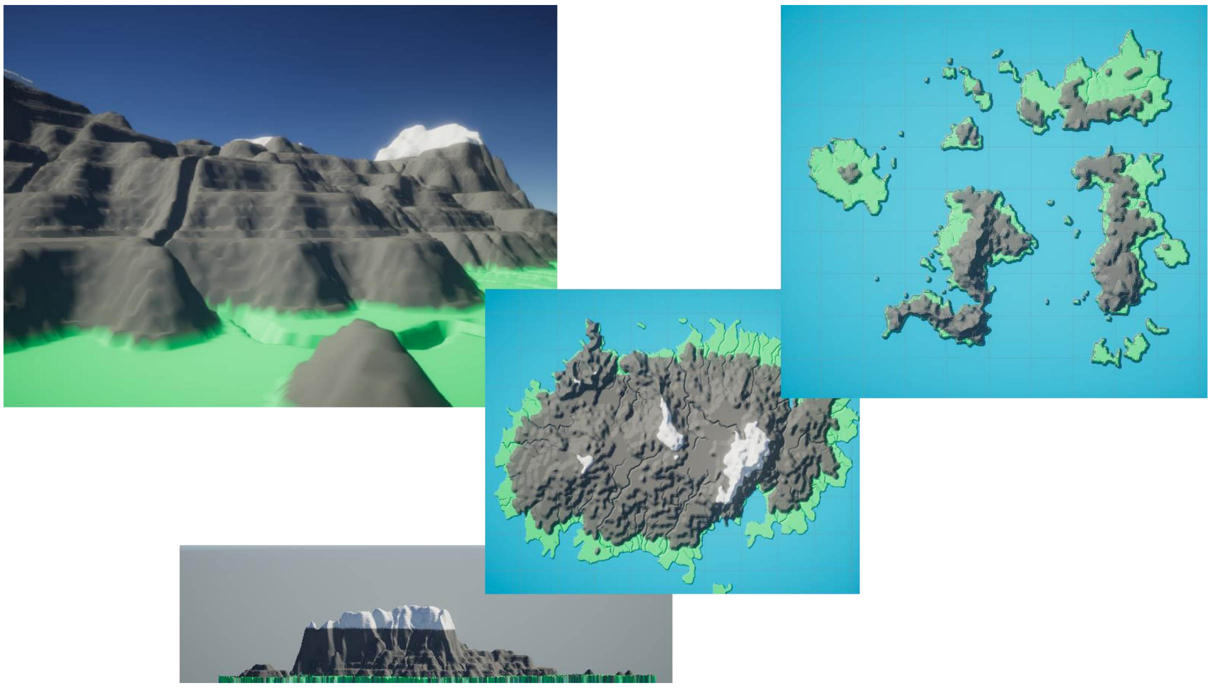
Terrain and *cylinder* generation



Date: 08.09.2021
Name: Luca Ruiters
Student nr: 500796991
GitHub: [ru1t3rl/GPE]

Table of Contents

Intro 3
Cylinder 3
  Vertices 3
  Triangles - Top and Bottom 3
  Triangles – Side 4
Terrain Generator 5
  Grid 5
    Vertices 5
    Triangles 5
    Result 6
  Heightmaps 7
  Chunks 9
Conclusion 10
References 10
Files 11

Intro

Before I started with this semester. I had no experience at all with generating a mesh or even a face/triangle in Unity. So to make an easy start I decided to start with a cylinder that you modify in real-time (when in play mode). But before I can get started with this, I need some basic knowledge for generating meshes. I followed this tutorial about a procedural, which also helped with the start of my terrain generator.

Cylinder

To get started, I took a look at Unity's cylinder in the wireframe. I saw it exists out of three segments; a top, a bottom and a side segment.

Vertices

I used Sine and Cosine to draw the circle. While placing the vertices, I came across some weird bugs. After reading through the unity scripting documentation, I saw Mathf.Sin and Cos both use radians instead of degrees

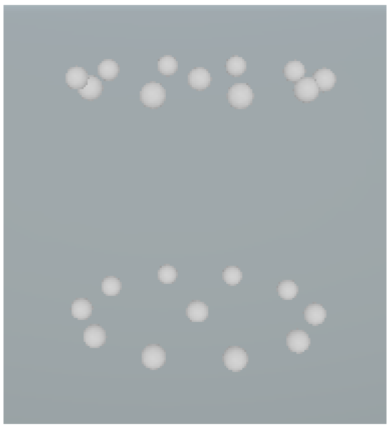
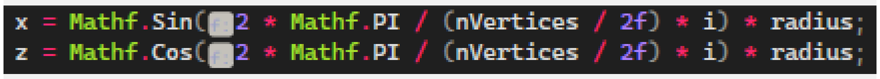


Figure 2 - Vertices for the cylinder



```
x = Mathf.Sin(f:2 * Mathf.PI / (nVertices / 2f) * i) * radius;
z = Mathf.Cos(f:2 * Mathf.PI / (nVertices / 2f) * i) * radius;
```

Figure 1 - Code used for placeing the vertices in a circle

Triangles - Top and Bottom

I placed the vertices in such order, that I can easily divide the array in half to get the first vertex of the other ring. This made it a lot easier and more efficient to create the top and bottom faces of the cylinder.

Before I can start drawing the triangles, I first need to know the triangle array size. I made some sketches and with these sketches (Note 1) came the formula **(vertices – 1) * 3**. The image below was the result of a previous attempt (without the formula).

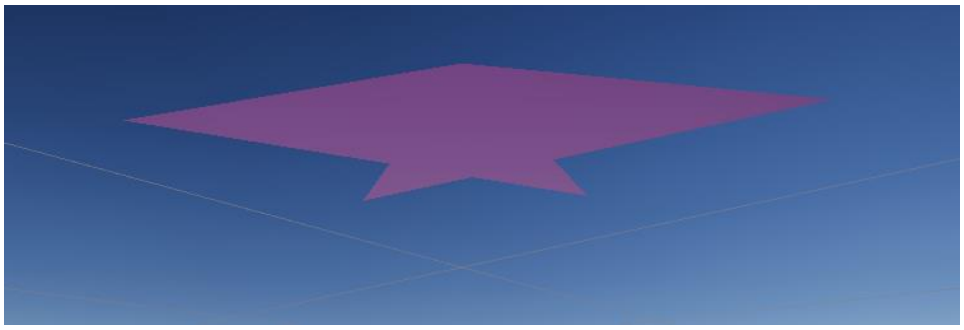
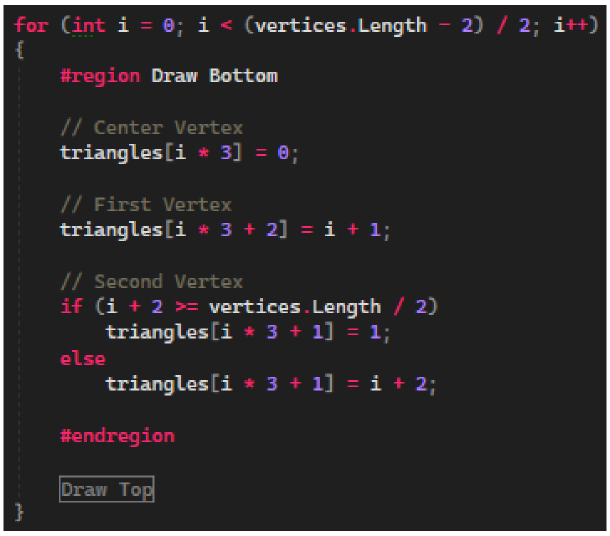


Figure 3 - A bug I encountered while trying to make a formula for the faces



```
for (int i = 0; i < (vertices.Length - 2) / 2; i++)
{
    #region Draw Bottom

    // Center Vertex
    triangles[i * 3] = 0;

    // First Vertex
    triangles[i * 3 + 2] = i + 1;

    // Second Vertex
    if (i + 2 >= vertices.Length / 2)
        triangles[i * 3 + 1] = 1;
    else
        triangles[i * 3 + 1] = i + 2;

    #endregion

    Draw Top
}
```

Figure 4 - Code for creating the top and bottom triangles

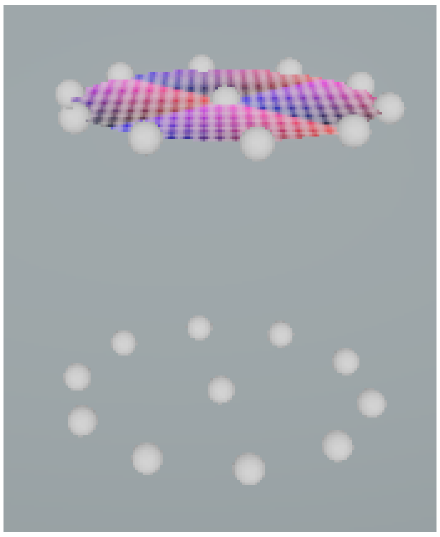


Figure 5 - The bottom faces aren't visible since the normals are facing down

Triangles – Side

To draw the sides, I drew pairs of triangles (like a quad). The steps for this were pretty easy. I only had to check if the select vertex was still on the same ring. I use a modulo to start over when the index is not on the same ring. If I did it again, I would do it differently. I came to this conclusion after looking at Unity's cylinder.

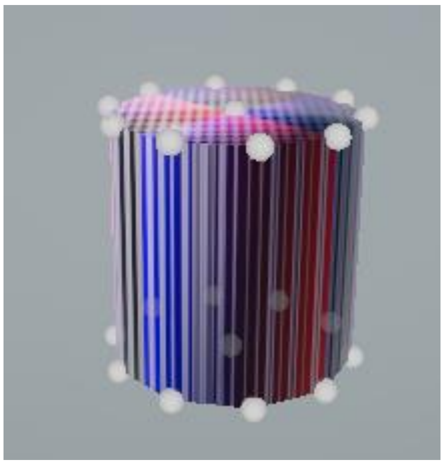
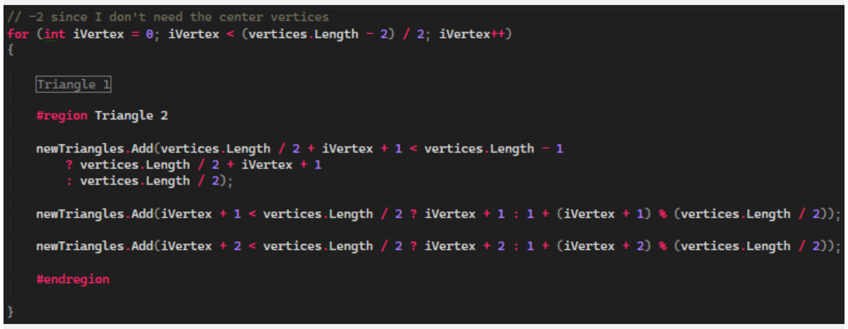
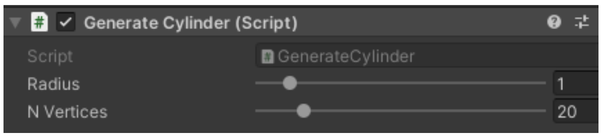


Figure 7 - I sadly wasn't able to fix the texture stretching



```
// -2 since I don't need the center vertices
for (int iVertex = 0; iVertex < (vertices.Length - 2) / 2; iVertex++)
{

    Triangle 1

    #region Triangle 2

    newTriangles.Add(vertices.Length / 2 + iVertex + 1 < vertices.Length - 1
        ? vertices.Length / 2 + iVertex + 1
        : vertices.Length / 2);

    newTriangles.Add(iVertex + 1 < vertices.Length / 2 ? iVertex + 1 : 1 + (iVertex + 1) % (vertices.Length / 2));

    newTriangles.Add(iVertex + 2 < vertices.Length / 2 ? iVertex + 2 : 1 + (iVertex + 2) % (vertices.Length / 2));

    #endregion

}
```

Figure 6 - Code for generating the side faces



```
Generate Cylinder (Script)
Script       GenerateCylinder
Radius       1
N Vertices   20
```

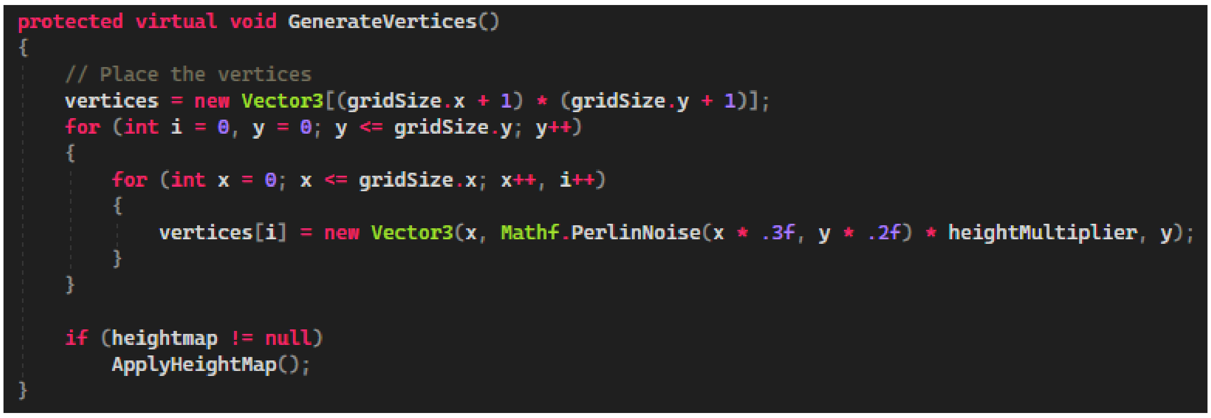
When I had finished all the code for a simple cylinder. I decided to add the option to update/modify at runtime. I accomplished this by saving the previous number of vertices and radius and check in the late update if this has changed or not; and when it has clear the mesh and generate it again.

Terrain Generator

Before I could get started with the terrain generator. I had to generate a grid of vertices. I managed to do this with a simple 2D array.

Grid

Vertices

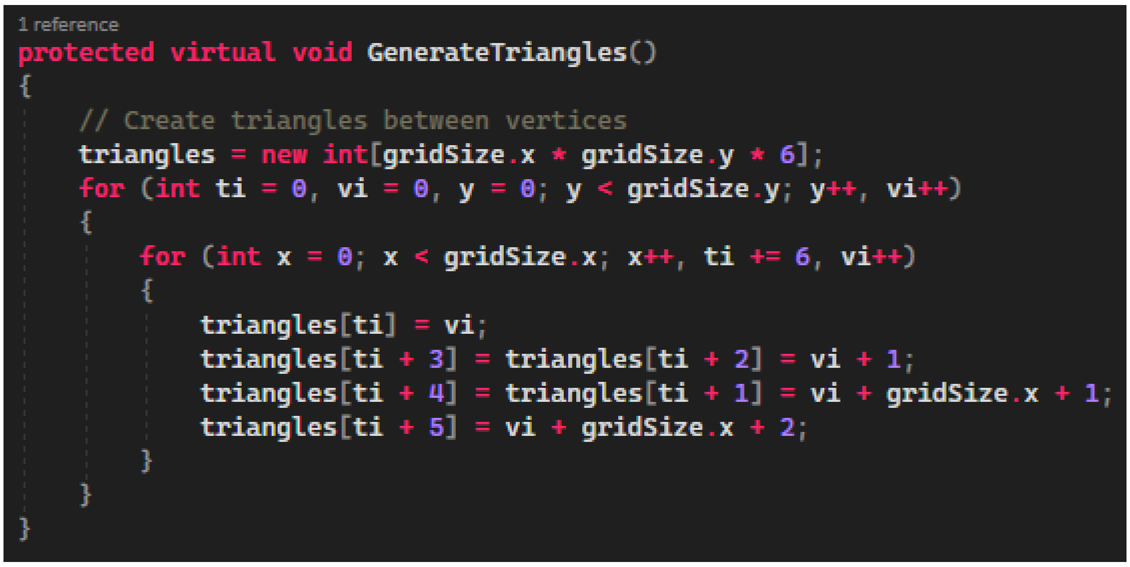


```
protected virtual void GenerateVertices()
{
    // Place the vertices
    vertices = new Vector3[(gridSize.x + 1) * (gridSize.y + 1)];
    for (int i = 0, y = 0; y <= gridSize.y; y++)
    {
        for (int x = 0; x <= gridSize.x; x++, i++)
        {
            vertices[i] = new Vector3(x, Mathf.PerlinNoise(x * .3f, y * .2f) * heightMultiplier, y);
        }
    }

    if (heightmap != null)
        ApplyHeightMap();
}
```

Figure 8 - Code for generating a grid of vertices. This screen does contain Perlin noise for height variations.

Triangles

After the vertices had been placed, I had to figure out a formula to generate the triangles with the vertices. With the code below we generate two triangles to form a quad. We will loop over this till the entire grid has been filled.

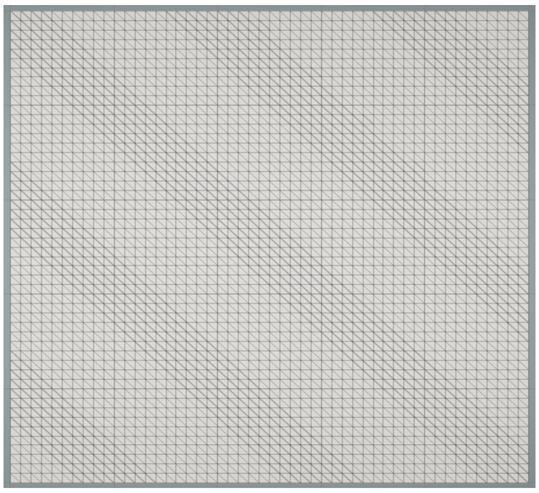


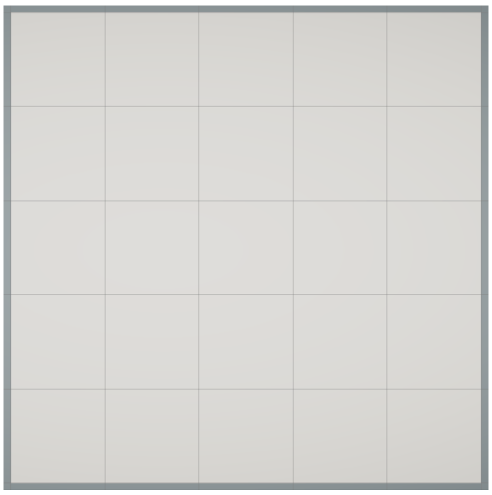
```
1 reference
protected virtual void GenerateTriangles()
{
    // Create triangles between vertices
    triangles = new int[gridSize.x * gridSize.y * 6];
    for (int ti = 0, vi = 0, y = 0; y < gridSize.y; y++, vi++)
    {
        for (int x = 0; x < gridSize.x; x++, ti += 6, vi++)
        {
            triangles[ti] = vi;
            triangles[ti + 3] = triangles[ti + 2] = vi + 1;
            triangles[ti + 4] = triangles[ti + 1] = vi + gridSize.x + 1;
            triangles[ti + 5] = vi + gridSize.x + 2;
        }
    }
}
```

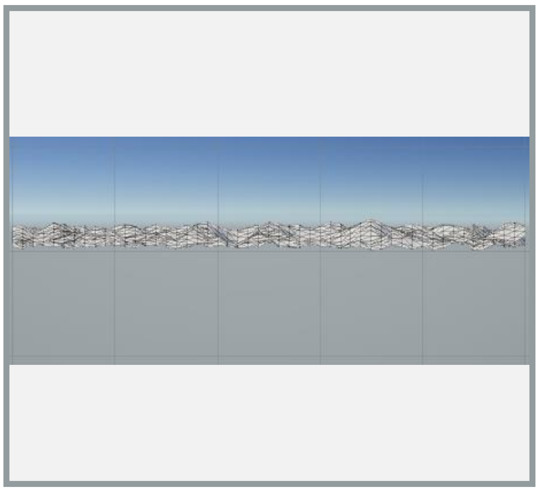
Figure 9 - This code partially comes from Catlikecoding

Result

I decided to make the grid changeable at runtime as well since the grid size was already a variable. But this isn't enough for a terrain generator. I want it to be able to load in a heightmap and assign a color to a vertex based on its height. But before I can assign the colors, I will first need to implement the heightmap option.

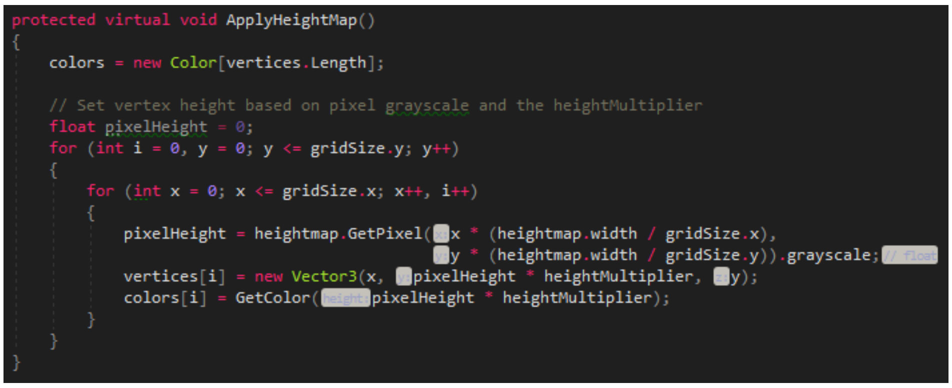




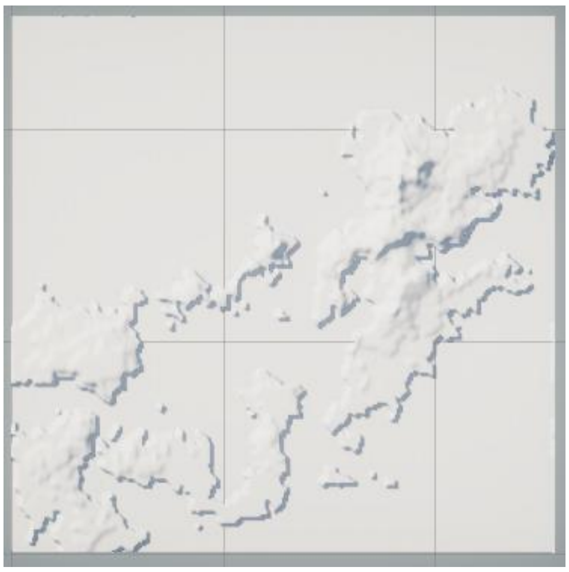


Heightmaps

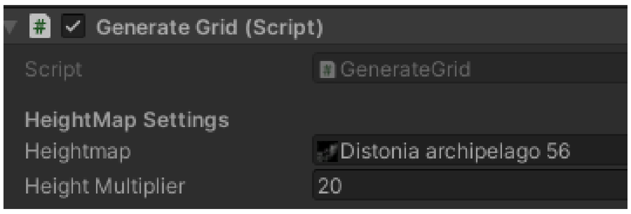
Implementing the heightmap was very easy. I could select a pixel based on the vertex index. Using the grayscale method this pixel returns a value between 0 and 1. This will be multiplied by a “height” multiplier to get a bit more flexibility.



```
protected virtual void ApplyHeightMap()
{
    colors = new Color[vertices.Length];

    // Set vertex height based on pixel grayscale and the heightMultiplier
    float pixelHeight = 0;
    for (int i = 0, y = 0; y <= gridSize.y; y++)
    {
        for (int x = 0; x <= gridSize.x; x++, i++)
        {
            pixelHeight = heightmap.GetPixel(x:x * (heightmap.width / gridSize.x),
                                             y:y * (heightmap.width / gridSize.y)).grayscale;// float
            vertices[i] = new Vector3(x, y:pixelHeight * heightMultiplier, z:y);
            colors[i] = GetColor(height:pixelHeight * heightMultiplier);
        }
    }
}
```



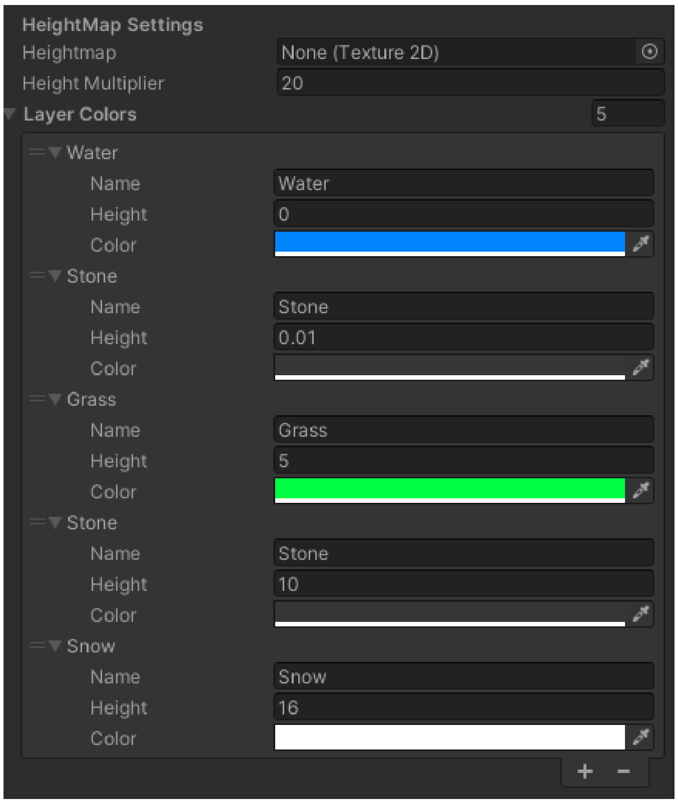
This is what it looks like when you generate terrain with this method. In the generate method we first check if there is a heightmap if there is we will apply it and else it will just use the Perlin noise.



Generate Grid (Script)

Script	GenerateGrid
HeightMap Settings	
Heightmap	Distonia archipelago 56
Height Multiplier	20

I have made a new class called ColorLayer which is serializable (so I can make it visible in the inspector) and contains values like height and color. The colors weren't visible at first since unity doesn't support vertex colors by default. To work around this I made a simple shader in ShaderGraph.



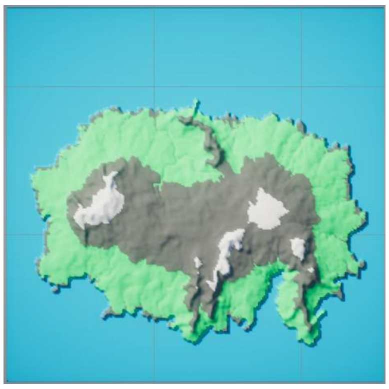
```
HeightMap Settings
Heightmap                 None (Texture 2D)
Height Multiplier         20
Layer Colors              5
  Water
    Name                  Water
    Height                0
    Color
  Stone
    Name                  Stone
    Height                0.01
    Color
  Grass
    Name                  Grass
    Height                5
    Color
  Stone
    Name                  Stone
    Height                10
    Color
  Snow
    Name                  Snow
    Height                16
    Color
```

The colors and the height multiplier are the only variables you aren't able to edit in real-time.

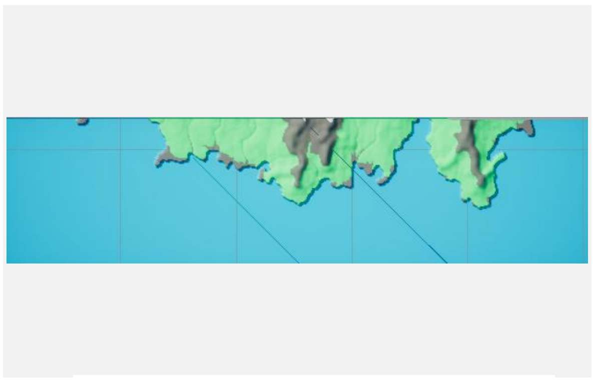
The images below show the final result of the terrain generation with a heightmap and automatic colors.

While playing around with the grid size values, I discovered a bug. I couldn't find the source of it. I'm still not sure why it happened. But if I have to guess, it has to do with a max amount of vertices per mesh. When I increase the grid size above 256x256 the amount of vertices doesn't change.

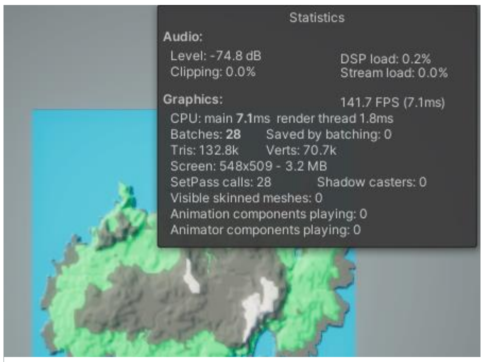
To work around this problem and improve performance, I made a grid system that uses chunks.



256x256 (What it should look like)

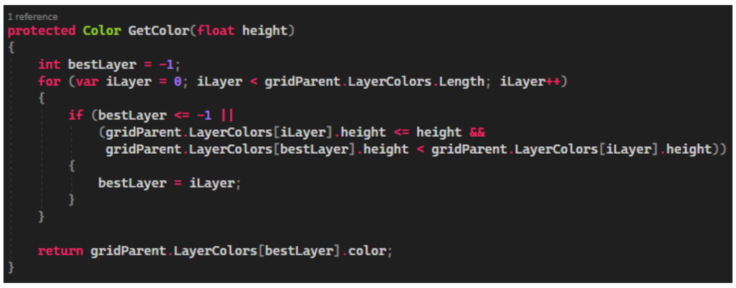


512x512 (The bug, only occurs on grid sizes above 256x256)



```
Statistics
Audio:
  Level: -74.8 dB                 DSP load: 0.2%
  Clipping: 0.0%                  Stream load: 0.0%

Graphics:                         141.7 FPS (7.1ms)
  CPU: main 7.1ms  render thread 1.8ms
  Batches: 28      Saved by batching: 0
  Tris: 132.8k     Verts: 70.7k
  Screen: 548x509 - 3.2 MB
  SetPass calls: 28          Shadow casters: 0
  Visible skinned meshes: 0
  Animation components playing: 0
  Animator components playing: 0
```

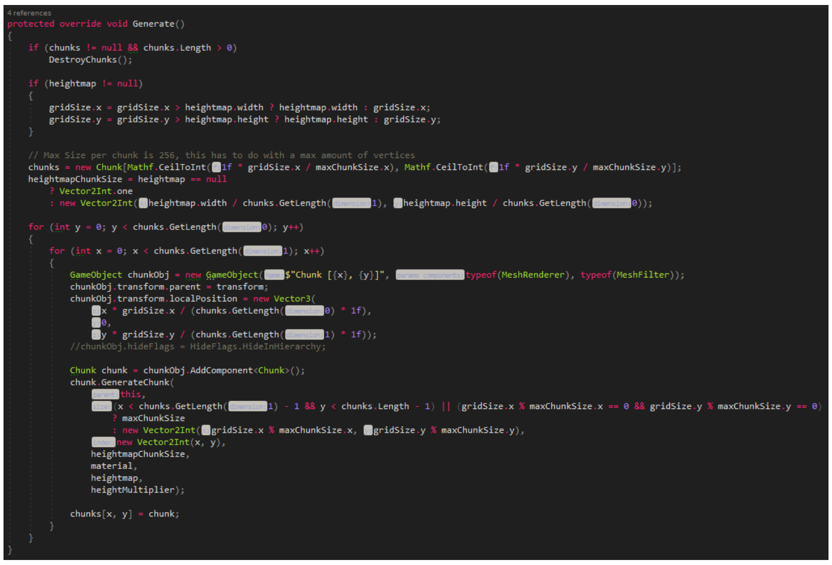
Figure 11 - The amount of tris and verts (256x256)



```
1 reference
protected Color GetColor(float height)
{
    int bestLayer = -1;
    for (var iLayer = 0; iLayer < gridParent.LayerColors.Length; iLayer++)
    {
        if (bestLayer <= -1 ||
            (gridParent.LayerColors[iLayer].height <= height &&
             gridParent.LayerColors[bestLayer].height < gridParent.LayerColors[iLayer].height))
        {
            bestLayer = iLayer;
        }
    }

    return gridParent.LayerColors[bestLayer].color;
}
```

Figure 10 - Method used to select a color layer

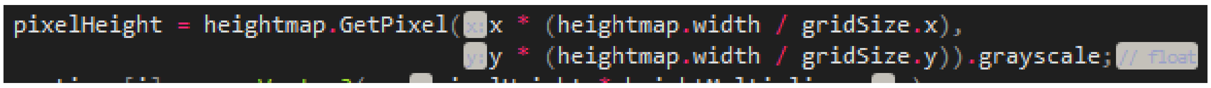
Chunks



```
4 references
protected override void Generate()
{
    if (chunks != null && chunks.Length > 0)
        DestroyChunks();

    if (heightmap != null)
    {
        gridSize.x = gridSize.x > heightmap.width ? heightmap.width : gridSize.x;
        gridSize.y = gridSize.y > heightmap.height ? heightmap.height : gridSize.y;
    }

    // Max Size per chunk is 256, this has to do with a max amount of vertices
    chunks = new Chunk[Mathf.CeilToInt(f:1f * gridSize.x / maxChunkSize.x), Mathf.CeilToInt(f:1f * gridSize.y / maxChunkSize.y)];
    heightmapChunkSize = heightmap == null
        ? Vector2Int.one
        : new Vector2Int(x:heightmap.width / chunks.GetLength(dimension:1), y:heightmap.height / chunks.GetLength(dimension:0));

    for (int y = 0; y < chunks.GetLength(dimension:0); y++)
    {
        for (int x = 0; x < chunks.GetLength(dimension:1); x++)
        {
            GameObject chunkObj = new GameObject(name:$"Chunk [{x}, {y}]", params components:typeof(MeshRenderer), typeof(MeshFilter));
            chunkObj.transform.parent = transform;
            chunkObj.transform.localPosition = new Vector3(
                x:x * gridSize.x / (chunks.GetLength(dimension:0) * 1f),
                y:0,
                z:y * gridSize.y / (chunks.GetLength(dimension:1) * 1f));
            //chunkObj.hideFlags = HideFlags.HideInHierarchy;

            Chunk chunk = chunkObj.AddComponent<Chunk>();
            chunk.GenerateChunk(
                parent:this,
                size:(x < chunks.GetLength(dimension:1) - 1 && y < chunks.Length - 1) || (gridSize.x % maxChunkSize.x == 0 && gridSize.y % maxChunkSize.y == 0)
                    ? maxChunkSize
                    : new Vector2Int(x:gridSize.x % maxChunkSize.x, y:gridSize.y % maxChunkSize.y),
                index:new Vector2Int(x, y),
                heightmapChunkSize,
                material,
                heightmap,
                heightMultiplier);

            chunks[x, y] = chunk;
        }
    }
}
```

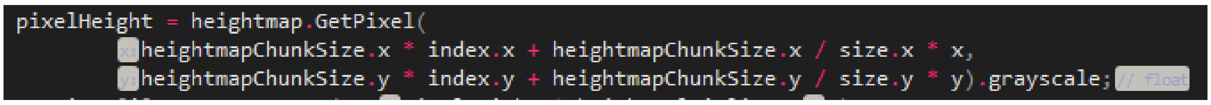
Figure 12 - Code to generate a grid of vertices based on chunk- and grid size

One of the more difficult parts of the chunks was to make the heightmap work across the entire object instead of repeating it on every chunk. To get this formula I made some sketches these can be found in Note 3 and 4.



```
pixelHeight = heightmap.GetPixel(x:x * (heightmap.width / gridSize.x),
                                 y:y * (heightmap.width / gridSize.y)).grayscale; // float
```

Figure 13 - Old code for loading a heightmap



```
pixelHeight = heightmap.GetPixel(
        x:heightmapChunkSize.x * index.x + heightmapChunkSize.x / size.x * x,
        y:heightmapChunkSize.y * index.y + heightmapChunkSize.y / size.y * y).grayscale; // float
```

Figure 14 - New code to process a heightmap when using chunks

The code to generate a chunk is almost equal to the code of generating a grid.

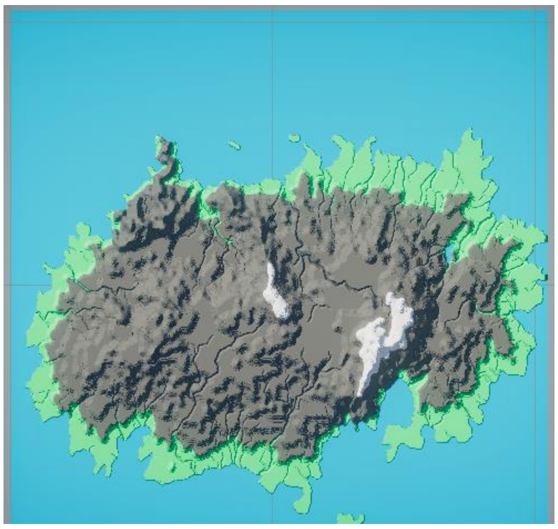


Figure 16 - 2048x2048 terrain with 128x128 chunks

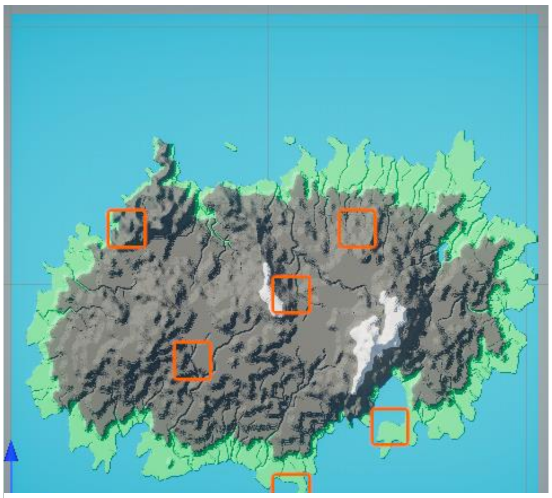


Figure 15 - Same as Figure 15 but with some chunks selected

Conclusion

I enjoyed working on the terrain generator and have learned a lot. From creating a simple shape to reading a heightmap and applying this to a mesh.

In the future, I would like to continue with this and do research in Marching Cubes. Sebastian Lague's video about marching cubes inspired me.

A link to the GitHub project/page can be found on the cover.

References

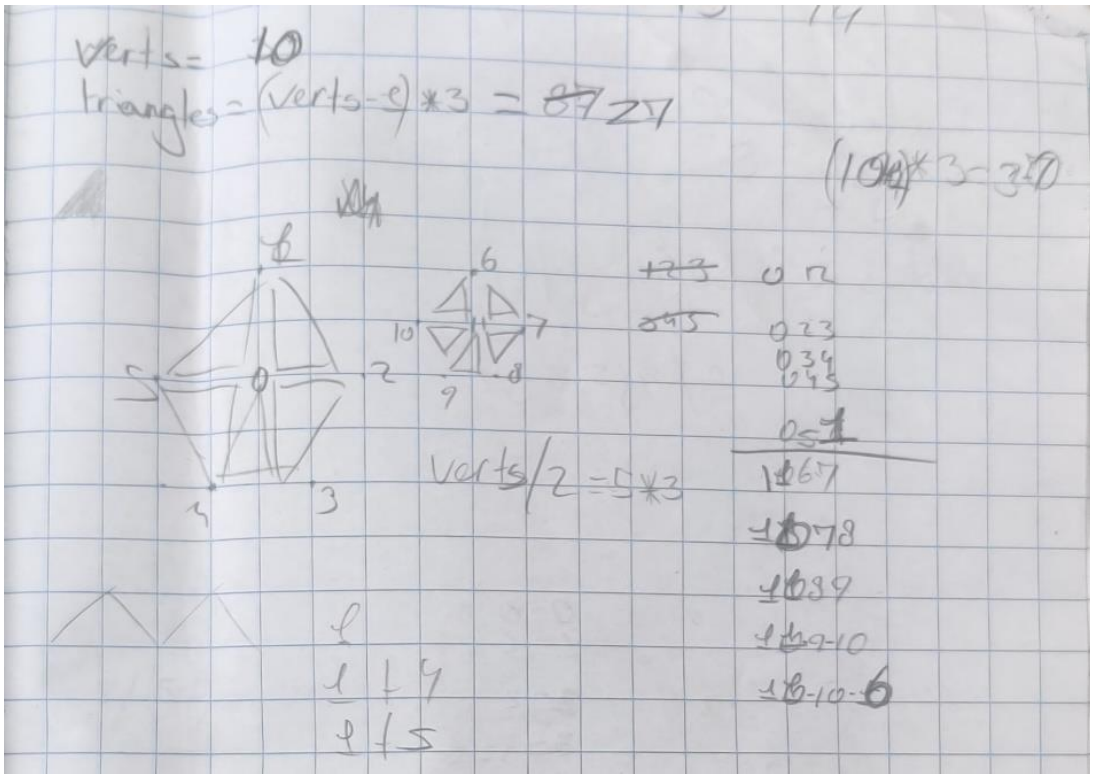
Heightmap on primitive plane - Unity Answers -> reading heightmap values

Procedural Grid, a Unity C# Tutorial (catlikecoding.com) -> procedural grid

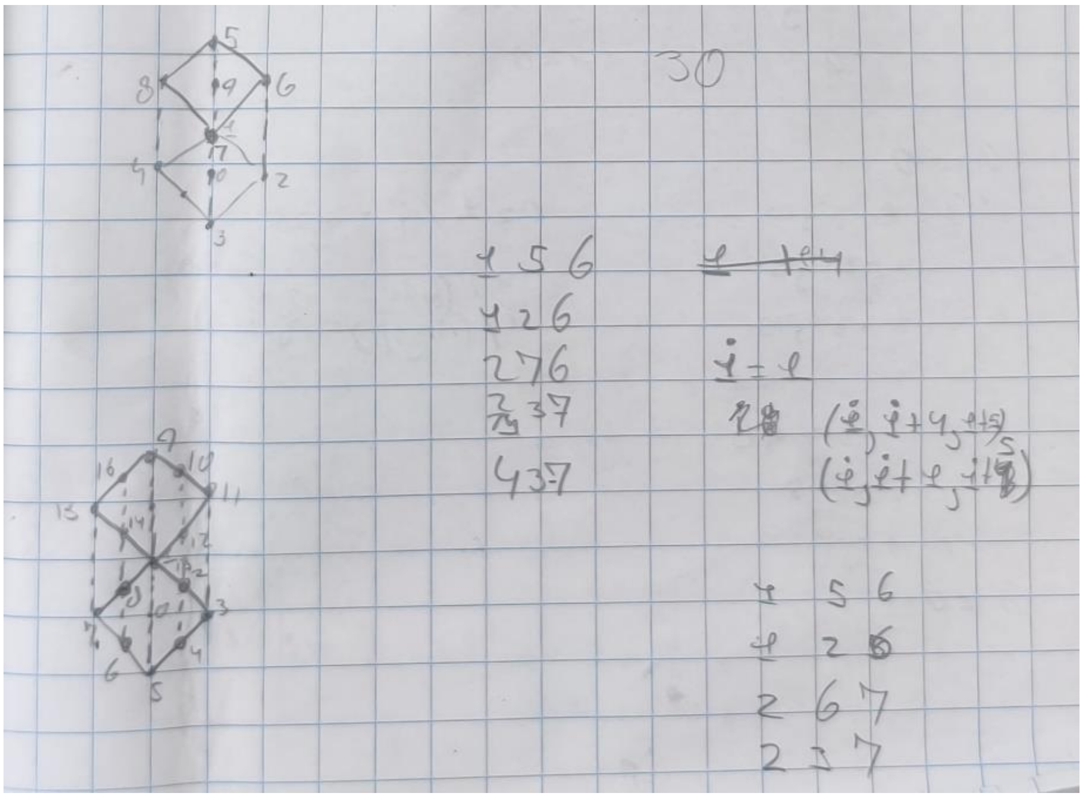
Unity - Scripting API: Mesh.colors (unity3d.com) -> vertex colors

https://www.youtube.com/watch?v=M3iI2l0ltbE -> marching cubes

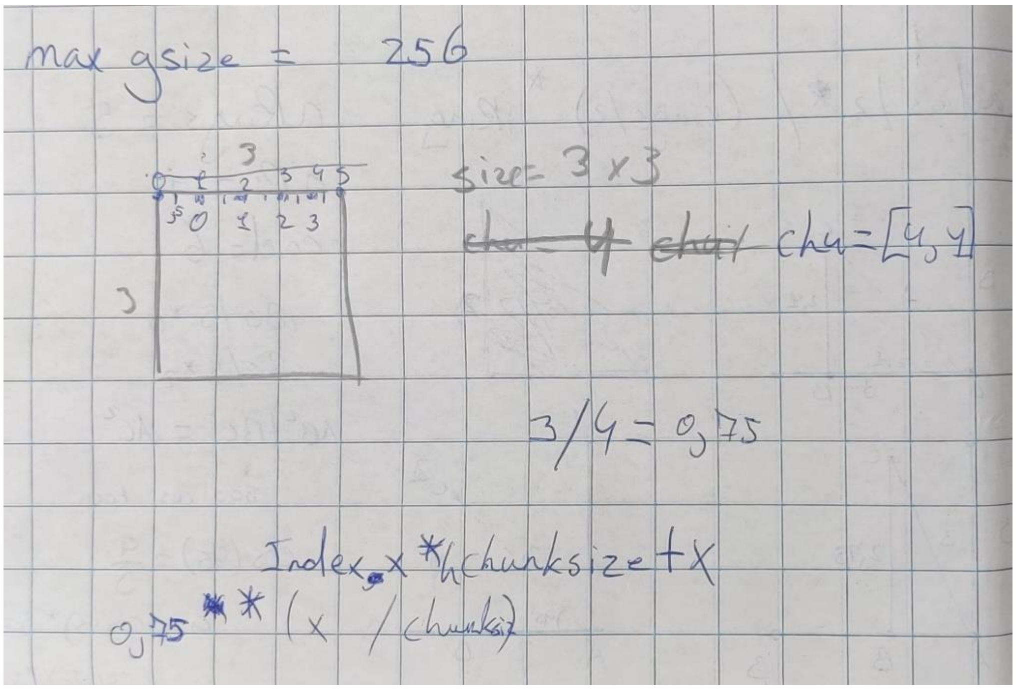
Files



Note 1 - Visualization for the triangle calculation (top & bottom faces)



Note 2 - Sketches made for generating the sides/walls of the cylinder


max gsize = 256

size = 3 x 3

~~chu = 4~~ ~~chu~~ chu = [4, 4]

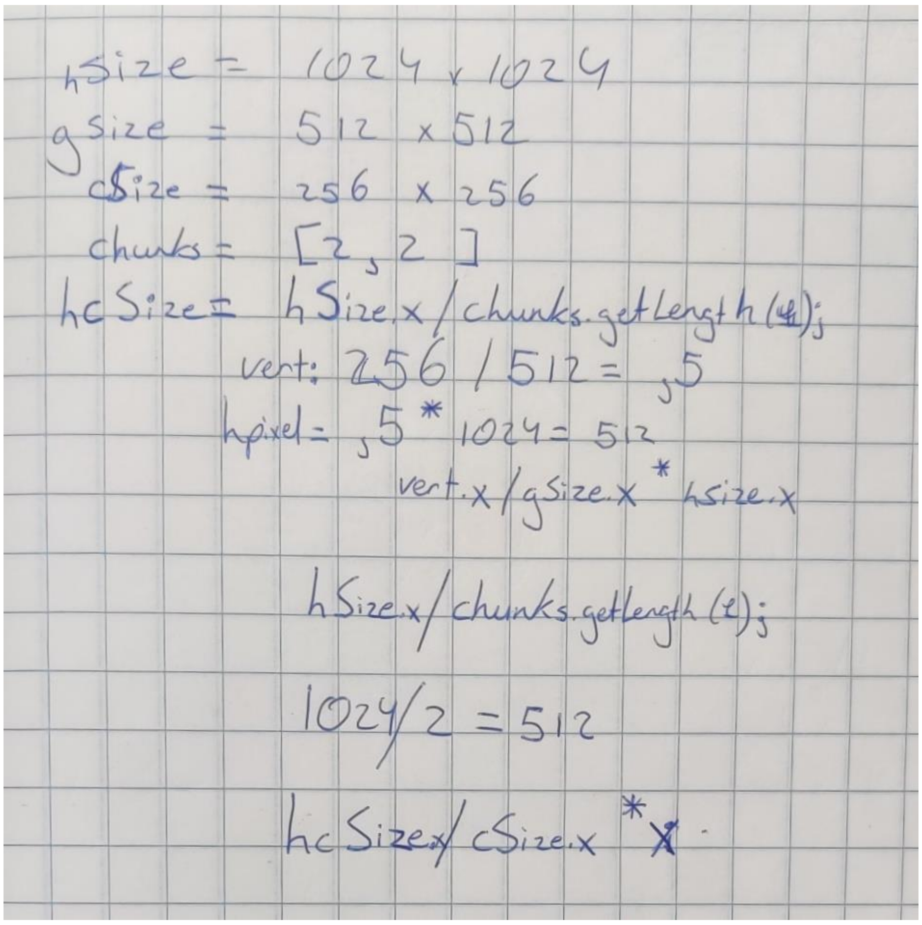
3/4 = 0,75

Index.x * hchunksize + x

0,75 * (x / chunksiz)

3
0 1 2 3 4 5
0 1 2 3
3

Note 3 - Calculations for applying the correct height map section to a chunk


hSize = 1024 x 1024
gSize = 512 x 512
cSize = 256 x 256
chunks = [2, 2]
hcSize = hSize.x / chunks.getLength(~~4~~);
vert: 256 / 512 = ,5
hpixel = ,5 * 1024 = 512
vert.x / gSize.x * hSize.x

hSize.x / chunks.getLength(1);

1024/2 = 512

hcSize / cSize.x * ~~x~~

Note 4 - Final calculation/check used for selecting a heightmap region for a chunk