

Algoritmos para generación procedural de paisajes

Trabajo de fin de grado (TFG)

Autor:

Paduraru, Cosmin Marian

Tutor:

Guil Asensio, Francisco de Asis

Convocatoria de Junio del curso 2021/22

22 de junio de 2022



Índice

1. Resumen	6
2. Extended Abstract	7
3. Introducción	11
4. Estado del arte	13
4.1. Estado del arte	14
4.2. Propuesta	17
5. Análisis de objetivos y metodología	17
5.1. Objetivos	18
5.2. Metodología	18
6. Diseño y resolución del trabajo realizado	20
6.1. Editor de clases	20
6.1.1. Bases del editor	21
6.1.2. Casillas marcables	21
6.1.3. Botones	21
6.1.4. Sliders	22
6.1.5. Modificadores de la escena	22
6.1.6. Otros aspectos	22
6.2. Primeros pasos	23
6.3. Generación de objetos básicos	23
6.4. Primeros pasos del generador de terreno	27
6.4.1. Generador de plano	27

6.5.	Generar malla editable	28
6.6.	Modificador del terreno	29
6.6.1.	Subdivisión de malla	29
6.7.	Bordes marginales	30
6.8.	Recuperación de mallas anteriores	31
6.9.	Ruido Perlin	32
6.9.1.	Ruido general	32
6.9.2.	Ruido por altura	35
6.9.3.	Ruido por zonas	35
6.10.	Octavas	38
6.11.	Suavizado	39
6.12.	Herramienta de empuje	40
6.12.1.	Empuje con dirección	40
6.12.2.	Empuje sin dirección	41
6.12.3.	Diferencias entre los dos métodos de empuje	42
6.13.	Texturizado y coloreado	44
6.13.1.	Shaders	45
6.13.2.	Texturizado	46
6.14.	Añadir objetos	46
6.14.1.	Añadir objetos de forma totalmente aleatoria	47
6.14.2.	Añadir objetos con ruido Perlin	47
6.14.3.	Modificadores al añadir los objetos	49
6.15.	Características adicionales	50
6.15.1.	Agua	50
6.15.2.	Nubes	51

6.15.3. Controlador en primera persona	51
6.15.4. Ciclo de día-noche, iluminación, postprocesado y cielo	52
6.16. Paisajes procedurales finales	52
7. Conclusiones y vías futuras	54
7.1. Conclusiones	54
7.2. Vías futuras	54

Índice de figuras

1.	Ejemplo de generación de terreno con Gaia Pro	15
2.	Ejemplo de generación de terreno con Terrain Composer 2	16
3.	Ejemplo de generación de terreno con TerrWorld	16
4.	Ejemplo de generación de terreno con Unity Terrain	17
5.	Dos triángulos con distintas orientaciones. El primero sería visible desde el punto de vista de la cámara y el otro no. Imagen obtenida de https://learnopengl.com/Advanced-OpenGL/Face-culling	24
6.	Imagen de los objetos básicos, de izquierda a derecha: Cubo, Cuadrado, Triángulo .	25
7.	Perpendiculares en caras (azul oscuro) y vértices (azul claro) de un objeto en Blender [3]	25
8.	Un mismo objeto con iluminación por caras o por vértices	26
9.	Imagen de cubo compartiendo vértices y sin compartir vértices	26
10.	Malla de lado $x = 3$ y $z = 2$	27
11.	Malla con dos vértices seleccionados	28
12.	Cuadrado y su subdivisión	29
13.	Malla subdividida una vez	30
14.	Misma malla subdividida varias veces	30
15.	Malla con altura elevada	31
16.	Malla con vértices marginales a cero	31
17.	Botones de recuperación de mallas	32
18.	Malla sin aplicar Perlin	33
19.	Malla con subdivisión y ruido Perlin	33
20.	Interfaz ruido Perlin	34
21.	Línea de altura	35
22.	Ruido dentro de caja	36

23.	Idea vértices dentro de otras figuras	37
24.	Malla en la que se aplican dos capas de ruido Perlin	38
25.	Valores de octavas	38
26.	Área suavizada	40
27.	Herramienta de empuje con dirección	41
28.	Herramienta de empuje sin dirección	42
29.	Diferencia lagos	42
30.	Diferencia montañas	43
31.	Diferencia cuevas	43
32.	Proceso de asignación de una textura a una malla. Imagen obtenida de https://en.wikipedia.org/wiki/UV_mapping	44
33.	Textura seamless aplicada una vez (izquierda) o varias veces (derecha) para cambiar el tamaño de los ladrillos	44
34.	Shader texturizado	45
35.	Ejemplo de uso de gradiente	46
36.	Árboles añadidos de forma aleatoria	47
37.	Explicación añadir objetos con ruido Perlin	48
38.	Árboles añadidos con Perlin Noise	49
39.	Lago generado con shader de agua	50
40.	Doble capa de nubes	51
41.	Controlador primera persona	52
42.	Escena con iluminación mejorada y postprocesado	52
43.	Primer escenario final	53
44.	Segundo escenario final	53

1. Resumen

Los videojuegos pueden ser considerados como una mezcla entre técnica y arte. Por ello, la creación de un videojuego conlleva la unión de muchas disciplinas, desde la programación hasta el diseño, pasando por las matemáticas y la física. Gracias a este arte podemos llegar a sitios que no hemos visto o vernos en situaciones en las que no podemos estar en el mundo real. La principal motivación para mi trabajo de fin de grado es poder simular estos lugares y situaciones que se mencionaban con anterioridad. Sin embargo, conforme los juegos han ido ganando en complejidad, la generación de estos mundos virtuales se ha ido haciendo cada vez más laboriosa. Para ayudar en esta tarea, cada vez se usan más técnicas de generación procedural, esto es, algoritmos que permiten generar de modo automático (o semi automático) partes del juego. Esta generación suele ser además aleatoria, lo que permite como ventaja adicional, que cada vez que se juegue la experiencia del jugador sea ligeramente distinta.

Por tanto, la generación procedural consiste en generar cualquier parte dentro de un videojuego de forma aleatoria. Esta generación debe tener sentido, es decir, debe ser coherente con los demás aspectos del juego. Uno ejemplo muy clásico de generación procedural fue Spelunky [6], donde las minas se generaban con esta técnica. Un ejemplo de generación procedural más actual pueden ser las minas del juego Stardew Valley, donde cada nivel se genera de forma procedural.[10] Por último tenemos el videojuego No Man's Sky, en el que tanto las criaturas, como los planetas y los biomas, se usa generación procedural [19].

Uno de las facetas en que más se ha usado la generación procedural es como ayuda en la creación de paisajes. Los paisajes generados constituyen un primer paso, a los que posteriormente se pueden aplicar ciertas modificaciones para personalizarlos. De esta forma se consiguen superficies únicas y personalizadas de forma relativamente sencilla (una vez se tiene un buen manejo de la herramienta).

En este trabajo nos vamos a centrar en la creación de un generador de terrenos procedurales y analizaremos los algoritmos utilizados en cada una de las herramientas que conlleva dicho generador. Este proceso se ha realizado mediante Unity [8], que es uno de los motores de videojuegos más usados en la industria actual (por ejemplo, es el más usado en la plataforma Steam [7]). Este motor es el que se enseña en la mención de computación en la asignatura de Fundamentos Computacionales de los Videojuegos. Aunque hay que destacar que Unity no tiene muchas herramientas de manejo de mallas como puede ser Blender [3].

En esta memoria podemos distinguir varios apartados que corresponden con el trabajo realizado en orden cronológico. En primer lugar, la sección dedicada a la introducción en la que se pone en contexto todo el trabajo. A continuación una segunda sección dedicada al **estado del arte y análisis** donde se estudia cuáles son los generadores procedurales actuales y las características que tiene que tener un generador procedural. Una vez vistas todas las características deseadas, se dividen el problema de crear un generador procedural de paisajes en varios subproblemas y se analiza cada una de ellos. A continuación se pasa al **diseño y resolución**, donde se implementarán dichos algoritmos (generador de mallas, modificador de terrenos, texturizado...). Por último se verá la **conclusión y las posibles vías futuras**.

2. Extended Abstract

Video games are widely considered as an art. The creation of a video game involves the union of many disciplines, from programming to design, through mathematics and physics, as I have been able to learn since I made my first video game in first year with the subject of Programming Technology to Computer Game Fundamentals and even the whole part of artificial intelligence that is seen in AI for the Development of Games. We often use these to escape from reality together with another means of recreation such as sport or television and to take a break from the everyday world. People expect to climb mountains, dive under the ocean, go through grasslands, discover fantastic animals, explore planets, see what's inside volcanoes, etc. This is the reason for my final thesis, to be able to simulate unheard places. For this porpuse I will be using procedural generation, which is used in enviroments beyond the creation of video games such as movies, clothing designs, board games, etc. Therefore, procedural generation is not exclusive to the generation of landscapes.

Procedural generation is a random but controlled process of creation, the best and easiest way to understand this is through the example of Minecraft [5]. A new world originates, but it follows common sense principles: where there has to be water there is water, there are no mountains floating, and you won't see biomes blended. The formal definition of procedural generation is the following: Method of creating data with algorithms rather than manually. The term procedural refers to the process that computes a particular function. An example of procedural are fractals.

One of the first times procedural generation was used was in 1978 [2], when it was used to create dungeons automatically. Procedural generation allowed dungeons to be created with algorithms and therefore the memory limit was not a problem, which it would have been if we had tried to create all the levels and save them, since that in that time the memory was much lower than we are used right now.

Procedural generation helps us in an expensive manner in the creation of landscapes since it is a very powerful tool to generate them quickly and apply certain modifications depending on the needs and preferences of the user.

Procedural generation is a combination of different tools, a few of which will be discused in this paper are the creation of meshes, applying noise (mainly Perlin noise), placing objects by height and noise, generating caves, crevices, lakes, smoothing terrain, applying textures, coloring terrain, waves, clouds, among others.

In this project we are going to see how the creation of a procedural terrain generator and the algorithms used in each of the tools involved in this generator has been carried out. Many of these implementations that were mentioned before are based on the tools offered by Unity. Implementing a game engine could have been done from scratch, but the mere creation of such engine would be material for another final degree project.

As mentioned before, this project has been carried out using Unity, since it is a videogame engine used in the current industry, together with Unreal Engine. It is the engine that is thaught at the computer science degree at the University of Murcia, so it is the one I am most familiar with and therefore the one I am most comfortable with.

However, it should be clear that I am no expert, since knowing how to handle Unity completely is a process of years of learning, as it is a very large engine in which there are usually entire teams behind creating a video game. An example of this games would be Hollow Knight [14] or Escape from Tarkov [18].

This paper contains the whole process carried out for the creation of this generator, not only how it has been implemented but also the creation process: from understanding the basics of how the meshes used for the terrain work to the generation of clouds, through how noise is applied and how objects are placed automatically.

It is important to note that before developing the code, I had to carry out a requirement analysis and a general diagram of what was to be created. During the project, modifications were made to this diagram, but the main structure was maintained. As there is no company or client, the requirements were decided by me and my tutor. We met once a week (usually on Tuesdays) to see what progress was being made from week to week and to see what could be improved and what could be implemented. Sometimes we came up with interesting ideas that were implemented. We communicated by email to make improvements to the project.

Furthermore, we were looking at all the procedural generators on the market at the time of writing this report. The most notable ones are the terrain generator implemented in Unity and a couple of others that are available in the asset store for a fee (in addition to those in other game engines). Obviously, this project is not meant to reach the level of complexity offered by these generators. However, they have served as a point of inspiration and a starting point.

In addition, they have been used to see what these generators have to offer in order to try to put the same or similar tools on our terrains. This is part of the state of the art and is used in today's working world, because after all, nobody nowadays implements anything from scratch. People see what is already implemented and, from there, they get inspired about what can be created.

The project started in early summer, which was when I first contacted Francisco Guil and told him that I was interested in procedural generation and the process of creating video games in Unity. Since then I have been working on this project. Sometimes I've had breaks that have lasted weeks or even a month, but as it was a project that I started with time, it has had time to mature and change over time. Therefore the project has been created continuously and unhurriedly.

During this process of creation, I have encountered many obstacles when it came to understanding how to begin with this final degree project. The first drawback was the creation of the meshes. As it is a simple process, but with which you have to be very careful, because as shown later on, depending on how many vertices are used, it will give one result or another.

The subdivision of meshes also took some time, as new meshes must be created taking into account the previous ones and always respecting an order. On the other hand, we also have to understand how Perlin noise works, since, although it is a relatively simple idea, many different variants can be applied to obtain very different results.

The second drawback, we have the creation of caves. This was difficult at first since we had to try to create caves with a certain shape and try to see which vertices were inside an object. And this

is not as simple as it seems, since there is no method of intersection between two objects.

The texturing and coloring of the terrain were not an easy task either. As for example, we considered using different textures depending on the type of terrain, however, it was observed that with the same gotelé texture on all edges and it was quite realistic. I have also learned to use shaders in Unity in a basic but solid way, which has allowed me to create water, clouds, among others.

It is important to emphasize that the project is running in Unity edit mode, so the terrain is changing as we edit our game. This has made me have to research and use Unity's script editors, which although it was not an arduous task, I had never used them before and so it took me some time to learn. Nevertheless it has resulted in a advanced and elegant editor. We modified the editor itself so it does not get messy and gets an elegant result.

Throughout the report, all these problems and the solution found for them will be discussed, as well as the ideas that arose, those that were implemented, and which were discarded. One of these ideas that arose was to implement an algorithm to convert the terrain from a mesh terrain to a cube terrain, simulating a Minecraft [5] world. Another quite interesting idea that was put forward was to generate planets or torus from a flat terrain.

The creation process I followed when creating a tool for the generator was first to see what people had created and what problems they had had in dealing with it. Then I would try to make myself a draft pseudocode, usually in a notebook. Afterwards I would write the code.

The reason behind this was to not jump straight into the code, as this is not and has not been directly a recommended practice. This is what we have been told throughout the years of university. This process has made it much easier for me and has allowed me to see errors in advance when programming the code.

If we were to use the generator, the logical (and recommended) process to generate a terrain is to give it a size of width and height. Afterwards use the main shapes of how we want our terrain to be, this will make it easier for us to give it our personal touch.

Then we proceed to subdivide the mesh and add details as considered, applying the Perlin noise and generating caves. Then we proceed to add the colors and texture. After that, I would add the objects randomly or by means of Perlin noise. This is where we will add the decorative details such as light, water, clouds and so on.

Finally, we have also made a small character that can move around the terrain from a third person point of view. Once all this is done, it is time to create the videogame and everything else that you like to be included.

Different resources and audiovisual media have been consulted for the creation of this work; it is true that there are people who have implemented similar things. Some techniques have been used as inspiration to create other tools, but all the work is original, except for some decorative objects that will be mentioned later. Since creating all the art involved in creating a landscape does not fit in my work for a matter of time, packages from the asset store have been used.

This final degree project has many things that could be further studied and I could spend two

or three years working on this procedural generator, since the best procedural generators on the market are created by entire teams with a extensive experience in the world of videogames.

The directions where I think new features could be implemented would be adding new shaders, making tools with brushes, adding more types of noises, automatic biomes according to the environment, desert effects, procedural rivers, procedural lights, volcano effects and many, many others.

In the end, my project is a basic but sufficient generator to create a terrain that can be used by anyone and that allows them to have a terrain to play on.

As a conclusion, I would like to emphasize that I am quite pleased with the generator that I have created and with everything that I have learnt while making it. Right now, I feel like I am able to learn how to make and implement many more procedural generation tools and even, in the future, I envision myself making a complete procedural generation, with houses, roads and even cities.

3. Introducción

Los videojuegos pueden ser considerados como una mezcla entre técnica y arte. La creación de un videojuego conlleva la unión de muchísimas disciplinas, desde la programación hasta el diseño, pasando por las matemáticas y la física, tal y como he podido aprender desde que hice mi primer videojuego en primero de carrera con la asignatura de Tecnología de la Programación hasta Fundamentos Computacionales de Videojuegos o la parte de inteligencia artificial que se ve en IA para el Desarrollo de Videojuegos.

Muchas veces usamos estos videojuegos para escapar de la realidad y poder tomar un descanso del mundo diario. Queremos escalar montañas, sumergirnos bajo el océano, ir por praderas, descubrir animales fantásticos, entre otros. Esta es la razón de mi trabajo de fin de grado, poder simular lugares inauditos. Una herramienta fundamental para estas simulaciones es la generación procedural, esto es, algoritmos que permiten generar de modo automático (o semi automático) partes del juego. Además de usarse en el mundo de los videojuegos, estas técnicas también se usan en el mundo de los juegos de mesa, en películas, diseños de ropa, etc. Por eso la generación procedural no solo abarca la generación de paisajes sino todo lo que nos rodea, incluso podríamos decir que la naturaleza en si es generación procedural. Podemos ver ejemplos de generadores procedurales en las minas de Stardew Valley [10] en los propios planetas y biomas de No Man's Sky [19] o en Minecraft [5], además de muchos otros juegos que implementan esta técnica. Al final la generación procedural es un proceso de creación aleatoria que genera cualquier contenido de forma controlada, pero con sentido.

La generación procedural nos ayuda mucho en la creación de paisajes puesto que es un mecanismo muy potente para generarlos de forma rápida y posteriormente aplicar ciertas modificaciones al gusto de la persona, es decir podemos crear un terreno base y luego hacer modificaciones que den un aspecto personal. La generación procedural es una combinación de distintos algoritmos y es prácticamente imposible abarcárlas todas. Las que se van a ver durante este trabajo se centran en la creación de mallas, aplicar ruido (principalmente ruido Perlin), colocar objetos por altura y con ruido, generar cuevas, hendiduras, lagos, suavizar terrenos, aplicar texturas, colorear terrenos, olas, nubes, entre otros.

Este trabajo se centra en la realización la creación de un generador de terrenos procedurales y los algoritmos utilizados en cada una de las herramientas que conlleva dicho generador. Las implementaciones mencionadas se han realizado apoyándonos en las utilidades que nos ofrece Unity[8]. Todo el trabajo se podría haber hecho desde cero, implementando un motor de videojuegos, pero solo la creación de un motor daría material para otro trabajo de fin de grado.

El proyecto se ha realizado mediante Unity[8] puesto que es uno de los motores de videojuegos que más se usan en la industria actual, junto con Unreal Engine[9] entre otros. Además es el que se enseña en la mención de computación, por lo que es con el que más tengo familiaridad y por lo tanto, con el que mejor me desenvuelvo. Aunque hay que recalcar que no soy ningún experto, puesto que saber manejar Unity al completo es un proceso de años de aprendizaje, ya que es muy software muy grande y complejo.

En esta memoria se va a detallar todo el proceso realizado para la creación de dicho generador, es decir no solo cómo se han implementado sino también el proceso creativo. Desde el entendimiento

de las bases de cómo funcionan las mallas usadas para el terreno hasta la generación de nubes, pasando por cómo se aplica ruido y cómo se colocan objetos de forma automática.

Es importante destacar que antes de ponerme a desarrollar el código se hizo un análisis de requisitos y un diagrama general de lo que se iba a crear. Durante el transcurso del proyecto se hicieron modificaciones de este diagrama, pero manteniendo la estructura principal. Como no se tiene una empresa ni un cliente, los requisitos fueron decididos a partir de las características presentes en generadores ya implementados. A partir de este análisis se creó una lista de características deseables con ayuda de mi tutor. Nos reuníamos una vez a la semana para ver el progreso que se hacía de una semana a otra y ver qué cosas se podían mejorar y qué otras se podían implementar. Normalmente seguíamos el plan de características ideado en el análisis de los generadores, aunque algunas veces se nos ocurrían ideas descabelladas que intentábamos implementar.

El proyecto empezó a desarrollarse a principios de verano del curso anterior, que fue cuando contacté por primera vez con Francisco Guil y le comenté que me llamaba mucho la atención la generación procedural y el proceso de creación de videojuegos. Desde entonces llevo trabajando en este proyecto, desde una fase inicial de análisis del problema (verano y parte del primer cuatrimestre) al desarrollo final del mismo una vez tenía la base suficiente para su realización (sobre todo el segundo cuatrimestre). Al ser un proyecto que empecé con tiempo, ha tenido tiempo de madurar y cambiar a lo largo del tiempo.

Durante este proceso de creación he encontrado muchos obstáculos a la hora de entender algunas de las técnicas necesarias para la realización del trabajo fin de grado. Lo primero de todo fue la creación de las mallas, un proceso simple pero con el que hay que llevar mucho cuidado, ya que como veremos más tarde según cuantos vértices se utilicen dará lugar a un resultado u otro. La subdivisión de mallas también llevó su tiempo ya que hay que crear nuevas mallas teniendo en cuenta las anteriores y siempre respetando un orden.

Es importante destacar que Unity a pesar de ser muy potente a la hora de crear los videojuegos no tiene suficientes herramientas para el manejo de mallas, como puede ser Blender [3]. Posteriormente veremos un ejemplo de esto con la intersección de mallas en la creación de cuevas. Al principio este objetivo fue difícil, puesto que había que intentar crear cuevas con una forma determinada y esto no es tan simple como parece.

Por otro lado, también tuve que entender cómo funciona el ruido Perlin, ya que, aunque es una idea relativamente sencilla, se pueden aplicar muchas variantes distintas (ruido Perlin simple, uso de octavas...) para obtener resultados muy dispares.

El texturizado y coloreado del terreno tampoco fue tarea más simple que la anterior. Se valoró usar distintas texturas según el tipo de terreno, pero luego se observó que con una textura igual por todo los bordes y que fuese similar a la técnica del gotelé, se obtenían resultados bastante realistas usando técnicas de coloreado de vértices. También he aprendido a utilizar de manera básica pero sólida el uso de shaders en Unity, que me ha permitido crear el agua o las nubes, entre otros.

Por último es importante recalcar que el proyecto se ejecuta en modo edición de Unity. Por lo tanto, el terreno va cambiando mientras editamos nuestro juego. Esto ha hecho que tenga que investigar y usar los editores de scripts de Unity, que aunque no son una ardua tarea, nunca los había usado

y por lo tanto me tomó algo de tiempo aprender a usarlos (no se enseñan en ningún momento de la carrera). Pero ha permitido dar lugar a un editor bastante avanzado y elegante. También hay que destacar que he aprendido C# un lenguaje que no había visto antes, solo de manera básica en FCV y un poco más en IADJ, pero principalmente mi aprendizaje ha sido con el TFG.

A lo largo de la memoria se irán comentando todos estos problemas y la solución para ellos, además de las ideas que surgieron, cuáles fueron implementadas y cuáles fueron descartadas. Entre las ideas descartadas por falta de tiempo figuran la implementación un algoritmo para pasar el terreno de malla a terreno de cubos, simulando un mundo de Minecraft [5], o la de generar planetas o toros a partir de un terreno plano.

Se han consultado distintos medios para la creación de este trabajo. Hay personas que han implementado cosas similares y algunas de esas técnicas se han usado como inspiración para crear otras herramientas. A partir de este punto de partida todo el trabajo es original mío, salvo quitando algunos objetos de decoración que se mencionarán más adelante (ya que crear todo el arte que involucra crear un paisaje no entra en mi trabajo por cuestión de tiempo) y para lo que se han utilizado paquetes de la Asset Store de Unity [1].

Este trabajo de fin de grado tiene muchísimas cosas donde se podría seguir avanzando y podría estar perfectamente dos o tres años trabajando en este generador procedural. De hecho, los mejores generadores procedurales del mercado están creados por equipos enteros con una amplia experiencia en el mundo de los videojuegos. Los principales aspectos donde creo que se podrían implementar nuevas funcionalidades serían añadir nuevos shaders, hacer técnicas con pinceles, añadir más tipos de ruidos, biomas según el entorno de forma automática, efectos de desierto, ríos procedurales, luces procedurales, efectos de volcanes y muchos otros. Se entrará más en detalle en estos aspectos en la sección de vías futuras.

Al final mi proyecto proporciona un generador básico pero suficiente para crear un terreno que pueda ser usado por cualquier persona y que permita a cualquiera tener un terreno donde poder jugar.

4. Estado del arte

En esta sección se hace un análisis de las tecnologías presentes en el mercado que pueden resolver el problema de la generación procedural y la descripción de estas. Además se describirá cuáles son sus puntos fuertes y sus puntos débiles.

Hay bastantes herramientas que sirven para generar terrenos. Pero nos centraremos en aquellas que principalmente se usan a día de hoy. Para ello, nos basaremos en aquellos generadores de paisajes que puedan ser usados para jugarlos actualmente, principalmente de la Asset Store, ya que como hemos mencionado anteriormente Unity será el motor de videojuegos que usaré.

Entre todos los generadores procedurales que podemos encontrar, podemos destacar los tres mejores generadores de terreno que se presentan en la Unity Asset Store: Gaia Pro [28], Terrain Composer 2 [16] y TerraWorld [24]. Además tenemos el generador propio de Unity, que también estudiaremos.

- GaiaPro: Gaia Pro es un generador completo, puesto que implementa una variedad de herramientas muy amplia. Está creado por la compañía Procedural Words y tiene un gran equipo a sus espaldas. En la figura 1 podemos ver un ejemplo de los resultados obtenidos con el software. Permite generación de vegetación, montañas, ríos, lagos, entre muchos otros, además el acabado es impecable.¹
- Terrain Composer 2: Por otro lado, Terrain Composer es un generador bastante bueno. No da unos resultados tan espectaculares como el primer generador, pero son resultados que (dependiendo del juego y de la complejidad que se requiera) son más que suficientes. Este software está creada por Nathaniel Doldersum, un desarrollador de software que tiene una experiencia de alrededor de 40 años programando (10 de ellos en Unity), por lo que podemos decir que es un experto en el tema. En la figura 2 podemos observar los resultados de este generador.²
- TerraWorld: es un generador muy interesante puesto que, aunque no es igual de bueno que el primero por su acabado, a cambio ofrece otras características a tener en cuenta. Entre ellas se encuentran la generación de localizaciones de la Tierra según las coordenadas que le indiques y una versión *Low-Poly* dando lugar a un estilo más retro. Además permite personalizar estos terrenos generados a partir de localizaciones del mundo real. TerraWorld fue creado por la compañía TerraUnity la cual tiene un equipo experimentado y bien formado. Podemos ver en la figura 3 como quedan los resultados de usar este generador.³
- Generador propio de Unity: Es un software bastante potente, aunque se usa principalmente con pinceles. La principal ventaja es que es gratuito y por lo tanto no se ve limitado por el precio como en los casos anteriores. Por otro lado, es una herramienta de Unity en sí y por lo tanto facilita mucho la creación de terrenos.⁴

4.1. Estado del arte

En la siguiente tabla podemos ver las características que ofrece cada uno y en las imágenes que siguen a continuación podemos ver ejemplos con los generadores mencionados anteriormente. De esta forma podemos comparar qué diferencias hay entre cada uno ellos de manera visual.

En la tabla las casillas que están marcadas con una *X* quiere decir que el programa indicado tiene dicha característica. Si está marcado con ? es que no se tiene suficiente información de si tiene dicha característica o no. Finalmente la marca # quiere decir que su propio generador no lo implementa, pero sí se podría hacer con las otras herramientas incluidas en Unity (sin necesidad de programación).

¹Ejemplo de uso Gaia Pro

²Ejemplo de uso Terrain Composer 2

³Ejemplo de uso TerraWorld

⁴Ejemplo uso generador de terreno de Unity

	Gaia Pro	Terrain Composer 2	Terra World	Unity	Mi generador
<i>Gen. terrenos</i>	X	X	X	X	X
<i>Gen. Vegetación</i>	X	X	X	X	X
<i>Gen. aguas</i>	X	X	X	X	X
<i>Gen. lluvia/nieve</i>	X		X	#	#
<i>Iluminación</i>	X		X	#	#
<i>Añadir objetos según altura</i>	X	X	X		X
<i>Nubes</i>	X	X	X	#	X
<i>Cuevas</i>	X	X	?		X
Precio	258,17	40,20	53,59	0	0



Figura 1: Ejemplo de generación de terreno con Gaia Pro



Figura 2: Ejemplo de generación de terreno con Terrain Composer 2



Figura 3: Ejemplo de generación de terreno con TerrWorld



Figura 4: Ejemplo de generación de terreno con Unity Terrain

Como se puede ver en la tabla la mayoría de terrenos tiene una generación de vegetación, agua y biomas. He considerado que Unity no tiene herramientas tales como la generación de lluvia, porque, aunque se puede hacer con el propio Unity, la herramienta que lleva incorporada de terrenos no tiene dicha herramienta. La mayor crítica que se le puede hacer a los primeros tres generadores es el precio, ya que, aunque a pesar de tener un resultado increíble, son bastante costosos. Por otro lado, Unity no tiene un generador de cuevas lo que supone un problema si uno necesita una.

4.2. Propuesta

Nuestro objetivo es crear un generador de terreno que cumpla con la mayoría de funciones vistas en la tabla superior, además de algunas características adicionales personales. La idea principal del proyecto es entender cómo funciona la generación procedural y hacer un generador de terrenos automático. E implementar un generador de cuevas, para poder suplir el problema que tiene Unity en su generador. Es obvio que con el tiempo disponible no se puede conseguir un resultado similar al de Gaia ya que no tenemos ni los recursos ni el tiempo suficiente para ello. Pero si se va a intentar tener un generador lo más completo posible con todas las herramientas que se pueda, además estará gratuito para que cualquier persona lo pueda utilizar si así lo desea.

5. Análisis de objetivos y metodología

En este apartado se indican los objetivos que se planteaban alcanzar con la creación de este proyecto. Una vez decididos estos, se subdividen en tareas y se ponen marcas a lo largo del tiempo para poder cumplirlas y completar el proyecto.

5.1. Objetivos

El objetivo final de este trabajo de fin de grado es desarrollar una herramienta de generador de paisajes procedurales. Esta herramienta debe ser lo más completa posible y permitir generar terrenos con los cuales se pueda desarrollar un videojuego. Las principales subtareas de este proyecto son:

- Estudio de la generación procedural
- Estudio de las tecnologías y herramientas actuales de generación procedural
- Análisis de requisitos
- Diseño de creación de mallas y subdivisión
- Aplicar ruido sobre las mallas
- Modificaciones sobre la malla (surcos, cuevas...)
- Añadir objetos sobre el terreno de forma procedural
- Texturización y coloreado de mallas
- Agua, nubes ...
- Iluminación y características adicionales

Hay que recalcar, tal y como se ha dicho con anterioridad, que no se pretende producir un generador de terrenos completo y comercial, sino se pretende hacer la creación de un generador funcional, básico. En el apartado de conclusiones y vías futuras se comentarán algunos aspectos que podrían mejorarse.

5.2. Metodología

Como ya hemos decidido nuestros objetivos principales, es hora de diseñar un plan de trabajo para cumplir dichos objetivos. Para ello se deben concretar y estipular las tareas y subtareas que permitan tales objetivos. Tenemos las siguientes:

- Análisis de generación procedural
 - Estudio de los conceptos de generación procedural: Se verán todos los conceptos y detalles de cómo se usa la generación procedural en el mercado actual.
 - Análisis y visualización de herramientas que usen generación procedural: Ver y analizar los generadores procedurales según la información que se tenga al respecto.
 - Estudio de qué game engine utilizar: Analizar los distintos game engine que hay en el mercado y decidir cuál es el que se va a utilizar.

- Estudio de las tecnologías actuales
 - Analizar generadores procedurales de terrenos: Una vez ya estén estos analizados ver en qué se diferencian y las características diferencian unos a otros.
- Estudio de mallas
 - Implementación de distintos objetos usando mallas: Se crean objetos básicos, tales como cuadrados, cubos, triángulos.
 - Creación de plano con mallas de distintos tamaños: Implementación de mallas basadas en triángulos
 - Subdivisión de mallas: Dada una malla, esta se subdividirá de forma que se tendrán más vértices y triángulos
- Aplicar ruido
 - Aplicar ruido general: Aplicar ruido Perlin a toda la malla, según unos parámetros de entrada
 - Aplicar ruido por zonas: Estudio y aplicación de ruido Perlin por zonas
 - Aplicar combinaciones de ruido Perlin (Octavas): Ver qué son las octavas y cómo se pueden implementar
 - Aplicar ruido por alturas: Estido y aplicar ruido Perlin por altura
- Modificaciones sobre la malla
 - Empujar para hacer surcos, cuevas: Estudiar los algoritmos para implementar una herramienta que permita empujar vértices para generar cavidades en una malla.
 - Suavizar zonas: Estudio de cómo suavizar un terreno, de forma que sea más plano y con menos ruido
 - Poder volver a una malla anterior: Implementación de una herramienta que permita deshacer los cambios en una malla y volver a la malla anterior
 - Volver a la malla original: Implementación de una herramienta que permita volver a la malla original y deshacer todos los cambios
- Añadir objetos sobre el terreno de forma procedural
 - Añadir objetos de forma aleatoria: Estudio e implementación de cómo se pueden añadir objetos al terreno de forma aleatoria
 - Añadir objetos con ruido Perlin: Añadir objetos de forma aleatoria usando ruido Perlin.
 - Añadir objetos de una lista dada: Usar objetos vacíos con distintos hijos y añadirlos de forma aleatoria. Uso para añadir distintos árboles y generar un bosque.
 - Añadir objetos, cambiando el tamaño, la rotación, rotación basada en el terreno: Añadir los objetos, pero con ciertas modificaciones tales que si se añaden dos instancias de un mismo objeto haya cierta variedad en estos a la hora de añadirlos.
- Texturización y coloreado de la malla

- Entender cómo funcionan los shaders básicos: Ver y analizar los shader y cómo se usan.
- Coloreado de la malla con gradientes: Colorear la malla con shaders y uso de un gradiente, de forma que se pueda colorear de forma personalizada
- Intentar distintas texturas y sus resultados: Estudiar cómo queda la malla con distintas texturas y comparar los resultados para elegir la textura que se considere más acorde
- Agua y nubes
 - Probar distintos tipos de agua y ver cual funciona mejor: Ver y analizar los distintos shaders de agua e implementar uno de estos.
 - Nubes: Ver y analizar los distintos shaders de nubes e implementar uno de estos.
- Iluminación y características adicionales
 - Ciclo de día - noche: Ver cómo se podría implementar un ciclo de día noche y hacer uso de este.
 - Postprocesado: Hacer uso de técnicas de postprocesado para mejorar el resultado final.
 - Jugador primera persona: Implementar un jugador en primera persona que pueda moverse por el terreno.
- Documentación
 - Elaborar memoria del TFG gradualmente
 - Elaborar presentación del TFG

La documentación y revisión se ha hecho paralelamente con las demás tareas, aunque al final se le dedicó una cantidad extra de horas para refinar y aclarar ciertos puntos.

6. Diseño y resolución del trabajo realizado

Esta sección está dedicada al trabajo de análisis e implementación de los algoritmos y herramientas del trabajo. Primero se verá el editor de clases, que se ha ido aprendiendo mientras se creaba el proyecto. Luego se analizará la generación de mallas en Unity, ya que no es trivial. Posteriormente se estudiará cómo modificar dicha malla inicial mediante vértices editables. Más tarde se hablará de la división de la malla y de la aplicación del ruido Perlin con las modificaciones vistas al inicio. Posteriormente se verán el texturizado y coloreado, cómo añadir objetos a la malla de forma aleatoria pero controlada y los detalles estéticos.

6.1. Editor de clases

Como he mencionado anteriormente a la hora de hacer este generador de paisajes procedural hay que tener en cuenta el factor de que cada vez que alguien lo ejecute quiera darle cierto toque de

personalidad al paisaje creado y que sea único. Por eso es conveniente implementar un editor de clases para cada uno de los scripts, de forma que se puedan modificar los valores de los parámetros de una forma cómoda tal y como se usaría un software profesional, por eso se modifica el propio editor que Unity tiene implementado de forma nativa. Esto se podría hacer todo con variables públicas y modificarlas, pero una forma más adecuada de hacerlo es modificar el propio entorno de Unity lo que proporciona una interfaz más sencilla para esta labor de ajuste.

A continuación se detallan as distintos aspectos que se han implementado en el editor y serán utilizados a lo largo de todo el proyecto.

6.1.1. Bases del editor

Para cada clase de Monobehaviour que se declare, se puede hacer un editor. Esto se hace indicándoselo con una cabecera. Un ejemplo de esto podría ser:

```
[CustomEditor(typeof(GridGenerator))]
```

Posteriormente lo que haremos será dentro de la función *OnInspectorGUI()* añadir lo que nosotros necesitemos. Ya que esta función lo que nos permite es personalizar el editor y añadirle los modificadores que necesitemos. Si queremos mantener todo lo original, es decir, mantener las variables públicas que se ponen por defecto tenemos que añadirle *DrawDefaultInspector()*;

6.1.2. Casillas marcables

Son básicamente checkboxes y sirven para establecer el valor a verdadero o a falso.

Ejemplo: Queremos que se muestren los cubos que representan a los vértices solamente cuando vayamos a editarlos. El código asociado sería el siguiente.

```
1 _hideVertices = EditorGUILayout.Toggle("Hide Vertices", _hideVertices);
2 if (_hideVertices)
3 {
4     gridGenerator.HideVertices();
5 }
6 else
7 {
8     gridGenerator.ShowVertices();
9 }
```

6.1.3. Botones

Los botones sirven para cuando se pulse un botón se ejecute una acción o una función.

Ejemplo: Para aplicar el ruido Perlin, para subdividir la malla. El código asociado sería el siguiente.

```

1 if (GUILayout.Button("Apply PN"))
2 {
3     gridTerrain.PerlinNoise(_perlinNoiseMultiplier);
4     gridTerrain.UpdateMesh();
5 }
```

6.1.4. Sliders

Una función muy interesante es el uso de sliders, que sirven para crear un control deslizante, de esta manera podríamos asignar un valor a una variable, con un mínimo y un máximo.

Ejemplo: Se ha usado para controlar el multiplicador del ruido Perlin, de forma que cuando el valor sea mayor, más será la alteración producida por este modificador.

```
1 perlinNoiseMultiplier = EditorGUILayout.Slider(_perlinNoiseMultiplier, 0.01f, 0.99f);
```

6.1.5. Modificadores de la escena

Muy importante puesto que hace referencia a todo lo que es la modificación de la escena de edición, lo que se conoce como *Scene View*.

Ejemplo: Suponemos que queremos dibujar dentro de la escena un círculo para poder colocar objetos que rodee el ratón. Una forma de hacerlo sería la siguiente:

```

1 private void OnSceneGUI(SceneView sv)
2 {
3     if (_drawCircle)
4     {
5         Vector3 mousePos = Event.current.mousePosition;
6         mousePos.y = sv.camera.pixelHeight - mousePos.y;
7         Ray ray = sv.camera.ScreenPointToRay(mousePos);
8         RaycastHit hit;
9         if (Physics.Raycast(ray, out hit))
10        {
11            Handles.color = Color.red;
12            Handles.DrawWireDisc(hit.point, hit.normal, radiosCircle);
13            sv.Repaint();
14        }
15    }
16 }
```

6.1.6. Otros aspectos

Otros elementos interesantes pero que no son de vital importancia para tener su propia sección son elementos estéticos, que hacen que el interfaz sea más intuitivo. Entre estos podemos destacar

EditorGUILayout.Space() que sirve para poner un salto de línea entre los distintos elementos. *EditorGUILayout.LabelField(Texto, Estilo)*, que sirve para poner títulos o texto en el propio Editor, yo lo he usado para dividir los botones y funcionalidades por secciones.

6.2. Primeros pasos

Mis primeros pasos a la hora de abordar este proyecto fue entender qué era la generación procedural. Para esto consulté varios libros y medios audiovisuales, el primer libro que consulté fue *Procedural Content Generation in Games* [22]. Este libro me ayudó a obtener una idea general de en qué consistía todo el mundo de la generación procedural. Otro artículo que me pareció interesante fue el de *Automatic evolution of programs for procedural generation of terrains for video games: Accessibility and edge length constraints* [17] donde se centra más en la implementación y desarrollo.

La generación procedural es generar cualquier tipo de contenido de forma aleatoria pero controlada.

Hubo también varios vídeos que incrementaron mi interés en este universo, pero en especial me gustaría destacar una charla que explicaba la generación procedural para todo el mundo, donde muestra qué cosas tan alucinantes se podían hacer [15]. Por otra parte se hace referencia y se citan videos de YouTube esto se debe principalmente a que si alguien quiere iniciarse en el mundo de la generación procedural le será mucho más sencillo a través de estos vídeos y posteriormente adentrarse en los libros y artículos que tienen una mayor extensión y se centran en aspectos más técnicos.

Después de consultar más contenidos, decidí que lo que más me llamaba la atención era la generación procedural de terrenos y paisajes. Un ejemplo muy clásico es el juego del que la mayoría hemos oído hablar *Minecraft* [5]. Este programa genera a partir de una serie de algoritmos y una semilla aleatoria un mundo entero, que ha logrado fascinar a millones de jugadores y que a día de hoy se sigue jugando.

6.3. Generación de objetos básicos

En primer lugar, debemos entender que el mundo de los videojuegos funciona principalmente con triángulos. ¿Por qué triángulos? porque son muy eficientes en cuanto a tiempos de ejecución para la máquina y no son ambiguos, ya que con tres puntos lo único que puedes hacer es un triángulo. En cambio, si cogemos cuatro puntos, podrían representar tanto un cuadrado como una pirámide triangular. Además cualquier figura geométrica plana sencillas se puede dividir en dos o más triángulos. [25]

En Unity, la clase *mesh* nos permite representar todo el mundo de la geometría a base de trabajar con vértices y triángulo. Para crear un triángulo simplemente se tienen que crear dos objetos, un *array* con los vértices (posiciones en el espacio) y un *array* con una lista de los índices de los vértices que forman el triángulo en el orden adecuado.

El sentido de los vértices se añade por una cuestión de eficiencia. Cuando se crea un objeto normalmente éste solo se ve desde un lado. Supongamos que tenemos un cuadro colgado en la pared, por ejemplo, la Mona Lisa (que representará un plano). Si la vemos de frente, veremos el cuadro sin problema, pero si pudiéramos ver La Mona Lisa desde detrás de la pared, no veríamos nada. Esto se hace debido a que en un escenario nadie puede ver desde detrás de las paredes. Otro ejemplo ilustrativo, sería cómo se representa un barril ya que normalmente solamente se ve desde fuera. Para evitar calcular todas las características del objeto visto desde los dos lados, Unity solamente dibuja los triángulos desde el punto de vista adecuado. Es decir, en nuestro ejemplo anterior el barril solo se ve desde fuera, pero si nos pusiéramos dentro del barril no veríamos el barril, sino directamente el exterior como si el barril no existiera. Para hacer esto, Unity tiene en cuenta el vector normal al triángulo y, en función de éste, decide si el triángulo está orientado hacia la cámara (y por tanto lo dibuja) o en dirección contraria (y no lo hace). Este algoritmo, llamado Face Culling, se puede observar en la figura 5

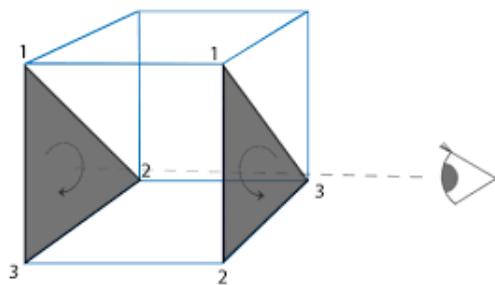


Figura 5: Dos triángulos con distintas orientaciones. El primero sería visible desde el punto de vista de la cámara y el otro no. Imagen obtenida de <https://learnopengl.com/Advanced-OpenGL/Face-culling>

Por tanto, el triángulo anterior se representaría por una lista de tres vectores $\{v_1, v_2, v_3\}$ y una lista de índices $[0, 1, 2]$.

Una vez generado un , el siguiente paso es construir un cuadrado. Éste se genera a partir de dos s, por lo que la idea es la misma que la de los s pero añadiendo un vértice más e indicando los dos s que se forman. Es decir, un cuadrado viene dado por una lista de cuatro vértices y una lista de seis índices, indicando los tres primeros los vértices que forman el primer triángulo y los otros tres el segundo triángulo.

Lo siguiente sería generar un cubo, para lo que necesitaríamos ocho vértices y doce triángulos, cada uno de ellos dado por una lista de tres índices.

En Figura 6 podemos ver los ejemplos mencionados anteriormente.

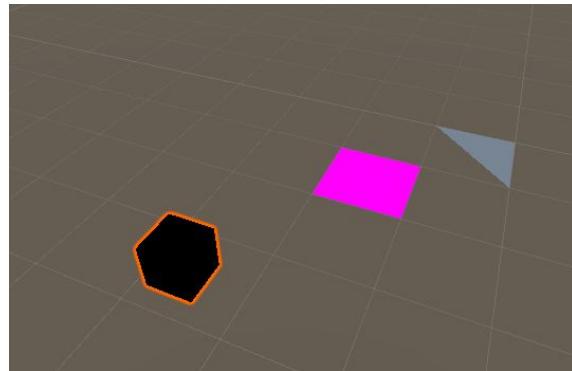


Figura 6: Imagen de los objetos básicos, de izquierda a derecha: Cubo, Cuadrado, Triángulo

Como podemos ver la primera figura se distingue que es un cubo, pero parece que solo se ve en dos dimensiones, es decir, no se distinguen las distintas caras del cubo. Esto se debe a que la iluminación de cada cara del objeto puede hacerse de dos formas: basado en vértices o basado en caras. En la iluminación basada en caras, la intensidad de color de la cara se calcula en función de la perpendicular a la cara mientras que en la basada en vértices se calcula la intensidad de color en cada vértice y luego se interpola en cada cara. En este último modelo, la ‘perpendicular’ en cada vértice se obtiene como la media de las perpendiculares de las caras a las que pertenece el vértice

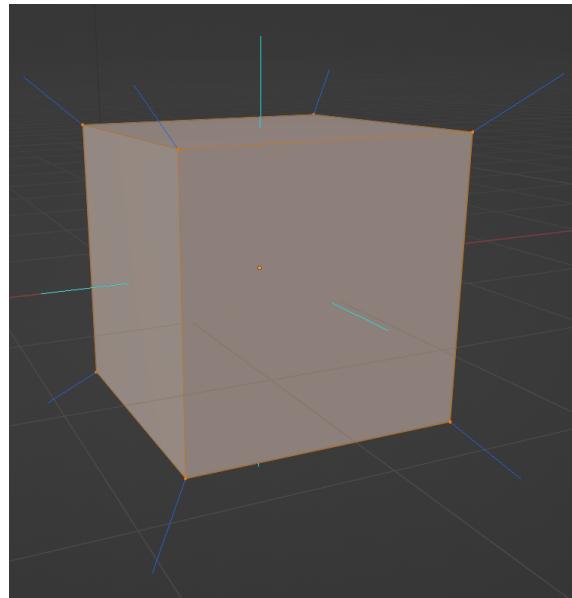


Figura 7: Perpendiculares en caras (azul oscuro) y vértices (azul claro) de un objeto en Blender [3]

Al usar iluminación por caras, cada cara tiene una iluminación uniforme y se diferencia claramente

de las caras adyacentes. Por contra, en la iluminación por vértices el color de una cara no es uniforme y la separación entre ellas es difícil de apreciar. Podemos ver esto claramente en la Figura 8

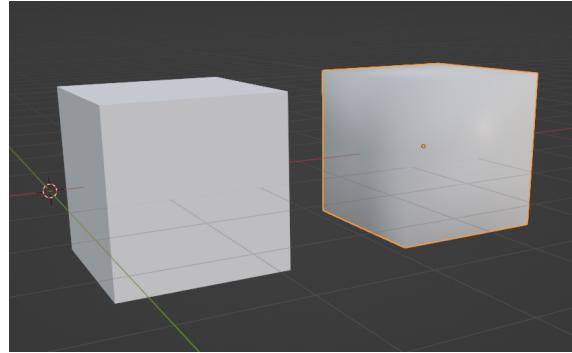


Figura 8: Un mismo objeto con iluminación por caras o por vértices

En Unity la iluminación se hace por vértices. Si queremos iluminación por caras, podemos simularla asignando a todos los vértices de la cara la perpendicular que queremos que tenga la cara. Pero en este caso no podemos tener vértices comunes a dos caras: cada vértice debe repetirse tantas veces como caras a las que pertenezca. Por esta razón aunque un cubo tiene ocho vértices, realmente en Unity tendremos que representarlo con $6 * 4 = 24$ vértices.

Una vez que entendemos esto, usando la idea de que cada cara tiene que tener sus propios vértices y por luego tenemos que crear las distintas caras podemos obtener figuras en las que las caras queden claramente diferenciadas (ver Figura 9). Ahora podemos distinguir claramente en la figura de la derecha las caras del cubo a diferencia del primer cubo. Para cada objeto, podemos decidir usar vértices separados para las caras o, si se necesita un efecto suave, usar vértices comunes a varias caras.

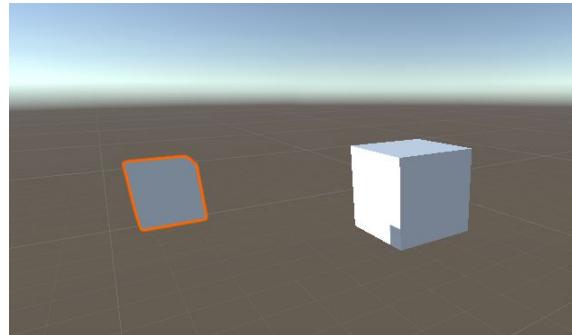


Figura 9: Imagen de cubo compartiendo vértices y sin compartir vértices

6.4. Primeros pasos del generador de terreno

Vamos a realizar un generador básico como los que mencionábamos en la sección anterior. Nuestro punto de partida es crear un grid, esto es una malla 2D plana a la que simplemente podamos cambiar el tamaño. Esta malla estará formada por cuadrados que estos a la vez estén constituidos por triángulos como mencionábamos en la primera sección.

6.4.1. Generador de plano

Para generar este grid lo primera que debería hacerse es generar cuadrados unos al lado de otros formando un plano en dos dimensiones. Por lo tanto, vamos a definir x como el número de cuadrados que tendría el plano en horizontal y z en profundidad. Nuestro plano tendría $x * z$ cuadrados, según nosotros queramos darle mayor o menor tamaño.

Nuestra malla está formada por vértices y triángulos. Entonces para sacar el número de vértices que tiene nuestra malla es $(x + 1) * (z + 1)$. Supongamos el ejemplo de que tenemos una malla donde $x = 3$ y $z = 2$ por lo que tendríamos 6 cuadrados y 12 vértices.

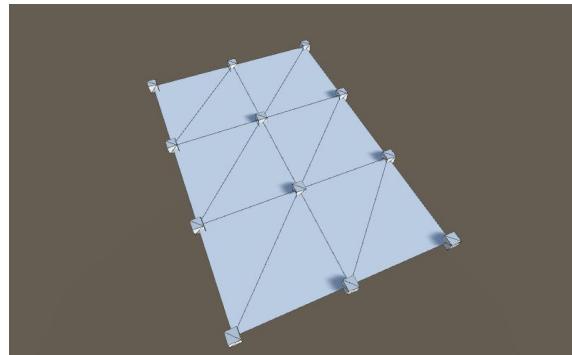


Figura 10: Malla de lado $x = 3$ y $z = 2$

Sabemos que cada triángulo está formado por tres enteros que nos indican qué vértices dentro del array de estos estamos utilizando. Es decir, una malla está formada por dos arrays. El array de los vértices que está formado por vectores de tres dimensiones, que nos indican la posición del vértice en el espacio x, y, z (en Unity se representan con la clase Vector3). El otro array que forma la malla es el que nos indica los índices de los vértices que crean cada triángulo de la malla. Estos índices indican que vértices del array de vértices estamos utilizando y qué sentido tienen en el triángulo, ya que como vimos anteriormente, esto hace que los triángulos se vean desde un punto de vista u otro.

Por lo tanto, si cada cuadrado está formado por dos triángulos y cada triángulo tiene tres vértices, el array de enteros que representa los triángulos tendrá un total de $x * z * 6$ enteros. De esta forma y manejando los dos arrays y las variables x e y podremos generar nuestra malla.

El plano se podrá cambiar mediante dos variables en el propio editor, este tendrá dos valores **x** e **z**. Si cambiamos el valor de alguno de estos estos regenerará una malla de estos valores. Por cuestiones de rendimiento he decidido limitar el tamaño máximo, esto se hace para que al principio siempre haya una malla editable sencilla. Luego se podrá subdividir la malla, de forma que el tamaño de ancho por alto no será un problema a la hora de añadir detalles.

6.5. Generar malla editable

Esta malla editable inicial nos permite generar un esbozo de cómo nos gustaría que fuese nuestro terreno: así se pueden indicar donde estarían las montañas, los lagos, etc. En la implementación realizada, la posición de los vértices del gris se visualizan con objetos generados en Unity. Cuando yo seleccione este objeto, en mi caso un cubo, si su posición cambia también cambiará la posición del vértice asociado al cubo y por lo tanto también el de la malla.

Esto se hace creando un array de figuras del tamaño del array de vértices, siendo las figuras usadas cubos. La idea principal es que si cambiamos la posición de una figura la malla cambia la posición del vértice a la misma posición de la figura. Como al fin y al cabo estamos seleccionando objetos dentro del editor, se pueden seleccionar varias figuras a la vez y moverlos. Además he añadido que cambie el color de la figura cada vez que la tenemos seleccionada de forma que se verá de color rojo.

Por simplicidad de manejo (y a imitación de lo que ocurre en el editor de terrenos de Unity) solo se permitirá mover el vértice en vertical. Se ha hecho así para evitar que movamos el vértice en otras direcciones, la estructura de la malla deja de ser rectangular y pueden salir resultados no deseados.

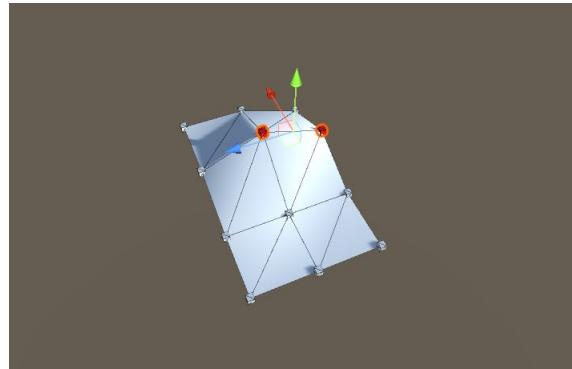


Figura 11: Malla con dos vértices seleccionados

También se permite que los vértices se oculten, de forma por si se quiere observar la malla sin los cubos para ver como quedará antes de aplicar modificaciones. He decidido crear un **Editor** de clases [4] para facilitar la lectura y estructuración del código.

Esto se podría hacer también mediante la declaración de las variables privadas/públicas y decidiendo si se muestran en la interfaz mediante `SerializeField` y `HideInInspector` que son modificadores de

la visibilidad de las variables en el inspector.

Una vez se ha decidido la forma inicial de la malla, se le dará a un botón y se creará una malla en la que ya habremos finalizado la parte inicial. Y pasaremos al siguiente paso que será la aplicación de ruido, aplicar hendiduras y más.

6.6. Modificador del terreno

Una vez existe un boceto de cómo se quiere el terreno con el generador de mallas, se puede pasar a darle forma y a aplicar todos los detalles que se quieran. Estas formas de aplicar detalle se ven a continuación.

6.6.1. Subdivisión de malla

Una vez creado la malla, es interesante poder aplicar subdivisiones para permitir darle más detalle a través de los modificadores de ruido y demás. La subdivisión consiste en que dado un cuadrado subdividirlo en otros cuatro cuadrados. De forma que si hay x cuadrados inicialmente con una subdivisión obtenemos $x * 4$ cuadrados.

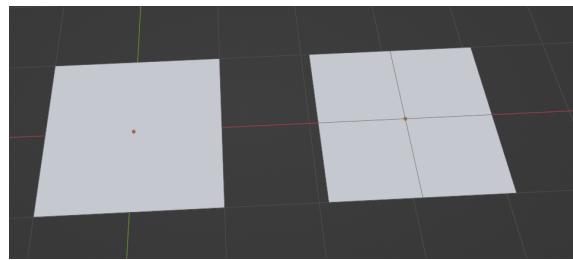


Figura 12: Cuadrado y su subdivisión

La subdivisión se realiza de forma que vamos recorriendo el array de triángulos de 6 en 6 (es decir, los seis vértices de cada cuadrado). A partir de aquí se aplica el siguiente método:

- A partir de los cuatro vértices del cuadrado (sin contar repetidos) se calculan los puntos medios (los cuatro puntos medios de los laterales y el punto central)
- A partir de estos cinco nuevos vértices junto con los cuatro vértices anteriores, podemos obtener los vértices del nuevo cuadrado subdividido
- Finalmente se generan las listas de triángulos

Esto se hace para todos los cuadrados de nuestra malla y el resultado se guarda en una nueva malla.

Cuando se insertan los cuadrados es importante insertarlos en el orden correcto puesto que si queremos volver a subdividir tenemos que coger los vértices en el mismo orden ya que si no estaríamos generando vértices incorrectos y la malla no se generaría correctamente. En la figura 13 podemos ver la malla anterior subdividida una vez. Mientras que en la figura 14 podemos ver la aplicación del algoritmo de subdivisión múltiples veces.

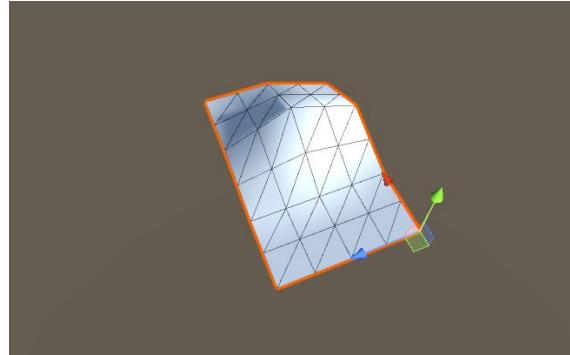


Figura 13: Malla subdividida una vez

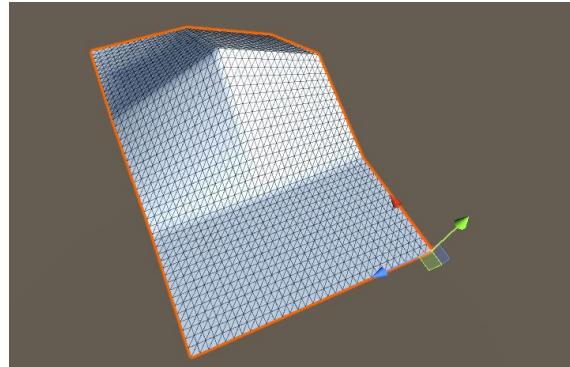


Figura 14: Misma malla subdividida varias veces

Como hemos comentado antes, esta subdivisión nos permitirá dar más detalle a la hora de querer aplicar ruido Perlin, hacer cuevas o para el texturizado final.

6.7. Bordes marginales

Es posible que nos interese que la malla acabe por todos los bordes con altura cero por si posteriormente queremos juntar dos mallas, si queremos transformar nuestro terreno a una esfera u otras posibles implementaciones. Este algoritmo es relativamente sencillo, simplemente consiste en coger aquellos bordes marginales y ponerlos a altura cero.

Supongamos que tenemos la siguiente malla (ver figura 15), que tiene todos los vértices elevados cierta altura.

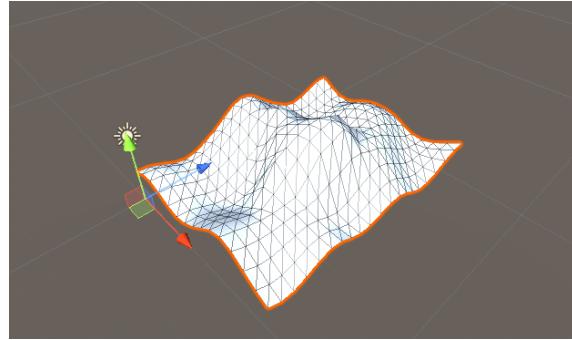


Figura 15: Malla con altura elevada

Cuando apliquemos el algoritmo de que los vértices marginales queden con altura cero quedará de la siguiente forma (ver figura 16):

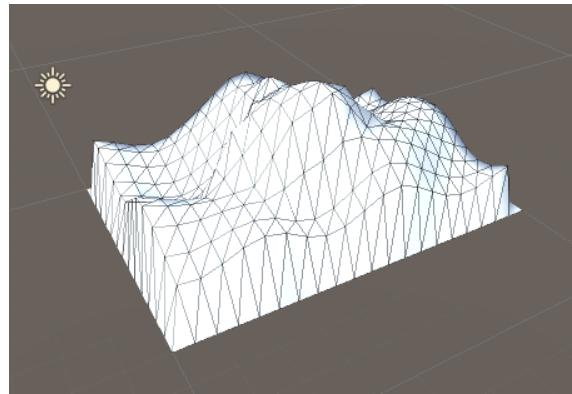


Figura 16: Malla con vértices marginales a cero

6.8. Recuperación de mallas anteriores

Se ha creado un historial de modificaciones en las que se almacenan los distintos estados de la malla al ir aplicándoles modificaciones. Considero que esta implementación es de suma importancia puesto que a veces se realizan cambios sobre la malla y queremos deshacerlos porque no nos ha gustado el resultado obtenido.

Se ha hecho un historial de tamaño máximo diez, de forma que podemos volver hasta diez modificaciones anteriores en el pasado. Por otro lado, siempre se puede volver a la malla inicial realizada con el primer script.

La parte gráfica se ha hecho con un slider, si se mueve este se cambia la malla. Si no se han llegado a diez cambios solo permitirá volver los n últimos cambios que se hayan realizado. El historial de cambios solo se actualiza cuando se modifica la malla con el script, pero no se actualiza si se vuelve atrás con el slider. En la figura 17 podemos ver la interfaz implementada.

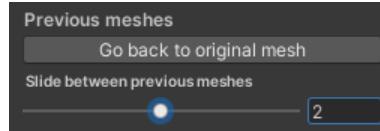


Figura 17: Botones de recuperación de mallas

Para la parte de implementación se ha hecho una lista que actúa como una cola de tamaño diez. Es decir, con cada cambio se saca un elemento de la más antigua y se añade uno al más reciente. Por otro lado, para la malla original lo que se ha hecho es guardarla en un conjunto de variables. Cuando se requiere esta malla simplemente se carga y se sobrescribe la actual con la original.

6.9. Ruido Perlin

El ruido Perlin es una base importante de este trabajo, puesto que permite generar una aleatoriedad controlada. Es una función matemática que utiliza la interpolación entre un gran número de gradientes precalculados de vectores que construyen un valor que varía pseudo aleatoriamente en el espacio tiempo. [27]

Esto quiere decir es que se generan una serie de puntos como si fueran divisiones en un grid y en cada uno de estos puntos se generan unos vectores. La dirección de estos vectores indican las diferencias entre zonas.

6.9.1. Ruido general

El caso más simple es aplicar ruido Perlin a toda la malla. Veamos un ejemplo de esto: partimos de una malla inicial que no cuenta con muchos vértices y vamos a aplicarle el algoritmo de subdivisión y posteriormente el ruido Perlin (observar figura 18).

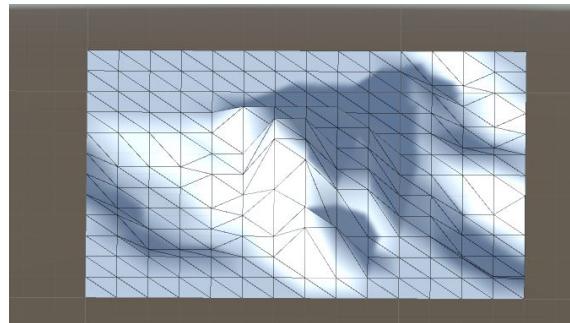


Figura 18: Malla sin aplicar Perlin

Dentro de la interfaz tenemos los valores de **Base Height**, **Perlin Noise Multiplier**.

- **Base Height:** cuánta altura quieras darle a tu ruido.
- **Perlin noise multiplier:** si el ruido va a ser más suave o más puntiagudo.

La fórmula que uso para aplicar una sola capa de ruido (ya veremos posteriormente que con octavas se pueden aplicar varias capas de ruido simultáneas) es la siguiente

$$(PerlinNoise(x * PNM, z * PNM) + 1) * (y + BH) \quad (1)$$

donde **PNM** es perlinNoiseMultiplier y **BH** es la abreviatura de baseHeight. Los dos parámetros que recibe la función PerlinNoise son la posición *x* del vértice y la posición *z* del vértice.

Como podemos observar la segunda malla se ha aplicado dos subdivisiones y un Perlin Noise (ver figura 19) con un valor de 0,737 y una altura de 1,18 y un valor de 0,5 del multiplicador de ruido Perlin. Como se puede ver parece un terreno bastante natural.

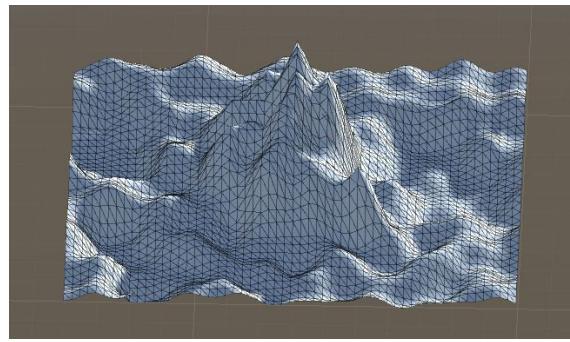


Figura 19: Malla con subdivisión y ruido Perlin

En el ejemplo anterior se ha aplicado el ruido a toda la malla. Se ha cogido cada valor de los vértices y sobre este se ha aplicado este ruido, de forma que aquellos que ya tenían mayor altitud la siguen teniendo. esto se debe a que la función Perlin Noise solamente tiene en cuenta los valores x y z sobre los que están situados en el espacio y los parámetros asignados mediante la interfaz y este ruido se suma a la altura original (observar figura 20).

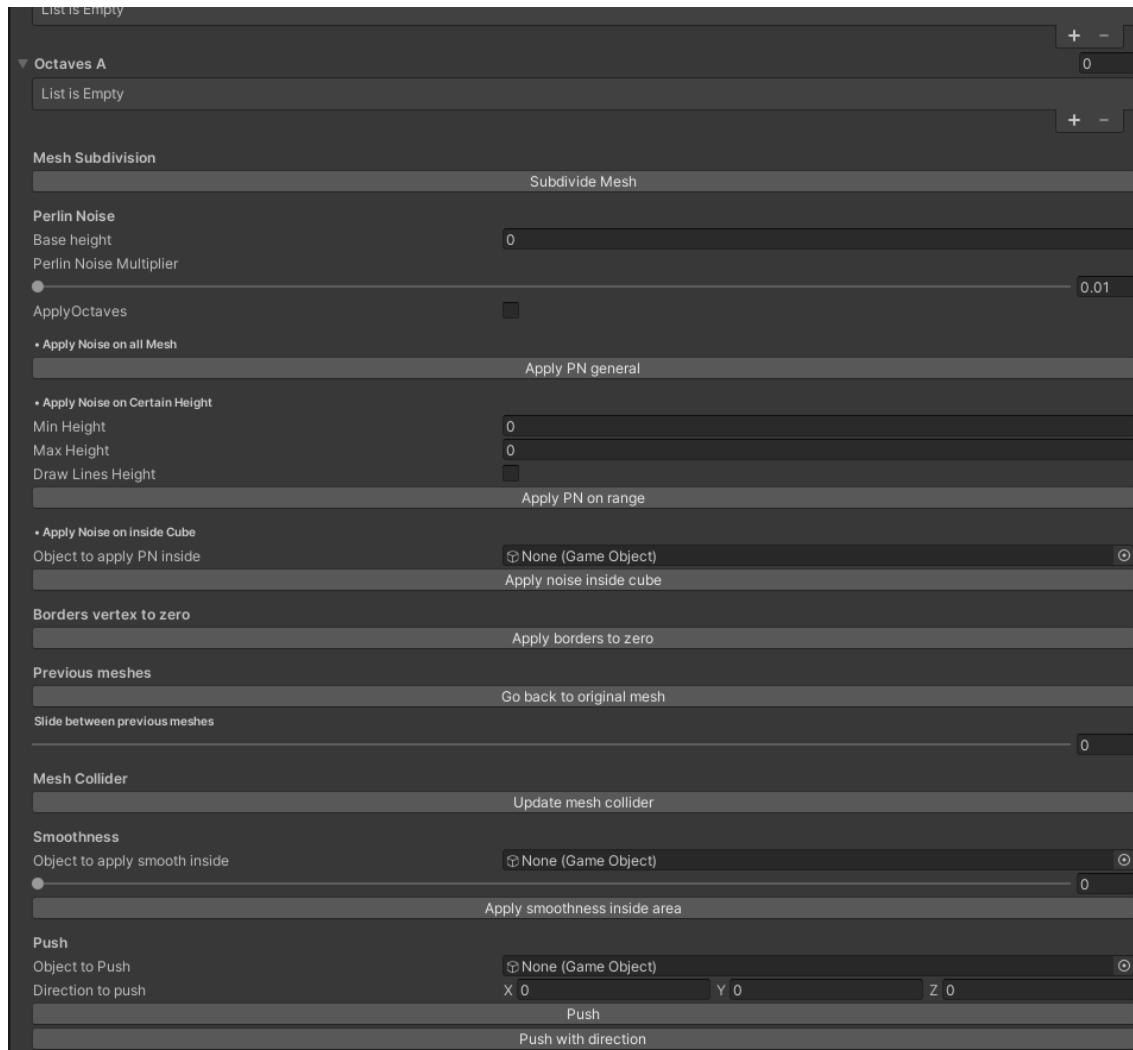


Figura 20: Interfaz ruido Perlin

6.9.2. Ruido por altura

Por otra parte, he conseguido aplicar ruido a los vértices que se encuentran dentro de un rango de altura (útil, por ejemplo, si solo queremos aplicarles ruido a las zonas montañosas de los picos). Para ello se asignan valores de altura mínima y de altura máxima, se cogen todos aquellos vértices que están dentro de este rango y se aplica el ruido.

Aunque sepamos aproximadamente los valores de altura mínima y máxima, a veces es difícil visualizar realmente las alturas. Por ello, cuando se le asignan los valores al rango mínimo y al rango máximo se dibuja una línea de color rojo, que nos indica a qué rango de vértices se van a aplicar ruido. Esto permite visualizar mejor el rango de alturas a las que queremos aplicar nuestro ruido (figura 21).

Dentro de la interfaz que muestra figura 20 tenemos el botón **Apply Noise on Certain Height**. Ahí también se puede ver la casilla, para dibujar o no la línea roja, que se mencionaba con anterioridad.

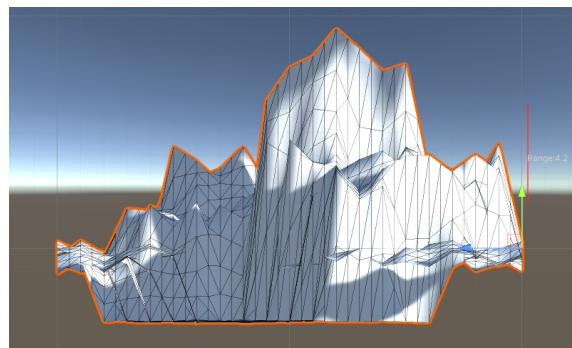


Figura 21: Línea de altura

6.9.3. Ruido por zonas

También se ha conseguido aplicar ruido Perlin dentro de un cubo/esfera/cilindro. Es posible que simplemente queramos aplicar ruido a aquellas zonas que estén dentro de un área, ya que es posible que queramos tener una zona rocosa, por ejemplo, por la caída de un meteorito.

La interfaz anterior permite colocar el cubo en la posición en la que queremos aplicar el ruido y luego arrastrarla dentro del inspector (se han preparado unos objetos que facilitan que se aplique ruido por zonas, ya que dichos objetos necesitan tener las normales invertidas como veremos a continuación). De esta manera todos aquellos vértices que caigan dentro de la figura geométrica correspondiente se les aplicará ruido (observar por ejemplo la figura 22).

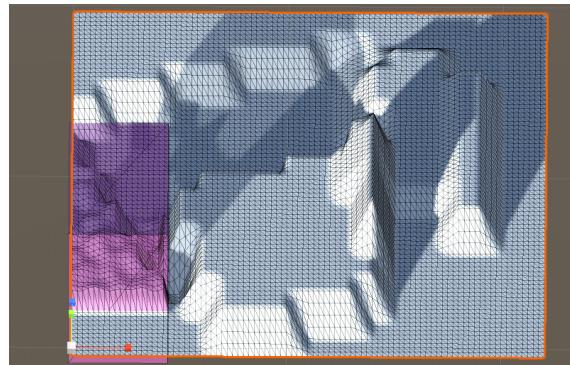


Figura 22: Ruido dentro de caja

Los valores que usamos tanto para el ruido por altura o para el ruido por caja son los mismos que al principio para intentar tener un editor lo más amigable posible.

Inicialmente, la comprobación de qué vértices están dentro del cubo lo se hacía es un método *bounds* (crear un cubo que engloba la figura geométrica y considerar los vértices que están dentro de ese cubo). Pero si quisieramos aplicar ruido dentro de un cilindro o una esfera tendríamos un problema, ya que se generaría un cubo alrededor del cilindro y por lo tanto, obtendríamos el mismo resultado que al usar un cubo y no es el resultado que esperamos. Por ello decidimos modificar la idea anterior y mejorarla para poder usar otras figuras (convexas) en vez del cubo.

Para entender esta implementación, vamos a trasladarnos al plano bidimensional. Suponemos que tenemos un círculo para entender este ejemplo.

- Primero cogemos todos los vértices que tenemos dentro de la malla y comprobamos si están dentro del bound
- Para estos vértices hacemos dos raycast por cada vértice (consiste en lanzar una línea desde donde se encuentra el vértice a una dirección) en sentidos opuestos
 - Una de ellas va hacia el centro de la figura geométrica
 - La otra en el sentido opuesto
- Si ambos rayos golpean un objeto significa que el vértice está dentro, en cualquier otro caso, el vértice está fuera

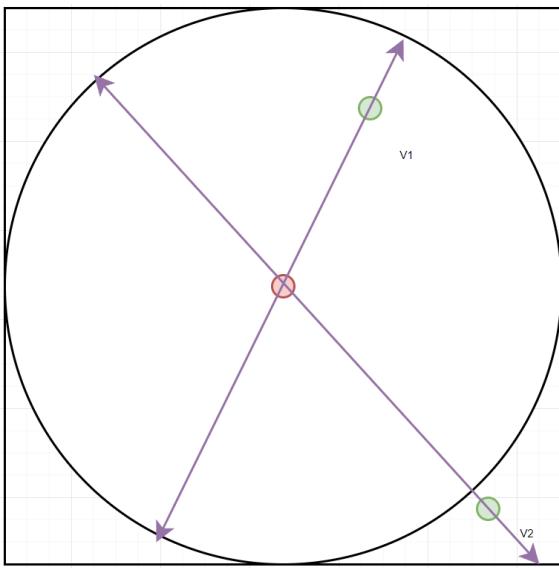


Figura 23: Idea vértices dentro de otras figuras

Observemos la figura 23. Para el vértice V_2 solo se ha detectado una colisión, mientras que para el vértice V_1 se han detectado dos. De esta manera podemos saber (para una gran variedad de objetos geométricos) cuando un vértice está o no está dentro de un objeto.

Se podía dar la situación de que el rayo colisione con la misma malla o incluso con otro objeto de la escena. Para evitar esta situación, se comprueba si el objeto con el que choca pertenece al grupo de Collision (usando etiquetas). Todos aquellos objetos que usemos para aplicar ruido Perlin tendrán la etiqueta Collision.

Un problema adicional es que los algoritmos de colisión de Unity están preparados para detectar colisiones solamente desde fuera de dichos objetos (por motivos de eficiencia como los comentados en el apartado de construcción de mallas). Como nosotros estamos en detectar colisiones desde dentro de ellos (para comprobar qué vértices están en el interior) necesitamos que estos objetos tengan las normales invertidas. Para poder hacer esto tenemos una serie de objetos preparados (para visualizarlos), y copias los mismos objetos con las normales invertidas (para detectar colisiones).

Este método funciona bien para zonas de un tamaño pequeño, mediano, aunque si usáramos figura geométricas muy grandes podría ser algo lento. Este método se puede optimizar si en zonas grandes usamos solamente alturas o zonas definidas por bounds.

Dentro de la interfaz, el proceso es asignar el objeto a usar dentro de la casilla **Object to apply PN inside** y pulsar el botón de aplicar ruido.

6.10. Octavas

Como hemos podido ver, se puede aplicar ruido Perlin a una malla, a una sección determinada, entre otros. Pero todavía no se ve natural del todo, hay cierta sensación de terreno artificial. La solución a esto es aplicar varias capas de ruido unas encimas de otras.

Este es el concepto que se conoce como octavas o ruido Perlin multicapa. La idea es, en vez de aplicar una capa de ruido Perlin se aplican varios ruidos simultáneamente y luego se suman los resultados. En las figuras 24 y 25 podemos ver los resultados de este proceso.

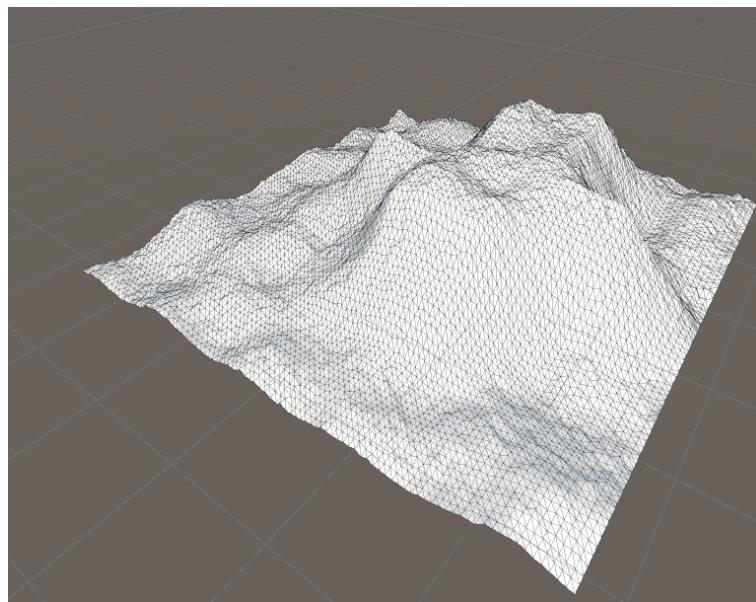


Figura 24: Malla en la que se aplican dos capas de ruido Perlin

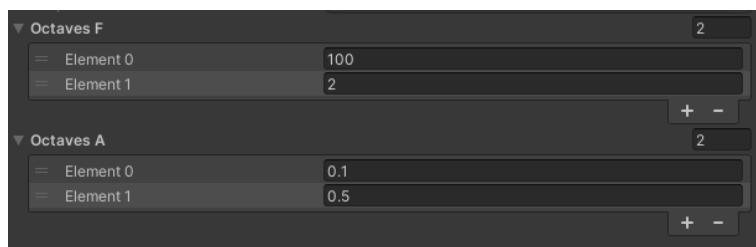


Figura 25: Valores de octavas

El algoritmo es:

1. Se hace un recorrido de cuantas capas se van a aplicar.
2. El resultado de aplicar cada capa de ruido se suma a una variable general.
3. Se cambian del vértice con la variable general.

La fórmula que se aplica en el punto es la siguiente:

$$r = \sum_{i=0}^{len-1} A_i * \text{PerlinNoise}(F_i * x * PNM, F_i * z * PNM) \quad (2)$$

Donde len, es el tamaño de capas que se van a aplicar, A_i es la amplitud i-ésima de la lista, mientras que F_i es la frecuencia i-ésima y, por último, r es la altura a sumar al vértice.

6.11. Suavizado

El proceso de suavizado consiste en

- Seleccionar un área de la misma forma que seleccionábamos para aplicar el ruido Perlin
- Calcular la altura media de los vértices de esa zona
- Usar esa altura media como valor al que acercar las alturas de los vértices aplicando la fórmula

$$p * V_y + (1 - p) * m \quad (3)$$

Donde p es un valor que vamos a usar para suavizar que estará entre uno y cero

Para un valor muy alto de p el suavizado es mínimo, ya que el valor del vértice será la que mayor importancia tenga y por lo tanto, su valor será el original. En cambio, si el valor de p es muy bajo, la altura se suavizará mucho, ya que sucederá lo contrario y la media es el valor que afecta más.

En la siguiente imagen se puede ver cómo se ha aplicado un suavizado dentro del área 26.

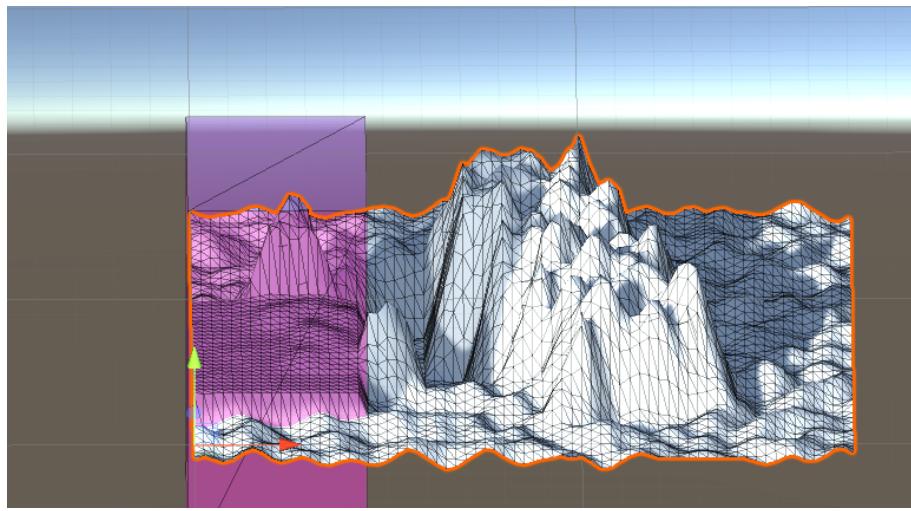


Figura 26: Área suavizada

6.12. Herramienta de empuje

Para la herramienta de empuje y hendiduras se han usado dos métodos distintos (que explicamos más adelante) a la hora de empujar la malla.

Para usarla:

1. Se selecciona un objeto de los elementos disponibles para seleccionar.
2. Se coloca dicho objeto donde se quiere empujar la malla
3. Seleccionamos el método de empuje que queramos

6.12.1. Empuje con dirección

Es necesario indicar la dirección en que se quiere empujar la malla así como el objeto que determina que zona se va a empujar

- Se seleccionan los vértices de la malla que estén dentro del objeto usado (igual que en los métodos vistos con anterioridad).
- Para cada vértice que esté dentro de la figura se hace un raycast desde ese vértice en un sentido determinado por un vector que se pasa como parámetro.
- Cuando se produce la colisión entre el rayo y la normal invertida del objeto se mueve dicho vértice hacia el punto de la colisión.

De esta manera se pueden generar cuevas, lagos, entre otros. Tal y como se puede ver a continuación (figura 27):

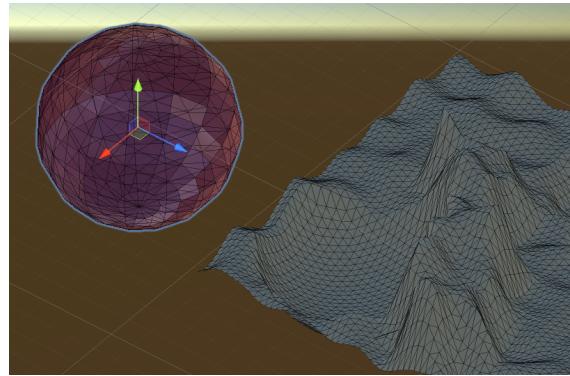


Figura 27: Herramienta de empuje con dirección

Se ha usado la esfera rosa para generar el lago que hay al pie de la montaña de la imagen anterior. De igual manera se pueden generar cuevas.

6.12.2. Empuje sin dirección

La otra forma de empujar es sin indicar la dirección que queremos empujar, ya que esta varía de un vértice a otro.

Para calcular la dirección que se va a empujar el vértice V_i viene dado por el vector que une el centro del objeto con V_i (normalizado).

Por lo tanto, cuando se vaya a empujar si el centro de la figura está debajo de la malla lo que se obtendrá serán extracciones del terreno, es decir, saldrán montañas. En cambio, si el centro de la figura está encima de la malla se generarán cuevas.

En la figura 28 se pueden observar los resultados de aplicar este método de empuje.

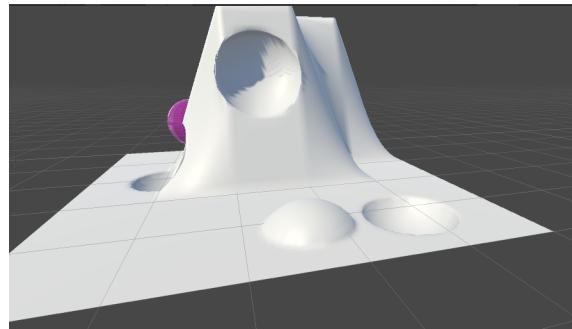


Figura 28: Herramienta de empuje sin dirección

6.12.3. Diferencias entre los dos métodos de empuje

Como hemos podido observar se pueden utilizar dos métodos distintos para empujar. Veamos algunas comparaciones entre ambos métodos. Para esta sección haré referencia al método de empujar con dirección como **M1** y al método de empujar sin dirección como **M2**.

Con ambas herramientas se pueden conseguir **lagos**. Veamos la principal diferencia entre ambos.

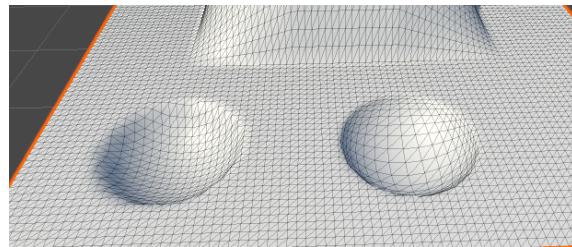


Figura 29: Diferencia lagos

Como se puede ver en la figura 29 en cuanto a los lagos M1 obtiene mejores resultados para la malla en las profundidades, en cambio M2 los bordes tienen mayor calidad. Hay que tener en consideración que estos ejemplos no tienen ningún tipo de ruido ni suavizado. Para este caso son igual de buenos ambos resultados.

Si queremos generar **montañas** ovaladas (por coger un ejemplo), las diferencias de usar los métodos de empuje son las siguientes. La parte de la izquierda está hecha con el M2, mientras que la parte de la derecha está hecha con M1. Aquí las diferencias de los bordes se notan mucho más, pero en cambio, la montaña tiene una resolución mucho mejor (figura 30)

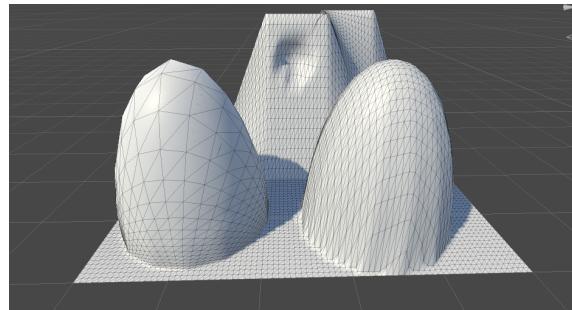


Figura 30: Diferencia montañas

Por último si queremos generar **cuevas** tenemos las siguientes diferencias. La cueva de arriba se ha hecho con la herramienta de empujar con M1, mientras que la cueva de abajo con M2. La hendidura parece más acorde a la figura con M2, mientras que con M1 el fondo es mejor (figura 31)

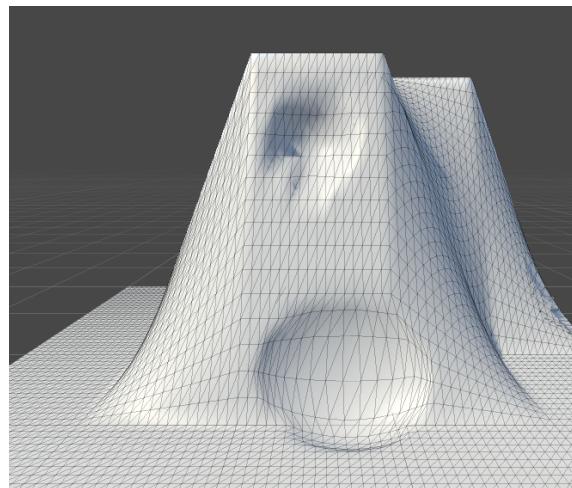


Figura 31: Diferencia cuevas

Lo bueno de M1 es que nos permite empujar en cualquier dirección, es decir, que si quisieramos empujar todos los métodos a una dirección en concreto no habría ningún problema, en cambio, con M2 habría que crear una figura especialmente para ello.

Como conclusión a este apartado, podemos decir que hay que tener en cuenta los resultados que se quieren obtener, pero ambos métodos son igual de buenos,. Simplemente hay que tener en consideración como funciona y los resultados que podemos obtener con cada uno.

6.13. Texturizado y coloreado

El proceso de texturizado consiste en aplicar una imagen (la textura) sobre una malla. Para ello, es necesario indicar para cada vértice qué pixel de la imagen se debe poner en esa posición. Es decir, para cada vértice se debe asignar una coordenada 2D (que oscilan entre (0,0) y (1,1)). Este proceso se llama 'unwrap' y las coordenadas de textura se conocen como uv's.

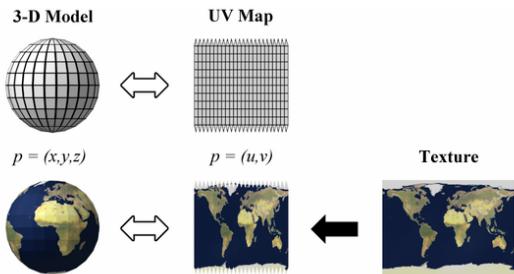


Figura 32: Proceso de asignación de una textura a una malla. Imagen obtenida de https://en.wikipedia.org/wiki/UV_mapping

Por tanto, para el texturizado lo primero que tenemos que hacer es calcular los *uv's* que nos indica cómo se coloca la textura que cojamos a nuestro objeto, que en nuestro caso es el paisaje procedural.

Es sencillo arrastrar una imagen y que ésta se quede sobre nuestro terreno. Para ajustar el tamaño del detalle proporcionado por la textura es importante que la textura sea **seamless**, es decir, que si ponemos dos copias de la textura una al lado, parezca que la textura es solo una. De esta forma, no se notan los bordes y podemos poner varias copias de la misma textura dando sensación de que los detalles de la textura son más pequeños.

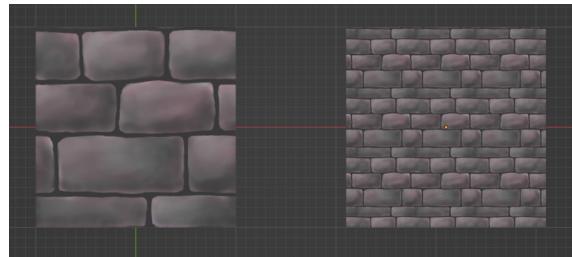


Figura 33: Textura seamless aplicada una vez (izquierda) o varias veces (derecha) para cambiar el tamaño de los ladrillos

Para obtener una textura seamless podemos hacerla nosotros, buscarla en internet o podemos coger una textura que a nosotros nos guste mucho y hacerla nosotros sin bordes. Esto se puede hacer con

el programa de edición GIMP de forma muy sencilla, ya que nos permite darle a un botón que lo hace automático por nosotros. [26]

Pero para que parezca un paisaje es preferible dar un gradiente de color y sobre éste aplicar una textura. Como muchos de los problemas de la informática, hay diferentes formas de resolverlo. La forma más común de hacerlo es usando **shaders**, que definen la forma en que el motor gráfico representa una malla teniendo en cuenta aspectos como las propiedades de los materiales, las texturas, etc.

6.13.1. Shaders

Unity cuenta con un interfaz gráfico de edición de shaders que nos permite construirlos integrando todas las herramientas disponibles para crear materiales de forma bastante simple con bloques y uniones (una herramienta muy similar a los efectos de partículas vistos en la asignatura de Fundamentos Computacionales de los Videojuegos y que permite simular lluvia, nieve, etc). [20]

En la figura 34 podemos ver un ejemplo de lo que mencionaba con anterioridad. Se tiene un nodo principal en el que irán los atributos y estos se pueden modificar con nodos, que normalmente son operaciones básicas que conocemos, tales como multiplicación, suma, diferencia y atributos como vertex color, texture 2D y muchos otros. La parte de derecha nos da información adicional sobre el nodo en concreto o el grafo en general.

Tal y como mencionaba con anterioridad los shaders son una herramienta muy potente, y este proyecto solamente se hace un uso básico de esta tecnología.

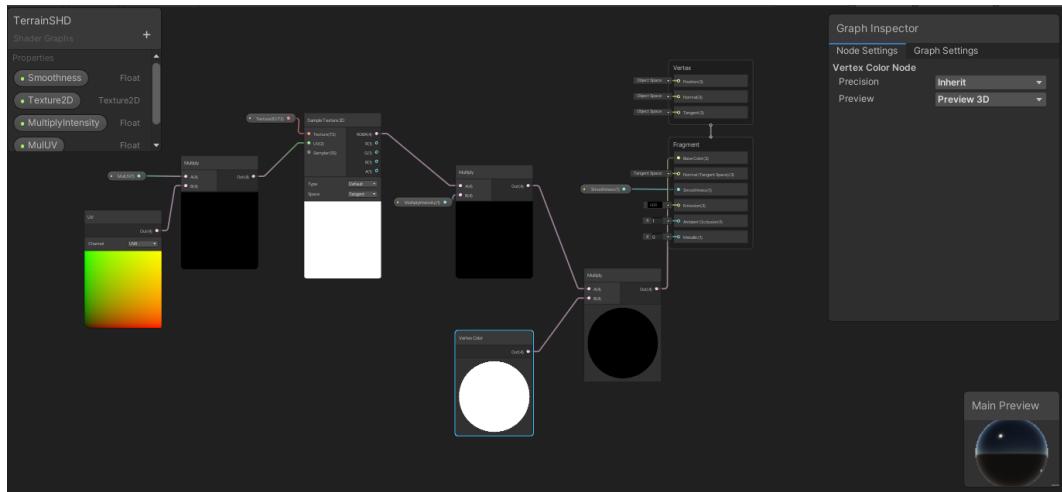


Figura 34: Shader texturizado

6.13.2. Texturizado

Para texturizar se probaron varias implementaciones sin usar los shaders, pero ninguna daba el efecto que se quería.

Por ejemplo, se intentó manejar los uvs manualmente y ajustarlos a la textura según distintos métodos, pero nunca quedaba del todo natural. Por eso se decidió usar shaders para mezclar estas texturas con colores asignados proceduralmente a los vértices (ver, por ejemplo, [12]). Para ello desarrollamos un shader básico que permite ajustar los colores de los vértices según un gradiente que se puede asignar. En la figura 35 podemos ver un ejemplo de uso de este shader

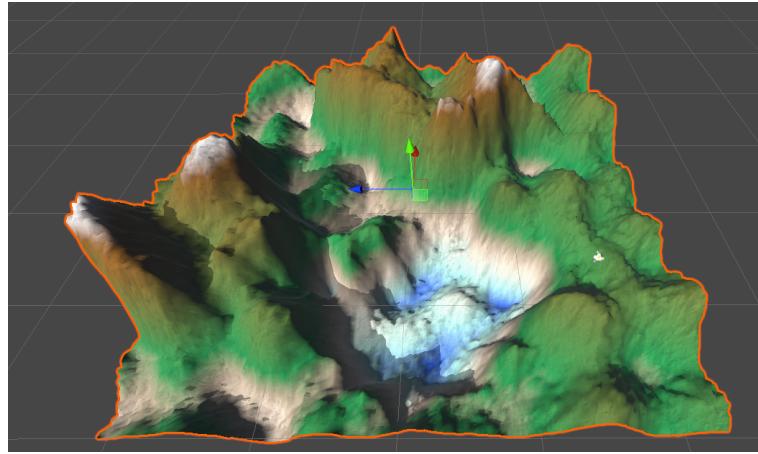


Figura 35: Ejemplo de uso de gradiente

Esta figura lleva un total de siete colores distintos definidos como un gradiente. En Unity los gradientes tienen un máximo de ocho colores, por lo que no se podrían añadir más intervalos que estos. Pero si se desease, se podrían crear un gradiente propio que sí lo permitiese. En nuestro caso, ocho valores distintos son más que suficientes para crear el coloreado deseado.

6.14. Añadir objetos

A la hora de crear un paisaje necesitaremos árboles, plantas, rocas. Son elementos decorativos, es decir, objetos que queremos añadir al terreno para que no parezca que está vacío. Estos objetos se podrían añadir manualmente, pero ya que como tenemos un generador procedural, tiene sentido que se puedan hacer de forma automática.

Si solo quisieramos añadir un solo objeto lo que tendríamos que hacer es lanzar un raycast desde una altura determinada (normalmente mucho más alta que el terreno) en dirección hacia el suelo. Cuando choque con el terreno, se coloca el objeto en el punto de colisión. En mi caso este método tiene una serie de **flags** que se utilizan para indicar cómo se quiere colocar el objeto en el suelo.

Estos flag permiten, por ejemplo, indicar si se quiere rotar el objeto en dirección al suelo, si se quiere cambiar el tamaño del objeto entre dos rangos de forma aleatoria, entre otros. Este método se va a denominar **PositionRaycast** que abreviaré como **PR**.

Vamos a ver de qué maneras se pueden añadir objetos al terreno:

6.14.1. Añadir objetos de forma totalmente aleatoria

Una forma sencilla es añadir objetos sobre el terreno de forma totalmente aleatoria, es decir, se genera unos números aleatorios y se coloca el objeto sobre el punto que cae. La idea es bastante sencilla ya que basta con generar una lista de vectores en la que sus variables x y su variable z tengan valores aleatorios entre el mínimo y el máximo del eje x y el eje z .

Una vez tenemos la lista de vectores para cada vector se lanza el método PR. En el punto que se produzca la colisión se añade el objeto (con los modificadores que se quieran). En la figura 36 se puede ver un ejemplo de este proceso.

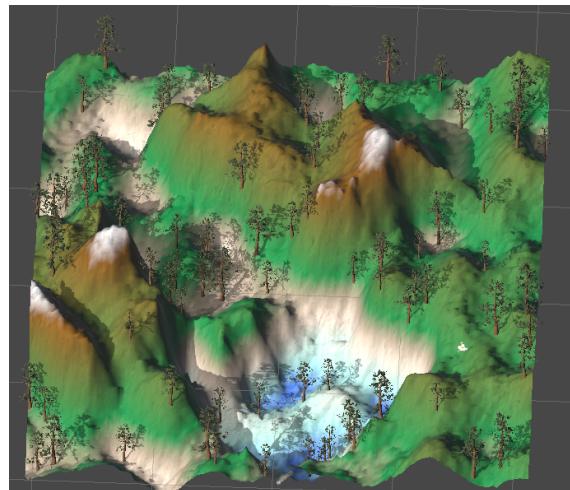


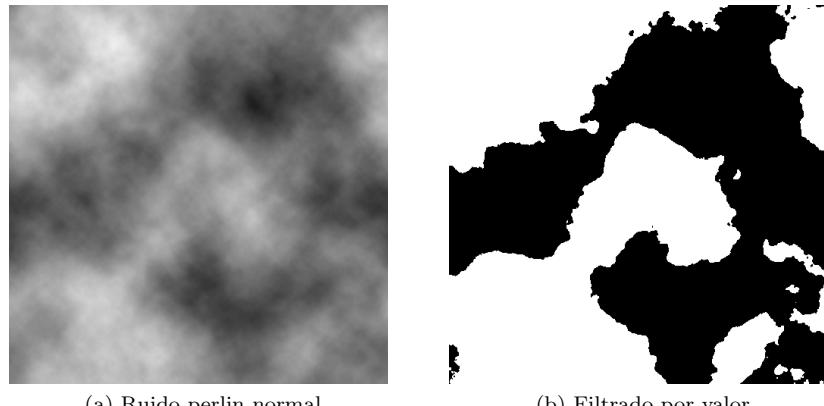
Figura 36: Árboles añadidos de forma aleatoria

Se puede observar que hay ciertos árboles que quedan al nivel del mar. Para evitarlo, contamos con un modificador que, si se aplica, se puede conseguir que solo se añadan objetos si el punto de choque está a cierta altura.

6.14.2. Añadir objetos con ruido Perlin

Es posible que queramos añadir los objetos de forma aleatoria, pero que parezca más natural de lo conseguido en el ejemplo anterior. Una forma de hacerlo es con ruido Perlin. La idea se basa

en establecer valores límite para cada posición que indiquen si es más o menos probable que se coloquen objetos en cada posición.



(a) Ruido perlin normal

(b) Filtrado por valor

Figura 37: Explicación añadir objetos con ruido Perlin

Para entenderlo de forma sencilla, podemos mirar una representación en la figura 37. Cada vez que intentamos colocar un objeto calculamos un número aleatorio. El objeto se crea solamente si este número es menor que el valor límite de la posición escogida. Los píxeles negros (valor 0) serían aquellos donde sería imposible colocar objetos y aquellos blancos donde sería más probable.

Podemos ver un ejemplo de este uso en la figura 38. De esta forma podemos añadir objetos de una forma aleatoria pero más natural y controlada.

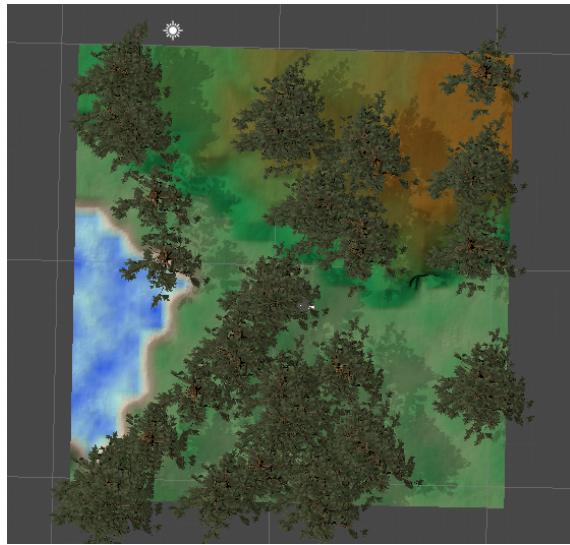


Figura 38: Árboles añadidos con Perlin Noise

6.14.3. Modificadores al añadir los objetos

Como hemos comentado se pueden añadir objetos de dos formas distintas y para cualquiera de ellas se pueden aplicar modificadores. En este apartado se van a recalcar qué modificadores se han implementado y una breve explicación de cada uno de ellos.

- **Rotar dirección terreno:** Si el terreno está inclinado, el objeto rota acorde para que esté en la misma dirección que el terreno.
- **Evitar colisiones:** Si no queremos que haya dos objetos cercanos uno a otros se pueden evitar las colisiones entre estos.
- **Añadir objetos por alturas:** Igual que en las herramientas anteriores, se pueden añadir objetos solo en un rango de altura.
- **Controlar el número de rayos que se van a lanzar:** Para añadir más o menos objetos.
- **Escalar los objetos aleatoriamente entre dos valores:** Permite dar mayor aleatoriedad controlando la escala de los objetos que se van a añadir.
- **Rotar en el eje Y aleatoriamente:** Se rota el objeto alrededor de su eje Y de forma aleatoria.
- **Añadir objetos con un objeto vacío padre:** En vez de añadir siempre el mismo objeto, permite añadir uno de entre una colección de ellos. Si, por ejemplo, se quiere generar un bosque podemos poner un objeto padre y varios hijos que serán distintos tipos de árboles. Cada vez que se va a colocar un árbol se elige aleatoriamente de entre esta colección.

6.15. Características adicionales

Todos los paisajes necesitan tener cierta parte estética. Lo bueno del motor de Unity, es que no se necesita ser un artista para conseguir resultados aceptables aunque, obviamente, alguien que maneje de la teoría de color y conozca toda la profundidad del arte hará los paisajes mucho más bonitos que el nuestro en cuanto a la sección estética.

En este apartado se han implementado ciertos apartados estéticos y además otro tipo de herramientas que no son generación procedural de paisajes, pero sí están relacionados fuertemente con ello.

6.15.1. Agua

Una vez que ya tenemos toda la malla creada, nos interesa que ciertas secciones sean de agua. Para esto he modificado un shader de agua [13] para adaptarlo a mis preferencias . Podemos ver un ejemplo de uso de esta herramienta en 39



Figura 39: Lago generado con shader de agua

Se pueden modificar distintos valores tales como la profundidad, las olas, los colores de profundidad, entre otros. Los resultados pueden variar mucho tocando los valores, según se quiera personalizar. No voy a entrar mucho en los detalles de cómo funciona el shader, pero si se quiere se puede consultar la bibliografía para más información.

6.15.2. Nubes

Para las nubes, me he basado en [21] para crear mi propio shader. Se han usado dos capas de nubes para darle un doble efecto de nubes y que sea más interesante. Este tutorial junto con el anterior, me han servido para mejorar mi conocimiento de los shaders.

En la siguiente figura 40 podemos ver como se ha usado lo explicado anteriormente.

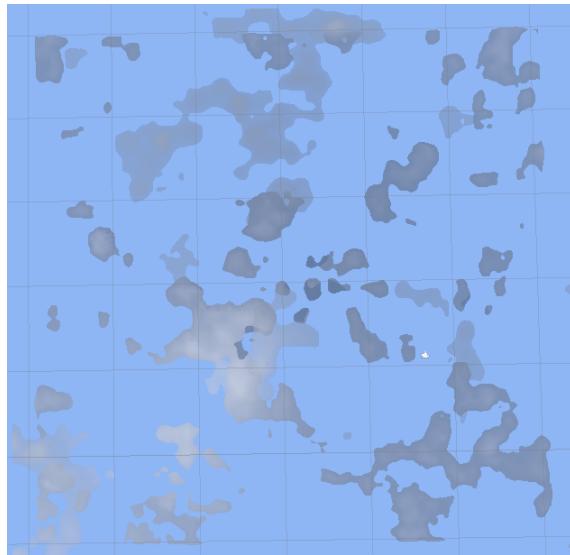


Figura 40: Doble capa de nubes

Como se puede observar se logran resultados bastante interesantes y suficientemente realistas para un juego (que no pretenda simular un terreno hiperrealista).

6.15.3. Controlador en primera persona

Se ha implementado un controlador en primera persona para ver cómo se vería el terreno desde el punto de vista de un jugador. Para esto se han usado los conocimientos aprendidos en la asignatura de Fundamentos Computacionales de los Videojuegos de controladores de personajes y el tutorial [11].

En la siguiente imagen 41 podemos ver cómo se vería nuestro terreno en modo juego.

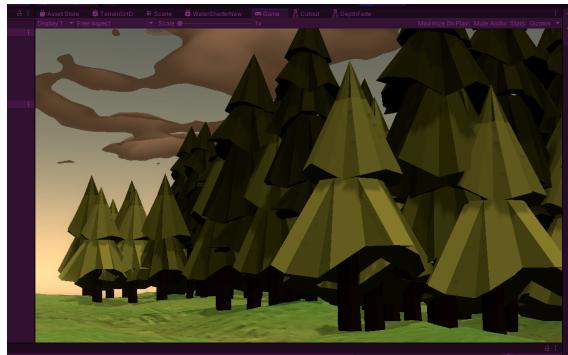


Figura 41: Controlador primera persona

6.15.4. Ciclo de día-noche, iluminación, postprocesado y cielo

La iluminación es un apartado muy importante a la hora de mejorar la apariencia de los terrenos. Para esto se ha implementado un script que permite modificar la posición del sol, según las horas [23]. Por otro lado, se crearon ciertas configuraciones de postprocesado basados en los conocimientos impartidos en la asignatura de Fundamentos Computacionales de Videojuegos. También se ha cambiado el color del skybox por algunos encontrados en internet o colores planos.

El resultado puede verse en la figura 42.

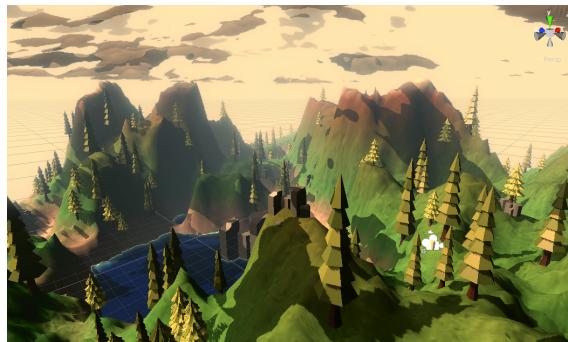


Figura 42: Escena con iluminación mejorada y postprocesado

6.16. Paisajes procedurales finales

En esta sección voy a mostrar ciertos paisajes obtenidos con todo lo aprendido y de los que estoy bastante satisfecho con los resultados obtenidos.



Figura 43: Primer escenario final



Figura 44: Segundo escenario final

7. Conclusiones y vías futuras

7.1. Conclusiones

Una vez visto los resultados que se pueden obtener y todo el aprendizaje que ha conllevado este proyecto puedo decir que personalmente estoy bastante orgulloso.

Esto se debe a que se han conseguido las mayorías de las funciones que se vieron en la tabla inicial, desde la generación de mallas, el ruido Perlin, añadir los objetos, además de aprender el editor de scripts.

También he aprendido multitud de algoritmos necesarios para que las herramientas de dichos generadores funcionan, como pueden ser la detección de vértices dentro de objetos.

Por otro lado, he extendido mi conocimiento sobre Unity y me siento muy cómodo usándolo en mi día a día. Por otro lado, he disfrutado mucho del proceso de creación y cuando me puse por primera vez en contacto con Francisco Guil nunca hubiera esperado llegar al punto donde estoy ahora mismo.

En un futuro tengo pensado mejorar mi proyecto y añadirle ciertas mejoras que se podrán ver a continuación en la sección de vías futuras.

7.2. Vías futuras

Tal y como he mencionado varias veces a lo largo de esta memoria, para hacer un generador procedural profesional tales como los enseñados en los primeros apartados se necesitaría un equipo entero de ingenieros además de otro equipo adicional para el apartado estético. Pero creo que este proyecto es una base sólida para un futuro proyecto.

Hay varias herramientas que con el tiempo suficiente me hubiera gustado poder implementar, tales como pueden ser:

- Generación de planetas a partir de la malla: Esta idea se propuso a principios del proyecto, pero por falta de tiempo no se hizo. La idea es poder pasar la malla a un planeta, de forma que podamos tener planetas procedurales.
- Algoritmos de erosión: Esta herramienta serviría para generar ríos realistas y otro tipo de cuevas.
- Mejorar el apartado estético: El apartado estético se podría mejorar de muchas maneras distintas, pero también se necesitaría a alguien que esté más conocimientos sobre arte. Se podrían añadir mejores post-procesados, shaders de nieve, entre otros.
- Generación de vida y npcs que interactúen con el entorno: Una vez que ya tenemos nuestro entorno estaría bien poder añadir cierta vida al terreno, tales como animales, incluso aldeas

de forma procedural.

Y muchísimas otras cosas, porque tal y como se puede ver, la generación procedural es un proceso continuo de aprendizaje y desarrollo que nunca se acaba.

Referencias

- [1] Asset store.
<https://assetstore.unity.com/>.
- [2] Beneath apple manor.
https://en.wikipedia.org/wiki/Beneath_Apple_Manor.
- [3] Blender.
<https://www.blender.com>.
- [4] Editor scripting.
<https://learn.unity.com/tutorial/editor-scripting#5c7f8528edbc2a002053b5f6>.
- [5] Minecraft.
<https://www.minecraft.net>.
- [6] Spelunky.
<https://spelunkyworld.com/original.html>.
- [7] Techraptor.
<https://techraptor.net/gaming/news/unity-is-most-popular-steam-game-engine-by-far>.
- [8] Unity.
<https://www.unity.com>.
- [9] Unreal engine.
<https://www.unrealengine.com/en-US>.
- [10] Eric Barone. Stardew valley, mines procedural generation.
<https://www.stardewvalley.net/mini-dev-update-5/>. 25 Mayo 2013.
- [11] Brackeys. First person movement in unity - fps controller.
https://www.youtube.com/watch?v=_QajrbabyTJc&t=1257s.
- [12] Brackeys. Mesh color.
<https://www.youtube.com/watch?v=lNyZ9K71Vhc>.
- [13] Gius Caminiti. Shader de agua estilizada con unity shader graph.
<https://www.youtube.com/watch?v=jRuCQnp78gk>.
- [14] Team Cherry. Hollow knight.
<https://www.hollowknight.com/>.
- [15] Kate Compton. Practical procedural generation for everyone.
https://www.youtube.com/watch?v=WumyfLEa6bU&ab_channel=GDC.
- [16] Nathaniel Doldersum. Terraincomposer 2.
<http://www.terraincomposer.com/terraincomposer-2-2/>.

- [17] Miguel Frade, Francisco Fernandez de Vega, and Carlos Cotta. Automatic evolution of programs for procedural generation of terrains for video games. *Soft Computing*, 16(11):1893–1914, 2012.
- [18] Battle State Games. Escape from tarkov.
<https://www.escapefromtarkov.com/>.
- [19] Hello Games. No man's sky - procedural generation.
https://nomanssky.fandom.com/wiki/Procedural_generation.
- [20] Joseph Hocking. Shaders.
<https://www.raywenderlich.com/5671826-introduction-to-shaders-in-unity>.
- [21] Gabriel Aguiar Prod. Unity shader graph - clouds tutorial.
<https://www.youtube.com/watch?v=xxhvUyvIH6s>.
- [22] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.
- [23] Probably Spoonie. Unity day and night.
<https://www.youtube.com/watch?v=m9hj9Pd0328>.
- [24] Metaverse & GIS Technology. Terraincomposer 2.
<https://terraunity.com/TerraWorld>.
- [25] Ashutosh Viramgama. Why video games are made of tiny triangles.
<https://ashutoshviramgama.com/tiny-triangles-are-used-to-make-video-games-instead-of-square-why>
- [26] Vicki Wenderlich. How to create seamless texture in gimp.
<https://www.gameartguppy.com/tutorial-how-to-create-a-seamless-texture-in-gimp/>.
- [27] Wikipedia contributors. Perlin noise — Wikipedia, the free encyclopedia.
https://en.wikipedia.org/w/index.php?title=Perlin_noise&oldid=1087476430, 2022.
- [28] Procedural Worlds. Gaia pro - rapid world creator.
<https://www.procedural-worlds.com/products/professional/gaia-pro/>.