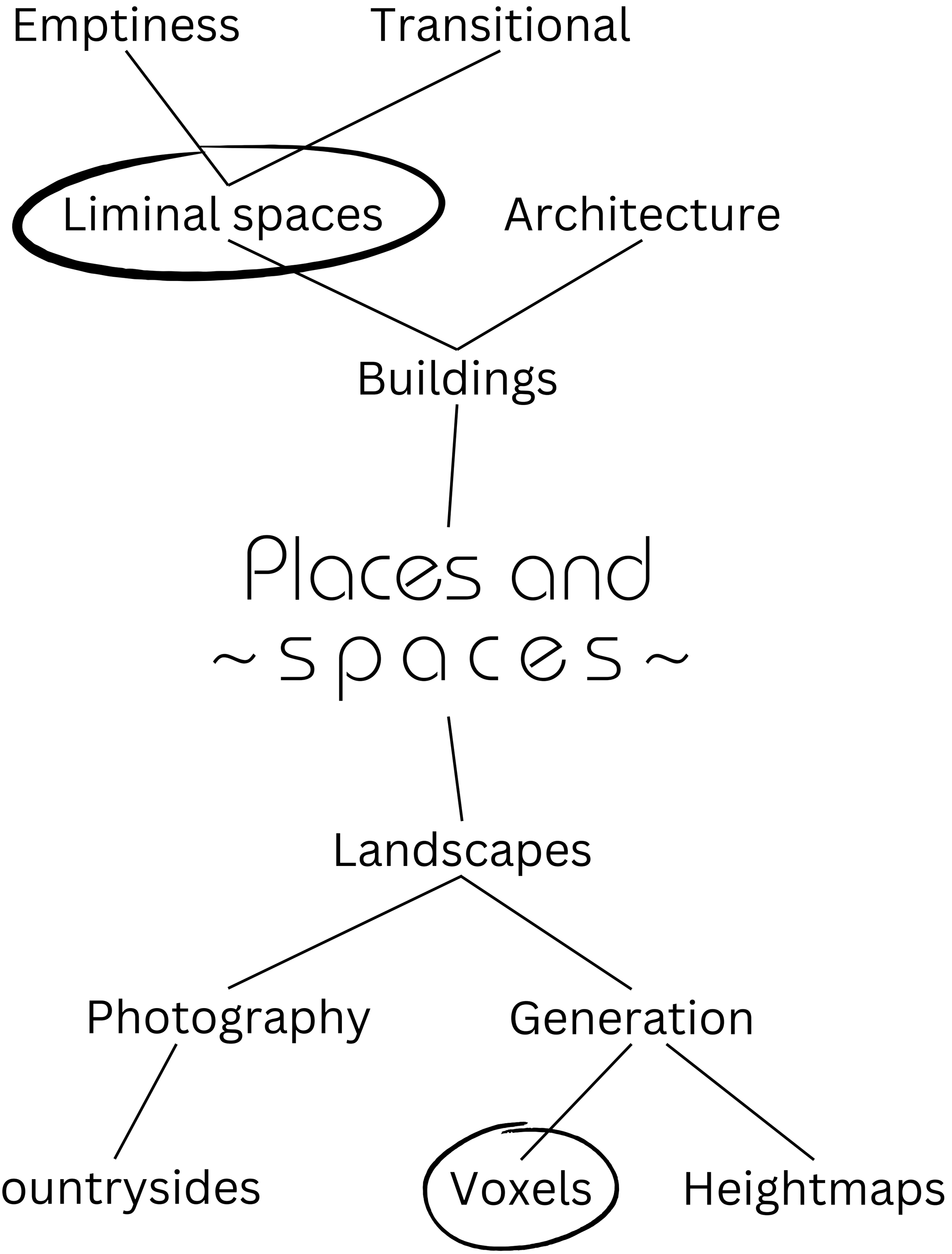
Places and ~spaces~

Mind map


Emptiness
Transitional
Liminal spaces
Architecture
Buildings
Places and
~spaces~
Landscapes
Photography
Generation
Countrysides
Voxels
Heightmaps

Artist 1:

MINECRAFT

Minecraft is easily the most obvious and recognisable example of a procedurally generated voxel world.

It is primarily a game about creativity, although offering a survivial game mode where you strive to make yourself a home base, max out your gear, and eventually slay the Ender Dragon, supposedly the end of the game.

A method called “perlin noise” is used to create hills, mountains, plains, and even caves. Each voxel is decorated depending on how high it is, what blocks surround it, and what biome it is in. And finally, structures and decorations are scattered around, like trees, ore veins, villages, pyramids, and dungeons. All completely infinitely, or due to some limitations, around 4 times the size of the real life Earth. This world is split up into 16x16x256 amounts of voxels called “chunks”, which are loaded in and out on the fly instead of rendering the entire world at once.

Image analysis

I’ve chosen this image in particular because it demonstrates the variety in Minecraft’s terrain generation and has examples on some different techniques used and where they can be seen.

We can actually see 4 different biome types in this image: the plains, the forest, the tundra, and the mountains.

Plains are the simplest biome in the game, they consist of some very gentle perlin noise, which creates a heightmap of the area and each voxel references that to know if it is a dirt, grass, or air block.

The forest just beside it is very similar, although the noise is actually flatter due to the other variety it has. Minecraft uses “decorations”, here seen with the trees, which scatters some pre-built or slightly procedural objects which it scatters along the ground. Explaining how each tree is generated is a bit more complicated and I intend on leaving it for later.

The tundra is very similar to the forest, although it also scatters layers of snow on the ground, by detecting which voxels have only air blocks above them, so they can simulate which areas would have snow falling on them, as long as you assume that snow falls directly downwards.

Over on the right, we see the elephant in the room: a big old mountain. These are generated with a more zoomed in and exaggerated perlin noise. We can also see an entrance to a cave on the mountain, which uses 3D perlin noise, something very complicated that even I don’t get entirely. The game uses the noise to cut out holes in the existing voxel data to create looping caves, and their enterances.

Finally, there’s a village in the plains. This is different from the “decorations” system, and is actually the “structures” system, where the game uses completely preset buildings and path pieces and snaps them together like a jigsaw puzzle in a mostly random manner to create things such as a village with houeses, wells, paths, and farms.

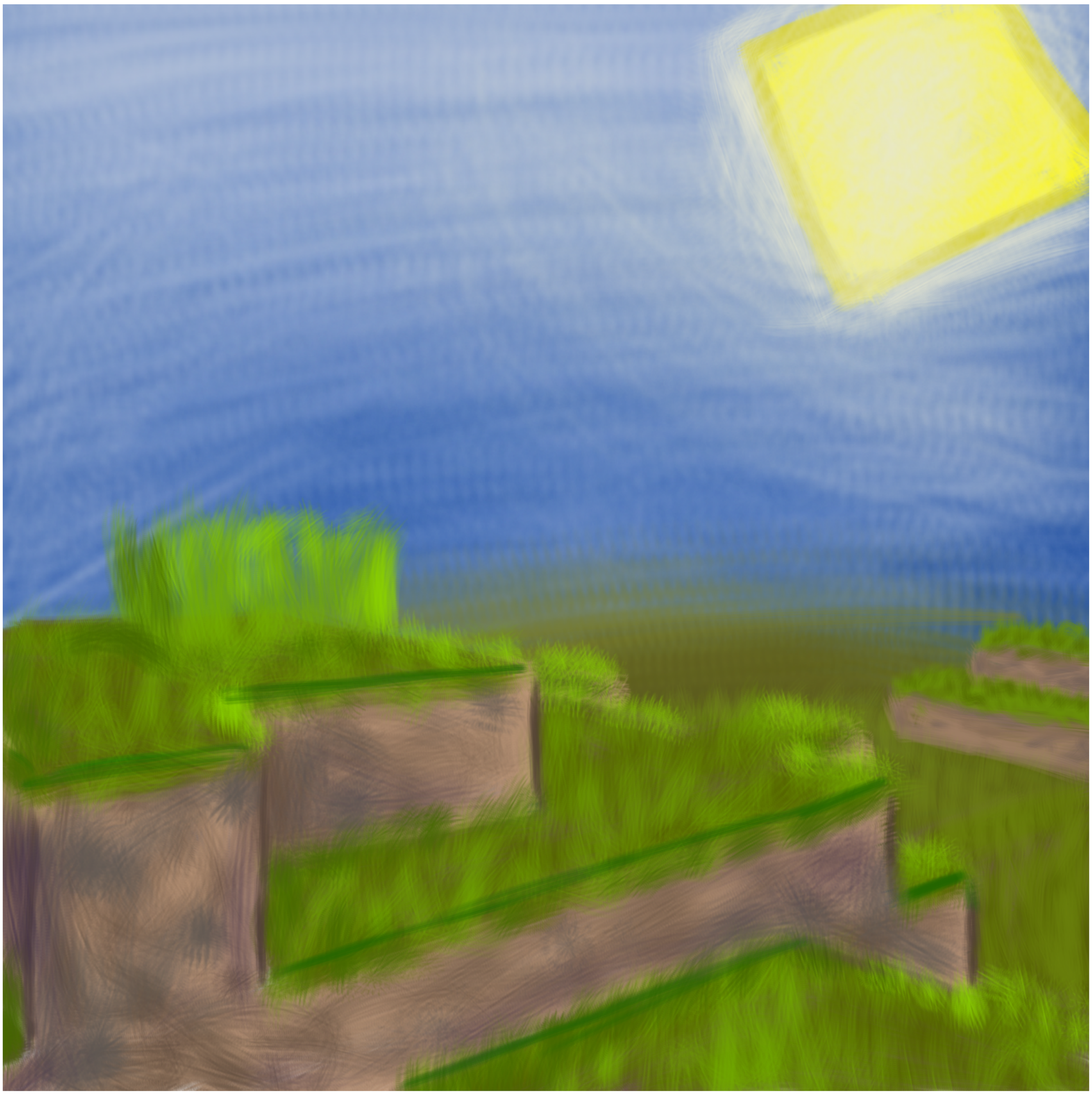
Statement of intent

My intention is to program a procedurally generated voxel terrain system, like Minecraft, with emphasis on visual prettiness and any optimisation techniques I can figure out along the way to present a vast and (hopefully) endless world made of nothing but little cubes using the Unity game engine, and the C# programming language.

Unity
version 2022.3.10f1

The project should explore the theme “places and spaces”, aswell as many computer science concepts like mesh generation, perlin noise, chunking, texture atlases, normal vectors, and taking a look at more complex methods like octree partitioning.

Formal elements such as line, shape, form, tone, colour, contrast, should be visible in this project.



Concept art I made for the project

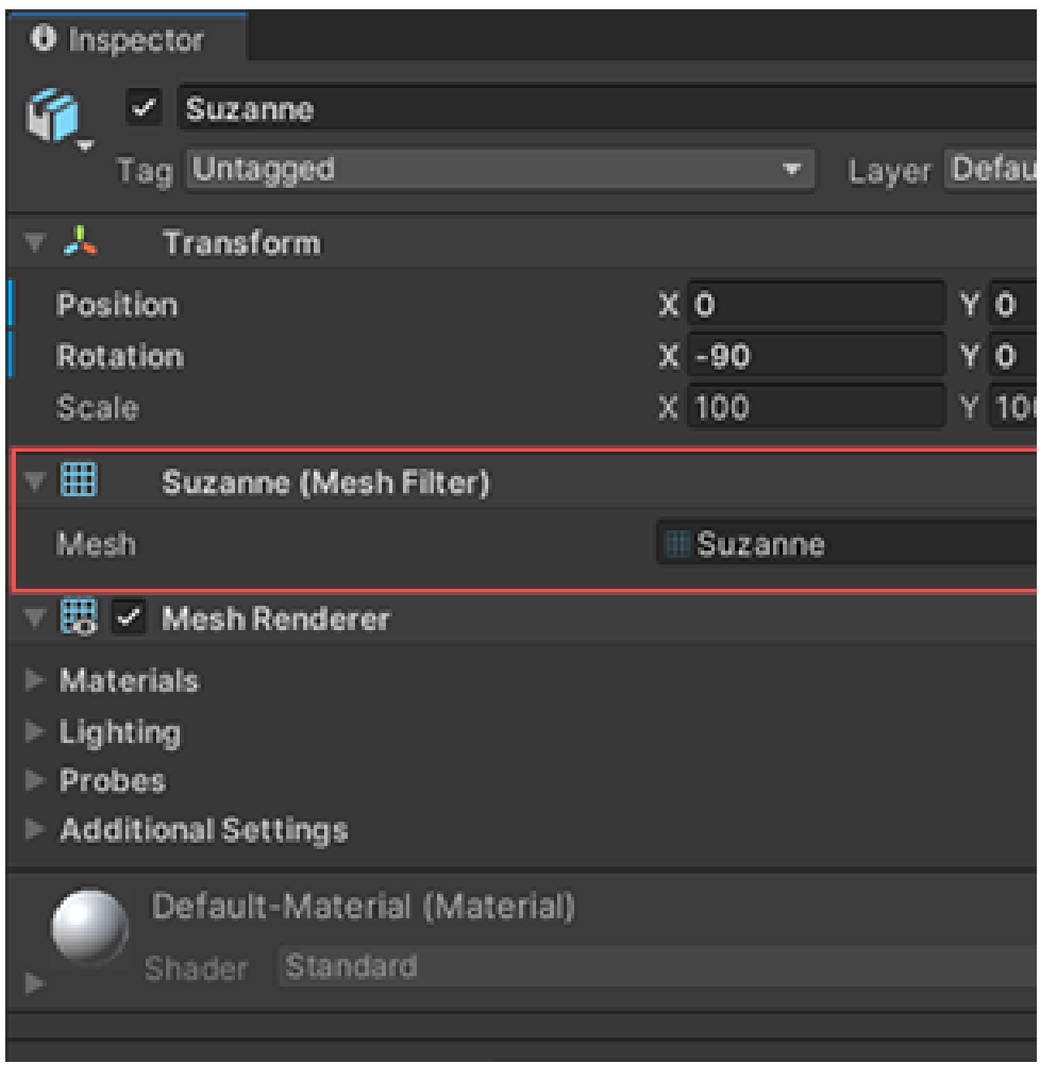
Basics of generating a mesh

Everything in 3D graphics is made out of meshes, which are made out of polygons, which are made out of triangles (tris), which are made out of vertices, edges, and faces. To create anything, we must first conquer the triangle. This means 3 vertices, 3 edges, and one face.

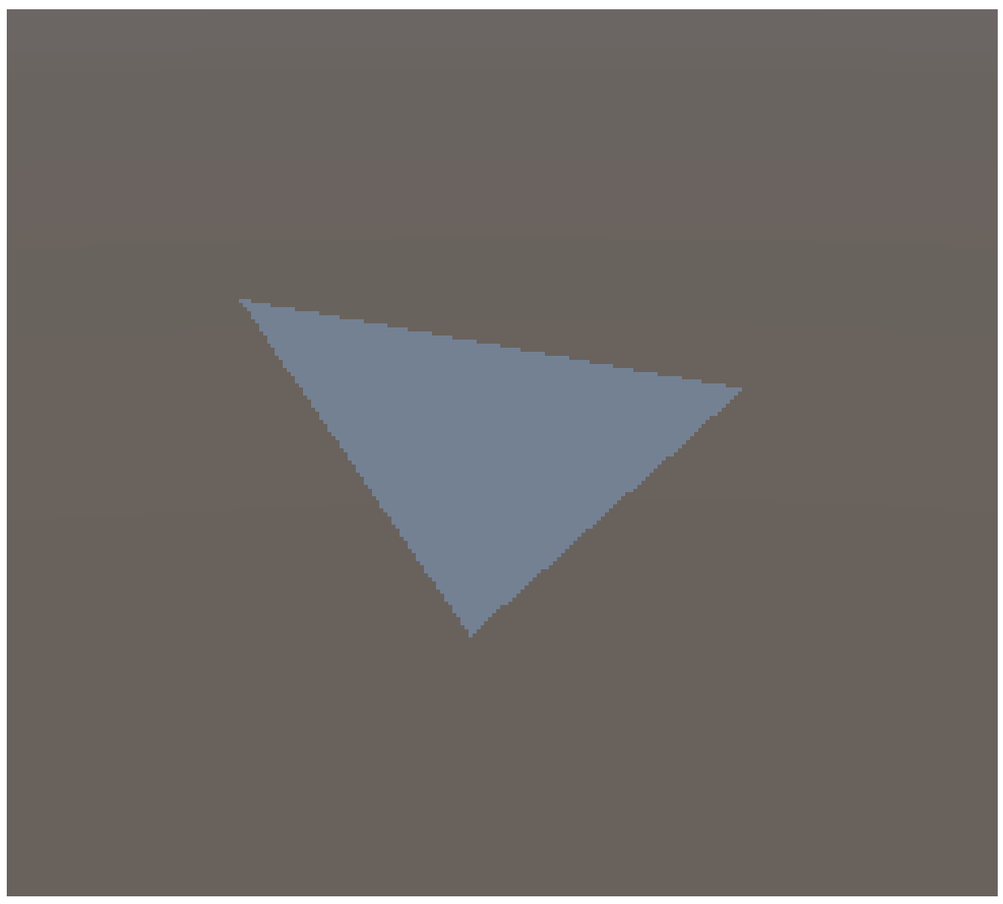
A vertex is a single point in 3D space, these are referenced by their co-ordinates, for example, (1, 0, 0).

An edge is a line which connects to vertices together, which the computer figures out when we define the triangles. We define these by listing which points they connect to, in lots of 3 for each tri. These must be listed in a clockwise order, or else they would be rendered inside out.

Here, I've shown the C# programming behind this. In Unity, we have a MeshFilter component on the game object, which is what we access to define the mesh we are generating. First we create a mesh called mesh, and then we go through each step of the mesh generation process and assigning these to the mesh, and assigning that mesh to the MeshFilter at the very end to finish off our first triangle.



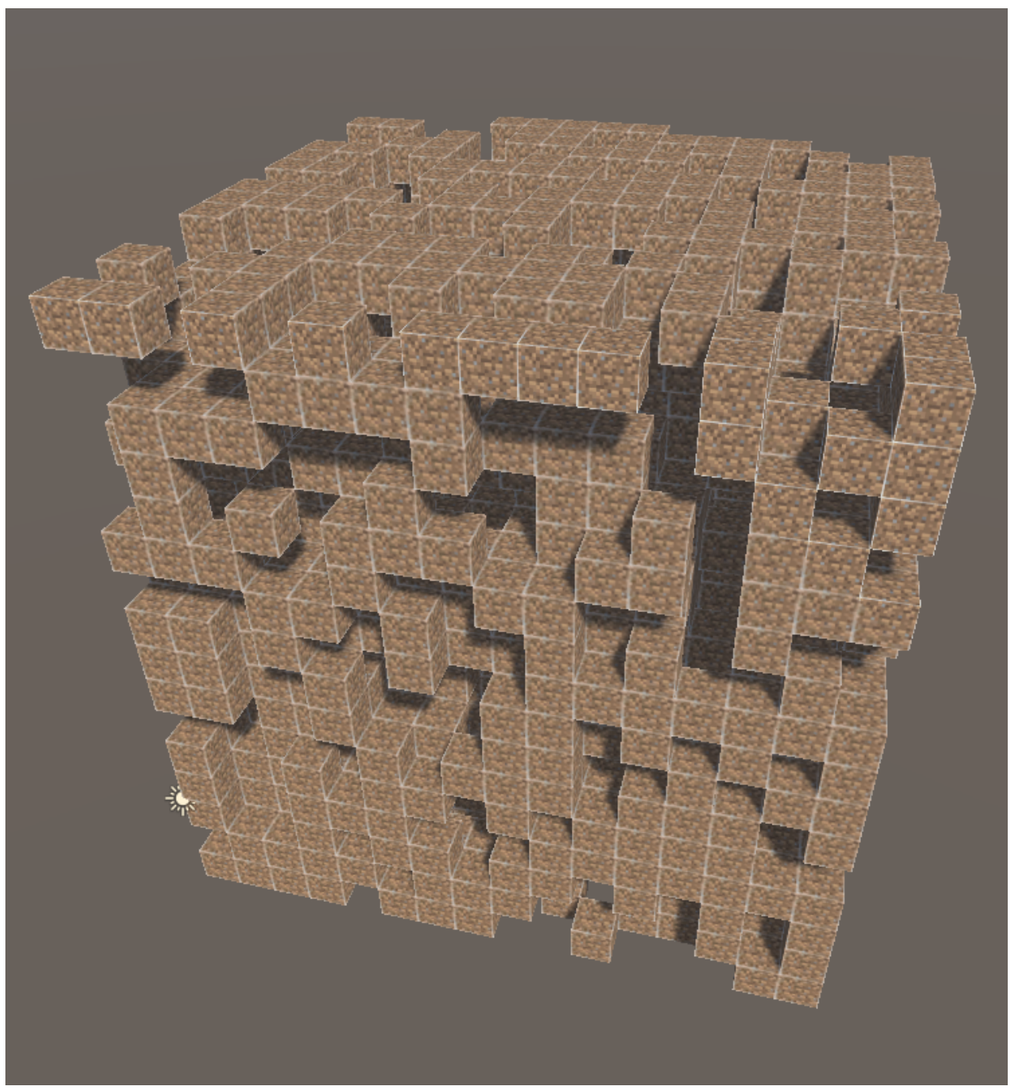
Example of the MeshFilter component shown in the Unity properties panel.



The rendered result of our first triangle.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MeshGen : MonoBehaviour
{
    public MeshFilter meshFilter;

    void Start()
    {
        Mesh mesh = new Mesh();

        // generate verticies
        Vector3[] vertices = new []
        {
            new Vector3( 0, 0, 0),
            new Vector3(-1, 1, 0),
            new Vector3( 1, 1, 0),
        };
        mesh.vertices = vertices;

        // generate triangles
        int[] triangles = new []
        {
            0,
            1,
            2,
        };
        mesh.triangles = triangles;

        meshFilter.mesh = mesh;
    }
}
```

First Chunk



Our first generated chunk of voxels.

```
// front
vertices.Add(new Vector3(i  , j  , k  )); // 0
vertices.Add(new Vector3(i  , j+1, k  )); // 1
vertices.Add(new Vector3(i+1, j  , k  )); // 2
vertices.Add(new Vector3(i+1, j+1, k  )); // 3
// left
vertices.Add(new Vector3(i  , j  , k+1)); // 4
vertices.Add(new Vector3(i  , j+1, k+1)); // 5
vertices.Add(new Vector3(i  , j  , k  )); // 6
vertices.Add(new Vector3(i  , j+1, k  )); // 7
//right
vertices.Add(new Vector3(i+1, j  , k  )); // 8
vertices.Add(new Vector3(i+1, j+1, k  )); // 9
vertices.Add(new Vector3(i+1, j  , k+1)); // 10
vertices.Add(new Vector3(i+1, j+1, k+1)); // 11
// back
vertices.Add(new Vector3(i  , j  , k+1)); // 12
vertices.Add(new Vector3(i  , j+1, k+1)); // 13
vertices.Add(new Vector3(i+1, j  , k+1)); // 14
vertices.Add(new Vector3(i+1, j+1, k+1)); // 15
// top
vertices.Add(new Vector3(i  , j+1, k  ));
vertices.Add(new Vector3(i  , j+1, k+1));
vertices.Add(new Vector3(i+1, j+1, k  ));
vertices.Add(new Vector3(i+1, j+1, k+1));
// bottom
vertices.Add(new Vector3(i  , j  , k  ));
vertices.Add(new Vector3(i  , j  , k+1));
vertices.Add(new Vector3(i+1, j  , k  ));
vertices.Add(new Vector3(i+1, j  , k+1));
```

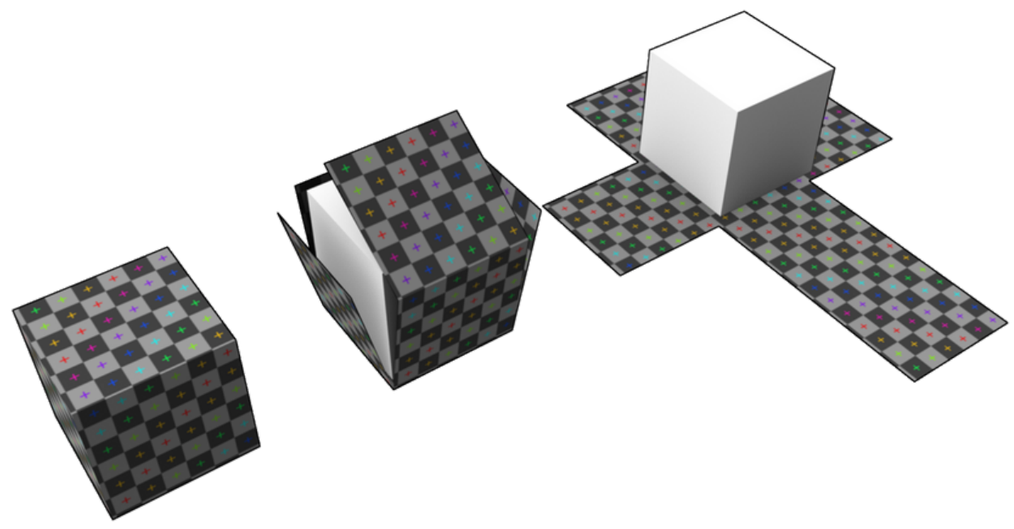
A snippet from the vertices generation

Now that we can generate something, we can get to generating an entire chunk of blocks. A “chunk” is a 16x16x16 group of voxels, all in one mesh. The world is split up into these so that the game can easily load and deload these when they’re close enough for the player to see them, since you can’t render the entire world all at once.

The voxel data is taken from an integer list, filled with 16^3 random 0s and 1s, just to test things out. A function cycles through each one, and building the mesh for that cube before moving onto the next one.

Getting the verts in the right spot was a pain.

This also required a new process of the mesh generation: UV mapping. Which is a method of defining where the 3D points reside in a seperate, 2D space, used to tell the game where to place textures and images on the faces of the mesh.

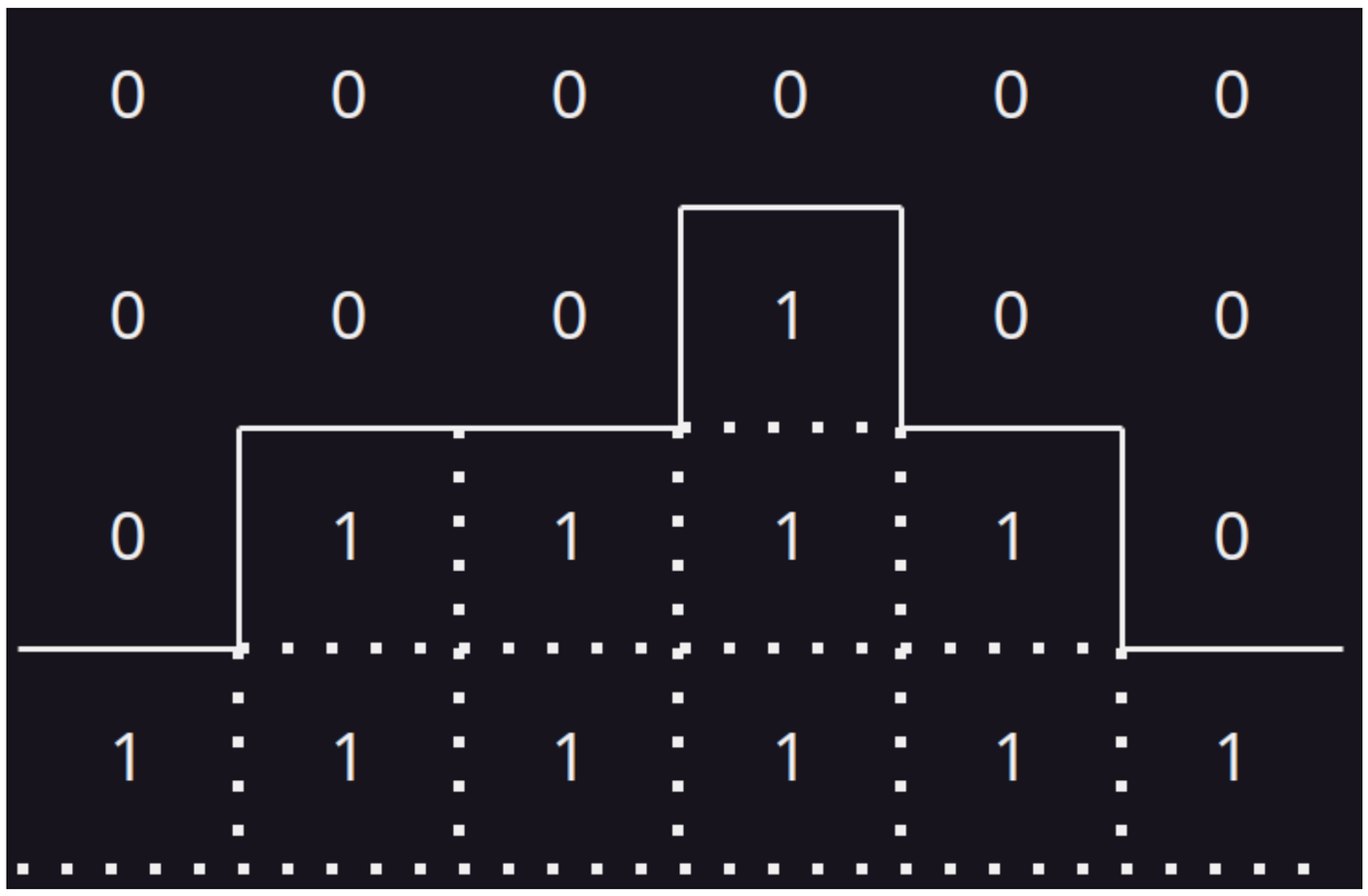


Visual example of a cube being UV mapped

Unfortunately, while I was trying to implement UV mapping, this also increased the vertex count for each voxel by 3 whole times. This is because each point on the UV map correlates to one of the vertices, which means that with only 8 vertices in a cube, the left, right, top and bottom of the cube would be completely blank because they’re sharing their UV co-ordinates with the front and back faces. To counter this, I had to allocate 4 vertices for each face, increasing vertices per voxel from 8 to 24. This is one of the big reasons why we now needed some major optimisations...

First optimisations

The issue with how we currently render our voxels, is that we are processing so many faces that aren't even visible at all, as you can see from this diagram, showing the side-on view of a single slice of our world. Each 0 represents an air block, while each 1 represents a filled voxel. The think lines are the faces which are facing the outside, and therefore could possibly be seen from the outside. All of the dotted lines, however, are just facing another filled voxel, meaning they would never be seen by the player. This is a whole lot of faces we dont need.

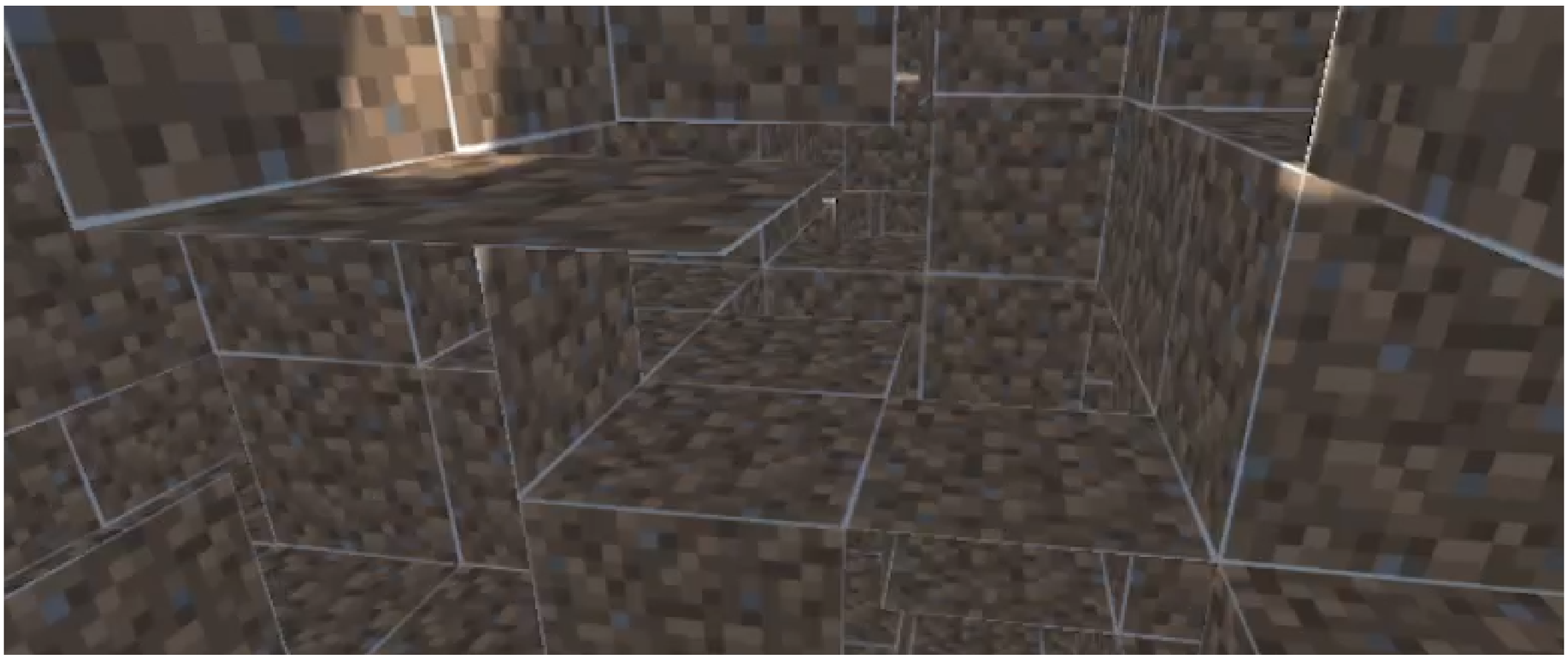


While rendering each face of a voxel, we also check the state of the surrounding voxels, making sure that the side is exposed before rendering it. If not, we skip that side and move on to the next.

This greatly improves performance, load times, and prevents the mesh maxing out.

```
// back
if (chunk[voxel+(1)] == 0) {}
// front
if (chunk[voxel-(1)] == 0) {}
// bottom
if (chunk[voxel-(chunkSize)] == 0) {}
// top
if (chunk[voxel+(chunkSize)] == 0) {}
// left
if (chunk[voxel-(chunkSize*chunkSize)] == 0) {}
// right
if (chunk[voxel+(chunkSize*chunkSize)] == 0) {}
```

Considering the chunk data is stored in a list, checking each face required these annoying functions to find the corresponding voxels. Taking completely one-dimensional data and extrapolating that into three makes my brain go a little numb, so these formulas took longer than they should have to figure out. The face rendering code isn't shown here, to highlight just the side detection code.

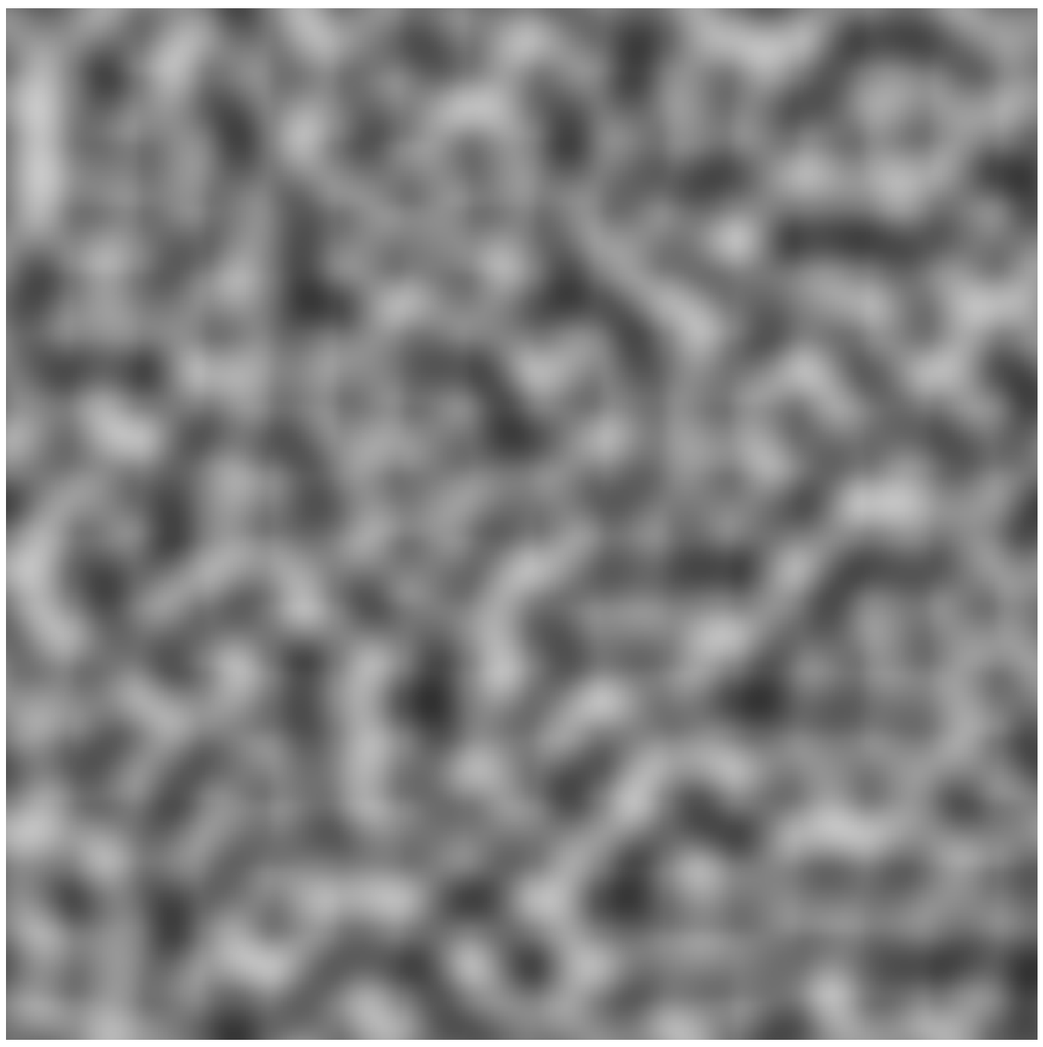


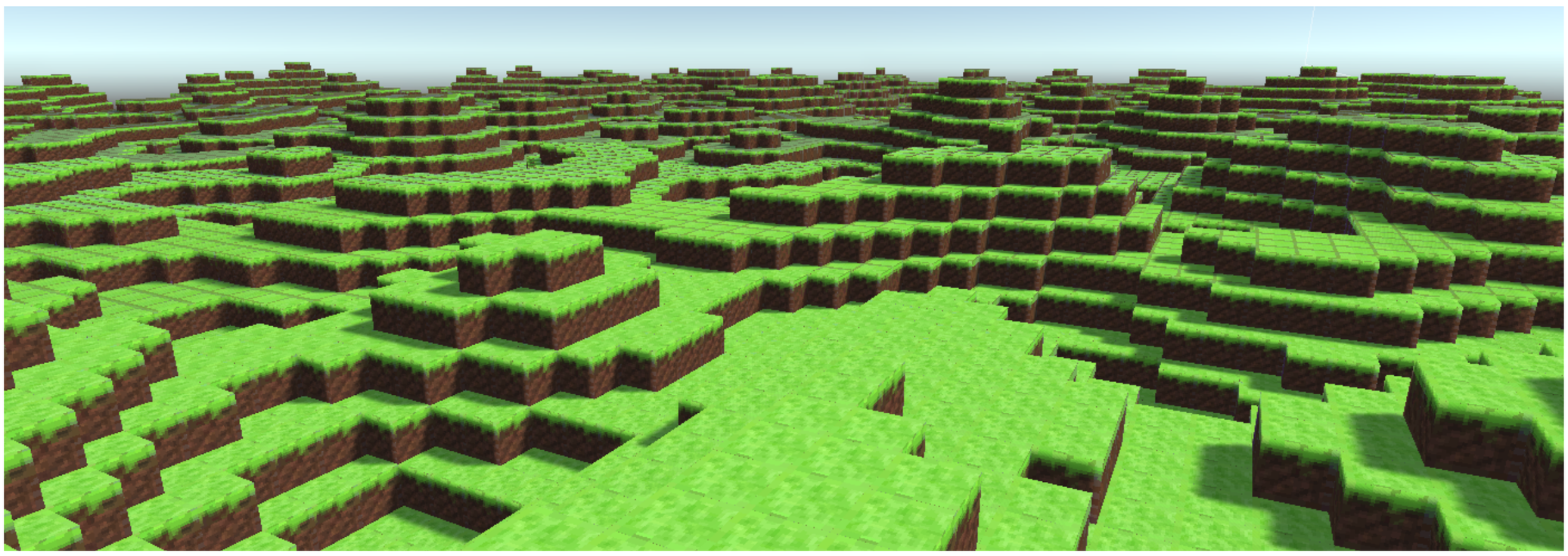
Example of faces being culled when not exposed to the outside (the camera is currently inside of a voxel so we can see this)

Attempting world generation

This, here is “perlin noise”, a type of gradient noise invented by Kevin Perlin (not super complicated, yknow).

This stuff is probably one of the best things invented in all of computer science history, being crucial for texture artists, procedural materials, particle effects, so many 2D artists, literally anything involving randomness that exceeds the first dimension. I’ve been using this countless times in my 5 or so years of getting creative on computers for an array of purposes, and in our case, we can use this to generate our lovely landscapes.

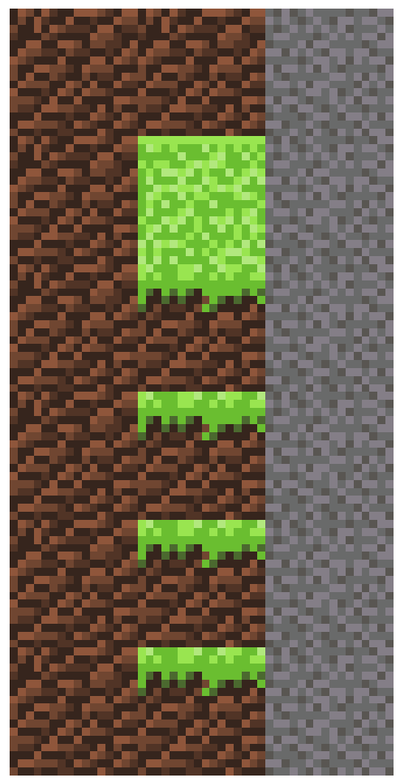




All we need to do is generate a noise texture when we load the world, and each voxel will sample that image to get the height of the land in that position. It checks if that height is above, or below itself, to determine if it should be a filled voxel. Once we impliment this, we get some grassy hills. We can also tweak the noise values to be more gentle, and get some large, flat plains with it, but that didn’t look as interesting just to test out the noise.

```
float noiseTop = Mathf.PerlinNoise(coord.x/10f, coord.z/10f);
 int id = 0;
 if ((float)coord.y / 10f < noiseTop) {id = 1;}
else {id = 0;}
```

As you can also see in the picture, I implemented different block types. These are stored in a texture atlas (seen to the right), which has every block texture in the game in a big grid. While generating blocks, each of them is assigned an “id” (code snippet only does air and dirt), which is used to determine how far along the X axis of the atlas the used texture will be, and the number of the current side being rendered will determine how far along the Y axis the used texture is. Since each block has 6 sides, and we have 3 types of blocks, that means a 3x6 grid of textures. In case of any new textures being added to the atlas, the dimensions are stored in a Vector2 value, currently being set to (3, 6), which the UV mapping code will take into account when sampling.



More optimisations

Half of this paper is going to just be optimisations, it may just get boring, but it really was most of the process. If not optimisations for speed, a lot were just workflow optimisaitons or making the code cleaner.

Instead of generating, storing, and then displaying the voxel data, we could make things faster and better by generating each block completely individually. Previously, for each chunk we would generate the block state of each voxel, before then telling the mesh renderer to visualise that. I have improved the program by making the mesh renderer determine the block state per voxel that it gets to, which is not only faster already, but also makes cleaner code, and means that we can check voxels from other chunks too, meaning that we can cull out the unseen faces on the edges of a chunk as well, being much more optimised.

You can see the cleaner code in action, since instead of the very confusing face checking formulas I did before, it's now as simple as such.

This may look longer, but it's easier to read, understand, and runs a bit faster overall.

```
// back
if (ChunkGen.GetBlockState(coord.x, coord.y, coord.z+1) == 0) {}
// front
if (ChunkGen.GetBlockState(coord.x, coord.y, coord.z-1) == 0) {}
// bottom
if (ChunkGen.GetBlockState(coord.x, coord.y-1, coord.z) == 0) {}
// top
if (ChunkGen.GetBlockState(coord.x, coord.y+1, coord.z) == 0) {}
// left
if (ChunkGen.GetBlockState(coord.x-1, coord.y, coord.z) == 0) {}
// right
if (ChunkGen.GetBlockState(coord.x+1, coord.y, coord.z) == 0) {}
```

I also moved all of the noise generation scripts from the chunk generator script, over to the mesh generator script. This means that less methods are having to be called between scripts, which is quite slow. Imagine having to go back to the library and retrieve every single page of a book one at a time just to read it. Having all your pages on you is much more efficient.

After putting it off for a while, the normal vectors had to be done properly now. Previously, I assinged all of them to -Vector3.forwards, but now the normal generation looks like:

```
for (int n = 0; n > sidesRendered*4; n++) {
    if (sideOrder[n] == 0) {normals.Add(Vector3.forwards);
    if (sideOrder[n] == 1) {normals.Add(Vector3.left);
    if (sideOrder[n] == 2) {normals.Add(Vector3.right);
```

And so on, accounting for each side. This should ensure that the lighting affects each block face correctly, since the computer can now accurately tell which direction the faces are facing.

Unfortunately, even with everything I could try to do, the more I made the world generation, the slower chunks could generate. I made the decision to keep the world at a fixed size, rather than making it infinite.

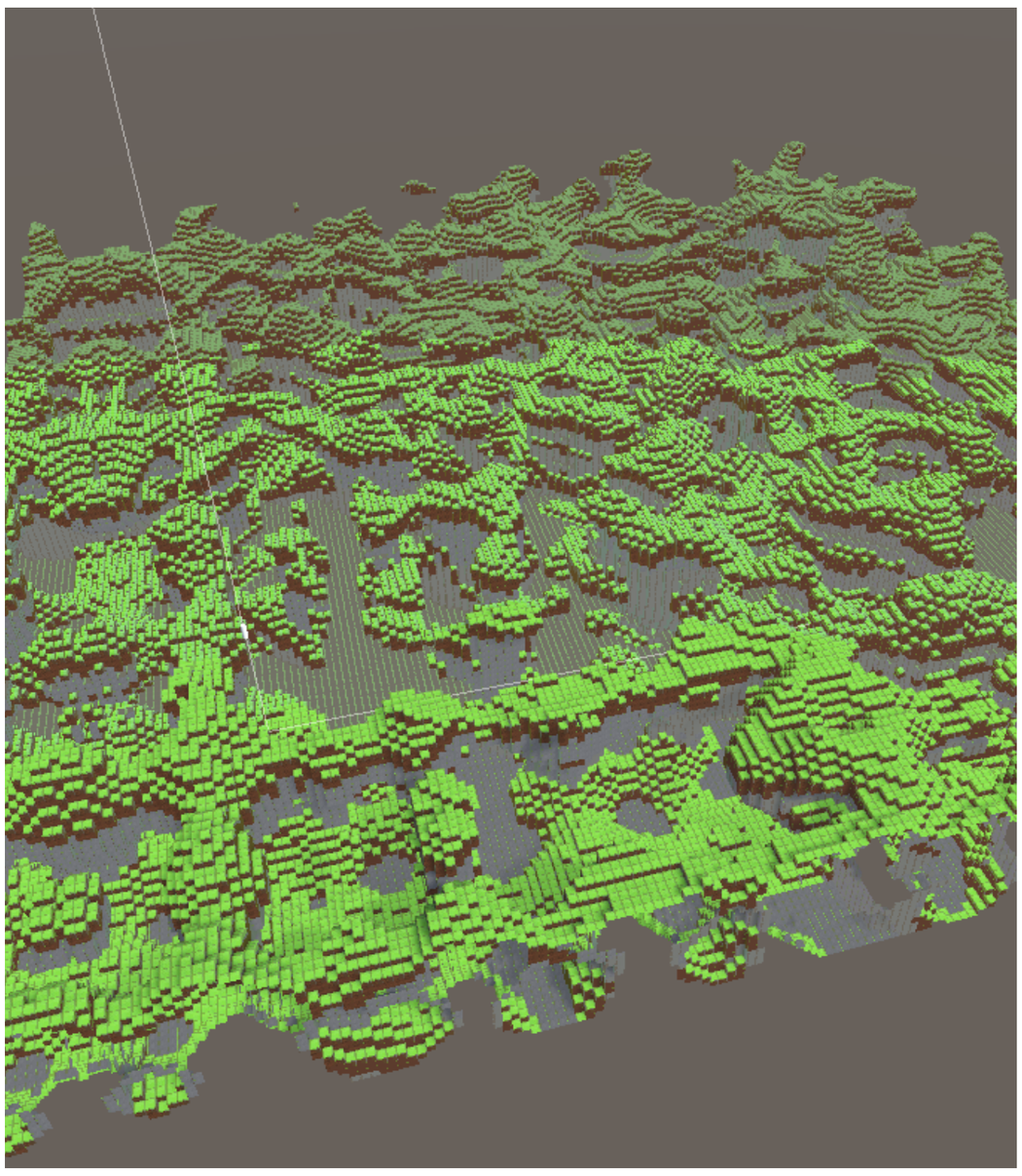
I also had to make the world not generate instantly: I added a delay between each chunk trying to render. This does technically make things slower, but your CPU trying to render every chunk at once makes the program freeze for ~20 seconds, which on some computers could just mean it crashing completely. Now it is a lot less likely to crash, and you get to see the world slowly pop in which looks quite nice.

3 Dimensional Perlin noise

We love noise, but what's better than noise? More noise, that is! By using this function I stole from StackOverflow, we can generate Perlin noise in 3D space.

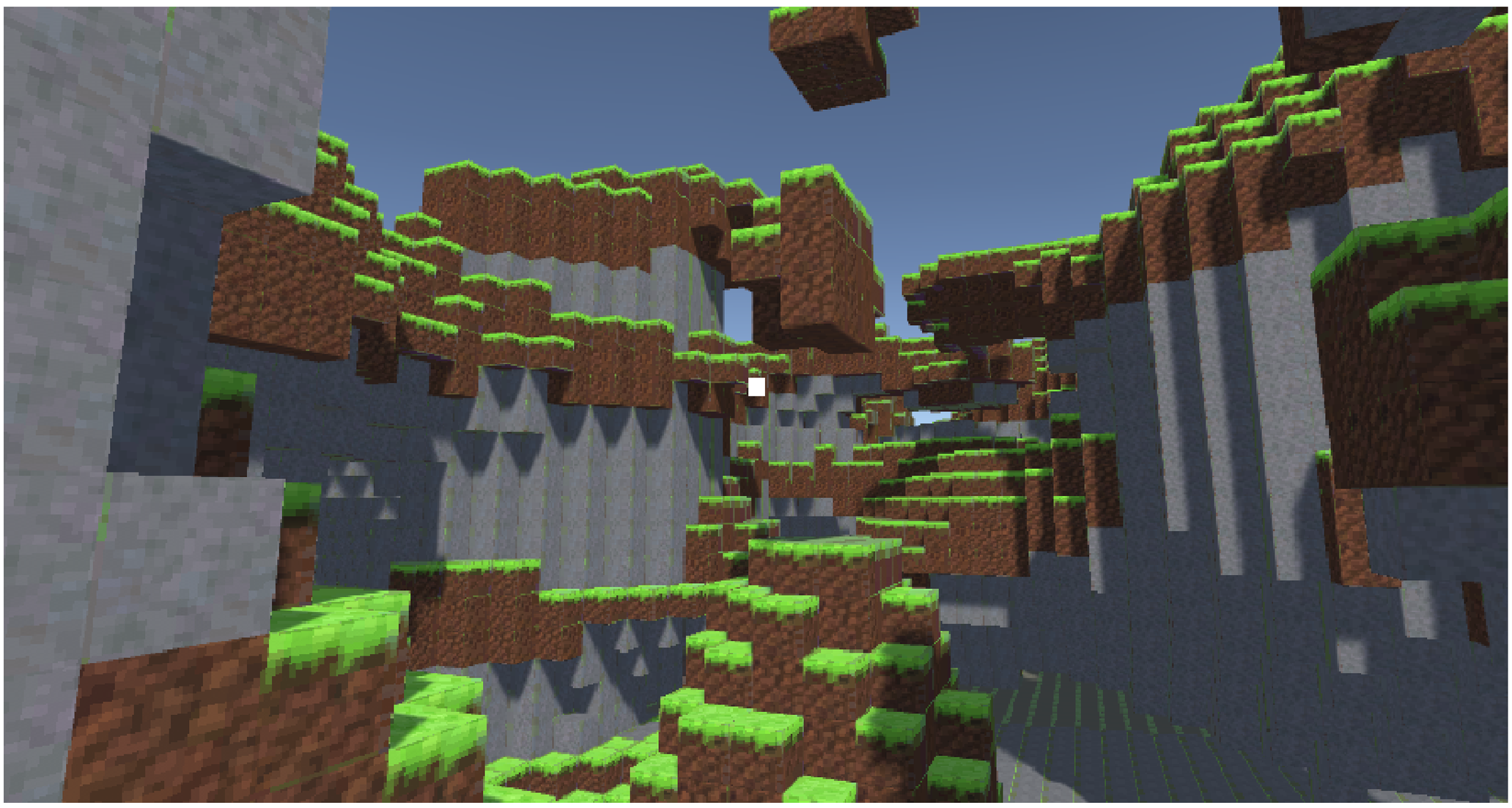
This just uses Unity's built-in 2D noise system, but does that along each axis both ways, and gives us back the mean result to get the average of all those 2D noises, appearing just like 3D noise.

```
float PerlinNoise3D(float x, float y, float z)
 {
 float xy = Mathf.PerlinNoise(x, y);
 float xz = Mathf.PerlinNoise(x, z);
 float yz = Mathf.PerlinNoise(y, z);
 float yx = Mathf.PerlinNoise(y, x);
 float zx = Mathf.PerlinNoise(z, x);
 float zy = Mathf.PerlinNoise(z, y);

 return (xy + xz + yz + yx + zx + zy) / 6;
 }
```



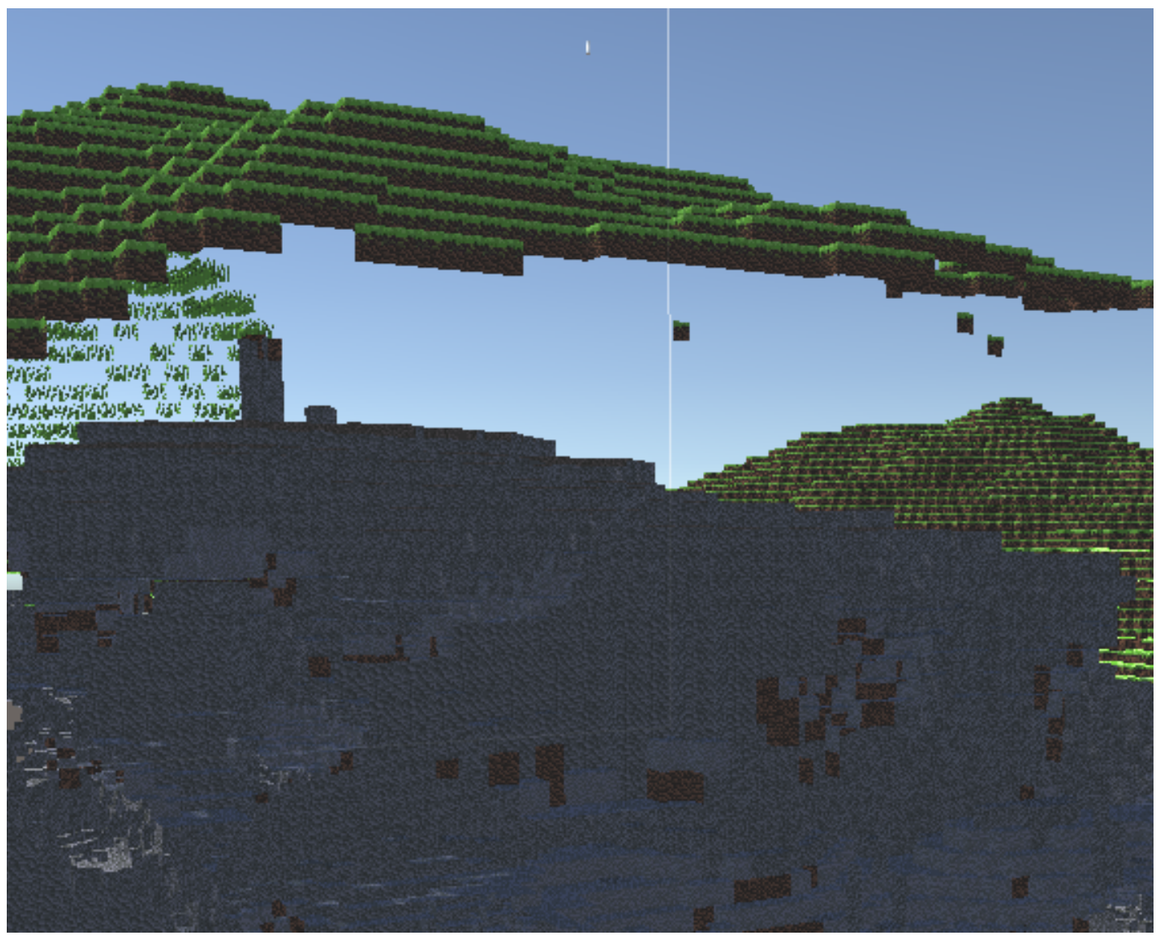
Once that we have our 3D noise, we can take this into account for our world generation. Now it just checks if the voxel intersects this new data every time a block is rendered. If we subtract the new noise from our previous voxel data, we get this funky generation. Doesn't look realistic, but with some water at the bottom and some fancy lighting, this could look nice.

```
int id = 0;
 if ((float)coord.y / 10f < noise2d &&
noise3d > 0.4f)
 {id = 1;}
 else
 {id = 0;}
 return id;
```



Better world generation

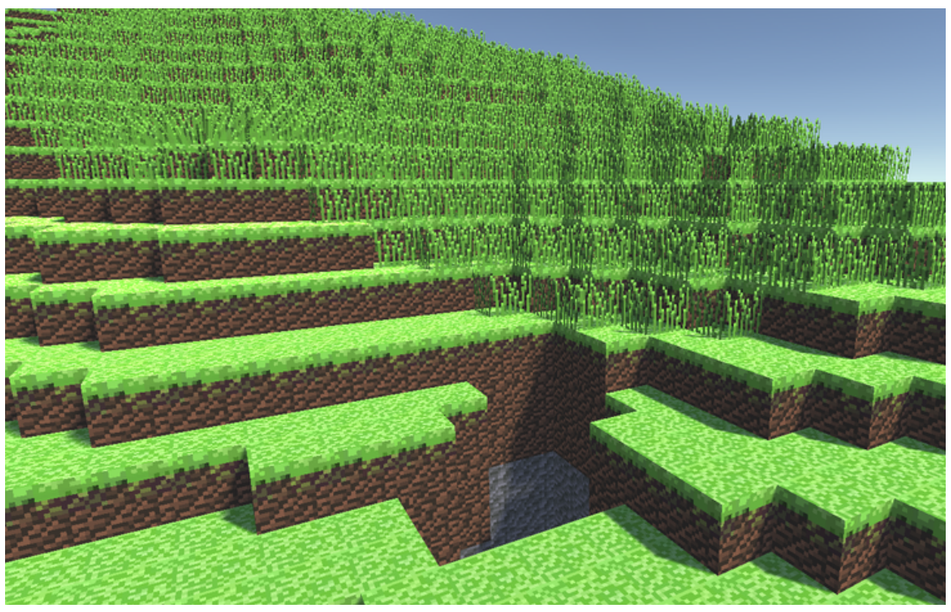
Using this 3D noise, we can generate caves! What's Minecraft without them? Previously, we used the 3D noise to subtract from the terrain and made... something - but that didn't look very nice. What I did was two steps: firstly, I lowered the threshold for it, making less of the noise leaking in actually subtract anything, and secondly, I made sure to check if it was too close to the surface, and not let it subtract any further if it was too close. Although, I did use another noise map to determine sections that were allowed to pass through anyway, giving us some cave entrances. You can see the cutoff on the right.

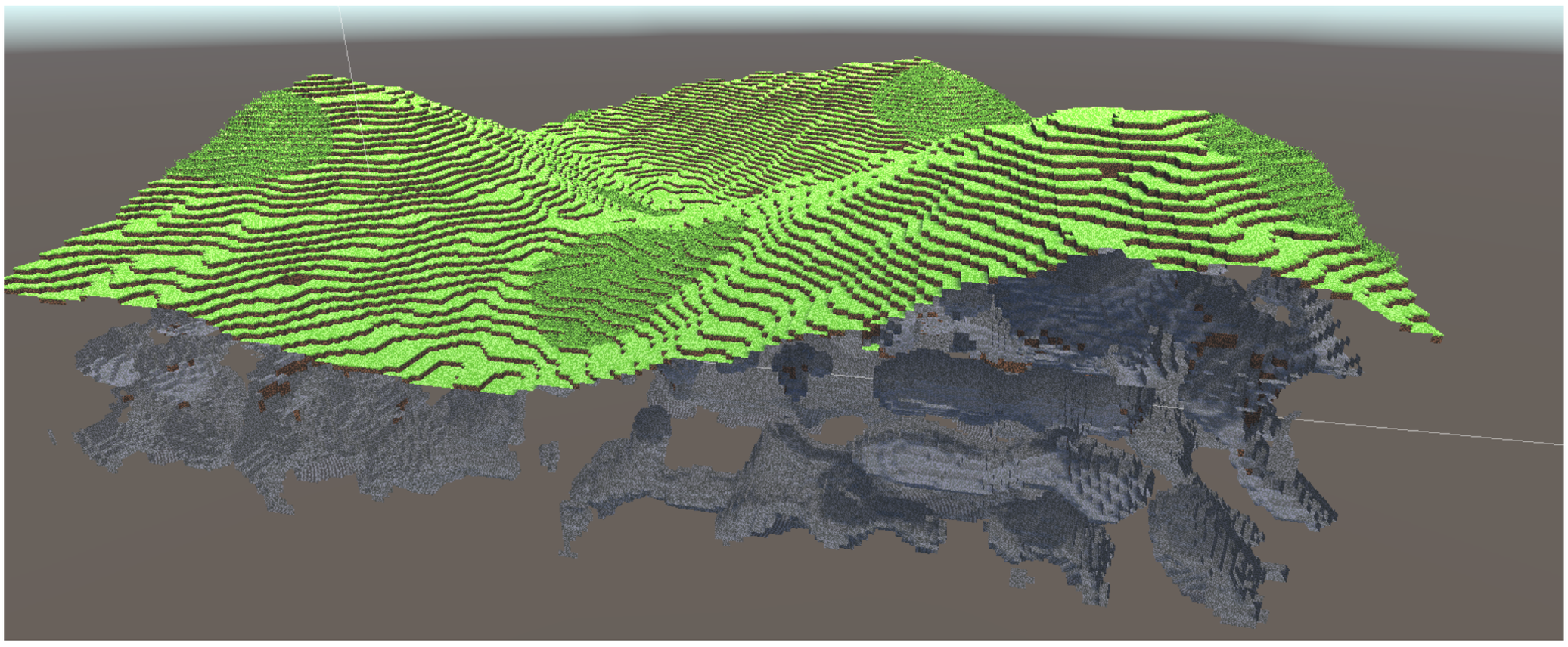


Caves being cut off for being too close to the surface

I've also made changes to the height of the terrain too, as you can tell below. Now we have two noise passes: noiseMount and noiseDetail. NoiseMount is a very gradual, and very exaggerated noise map which determines the general mountain-ness of the world, while noiseDetail is smaller and less exaggerated, which does the finer details of the ground, making everything a bit bumpier than otherwise. Adding both of these noise maps together makes for a more varied terrain, as some voxel engines you'll see online forget to do, making very boring terrain.

There's even tall grass now! This is the first transparent block type, and required extra exceptions in the optimisation code to account for it. These generate on top of any grass voxels, and also use their own noise map, meaning they spawn in clumps like so. You can also see an example of a cave entrance in the image on the right.





Shortcomings

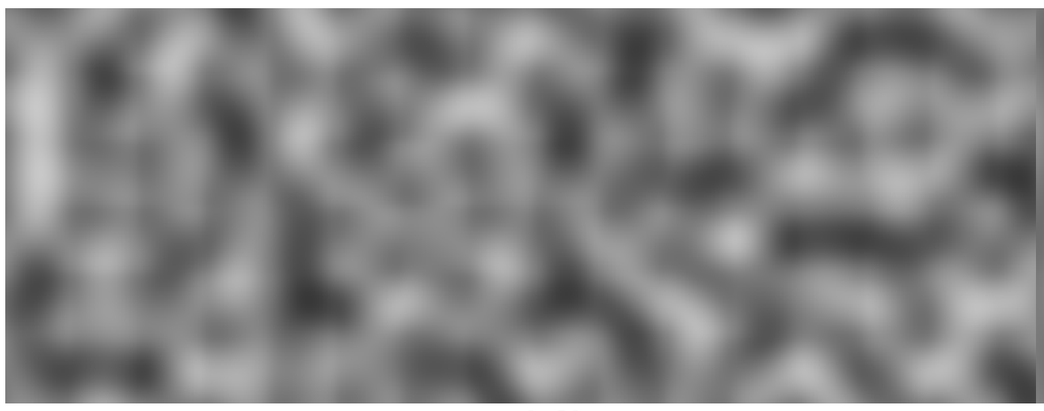
There are many things I wanted to do with this project which I simply couldn’t get done in time. Some required redoing too much of the engine for them to be percievable for this project.

Trees were the first roadblock I came across. Genuinely had no idea how to make them. They’re the first thing I tried to do that weren’t just:

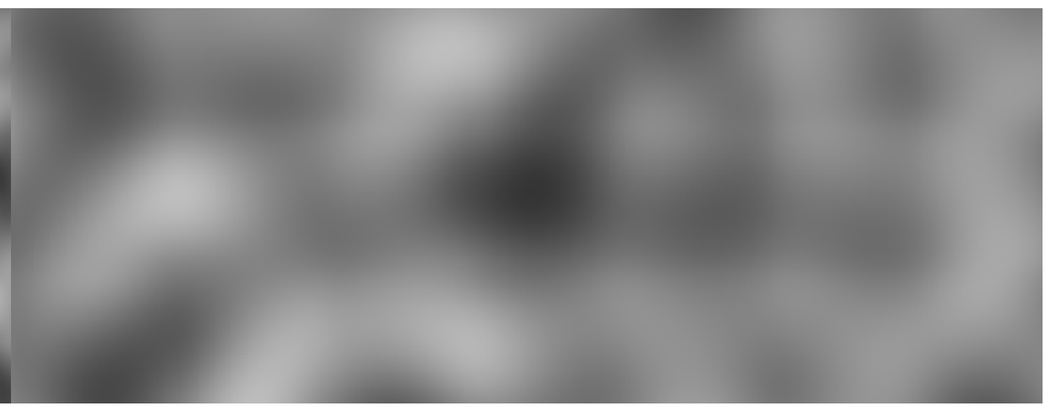
```
if (noise == something)
   {
      block = dirt;
   }
```

I tried researching voxel generation some more, but couldn’t find anything that wasn’t either too oversimplified, only covered things I’ve already done, or were too overcomplicated. I even tried to talk to other people making C# voxel implimentations but they were too famous for me I guess. For example, the developer of PlanetSmith (a voxel game made in Unity that actually has spherical worlds) did have a public Discord server but his DMs were turned off and I got muted for trying to ping him. I’m not super sure what the point of his Discord was: just to talk to random people about his game? And his account is right there but you’re not allowed to talk to him?

The main issue with biomes was that I couldn’t get them to blend together properly. Markus Persson himself (creator of Minecraft) did publish a research paper on his approach to biomes, where he had two new noise maps for heat and humidity, where each biome type would be assigned to a different combination of values. The gradual change in heat and humidity would make biomes that world next to eachother actually generate near eachother, so you wouldn’t end up seeing a tundra right next to a desert.



Humidity



Heat

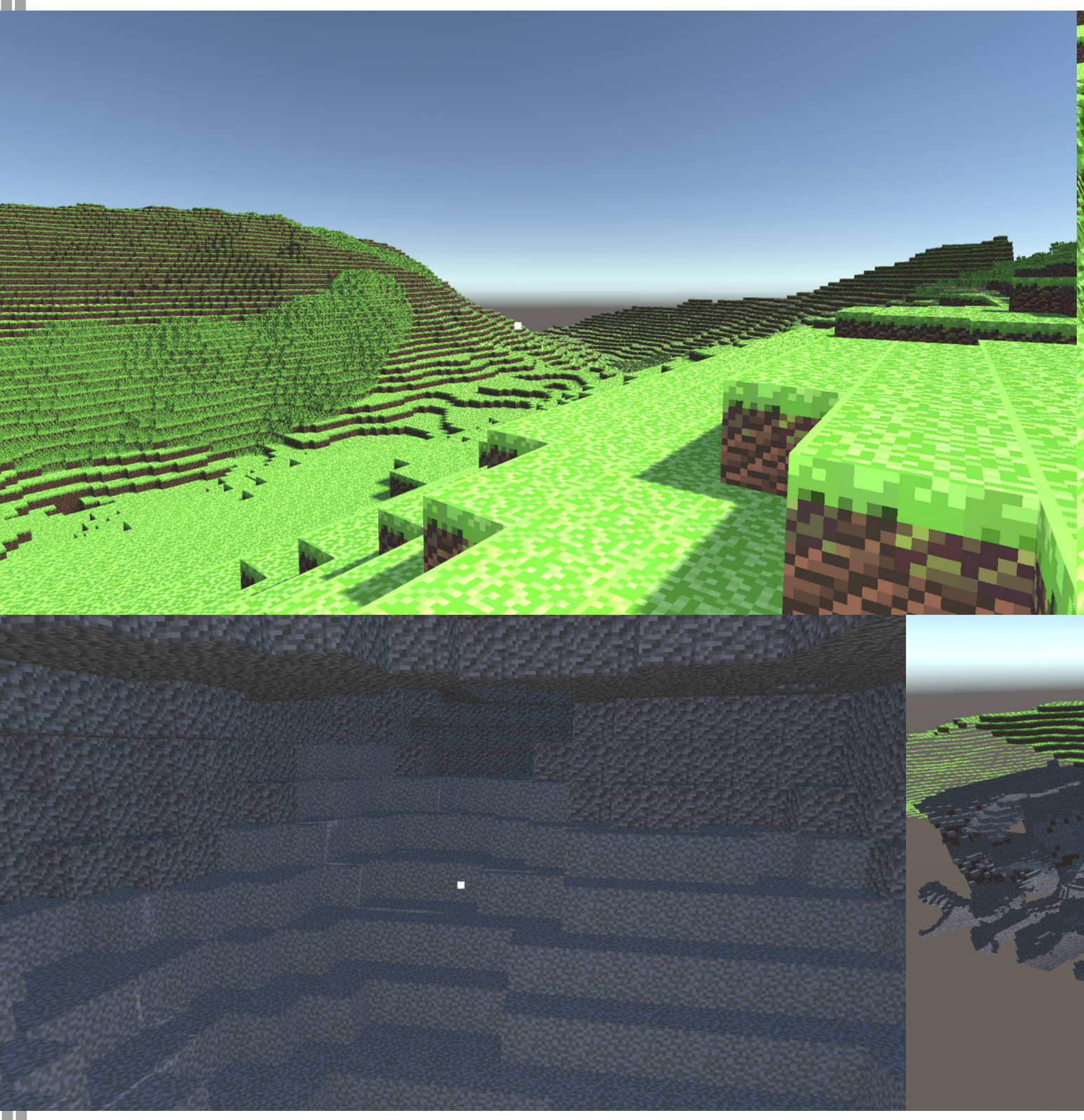
I probably could’ve gotten some sort of implimentation of biomes, but I couldn’t figure it out in time.

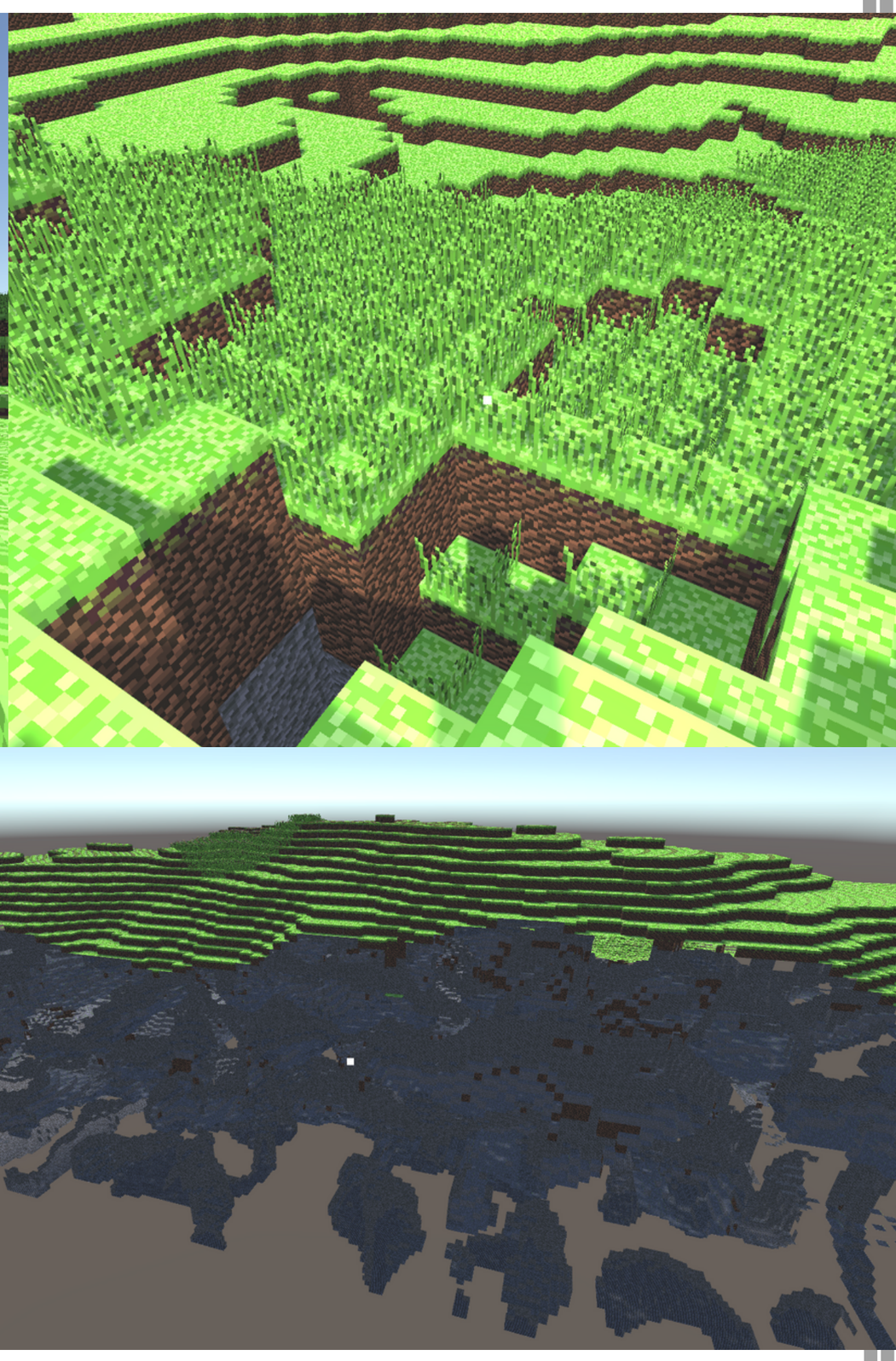
Water. I don’t think I need to explain that much why water was out of the question. Even having it static and take up full blocks alone would require a rewrite of the optimisation code, accounting for not just transparent blocks but having the borders of the clumps of those transparent blocks be accounted for.

Infinite worlds was also something I had to scrap. Each chunk generating took up so much CPU power that it would freeze the game for a solid half a second or so. Having such a lag spike every time you move 16 blocks is an extremely unpleasant experience. Deciding on a set world size was just a better idea for everyone involved (me).

Results

Screenshots:





Source code:
https://github.com/idoblenderstuffs/crappy-mc-terrain

WebGL build:
https://idoblenderstuffs.github.io/crappy-mc-terrain/webgl/

Windows build:
https://idoblenderstuffs.github.io/crappy-mc-terrain/windows/

Sources used

Generating Meshes at Runtime in Unity:
https://blog.yarsalabs.com/generating-mesh-in-runtime-unity/

Perlin Noise on Wikipedia (thanks for the generation code):
https://en.wikipedia.org/w/index.php?title=Perlin_noise&oldid=1213379963

Voxel World Optimisations:
https://vercidium.com/blog/voxel-world-optimisations/