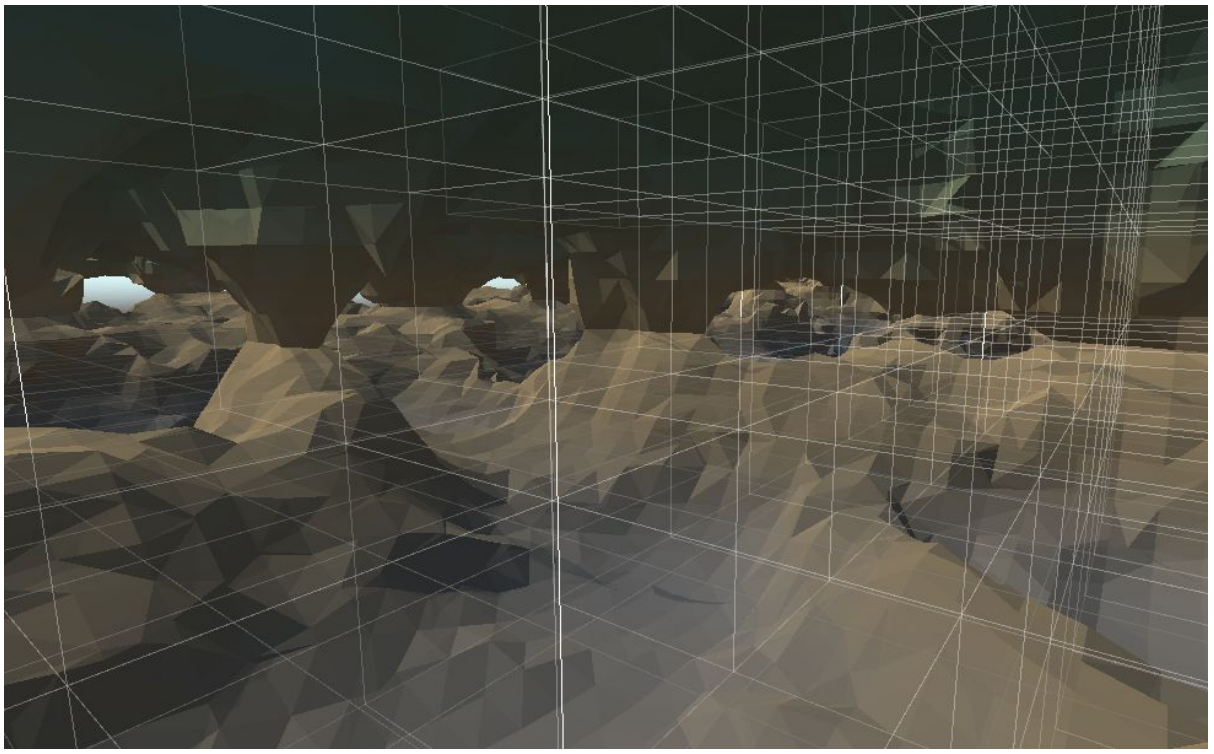
Generación procedural con unity



Índice

Introducción

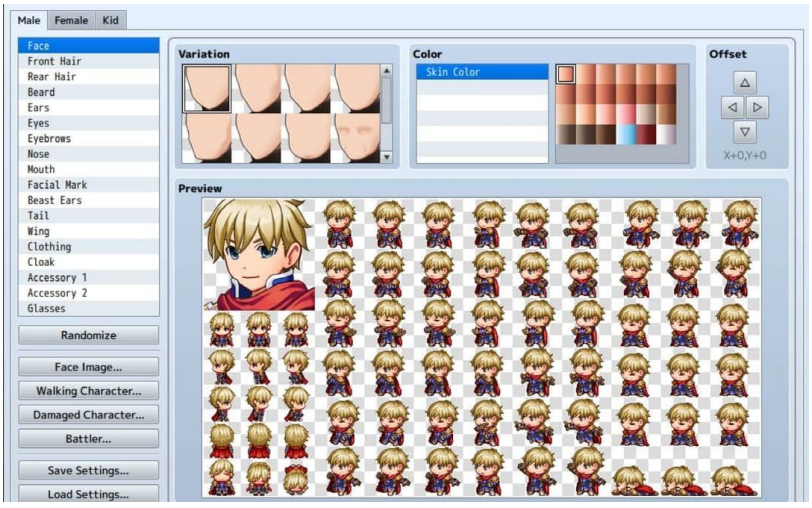
Para plantear este proyecto, se ha tenido que empezar por entender varios puntos:

- ¿Qué es la generación procedural?
- ¿Cómo se crea?

Según el proyecto fue avanzando, estas preguntas se irán desarrollando y subdividiendo.

¿Qué es la generación procedural?

La generación procedural es una técnica por la cual se genera algo (objetos, personajes, planetas, terrenos, historias) a base de números aleatorios. Dicho así no suena muy difícil, solo es coger un generador de números aleatorios y asignarlos a lo que quieras hacer. Esto en principio funciona, puedes hacer un generador procedural de personajes en base así. Si tienes un modelo base, 10 gorros, 10 camisetas y 10 pantalones, puedes generar un numero de 3 cifras y según cual sea cada una de las cifras, definir que prenda de ropa lleva el modelo.

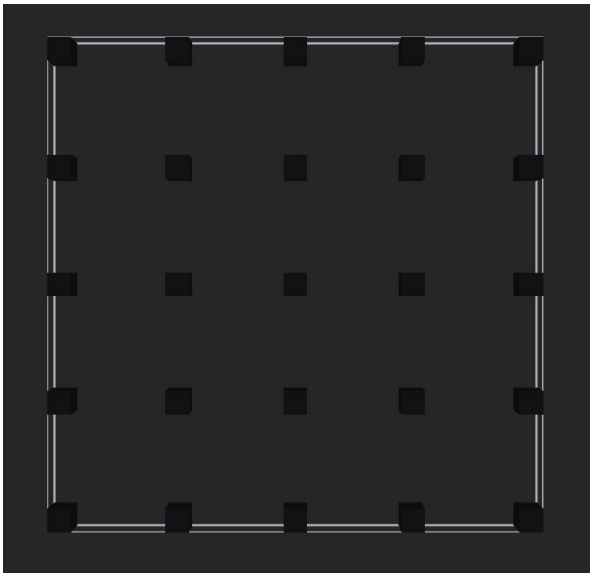


Este es un ejemplo del creador de personajes de RPG Maker, el cual funciona con una lista de elementos creados con antelación que se van seleccionando para crear personajes diferentes. Esto se podría automatizar mediante un simple generador de números aleatorios y tener miles de personajes diferentes.

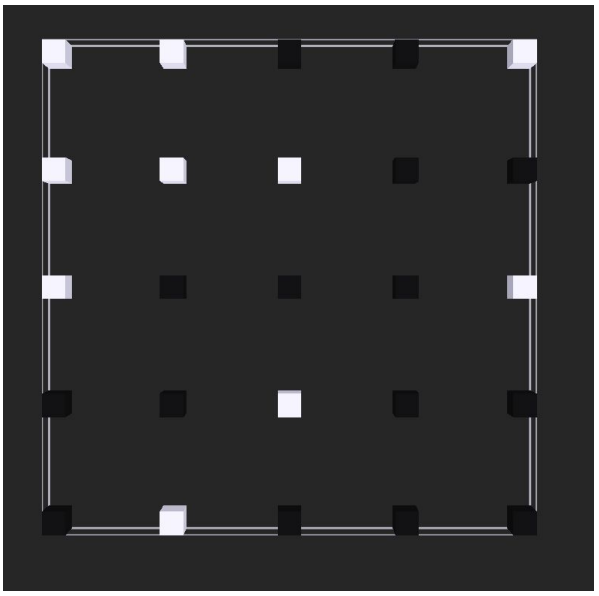
El problema no viene ahí. El problema es que esto solo funciona cuando lo que pretendes generar tiene una sola dimensión. Si con esta técnica quisieras generar formas geométricas planas, lo cual ya seria 2D, te encontrarías con el primer problema.

JO261072		Aique Montes Maestre

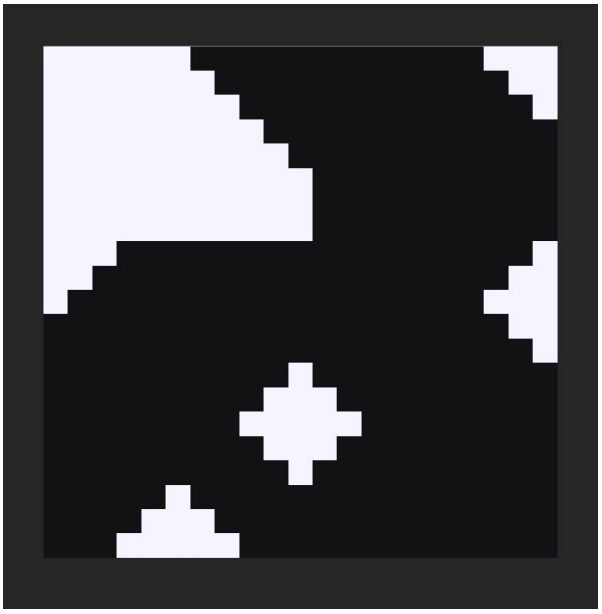
Supongamos un plano con 25 puntos definidos:



Ahora generemos un numero entre 0 y 1 para cada uno de ellos, y pintémoslos de blanco si es 1.



Si quisiéramos generar formas geométricas con eso, por ejemplo, uniendo puntos blancos adyacentes, nos saldría esto.

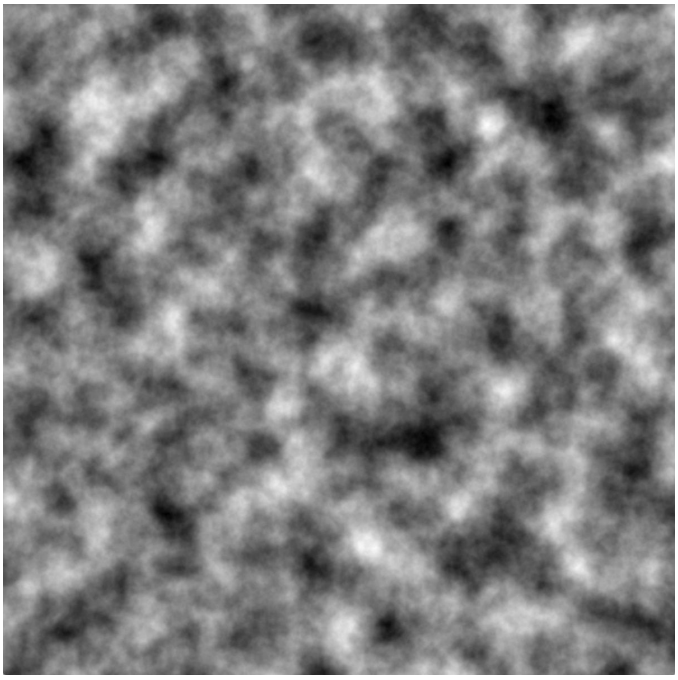


Claramente, esto no es lo que se buscaba. Hay puntos blancos sueltos que no están unidos a ninguna otra figura, la parte en sí que podría aparentar ser una figura es más bien una forma abstracta. Esto se debe a que esta técnica de generación procedural no tiene coherencia, le da igual cual es el numero asignado de cualquiera de los demás cuadrados, va a generar un numero sin importar nada más. Esto, en un entorno en el cual solo buscas saber si va a tener un gorro rojo o azul, no hay problema, puesto que te valen los dos y solo le estas pidiendo que te elija uno. Pero en un entorno como este en el que para formar una forma se requiere de un cierto grado de coordinación, el método no funciona correctamente.

Y entonces aquí llegamos a la siguiente pregunta:

¿Cómo se crea?

La respuesta a esto es ruido. Se necesita un ruido bidimensional. Dicho fácil y rápido, esto sería un generador de números aleatorios bidimensional. De forma visual, el ruido es algo como esto:



Aquí se puede apreciar claramente que ya no todo es tan aleatorio, se forman cúmulos y agrupaciones dándole más coherencia visual, sin dejar de ser aleatorio. Y en este punto es en el que empiezo la parte más practica del proyecto.

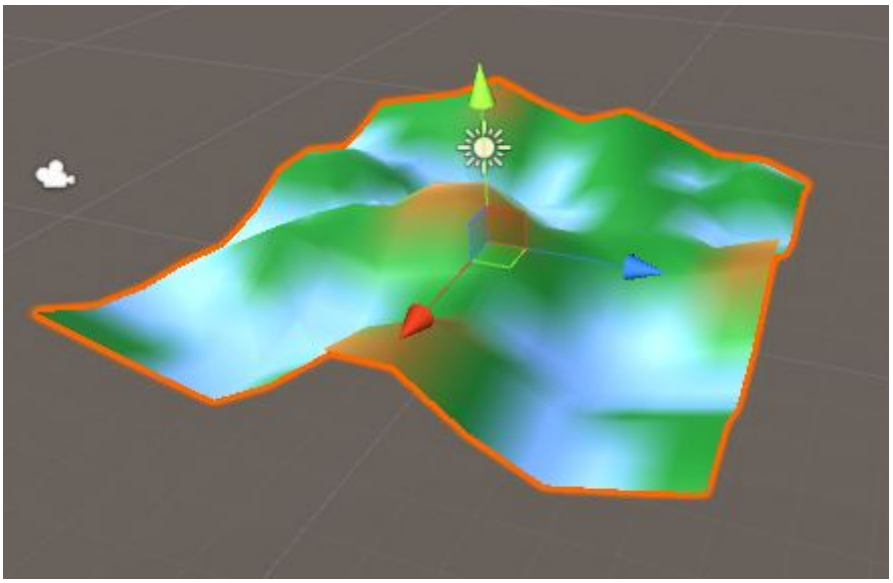
Generación de un terreno

Para esta parte realmente lo único que hice es seguir un tutorial encontrado por internet. No me salí mucho de él, y, de hecho, ni siquiera lo complete, ya que lo utilice al modo de familiarizarme un poco con la metodología y como introducción. En el proyecto de unity se corresponde con la escena llamada procedural.

El tutorial comienza con la generación de ruido de Perlin. Este es un tipo de ruido con una cierta continuidad que utiliza una fórmula matemática creada por Ken Perlin. El resultado es algo parecido a la imagen del ruido de arriba.

```
public float[,] GenerateNoiseMap(int mapDepth, int mapWidth, float scale) {
        // create an empty noise map with the mapDepth and mapWidth coordinates
float[,] noiseMap = new float[mapDepth, mapWidth];

for (int zIndex = 0; zIndex < mapDepth; zIndex ++) {
for (int xIndex = 0; xIndex < mapWidth; xIndex++) {
                    // calculate sample indices based on the coordinates and the scale
float sampleX = xIndex / scale;
float sampleZ = zIndex / scale;

                    // generate noise value using PerlinNoise
float noise = Mathf.PerlinNoise (sampleX, sampleZ);

noiseMap [zIndex, xIndex] = noise;
}
}

return noiseMap;
```

Con esto, lo que se debe hacer después es asignarle a cada color una altura. Para ello, primero creamos un objeto en unity al que le daremos el relieve. Además, a cada relieve también le daremos un color. Por ejemplo, a la parte más baja será azul para representar que es el mar. La más alta será marrón y la intermedia será verde. Realmente todo esto es un proceso más largo del que se cuenta aquí, pero no quiero llenarlo todo de capturas de código cuando ya hay un tutorial, cuyo link estará en la bibliografía que lo explica igual o mejor.

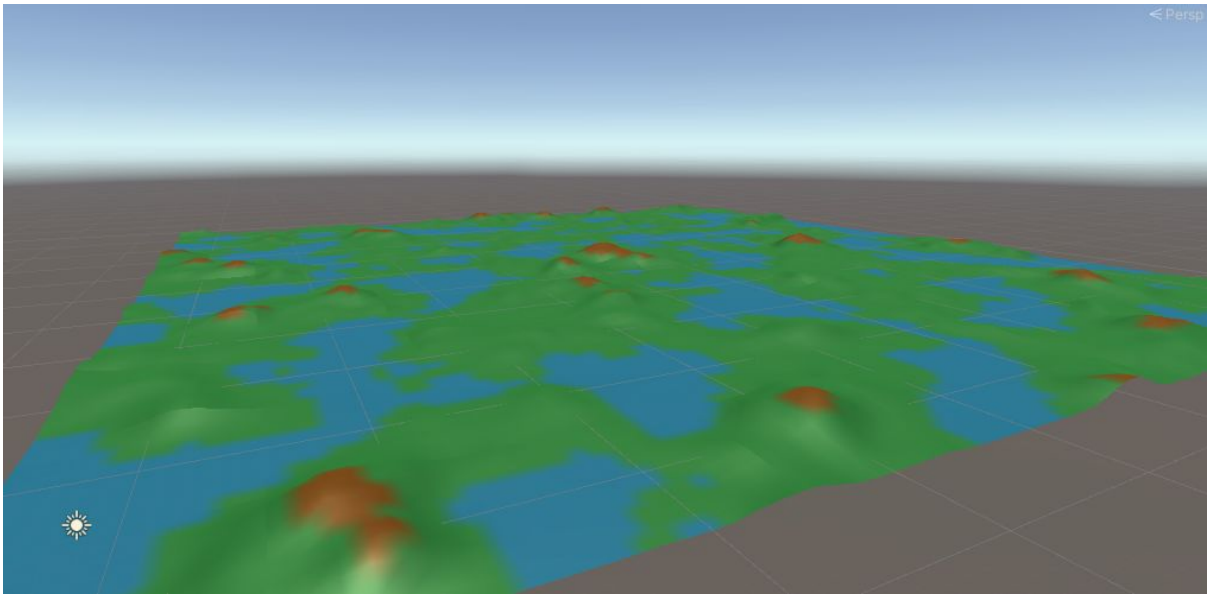


Con todo eso hecho, acabamos con algo así, que si bien, se puede apreciar claramente lo que es, no acaba de ser lo que buscábamos. Hay muy poca resolución en el terreno, que varía a saltos y de formas extrañas, la zona definida como mar sube colina arriba, nada es plano, sino que es una transición clara entre puntos.

Para solucionar esto hay que normalizarlo. Para ello crearemos diferentes olas de generación. Esto significa que generaremos ruido encima de ruido con diferentes semillas y frecuencias de forma que el relieve se suavice y se haga más consistente. Para eso, cambiamos la generación de ruido de esta forma:

```
public float[,] GenerateNoiseMap(int mapDepth, int mapWidth, float scale, float offsetX, float offsetZ, Wave[] waves)
{
    // create an empty noise map with the mapDepth and mapWidth coordinates
    float[,] noiseMap = new float[mapDepth, mapWidth];

    for (int zIndex = 0; zIndex < mapDepth; zIndex++)
    {
        for (int xIndex = 0; xIndex < mapWidth; xIndex++)
        {
            // calculate sample indices based on the coordinates, the scale and the offset
            float sampleX = (xIndex + offsetX) / scale;
            float sampleZ = (zIndex + offsetZ) / scale;

            float noise = 0f;
            float normalization = 0f;
            foreach (Wave wave in waves)
            {
                // generate noise value using PerlinNoise for a given Wave
                noise += wave.amplitude * Mathf.PerlinNoise(sampleX * wave.frequency + wave.seed, sampleZ * wave.frequency + wave.seed);
                normalization += wave.amplitude;
            }
            // normalize the noise value so that it is within 0 and 1
            noise /= normalization;

            noiseMap[zIndex, xIndex] = noise;
        }
    }

    return noiseMap;
}
```

Con esto conseguimos un resultado mucho mejor:



Tras llegar a este punto, se acaba el primer apartado del tutorial que estaba siguiendo. Le eché un vistazo a las otras dos partes y vi que más o menos iteraban sobre lo mismo, solo que lo que hacían era añadir climas en base a mapas de ruido para temperatura y humedad, y añadir árboles.

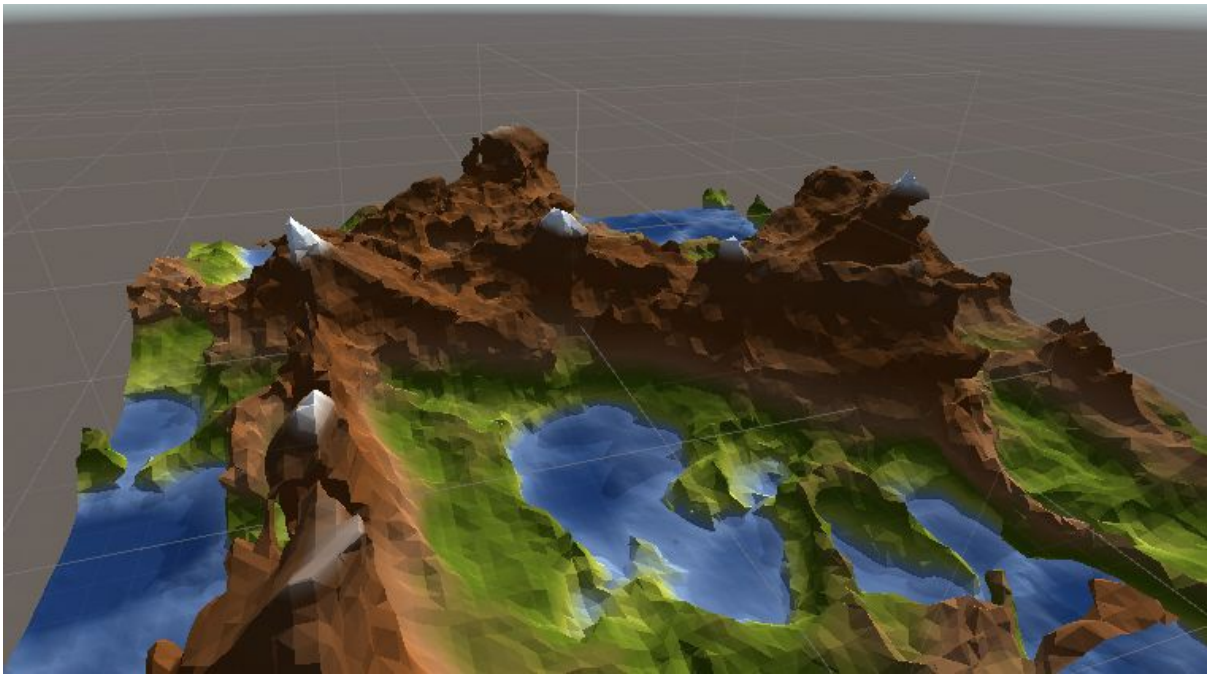
En ese momento, recordé que hacia un tiempo había visto un video en YouTube que trataba el tema de la generación procedural en unity. Así que me dispuse a buscarlo, y no tarde mucho en encontrarlo. Hago una parada aquí para recomendar el canal, toca un montón de temas relacionados con la programación en unity y tiene muchos temas interesantes.

Marching Cubes

Me llamo particularmente la atención un video llamado Marching Cubes. Así que me descargué el código del video y me dispuse a probarlo.

A partir de aquí esto se puede volver un poco teórico. Marching Cubes es una técnica de modelado 3D bastante interesante. Fue propuesta por primera vez en 1987 y es bastante simple. Consiste en definir cubos y darle un valor a cada vértice. Después se define un valor de superficie. Tras esto se crea una superficie y cualquier valor por debajo del de superficie queda por debajo de esta. Se repite eso tantas veces como se quiera, y se tiene un modelo 3D.

Usando esto, se puede utilizar métodos similares a los vistos anteriormente y dar valores a esos vértices, creando así un terreno. La ventaja de un método así es que se le puede aplicar ciertas normas por encima. Estas nos pueden permitir definir las probabilidades de forma que, por ejemplo, el suelo nunca baje de x valor, o que a partir de cierta altura se reduzca la posibilidad de puntos por debajo de la superficie. Usando este método es cómo funciona el proyecto que conseguí de generación de terrenos.



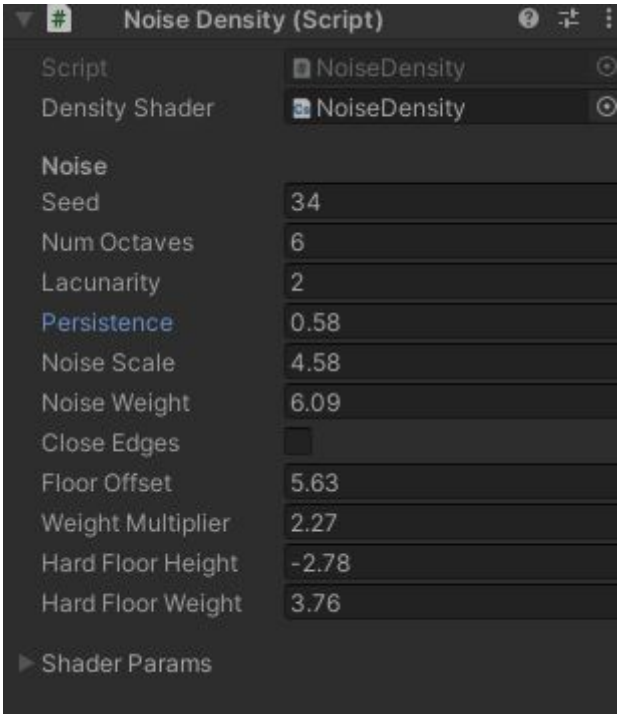
Generador procedural de cuevas

Tras estar revisando el código de este proyecto y tratando de entenderlo, decidí hacerle cambios y que ese fuera el resultado final de este trabajo. Recordé que había un grupo que en su proyecto grupal había tenido problemas con que necesitaban un asset de pago para poder hacer cuevas, así que decidí hacer un generador de cuevas procedural.

Mi primer planteamiento fue el de tratar de generar 2 veces el mapa con semillas diferentes y simplemente generar uno encima del otro e invertido. Esto no era tan fácil como pensaba y tras muchas iteraciones, además de parecerme una solución bastante mala, la descarte porque se complicaba innecesariamente y los resultados no eran del todo buenos.

Tras esto volví a revisar el código y decidí usar una mecánica de la generación que ya existía y cambiar su uso.

El generador de ruido de este proyecto tiene varias variables que permiten personalizarlo un poco.



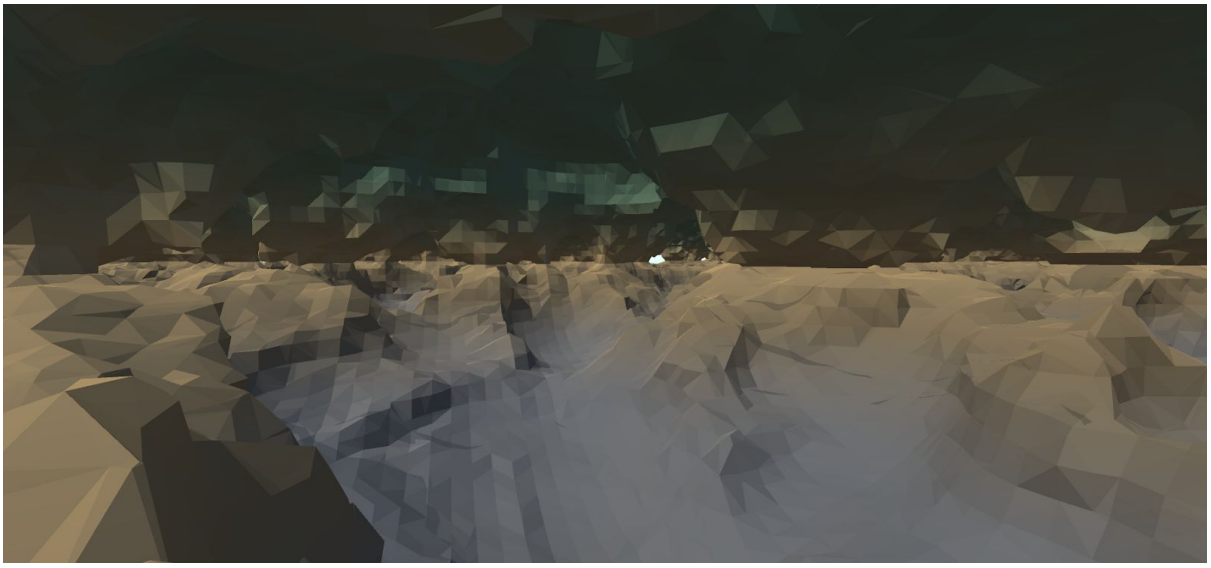
Mi decisión fue que probablemente podría usar algo parecido a lo que existe para el suelo y hacer que asegurara la existencia de un techo. Añadí a las variables ya existentes hardCelling y hardCellingWeight.

```
StructuredBuffer<float3> offsets;
int octaves;
float lacunarity;
float persistence;
float noiseScale;
float noiseWeight;
float floorOffset;
float weightMultiplier;
bool closeEdges;
float hardFloor;
float hardFloorWeight;
float hardCelling;
float hardCellingWeight;
```

Y posteriormente dividí los posibles valores en tres estados: los que están por debajo del suelo, los que están por encima del techo, y los que están en el medio.

```
float finalVal;
if(pos.y < hardCelling){
    finalVal = -(pos.y + floorOffset) + noise * noiseWeight + (pos.y%params.x) * params.y;
}
if(pos.y>=hardCelling){
    finalVal = (pos.y + floorOffset) + noise * noiseWeight + (pos.y%params.x) * params.y;
}

if (pos.y < hardFloor) {
    finalVal += hardFloorWeight;
}
if (pos.y > hardCelling) {
    finalVal += -hardCellingWeight;
}
```

Después todo lo que quedo por hacer fue retocar los valores desde el editor hasta encontrar el que más me gustara, y hacer algunos ajustes para solventar algunos errores. Tras eso, añadí un sistema de movimiento básico en VR que te permite volar para verlo todo, y apliqué las opciones que traían el proyecto para poder generar chunks nuevos. Este fue el resultado:



Bibliografía

https://es.wikipedia.org/wiki/Generación_por_procedimientos

Tutorial Generación procedural en Unity

https://gamedevacademy.org/complete-guide-to-procedural-level-generation-in-unity-part-1/

Marching Cubes

https://en.wikipedia.org/wiki/Marching_cubes
https://web.archive.org/web/20161124091408/http://ab.cba.mit.edu/classes/S62.12/docs/Lorensen_marching_cubes.pdf
https://web.archive.org/web/20190818160414/http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html

Canal de YouTube sobre Unity

https://www.youtube.com/watch?v=M3iI2l0ltbE