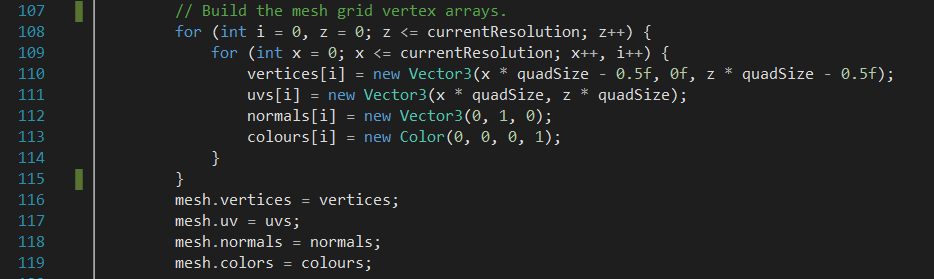
**Mesh Terrain Genesis using Noise – Adam Joyce**

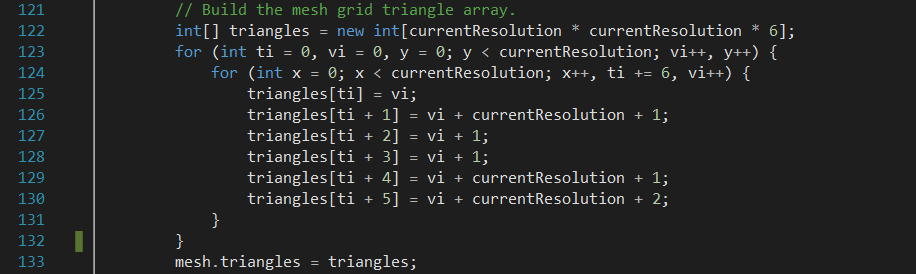
For this coursework I focused on generating a scalable mesh that could be combined with noise generation to create different terrains. This report will act as a guide to the code and will cover the key areas of the project.

**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 the quads based on their noise value and thus height.



Similarly, the second step involves constructing the triangle array for the mesh. These arrays are assigned to the scene’s mesh object.

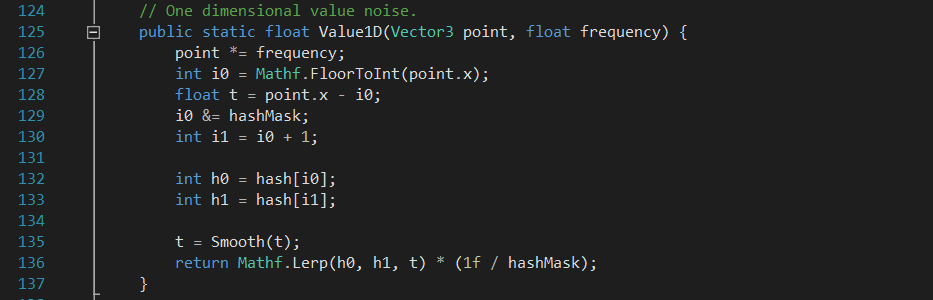


These two steps result in a simple surface mesh which is scalable by altering the resolution variable.

**Noise Generation – Noise.cs**

**Value Noise – Value1D(), Value2D(), Value3D()**

I chose to implement value noise as a quick and simple method to test noise 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 axes to produce two and three dimensional value noise.

**Perlin Noise – Perlin1D(), Perlin2D(), Perlin3D()**

While value noise produces adequate results, its appearance retains a blocky unnatural structure. 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.

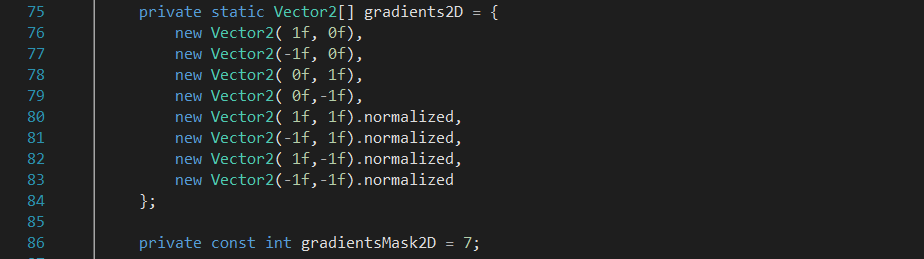
For the purpose of this report I am going to focus on two-dimensional Perlin noise. As before with the one-dimensional value noise, it is relatively straight forward to apply this technique to one dimension and three dimensions. Noise.cs contains a one, two, and three dimensional method for both value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

**Dot Product**

The next step is to determine which grid cell the point falls in. The relative position of the point in the cell is calculated, meaning distance vectors between the point and each of the cell corners can be found. For each corner node the dot product between the gradient vector and that node’s distance vector is found, and thus the influence of each corner on the point.

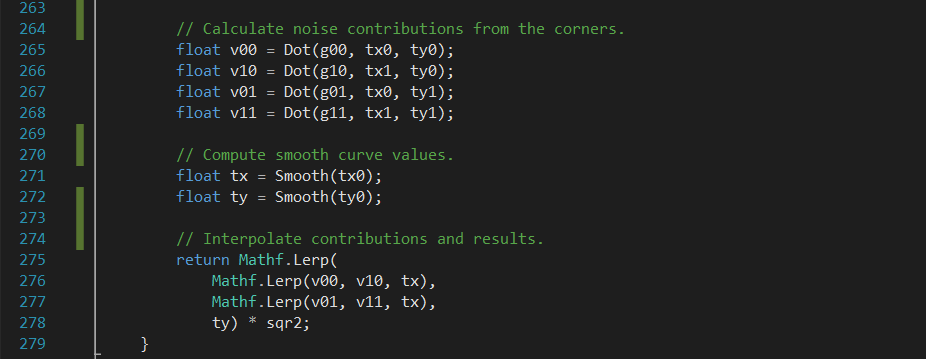
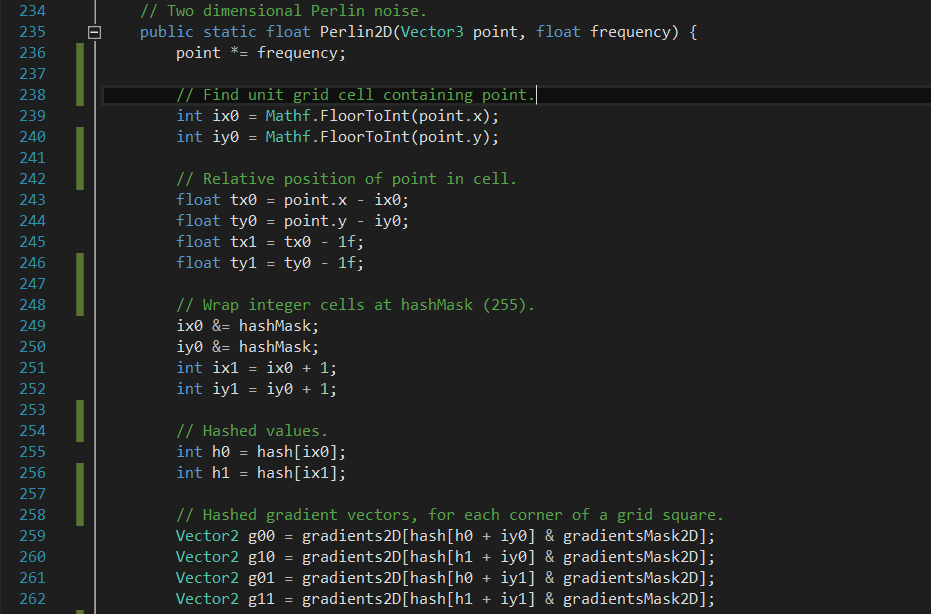
**Interpolation**

The final step is to interpolate between the dot products of each of the node corners. To avoid sharp transitions I run the interpolants – the x and y position of the point relative to that cell – through a smoothing function before the interpolations take place.

The implementation of Perlin2D() can be found on the next page.

**Fractal Noise – Sum()**

The final method in Noise.cs turns our single sample noise into fractal, or ‘layered’ noise. It does this by adding together several samples, or ‘octaves’, of noise that have different frequency and amplitudes. In Sum() I use the variables ‘lacunarity’ and ‘persistence’ to increment and decrement the frequency and amplitude respectively for each successive octave. This provides further customisation of the mesh terrain.

*Perlin2D() Implementation:* 

**Class and Method Overview**

In this section I detail some information about the remaining classes and their methods that have not been covered so far in this report.

***SurfaceGenerator.cs***

**Refresh()**

This is used as a main method to update the terrain mesh whenever a change is made. It calls important methods to both create the mesh and generate the noise which it then applies to the mesh. It also takes into account various setting that effect how the mesh and noise is created and displayed.

**CreateMeshGrid() – Covered previously in report.**

This method deals with generating the basic mesh for the scene.

**OnEnable()**

Ensures that the surface game object has a mesh object applied to it and adds a mesh collider to allow basic sculpting functionality.

**OnDrawGizmos()**

Used to display the normal of each triangle in the mesh when gizmos are turned on in the game view.

***Noise.cs***

**Dot() - Covered previously in report.**

Calculates the dot product between inputs. The function is overloaded with both a two and three dimensional variation.

**Smooth() - Covered previously in report.**

A smoothing function.

**Value1D(), Value2D(), Value3D() - Covered previously in report.**

Implementations of value noise methods for one, two, and three dimensions.

**Perlin1D(), Perlin2D(), Perlin3D() - Covered previously in report.**

Implementations of Perlin noise methods for one, two, and three dimensions.

***GUIControls.cs***

This script handles the OnValueChanged() and OnEndEdit() callbacks from the GUI elements in the scene.

***CameraControls.cs***

This script allows the user to rotate the camera around the mesh terrain by clicking and holding the right mouse button. It also manages the second camera in the scene which is setup to render the mesh terrain in wireframe.

***Wireframe.cs***

This script has the camera it is attached to render all objects on the wireframe layer in wireframe mode.

***Sculptor.cs***

Very basic triangle sculpting manipulation. Use by clicking and holding with the left mouse button on the mesh terrain.

***SurfaceGeneratorEditor.cs***

An editor class used to update the mesh terrain in the scene view immediately.

**Future Development**

Due to other project and funding commitments I did not reach the stage I would have like to with this project. Going forward I would focus on fleshing out the sculptor script to allow for more comprehensive brush capabilities. I would change it from simply transforming the vertices to editing the noise values instead. This would allow for changes in colour depending on the triangles elevation.

Secondly, I would look into methods of rendering the normals of the triangles in the deployed application – something that is not possible with either my OnDrawGizmos() or my commented out DrawNormals() methods. This could be achieved using Unity’s built-in LineRenderer or more efficiently with OpenGL.

**Video Link**

<https://youtu.be/CCmEj2lIfjU>

**References**

<http://docs.unity3d.com/Manual/GeneratingMeshGeometryProcedurally.html>

<https://en.wikipedia.org/wiki/Perlin_noise>

<https://en.wikipedia.org/wiki/Simplex_noise>

<http://catlikecoding.com/unity/tutorials/noise/>

<http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf>