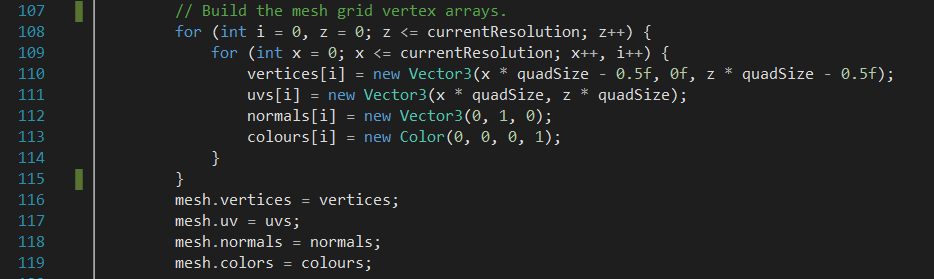
**Terrain Genesis**

**Introduction**

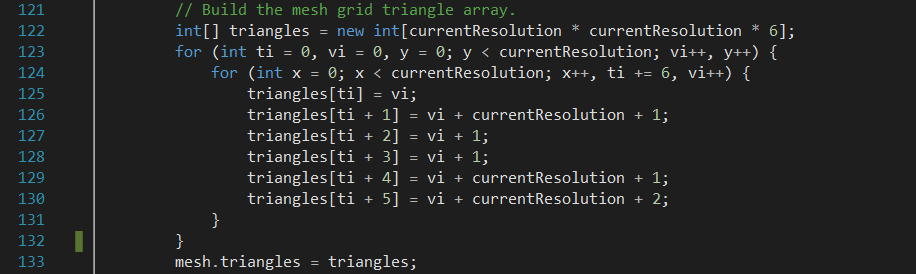
For this coursework I focused on generating a scalable mesh that could be combined with noise generation to create different terrains. This report will act as a guide to the code and will cover the key areas of the project.

**Mesh Generation – SurfaceGenerator.cs – CreateMeshGrid()**

I began by first generating a grid of quads to act as the mesh that the noise would later be applied to. This is done by first building arrays for the mesh’s vertices, uvs, and normals. I also created a colour array to distinguish the mesh from the rest of the scene. This array was later overwritten as I went on to colour each quad corresponding to their noise value and thus height.



Similarly, the second step involves constructing the triangle array for the mesh. All of these arrays are assigned to the mesh object.

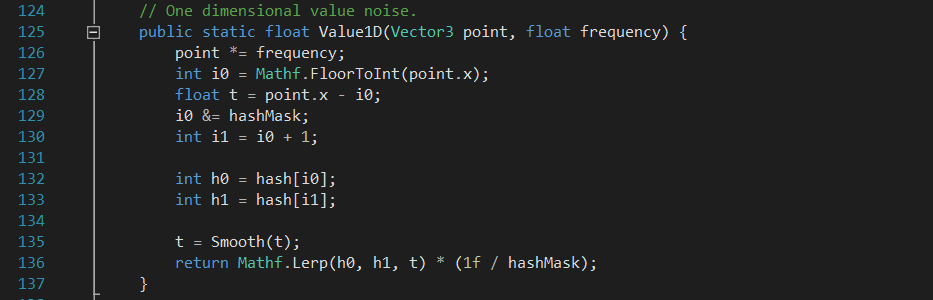


These two steps result in a simple surface mesh which is scalable by altering the resolution variable.

**Noise Generation – Noise.cs**

**Value Noise**

I chose to implement value noise as a quick and simple method to test on the mesh. For one dimensional value noise this was achieved by assigning random values to the x coordinates of the mesh, and interpolating between the adjacent lattice points. These random values are assigned through the use of hashing via a permutation table.



It was then trivial matter of expanding the code to incorporate the other axis to produce two and three dimensional value noise.

**Perlin Noise**

While value noise produces adequate results, its appearance retains a blocky unnatural feel. Because of this downside I chose to implement a type of gradient noise which would produce a much more chaotic feel. I decided that Perlin noise would be suitable for this application for a few reasons:

1. I didn’t intend to go into dimensions greater than the third dimension,
2. the application was unlikely to be resource intensive,
3. and I had previously spend time working with simplex noise.

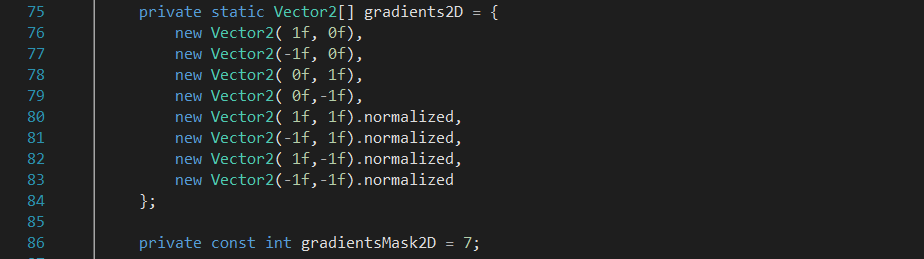
For the purpose of this report I am going to focus on two-dimensional Perlin noise. As before with the one-dimensional value noise, it is relatively straight forward to apply this technique to one dimension and three dimensions. Noise.cs contains a one, two, and three dimensional method for both value and Perlin noise.

Perlin noise generation can be split up into three key steps:

1. Grid definition
2. Dot product
3. Interpolation

**Grid Definition**

This step involves defining an n-dimensional grid and assigning each grid node a random gradient vector. For this project the mesh is acting as the grid. I decided to use arrays of precomputed vectors to reduce the cost of computing new vectors for each grid node.



I used a combination of axis-aligned and diagonal vectors to reduce any alignment influences in the resulting noise. The diagonal vectors are normalised to ensure their magnitude is the same as the axis-aligned vectors.