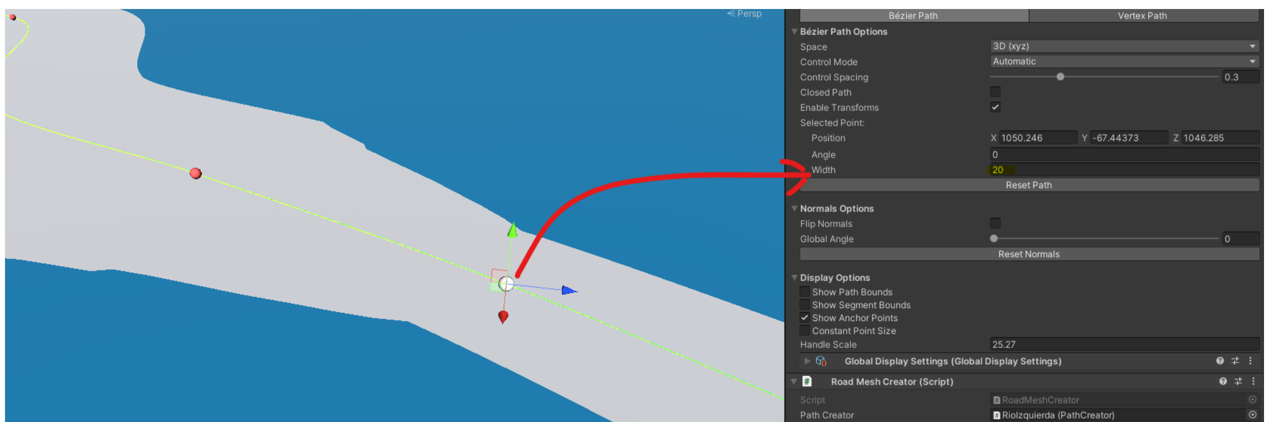

Figura 6.11.7 Configurar ancho mesh variable (2)

Para realizar la interpolación, simplemente cogemos el grosor de un punto, el grosor del siguiente punto y, a la hora de crear las subdivisiones, hacemos la interpolación lineal entre los dos, basándonos en la distancia que hay entre uno y el otro.

```
float startWidth = bezierPath.GetAnchorWidth(anchorIndex);
float endWidth = bezierPath.GetAnchorWidth(nextAnchorIndex);

for (int i = 0; i < num; i++) {
    int vertIndex = startVertIndex + i;
    float t = i / (num - 1f);
    float angle = startAngle + deltaAngle * t;
    Quaternion rot = Quaternion.AngleAxis (angle, localTangents[vertIndex]);
    localNormals[vertIndex] = (rot * localNormals[vertIndex]) * ((bezierPath.FlipNormals) ? -1 : 1);

    localWidths[vertIndex] = Mathf.Lerp(startWidth, endWidth, t);
}
```

Figura 6.11.8 Interpolación del ancho entre puntos

En la Figura 6.11.9 se puede apreciar como quedaría la mesh generada si no interpoláramos los valores de grosor.

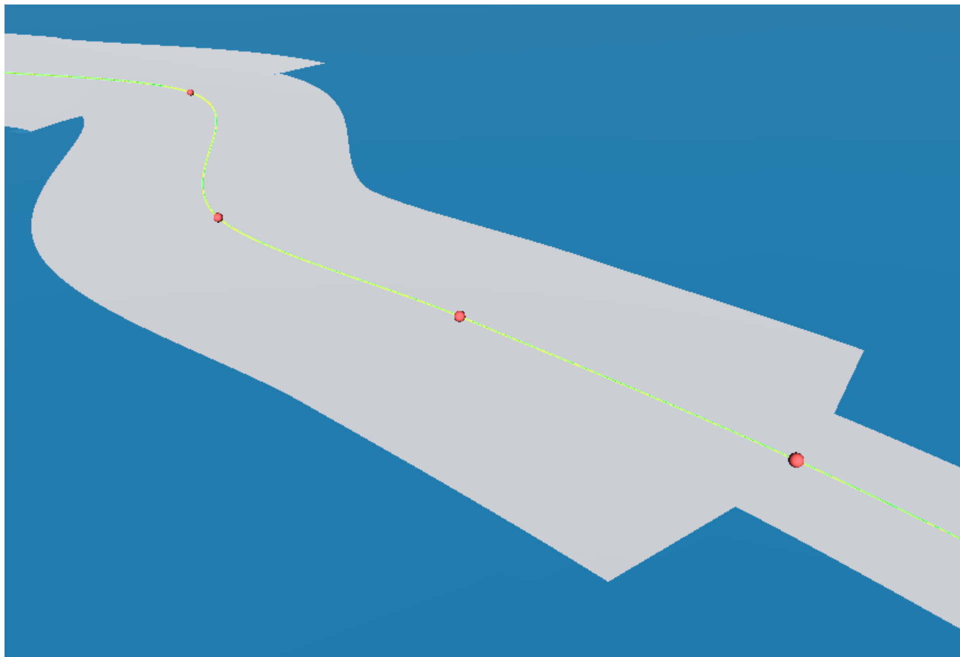

Figura 6.11.9 Ancho del mesh sin interpolación

Para apreciar mejor como se genera la mesh basándose en los puntos del camino de Bézier, podemos observar la Figura 6.11.10 donde podemos ver las aristas del mesh generado.

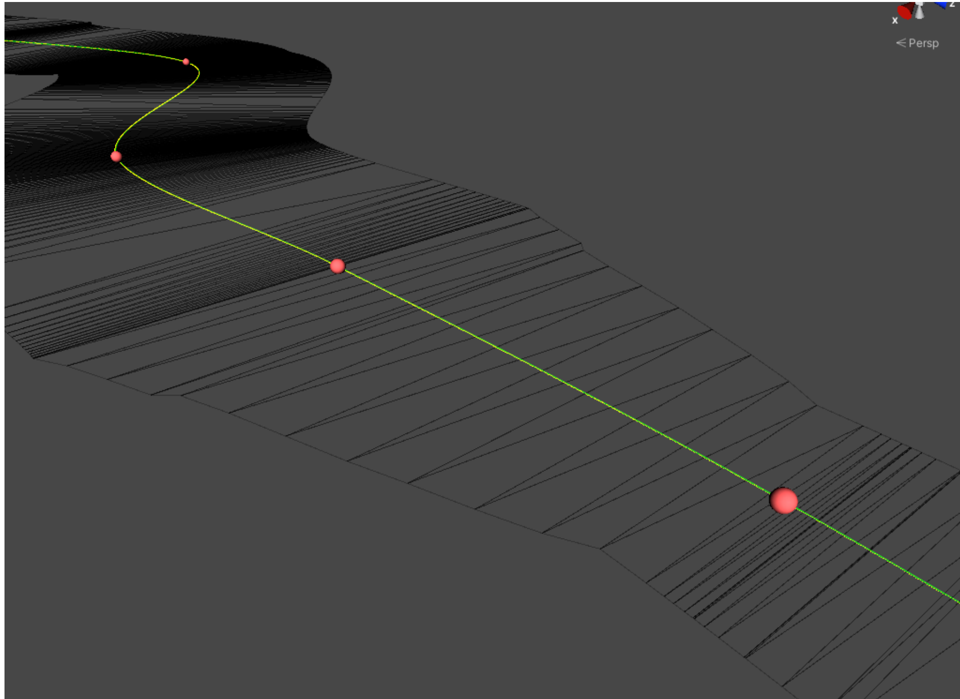

Figura 6.11.10 Ancho del mesh con interpolación

Un problema que nos encontramos a la hora de añadir los grosores personalizables es que el script que generaba el mesh no detectaba el grosor 0 como el grosor por defecto y no dibujaba los tramos que no se había puesto un grosor personalizado. Para solucionarlo, antes de crear el mesh le asignamos a todos los puntos que tengan un valor de grosor 0 el valor por defecto, y después de generarlo lo volvemos a poner a valor 0, porque si no, al cambiar el valor por defecto, estos no se verían afectados, ya que no los entendería como valores predeterminados.

```
for (int i = 0; i < pathCreator.bezierPath.NumAnchorPoints; i++) {
    if (pathCreator.bezierPath.GetAnchorWidth(i) == 0f)
        pathCreator.bezierPath.SetAnchorWidth(i, roadWidth);
}

for (int i = 0; i < path.NumPoints; i++)
{
    // Triangular el mesh
    // ....
}

for (int i = 0; i < pathCreator.bezierPath.NumAnchorPoints; i++) {
    if (pathCreator.bezierPath.GetAnchorWidth(i) == roadWidth)
        pathCreator.bezierPath.SetAnchorWidth(i, 0f);
}
```

Figura 6.11.11 Solucionar error al tener anchuras predeterminadas

En la Figura 6.11.12 se puede ver el resultado final de crear la mesh de uno de los ríos usando este sistema.

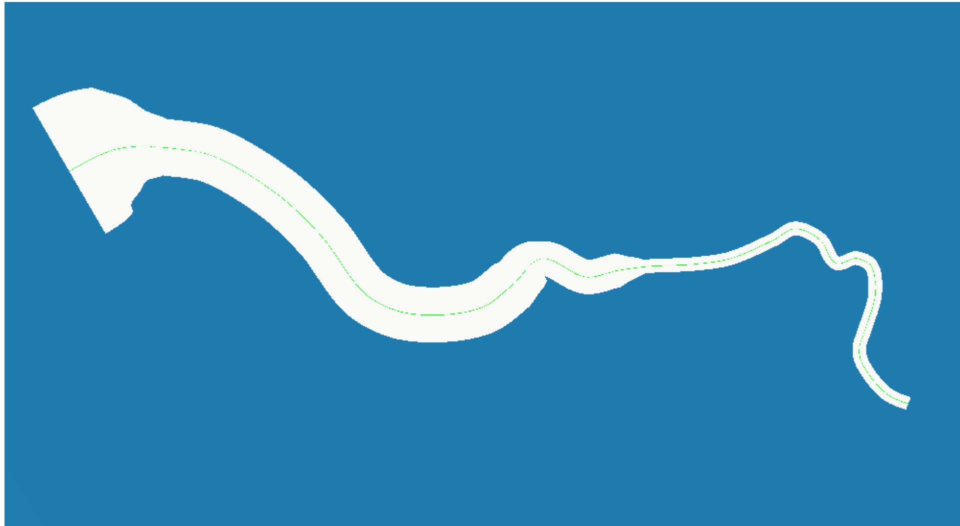

Figura 6.11.12 Mesh final del río

El principal problema que nos hemos encontrado al adaptar la librería de Sebastián ha sido entender e interpretar su código, ya que tenemos maneras muy diferentes de programar y ha usado una programación con muchas dependencias que ha hecho más lento entender el funcionamiento, ya que hemos tenido que hacer muchos saltos entre ficheros.

6.11.2 Shader agua

Los shaders de agua los hemos hecho utilizando el Shader Graph de Unity y las propiedades que hemos configurado son:

- **Refracción**: Añadir refracción de la luz para dar sensación de que el agua está en movimiento. Aquí, el diseñador puede definir la velocidad de refracción y la fuerza de esta. Para ello lo que hacemos es generar un ruido de gradiente con movimiento respecto a la velocidad que ha definido el diseñador. Luego, lo convertimos a un mapa de normales, que multiplicamos por la fuerza que ha definido el diseñador, y finalmente sumamos el mapa de normales al mesh.

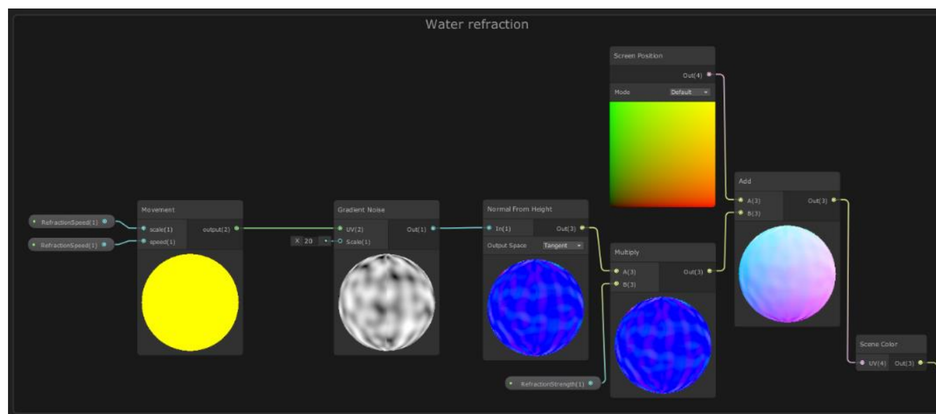

Figura 6.11.13 Nodos refracción agua

- **Color basado en profundidad:** Nos permite modificar el color del agua según lo profunda que sea, para poder hacer más oscuras las zonas donde menos llegaría la luz. El diseñador puede definir la profundidad máxima del agua, el color del agua profunda y el color del agua superficial. Para calcular el color del agua hacemos una interpolación lineal entre el color de la superficie y el de profundidad basándonos en la profundidad extraída de la escena.

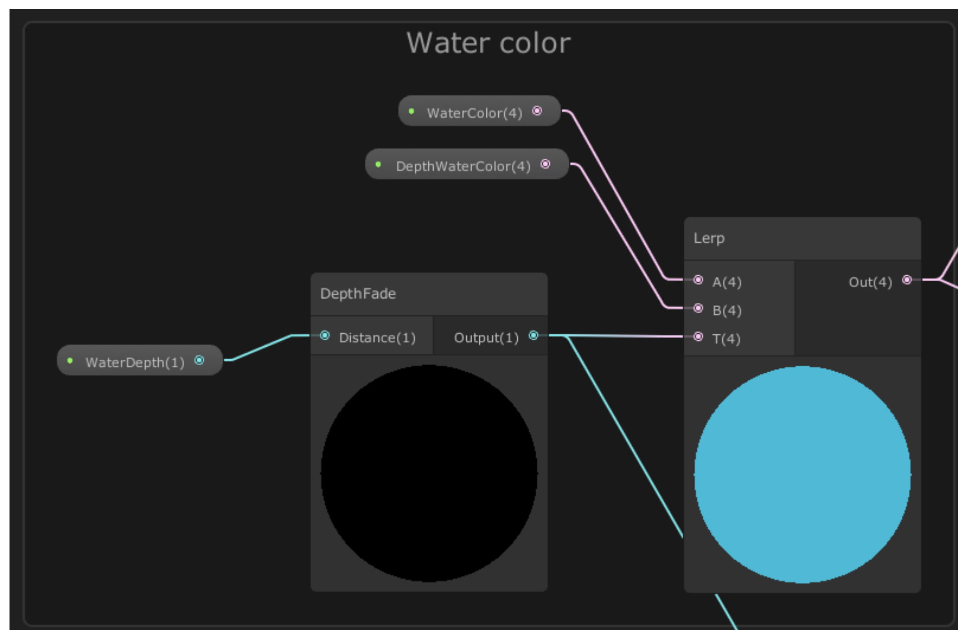

Figura 6.11.14 Nodos color basado en profundidad

- **Ondulaciones:** Ondulaciones blancas para darle un toque estilizado al agua y aumentar la sensación de movimiento de ésta. El diseñador puede definir el tamaño y la velocidad de las ondulaciones. Para hacer las ondulaciones generamos un ruido de Voronoi en movimiento relativo a la velocidad definida el diseñador, y le aplicamos una potencia de 10 para quedarnos únicamente con las zonas más blancas del resultado.

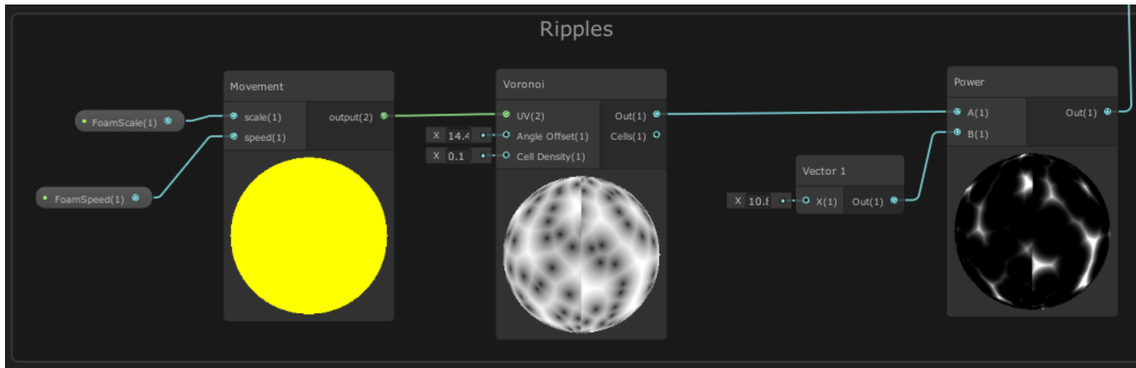

Figura 6.11.15 Nodos ondulaciones agua

- **Espuma:** Espuma blanca generada por contacto, en la superficie del río, con la tierra y otros objetos o elementos. El diseñador puede definir la velocidad y escala de la espuma, a que profundidad se corta la espuma y el color de la misma. Utilizamos, otra vez, la profundidad extraída de la escena y la multiplicamos por la profundidad de corte. El valor resultante lo binarizamos utilizando el threshold extraído de un ruido de Voronoi. Para finalizar, todo esto se multiplica por el color definido por el diseñador.

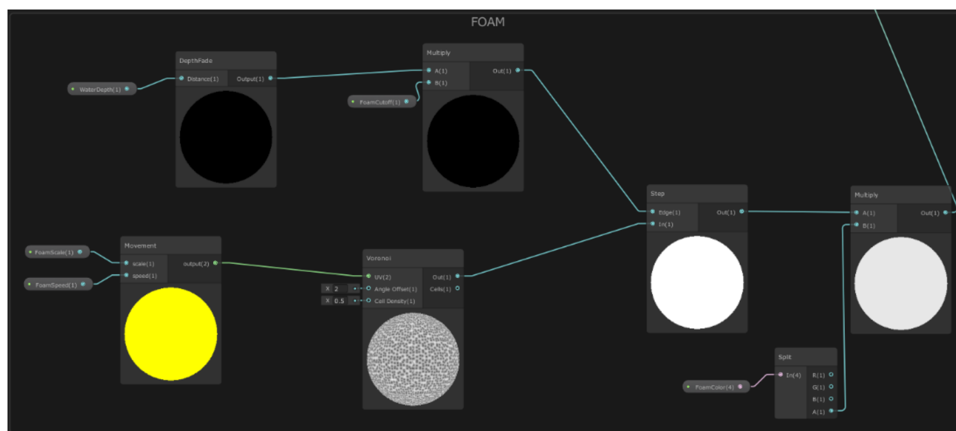

Figura 6.11.16 Nodos espuma agua

Para acabar se juntan los 4 módulos creados y el gráfico resultante lo podemos ver en la Figura 6.11.17.

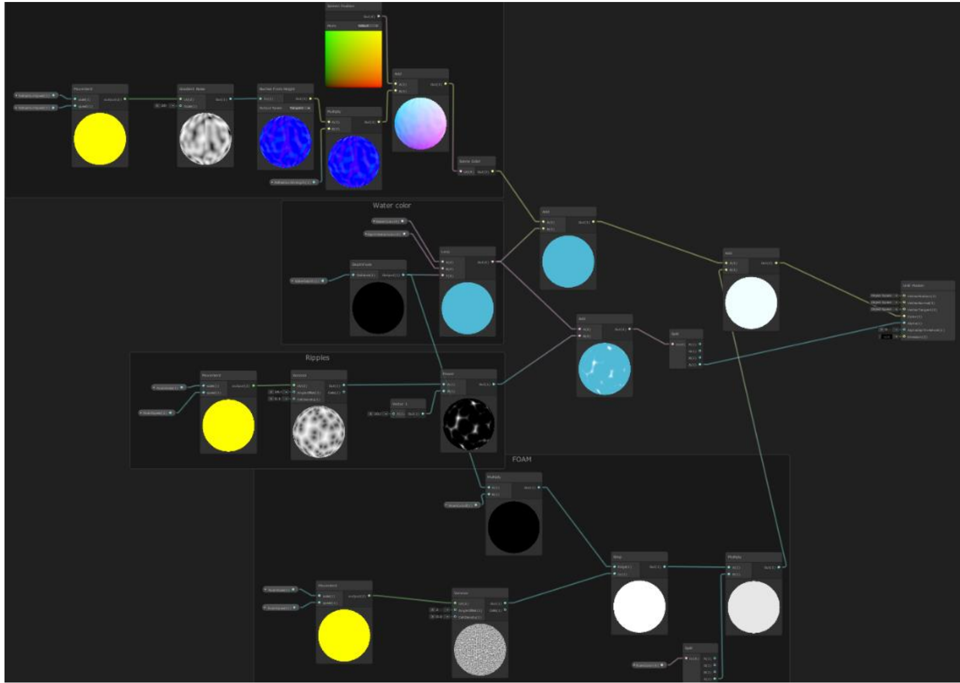

Figura 6.11.17 Diagrama de nodos completo

El resultado final, en el juego, lo podemos ver en la Figura 6.11.18 y la configuración utilizada en la Figura 6.11.19.

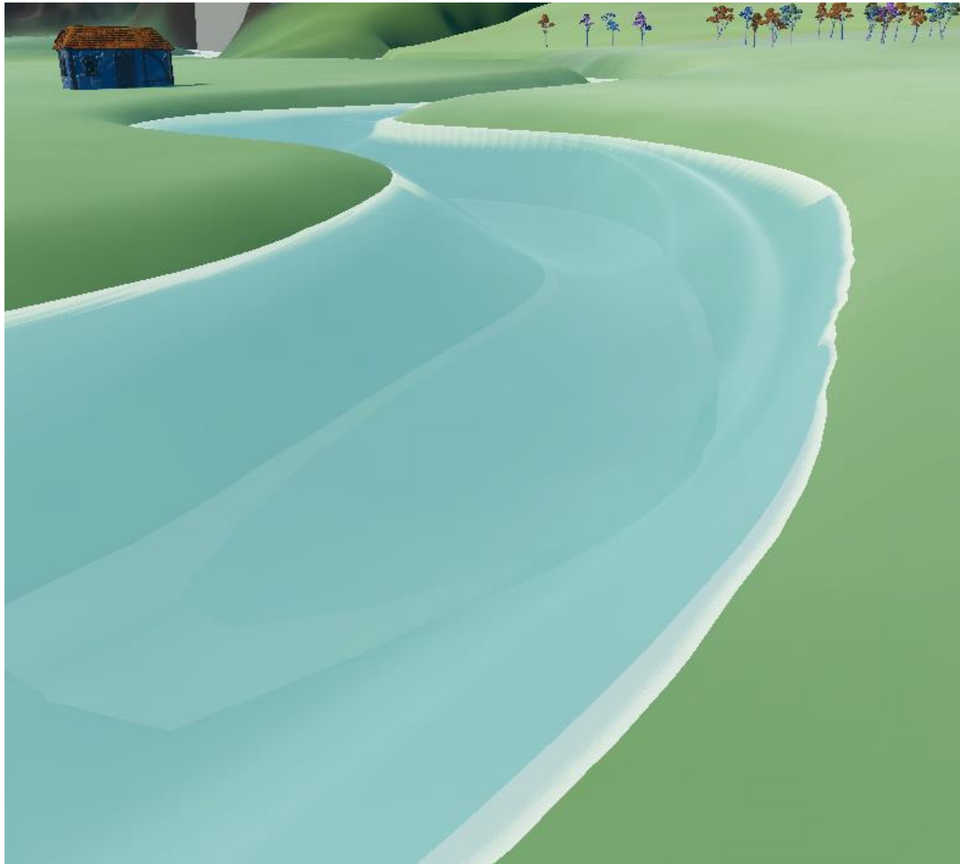

Figura 6.11.18 Río con el material de agua

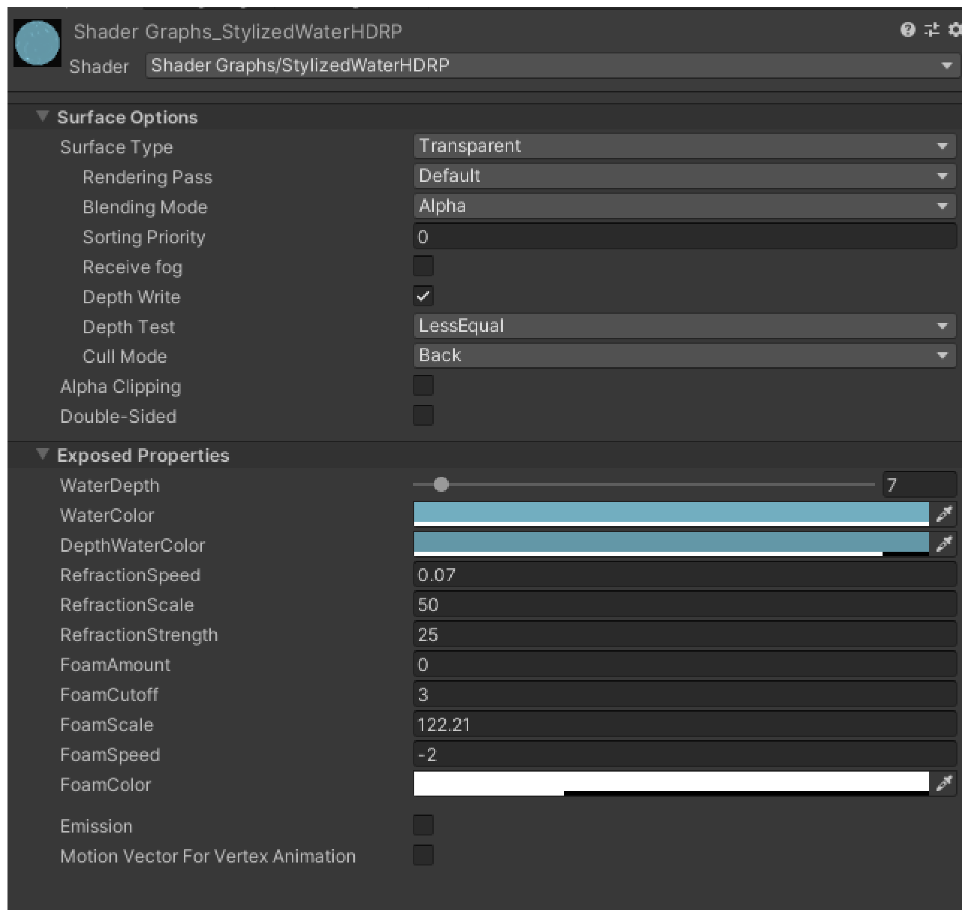

Figura 6.11.19 Configuración agua

6.12. Diálogo

A continuación, explicaremos cómo desarrollamos los sistemas de creación y utilización del diálogo dentro de Debt Trading.

6.12.1 Creador de ficheros de diálogo (*Dialogue Maker*)

Desde el principio del desarrollo entendimos que no podíamos crear los diálogos a mano; la cantidad que queríamos implementar y el hecho querer formatearlos en el formato json, para así facilitar su lectura dentro del programa, indicaban que crear un programa no solo sería más rápido, sino mucho más sencillo de manejar.

El resultado de esta mentalidad fue la creación del programa *Dialogue Maker*. De forma resumida, el usuario va introduciendo líneas de diálogo o respuestas dentro de las casillas. El programa lee la información de estas casillas, la transforma al formato json y la inserta dentro del fichero correspondiente, como está programado en la Figura 6.12.1.

```
//Inicializar valores
Conversation c = new Conversation();
c.version = 1;
c.dialogue = new List<DialogueLine>();
c.dialogue.Clear();
c.answers = new List<Answer>();
c.answers.Clear();
a.number = 1;
c.answers.Add(a);
FullConversation fc = new FullConversation();
fc.conversations = new List<Conversation>();
fc.conversations.Clear();
fc.conversations.Add(c);
//Creamos el fichero json y escribimos la primera conversación.
string json = JsonUtility.ToJson(fc);
File.WriteAllText(path, json);
```

Figura 6.12.1 Formatear dialogo a JSON

Si un usuario quiere crear una línea de diálogo, tendrá que insertar la siguiente información:

- La **emoción** (*emotion*) que transmite la línea de diálogo.
- El **personaje** (*speaker*) que está diciendo la línea de diálogo.
- La línea de **diálogo** (*dialogue*)

En cambio, si lo que quiere crear un usuario es una respuesta a una pregunta, el usuario simplemente tendrá que escribirla en la casilla de **respuesta** (*answer*). Las respuestas sólo tendrán lugar al final de una conversación.

Obligatoriamente, en los dos casos anteriores el usuario tendrá que especificar el **nombre del fichero** (*filename*) al cual está añadiendo la línea o pregunta. Si el fichero no existe, se creará automáticamente.

Por otra parte, los valores restantes son opcionales: la **versión** (*version*) nos indica en qué día está teniendo lugar el diálogo (versión 1 será el diálogo del día 1, por ejemplo), mientras que el **número** (*number*) nos indica en qué posición se encuentra la línea o pregunta escrita. Si estos valores se dejan vacíos, el programa automáticamente los insertará en los últimos valores existentes.

Como ejemplo, digamos que queremos empezar el diálogo de una misión llamada *Today is the Day*, en la que Rolan y Mena tendrán una pequeña conversación como la siguiente:

Mena: Hey, Rolan.

Rolan: What is it, kid?

Mena: You know what day it is today?

Rolan: No, I don't. What is today?

Y al finalizar esta pequeña conversación, el jugador puede añadir las siguientes respuestas a la pregunta de Rolan.

Today is the day you die.

Today is the day you live.

Comenzaremos con la primera línea de diálogo: pondremos el título del fichero, (*TodayIsTheDay*), el nombre de la persona que habla (*Mena*), la emoción que transmite esta línea (*Neutral,* en nuestro caso), y la línea de diálogo (*Hey, Rolan*). Después de darle al botón de **Sumbit**, se leerá esta información y se creará el fichero json *TodayIsTheDay*, donde en su primera versión tendrá como primera línea de diálogo lo que acabamos de escribir, como se muestran en las Figuras 6.12.2 y 6.12.3.

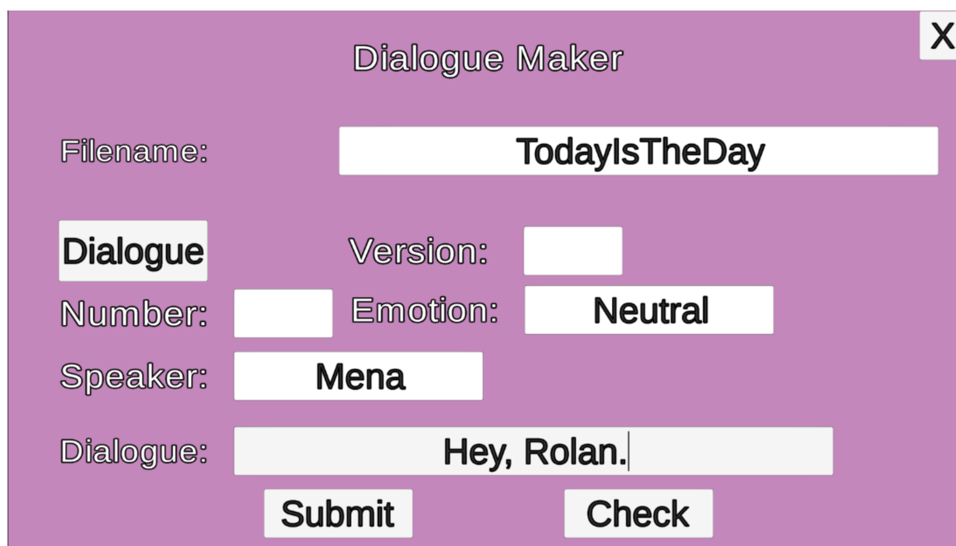

Figura 6.12.2 El programa DialogueMaker mostrando como escribir la primera línea de la misión Today is the Day

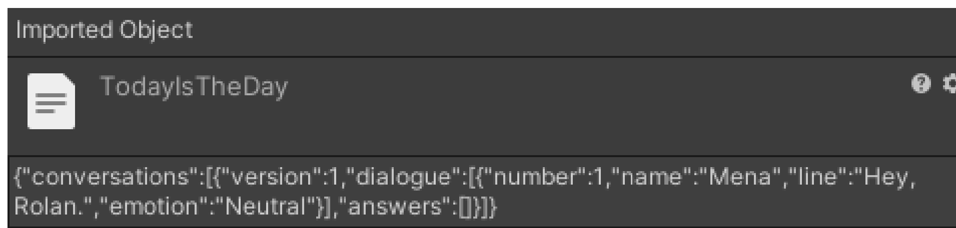

Figura 6.12.3 JSON de la misión Today is the Day después de escribir la primera frase.

De esta manera, iremos escribiendo hasta que tengamos las cuatro líneas de diálogo especificadas anteriormente, como se muestra en la Figura 6.12.4.

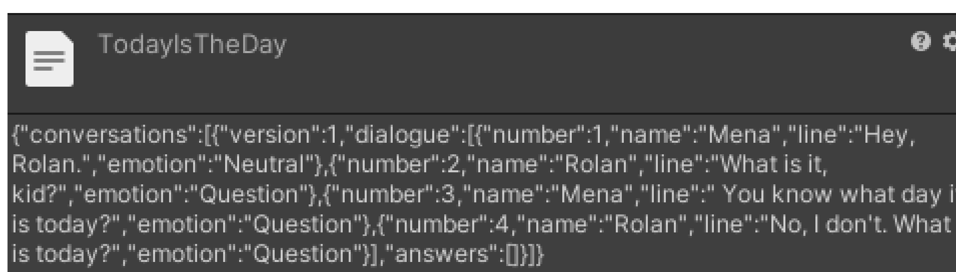

Figura 6.12.4 JSON de la misión Today is the Day con todo el diálogo.

Para añadir las preguntas, ahora le daremos al botón que se encuentra en la derecha, que ahora mismo indica que estamos escribiendo el diálogo. Al darle clic pasaremos a la interfaz de escribir respuestas, y aquí escribiremos las dos respuestas posibles, que se añadirán al fichero de diálogo, como se ve en la Figura 6.12.5.

TodayIsTheDay

{"conversations":[{"version":1,"dialogue":[{"number":1,"name":"Mena","line":"Hey, Rolan.","emotion":"Neutral"},{"number":2,"name":"Rolan","line":"What is it, kid?","emotion":"Question"},{"number":3,"name":"Mena","line":" You know what day it is today?","emotion":"Question"},{"number":4,"name":"Rolan","line":"No, I don't. What is today?","emotion":"Question"}],"answers":[{"number":1,"line":"Today is the day you die."},{"number":2,"line":"Today is the day you live."}]}]}

Figura 6.12.5 JSON de la misión Today is the Day con todas las respuestas.

Ahora, digamos que hemos planteado mejor el diálogo, y queremos cambiar la segunda línea de Mena, *You know what day it is today?,* por otra que diga *Do you happen to know what is today?*.

Como se puede observar en las figuras anteriores, cada línea y pregunta tiene un número asignado. De esta manera, si queremos cambiarla solo tendremos que especificar la línea que queremos cambiar, como se ve en la Figura 6.12.6, y el programa automáticamente la reemplazará por lo que hemos entrado, siendo el resultado visible en la Figura 6.12.7.

Dialogue Maker
X
Filename: TodayIsTheDay
Dialogue
Version:
Number: 3
Emotion: Question
Speaker: Mena
Dialogue: Do you happen to know what is today?
Submit
Check

Figura 6.12.6 El programa DialogueMaker mostrando como corregir la línea número 3 de la misión Today is the Day.

TodayIsTheDay

{"conversations":[{"version":1,"dialogue":[{"number":1,"name":"Mena","line":"Hey, Rolan.","emotion":"Neutral"},{"number":2,"name":"Rolan","line":"What is it, kid?","emotion":"Question"},{"number":3,"name":"Mena","line":"Do you happen to know what is today?","emotion":"Question"},{"number":4,"name":"Rolan","line":"No, I don't. What is today?","emotion":"Question"}],"answers":[{"number":1,"line":"Today is the day you die."},{"number":2,"line":"Today is the day you live."}]}]}

Figura 6.12.7 JSON de la misión Today is the Day con la frase alterada.

Como se puede observar, el programa que hemos desarrollado es muy competente para la tarea de la creación de diálogo. El único problema que encontramos fue que nos faltó tiempo para implementar una manera de ver los diálogos de manera más cómoda, ya que el formato json es un poco complicado de leer a simple vista.

Por este motivo, decidimos escribir primero los diálogos en Excel, para así poder corregirlos y revisarlos antes de pasarlos a json con DialogueMaker. La Figura 6.12.8 es un ejemplo de ello para el comienzo del diálogo de la misión *Mother's Favorites*.

Filename	MothersFavorites_1_1
Version	1
Speaker	Rolan
Emotion	*Tired*
Linia	Hey, kid? Sorry, I'm pretty busy right now. Leave me alone, please.
Speaker	Mena
Emotion	*Neutral*
Linia	But I've been told you can make my carriage faster...
Speaker	Rolan
Emotion	*Neutral*
Linia	You need help with a carriage, huh...
Speaker	Rolan
Emotion	*Neutral*
Linia	Okay, how about this: If you help me with something quick, I'll lend you a hand with that. Sounds good?
Respuesta 1	Sounds good. (1_2)
Respuesta 2	It doesn't sound good. (1_4)
Filename	MothersFavorites_1_2

Figura 6.12.8 Visual del fichero Excel de la misión Mother's Favorites.

6.12.2. Sistema de diálogo

En Debt Trading, el diálogo es una de las mecánicas principales y el sistema que utilizamos para transmitir la mayor parte de la historia, y es por eso que diseñamos un sistema robusto y completo, que funcionara de manera correcta.

Para que todo funcione correctamente, tenemos todo dividido en tres scripts principales:

DialogueClasses contiene todas las declaraciones de las clases que utilizamos para la utilización del diálogo:

- **DialogueLine** representa una línea de diálogo: tiene un *número*, el *nombre* de quien la dice, el texto de la línea y la emoción que esta representa.
- **Answer** define una respuesta: tiene como valores asignables un *número* y el *texto* de la respuesta, similar a DialogueLine.
- **Conversation** encapsula una versión del diálogo: está formado por el *número* de la versión, una *lista de DialogueLine*s y otra *lista de Answer*s.
- Finalmente, **FullConversation** representa todas las versiones de diálogo posibles, y es por ello que la clase solamente contiene una lista de *Conversation*s.

La Figura 6.12.9 muestra cómo funcionan estas clases de manera visual.

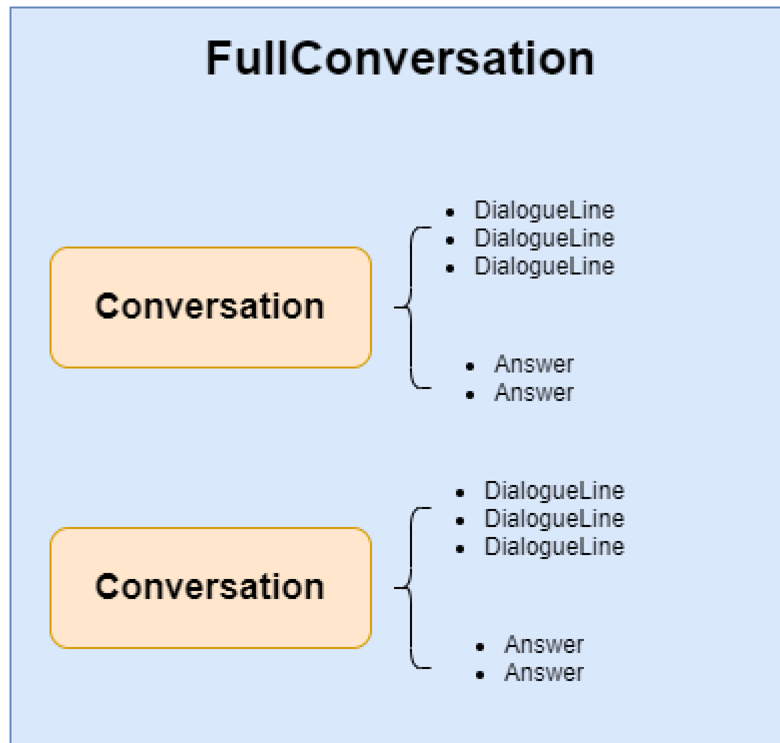

Figura 6.12.9 Diagrama visual de las clases incluídas en el script DialogueClasses.

DialogueChoiceButton es el script que utilizamos al crear los botones que utilizamos como respuestas en el diálogo.

DialogueWindow controla la ventana de diálogo, y es el script principal que utilizamos para mostrar el diálogo.

Así, la ventana de diálogo es un objeto que hemos definido en Unity, que tiene la estructura que se muestra en la Figura 6.12.10.

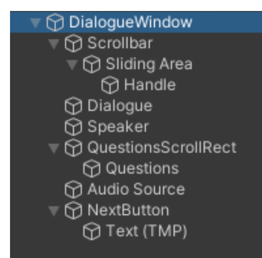

Figura 6.12.10 Estructura de un objeto DialogueWindow.

De manera simplificada, el funcionamiento de una conversación de diálogo en Debt Trading es el siguiente:

1. Cuando se inicia el programa, se inicializan los valores y se hace la ventana invisible, pero manteniéndola cargada.
2. Cuando se tiene que iniciar una conversación, *DialogueWindow* hace visible la ventana de diálogo y el botón de siguiente. Carga la primera línea de diálogo del fichero, la añade a la ventana, y escribe quién la está diciendo.
3. Cuando se carga una línea de diálogo, el programa lee la emoción de la línea y busca si hay un fichero de audio de esa emoción con el personaje que la dice. Si no se encuentra el fichero, no suena nada.
4. Cuando el jugador le da a siguiente, el proceso del paso 2 se repite hasta que ya no haya más líneas de diálogo.
5. Si existen respuestas, el programa lee el texto, lo aplica a un botón y este se añade a la ventana de diálogo.
6. Cuando el jugador elige una respuesta, el número de la respuesta se envía al nodo de actualizar la misión correspondiente.
7. Se eliminan todos los botones de respuesta, se reinician los valores y la ventana de diálogo vuelve a hacerse invisible.

Aunque las bases de esta implementación se han mantenido desde el principio del desarrollo, pasamos por muchas iteraciones hasta conseguir el resultado final, sobre todo en los detalles de la muestra de líneas de diálogo, la carga de sonidos y el sistema de respuestas.

Para empezar, las líneas de diálogo se mostraban completamente al principio, pero al poco decidimos hacer que se mostraran letra por letra. Esto trajo algunos problemas ya que, al darle al botón Next, saltaba la línea, aunque no estuviera completada. Esto lo arreglamos añadiendo una variable que nos señalaba si la línea se había completado; si no lo había hecho, la mostrábamos completamente. Esta implementación se puede observar en la Figura 6.12.11.

```
//Termina la coroutina de cargar la linia y la muestra completa.
private void FinishLine()
{
    StopCoroutine(linia);
    dialogueText.text = conv.dialogue[actualLine].line;
}

//Carga la siguiente línea
public void NextLine()
{
    if (dialogueText.text != conv.dialogue[actualLine].line) FinishLine();
    else
    {
        actualLine += 1;
        ShowDialogueLine();
    }
}
```

Figura 6.12.11 Visual del código que se utiliza al darle al botón Next.

A continuación, un elemento que implementamos hacia el final del desarrollo fue la carga de audios, ya que esperamos a tener los audios de los personajes creados por nosotros. Cuando se inicializa el juego, obtenemos el *path* en el que se encuentra la carpeta con todos los audios, como se muestra en la Figura 6.12.12.

```
voicesPath = Application.dataPath + "/Resources/Audio/DialogueAudio";
```

Figura 6.12.12 Línea de código que utilizamos para obtener el camino de las carpetas de audio.

Al cargar cada línea, generamos un número al azar dentro de un rango, según la persona que esté hablando, y buscamos un fichero de audio con esa emoción que termine en ese número (AzarielAngry2, por ejemplo). Si ese fichero no existe, buscamos el fichero con el número siendo uno; si no, indicamos que no existe el audio. El código que hace todo esto posible es el de la Figura 6.12.13.

```
//Detecta si el audio existe
private bool CheckAudioFile(string name, string emotion)
{
    int maxNumber;
    if (name is "Mena") maxNumber = 4;
    else if (name is "Guard") maxNumber = 2;
    else maxNumber = 3;
    bool exists = true;
    if (Directory.Exists(voicesPath + "/" + name))
    {
        int number = Random.Range(1, maxNumber);
        if (Resources.Load("Audio/DialogueAudio/" + name + "/" + name + emotion + number) as AudioClip != null)
            audioS.clip = Resources.Load("Audio/DialogueAudio/" + name + "/" + name + emotion + number) as AudioClip;
        else if (Resources.Load("Audio/DialogueAudio/" + name + "/" + name + emotion + 1) as AudioClip != null)
            audioS.clip = Resources.Load("Audio/DialogueAudio/" + name + "/" + name + emotion + 1) as AudioClip;
        else exists = false;
    }
    return exists;
}
```

Figura 6.12.13 Código de la función CheckAudioFile.

Finalmente, el sistema de generar las respuestas fue uno de los aspectos que pasó por más versiones, ya que tuvimos muchas dudas sobre cómo hacerlo. Al principio, simplemente era texto que podíamos seleccionar, pero eventualmente nos decidimos por hacerlos botones, que era algo mucho más sencillo.

Cada botón tiene como componente el script **DialogueChoiceButton**, que guarda el número de la respuesta y el texto. Al cargar las respuestas, como se muestra en el código de la Figura 6.12.14, asignamos la primera con sus valores y cada consecuente botón tendrá un valor PosY más bajo, para que no se superpongan.

```
//Carga, si existen, las respuestas disponibles que puede dar el jugador.
private void LoadAnswers()
{
    bool firstQuestion = true;
    int posY = 0;
    answersBackground.enabled = true;
    answersBackground.GetComponent<ScrollRect>().verticalScrollbar = answerScrollbar;
    answerScrollbar.gameObject.SetActive(true);
    lChoices = new List<DialogueChoiceButton>();
    if (conv.answers.Count > 0)
    {
        for (int i = 0; i < conv.answers.Count; i++)
        {
            DialogueChoiceButton choice = Instantiate(prefab);
            lChoices.Add(choice);
            choice.transform.SetParent(GameObject.Find("Questions").transform, false);
            Debug.Log("Numero: " + conv.answers[i].number + ", linea: " + conv.answers[i].line + ", posY: " + posY);
            if (firstQuestion) firstQuestion = false;
            else
            {
                var pos = choice.transform.position;
                pos.y += posY;
                choice.transform.position = pos;
            }
            choice.AssignValues(conv.answers[i].number, conv.answers[i].line);
            posY -= 40;
        }
    }
    else
    {
        HideDialogueBox();
        linia = null;
        end = true;
    }
}
```

Figura 6.12.14 Código que muestra la función LoadAnswers.

Un detalle interesante es que en el json el primer valor es el 1, cuando en la programación en C# (el lenguaje que usamos) el primer valor de cualquier orden numérico es el 0; por este motivo, al enviar al nodo la respuesta elegida, le restamos 1.

6.13 Generar ciudades

En este apartado hablaremos de la implementación del generador de ciudades medievales procedurales.

6.13.1 Los planos

Uno de los puntos que nos daba más miedo era crear la estructura de la ciudad, el cómo y dónde deben estar puestas las casas, las calles, dónde está la muralla... entre otros elementos. Por ello mientras estudiábamos cómo crear estos planos, encontramos una aplicación, *Medieval Fantasy City Generator* en itch.io[7], hecha por Oleg Dolya, que encajaba a la perfección con nuestras necesidades. Este, es un generador de planos de ciudades medievales de fantasía en el cual hay diferentes opciones de configuración, como se puede ver en la Figura 6.13.1. Pero lo más importante para nosotros, es que se puede exportar en formato JSON. Teniendo el plano de la Figura en formato JSON, ya podemos empezar a montar pasar la información a Unity.

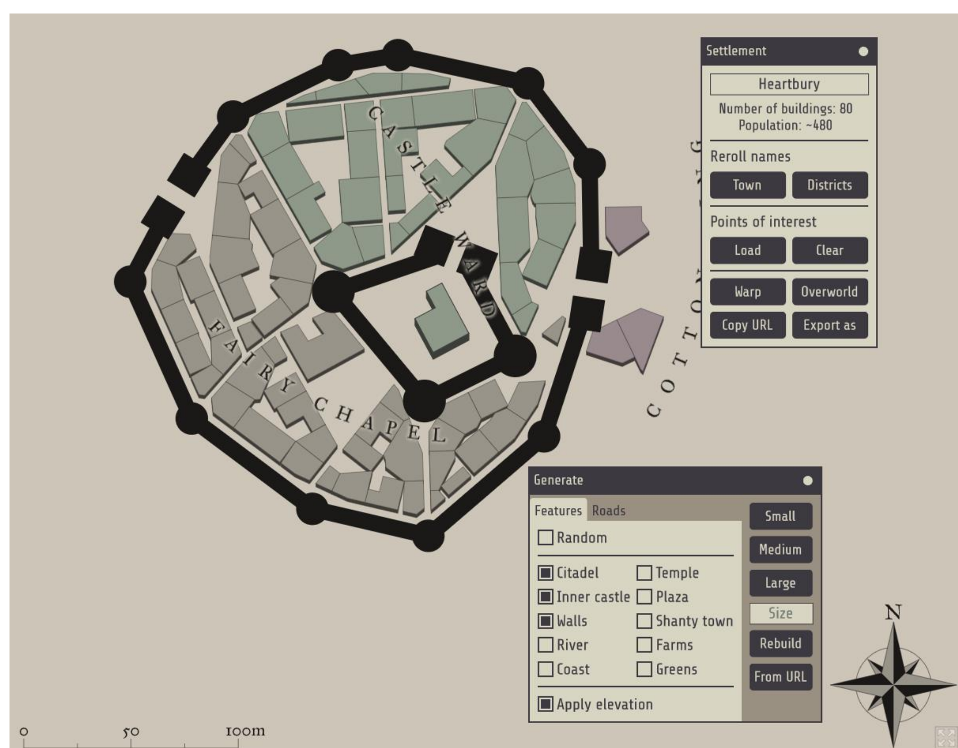

Figura 6.13.1 Medieval Fantasy City Generator

6.13.2 Interpretar el JSON en Unity

Ahora ya tenemos todos los planos de ciudades que queramos, la estructura de estos seria como la de la Figura 6.13.2.

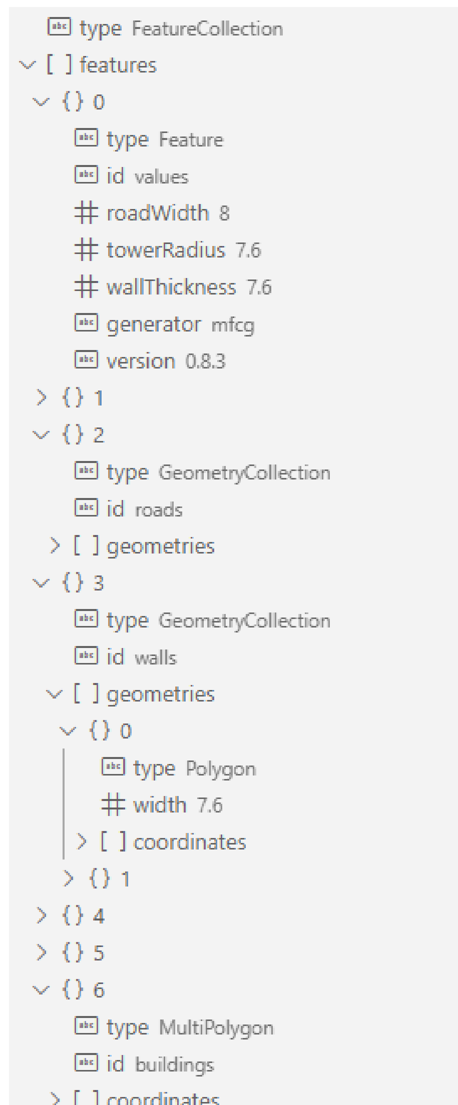

Figura 6.13.2 Estructura JSON Medieval Fantasy City Generator

El problema está en que las utilidades de Unity, en cuanto a lectura de ficheros JSON, son un poco limitadas. Por eso hemos cogido la librería SimpleJSON, de la wiki de Unity3D[8], con la que podemos leer el JSON como nodos dinámicos, los cuales podemos ir recorriendo hasta llegar a la información que necesitamos, las coordenadas en 2D de los diferentes elementos a construir.

```
JSONNode jn = GetJson();

foreach (JSONNode n in jn["features"])
{
    switch (n["id"].Value)
    {
        case "buildings":
            JSONNode coords = n["coordinates"];

            foreach (JSONArray blocks in coords.AsArray)
                foreach (JSONArray builds in blocks.AsArray)
                    foreach (JSONArray vert in builds.AsArray)
                        // Otener vertices casas

            break;

        case "walls":
            JSONNode walls = n["geometries"];

            foreach (JSONNode test in walls.AsArray) {
                JSONNode coordsB = test["coordinates"];
                float width = test["width"];

                foreach (JSONArray builds in coordsB.AsArray)
                    foreach (JSONArray vert in builds.AsArray)
                        // Obtener vertices muralla
            }
            break;

        case "roads":
            JSONNode roads = n["geometries"];

            foreach (JSONNode r in roads.AsArray) {
                JSONNode coordsR = r["coordinates"];

                foreach (JSONArray roadxy in coordsR.AsArray)
                    // Obtener vertices carretera

            }
            break;
    }
}
```

Figura 6.13.3 Obtener información del JSON

Para pasarlos a coordenadas 3D, lo que hacemos es traducir el vector de 2 puntos (x, y) al vector de 3 puntos (x, 0, y), dándole a la *z* el valor *y* mientras que a *y* le damos valor 0. Más adelante explicaremos qué hacemos con el valor *y*. Con esto ya podemos obtener la base de todas las casas y la muralla de la ciudad, en caso de que tenga.

Lo primero que hicimos para comprobar que el mapa se estaba leyendo bien, es recorrer todas las casas que hemos obtenido del JSON y, para cada una, crear un plano usando sus vértices. Le hemos puesto un color aleatorio a cada plano para ver que se separaban bien una casa de la otra. Ver Figura 6.13.4.

Figura 6.13.4 Plano traducido y puesto en Unity

6.13.3 Las paredes

Ahora que ya tenemos los “cimientos” de la casa, tenemos que levantar las paredes. Para ello hemos creado una lista de bloques predefinidos qué pueden actuar como pared, los cuales hemos configurado para saber que otros bloques pueden estar a su lado y con qué probabilidad. En la Figura 6.13.5, podemos ver todos los bloques que hemos utilizado.

Figura 6.13.5 Bloques de pared

Para hacer esto, lo que hemos hecho es crear dos Scriptable Objects, uno para definir las propiedades de cada bloque referente a las paredes y el otro para definir tipos de casas, qué archivos de configuración de paredes utilizara. En caso de la configuración de las paredes se puede definir cuál es el objeto prefabricado (los de la Figura 6.13.5), que tipo de bloque es (pared o puerta), si sólo puede aparecer en un piso especifico (0 para indicar que no hay limitación), y cuáles son sus posibles vecinos y cuál es la probabilidad de ser escogido. En la Figura 6.13.6 podemos ver la configuración de uno de los bloques de pared.

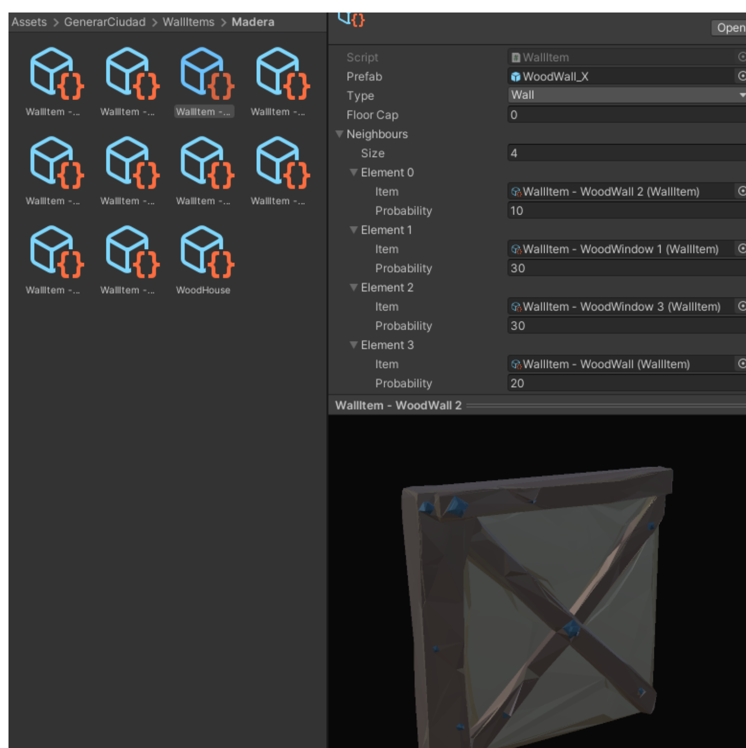

Figura 6.13.6 Configuración bloques pared

En este caso, para facilitar la configuración de todos los bloques, añadimos una ventana en el editor para ver el objeto que estábamos configurando en la parte inferior. Por otro lado, tenemos la configuración de las casas, que no es más que un listado de todas las paredes definidas anteriormente que pueden formar parte de la casa.

Ahora que ya están definidos estos objetos, sólo tenemos que hacer que el generador coloque estas paredes en los lugares adecuados, formando así las casas. Lo primero que hacemos es calcular aleatoriamente, entre dos valores definidos, cuántos pisos tendrá la casa. Para darle un poco más de variación también calculamos una pequeña variación de escalado vertical para las paredes para que dos casas contiguas de los mismos bloques no se vean de la misma altura. Una vez tenemos calculada la verticalidad, pasamos a construir las paredes. Para cada pareja de vértices consecutivos de una casa calculamos la distancia que hay y con ello calculamos cuántos bloques de pared pondremos. Para que no sobresalga se calculará una escala horizontal, repartida entre todos los bloques de ésta, que nos permita ajustárnos mejor a los vértices.

```
int nBlocks = (int)Mathf.Round(wall.distance / blockSize);
float sizeBloques = nBlocks * blockSize;
float scale = wall.distance / sizeBloques;
```

Figura 6.13.7 Escala ponderada para todos los bloques

Teniendo esos valores claros, sólo nos queda colocar todos los bloques de cada uno de los pisos de la pared, escalarlo para cumplir con los cálculos previos para que no sobresalga, y darle la variación de altura y hacer que sea hijo del objeto casa, para tenerlo todo ordenado y poder aplicar transformaciones por jerarquía.

```
for (int i = 0; i < wall.floors; i++)
{
    Vector3 position = wall.start + wall.direction * blockSize * scale + Vector3.up * blockSize * wall.verticalScale * i;

    WallItem item = null;
    for (int j = 0; j < nBlocks; j++){
        if (item == null){
            bool continueIterating = true;
            while (continueIterating){
                item = config.wallItems[Random.Range(0, config.wallItems.Count)];

                if (item.floorCap > 0 && i + 1 != item.floorCap) // si no compleix restriccio de pis, torna a tirar
                    continue;

                continueIterating = false;
            }
        }
        else
            item = item.SelectNeighbour(i + 1).item;

        GameObject go = GameObject.Instantiate(item.prefab, position, Quaternion.LookRotation(wall.direction, Vector3.up));
        go.name = "Floor:" + i + "_Block:" + j;

        Vector3 vScale = Vector3.one;
        vScale.x *= scale;
        vScale.y *= wall.verticalScale;

        go.transform.localScale = Vector3.Scale(go.transform.localScale, vScale);
        go.transform.localRotation *= rotationOffset;
        go.transform.parent = wallGO.transform;

        position += wall.direction * blockSize * scale;
    }
}
```

Figura 6.13.8 Construir paredes

En el punto actual el resultado de generar las casas sería el que podemos ver en la Figura 6.13.9.

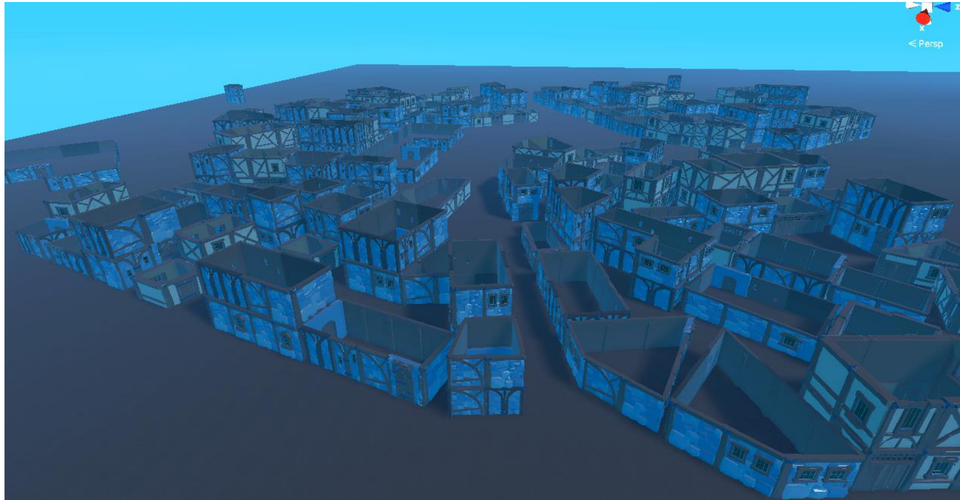

Figura 6.13.9 Paredes construidas

Cabe destacar que hacemos una serie de transformaciones a los vértices antes de construir las paredes. Éstas son: desplazar los vértices hacia el centro para que no colapsen las casas y darles variación, eliminar las paredes pequeñas, convertir en una las paredes que siguen una misma línea en una única pared, eliminar casas de 3 paredes y eliminar casas extremadamente pequeñas. También, después de construir la pared, añadimos un box collider que cubra toda la pared para que el jugador no las pueda atravesar.

6.13.4 Los tejados

Los tejados ha sido la parte más difícil de desarrollar. Hemos desarrollado 3 tipos de tejados distintos, pero dos de ellos tienen la limitación de que sólo se pueden construir en casas de 4 paredes.

Tejado triangular

El primero de estos es un tejado con los laterales triangulares, como el de la Figura 6.13.10.

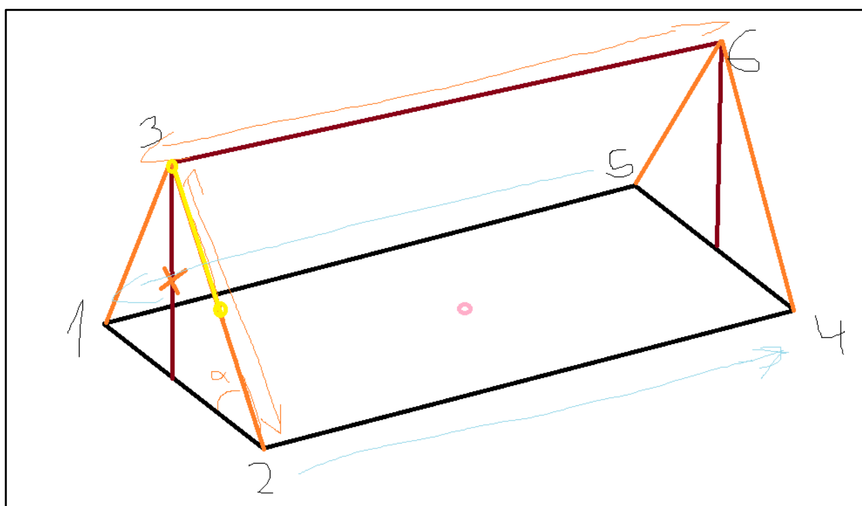

Figura 6.13.10 Esquema construcción tejado triangular

Para hacer este tejado, que sólo se puede hacer en casas de 4 paredes, lo primero que hacemos es buscar la pared más pequeña. Esa pared y la paralela serán la que tendrán el triángulo, mientras que

las otras dos tendrán las tejas. Seguidamente calcularemos un valor de altura aleatorio para darle variación a este tipo de tejado. Con esa altura ya tendremos los 6 vértices necesarios para construir el tejado, indicados en la Figura 6.13.10. Con esos vértices construiremos los triángulos, el primero con el orden [1, 2, 3] y el segundo con el orden [6, 4, 5]. El orden debe ser éste, para que la normal del plano generado apunte para afuera y no se vea afectado por el *backface culling.* Después creamos una viga que va del vértice 3 al 6 y seguidamente llenaremos los dos laterales restantes de tejas, más adelante en este mismo apartado, explicaremos como hemos colocado las tejas.

Tejado inclinado

El siguiente tejado, para casas de 4 paredes, es el que hemos nombrado tejado inclinado. Para construir este tejado empezaremos como el anterior, buscando la pared más pequeña y la paralela. Para cada una de éstas calcularemos aleatoriamente una altura. En la Figura 6.13.11 podemos ver como quedan los vértices de la estructura del tejado.

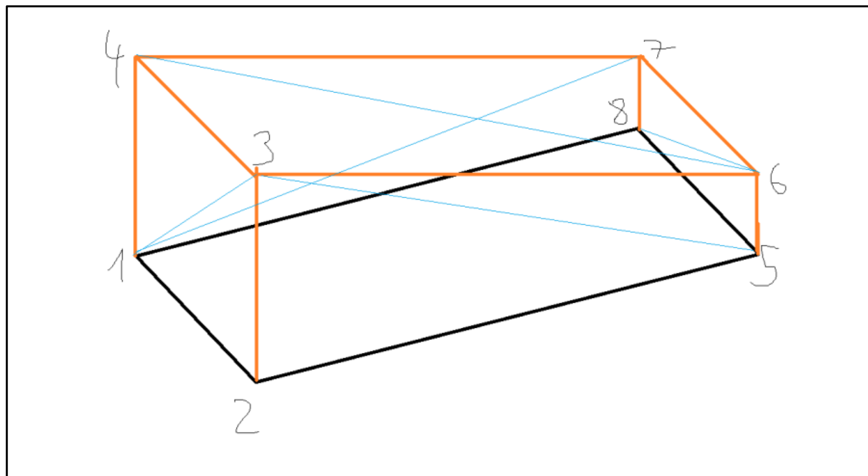

Figura 6.13.11 Esquema construcción tejado inclinado

Con esto construiremos paredes en todas las caras con los siguientes triángulos:

- [1, 2, 3] - [1, 3, 4]
- [3, 2, 5] - [5, 6, 3]
- [7, 6, 8] - [6, 5, 8]
- [7, 8, 1] - [1, 4, 7]

En este caso también tendremos vigas, pero serán 4 vigas verticales que unirán los 4 vértices inferiores con los 4 vértices superiores. La cara superior, vértices [3,4,6,7], quedará cubierta de tejas.

Tejado biselado

El tercer y último tejado que hemos desarrollado es el tejado biselado. Este tejado es apto para todas las casas, independientemente del número de paredes que tengan. La estructura es la de la Figura 6.13.12.

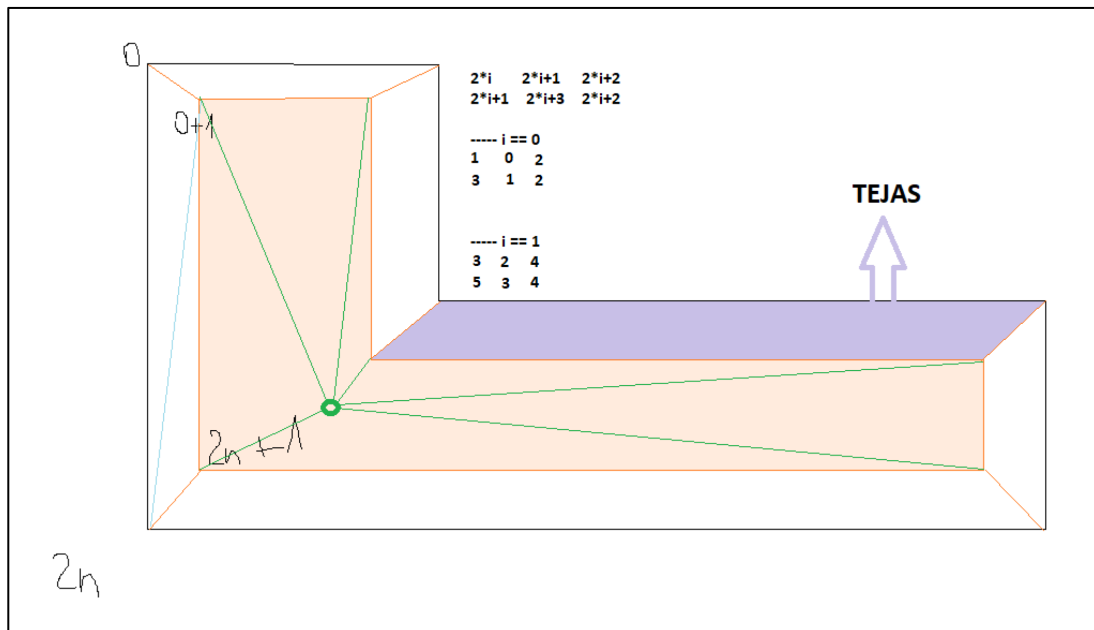

Figura 6.13.12 Esquema construcción tejado biselado

Este tejado consiste en un biselado a una altura aleatoria en donde la capa superior es un plano liso y todas las caras laterales están rellenadas con tejas. Para calcular los vértices interiores, lo que hacemos es recorrer todos los vértices base, y por cada vértice hemos calculado la dirección de la bisectriz respecto el vértice anterior y el siguiente. Después hemos cogido un punto ligeramente desplazado en esa dirección y hemos comprobado si es un punto interior o exterior al polígono. En caso de ser un punto exterior, invertiremos la dirección del desplazamiento.

```
Vector3 v1 = (ant - current).normalized;
Vector3 v2 = (next - current).normalized;

float angle = Vector3.SignedAngle(v1, v2, Vector3.up) / 2;
Vector3 vCenter = Quaternion.AngleAxis(angle, Vector3.up) * v1;

Vector3 point = current + vCenter * 3;

if (!GFG.isInside(polygon, polygon.Length, new Point(point.x, point.z)))
{
    vCenter = -vCenter;
}
```

Figura 6.13.13 Calcular los vértices del biselado

Esta comprobación es necesaria ya que, como podemos ver en la Figura 6.13.14, en algunos casos nos encontrábamos con que la bisectriz nos daba puntos exteriores al polígono y se generaban tejados con formas irregulares.

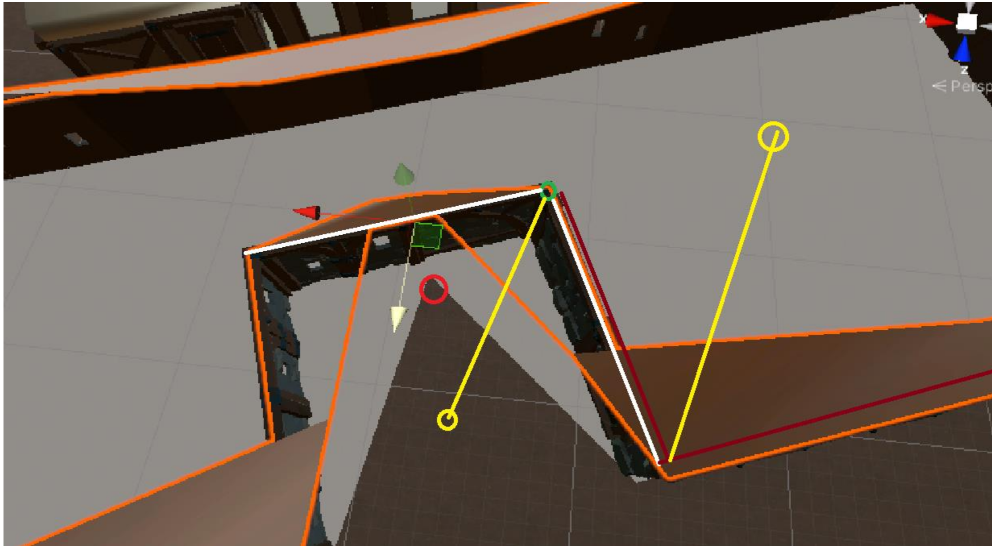

Figura 6.13.14 Error calculando las bisectrices

Una vez tenemos todos los vértices del bisel calculados, podemos empezar a crear la estructura. Para ello creamos una mesh de todos los vértices el bisel. En este caso, en vez de hacer los triángulos, nosotros hemos utilizado el triangulador de la librería Triangulator de la wiki de Unity3D[9], ya qué realiza cálculos extra para triangular en casos que el "centro" se encuentre fuera del polígono. En la Figura 6.13.14 se puede observar cómo se generaría utilizando el centro lógico. Después, crearemos unas vigas que unan los vértices base con sus respectivos vértices del bisel y finalmente rellenaremos los laterales con tejas.

Construir tejas

Las tejas las hemos creado en Blender y hemos construido 3 tejas distintas. De cada una hemos hecho dos modelos distintos, uno con la parte trasera tapada y otra sin tapar. Para ahorrar cálculos a la GPU, en la Figura 6.13.15 podemos ver los modelos de tejas que hemos utilizado.

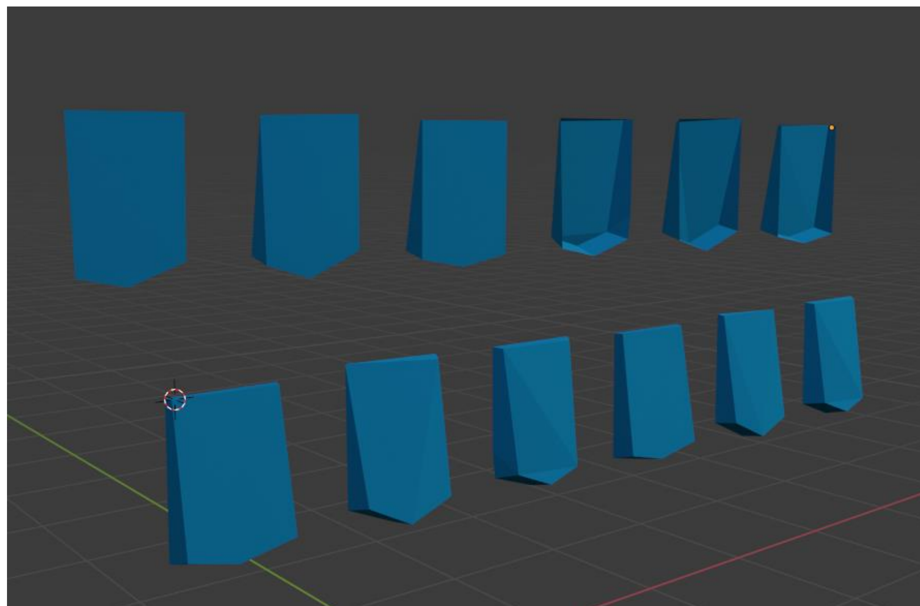

Figura 6.13.15 Modelos tejas

Para crear las tejas hemos creado una función que recibe los 4 vértices por parámetro y calcula, de una manera similar a las paredes, la cantidad de tejas que tiene que colocar horizontal y verticalmente, con una escala repartida entre todas las tejas para que encajen bien y no sobresalgan por los laterales del tejado, con la diferencia de que en este caso tenemos que calcular también el ángulo del tejado para poder colocar las tejas con la inclinación correcta.

```
Vector3 leftDir = (v1 - v0).normalized;
float leftDist = Vector3.Distance(v1, v0);

Vector3 rightDir = (v3 - v2).normalized;
float rightDist = Vector3.Distance(v3, v2);

// vertical
int nBlocksV = (int)Mathf.Round((leftDist / tileSize.y + rightDist / tileSize.y) / 2);

float lBlocksSizeV = nBlocksV * tileSize.y;
float lScaleV = leftDist / lBlocksSizeV;
float lAngle = Vector3.Angle(v1 - v0, Vector3.up);
```

Figura 6.13.16 Calcular ángulo y escala tejas

El problema de utilizar estos valores es que las paredes no son paralelas ni tienen las mismas longitudes, y nos podemos encontrar con problemas como el de la Figura 6.13.17.

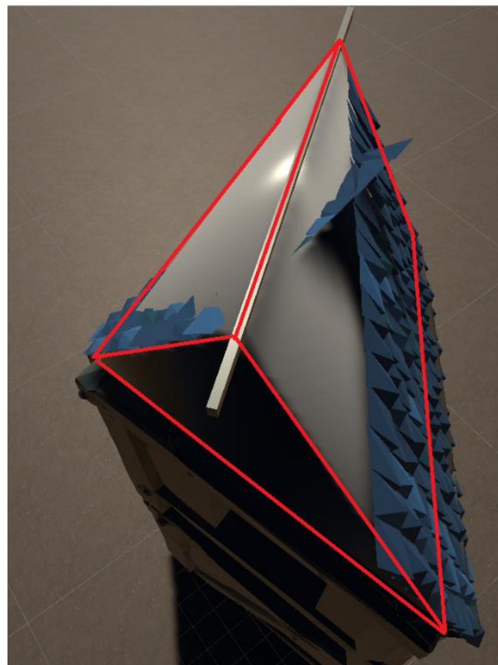

Figura 6.13.17 Error colocando las tejas

Para solucionar esto, lo que hacemos es coger el ángulo y escala vertical que deberían tener ambos laterales del tejado y con ello, al generar las tejas, podemos interpolar la escala y la rotación según estén más cerca del lateral izquierdo o del derecho.

```
int nBlocksV = (int)Mathf.Round((leftDist / tileSize.y + rightDist / tileSize.y) / 2);

float lBlocksSizeV = nBlocksV * tileSize.y;
float lScaleV = leftDist / lBlocksSizeV;
float lAngle = Vector3.Angle(v1 - v0, Vector3.up);

float rBlocksSizeV = nBlocksV * tileSize.y;
float rScaleV = rightDist / rBlocksSizeV;
float rAngle = Vector3.Angle(v3 - v2, Vector3.up);
for (int j = 0; j < nBlocks; j++) // horizontal
{
    if (i <= 1 || j == 0 || j == nBlocks - 1)
        go = GameObject.Instantiate(tiles[Random.Range(0, tiles.Count)].tileCovered, position, Quaternion.LookRotation(direction, Vector3.up));
    else
        go = GameObject.Instantiate(tiles[Random.Range(0, tiles.Count)].tile, position, Quaternion.LookRotation(direction, Vector3.up));
    go.name = "Roof Block: " + i + "-" + j;

    vScale = Vector3.one;
    vScale.x = scale;
    vScale.y = Mathf.Lerp(lScaleV, rScaleV, (float)j / nBlocks);
    go.transform.localScale = Vector3.Scale(go.transform.localScale, vScale);

    rot = go.transform.localRotation.eulerAngles;
    rot.x += Random.Range(1.5f, -1.5f);
    rot.y += Random.Range(1.5f, -1.5f);
    rot.z += Random.Range(3f, -3f);
    rot.z -= Mathf.Lerp(lAngle, rAngle, (float)j / nBlocks);

    go.transform.localRotation = Quaternion.Euler(rot);

    mat = tilesMaterials[Random.Range(0, tilesMaterials.Count)];
    foreach (Renderer r in go.GetComponentsInChildren<Renderer>())
    {
        r.material = mat;
    }

    go.transform.parent = roofgo.transform;

    // GameObjectUtility.SetStaticEditorFlags(go, staticEditorFlags);

    position += direction * tileSize.x * scale;

    tilesCreated++;
}
```

Figura 6.13.18 Angulo y escala tejas interpolado

En la Figura 6.13.19 podemos ver el resultado de colocar las tejas aplicando la interpolación para que se adapte mejor a las diferentes formas que pueden tener los tejados.

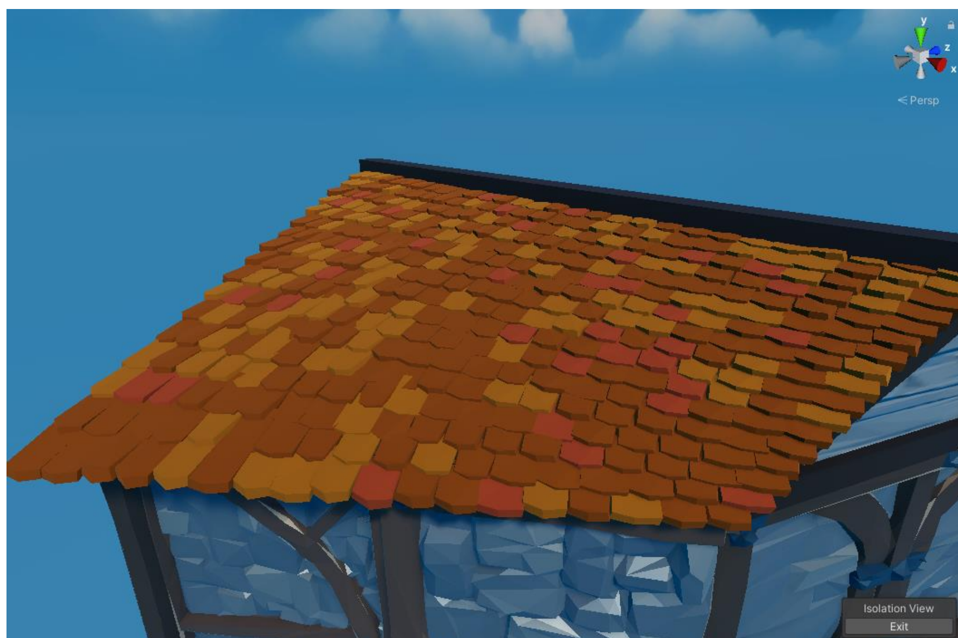

Figura 6.13.19 Tejas con la interpolación funcionando

Como hemos comentado antes, tenemos tejas que tienen la parte trasera tapada y otras que no. Las que tienen la parte trasera tapada las usaremos en caso de que estemos colocando las tejas que pertenecen a la primera o última fila o las que pertenecen a la primera o última columna. Esto lo hacemos para que, si sobresalen y el jugador lo ve desde abajo, lo vea todo correctamente y no se vea afectado por el *backface culling,* ver Figura 6.13.20.

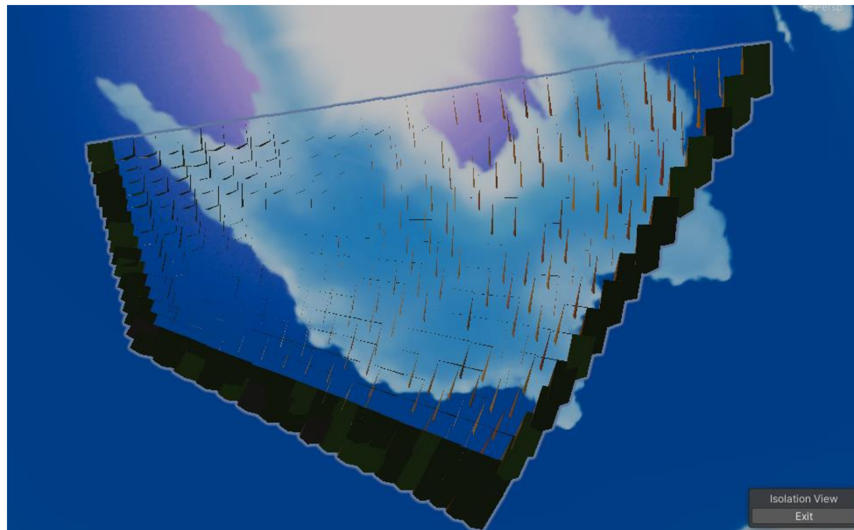

Figura 6.13.20 Backface culling en las tejas

Hay que tener en cuenta que, si una ciudad tiene 179 edificios y cada edificio puede tener una media de 2000 tejas, para una única ciudad tenemos 358.000 instancias de tejas, que son llamadas que el sistema está haciendo a la tubería de renderizado. Para reducir este número de llamadas hemos utilizado el *MeshCombiner* que hemos explicado en el Apartado 6.4.1.

Finalmente cabe destacar que, con la intención de añadir más variación y darle un toque más estético, cada tejado está compuesto de 3 colores que se colocan en cada teja de manera aleatoria. En la Figura 6.13.21 podemos ver el resultado final de la generación de los tejados con los 3 tipos distintos.

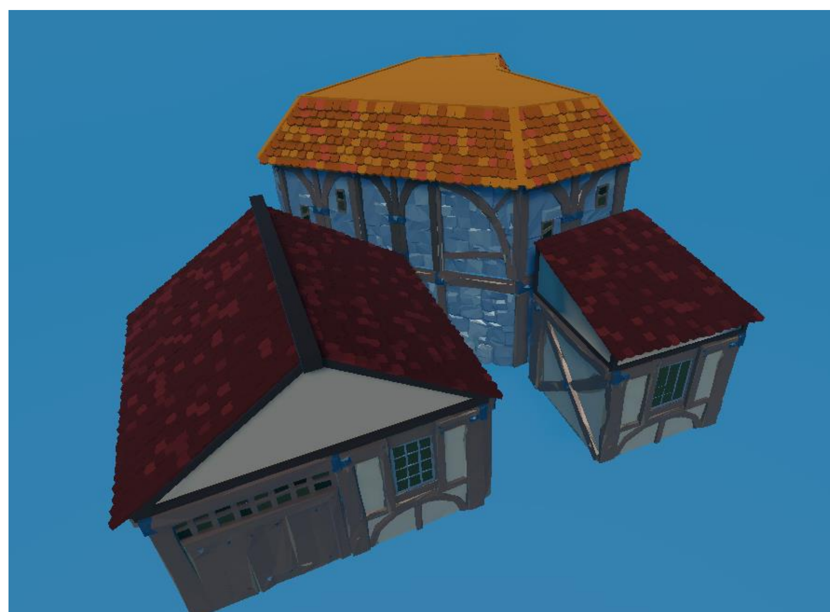

Figura 6.13.21 Distintos tipos de tejados desarrollados

6.13.5 Murallas

La muralla ha sido la parte más sencilla de implementar, ya que ha consistido únicamente en instanciar los bloques que hemos creado en Blender con la rotación adecuada para que vaya de un vértice al siguiente. En los vértices que conforman las uniones de las paredes de la muralla hemos instanciado torreones. Ver Figura 6.13.22.

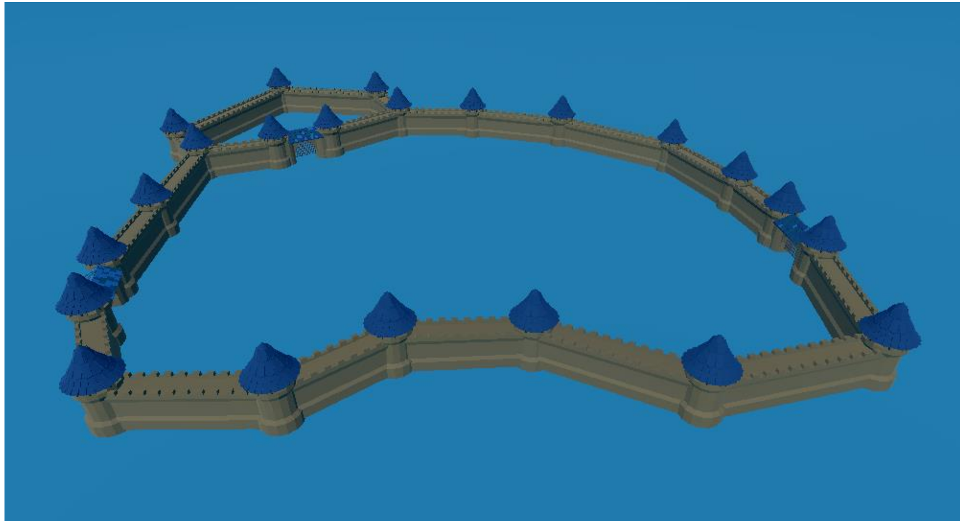

Figura 6.13.22 Muralla de la ciudad

Colocar las puertas de las murallas ya ha sido más complicado, ya que el JSON no nos da esa información, sino que hemos tenido que buscar la intersección de las carreteras con la muralla y crear vértices alrededor para hacer hueco en la muralla y poder instanciar la puerta en el lugar adecuado.

```
for (int i = 0; i < wallVertList.Count; i++)
{
    for (int j = 0; j < wallVertList[i].Count; j++)
    {

        if (roadVertList.Any(p => p == wallVertList[i][j]))
        {

            int iPrev = j > 0 ? j - 1 : wallVertList[i].Count - 1;
            int iNext = j < wallVertList[i].Count - 1 ? j + 1 : 0;

            Vector3 prevDir = (wallVertList[i][iPrev] - wallVertList[i][j]).normalized;
            Vector3 nextDir = (wallVertList[i][iNext] - wallVertList[i][j]).normalized;

            wallVertList[i].Insert(j, wallVertList[i][j]);

            wallVertList[i][j] += prevDir * 18;
            wallVertList[i][j + 1] += nextDir * 18;

            TownWall obj = new TownWall();
            obj.start = wallVertList[i][j];
            obj.end = wallVertList[i][j + 1];
            wallDoorList.Add(obj);
        }
    }
}
```

Figura 6.13.23 Crear espacio para las puertas

En la Figura 6.13.24, se puede observar cómo, utilizando otra configuración de modelos para la muralla, podemos obtener murallas variadas para diferentes tipos de ciudades.

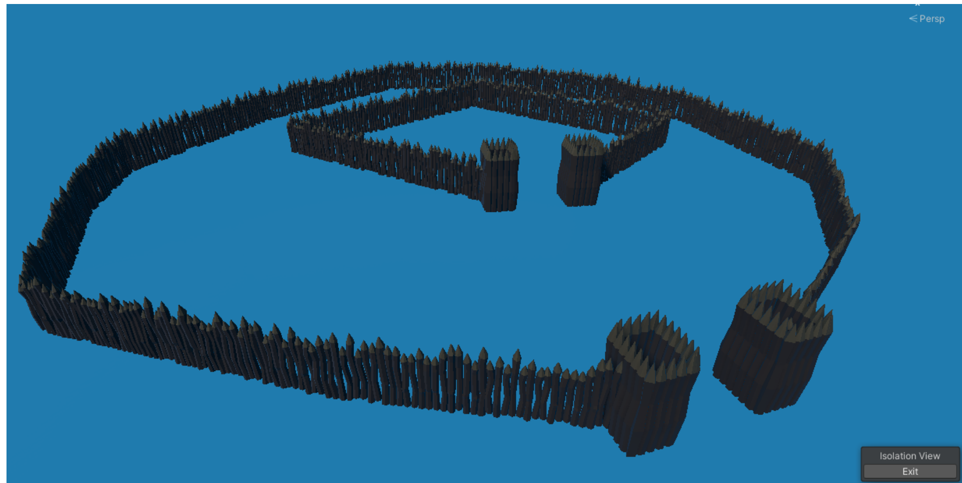

Figura 6.13.24 Diseño alternativo muralla

6.13.6 Adaptar al terreno

Debido a que las ciudades nunca van a estar en un terreno completamente liso, tuvimos la necesidad de hacer que las construcciones se adaptaran al terreno. Para ello cada casa, por cada uno de sus vértices base, lanzara un rayo vertical buscando colisionar con algún objeto de la capa Ground. Al colisionar guardaremos el rayo que ha encontrado la posición más baja y se moverá la casa a esa altura para que quede bien integrada con el terreno. Con las murallas haremos lo mismo, pero a diferencia de las casas, con éstas lo haremos antes de construirlas, así las murallas en vez de generarse planas se generarán con inclinación buscando una integración mayor con el terreno.

En la Figura 6.13.25 podemos ver como se adaptan las casas al terreno.

Figura 6.13.25 Casas adaptándose al terreno

6.13.7 Guardar ciudades

Generar esto cada vez al cargar la ciudad suponía un coste computacional demasiado alto, asi que decidimos sacrificar más espacio de disco de los usuarios por una carga más rápida del juego. Para ello era necesario poder guardar las ciudades generadas en un prefab de unity, que funcionaba bien a excepción de los tejados, ya que los planos que se usan en los tejados son creados en meshes directamente escribiendo los vértices y su triangulación, y estos se guardaban en RAM. Para poder guardar estos meshes como "modelos" en Unity hemos creado un sistema para guardar objetos como assets del proyecto, el *SceneDataUtil* que nos permite guardar los objetos que queramos en una carpeta data de la escena.

```
public static class SceneDataUtil
{

    0 references
    public static void CreateAsset(Object asset, string name)
    {
        CreateAsset(asset, "", name);
    }

    7 references
    public static void CreateAsset(Object asset, string subpath, string name)
    {
        string path = GetCurrentDataPath();
        if (subpath != "") path += "/" + subpath;

        path = path.Trim().Replace("//", "/");

        CreateFolderIfNotExist(path);

        string fullPath = path + "/" + name;

        fullPath = fullPath.Trim().Replace("//", "/");

        AssetDatabase.CreateAsset(asset, fullPath);
    }
```

Figura 6.13.26 Guardar assets en el disco duro

Con esto implementado, ya podemos guardar toda la información de una ciudad en un prefab y poder hacer los retoques manuales que creamos necesarios, ya que ahora serán persistentes.

6.13.8 Resultados

En las Figuras 6.13.27 y 6.13.28 podemos ver el resultado final de generar las ciudades del mar y de la montaña, respectivamente, utilizando nuestro sistema de generación procedural.

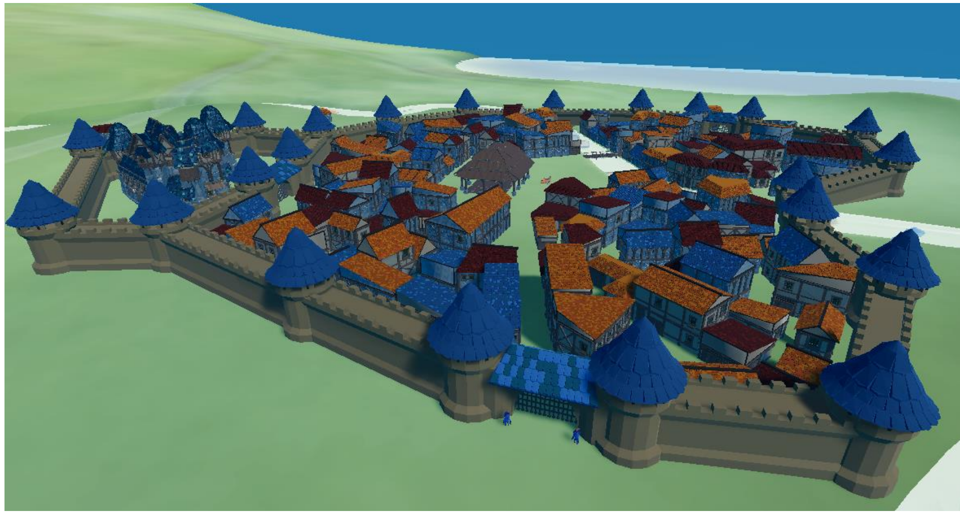

Figura 6.13.27 Ciudad del mar

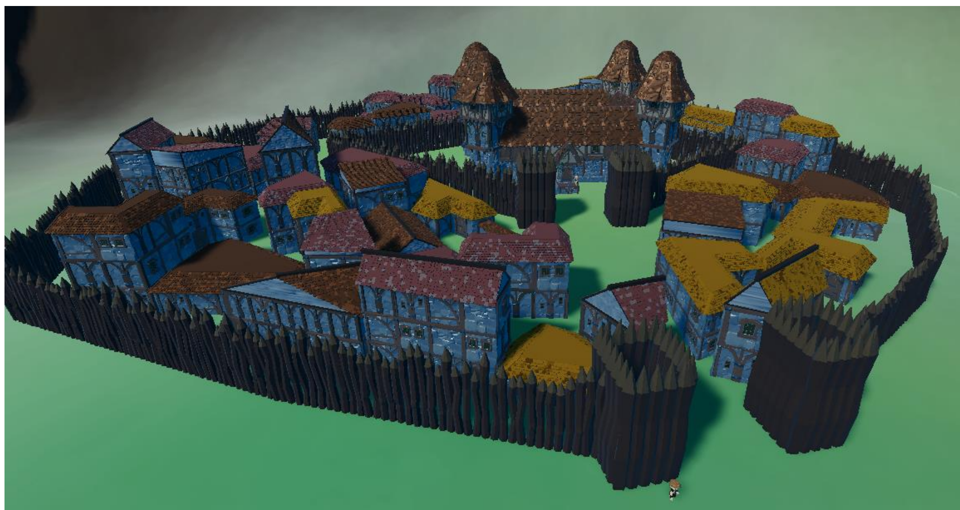

Figura 6.13.28 Ciudad de la montaña

7. Resultados

7.1 Legislación y normativa vigente

El juego desarrollado no presenta ningún problema en aspectos legislativos. Nunca guardamos información de carácter personal del jugador, por tanto, no se aplica en ninguna situación la LOPD (Ley Orgánica de Protección de Datos). Tampoco aplicamos la LSSICE (Ley de Servicios de la Sociedad de la información y Comercio Electrónico), y a que el proyecto no constituye ninguna actividad económica.

Por lo que resulta a los problemas de copyright, a excepción de algunos sonidos de uso completamente libre, el juego ha sido desarrollado completamente por nosotros, por tanto, no nos supone ningún problema si quisiésemos comercializarlo. Aun así, si la comercialización del juego fuese un éxito y superase un cierto lindar de ingresos definidos por los propietarios de Unity, nos veríamos obligados a comprar licencias profesionales del programa para no tener problemas legales.

7.2 Pegi

El sistema Pan Eurpean Game Information (PEGI) es un sistema europeo de calificación de los videojuegos mediante iconos. Se utiliza para clasificar los juegos por edad recomendada y describir el contenido sensible que aparece en el juego, como la presencia de violencia, drogas o sexo. En ningún momento las clasificaciones son de cumplimiento obligatorio, las clasificaciones son meramente informativas. Es decir, a una persona menor de edad nunca le prohibirán comprar un juego que tenga la edad recomendada de 18 años. En la Figura 7.2.1 podemos ver todas las etiquetas de clasificación PEGI.

Figura 7.2.1 Etiquetas calificación PEGI

En el caso de Debt Trading, al ser un juego que usa lenguaje muy soez, muestra violencia de carácter realista y hace mención a las drogas, la etiqueta PEGI correspondiente sería l PEGI 16.

7.3 Resultado final

Figura 7.3.1 Menú inicio Debt Trading

Figura 7.3.2 Mena en la plaza de la ciudad

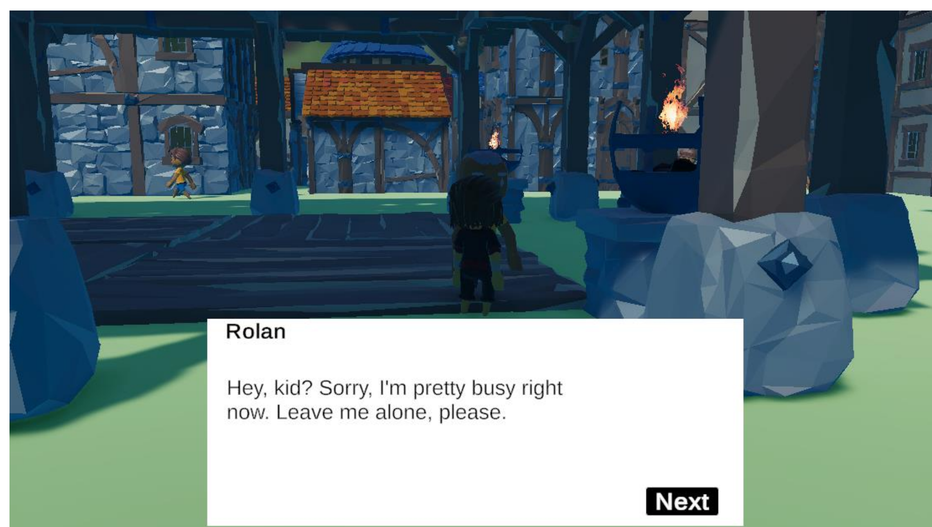

Figura 7.3.3 Mena dialogando con Rolan

Figura 7.3.4 Mena conduciendo el carro

Figura 7.3.5 Mena en las ruinas

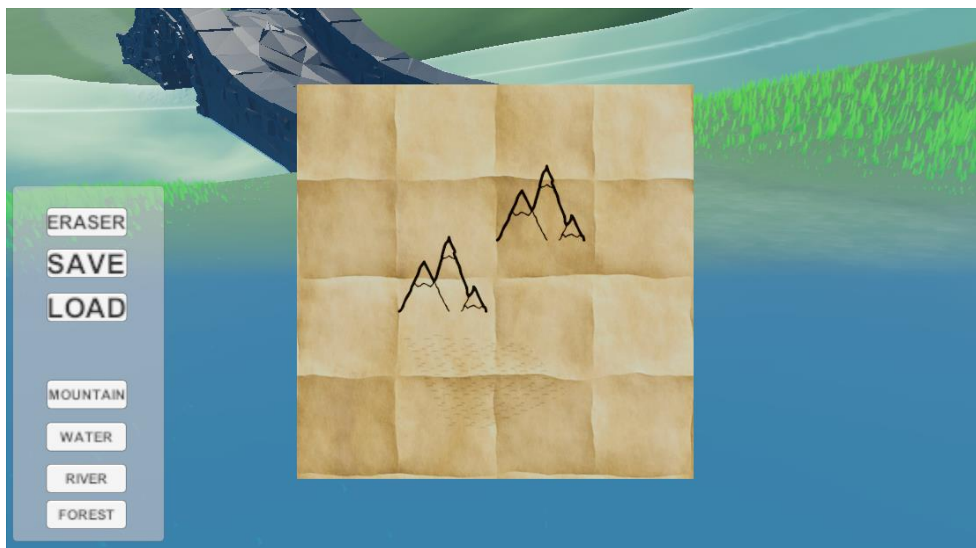

Figura 7.3.6 Modificando el mapa

Figura 7.3.7 Mena comerciando

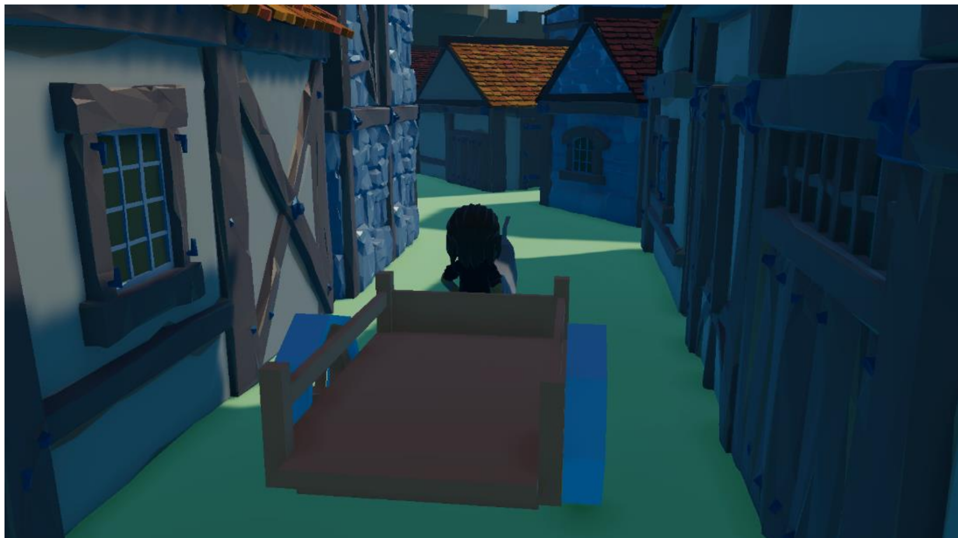

Figura 7.3.8 Mena conduciendo por la ciudad

Figura 7.3.9 Mena conduciendo por la ciudad (2)

Figura 7.3.10 Mena en el prado de fuera de la ciudad

Figura 7.3.11 Mena cogiendo flores

Figura 7.3.12 Mena por las callejuelas

Figura 7.3.13 Mena cogiendo flores

Figura 7.3.14 Castillo de Azariel

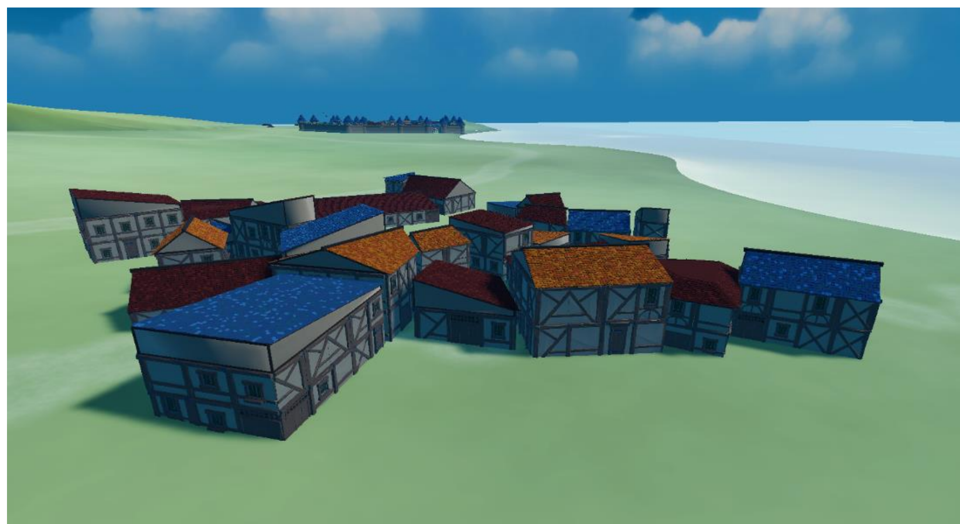

Figura 7.3.15 Aldea del mar

Figura 7.3.16 Vista global del mapa

Figura 7.3.17 Ruinas

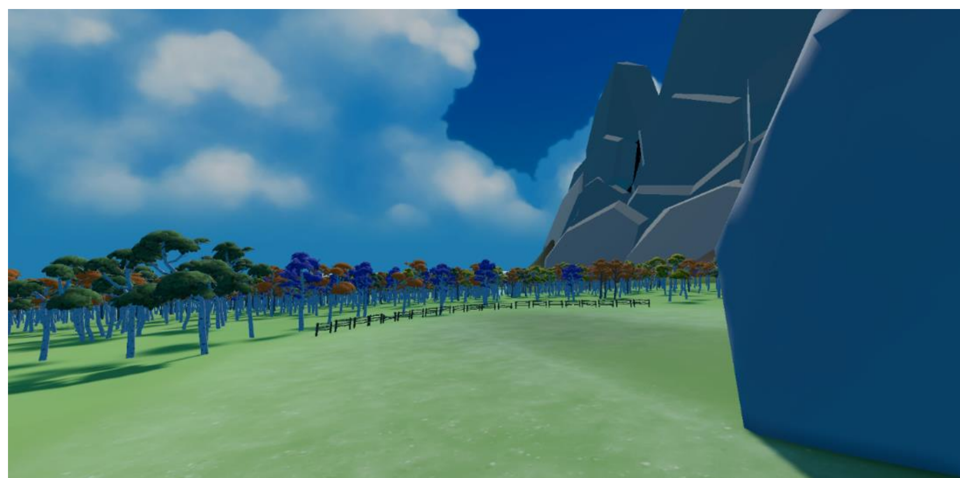

Figura 7.3.18 Bosque

Figura 7.3.18 Salida Bosque

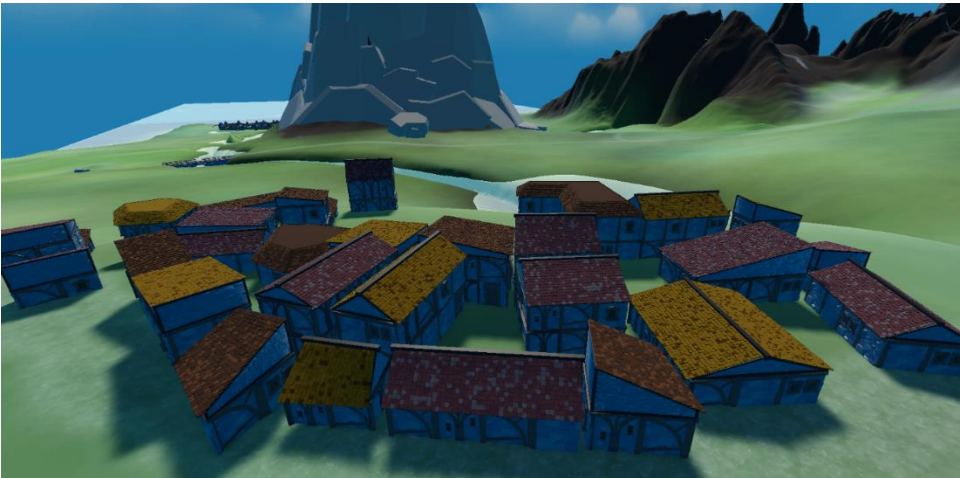

Figura 7.3.19 Aldea Montaña

Figura 7.3.20 Entrada ciudad montaña

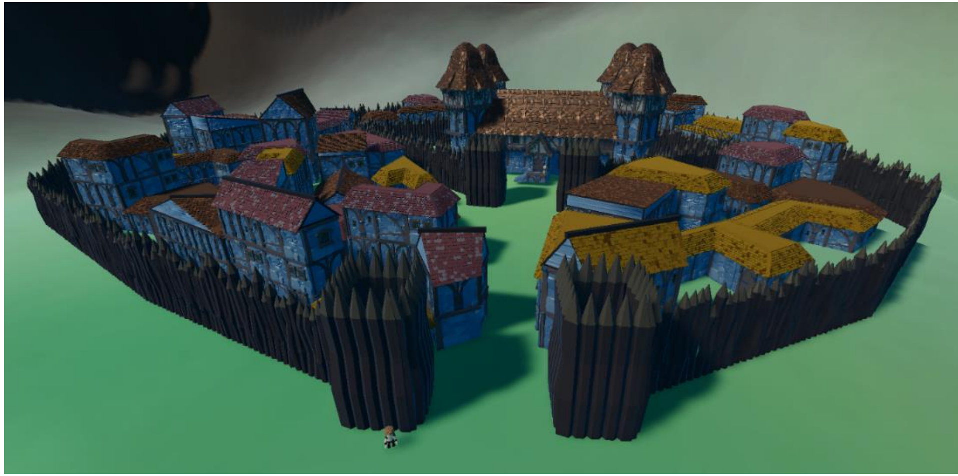

Figura 7.3.21 Vista global ciudad montaña

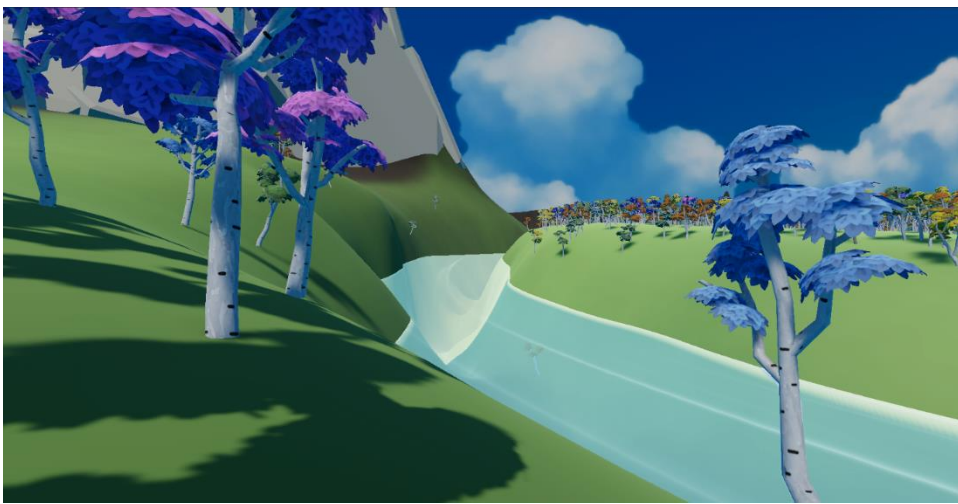

Figura 7.3.22 Río

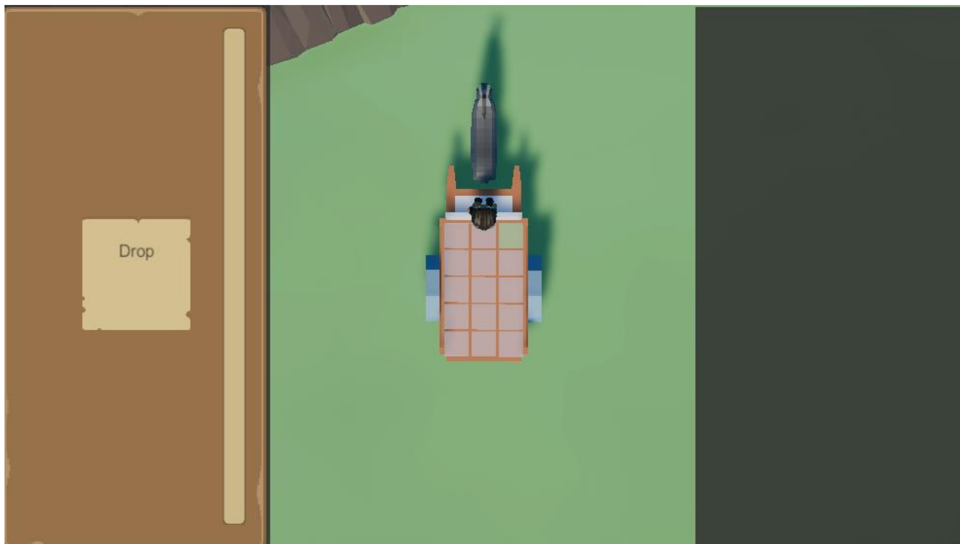

Figura 7.3.23 Inventario carro

8. Conclusiones

8.1 Valoración del trabajo

Debt Trading no es todo lo que habíamos esperado, pero estamos muy orgullosos del trabajo que hemos conseguido hacer en el límite de tiempo disponible.

Durante los años que hemos estado en la carrera de diseño y desarrollo de videojuegos, una de las ideas que hemos aprendido es que todo proyecto siempre será más complicado de lo que uno espera: da igual cuánto te planifiques, cuánto prepares de antemano y cuántos cálculos de tiempo hagas, ya que siempre surgirán problemas, las cosas más básicas se complicarán, y todo tardará más tiempo del esperado.

Es por este motivo principal que, incluso con una idea tan ambiciosa como la nuestra, decidimos recortar todo lo posible. Nos centramos en hacer solo dos ciudades, con una cantidad pequeña de personajes, y con sistemas que fueran simples, pero pudieran dar mucha jugabilidad. Aún con todo esto en mente surgieron problemas y complicaciones, pero conseguimos mantenernos centrados y trabajar al máximo nivel posible.

El tener una idea tan clara nos permitió aprender mucho a la hora de programar. Durante la carrera nos habíamos centrado en hacer juegos de puzles y plataformas, mientras que en este proyecto trabajamos en sistemas como diálogos y misiones, generación de ciudades automática, inventario y movimiento con vehículos, algo nuevo para nosotros.

También tuvimos que trabajar el audio con mucha atención, algo que siempre había estado en segundo plano en nuestros proyectos anteriores. Por primera vez pusimos voz a personajes, y nos centramos mucho en hacer que todos los sonidos, voces y música fueran coherentes con el conjunto del proyecto a un nivel que nunca antes habíamos considerado.

Aún con los diferentes problemas y dificultades que surgieron durante el desarrollo, fuimos capaces de completar todos los objetivos que nos propusimos para el TFG: creamos sistemas de generación de ciudades, un editor de misiones, un sistema de compra venta de objetos y una narrativa que incentivara al jugador a interactuar con los sistemas de juego.

Finalmente, a la hora de hablar de un producto que fuéramos a sacar al mercado, tenemos claro que habríamos de mejorar muchos aspectos, como opciones de accesibilidad, más personajes y misiones y controles por mando, entre otros.

8.2 Desviaciones del trabajo original

Desde el principio ya entendimos que un proyecto de esta envergadura traería diversos problemas que nos cambiarían la planificación original. La generación de ciudades se complicó más de lo esperado, y la unión del sistema de diálogo junto al de misiones trajo problemas inesperados que tardamos en solucionar.

También tuvimos dificultades fuera del ámbito del proyecto: dos de los miembros del equipo encontraron trabajos a jornada completa, mientras que otro aumentó su jornada laboral. Además, un miembro tuvo que hacer una mudanza durante el proyecto, y otros dos empezaron a buscar piso durante los últimos meses, con el tiempo que eso conlleva.

Todos estos detalles causaron que el proyecto se alargara más tiempo del previsto, pero con algunas jornadas un poco más intensas de lo habitual conseguimos corregir y mantenernos, al menos en la idea general, en el margen de tiempo especificado.

9. Trabajo futuro

Estamos muy orgullosos del trabajo que conseguimos hacer para este proyecto, pero aun así consideramos que siempre hay cosas que mejorar. Algunas de las cosas más importantes que nos gustaría añadir a futuro son las siguientes:

- Para comenzar, nos gustaría **mejorar los sistemas que ya se encuentran creados**, como el movimiento del personaje y la compra-venta. A lo largo del desarrollo los hemos mejorado lo que nos ha sido posible, pero somos conscientes de que hay ciertos detalles que pueden estar mejor diseñados.
- También nos gustaría **añadir más animaciones**, sobre todo para Mena y los personajes principales. Las animaciones que hemos usado son las que ya obtuvimos de manera gratuita con el paquete de *assets* de los personajes, y aunque cumplen para este proyecto, si queremos mejorar este videojuego nos hará falta crear animaciones más trabajadas y variadas.
- Aunque el sistema de diálogo está bien implementado y cumple lo que tiene que hacer, nos gustaría, como mínimo, **añadir animaciones durante el diálogo y una cámara cambiante**, que apuntara directamente al personaje que está hablando en un simple plano-contraplano. El diálogo que ya tenemos implementado con estos dos elementos haría el sistema mucho mejor.
- Siguiendo con el punto anterior, también nos gustaría **crear más misiones**, permitiendo al jugador explorar este mundo y conocerlo todavía más.
- Además, **crear y diseñar más personajes principales y secundarios** ayudará todavía más a hacer sentir el mundo más complejo, creando diferentes perspectivas de la situación actual.
- **Implementar controles por mando** es una tarea que se encontraba fuera del alcance para la entrega de este proyecto, pero que en el futuro nos gustaría implementar.
- También, habiendo estudiado durante la carrera temas de accesibilidad, nos gustaría **implementar diferentes opciones de accesibilidad**, que permitieran a gente con problemas visuales y motores disfrutar de nuestro videojuego.
- Finalmente, para mejorar la calidad del desarrollo, nos gustaría **hacer un sistema de diseño de casas**, que permitiera crear construcciones más variadas.

10. Bibliografía

[1] SteamSpy. (2020). *All the data and stats about Steam Games.* https://steamspy.com/year/

[2] PayScale. (2020). *Salary comparison.* https://www.payscale.com/

[3] Pablo Vázquez, VidaExtra. (2019). *Hideo Kojima habla de Death Stranding como un género totalmente nuevo.* https://www.vidaextra.com/accion/hideo-kojima-habla-death-stranding-como-genero-totalmente-nuevo

[4] Hack&Plan. *Project management for games.* https://hacknplan.com/

[5] Shubhankar Parijat, Gamingbolt. (2019). *Death Stranding – Hideo Kojima Clarifies What He Means When He Calls "Social Strand System" A New Genre.* https://gamingbolt.com/death-stranding-hideo-kojima-clarifies-what-he-means-when-he-calls-social-strand-system-a-new-genre

[6] Sebastian Lagüe. (2019). *Bézier Path Creator.* https://assetstore.unity.com/packages/tools/utilities/b-zier-path-creator-136082

[7] Oleg Dolya. (2020). *Medieval Fantasy City Generator.* https://watabou.itch.io/medieval-fantasy-city-generator

[8] Markus Göbel, Unity Community Wiki. (2017). *SimpleJSON.* http://wiki.unity3d.com/index.php/SimpleJSON?_ga=2.192228079.412822287.1595104850-899423616.1587504263

[9] Unity Community Wiki. (2018). *Triangulator.* http://wiki.unity3d.com/index.php?title=Triangulator&_ga=2.225588058.1773793490.1600612646-310516500.1600088647

11. Manual de usuario

11.1 Iniciar el proyecto

Para iniciar el proyecto, el usuario tendrá que entrar en la carpeta de éste y darle clic al ejecutable titulado "DebtTrading.exe". El proyecto se iniciará de forma automática, y cuando termine de cargar el usuario podrá jugar.

11.2 Objetivo del videojuego

En este prototipo, el objetivo del usuario es, como mínimo, completar la misión principal *You Have to be Legal*, que hace que el jugador tenga que obtener un permiso firmado de los dos reyes para poder comerciar en los reinos. Opcionalmente, también se puede completar *Mother's Favorites*, que hace que el jugador tenga que llevar unas flores a Rolan.

11.3 Controles

- Tanto en el burro como a pie, el jugador se mueve con las teclas W, A, S y D.
- Para interactuar con los personajes y objetos (por ejemplo, para iniciar diálogos), el jugador tiene que presionar la tecla E.
- Para abrir el mapa, el jugador ha de presionar la tecla M. Dentro de éste, tiene que interactuar con el ratón exclusivamente, eligiendo qué opciones escoger con un clic y pulsando y arrastrando el ratón en el mapa para dibujar en él.
- Al pulsar la tecla O se abre el libro de cuentas, donde el jugador puede revisar el inventario y las misiones. En el inventario del carro, la tecla R rota los objetos.

- La compra-venta se completa con el uso del ratón para la elección de objetos, y las teclas de las flechas para los minijuegos del sistema.

11.4 Menú de opciones

En la pantalla de inicio del proyecto el usuario puede alterar una serie de opciones, que son la **resolución**, los **fps** (frames por segundo) y si el programa corre a **pantalla completa**.

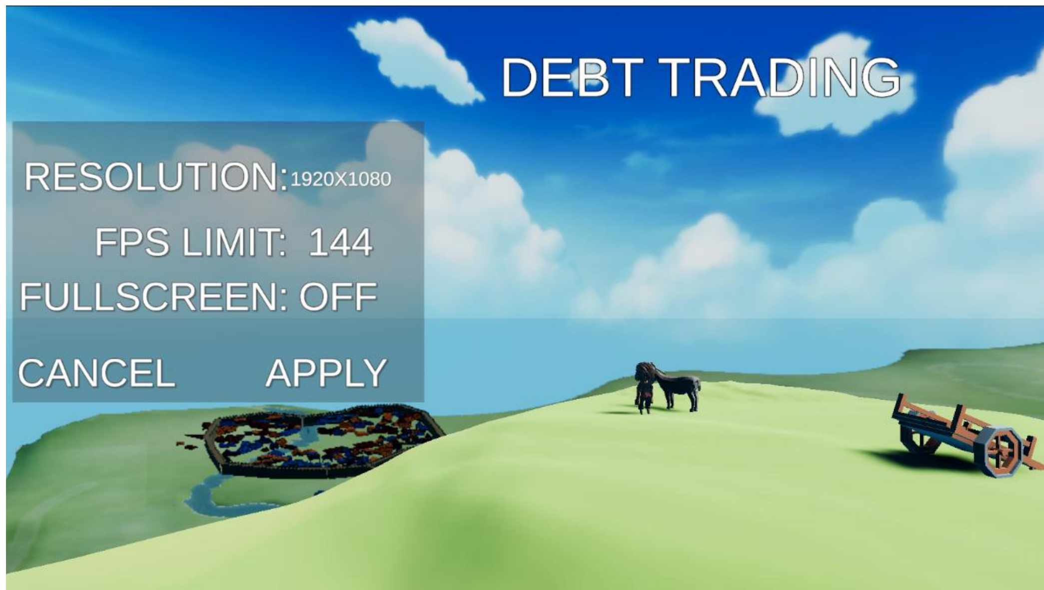

Figura 11.1 Menú de opciones