Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Generación procedimental de entornos exteriores para videojuegos 3D

Procedural generation of outdoor environments for 3D video games

Pablo Sebastián Caballero

La Laguna, 9 de junio de 2020

D. **José Ignacio Estévez Damas**, con N.I.F. 43.786.097-P profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas, como tutor

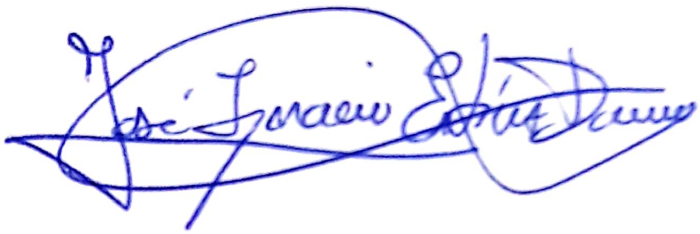
C E R T I F I C A

Que la presente memoria titulada:

”*Generación procedimental de entornos exteriores para videojuegos 3D*”

ha sido realizada bajo su dirección por D. **Pablo Sebastián Caballero**, con N.I.F. X-64.86.491-P.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 9 de junio de 2020



Licencia

Resumen

El objetivo de este trabajo de fin de grado ha sido la creación de un landscape o entorno exterior para un prototipo de videojuego haciendo uso de las técnicas de generación procedural. En este juego, una vez se ha creado el mundo, el jugador puede explorar un amplio terreno compuesto por montañas, árboles, diferentes tipos de vegetación y rocas. Además, el entorno creado es único, por lo que al volver a crear un terreno, no se repetirá el mismo. Para poder conseguir crear este videojuego se hace uso de un motor de videojuegos existente, concretamente el motor Unity, que nos facilita ampliamente la creación de este terreno.

Palabras clave: Landscape, entorno exterior, Generación procedural, videojuego, prototipo, motor de videojuegos, Unity

Abstract

The objective of this final degree project has been to create a landscape or external environment for a videogame prototype using procedural generation techniques. In this game, once the world has been created, the player can explore a wide terrain made up of mountains, trees, different types of vegetation and rocks. In addition, the environment created is unique, so when you recreate a terrain, it will not be repeated. In order to create this videogame, an existing video game engine is used, specifically the Unity engine, which greatly facilitates the creation of this terrain.

Keywords: Landscape, external enviroment, procedural generation, videogame, prototype, videogame engine, Unity

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Generación procedimental y aplicación en los videojuegos

En computación, la generación procedimental puede definirse como el método que permite la creación de datos mediante la aplicación de algoritmos. De este modo se evita la recolección de datos sobre un fenómeno concreto o la producción de los mismos de forma manual.

Esta definición general se puede particularizar al ámbito de las aplicaciones multimedia, que incluye el desarrollo de videojuegos. Los objetos multimedia susceptibles de ser generados por procedimientos son muy variados: texto, imágenes estáticas, vídeos, sonidos ambientales, música, etcétera. En este trabajo nos interesa la creación por procedimientos de escenarios 3D para videojuegos.

¿Qué ventajas obtenemos mediante la aplicación de técnicas procedimentales?. Una ventaja inmediata es la reducción del tamaño de los archivos que conforman la aplicación, pero hay otras características incluso más importantes que van a ser muy relevantes para el videojuego tanto en la fase de diseño del mismo, como en tiempo de ejecución.

Por ejemplo, por un lado, la utilización de algoritmos para diseñar automáticamente contenidos en la fase de diseño, permite reducir el ingente trabajo del equipo artístico. Por otro lado, si hablamos del videojuego en tiempo de ejecución, podremos utilizar algoritmos procedimentales para generar aleatoriedad en el contenido, creando la ilusión de una experiencia renovada y diferente a la ya vivida cada vez que el jugador emprende una partida.

La generación por procedimientos puede ser de dos tipos: determinista y aleatoria. En el primer caso, dados unos parámetros de generación, el algoritmo produce siempre el mismo diseño. En el segundo caso, el proceso generativo emula un experimento aleatorio, es decir, para un conjunto de parámetros hay una distribución de probabilidad asociada a todos los posibles diseños que pueden recrearse por el algoritmo. Este trabajo se basa en generación procedimental aleatoria para construir entornos exteriores para videojuegos.

Esta metodología de generar información se aplica en diferentes facetas del desarrollo de videojuegos. Estas facetas suelen ser:

- Generar un **entorno**, tanto para juegos 2D y 3D, para que el jugador disponga un espacio por el que moverse.

- **Puzzles** en aquellos juegos cuyo objetivo es resolverlos.
- Crear **texturas**. En algunos juegos, existen texturas que se pueden crear a través de la generación procedimental.
- Generar **personajes**. Por ejemplo, personas de una ciudad.

1.2. Objetivo

Durante el desarrollo de este trabajo de fin de grado nos centraremos en la creación de un prototipo de videojuego en el que se genera por procedimiento el entorno exterior en tiempo de ejecución. Por lo tanto, tras el evento que marca el comienzo de la ejecución del juego, se utilizarán un conjunto de parámetros para generar procedimentalmente el entorno. La duración de este proceso debe ser la menor posible para evitar afectar la "experiencia de juego". Dicho entorno es similar al de los juegos de "mundo abierto", en los que el jugador puede visitar diferentes partes del entorno que se ha generado, que son diferentes y con características únicas.

Además, hay que crear un entorno que tenga sentido. Es decir, ciertos elementos deben estar colocados de manera correcta, como por ejemplo, las rocas, césped y árboles deben estar distribuidos por zonas en las que tenga sentido su ubicación. También, deben existir zonas que sean transitables para el jugador y otras zonas que no sean transitables, como podrían ser los bordes del mapa, para que el jugador no los traspase, puesto que si los traspasa caería al vacío.

1.3. Antecedentes y estado actual del tema

En estos antecedentes repasaremos algunos ejemplos de generación procedimental de entornos.

La tipología del videojuego determina qué características del entorno deben ser atendidas con mayor cuidado por el algoritmo de generación por procedimientos. Por ejemplo, en videojuegos como Rogue [8] o Spelunky [11] la generación de mazmorras o recorridos interiores es el aspecto más crítico, ya que se deben cumplir ciertos criterios para que cada nivel sea resoluble con cierto grado de dificultad.

Lo mismo ocurre con videojuegos de estrategia, como es el caso de Elite [3]. En este caso se trata de simular la exploración y combate espacial y lo que se genera procedimentalmente sería el espacio, que está compuesto por un total de 8 galaxias, con sus respectivos planetas y estrellas, cada uno con unas características casi únicas.

Sin embargo, de cara a este proyecto nos interesa dirigir la generación procedimental a la creación del aspecto del paisaje exterior: desde las características del relieve, los caminos transitables, vegetación y otros elementos distribuidos aleatoriamente. En esa línea nuestro proyecto puede tomar como referencia otros videojuegos más recientes como Minecraft [6], que se lanzó en 2011. En este juego se genera un mundo en el que predominan las figuras cúbicas. Además contiene variedad de tipos terrenos y biomas

gracias a una semilla que se usa para generarlos. De manera que, si otro usuario utiliza esa misma semilla se crearía un mundo idéntico.

Por último, tenemos el videojuego No man’s sky [4], en el cual podemos explorar el espacio compuesto por alrededor de 18 quintillones de planetas, cada uno con una fauna y vegetación propia. Estos planetas, animales, plantas, etc. Son generados de manera procedimental, por lo que existe gran diversidad a pesar del elevado número de planetas.

En la actualidad, son varios los desarrolladores que utilizan en sus proyectos la generación procedimental para diseñar el espacio de juego de los usuarios. Concretamente los juegos de mundo abierto son los que más se han beneficiado de esta forma de diseñar sus terrenos, pues reduce los tiempo a la hora de elaborarlos y pueden centrarse en otras partes del juego.

Cabe destacar que la generación procedimental se ha conseguido llevar a cabo gracias a las funciones matemáticas que producen valores pseudo-aleatorios. Uno de los más conocidos es el ruido de Perlin, que genera valores aleatorios pero sin perder la continuidad, es decir, que el valor del adyacente debe ser similar. Por este motivo, se suele emplear para la creación de los mapas de alturas de los terrenos.

1.4. Estructura de la memoria

A lo largo de esta memoria se tratará el concepto de generación procedimental, sus antecedentes y estado actual del tema, o el objetivo del trabajo de fin de grado. Más adelante, se describirán aquellas tecnologías y herramientas que se han usado en el desarrollo de este videojuego. Se hablará del concepto de motor de videojuegos y cual se ha escogido, el lenguaje sobre el que se programará, y otras herramientas que nos proporciona el motor.

Luego, se explicará como ha sido el desarrollo, mostrando los problemas surgidos y las soluciones que se han ido proponiendo a lo largo de la elaboración de este videojuego. Esta parte técnica de la memoria, incluye un apartado importante: el relativo al uso de técnicas para optimizar el rendimiento del juego.

La memoria finaliza con la enumeración de las principales conclusiones, así como un análisis sobre mejoras y líneas futuras del trabajo. Se incluye además un resumen del trabajo en inglés. Y por último, se muestra un presupuesto de la cantidad aproximada de dinero que podría costar el desarrollo de este prototipo.

Capítulo 2

Tecnologías utilizadas

Durante la realización de este trabajo de fin de grado se han empleado diversas tecnologías y herramientas. En este apartado se explicará características sobre ellas y su utilidad.

2.1. Unity

2.1.1. Motor de videojuegos

Se entiende como motor de videojuegos [7] al conjunto de herramientas y librerías de programación que permiten diseñar y poner en funcionamiento un videojuego, y que a su vez facilitan el desarrollo de los juegos a los programadores. Los motores más complejos como Unity, Unreal Engine o Cry Engine, tienen una interfaz que nos permite administrar todos los elementos del juego, como la organización en carpetas de los scripts, objetos, iluminación y otros elementos importantes.

Un claro ejemplo en el que se aprecia la ayuda de estos motores sería la creación de escenarios. En este ejemplo, el motor facilita el trabajo a los diseñadores que se dedican a crear entornos para los jugadores escogiendo objetos que se habían creado previamente y colocándolos en el sitio deseado, pudiendo duplicar el objeto o modificar sus parámetros rápidamente.

EL motor de los videojuegos proporciona la abstracción de hardware, que permite a los programadores crear videojuegos sin tener que conocer la arquitectura hardware de la plataforma para la que se va a trabajar. Gracias a esto, gran parte de los motores se desarrollan a partir de APIs existentes como OpenGL o DirectX.

Además, cabe destacar que en la mayoría de motores tenemos otros dos motores muy importantes:

- Motor **gráfico** que se utiliza para mostrar por pantalla los elementos 2D o 3D del juego.
- Motor de **físicas** que se encarga de la simulación de colisiones, gravedad e inercia de los objetos.

Otras funciones de las que se encarga un motor sería:

- Animación de personajes
- Ejecución de scripts
- Sonido
- Inteligencia artificial
- Administración de memoria
- Sistema de gestión de eventos del juego
- Entrada/Salida
- Organización y control general del flujo del juego
- Sistema distribuido para juegos multijugador en línea y comunicaciones

2.1.2. Motor Unity

Unity es un motor de videojuegos desarrollado por la empresa Unity Technologies, que lanzó su primera versión a mediados de 2005, y es el motor que se utilizará para elaborar el prototipo del videojuego. Este motor es empleado en gran parte por lo desarrolladores independientes que no pueden permitirse la creación de su propio motor debido a la complejidad, tiempo y esfuerzo que esto conlleva. Por ello, son muchos los desarrolladores independientes que descartan la creación de su propio motor, sabiendo que existen motores como Unity que, además de ser gratuitos, ofrecen herramientas avanzadas que quizás uno mismo no puede desarrollar.

Unity [9] posee características que lo hacen atractivo para todos los usuarios e incluso empresas desarrolladoras de videojuegos. Entre todas ellas se puede destacar que su curva de aprendizaje es mucho más suave que otros motores, lo que permite que se puedan implementar aquellas ideas que tengamos de una manera más ágil.

También, tiene una interfaz que podemos ver en la figura 2.1 la cual es muy intuitiva y que hace mucho más fácil la creación de nuestro juego. En esta interfaz tenemos en la parte izquierda las lista de objetos que se encuentran en la escena, como podría ser el terreno, los arboles, rocas, etc. Luego, en la parte inferior tenemos el acceso a las distintas carpetas de nuestro proyecto, desde la cuales podemos acceder a nuestros scripts, materiales u objetos prefabricados para modificarlos o añadirlos a la escena.

Por la parte derecha tenemos la pestaña llamada "Inspector", que nos permite ver los parámetros del objeto que tengamos seleccionado y modificarlos a nuestro gusto. Y por último, en la parte central tenemos la vista del juego, tanto desde la perspectiva del editor que nos permite movernos rápidamente en el entorno que hemos creado, y la vista del jugador, que es la vista desde la cámara que controla el jugador. Además, están los botones de "play", para ejecutar el juego o reanudarlo después de haberlo pausado con el botón de pausa, que se suele utilizar para detener cualquier ejecución en el juego y poder ver el estado actual de los objetos. Y también tenemos el botón para pasar a la siguiente escena, en caso que tengamos más de una.

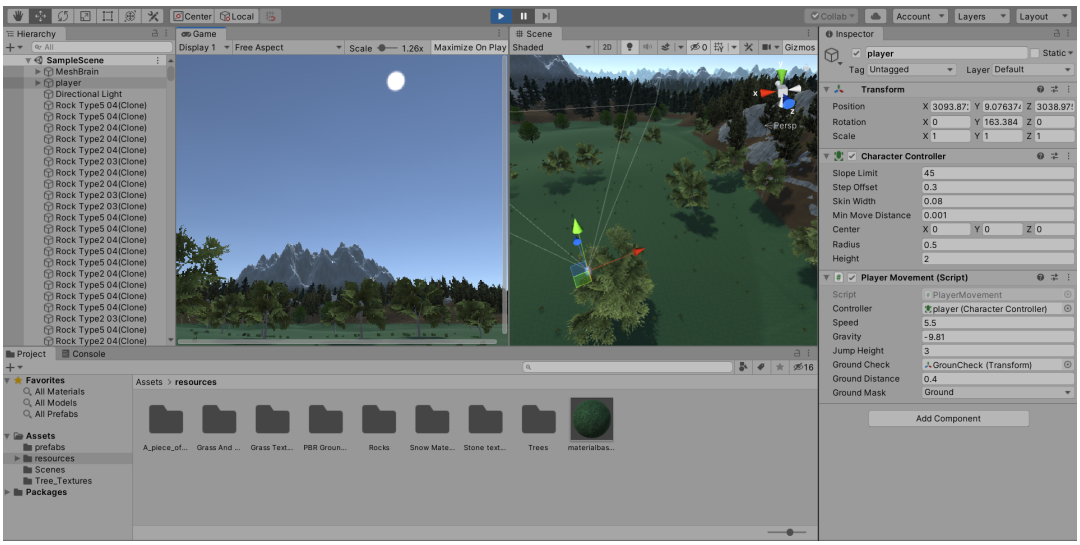


Figura 2.1: Interfaz del editor de Unity

También hay que resaltar que Unity posee varios tipos de planes para los usuarios. El plan más básico sería el personal que permite usar Unity de manera gratuita y que solamente se recortan un par de características más avanzadas que para el desarrollo de nuestros juegos no nos perjudicarían.

Otro de los puntos fuertes de Unity es que es considerado el software más versátil a la hora del desarrollo multiplataforma. Esto se debe a que permite desarrollar juegos en más de 25 plataformas, de las que se destacan IOS, Android, Oculus, Steam VR, Gear VR, Windows, Mac, Linux, PlayStarion 4, Xbox One, Nintendo 3DS o Nintendo Switch.

Esta capacidad de poder crear videojuegos para múltiples plataformas, a pesar de programar en un único lenguaje que no es nativo para la plataforma objetivo, se consigue gracias a Mono. Mono es un software de código abierto incluido en Unity que posee un gran número de librerías para hacer funcionar un determinado tipo de lenguaje, en este caso sería c#, y que en tiempo de ejecución puede convertirlo en el código nativo. Por lo que, al compilar nuestro juego, Mono añade al juego compilado un entorno de ejecución que actúa como maquina virtual para hacer funcionar el código en cualquier plataforma.

Este motor se puede comparar directamente con el Unreal Engine, ya que ambos son los motores de juegos más utilizados por los desarrolladores independientes debido a que ofrecen muchas facilidades a la hora de trabajar además de ser gratuitos. Sin embargo, cada uno de estos motores tienen sus ventajas y desventajas. Comenzando por Unity hay que destacar que:

- El desarrollo con Unity es más fácil, pensado más para los principiantes.
- El lenguaje C# es potente y sencillo de aprender.
- La interfaz de Unity es fácil de utilizar.
- Existe mucha documentación y una gran comunidad para resolver dudas.
- El desarrollo con Unity es cómodo, rápido y eficiente.

- Unity tiene el concepto de prefab, que es un conjunto de objetos con variables ya predefinidas y que agilizan la instanciación de las copias de estos prefabs.
- En Unity la mayoría de los materiales pueden crearse usando el material estándar.
- El precalculdado de la luz es genial y rápido. Pero el post-procesado es débil.
- Se puede exportar a un mayor número de plataformas y de manera sencilla.

Por otro lado, de Unreal Engine podemos destacar que:

- Es más difícil de aprender a usar que Unity. No está pensado para los principiantes.
- El lenguaje C++ es más complejo. Los tiempos de compilación son más largos.
- Es de código abierto.
- Ofrece la posibilidad del desarrollo más visual gracias a los 'blueprints'.
- Unreal presenta herramientas más avanzadas para la creación de terrenos y vegetación.
- La creación de materiales es más avanzada que la de Unity. Pudiendo crear materiales y shaders de manera fácil.
- Los archivos de los proyectos son más grandes que en Unity, pero tienen un resultado más sólido.

2.2. C Sharp

C Sharp o conocido también como C#, es un lenguaje de programación orientado a objetos que se ha desarrollado por Microsoft [5]. Este lenguaje se empleará para crear los scripts en Unity. Estos scripts son los que permiten la creación del terreno de manera procedimental.

Este lenguaje surge como la evolución de los lenguajes C y C++, adoptando aquellos atributos positivos de estos dos y adaptándolo a los tiempos actuales. Gracias a ello, C# tiene una sintaxis sencilla y fácil de aprender para todos aquellos que ya hayan trabajado con C, C++ o Java.

A pesar de que los lenguajes C# y C++ son similares, se diferencian en ciertas características que hacen a los usuarios decantarse por un lenguaje u otro [2].

- A lo hora de aprender, el lenguaje c# es mucho más sencillo que C++.
- C# tiene un recolector de basura que hace las cosas más fácil, al no tener que preocuparnos tanto por problemas de la memoria.
- En C++ podemos utilizar los punteros aritméticos para referirnos a un espacio de memoria, mientras que en C# no existen. En su lugar, en C# existen los delegados, que permiten almacenar la referencia de un método.

- El lenguaje C++ se encuentra en muchas plataformas, mientras que c# solo se encuentra en sistemas Windows.
- c# al ser un lenguaje relativamente nuevo, no tiene tanta popularidad y no se utiliza en muchas aplicaciones. Además, existe menos documentación sobre c# que sobre C++.

2.3. Depurador de Unity

A la hora de desarrollar videojuegos suelen surgir problemas que a simple vista no podemos entender porqué suceden. Es en estos casos que se suele recurrir a los depuradores. En Unity tenemos un depurador bastante útil y que nos brinda todo tipo de información. Este depurador tiene el nombre de "profilerz a través de una gráfica nos muestra por cada instante de ejecución el tiempo que tarda en ejecutarse partes del juego como las físicas, script o el renderizar los elementos por pantalla. Gracias al profiler, también podemos ver el estado actual del funcionamiento de nuestro juego, pudiendo obtener información acerca del uso de CPU, renderizado, la memoria, Audio, las físicas, así como la iluminación global.

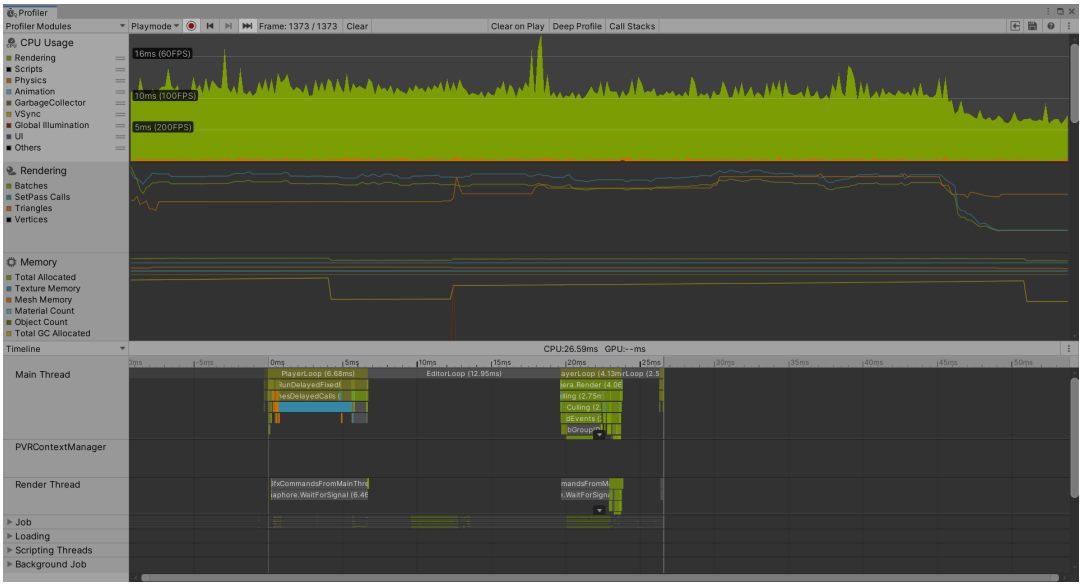


Figura 2.2: Gráficas y estadísticas obtenidas del profiler

Además, si seleccionamos por ejemplo, en "memory"nos mostrará mas estadísticas que nos indican la cantidad de memoria que ocupan las texturas o los modelos que estén en el videojuego.

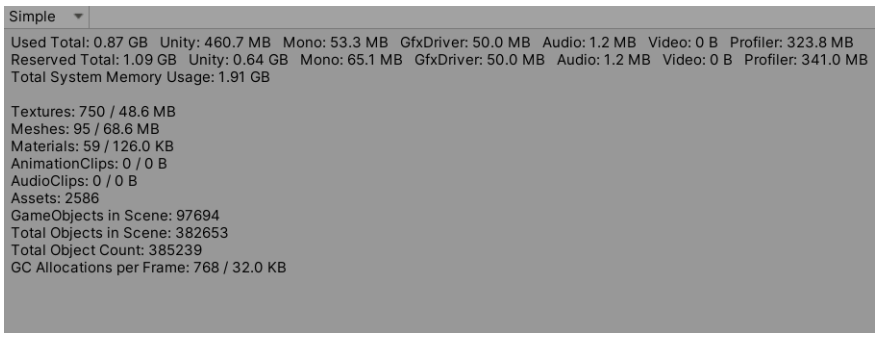


Figura 2.3: Datos obtenidos sobre la memoria durante la ejecución del juego

Por otro lado, existe otro modo de funcionamiento más completo de este depurador conocido como “deep profiler”. En este modo, podemos ver estadísticas de la ejecución de cada uno de las funciones de nuestros scripts, como poder ver si nuestra función tarda demasiado en finalizar su ejecución o si está ralentizando el funcionamiento de nuestro juego al consumir mucha memoria.

2.4. Asset Store

El Asset Store de Unity es una especie de biblioteca que contiene una gran variedad de tipos de elementos que se pueden aplicar a los juegos que creemos. Estos elementos pueden ser objetos como árboles, rocas, materiales, sonidos, efectos, etc. La mayoría de estos elementos son subidos por los propios usuarios, y además, existen tanto assets de pago como gratuitos. Para este prototipo, se recurrirá a la asset store principalmente para importar a este proyecto algunos materiales y algunos modelos como árboles y rocas.

Capítulo 3

Desarrollo

En este capítulo se mostrará como ha sido el desarrollo de este trabajo de fin de grado, viendo los problemas surgidos y la solución planteada para resolver cada uno de estos problemas.

3.1. Construcción por procedimientos de la superficie con sub-mallas

Para poder representar el terreno de este videojuego había que buscar la manera de poder crear la geometría del terreno y poder manipular dicha geometría a nuestro gusto. En este caso, se ha escogido utilizar la clase Mesh de Unity, ya que nos permiten crear mallas con la geometría que queramos y poder modificar las características que necesitemos de esta malla. Gracias a las mallas, fue posible crear una especie de plano geométrico de un ancho y un largo que se quisiera.

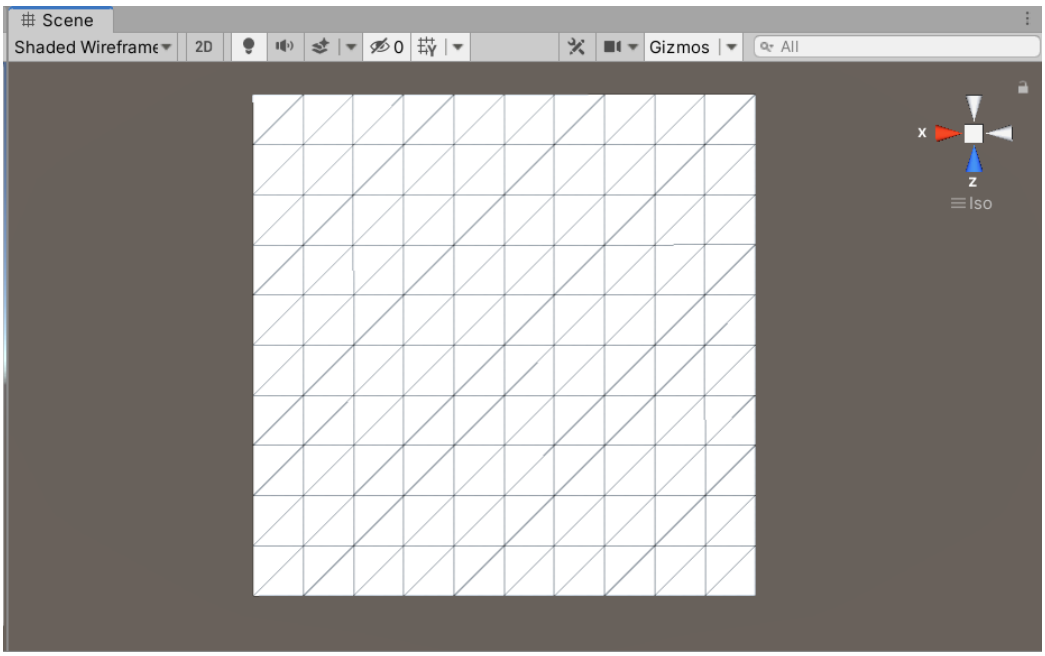


Figura 3.1: Ejemplo de un mesh sencillo

Como se puede ver en la figura 3.1, se ha creado una malla de tamaño 10 x 10, lo que

equivale a 100 metros cuadrados. Para crear esta malla se ha implementado un algoritmo que dependiendo del ancho y largo escogido, calcula el número de vértices y triángulos necesarios y los coloca en el lugar correspondiente para poder representar la malla en el juego.

3.2. Altura de los vértices

Ahora que ya disponemos de este plano, hay que darle forma para que tenga un aspecto más parecido a un terreno natural. Para darle forma, lo que se hace es coger cada uno de los vértices de la malla y se le asigna una altura. Sin embargo, esto no es tan sencillo como parece. Una de las primeras ideas que surgieron para calcular la altura de estos vértices fue la de utilizar la función que genera números aleatorios, pero esta función no iba darnos el aspecto deseado debido a que la altura de los vértices variaría mucho respecto a la de los vértices adyacentes.

Después de investigar la manera en la que se podía darle forma a la malla para que luzca más natural, encontré el Ruido de Perlin o Perlin Noise [1]. Dicha función lo que consigue es generar números pseudo-aleatorios pero siguiendo, por así decirlo, una continuidad.

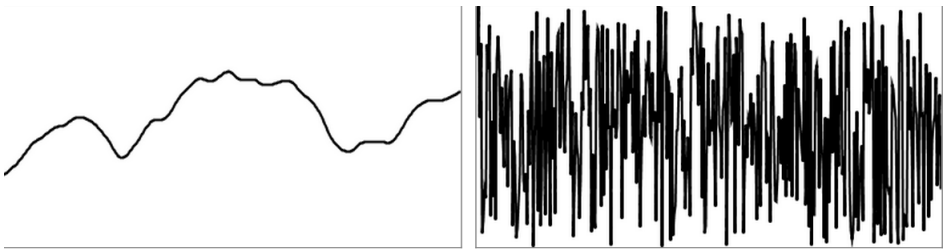


Figura 3.2: Comparativa entre la función Perlin Noise a la izquierda y la función random a la derecha

En la figura 3.2 tenemos una comparación entre el Perlin noise a la izquierda y la función random a la derecha. Fácilmente se puede apreciar la continuidad que se había mencionado previamente del ruido de Perlin, lo cual hace que sirve de gran ayuda para este prototipo.

Para usar esta función, la cual ya viene implementada en Unity, necesitamos pasarle unos parámetros que son las coordenadas del eje X y eje Z, y nos devolverá como resultado un número flotante entre 0 y 1. Como conocemos la ubicación exacta de cada uno de los vértices de nuestra malla, podemos utilizar el Perlin noise para asignarle una altura con las coordenadas de cada vértice.

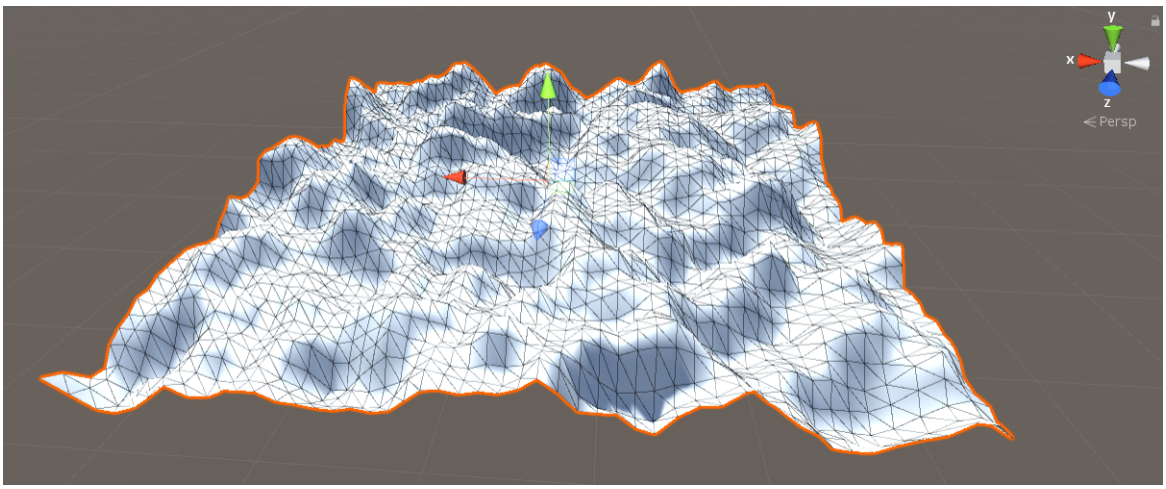


Figura 3.3: Malla con alturas asignadas a cada vértice

Como podemos observar en la figura 3.3, la forma terreno luce algo más natural. Pero todavía podemos realizar más ajustes a través de la combinación de diferentes ruidos de Perlin, que se tratará en la sección 3.4.

3.3. Tamaño de la malla y combinación de mallas

Ahora que ya podemos crear una malla y darle la forma deseada, lo ideal sería generar un terreno de 1 kilómetro de ancho y de largo. Teniendo en cuenta que en nuestra malla cada uno de los vértices de cada fila y columna están separados a 1 metro de distancia, habría que crear una malla de alrededor del 1 millón de vértices. No obstante, existe un límite en cuanto a las dimensiones de la malla. En este motor se limita el número de vértices de cada malla a 65535. Por lo que surgen dos nuevos problemas. En primer lugar, hay que buscar un tamaño que sea idóneo y que no sobrepase el límite establecido por el motor. Por otro lado, al no poder utilizar una única malla para todo el videojuego habrá que ir creando el terreno por secciones más pequeñas.

El tamaño de la malla que se ha escogido es de 240 metros, lo que equivale a 241 vértices tanto a lo ancho como a lo largo. Se ha elegido este tamaño debido a que es de los números más altos posibles que se puede escoger, y además cuenta con un gran número de divisores, cuya utilidad se explicará detalladamente en el siguiente apartado.

Para poder crear el terreno del videojuego y cuyo tamaño sea relativamente amplio, habrá que ir creándolo por secciones o chunks. Cada chunk se compone principalmente de una malla y cada una de ellas son del mismo tamaño y se generan de la misma manera. Siguiendo estos pasos, cada chunk estaría ubicado en el mismo espacio y el aspecto sería igual para todos. Es por ello que habría que desplazarlos cada uno de los chunks a su posición correcta y de manera que no existan espacios entre las mallas ni que estén superpuestas entre ellas.

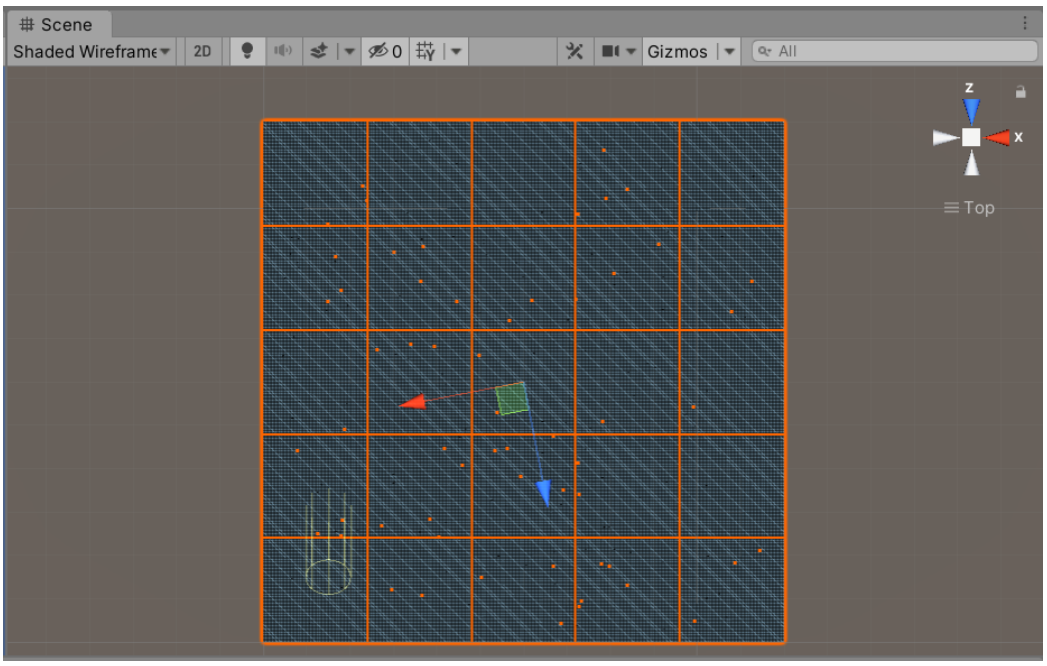


Figura 3.4: Conjunto de 25 mallas

Como podemos observar, en la figura 3.4 tenemos un terreno de 5x5, es decir, que tiene un total de 25 chunks o mallas, y que todas están bien colocadas.

Otro punto que hay que tener en cuenta es que cada malla debe usar el ruido de Perlin pero desplazado, para que no se genere siempre la misma malla. De modo que la primera malla que se genere tendrá de desplazamiento en ambos ejes de 0 unidades, mientras que la que se haya creado a su derecha tendrá 240 unidades de desplazamiento solamente en el eje X y así con todas las mallas.

3.4. Tipos de terreno y texturizado

Uno de los objetivos que también se debe cumplir en este videojuego es que en el terreno existan diferentes tipos de terrenos y que a su vez tengan una textura para cada zona, lo que permitirá diferenciarlas. Por lo que se crearán zonas en las que el terreno sea más llano y el jugador se pueda desplazar sin problemas. Otra de las zonas que se creará será la montañosa, en la que al jugador le es casi imposible transitar debido a que la pendiente de estas es muy pronunciada. Y por último, una zona intermedia que no sea ni tan llana como la primera ni con tanta pendiente como la montañosa, para que el jugador pueda moverse pero que no sea tan sencillo. Para conseguir esto, se creará lo que se conoce como un mapa de alturas utilizando el ruido de Perlin.

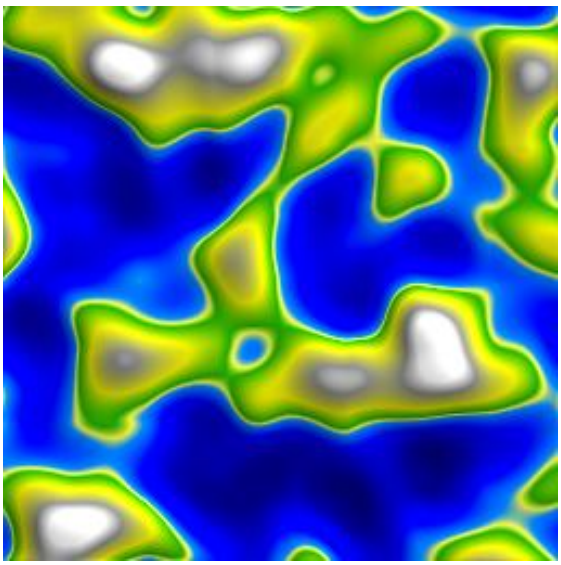


Figura 3.5: Ejemplo de un mapa de alturas

En la imagen de la figura 3.5 se muestra un ejemplo de un mapa de alturas en la que dependiendo de la altura se pinta de un color u otro. Para este prototipo se utilizará una idea similar. El proceso aleatorio para determinar las alturas que se ha implementado tiene dos niveles: en primer lugar se decide el tipo de terreno, y en segundo lugar se obtiene la altura final del vértice.

Para decidir el tipo de terreno se hace uso de un mapa de alturas que se ha creado utilizando el ruido de Perlin. Sabiendo que esta función devuelve valores entre 0 y 1, se puede crear las diferentes zonas utilizando rangos de valores de la función. De este modo, por debajo de 0.5 se crearán terrenos llanos, entre 0.5 y 0.7 habrá terrenos intermedios, y el entre 0.7 y 1 se creará terrenos montañosos o más elevados.

Como ya sabemos las zonas a las que pertenece cada vértice de las mallas, procedemos ahora a modificar la función que calculaba la altura de los vértices. Para ello, habrá que utilizar una función de Perlin diferente para cada una de las zonas existentes. No obstante, esta estrategia tiene el problema de que no se consiguen transiciones suaves entre los diferentes tipos de terreno.

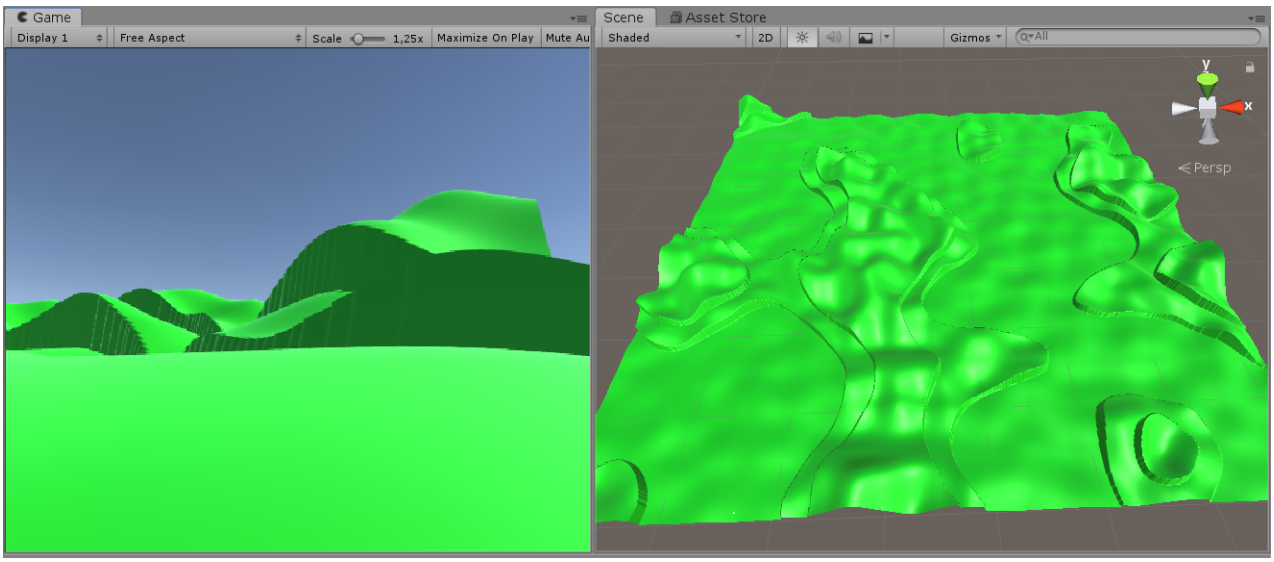


Figura 3.6: Terreno con problemas en la transición de zonas

En la figura 3.6, podemos observar la existencia de diferentes zonas en el terreno. Sin embargo, lo que habría que lograr es que haya una transición más suave entre los distintos tipos de zonas y que no haya límites tan abruptos. Para poder resolver este problema, se utiliza la técnica la interpolación lineal, que en Unity se consigue con la función Lerp. Dicha función, necesita 3 parámetros para funcionar: el primero es el valor mínimo, el segundo es el valor máximo, y el tercero sería el valor de interpolación que varía entre 0 y 1. Un ejemplo sencillo sería utilizando los valores 3 como mínimo , 5 como máximo y 0.5 como valor de interpolación, el valor que nos devolvería la función Lerp sería 4.

Para poder aplicarlo en este videojuego y solucionar los problemas del terreno, primero habrá que crear otras dos nuevas zonas que se usaran para pasar de una zona a otra. Para ello, en el caso de la transición de la zona llana y la zona intermedia, habrá que hacer un interpolación lineal teniendo como valor mínimo el resultado que de la función de la zona llana, como valor máximo se utilizará el resultado de la función de la zona intermedia, y el valor de interpolación variará según el punto de la zona de transición que vayamos a utilizar.

```
public float asignar_altura(float x,float z){

    float xp = x+xPosition;
    float zp = z+zPosition;

    float frec = .002f; //.005
    float mapa = Mathf.PerlinNoise((xp+map_seed)*frec,(zp+map_seed)*frec)*1f;

    if(mapa<0.4){
        return flat(x,z);
    }

    if((mapa>=.4)&&(mapa<0.6)){
        float rango = (mapa -.4f)/(.6f-.4f);
        return Mathf.Lerp(flat(x,z),hill(x,z),rango);
    }

    if((mapa>=0.6)&&(mapa<0.7)){
        return hill(x,z);
    }
    if((mapa>=0.7)&&(mapa<0.85)){
        float rango = (mapa -.7f)/(.85f-.7f);
        return Mathf.Lerp(hill(x,z),mountain(x,z),rango);
    }
    if(mapa>=0.85){
        return mountain(x,z);
    }
    return 100f;
}
```

Figura 3.7: Código del algoritmo de las alturas

En el código que se ve en la figura 3.7, tenemos una función a la que se la pasan 2 valores, que representan la posición en los ejes X y el eje Z de un vértice de una malla. Luego, se calcula cual sería el valor del mapa de alturas para la posición del vértice. Como podemos ver, existen hasta 5 rangos diferentes. Hay 3 rangos que hacen referencia a las 3 zonas que existen en el mapa que son la zona llana, la intermedia y la montañosa. Y los dos rangos restantes corresponden a las zona de transición entre las 3 zonas principales. Como podemos observar, para cada zonas se retorna otra función que calcula la altura que debe tener el vértice.

```
float flat(float x,float z){
    float xp = x+xPosition;
    float zp = z+zPosition;
    float f1=.01f;//.02

    float a1 =10f;//10
    return (Mathf.PerlinNoise(xp*f1,zp*f1)*a1);
}
float hill(float x,float z){
    float xp = x+xPosition;
    float zp = z+zPosition;
    float f1=.01f;//.02
    float f2=.025f;//.05

    float a2=40f;//20
    return (Mathf.PerlinNoise(xp*f1,zp*f1)+
    Mathf.PerlinNoise(xp*f2,zp*f2))*a2 +
    Mathf.PerlinNoise(xp*0.2f,zp*0.2f)*2.0f;
}
float mountain(float x, float z){
    float xp = x+xPosition;
    float zp = z+zPosition;
    float f1=.005f;//.02
    float f2=.012f;//.05
    float f3=.025f;//.1

    float a3=100f;//50
    return (Mathf.PerlinNoise(xp*f1,zp*f1)+
    Mathf.PerlinNoise(xp*f2,zp*f2)+
    Mathf.PerlinNoise(xp*f3,zp*f3))*a3+
    Mathf.PerlinNoise(xp*0.1f,zp*0.1f)*15.0f+
    Mathf.PerlinNoise(xp*0.2f,zp*0.2f)*10.0f;
}
```

Figura 3.8: Código de las funciones para calcular las alturas

En la figura 3.8 podemos ver las distintas funciones que se utilizan para calcular las alturas. También se puede apreciar que se combinan diferentes ruidos de Perlin dependiendo de la zona.

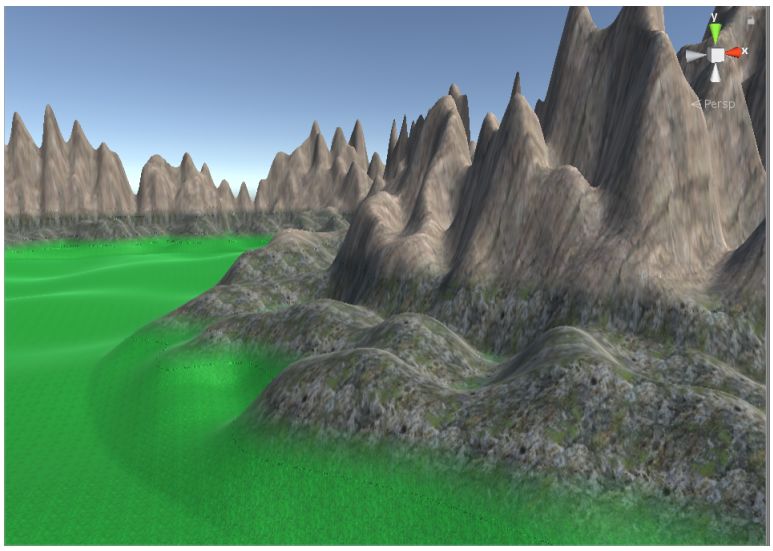


Figura 3.9: Distinción de zonas y de texturas en el terreno

Como se puede apreciar en la figura 3.9, se ha conseguido que la transición entre las zonas sea mucho más suave, lo que le aporta un toque más de realismo.

Además, podemos ver que el terreno cuenta con texturas. La manera en la que se aplican estas texturas es a través de un shader [10]. Un shader es un pequeño script que permite indicarle al motor cómo se debe procesar el color de cada píxel de los materiales de un determinado objeto. En este caso, el shader que se ha programado es bastante sencillo, y permite que en función de la altura a la que se encuentre el píxel de la malla, pintará el píxel con una textura u otra. Destacar también que la transición entre texturas que se puede ver en la figura 3.10 se realiza con la interpolación lineal.

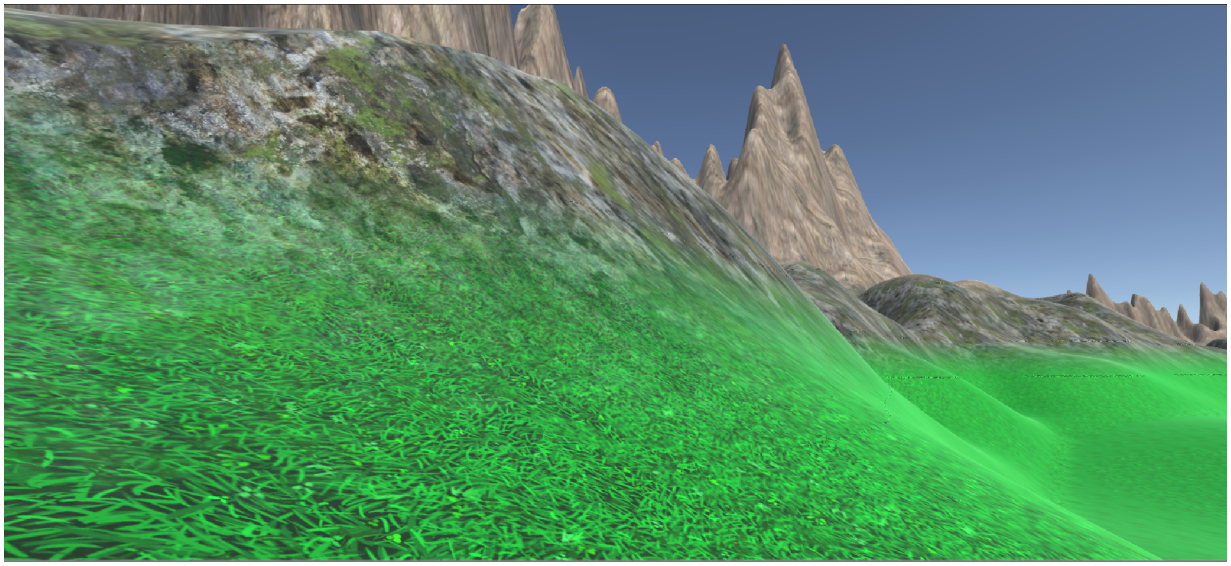


Figura 3.10: Texturas mezcladas

3.5. Nivel de detalle de la malla

En este punto, en el prototipo se puede generar un terreno del tamaño que queramos gracias a que está dividido en chunks, y el aspecto que tiene se acerca al deseado. Sin embargo, al ir testeando mapas de mayor tamaño se puede ver que el rendimiento a la hora de generar el terreno va siendo cada vez más lento. Esto se debe a la gran cantidad de triángulos y vértices de la que esta compuesto cada malla.

De modo que para poder solucionar este problema, se eligió como solución incorporar al juego un algoritmo que permita ir modificando el aspecto de la malla en función de la ubicación del jugador. Este algoritmo se ejecuta en cada uno de los chunks, por lo que cada uno de ellos comprueba la ubicación del jugador para poder establecer el nivel de detalle con el que se representa su malla.

Para poder entender como funciona este algoritmo, se utilizará un pequeño ejemplo de una malla de tamaño 12 x 12.

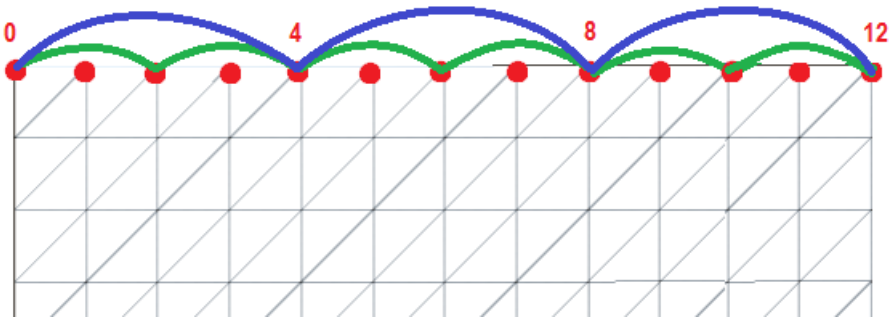


Figura 3.11: LOD aplicado a una malla de 12 x 12

En la figura 3.11, tenemos representados 3 niveles de detalle. En el nivel 1, se unen los vértices uno a uno, por lo que el tamaño sería el mismo que el original. El nivel 2 está representado de color verde, y se puede observar que los saltos entre los vértices se realizan de dos en dos. De modo que se reduce la cantidad de vértices utilizados sin alterar el tamaño en el espacio que ocupa la malla, por lo que conseguimos una malla de tamaño 6 x 6. Los mismo sucede con el nivel 3 que está representado con el color azul. Similar al caso del nivel 2, pero en esta caso, los saltos de vértices se hacen de cuatro en cuatro, consiguiendo así una malla de 3x3. En este ejemplo a pequeña escala, los cambios producidos no son muy notorios, pero aplicando este concepto a nuestro terreno, se consigue una mejora de rendimiento muy significativa.

Como se ha mencionado anteriormente, el tamaño original de cada uno de las mallas es de 240 metros o 241 vértices a lo largo y ancho, que equivale a un total de 58.081 vértices. Esto es ideal para los casos en los que el jugador esta directamente en el chunk o cerca de él. Sin embargo, cuando el jugador se encuentra a una distancia más lejana le es casi imposible percibir que esa malla tiene un buen nivel de detalle, por lo que lo más óptimo es reducir su nivel de detalle.

En el siguiente nivel de detalle se reducen los vértices a 121 de ancho y largo, lo que da lugar a 14.641 vértices en total. Y por último, el tercer nivel de detalle contaría con 61

vértices de ancho y largo, teniendo en total de 3.721 vértices.

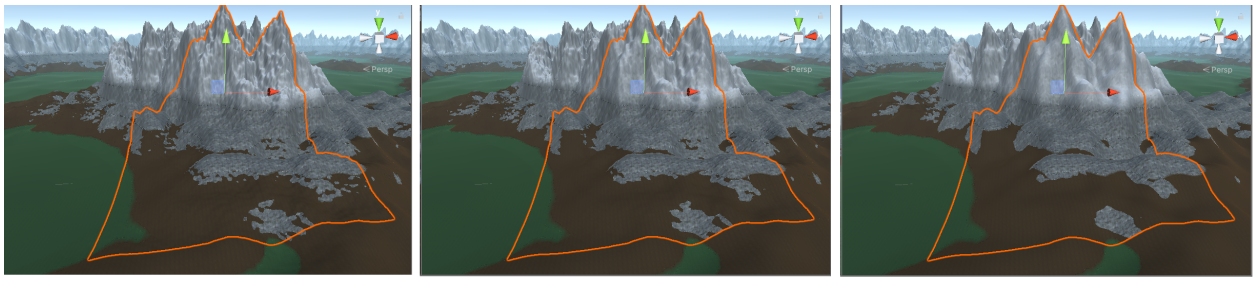


Figura 3.12: Diferentes LOD en el juego, de izquierda a derecha disminuye el detalle

Al observar la figura 3.12, podemos ver que desde cierta distancia, al ir cambiando el nivel de detalle no se aprecian grandes cambios para el jugador. Además, a la hora de generar el mapa se reduce drásticamente el tiempo de ejecución, pasando de tardar al rededor de 30 segundos para generar un terreno de 10 x 10 a solamente 3 segundos.

3.6. Detección de colisiones

A pesar de haber mejorar el rendimiento del algoritmo a la hora de generar el terreno, se creó otro problema a la hora de jugar. El problema consistía en que el juego se congelaba durante alrededor de un segundo cada vez que se cambiaba los niveles de detalle de las mallas. Esto se debe a que se actualiza el collider de las mallas, que es un componente contiene los datos geométricos de la malla para calcular las colisiones con otros objetos como el jugador. En este motor, la acción de modificar el collider de la malla para que coincida con el aspecto actual es una operación muy costosa y complicada. Por lo que para poder solucionar este problema se optó por actualizar solo el collider de la malla en el que se encuentra el jugador. De este modo, el rendimiento del juego se ve optimizado y el congelamiento que se produce es casi imperceptible.

Sin embargo, esta solución crearía otro problema en el caso de tener objetos dinámicos que se estén moviendo por el terreno. En algunos casos podrían verse objetos que estén flotando o un poco enterrados. Esto se debe, como se ha comentado anteriormente, a que el collider de la malla solo se actualiza cuando el jugador está en esa malla. Por lo tanto, en muchas ocasiones el collider no coincidirá con la parte visual de la malla hasta que el jugador se encuentre en ella. La solución radica en conseguir en optimizar el cálculo de los cambios en el nivel de detalle, que se podría lograr haciendo uso de un hilo que se encargue exclusivamente de estos cálculos.

3.7. Límites del mapa

Para que el jugador no puede escapar del espacio de juego generado y evitar así que caiga indefinidamente en el vacío, se deben generar uno bordes en los límites del terreno. Para poder lograrlo, lo que se ha hecho es crear un parámetro que nos permite indicar cual es el tamaño que queremos utilizar para el borde. En este caso, utilizaremos un valor entre 0 y 1, que nos lo usaremos como porcentaje. Por lo que si tenemos un mapa con un tamaño

de 240 metros y el porcentaje del borde es de 0.1, los 24 metros de cada borde corresponderán al borde del mapa, que iré aumentando en altura cuando más al límite se encuentre.

.

```
public float asignar_altura(float x,float z){

    float xp = x+xPosition;
    float zp = z+zPosition;

    float frec = .002f; //.005
    float mapa = Mathf.PerlinNoise((xp+map_seed)*frec,(zp+map_seed)*frec)*1f;

    //-----------------------------Bordes del mapa-------------------------------//
    if(xp <= 240*xMapSize*porcentaje_borde){
        float rango=(240*xMapSize*porcentaje_borde -xp)/(240*xMapSize*porcentaje_borde);
        mapa = Mathf.Lerp(mapa,0.85f,rango);
    }
    if(xp>=(240*xMapSize)-(240*xMapSize*porcentaje_borde)){
        float rango=(240*xMapSize-xp)/(240*xMapSize*porcentaje_borde);
        mapa = Mathf.Lerp(mapa,0.85f,1-rango);
    }
    if(zp <= 240*zMapSize*porcentaje_borde){
        float rango=(240*zMapSize*porcentaje_borde -zp)/(240*zMapSize*porcentaje_borde);
        mapa = Mathf.Lerp(mapa,0.85f,rango);
    }
    if(zp>=(240*zMapSize)-(240*zMapSize*porcentaje_borde)){
        float rango=(240*zMapSize-zp)/(240*zMapSize*porcentaje_borde);
        mapa = Mathf.Lerp(mapa,0.85f,1-rango);
    }

    //---------------------------------------------------------------------------//

    if(mapa<0.4){
        return flat(x,z);
    }

    if((mapa>=.4)&&(mapa<0.6)){
        float rango = (mapa -.4f)/(.6f-.4f);
        return Mathf.Lerp(flat(x,z),hill(x,z),rango);
    }

    if((mapa>=0.6)&&(mapa<0.7)){
        return hill(x,z);
    }
    if((mapa>=0.7)&&(mapa<0.85)){
        float rango = (mapa -.7f)/(.85f-.7f);
        return Mathf.Lerp(hill(x,z),mountain(x,z),rango);
    }
    if(mapa>=0.85){
        return mountain(x,z);
    }
    return 100f;
}
```

Figura 3.13: Código del cálculo de la altura de los bordes

El código presente en la figura 3.13 corresponde a la misma función vista en la figura 3.7, pero añadiendo el cálculo a los bordes del mapa. Lo que hace en esta función es ir aumentando el valor de la variable mapa a través de la interpolación lineal, de manera que se transforme a una zona de montaña.

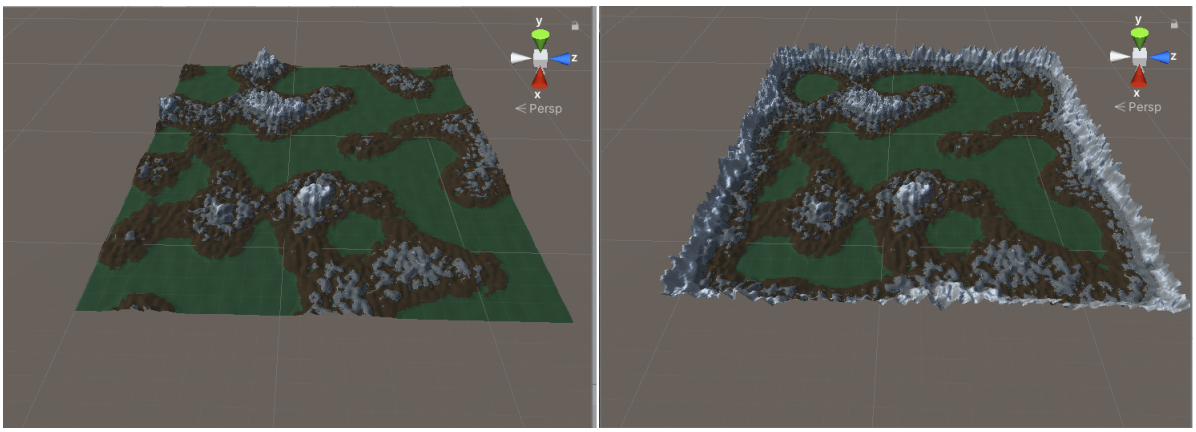


Figura 3.14: Comparativa del mapa sin borde y con borde

Como se puede apreciar en la figura 3.14, el borde creado tiene una altura suficiente que el jugador es incapaz de atravesar. Esto se debe a la pendiente que se crea con las montañas, supera el valor máximo de pendiente que el jugador puede subir.

3.8. Distribución de elementos

A partir de este momento solo quedaría repartir los elementos más comunes que puede encontrarse en cualquier landscape.

3.8.1. Árboles

Los primeros objetos que se han introducido en el terreno son los árboles.

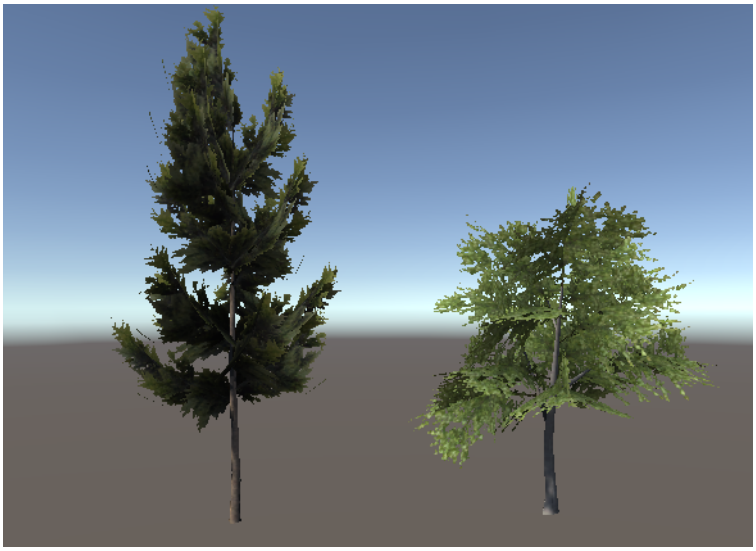


Figura 3.15: Diferentes tipos de árboles en el juego

Para este prototipo se dispone de dos tipos de árboles que se pueden ver en la figura 3.15. El árbol que se asemeja a un pino se colocará en la zona intermedia del mapa, es

decir, la que no es ni montañosa ni llana. Mientras que el árbol de la derecha, que es una especie de roble, se distribuirá por las zonas llanas.

El modo en el que se distribuyen estos árboles es bastante sencillo. Lo que se hace es generar por cada chunk 150 puntos aleatorios del mapa, y luego se comprueba la altura en la que se encuentra cada punto. Dependiendo de la altura del punto, se puede saber si pertenece a una zona u otra. En el caso de la zona intermedia, cada punto que se encuentra en esa zona aparecerá un pino, mientras que en la zona llana existe una probabilidad del 1/10 para que el árbol se pueda instanciar en el mapa. Esto se ha hecho de esta forma para que el aspecto de la zona intermedia sea más parecido a la de un bosque, mientras que la zona llana puede tener lugares más amplios al tener árboles más esparcidos.

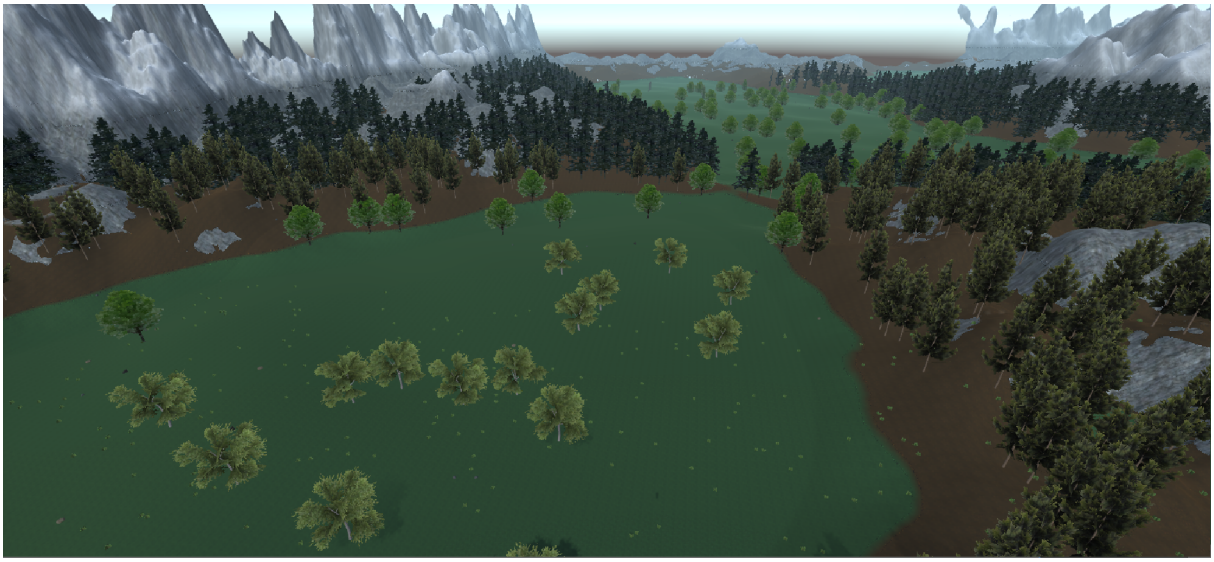


Figura 3.16: Distribución de árboles por el terreno

Como podemos apreciar en la figura 3.16, se consigue que en las zonas intermedias exista una mayor densidad de árboles, mientras que en la zona más llana los robles están más separados al tener menos cantidad.

Pero ahora surge otro problema debido a la gran cantidad de árboles que existen por el mapa. El problema se aprecia principalmente en los terrenos generados que tienen un gran tamaño, puesto que los árboles por el momento están siempre instanciados en el juego. Esto hace que los árboles que ni siquiera el jugador sea capaz de ver estén instanciados sin ningún propósito y utilizando un espacio de memoria que se podría destinar a otras cosas.

Para poder solucionar este problema, lo que se hace es que cada chunk almacena una lista con las posiciones de cada uno de sus árboles. Luego, cuando el jugador se encuentra a una distancia en la que es posible que el árbol sea visible lo instancia en esa posición. Y cuando el jugador se aleja a una determinada distancia del chunk, se realiza la acción contraria, por lo que se destruye todos los árboles instanciados, manteniendo la lista de posiciones por si es necesario volver a instanciarlos.

```
void generateTreePosition(){
    for(int i=0;i<150;i++){

        float x_random = Random.Range(0.0f, 240f-1f);
        float z_random = Random.Range(0.0f, 240f-1f);
        var height = asignar_altura(x_random,z_random);
        if((height<=10)&&((Random.Range(0,10)==1))){
            Vector3 position = new Vector3(xPosition + x_random, asignar_altura(x_random,z_random)-1f,zPosition + z_random);
            treePosition.Add(position);
        }
        else{
            if((height>10)&&(height<=40)){
                Vector3 position = new Vector3(xPosition + x_random, asignar_altura(x_random,z_random)-1f,zPosition + z_random);
                treePosition.Add(position);
            }

        }
    }

}
```

Figura 3.17: Código de la generación de las posiciones de los árboles

El código que aparece en la figura 3.17 se utiliza para calcular las posiciones donde se instanciarán los árboles en cada una de las mallas.

3.8.2. Vegetación

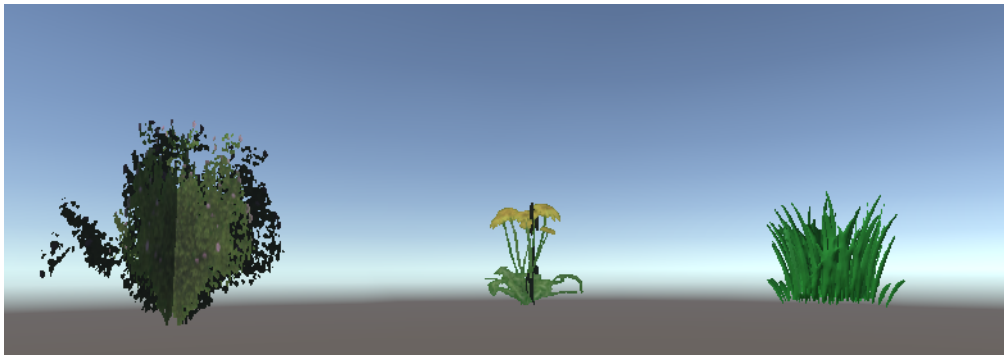


Figura 3.18: Distintos tipos de vegetación

En cuanto a la vegetación del terreno, podemos diferenciar 3 tipos diferentes: la hierba, las flores y los arbustos, que se pueden apreciar en la figura 3.18. La forma de distribuirse es muy similar a la de los árboles. En este caso, se generan hasta 5.000 puntos en los que puede existir vegetación, y las zonas en las que puede haber vegetación son las zonas intermedias en las zonas llanas. Luego, guardamos en una lista las posiciones de aquellos puntos en los que es posible que haya vegetación y el tipo al que corresponde.

Además, para conseguir que existan agrupaciones de flores, lo que se ha hecho es utilizar un mapa de alturas como el que se utilizó para determinar las diferentes zonas del terreno. De este modo, se consigue que predomine la vegetación de la hierba, mientras que los arbustos y las flores tienen menos probabilidades de aparecer.

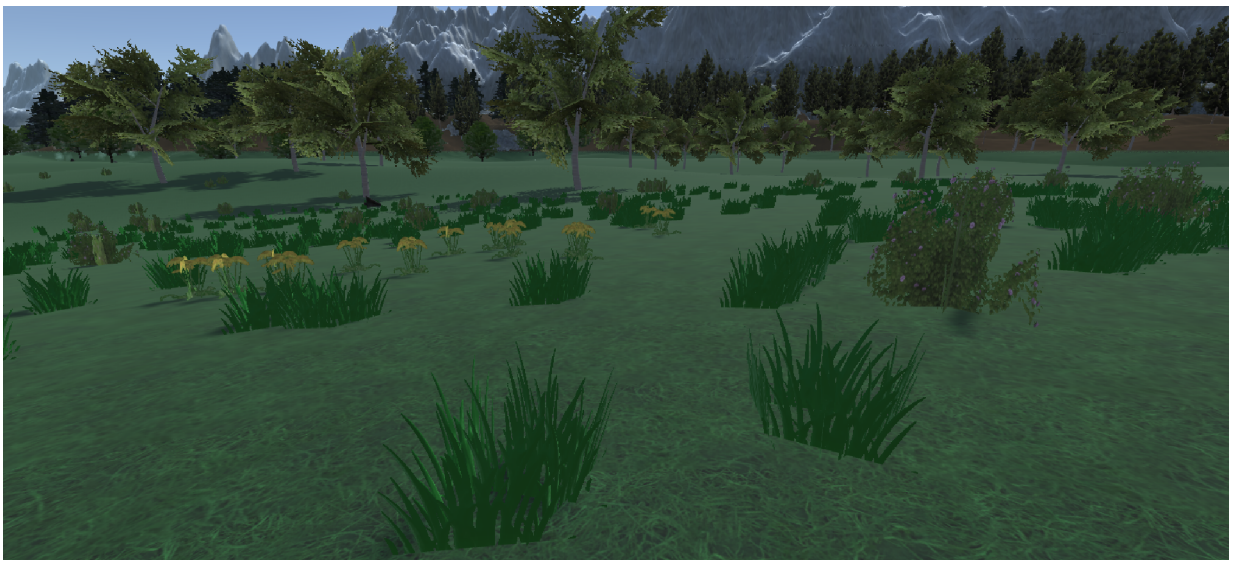


Figura 3.19: Vista del jugador de la vegetación

En la figura 3.19 se puede observar una imagen del juego en el que se aprecia que predomina la hierba como se ha mencionado anteriormente, mientras que aparecen un par de arbustos y una agrupación de flores.

Al igual que con los árboles, la inmensa cantidad de elementos de la vegetación hace que se reduzca mucho el rendimiento del juego. Por lo que solo se instanciarán los elementos que se encuentren cerca del jugador, mientras que lo que estén lejos esperarán a ser instanciados.

3.8.3. Rocas

Al igual que los árboles y la vegetación, el método de distribución de las rocas consiste en repartir aleatoriamente por el mapa estas rocas. Existen 3 tipos de rocas que podemos ver en la figura 3.20. Estas rocas solo varían en su aspecto y textura, y pueden estar en cualquiera de las diferentes zonas.

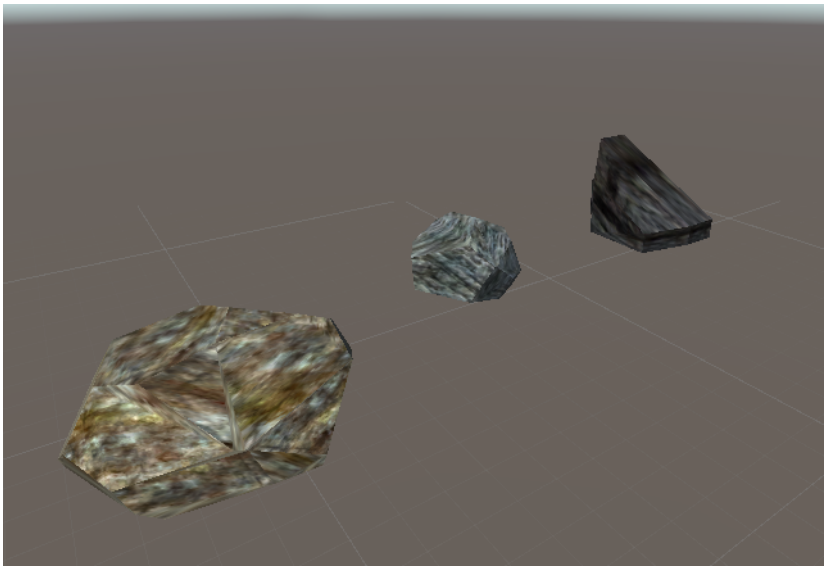


Figura 3.20: Distintos tipos de rocas

3.8.4. Partículas

El último tipo de elemento que se puede añadir a este landscape son los efectos de partículas. En este prototipo, los efectos de partículas son bastante sencillos e intentan simular la propagación de humo o polvo en suspensión. Otro efecto que se consigue gracias a las partículas es la imitación del movimiento de insectos como moscas por el mapa.

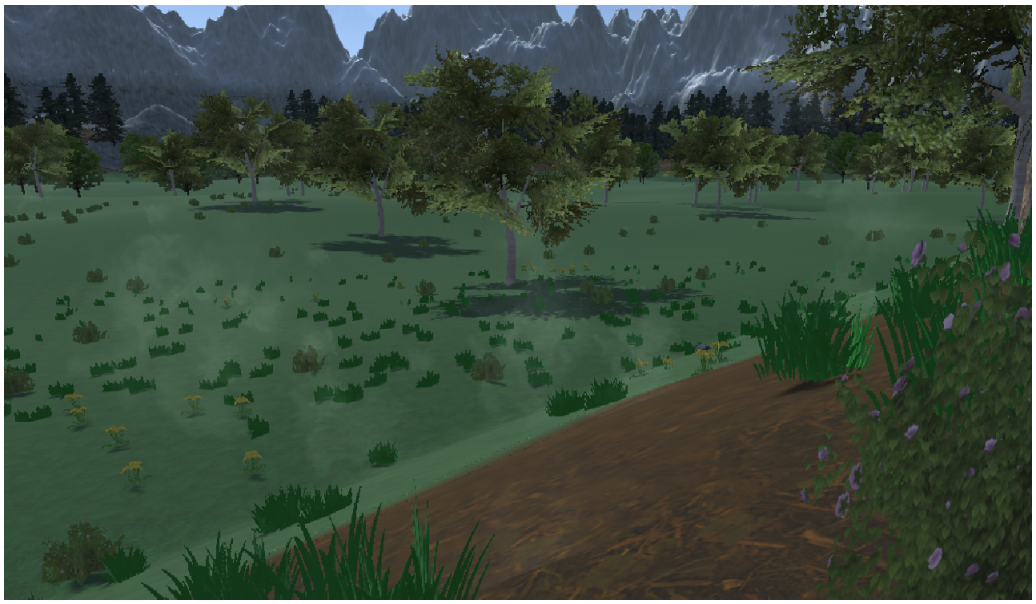


Figura 3.21: Efectos de partículas repartidas por el terreno

3.9. Funcionamiento del prototipo

Este proyecto se encuentra público en un repositorio de GitHub que se puede acceder a través del siguiente enlace: `https://github.com/alu0100812428/TFG_Unity/tree/master/Assets` .El proyecto está estructurado en 5 carpetas diferentes: Scripts, Shaders, Prefabs, resources y Scenes.

3.9.1. Scripts

Esta carpeta contiene los ficheros con las clases que hacen funcionar el juego. La clase "MeshBrain" es la que se encarga de ir creando los chunks del terreno, asignarle los parámetros que necesitan y colocarlos en el sitio correcto. Además, se encarga de indicar a los chunks que objetos son los que se van a ir distribuyendo por el entorno.

Estos chunks que se han ido creando, utilizan la clase "MeshGenerator". Esta clase es la que se encarga de crear su propia malla y darle forma, además de crear listas con las posiciones donde instanciar los árboles y la vegetación. También está pendiente de la posición del jugador en cada momento para poder ir cambiando el nivel de detalle de la malla e instanciar o destruir los objetos de las listas.

Luego tenemos dos clases asociadas al jugador, que son "PlayerMovement" y "MouseLook". La primera clase se encarga de establecer como se moverá el jugador y cuáles serán sus controles. La segunda clase se utiliza para poder mover la cámara del jugador con el ratón.

Por último, tenemos dos clases más sencillas que son "Billboard_tree" y "LightScript". La primera clase se utiliza cuando los árboles del juego se encuentran lejos del jugador y están utilizando el segundo nivel de detalle, que corresponde a una figura plana. El objetivo de esta clase es hacer que esta figura plana esté mirando siempre a la cámara del jugador, para que desde el punto de vista del jugador parezca un objeto de mayor detalle y no se aprecie que es un objeto plano. La segunda clase tiene como función ir rotando la luz del entorno para simular el ciclo día y noche. Actualmente no se está utilizando, debido a que es una función descartada.

3.9.2. Shaders

Aquí tenemos los dos shaders que se han utilizado en este proyecto . El shader con el nombre de "TextureShader"se utiliza para poder pintar una textura u otra en las mallas del terreno. Las texturas varían dependiendo de la altura a la que se encuentre cada píxel y la inclinación del terreno.

También tenemos el shader con el nombre de "GrassShader"que se utiliza para poder renderizar la parte posterior de la hierba. Si no se aplicase este shader a la hierba solo se podría ver una de las caras, siendo el otro lado invisible.

3.9.3. Prefabs

En la carpeta "Prefabs"se encuentran todos los objetos prefabricados que se utilizarán en el juego. Los objetos prefabricados son objetos que tienen parámetros que ya están establecidos. Los prefabricados que se usan en el proyecto son los relacionados con los árboles, rocas, vegetación y los efectos de partículas. Luego tenemos prefabricados más complejos como el "MeshBrain", que corresponde a un objeto que utiliza la clase "MeshBrainz que ya tiene asociados todos los prefabricados mencionados anteriormente y otros parámetros definidos como el tamaño y el borde total del mapa. Todos estos parámetros aparecen representados en la figura 3.22.

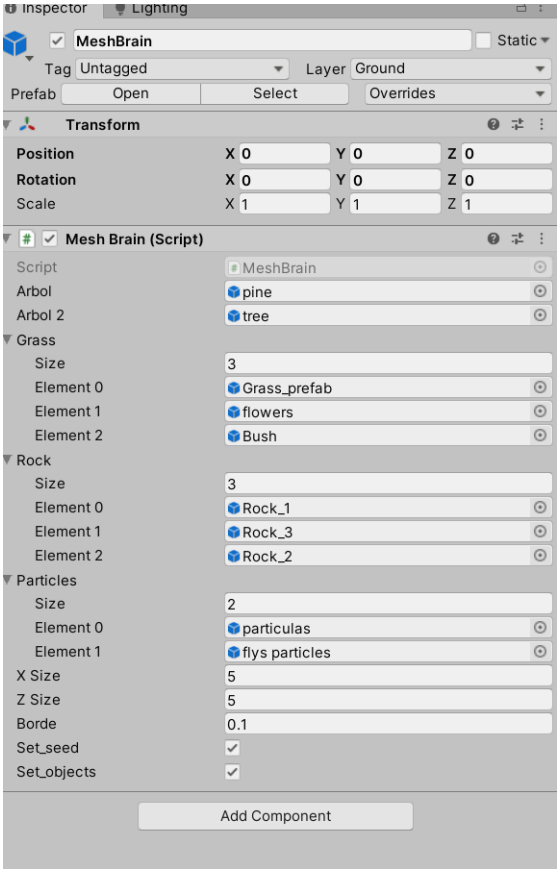


Figura 3.22: Parámetros del MeshBrain

Por lo tanto, para que el juego pudiera funcionar correctamente, habría que colocar en la escena el prefabricado del jugador y de la luz direccional, y el "MeshBrain". Este juego al ser jugado en primera persona no se ha trabajado en el modelado del jugador, que es el objeto cilíndrico que se puede ver la figura 3.23. Con estos 3 prefabricados se consigue generar el terreno al completo y controlar al jugador.

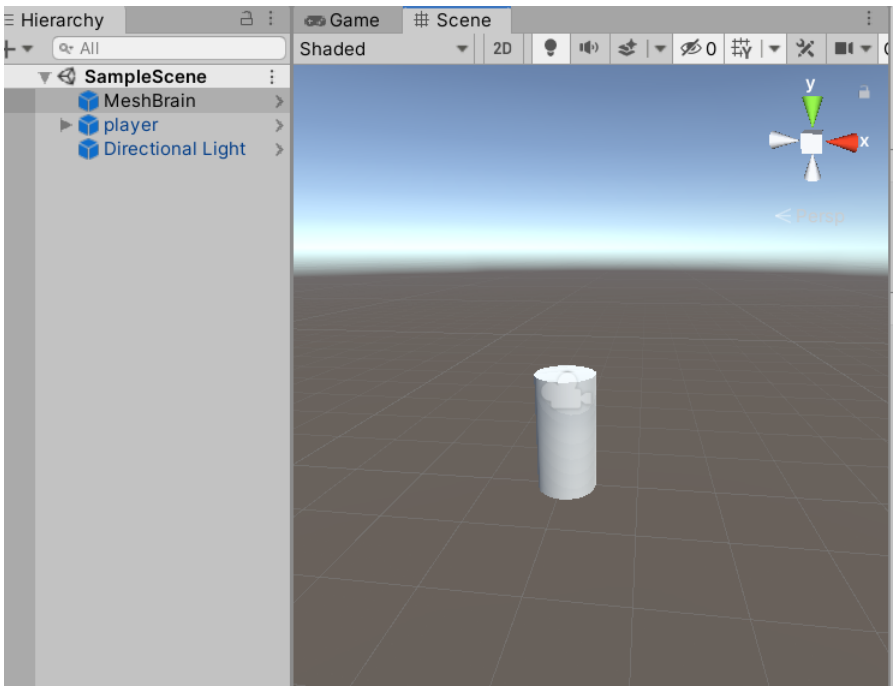


Figura 3.23: Escena antes de la generación del entorno

Al comenzar la ejecución del juego, se generan los chunks que contienen las mallas del terreno, y se crean múltiples copias de los prefabricados mencionados anteriormente. En la figura 3.24 tenemos los prefabricados que aparecían antes de la ejecución del juego y todos los nuevos objetos instanciados.

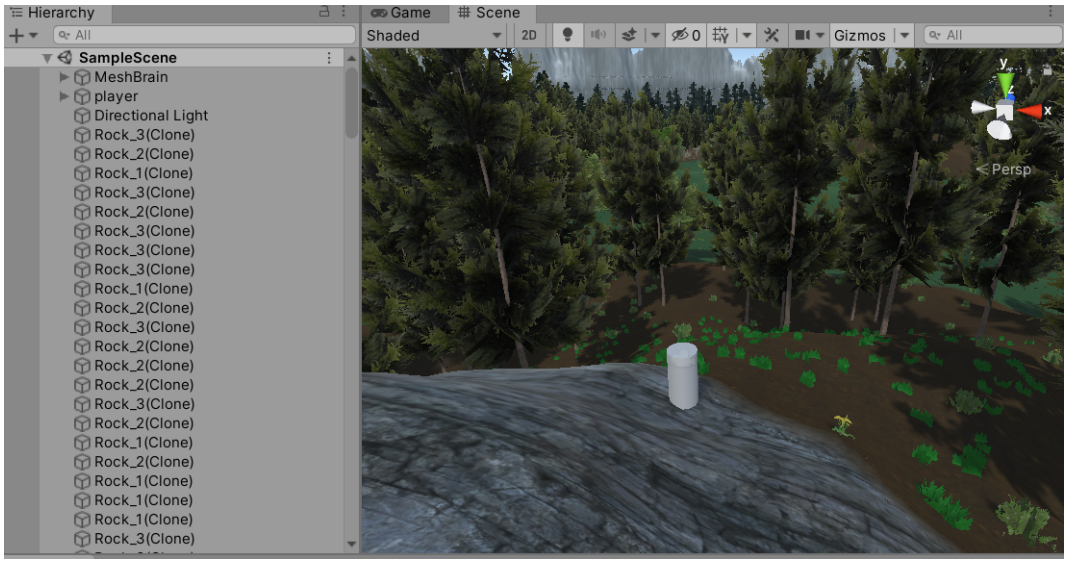


Figura 3.24: Escena después de la generación del entorno

3.9.4. Resources

Aquí se guardan todos los materiales y texturas que se han utilizado en los prefabricados y en el terreno. Además, se guardan los modelos originales de todos los objetos

prefabricados. Actualmente existen materiales y modelos de objetos que no se han usado en ningún momento, pero que se han guardado por si se necesitan en el futuro.

3.9.5. Scenes

En esta carpeta se guardan las escenas o niveles de nuestro prototipo. Actualmente solo existe una escena que está compuesta por los prefabricados "MeshBrain", el jugador y la luz direccional. De forma que si se pone en funcionamiento esta escena, nos generaría procedimentalmente un landscape.

Capítulo 4

Técnicas para optimizar el rendimiento

Ente capítulo se hablará acerca de las técnicas que se han empleado para poder optimizar el rendimiento del juego.

4.1. Nivel de detalle

Cuando tenemos tantos objetos en nuestro videojuego, en ocasiones, la geometría que presentan puede ser compleja. En estos casos, lo que ocurre es que nuestro ordenador quizás no puede estar procesando tantos triángulos y hace que el rendimiento del juego disminuya. Una manera de solucionar este problema es ir creando por cada uno de los objetos el mismo objeto pero ir reduciendo cada vez más el número de polígonos que lo componen, de modo que cuando más lejos se encuentre el jugador de dicho objeto, se mostrará el modelado que menos detalle posee. Es por ello que en la mayoría de los objetos que componen nuestro videojuego se utilicen distintos niveles de detalle.

El nivel de detalle o LOD, por su abreviación del inglés "Level of Detail", es el nivel con el que se está renderizando un objeto. Por lo tanto, en Unity en el nivel más bajo, que suele ser el nivel 0, el objeto en su forma original que es la que posee mayor detalle. Al contrario, cuanto mayor el nivel, el objeto tendrá muchos menos polígonos.

En cuanto la aplicación de esta técnica en este prototipo, se ha utilizado para poder reducir en gran medida la geometría de la superficie del juego, como se ha explicado en el punto 3.5. Gracias a la aplicación de un algoritmo sobre cada una de la mallas, se ha conseguido que aquellas que se encuentren distantes del jugador, vean reducido el número de triángulos que lo forman de manera muy significativa.

En el caso de los demás elementos del juego, se ha utilizado un componente de Unity llamado LOD group. Este componente nos facilita la organización de los niveles de detalle en función de la distancia a la que se encuentre el jugador.

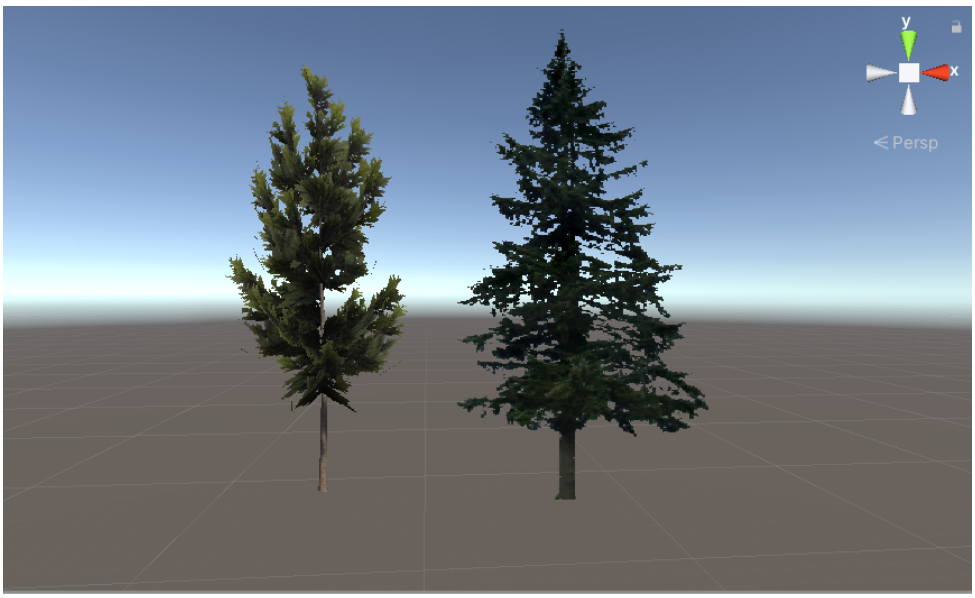


Figura 4.1: Comparativa de los niveles de detalle del pino

En este ejemplo, tenemos a la izquierda un pino que cuenta con un total de 471 triángulos, que viene siendo el nivel 0 en el LOD. Mientras que a la derecha tenemos un pino que cuenta sólo con 2 triángulos, que es el nivel 1 en el LOD. Por último, tenemos el nivel en el que directamente no se renderiza el objeto, por lo que se suele aplicar a distancias muy lejanas.

El pino que aparece en el nivel 1 se ha conseguido que cuente con tan pocos triángulos debido a que se ha plasmado una imagen de un pino sobre una objeto plano. De este modo, si lo observamos de frente, se vería bien. Pero si lo observamos desde cualquier otro punto de vista, podríamos ver que es un objeto plano.

Para solucionar esto, lo que se hace es aplicarle un script a este objeto plano para que siempre esté mirando a la cámara. De este modo, en cualquier punto en el que se encuentre el jugador, estos objetos están siempre frente a él y no de lado.

Hay que destacar también que en este prototipo se ha usado la imagen de un pino que difiere mucho del pino que tiene un mayor nivel de detalle. Por lo que al hacer un cambio en el nivel de detalle, el jugador puede percatarse a simple vista de esta diferencia. Para poder solucionarlo, se podría haber utilizado una imagen exacta del modelo original e incluso añadir modelos intermedios que vayan disminuyendo el número de polígonos que lo forman. Es por ello que en los juegos es difícil poder ir reduciendo el nivel de detalle de un objeto sin que el jugador aprecie grandes cambios y sin sobrecargar el juego con tantas variaciones de un mismo modelo.

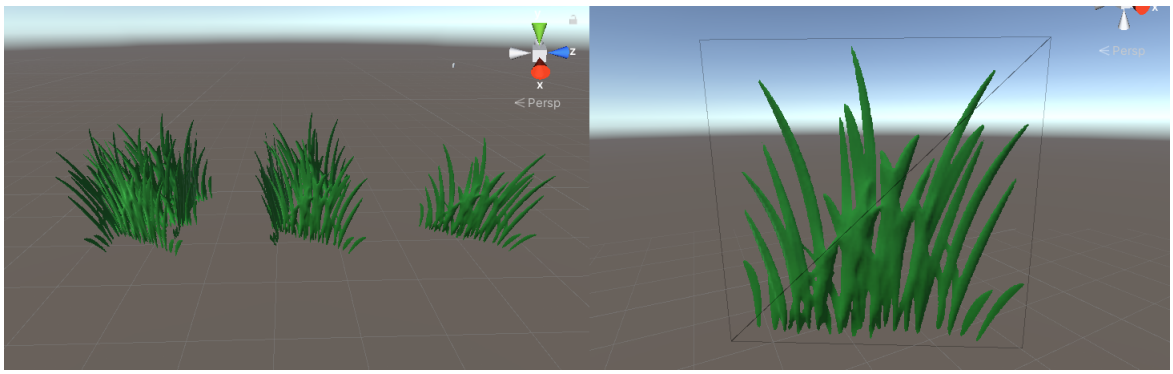


Figura 4.2: Comparativa de los niveles de detalle de la hierba

También se utilizan distinto nivel de detalle tanto para la hierba como para las flores y arbustos que aparecen por el terreno. Como podemos ver en la figura 4.2, cada parte que compone la hierba está formado por una imagen plasmada sobre una figura plana de 2 triángulos. En cuanto al nivel de detalle se observa que de izquierda a derecha va disminuyendo la densidad de la hierba. Por lo que en el primero de los casos, que es el de mayor detalle y que más cercano se encuentra al jugador, se muestran 5 planos de hierba, mientras que el segundo está compuesto por 3. Luego tenemos el tercer caso en el que solamente se renderiza un plano de hierba. Y por último, existe el caso en el que no se muestra nada de hierba.

4.2. GPU instancing

El GPU instancing es una técnica que se usa para que el motor pueda dibujar múltiples copias de una mismo malla a la vez, reduciendo el número de llamadas para dibujar, conocidas como draw calls. A la hora de dibujar cada uno de los objetos en pantalla, el motor debe mandar un draw call al API de gráficos como pueden ser OpenGL o Direct3D. Estos draw calls son operaciones costosas, lo que hace que se sobrecargue de trabajo a la CPU mientras que la GPU está con poca utilización.

Gracias al GPU instancing, podemos mejorar el rendimientos de nuestro juego a la hora de renderizar gran multitud de elementos que tienden a repetiraw, como son los árboles, la hierba, rocas, etc. Como resultado, en casos en los que tengamos al rededor de 2000 elementos en pantalla, el número de draw calls pasará estar sobre los 2000 a ser simplemente 20 o 30. Para poder lograrlo, en los objetos que queramos aplicar esta técnica, debemos activar una opción en los materiales de los objetos que se llama "Enable GPU Instancing".

4.3. Camera Culling

El ”Camera Culling” es una técnica que se encuentra en casi todos los juegos, y que nos permite reducir también el número de objetos que se están renderizando. La forma en la que se consigue esto es no renderizando aquellos elementos que el jugado no es capaz de ver. Por lo tanto, todos aquellos objetos que estén a la espalda del jugador, no

se renderizarán. Lo mismo sucede con los objetos que estén ocultos por objetos más grandes.

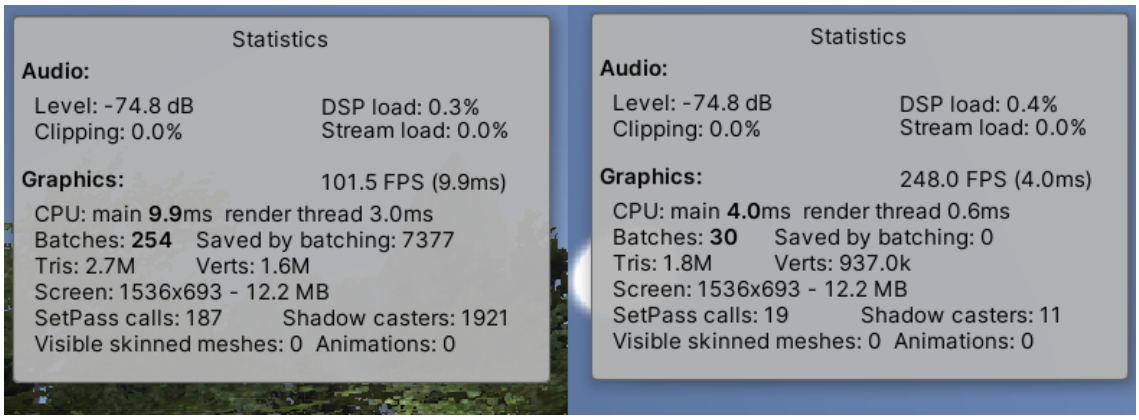


Figura 4.3: Comparación de las estadísticas en el juego

Para pode comprobar el resultado de la implementación de esta técnica, se han realizado capturas de las estadísticas durante la ejecución del juego. En la figura 4.3, podemos ver que en la imagen izquierda que corresponde al jugador observando el entorno en una situación bastante habitual en un videojuego, los FPS se sitúan entorno a los 100. Mientras que en la imagen de la derecha, que muestra el caso en el que el jugador mira directamente al cielo y no aparece ningún objeto, los FPS ascienden hasta los 248.

Capítulo 5

Conclusiones y líneas futuras

A lo largo del desarrollo de este videojuego, se ha podido comprobar que la creación de un videojuego es bastante compleja. Para empezar, surgió el problema de la limitación del número de vértices que podía tener una malla, que se solventó dividiendo el terreno en sub-mallas. También había que buscar la manera de poder asignar alturas a estos vértices y que el aspecto fuese natural. Es aquí cuando se empieza a aplicar el ruido de Perlin para crear mapas del altura para dividir el terreno en diferentes zonas y también para calcular la altura de los vértices. Esto se consigue gracias a la combinación de distintos niveles de generación de Perlin para cada una de las zonas. Sin embargo, surgió el problema de que la transición entre estas zonas era muy abrupta. Por lo que para conseguir que la transición fuera suave se recurrió al uso de la interpolación lineal, técnica que se emplearía también para conseguir mezclar las texturas del terreno.

A la hora de generar el landscape a una mayor escala, se pudo comprobar que los tiempo de ejecución iban empeorando, y además el rendimiento se veía reducido debido a la inmensa cantidad de triángulos que debían renderizarse por cada frame. Para poder solucionar estos dos problemas, se optó por ir cambiando el nivel de detalle de las mallas en función de la ubicación del jugador. Por lo que aquellas mallas que estuvieran cerca del jugador tendrían muchos más triángulos mientras que las que estuvieran lejos tendrían muchos menos. Esto mejoró considerablemente el rendimiento a la hora de jugar a la vez que se redujo el tiempo de ejecución, puesto que solo un par de mallas de las que se generaban tenían un alto nivel de detalle. No obstante, esta mejora trajo consigo otro problema relacionado con el collider de estas mallas, que se actualizaba cada vez que se cambiaba el nivel de detalle. En Unity, actualizar el collider es una operación muy costosa y si se hace en una malla con muchos vértices empeora la situación, lo que hace que la ejecución del juego se congele si se produce muchas de estas actualizaciones en un mismo instante. Por lo que la solución que se aplicó fue que solo se actualice el collider de la malla en la que se encuentra el jugador. Gracias a ello se consiguió mejorar el rendimiento al cambiar el nivel de detalle de las mallas, aunque todavía existe un pequeño congelamiento al realizar esta acción.

Más adelante, se añadieron diferentes tipos de objetos en el entorno como los árboles y la hierba. Al principio, al ir añadiendo estos objetos el rendimiento no se veía afectado. Conforme el se iban aumentando la cantidad de árboles y de hierba se pudo observar que el rendimiento bajaba demasiado y en algunos casos podía conseguir que el editor dejara de responder. Por lo que se tuvo que tomar dos medidas diferentes para solucionar este problema. En primer lugar, se tuvieron que crear diferentes niveles de detalles para

estos objetos, para que cuando el jugador este en una posición lejana no tuviera que esforzarse tanto el motor en representar estos elementos. Por otro lado, utilizando el GPU instancing se consiguió reducir el número de veces que la GPU debe renderizar un objeto que está repetido varias veces. Sin embargo, conforme se iba aumentando la escala del terreno generado se iban añadiendo más elementos. Esto hacía que cada uno de estos objetos consumieran un espacio en la memoria, por lo que al tener demasiado objetos el juego dejase de responder. Para solventar este problema se optó por no tener todos estos objetos colocados en el mapa en todo momento, sino que se vayan instanciando según el jugador estuviera a una distancia que sí se necesitase dicho objeto. Gracias a ello, se pudo lograr que el rendimiento del juego se mantuviese en un estado óptimo sin depender de lo grande que fuese el terreno.

Como en este prototipo todavía quedan cosas por mejorar y añadir, se proponen como líneas futuras los siguientes puntos:

- Mejorar la transición de las texturas del terreno.
- Hacer el cambio de nivel de detalle de las mallas más eficiente, puesto que el juego se congela durante un breve periodo de tiempo.
- Añadir más variedad de árboles, vegetación o rocas.
- Añadir áreas y caminos.
- Estudiar el comportamiento de objetos dinámicos en el terreno.
- Mejorar la colocación de las rocas para que coincidan con la pendiente del terreno.

Capítulo 6

Summary and Conclusions

Throughout the development of this video game, it has been verified that the creation of a video game is quite complex. To begin with, the problem of limiting the number of vertices that a mesh could have arose, which was solved by dividing the terrain into sub-meshes. It was also necessary to find a way to assign heights to these vertices and that the appearance was natural. This is when Perlin noise begins to be applied to create height maps to divide the terrain into different areas and also to calculate the height of the vertices. This is achieved thanks to the combination of different Perlin generation levels for each of the zones. However, the problem that the transition between these areas was very abrupt arose. So to achieve a smooth transition, the use of linear interpolation was used, a technique that would also be used to mix the textures of the terrain.

When generating the landscape on a larger scale, it was possible to verify that the execution time was getting worse, and also the performance was reduced due to the immense number of triangles that had to be rendered for each frame. In order to solve these two problems, it was decided to change the level of detail of the meshes depending on the location of the player. So those meshes that were close to the player would have more triangles while those that were far away would have much less. This greatly improved gaming performance while reducing runtime, as only a couple of meshes being generated had a high level of detail. However, this improvement brought with it another problem related to the collider of these meshes, which were updated every time the level of detail was changed. In Unity, updating the collider is a very expensive operation and if it is done on a mesh with many vertices, it worsens the situation, which causes the game execution to freeze if many of these updates occur at the same time. So the solution that was applied was that of all the collider of the meshes, only the one of the mesh in which the player is located is updated. Thanks to this, it was possible to improve performance by changing the level of detail of the meshes, although there is still a small freeze when doing this.

Later, different types of objects were added to the environment such as trees and grass. At first, as these objects were added, performance was not affected. As the number of trees and grass increased, it was observed that the performance decreased too much and in some cases could get the editor to stop responding. Therefore, two different soluciones had to be taken to solve this problem. First of all, different levels of detail had to be created for these objects, so that when the player is in a distant position, the engine does not have to try so hard to represent these elements. On the other hand, using the GPU instancing it was possible to reduce the number of times that the GPU must render an object that is repeated several times. However, as the scale of the generated

terrain was increased, more elements were added. This caused each of these objects to consume a space in the memory, so when having too many objects the game stopped responding. To solve this problem, it was decided not to have all these objects placed on the map at all times, but to be instantiated according to the player being at a distance that said object was needed. Thanks to this, it was possible to ensure that the performance of the game was kept in an optimal state without depending on how big the terrain was.

Capítulo 7

Presupuesto

En este capítulo se trata el posible presupuesto requerido para la realización de este proyecto, teniendo en cuenta que el videojuego cumple con todos los objetivos propuestos, funciona sin fallas y un rendimiento óptimo.

Teniendo en cuenta que el número de horas aproximado para la realización completa de este videojuego puede alcanzar las 600 horas, y que cada una de estas horas desarrollo costase 15€, el coste total destinado al desarrollo asciende hasta los 9.000€.

Además, hay que tener en cuenta que puede necesitarse la inclusión en el juego de assets que sean de pago. Por lo que el coste total de todos los assets podría rondar los 200€. También puede surgir la necesidad de adquirir un plan más avanzado de Unity que ofrece más funcionalidades y herramientas y que se paga mes a mes. Por lo que el coste que supondría la suscripción de un par de meses de estos planes alcanzaría los 300€. Por último, se necesita disponer de un ordenador de gama media y que disponga una tarjeta gráfica.

Tipos	Coste
Tiempo de desarrollo	9000€
Assets de pago	200€
Suscripción de un plan	300€
Ordenador de gama media	700€

Tabla 7.1: Presupuesto del desarrollo del prototipo

Por lo tanto, al sumar todos estos costes, el presupuesto podría llegar a rondar los 10.200€.

Bibliografía

[1] Archer, T. (2011). Procedurally generating terrain. https://pdfs.semanticscholar.org/d60c/202d166b51338ec8b684f6954ebc506b2f3a.pdf.

[2] Bates, B. (2003). C# as a first language: A comparison with c++. https://dl.acm.org/doi/pdf/10.5555/948835.948843.

[3] Bell, I. and Braben, D. (1984). Elite. Acornsoft.

[4] Games, H. (2016). No man's sky. Hello Games.

[5] Microsoft (2015). Introducción al lenguaje c# y .net framework. https://docs.microsoft.com/es-es/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework.

[6] Persson, M. (2011). Minecraft. Mojang.

[7] Ruelas, U. (2017). ¿qué es un motor de videojuegos (game engine)? https://codingornot.com/que-es-un-motor-de-videojuegos-game-engine.

[8] Toy, M. and Witchman, G. (1980). Rogue.

[9] UA, M. M. (2018). El motor unity. https://mastermoviles.gitbook.io/graficos-y-multimedia/el-motor-unity.

[10] Unity (2020). Unity manual: Writing shaders. https://docs.unity3d.com/Manual/ShadersOverview.html.

[11] Yu, D. and Hull, A. (2008). Spelunky. Mossmouth, LLC - Microsoft Studios.