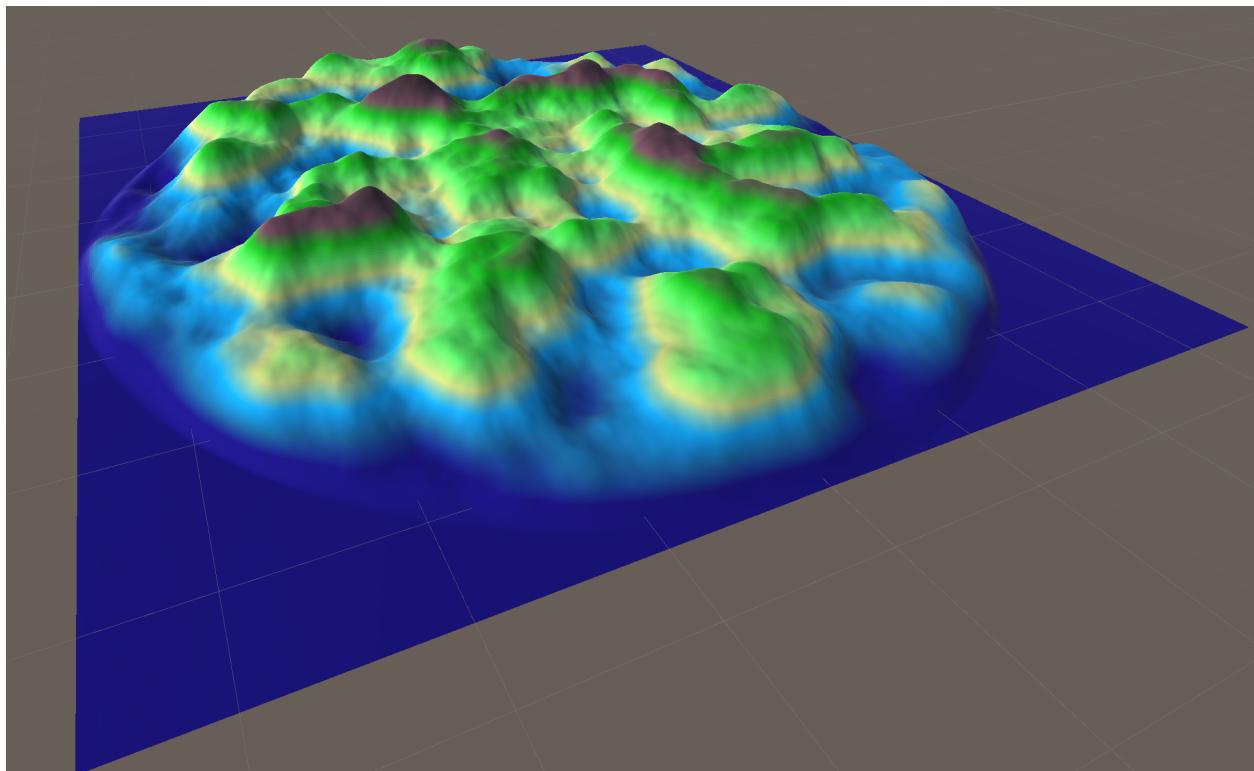


Génération procédurale



Intro :

Pour mon engagement étudiant de troisième année j'ai décidé de développer un générateur procédural de terrain en 3 dimensions.

L'intérêt de la génération procédurale est qu' une fois l'algorithme de génération fini il est possible de générer une infinité de paysages différents de manière automatique. Le but du projet est d'obtenir un algorithme capable de générer un terrain 3D régi par différents paramètres et par une graine aléatoire, que nous pourrions changer à volonté pour obtenir des terrains différents mais avec des caractéristiques semblables.

J'ai décidé d'utiliser le logiciel Unity qui est un moteur de jeux vidéo couramment utilisé dans l'industrie du jeux vidéo. Ce logiciel m'a notamment mis à disposition des outils de rendu de mesh en 3D, cependant l'entièreté du code de la génération procédurale sera fait à la main en C#, le langage utilisé par Unity.

I Principe de l'algorithme

Notre algorithme se déroulera en trois temps:

- Génération de la carte de hauteur
- Génération des textures
- Génération du mesh

Pour créer votre terrain nous allons créer un tableau/ une carte des hauteurs du terrain. Cette carte de hauteur définit la forme de notre terrain, elle doit donc être aléatoire, mais un aléatoire soumis à des paramètres pour obtenir une carte intéressante. Pour cela nous utiliserons des algorithmes de génération de bruits aléatoires. Ensuite nous créerons les textures pour colorer le terrain en fonction de sa hauteur. Nous assemblerons le tout pour créer un Mesh en 3D entièrement utilisable.

II Génération de la carte de hauteur

La première étape consiste à créer un tableau en deux dimensions (x et z) décrivant la hauteur du terrain au coordonnées (x, z). Ainsi le tableau suivant : [[0.1, 0.2], [-0.2, 1]] décrit un terrain où le point de coordonnée (0, 0) a une hauteur de 0.1 unité. Ainsi lors de la création du Mesh ce sommet aura comme coordonné en 3D (0,0,0.1). Créer un tel tableau est clairement une tâche fastidieuse, ainsi nous allons séparer cela en plusieurs petites étapes.

Nous allons utiliser des algorithmes de bruits pour générer notre carte.

Le bruit (noise in english) est une légères variations de valeur aléatoire. On le représente sous la forme d'une image 2D en nuances de gris, les zones claires sont là où le bruit est très faible, les zones sombres où il est plus élevé.

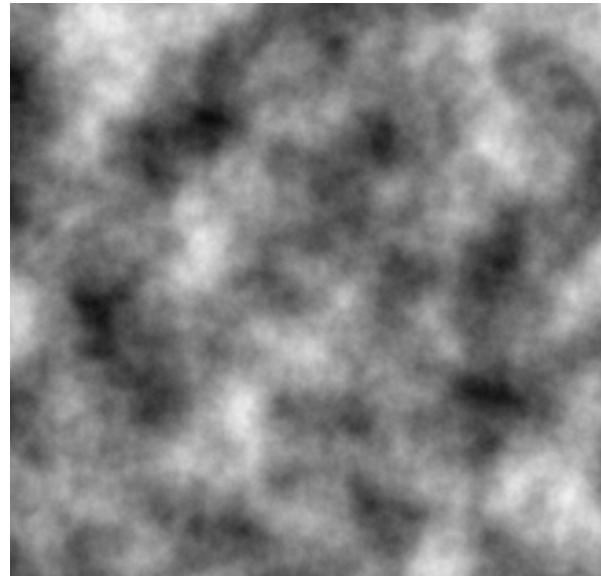
Il en existe plusieurs sorte, le plus simple est le regular noise où chaque pixel est un nombre entre 0 et 1 choisi aléatoirement:



Le problème de ce type de bruit est qu'il n'est pas continu, en effet la valeur de chaque pixel est indépendante des valeurs des pixels adjacents, ainsi avec ce type de bruit notre terrain serait complètement chaotique.

Il existe des bruits continus, nous utiliserons et présenterons ici le Perlin noise créé par le mathématicien Ken Perlin en 1983.

Contrairement au regular noise, ce bruit est bien continu ce qui est parfait pour notre génération de terrain.



Perlin noise

L'algorithme de Perlin noise est relativement compliqué et je ne l'expliquerai pas ici, cependant il existe une fonction PerlinNoise sous Unity nous permettant de l'utiliser.

Commençons à coder, je vais créer un script en C# qui s'occupera de générer la carte de bruit. Ce script contiendra une fonction qui générera la carte de bruit en fonction de la taille de la carte, de la graine aléatoire, de l'échelle du bruit (le scale) et d'un décalage (offset).

```

1   using UnityEngine;
2
3   public static class Noise
4   {
5       public static NoiseMap GenerateNoiseMap(in int mapWidth, in int mapHeight, in int seed, Vector2 scale, in Vector2 offset)
6       {
7           //vérification des valeurs pris en paramètre
8           if (scale.x < 0f)
9               scale.x *= -1f;
10          if (scale.y < 0f)
11              scale.y *= -1f;
12          if (scale.sqrMagnitude <= Mathf.Epsilon)
13              scale = new Vector2(20f, 20f);
14
15          float[,] noiseMap = new float[mapWidth, mapHeight];// la carte finale
16          Random.SetRandomSeed(seed);//On définit la graine d'aléatoire
17
18          //on décale le tout pour que le point (0,0) soit au milieu de la carte
19          //et non pas en haut à gauche.
20          float halfWidth = mapWidth * 0.5f + offset.x;
21          float halfHeight = mapHeight * 0.5f + offset.y;
22
23          for (int y = 0; y < mapHeight; y++)
24          {
25              for (int x = 0; x < mapWidth; x++)
26              {
27                  //On recalcule les coord x et y pour y appliquer le scale
28                  float sampleX = (x - halfWidth) / scale.x;
29                  float sampleY = (y - halfHeight) / scale.y;
30                  noiseMap[x, y] = Mathf.PerlinNoise(sampleX, sampleY) * 2f - 1f;
31              }
32          }
33      return new NoiseMap(noiseMap);
34  }
35 }
```

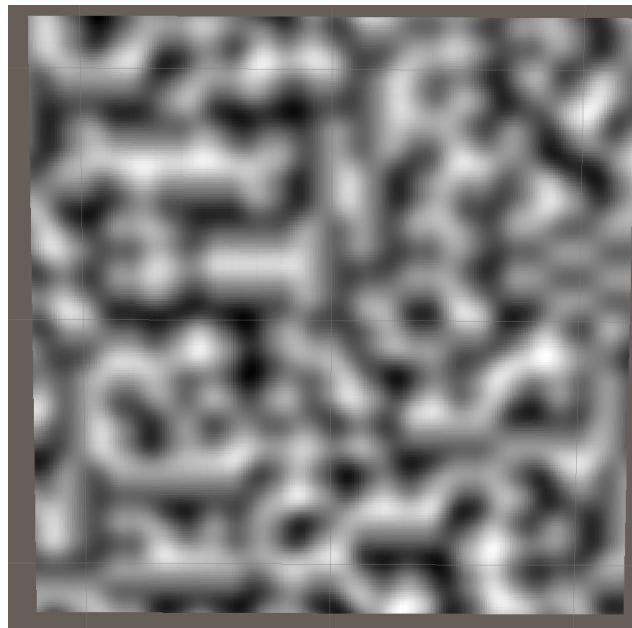
L'algorithme de perlin nous donne des valeurs entre -1 et 1, comme je souhaite utiliser des valeurs entre 0 et 1 je multiplie la valeur du bruit par 2 et j'enlève 1 pour me ramener dans l'intervalle [0,1]

Pour visualiser notre bruit je vais créer une texture en nuance de gris et l'appliquer sur un plan pour visualiser le résultat.

```

1   using UnityEngine;
2
3   //Génère la texture depuis une noiseMap
4   public static class TextureGenerator
5   {
6       public static Texture2D TextureFromColorMap(in Color[] colorMap, in int width, in int height)
7       {
8           Texture2D texture = new Texture2D(width, height);
9           texture.filterMode = FilterMode.Point;//on enlève les filtres de lissage
10          texture.wrapMode = TextureWrapMode.Clamp;
11          texture.SetPixels(colorMap);
12          texture.Apply();
13          return texture;
14      }
15
16      public static Texture2D TextureFromHeight(in float[,] heightMap)
17      {
18          int width = heightMap.GetLength(0);
19          int height = heightMap.GetLength(1);
20
21          Color[] colorMap = new Color[width * height];
22          for (int y = 0; y < height; y++)
23          {
24              for (int x = 0; x < width; x++)
25              {
26                  colorMap[y * width + x] = Color.Lerp(Color.black, Color.white, heightMap[x, y]);
27              }
28          }
29      return TextureFromColorMap(colorMap, width, height);
30  }
31 }
```

Nous obtenons ce résultat qui est correct :



Cependant je voudrais plus de détails et de subtilité dans cette carte, pour cela au lieu de créer une unique carte de hauteur nous allons en créer plusieurs que nous superposerons, nous appellerons ces cartes des octaves. Ainsi nous allons ajouter deux nouveaux paramètre, la lacunarité qui définit l'augmentation du scale des différentes octaves, ainsi la lacunarité est un nombre supérieur à 1, la persistance sera l'évolution de l'importance / du poids de chaque octaves, ainsi la persistance sera un nombre entre 0 et 1.

De plus, il faudra créer un offset différent pour chaque octave.

Changeons notre code pour pouvoir ajouter autant d'octaves que souhaité à notre noiseMap.

```
Random.SetRandomSeed(seed);
Vector2[] octaveOffset = new Vector2[octave];
float amplitude = 1f;
float frequency = 1f;
for (int i = 0; i < octave; i++)
{
    //on met un offset différent pour chaque octaves
    float offsetX = Random.Rand(-100000f, +100000f) + offset.x;
    float offsetY = Random.Rand(-100000f, +100000f) + offset.y;
    octaveOffset[i] = new Vector2(offsetX, offsetY);
}

float maxNoiseHeight = float.MinValue;
float minNoiseHeight = float.MaxValue;
```

J'ajoute ce morceau de code au début de la fonction pour initialiser les variables.

```

for (int y = 0; y < mapHeight; y++)
{
    for (int x = 0; x < mapWidth; x++)
    {
        amplitude = 1f;
        frequency = 1f;
        float noiseHeight = 0f;
        for (int i = 0; i < octave; i++)
        {
            float sampleX = (x - halfWidth + octaveOffset[i].x) / scale.x * frequency;
            float sampleY = (y - halfHeight + octaveOffset[i].y) / scale.y * frequency;
            float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2f - 1f;
            noiseHeight += perlinValue * amplitude;
            amplitude *= persistance;
            frequency *= lacunarity;
        }
        noiseMap[x, y] = noiseHeight;
        minNoiseHeight = Mathf.Min(minNoiseHeight, noiseHeight);
        maxNoiseHeight = Mathf.Max(maxNoiseHeight, noiseHeight);
    }
}
//on normalise pour avoir des nombre entre 0 et 1
for (int y = 0; y < mapHeight; y++)
{
    for (int x = 0; x < mapWidth; x++)
    {
        noiseMap[x, y] = Mathf.InverseLerp(minNoiseHeight, maxNoiseHeight, noiseMap[x, y]);
    }
}
return new NoiseMap(noiseMap);

```

Je rajoute une troisième boucle for pour faire une nouvelle itération par octave, et à chaque itérations je modifie la fréquence et l'amplitude pour faire varier l'échelle et l'importance des différents octaves. Comme j'ajoute plusieurs valeurs de bruits de Perlin, les valeurs de la carte ne seront plus obligatoirement dans [0,1] d'où le fait de stocker le minimum et maximum global de la carte pour ensuite tout normaliser.

III Génération du mesh et des textures

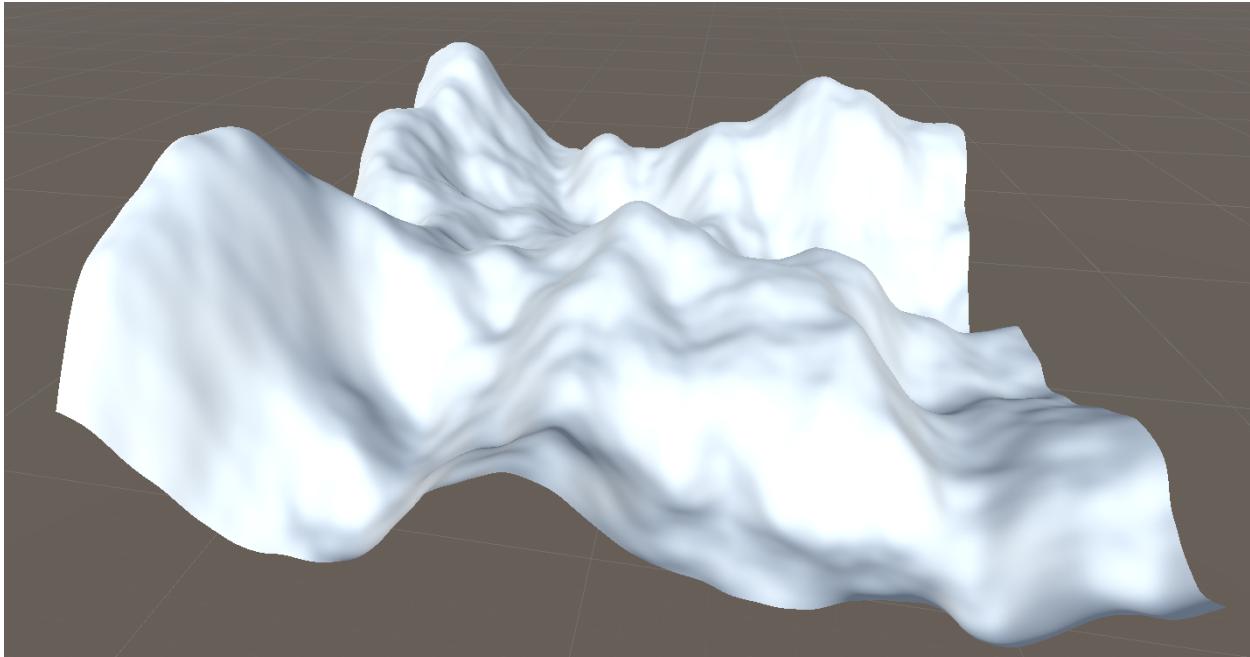
Pour pouvoir tester notre algorithme il faut convertir cette carte de hauteur en mesh, certaines extensions le font très bien mais j'ai décidé de recoder la fonction de conversion car je voulais ajouter un système de niveau de détail(LOD).

Pour créer le mesh il faut indiquer à Unity les différents triangles composant le mesh ainsi que les UV, utiliser pour appliquer des textures au mesh.

Concernant le LOD, il permet de réduire les détails de la carte pour économiser en temps de calcul, pour cela on décrémente le nombre de triangles en sautant une partie des points de la carte de hauteur, ceci est un détail, je ne rentrerai pas plus dans les détails ici.

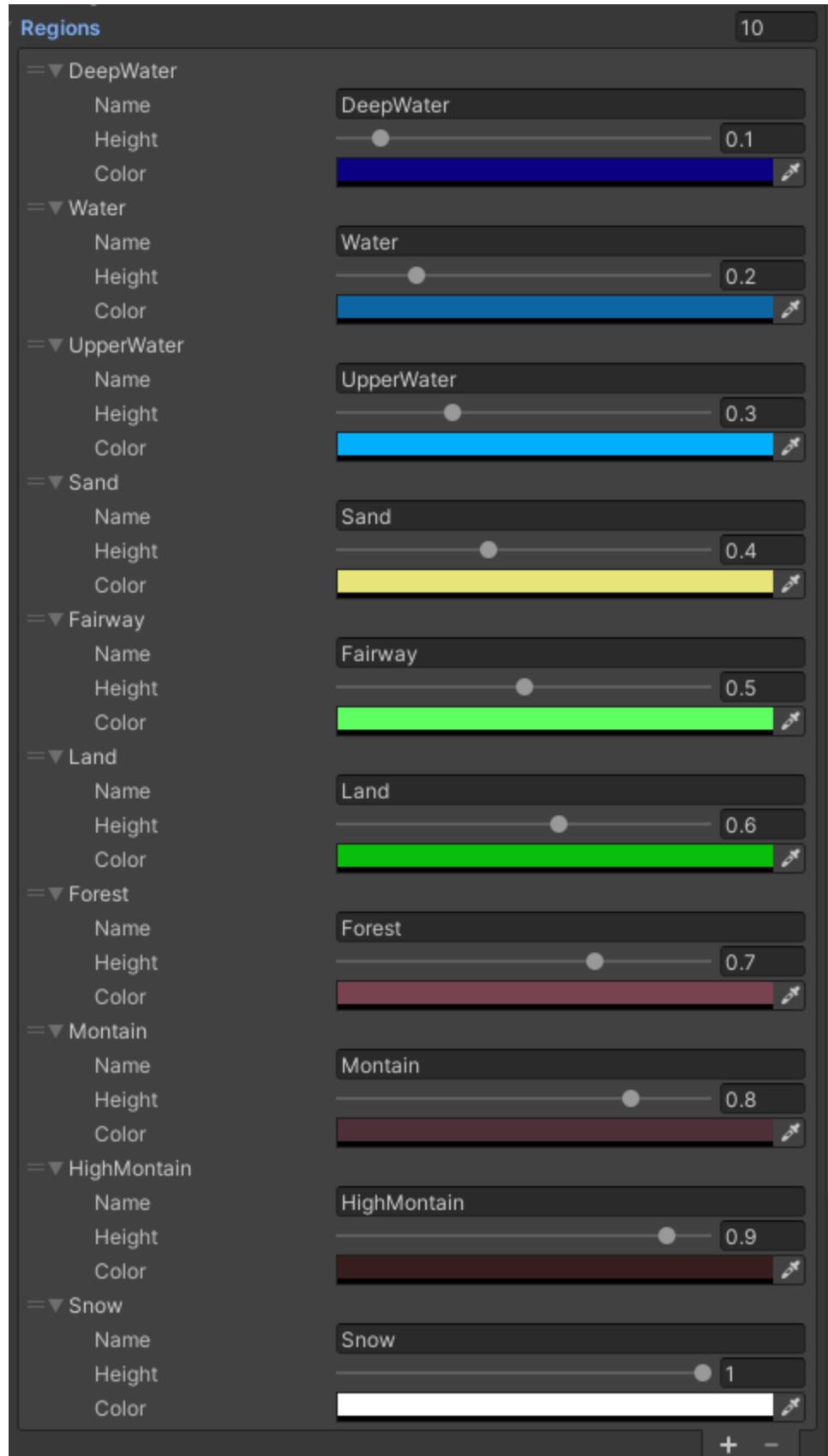
Il faut cependant multiplier les valeurs de la carte de hauteur par un coefficient multiplicateur pour obtenir un terrain avec des montagnes assez hautes.

Pour une carte de 241*241, 6 octaves, une lacunarité de 1.7 et une persistance de 0.377 j'obtiens le résultat suivant :



La forme générale semble prometteuse mais pour rendre le terrain bien plus beau il faudrait lui appliquer une texture. Pour coder cela chaque pixel de la texture sera définie en fonction de la hauteur de la carte à cet endroit.

Je décide donc de créer une liste de régions. Chaque région possède un nom, un interval de hauteur et une couleur.



Dans cet exemple de région, un point de la carte ayant une hauteur comprise dans l'intervalle $[0, 0.1[$ sera bleu foncé.

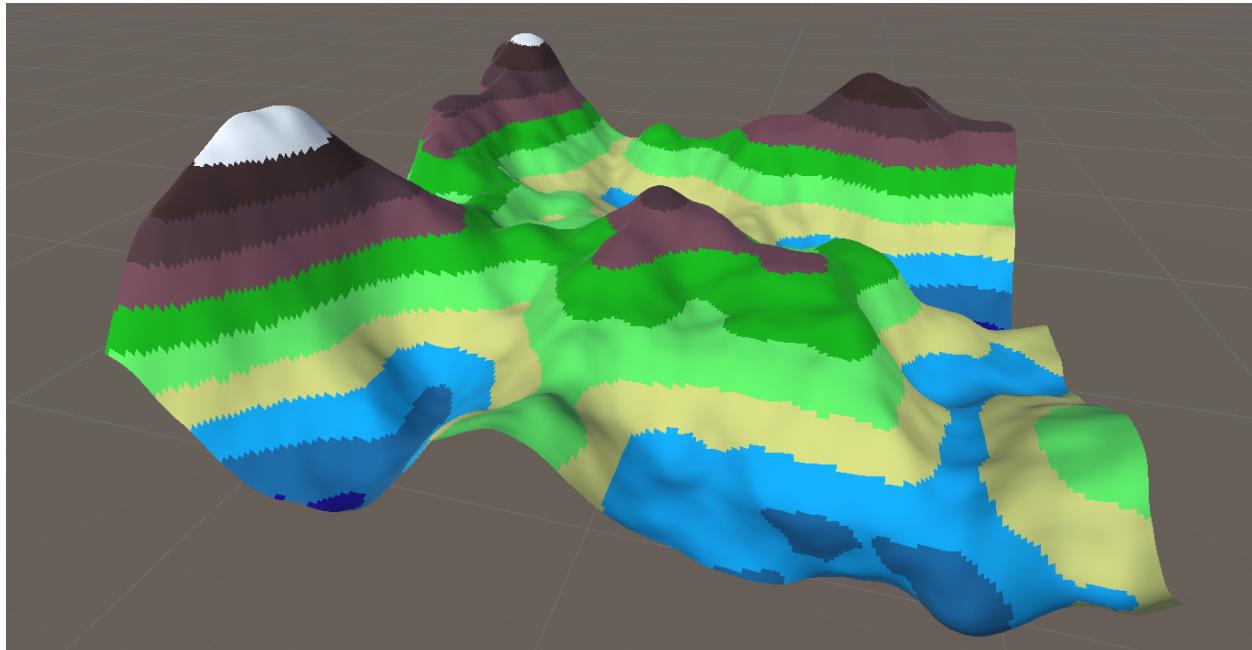
En revanche un point avec une hauteur comprise dans l'intervalle $[0.4, 0.5[$ sera jaune

Ainsi je code une fonction pour créer un tableau de couleur dépendant de la hauteur de la carte et du tableau de région.

```
Color[] colorMap = new Color[mapWidth * mapHeight];
if(useRegion)
{
    //On remplit directement la colormap
    for (int y = 0; y < mapWidth; y++)
    {
        for (int x = 0; x < mapHeight; x++)
        {
            float currentHeight = noiseChunk[x, y];
            for (int i = 0; i < regions.Count; i++)
            {
                if (currentHeight <= regions[i].height)
                {
                    colorMap[y * mapChunkSize + x] = regions[i].color;
                    break;
                }
            }
        }
    }
}
```

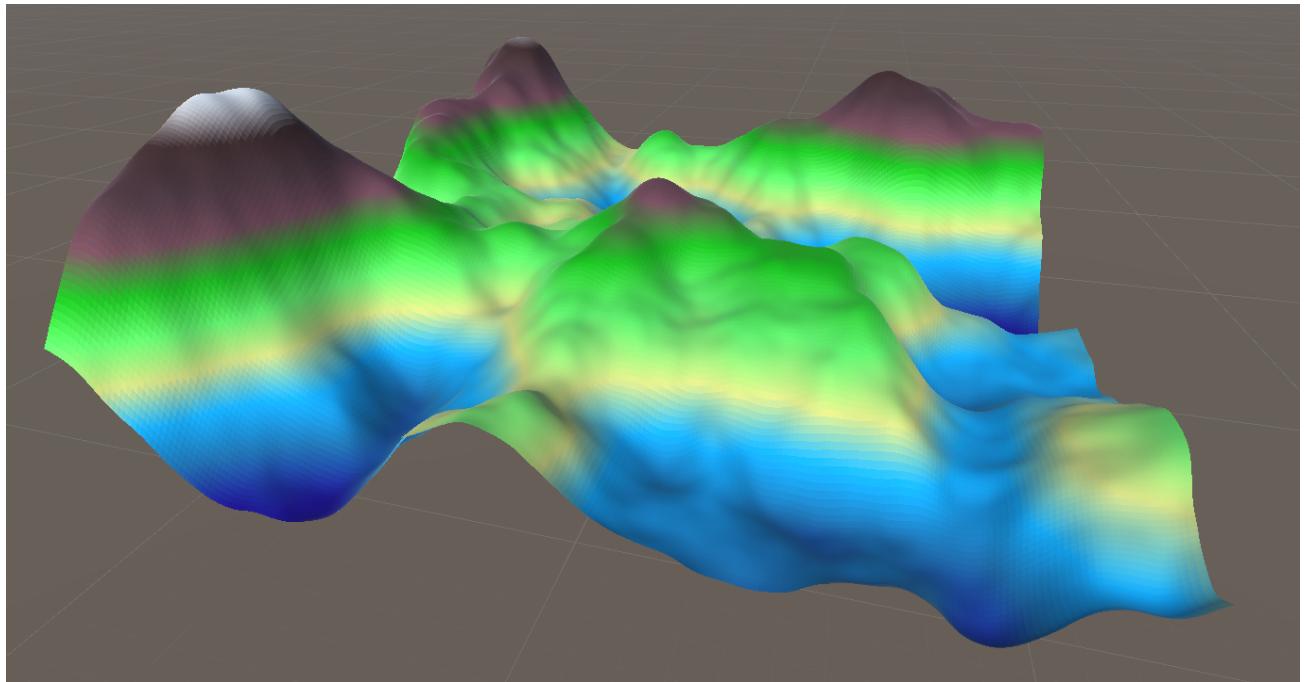
On crée ici un tableau de couleur à une seule dimension car les textures se créent avec des tableaux en une dimension

On obtient ce résultat :



Unity propose un système de gradient de couleur, cette outil permet de lisser les couleurs et ainsi d'éviter ces sauts brutaux de couleurs.

Bien sûr, je rajoute un bout de code générant automatiquement de tel gradient en fonction de la carte des régions.



Je trouve personnellement ce résultat bien plus esthétique.

Il reste cependant un problème à notre terrain au niveau de la bordure. Il faudrait idéalement que au bord du terrains la hauteur tende vers 0 pour éviter ces ruptures nettes.

Pour cela je décide de créer une carte de chute. C'est un tableau de coefficients entre 0 et 1, où ces coefficients tendent vers 0 au bord du tableau, comme ça nous pourrions multiplier chaque hauteur de la noiseMap avec le coefficient de la carte de chute.

Ainsi les points proches du bord auront des hauteurs proches de 0.

Je décide de coder deux type de carte de chute, une en forme de cercle et une autre en forme de rectangle.

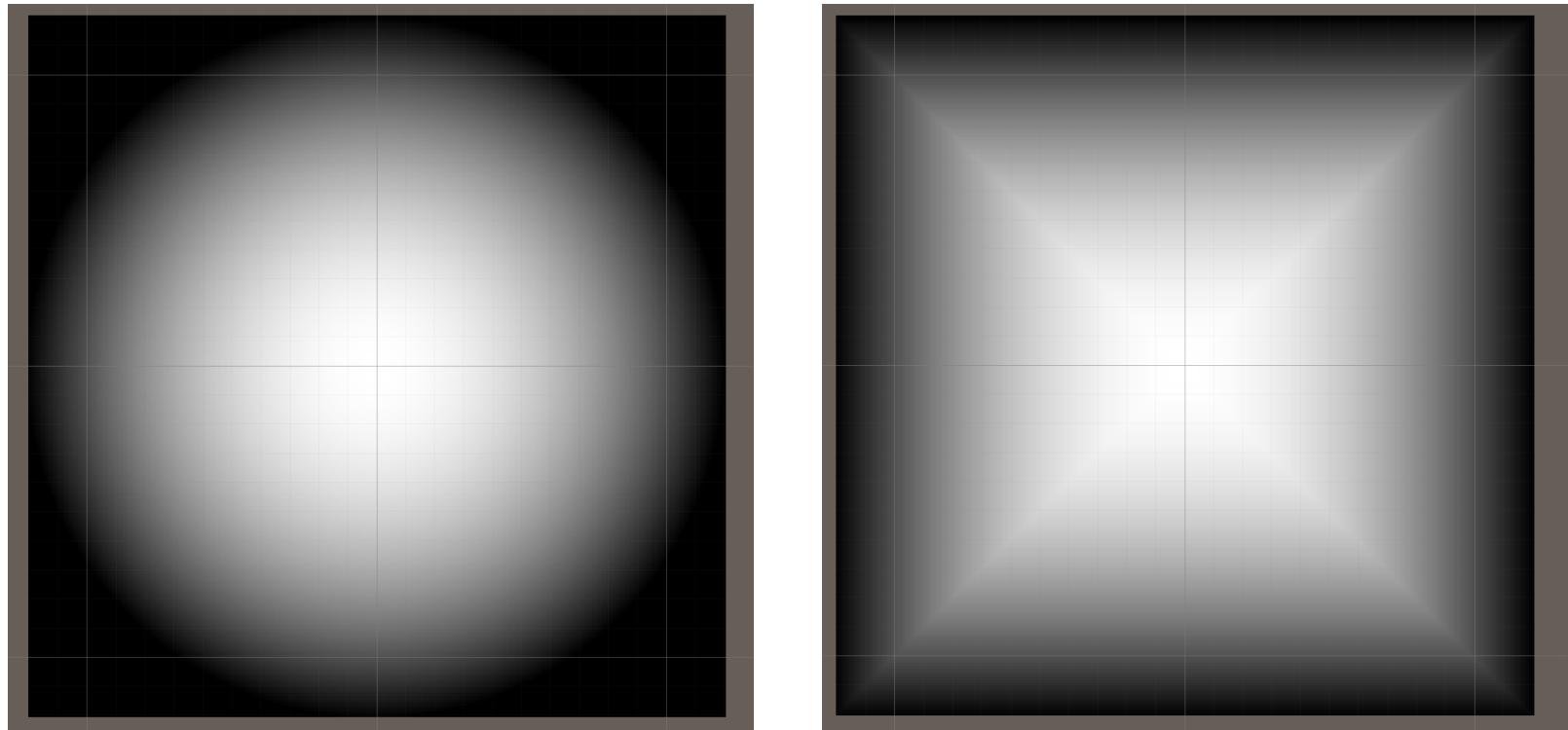
J'ajoute pour plus de personnalisation une courbe (nommée falloffCurve) permettant de modifier la répartition des coefficients de la carte de chute.

```

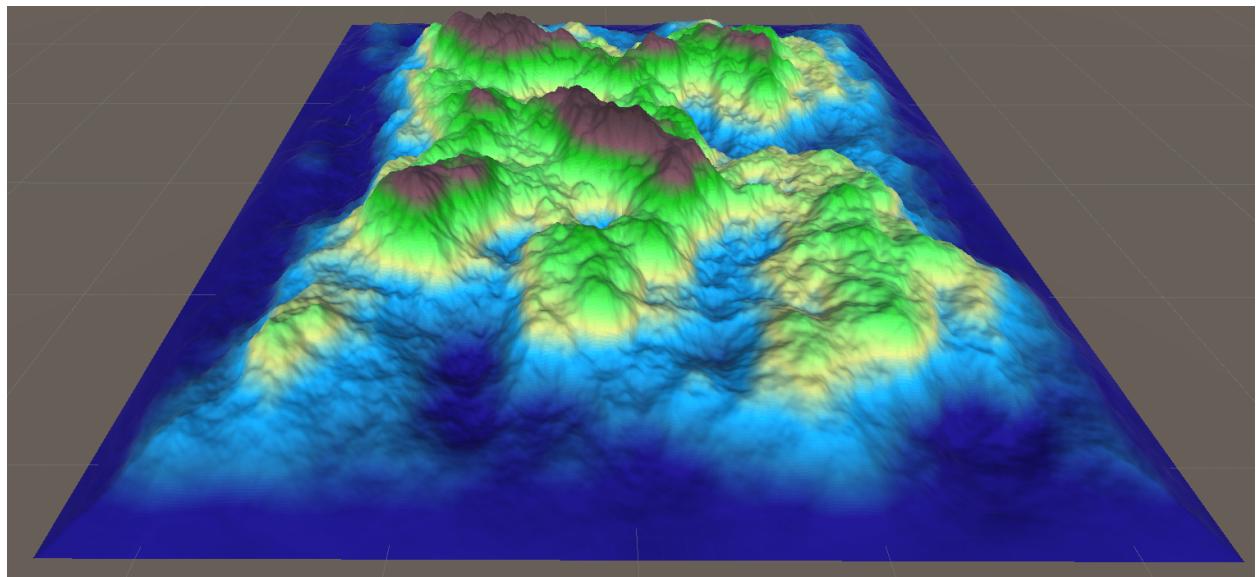
public void GenerateFallOffMap(NoiseMap noiseMap)
{
    int w = noiseMap.map.GetLength(0);
    int h = noiseMap.map.GetLength(1);
    fallOffMap = new float[w, h];
    Vector2 center = new Vector2(w * 0.5f, h * 0.5f);
    switch (fallOffMode)
    {
        case FallOffMode.circle:
            float maxDist = Mathf.Min(w * 0.5f, h * 0.5f);
            for (int y = 0; y < h; y++)
            {
                for (int x = 0; x < w; x++)
                {
                    float distNormalized = Vector2.Distance(center, new Vector2(x, y)) / maxDist;
                    float f = Mathf.InverseLerp(1f, 0f, Mathf.Clamp(distNormalized, 0f, 1f));
                    fallOffMap[x, y] = fallOffCurve.Evaluate(f);
                }
            }
            break;
        case FallOffMode.rectangle:
            float w02 = w * 0.5f;
            float h02 = h * 0.5f;
            for (int y = 0; y < h; y++)
            {
                for (int x = 0; x < w; x++)
                {
                    float distx = Mathf.Abs(x - center.x);
                    float disty = Mathf.Abs(y - center.y);
                    float distNormalized = Mathf.Max(distx / w02, disty / h02);
                    float f = Mathf.InverseLerp(1f, 0f, Mathf.Clamp(distNormalized, 0f, 1f));
                    fallOffMap[x, y] = fallOffCurve.Evaluate(f);
                }
            }
            break;
    }
}

```

Nous pouvons également visualiser la carte de chute en niveau de gris :



En jouant sur tous les paramètres j'obtiens des cartes largement perfectibles mais qui correspondent à mes attentes.



En appliquant c'est carte de chute nous pouvons obtenir des terrains ressemblant à des îles

