Terrain Generator Research Report

Introduction

With today's technology, the ability to create vast landscapes in video games is more readily available than ever. Engines such as Unreal 4 and Unity have built-in editors for creating levels with terrain and there are several algorithms that can be used to generate terrain procedurally. The goal of this project was to implement a 3D simulation capable of producing terrain from a given heightmap, as well as procedurally generating it with Perlin noise. These methods of terrain generation are useful depending on what type of game is being produced. A game that has one map can store its world's topological data in a heightmap (or multiple heightmaps for really large worlds) and read from that data to load the terrain during runtime. Triple-A role-playing games are likely to want to use heightmaps because of this and examples that do use them are Skyrim and The Legend of Zelda: Breath of the Wild. In contrast to this, procedural generation is used more in games that want to create large "infinite" worlds or quickly produce varying levels from a set of parameters. Games that procedurally generate their worlds include: Worms, Minecraft and Spore.

Research

Rendering Terrain

The most important part of generating terrain is being able to render it to the screen. In computer graphics a terrain is represented with a surface mesh that is made up of lots of vertex positions. These vertices are stored in triangles and they hold data such as position, normal and texture coordinates that are used to render the terrain's mesh model during runtime. For small-scale terrains this is enough to generate what is needed, however for larger meshes with millions of vertices additional procedures are required to keep performance reasonable. Level of Detail (LOD) methods such as geomipmapping [1] and real-time optimally adapting meshes (ROAMs) [2] can be used to alleviate this problem.

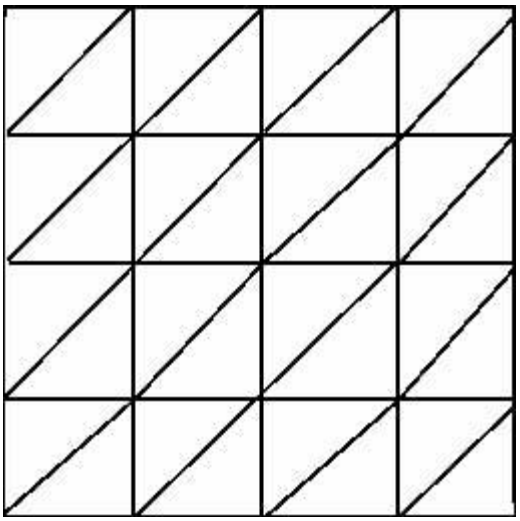


Figure 1. *Example of the grid layout formed from a set of vertices*

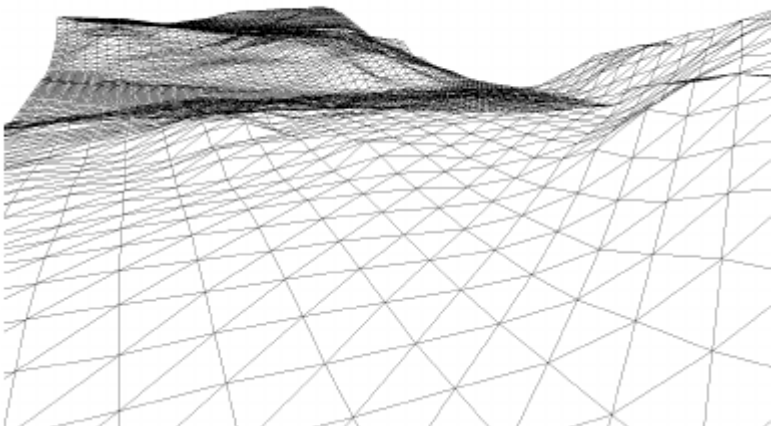


Figure 2. *A terrain-like mesh created from a grid of vertices with differing height values*

Heightmaps

Heightmaps are images that store the height values of each terrain grid node in their pixels; they are usually represented as a greyscale bitmap with the heights ranging from 0-255. Heightmaps save a lot of memory space compared to storing the terrain model as a mesh since the important height data is compressed into a single byte for each pixel and any extra mesh specific data can be ignored. They also are easy to edit, since all the developer needs to do is change the pixel values. The downsides to using heightmaps are that they cannot produce overhangs such as ledges and caves and textures can stretch when the terrain has a steep gradient like when there is a cliff.

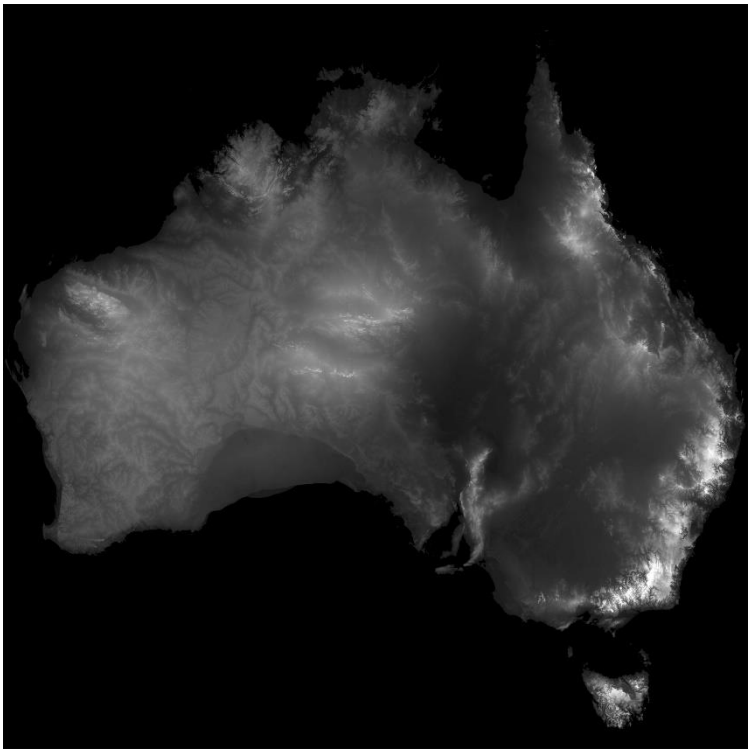


Figure 3. *Satellite image of Australia converted to a greyscale heightmap*

Perlin Noise

There are several different types of noise that can be procedurally generated. Perlin noise is a type of gradient noise that uses a series of pseudorandom gradient vectors to generate heightmaps. The algorithm for this noise was first written in the 1980s by Ken Perlin [3], however it has since undergone some changes to improve its computational speed [4].

To generate Perlin noise, first a unit square (or unit cube in 3D) is created. Each vertex of the square is then assigned a pseudorandom gradient vector. Next a coordinate point that lies within the square is given as the input. The distances between the coordinate and each point of the square are then calculated to produce a distance vector. After that the dot product of the gradient vectors and distance vectors are calculated to produce a set of influence values. These values affect the levels of noise at different parts of the grid. The influence values are linearly interpolated to obtain an average weighting between the grid

points, then finally a fade function is applied to the results to smooth the changes in the values produced. In a grid the Perlin noise is calculated for each cell to produce an array of noise values. These values can then be written to a heightmap to be read later.

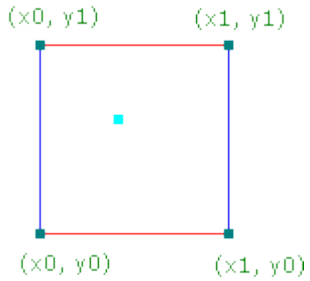


Figure 4. *Unit square with input coordinate inside (the blue square)*

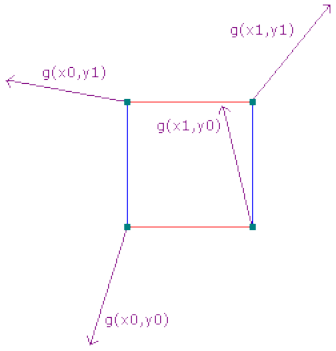


Figure 5. *Gradient vectors applied to each node of the grid*

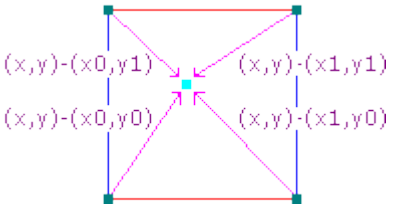


Figure 6. *Calculate the distance vectors from each corner to the input coordinate*

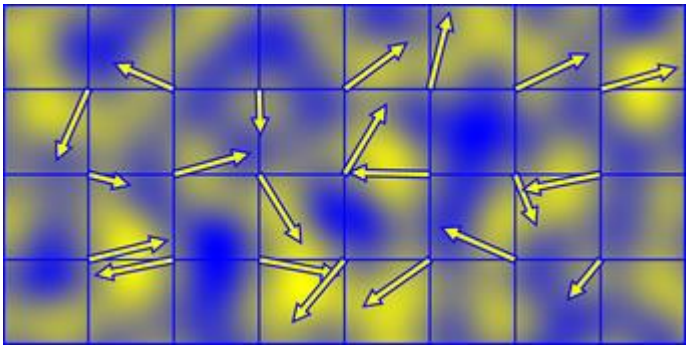


Figure 7. *Influence values shown affecting the noise. Yellow shows positive noise (from the direction the values point in) while blue shows negative accumulation. [5]*

Figure 8. *Example noise generated using Perlin's algorithm*

Generating Perlin noise for terrain heightmaps is usually done in 2D, however the algorithm is scalable to other dimensions. 3D noise can be used to generate animated heightmaps by setting the z component of the input coordinate to be a changing value such as a time delta [6].

Minecraft makes use of Perlin noise to generate its worlds from a given seed. Markus 'Notch' Persson, creator of Minecraft, wrote a technical post for his Tumblr blog that discussed some of the earlier versions of Perlin noise he used to generate Minecraft's landscapes.

"In the very earliest version of Minecraft, I used a 2D Perlin noise heightmap to set the shape of the world. Or, rather, I used quite a few of them. One for overall elevation, one for terrain roughness, and one for local detail." Persson (2011) [7]

Since 2D Perlin noise be used cannot generate overhangs, Persson later switched to using a combination of both 2D and 3D Perlin noise to produce the terrain that Minecraft currently generates.

Figure 9. *In-game screenshot from Minecraft showing overhanging landscape generated with Persson's adapted Perlin noise implementation*

Simplex Noise

Simplex noise was created by Ken Perlin in 2001 as a better alternative to his original noise algorithm. This algorithm has much less computational complexity than classic Perlin noise, meaning it can scale up to higher dimensions much easier; the complexity for simplex noise is $O(N^2)$ whereas classic Perlin noise has a complexity of $O(2^N)$.

In 2D, simplex noise first takes a grid and input coordinate and skews them to produce a rhombus made up of two equilateral triangles (called simplexes). Then, similarly to classic noise, the skewed coordinate is used to find where the input point lies within the new grid shape. Checking which magnitude is greater helps determine which simplex the coordinate lies in. In Figure 10 a coordinate with a greater x value is located in the lower simplex and vice versa for greater y values. Once this has been determined, the vertex points of the simplex where the coordinate is in are added to the skewed base coordinate. This result is then hashed into a pseudo-random gradient vector. The gradients are usually stored in a pre-made permutation array, this is similar to what Perlin did for his classic noise algorithm. The final stage of the algorithm calculates the contributions from the three corners of the simplex. This is done by first summing the unskewed coordinates of each of the three vertices and subtracting them from the input coordinate to get an unskewed displacement vector, then using the result along with the dot product between the displacement vector and the gradient vector to get the final contribution value. These values are added together (and normalised to be between -1 and 1) to return the final noise value. Figure 11 shows an implementation of simplex noise in Java. Stefan Gustavson wrote a paper in 2005 to better explain how simplex noise works [8].

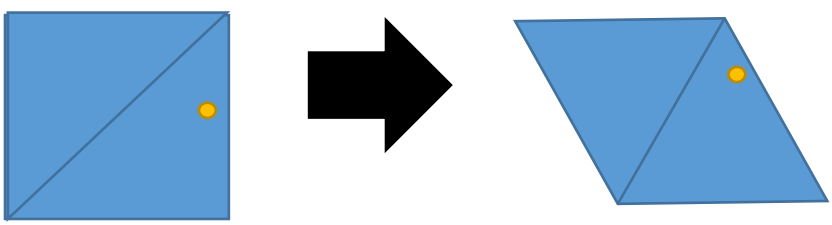


Figure 10. *Example of what a single grid cell looks like after skewing. The input coordinate is represented by the yellow circle.*

Although simplex noise provides better runtime performance than classic Perlin noise, it is still seen less often in commercial applications. This is mostly due to the patent that Ken Perlin owns for his simplex algorithm which prevents its use with textured image synthesis in 3D or higher [9]. It is easier for developers to just use a different algorithm such as classic Perlin noise and avoid a potential lawsuit. Another reason that simplex noise is not seen as often as classic noise could be to do with how well understood its implementation is. Classic noise is much easier to visualise and it has been around much longer, making it a more widely understood method for generating terrain.

```
// 2D simplex noise
public static double noise(double xin, double yin) {
  double n0, n1, n2; // Noise contributions from the three corners
  // Skew the input space to determine which simplex cell we're in
  double s = (xin+yin)*F2; // Hairy factor for 2D
  int i = fastfloor(xin+s);
  int j = fastfloor(yin+s);
  double t = (i+j)*G2;
  double X0 = i-t; // Unskew the cell origin back to (x,y) space
  double Y0 = j-t;
  double x0 = xin-X0; // The x,y distances from the cell origin
  double y0 = yin-Y0;
  // For the 2D case, the simplex shape is an equilateral triangle.
  // Determine which simplex we are in.
  int i1, j1; // Offsets for second (middle) corner of simplex in (i,j) coords
  if(x0>y0) {i1=1; j1=0;} // lower triangle, XY order: (0,0)->(1,0)->(1,1)
  else {i1=0; j1=1;}      // upper triangle, YX order: (0,0)->(0,1)->(1,1)
  // A step of (1,0) in (i,j) means a step of (1-c,-c) in (x,y), and
  // a step of (0,1) in (i,j) means a step of (-c,1-c) in (x,y), where
  // c = (3-sqrt(3))/6
  double x1 = x0 - i1 + G2; // Offsets for middle corner in (x,y) unskewed coords
  double y1 = y0 - j1 + G2;
  double x2 = x0 - 1.0 + 2.0 * G2; // Offsets for last corner in (x,y) unskewed coords
  double y2 = y0 - 1.0 + 2.0 * G2;
  // Work out the hashed gradient indices of the three simplex corners
  int ii = i & 255;
  int jj = j & 255;
  int gi0 = permMod12[ii+perm[jj]];
  int gi1 = permMod12[ii+i1+perm[jj+j1]];
  int gi2 = permMod12[ii+1+perm[jj+1]];
  // Calculate the contribution from the three corners
  double t0 = 0.5 - x0*x0-y0*y0;
  if(t0<0) n0 = 0.0;
  else {
    t0 *= t0;
    n0 = t0 * t0 * dot(grad3[gi0], x0, y0);  // (x,y) of grad3 used for 2D gradient
  }
  double t1 = 0.5 - x1*x1-y1*y1;
  if(t1<0) n1 = 0.0;
  else {
    t1 *= t1;
    n1 = t1 * t1 * dot(grad3[gi1], x1, y1);
  }
  double t2 = 0.5 - x2*x2-y2*y2;
  if(t2<0) n2 = 0.0;
  else {
    t2 *= t2;
    n2 = t2 * t2 * dot(grad3[gi2], x2, y2);
  }
  // Add contributions from each corner to get the final noise value.
  // The result is scaled to return values in the interval [-1,1].
  return 70.0 * (n0 + n1 + n2);
}
```

Figure 11. *Java implementation of Perlin's simplex noise algorithm [10]*

Weathering and Erosion

A final way of generating terrain makes use of realistic geographical processes to shape the mesh. An article in *Computer Graphics Forum* showed how the formulae used to calculate real-world tectonic uplift, the rate at which a terrain's height increases each year, as well as rate of erosion by water can be used to generate very realistic looking terrain [11]. Despite its detail however, the application of these calculation intensive formulae to a game engine would put significant stress on runtime performance making its use in this project less desirable. Video games rarely (if ever) aim to be one-hundred percent true to the real-world anyway, so using these generation methods could prove to be a restriction to the level designer's creativity.

For this project Perlin noise was chosen to be implemented because it is the most widely used algorithm for generating heightmaps and it is flexible with how it can be applied. Simplex noise was considered however the complexity of the algorithm was deemed beyond the scope of this project. The implementation was also designed to be able to read from and write to heightmaps to allow flexibility in what terrains it can generate.

Implementation

VBTerrain Class

```
{
public:
    VBTerrain() {};
    virtual ~VBTerrain();

    //initialise the vertices for a flat grid
    void init(ID3D11Device* _GD);

    //General functions
    void raiseTerrain();
    void normaliseHeightmap();
    void initialiseNormals();
    void buildMesh(ID3D11Device* _GD);

    //Heightmap/bitmap related functions
    void initWithHeightMap(ID3D11Device* _GD, char* _filename);
    bool readFromBmp(char* _filename);
    bool writeToBmp(std::string _filename);

    //Perlin related functions
    void initWithPerlin(int size, ID3D11Device* _GD);


protected:
    int m_numVerts = 0;
    struct HeightMap
    {
        double height;
    };

    WORD* m_indices;
    myVertex* m_vertices;
    HeightMap* m_heightmap;
    float m_normaliseMultiple = 10.0f;
    int m_width = 1024;
    int m_height = 1024;
};
```

Figure 12. *Code for the VBTerrain class*

The VBTerrain class is the main class used in the simulation. It contains methods for generating the vertices for the terrain mesh and changing their height values. The initWithPerlin and initWithHeightMap methods setup the terrain using the general methods: init; raiseTerrain; initialiseNormals and buildMesh, as well as methods specific to their own setup. m_vertices stores all of the vertices of the terrain and m_heightmap stores height values that change the base vertex positions when raiseTerrain is called; both members are arrays.

Base Grid

When creating a terrain mesh the first process the simulation does is create a base grid of vertices. These vertices are set up in the same arrangement as in Figure 1. The number of vertices created is equal to:

6 * (grid_width – 1) * (grid_height – 1)

A for loop is used to initialise the base values for each vertex as seen to the right. The commented numbers show which vertex in the current quad is being initialised. The vertices in each quad are arranged like this:

1 --- 2

| / |

3 --- 4

…producing two triangles with vertices (1, 2, 3) and (3, 2, 4).

```
//1
m_vertices[vert].Color = Color(1.0f, 1.0f, 1.0f, 1.0f);
m_vertices[vert].Pos = Vector3((float)i, (float)(0), (float)j);
m_vertices[vert].texCoord.x = 1.0f;
m_vertices[vert].texCoord.y = 1.0f;
vert++;

//2
m_vertices[vert].Color = Color(1.0f, 1.0f, 1.0f, 1.0f);
m_vertices[vert].Pos = Vector3((float)i, (float)(0), (float)(j + 1));
m_vertices[vert].texCoord.x = 1.0f;
m_vertices[vert].texCoord.y = 0.0f;
vert++;

//3
m_vertices[vert].Color = Color(1.0f, 1.0f, 1.0f, 1.0f);
m_vertices[vert].Pos = Vector3((float)(i + 1), (float)(0), (float)j);
m_vertices[vert].texCoord.x = 0.0f;
m_vertices[vert].texCoord.y = 1.0f;
vert++;

//3
m_vertices[vert].Color = Color(1.0f, 1.0f, 1.0f, 1.0f);
m_vertices[vert].Pos = Vector3((float)(i + 1), (float)(0), (float)j);
m_vertices[vert].texCoord.x = 0.0f;
m_vertices[vert].texCoord.y = 1.0f;
vert++;

//2
m_vertices[vert].Color = Color(1.0f, 1.0f, 1.0f, 1.0f);
m_vertices[vert].Pos = Vector3((float)i, (float)(0), (float)(j + 1));
m_vertices[vert].texCoord.x = 1.0f;
m_vertices[vert].texCoord.y = 0.0f;
vert++;

//4
m_vertices[vert].Color = Color(1.0f, 1.0f, 1.0f, 1.0f);
m_vertices[vert].Pos = Vector3((float)(i + 1), (float)(0), (float)(j + 1));
m_vertices[vert].texCoord.x = 0.0f;
m_vertices[vert].texCoord.y = 0.0f;
vert++;
}
```

Figure 13. *This code is looped through when setting up the vertices for the base grid. Note that the texture coordinates (texcoord) must be set like this to produce the correct texture orientation as seen in* ***Figure 14.***

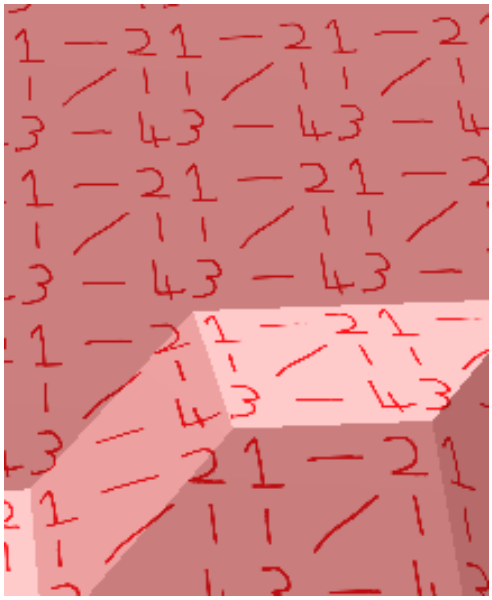


Figure 14. *Close-up runtime image of a debug image used to test the simulation's texture mapping orientation.*

Now that the base grid is set up, the heights for each node can be changed via noise or heightmap values. This is done with a call to the raiseTerrain method, which loops through each vertex and assigns the height value stored in m_heightmap to them.

```
void VBTerrain::raiseTerrain()
{
    int vert = 0;
    int currentHeightMap = 0;

    for (int i = 0; i < m_height - 1; i++)
    {
        for (int j = 0; j < m_width - 1; j++)
        {
            //The comments below represent the vertex number in the current quad
            //1
            m_vertices[vert++].Pos.y = m_heightmap[currentHeightMap].height;

            //2
            m_vertices[vert++].Pos.y = m_heightmap[currentHeightMap + 1].height;

            //3
            m_vertices[vert++].Pos.y = m_heightmap[currentHeightMap + m_width].height;

            //3
            m_vertices[vert++].Pos.y = m_heightmap[currentHeightMap + m_width].height;

            //2
            m_vertices[vert++].Pos.y = m_heightmap[currentHeightMap + 1].height;

            //4
            m_vertices[vert++].Pos.y = m_heightmap[currentHeightMap + m_width + 1].height;
            currentHeightMap++;
        }
        currentHeightMap++;
    }
}
```

Figure 15. *raiseTerrain() code snippet.*

Loading Heightmaps

The readFromBmp method takes in a file path and loads in the pixel data for each pixel of the image. The pixel value of each pixel translates to the height at that position in the mesh. These values are passed to the mesh vertices with a call to raiseTerrain. This code is also used when generating Perlin noise because the program writes the generated values to a bitmap file and then reads from that file to setup the terrain heights.

```
241         //get the dimensions of the terrain
242         m_width = bitmapInfoHeader.biWidth;
243         m_height = bitmapInfoHeader.biHeight;
244
245         //calculate the size of the image data
246         int imageSize = m_width * m_height * 3;
247
248         //if image size is odd add another byte to each line
249         if (m_width % 2 == 1)
250         {
251             imageSize = ((m_width * 3) + 1) * m_height;
252         }
253
254         //allocate memory for the bitmap image data
255         bitmapImage = new unsigned char[imageSize];
256         if (!bitmapImage)
257         {
258             return false;
259         }
260
261         //move to the beginning of the bitmap data
262         fseek(filePtr, bitmapFileHeader.bfOffBits, SEEK_SET);
263
264         //read in the image data
265         count = fread(bitmapImage, 1, imageSize, filePtr);
266         if (count != imageSize)
```

Figure 16. *Code used for reading in pixel data from a bitmap image. The data is read at line 265.*

```
287
288         //initialise the start position for the image data
289         int position = 0;
290         //...and the height value to be read in
291         unsigned char height;
292
293         int index;
294
295         //read the image data into the heightmap
296         for (int i = 0; i < m_height; i++)
297         {
298             for (int j = 0; j < m_width; j++)
299             {
300                 height = bitmapImage[position];
301
302                 index = (m_width * i) + j;
303
304                 m_heightmap[index].height = (float)height;
305
306                 position += 3;
307             }
308             //need to increment again to compensate for the extra byte for odd sizes
309             if (m_width % 2 == 1)
310             {
311                 position++;
312             }
313         }
314
315         std::cout << "Heightmap setup complete\n\n";
316
317         //release the bitmap data
318         delete[] bitmapImage;
319         bitmapImage = 0;
320
321         return true;
322     }
323
```

Figure 17. *Code for adding the pixel data into the heightmap array.*

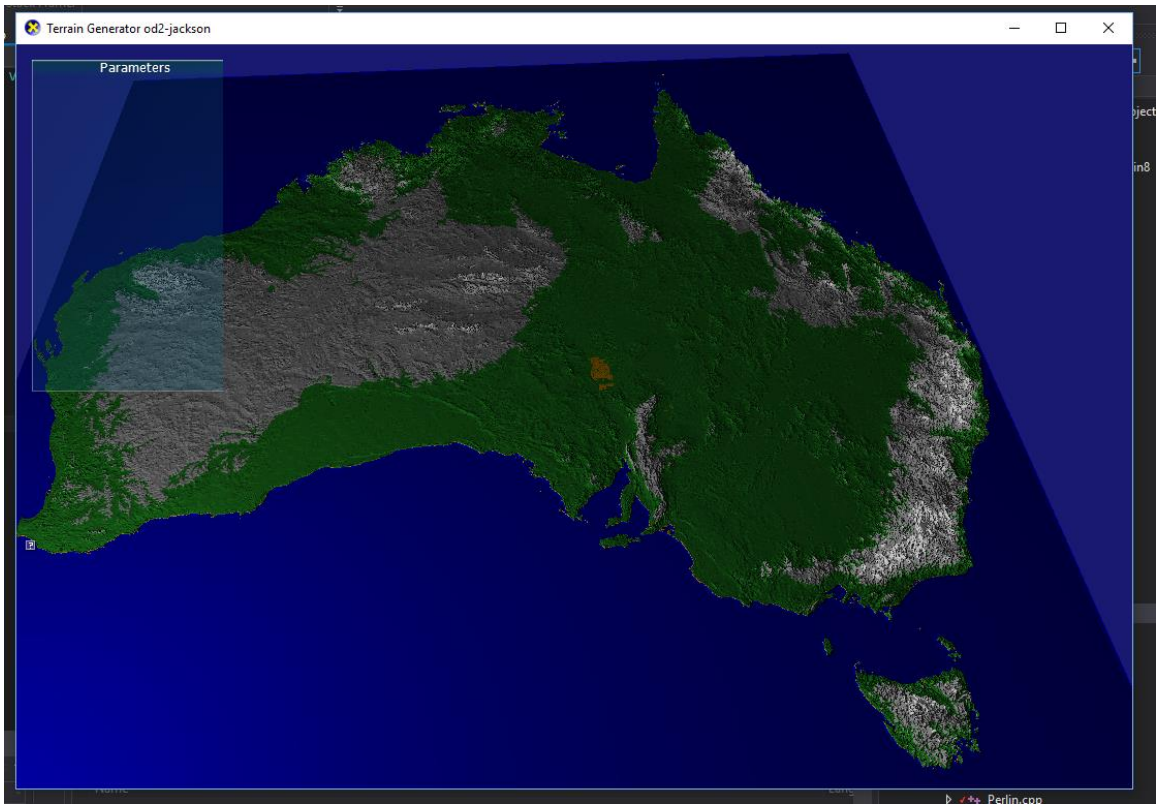


Figure 18. *Runtime of the simulation when reading from the heightmap of Australia seen in Figure 3.*

Generating Perlin Noise

One of the main goals of the implementation was to get it to generate a set of Perlin noise values and use them to create procedurally generated terrain. Below is the algorithm used to generate the noise values. It is almost identical to Ken Perlin's implementation of improved noise [12] except for the normalisation of the final result at the end. The permutations used in this implementation also differ to Perlin's pre-set values since they are instead pseudo-randomly generated when the terrain is initialised to be used with noise. In a video game this method can be seeded to procedurally generate the landscape or to produce specific desired results. For example, Minecraft players can share world seeds that generate interesting geological features so that others can also see them [13]. This implementation also uses the 3D algorithm for generating noise (the z value is kept constant) meaning it can potentially produce changing environments (see Perlin Noise in the Research section).

```
26  //This algorithm uses Ken Perlin's own implementation:
27  //http://mrl.nyu.edu/~perlin/noise/
28  double Perlin::generateNoise(double x, double y, double z)
29  {
30      int X = (int)floor(x) & 255,                  // FIND UNIT CUBE THAT
31          Y = (int)floor(y) & 255,                  // CONTAINS POINT.
32          Z = (int)floor(z) & 255;
33      double oldZ = z;
34      x -= floor(x);                                // FIND RELATIVE X,Y,Z
35      y -= floor(y);                                // OF POINT IN CUBE.
36      z -= floor(z);
37      double u = fade(x),                                // COMPUTE FADE CURVES
38          v = fade(y),                                // FOR EACH OF X,Y,Z.
39          w = fade(z);
40
41      int A = p[X] + Y,
42          AA = p[A] + Z,
43          AB = p[A + 1] + Z,      // HASH COORDINATES OF
44          B = p[X + 1] + Y,
45          BA = p[B] + Z,
46          BB = p[B + 1] + Z;      // THE 8 CUBE CORNERS,
47
48      double x1 = lerp(grad(p[AA], x, y, z), grad(p[BA], x - 1, y, z), u);
49      double x2 = lerp(grad(p[AB], x, y - 1, z), grad(p[BB], x - 1, y - 1, z), u);
50      double r1 = lerp(x1, x2, v);
51
52      x1 = lerp(grad(p[AA + 1], x, y, z - 1), grad(p[BA + 1], x - 1, y, z - 1), u);
53      x2 = lerp(grad(p[AB + 1], x, y - 1, z - 1), grad(p[BB + 1], x - 1, y - 1, z - 1), u);
54      double r2 = lerp(x1, x2, v);
55
56      //result will range between -1 and 1
57      double result = lerp(r1, r2, w);
58
59      //normalise result to be between 0 and 1
60      return (result + 1.0f) / 2;
61  }
62
63  double Perlin::fade(double t)
64  {
65      return t * t * t * (t * (t * 6 - 15) + 10); //Ken Perlin's improved function
66  }
67
```

Figure 19. *Implementation of Perlin noise. Note: the fade function used is Perlin's own improved formula of* $6t^5 - 15t^4 + 10t^3$

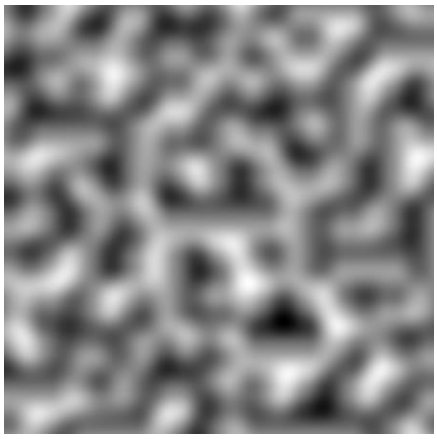


Figure 20. *Example Perlin noise produced by this simulation*

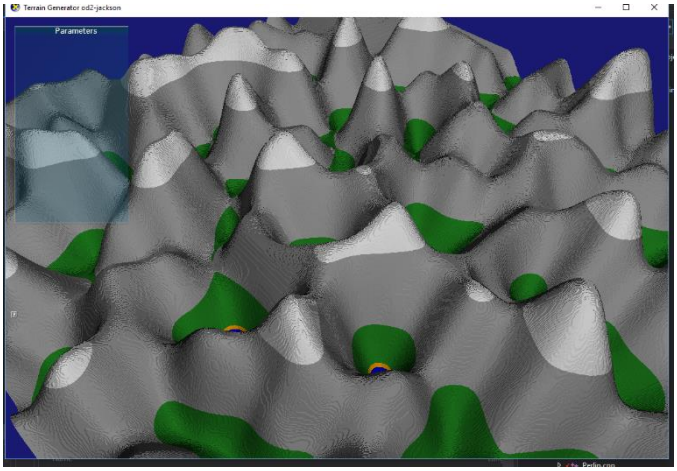


Figure 21. *Runtime screenshot of terrain generated from Perlin noise*

Combining Perlin Noise

Using one run of the Perlin noise produces extremely varying terrain as seen in Figure 21. To produce smoother, more realistic looking terrain the noise images can be combined to create a fractal image. This is done using octaves; each octave affects the overall image less and less, refining the detail that the image displays. This is done by changing the amplitude and frequency of each octave value.

```
74  double Perlin::FBM(double x, double y, double z, int octaves, double persistence)
75  {
76      double total = 0;
77      double frequency = 1;
78      double amplitude = 1;
79      double maxValue = 0;    //Used to normalise the result to between 0.0 and 1.0
80      //generate perlin noise multiple times based on number of octaves
81      for (int i = 0; i < octaves; i++)
82      {
83          total += generateNoise(x * frequency, y * frequency, z * frequency) * amplitude;
84          maxValue += amplitude;
85
86          amplitude *= persistence;
87          frequency *= 2;
88
89      }
90      return total / maxValue;
91  }
```

Figure 22. *Implementation to produce octaves of Perlin noise. Although the method is called FBM (for Fractal Brownian Motion), this algorithm may be more closely related to pink noise or 1/f noise. This is where the frequency of each octave is inversely proportional to its amplitude.*

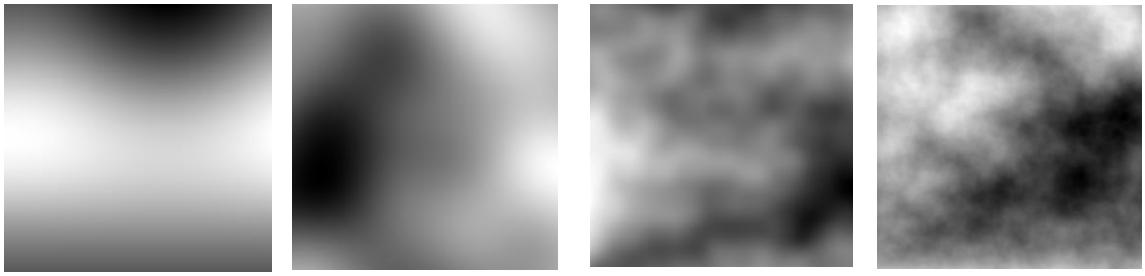


Figure 23. *A set of images produced from using an increasing number of octaves. The number of octaves used in each image are (left to right): 1, 2, 4 and 8.*

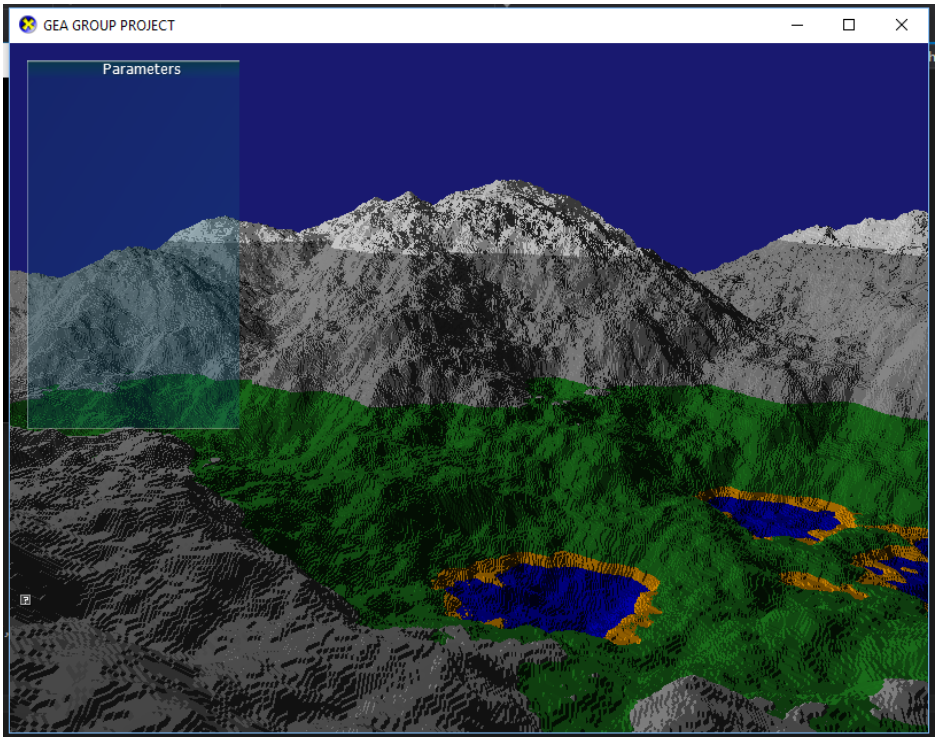
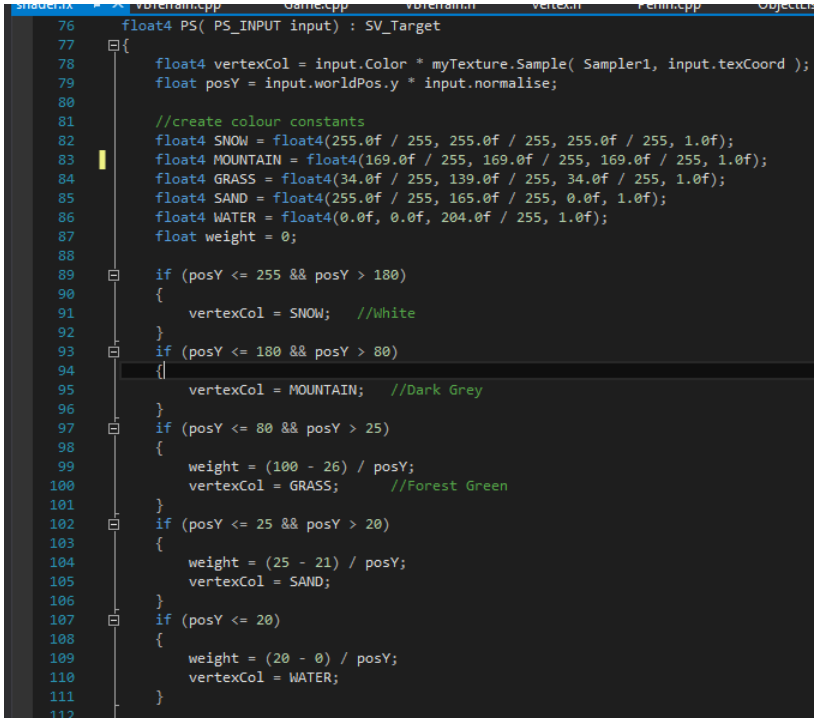


Figure 24. *Runtime screenshot of a terrain produced with 8 octaves of combined Perlin noise.*

Shaders

The default colour for each vertex is set to be white when the grid is initialised. The base pixel shader was edited to change the colour of the vertex based on its height in the world space. Although the implementation is basic, by adding variables such as humidity and temperature a climate like system can be created. The system can then attribute colours or textures to biomes, similar to what this system does with the colour's variable names. For example a position with a low humidity and a high temperature could be initialised with a desert texture. This is what Dwarf Fortress does when generating its world maps [14].



```
76  float4 PS( PS_INPUT input) : SV_Target
77  {
78      float4 vertexCol = input.Color * myTexture.Sample( Sampler1, input.texCoord );
79      float posY = input.worldPos.y * input.normalise;
80
81      //create colour constants
82      float4 SNOW = float4(255.0f / 255, 255.0f / 255, 255.0f / 255, 1.0f);
83      float4 MOUNTAIN = float4(169.0f / 255, 169.0f / 255, 169.0f / 255, 1.0f);
84      float4 GRASS = float4(34.0f / 255, 139.0f / 255, 34.0f / 255, 1.0f);
85      float4 SAND = float4(255.0f / 255, 165.0f / 255, 0.0f, 1.0f);
86      float4 WATER = float4(0.0f, 0.0f, 204.0f / 255, 1.0f);
87      float weight = 0;
88
89      if (posY <= 255 && posY > 180)
90      {
91          vertexCol = SNOW;   //White
92      }
93      if (posY <= 180 && posY > 80)
94      {
95          vertexCol = MOUNTAIN;   //Dark Grey
96      }
97      if (posY <= 80 && posY > 25)
98      {
99          weight = (100 - 26) / posY;
100         vertexCol = GRASS;      //Forest Green
101     }
102     if (posY <= 25 && posY > 20)
103     {
104         weight = (25 - 21) / posY;
105         vertexCol = SAND;
106     }
107     if (posY <= 20)
108     {
109         weight = (20 - 0) / posY;
110         vertexCol = WATER;
111     }
112
```

Figure 25. *Pixel shader added code. The shader performs basic height checks and then assigns a colour based on pre-set ranges*

Project Evaluation

The system implemented fulfils the goals set out by the project proposal as well as having partial success with some of the stretch goals. The implementation of a first-person camera in early development improved the testing of the simulation by making it much easier to notice bugs in the generation from a close-up view. There was some difficulty implementing the Perlin noise algorithm – the implementation was originally going to use 2D noise since that was all that was needed to produce the required heightmaps. The problem actually stemmed from a long unnoticed x/y value mix-up and by the time it was fixed the 3D version of Perlin noise had already been implemented.

The biggest improvement that could be made to the system is the addition of the AntTweakBar. An empty window for the AntTweakBar can be seen in most of the runtime screenshots (on the left of the screen). The bar would allow the user to be able to edit different values during runtime and apply changes to the terrain object based what they edit. Some parameters that were thought of when adding the bar included: number of octaves used for Perlin generation, the starting frequency and persistence of the noise in an octave and the scale of the terrain object. This would allow the user to generate terrain multiple times during runtime instead of needing to reload the application every time they wanted to change something, which is how the current implementation works. Other improvements would include the aforementioned shader changes and runtime optimisations such as quadtrees [15] and LOD-ing methods such as those mentioned in the Rendering Terrain section.

References:

[1]: de Boer, W., (2000), *Fast Terrain Rendering Using Geometrical MipMapping*

[2]: White, M. (2008), Real-Time Optimally Adapting Meshes: Terrain Visualization in Games, *International Journal of Computer Games Technology*, vol. 2008, Article ID 753584, 7 pages, doi:10.1155/2008/753584

[3]: Perlin, K. n.d, *Noise and Turbulence* [Online]
Available from: http://mrl.nyu.edu/~perlin/doc/oscar.html#noise
[Accessed 5 April 2017]

[4]: Perlin, K., (2002). Improving Noise. *SIGGRAPH.*
Available from: http://mrl.nyu.edu/~perlin/paper445.pdf
[Accessed 5 April 2017].

[5]: Flafla2, (2014). *adrian's soapbox - Understanding Perlin Noise.* [Online]
Available from: http://flafla2.github.io/2014/08/09/perlinnoise.html
[Accessed 5 April 2017]

[6]: TylerGlaiel, (2014). *Perlin Noise (animated)*. [Online]
Available from: https://www.shadertoy.com/view/4sXSRN
[Accessed 5 April 2017]

[7]: Persson, M. (2011). *Terrain generation, Part 1* [blog]. 9 March
Available from: https://notch.tumblr.com/post/3746989361/terrain-generation-part-1
[Accessed 5 April 2017]

[8]: Gustavson, S. (2005). *Simplex noise demystified.* [Online]
Available from: http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf
[Accessed 6 April 2017]

[9]: Perlin, K., (2002). *Standard for perlin noise.* United States of America, Patent No. 6867776.
Available from: https://www.google.com/patents/US6867776
[Accessed 6 April 2017].

[10]: Gustavson, S. (2012). Java implementation of Simplex Noise [Online]
Available from: http://weber.itn.liu.se/~stegu/simplexnoise/SimplexNoise.java
[Accessed 6 April 2017]

[11]: Cordonnier, G., (2016). Large Scale Terrain Generation from Tectonic Uplift and Fluvial Erosion. *Computer Graphics Forum*, 35(2), pp. 165-175.
Available from: http://onlinelibrary.wiley.com/doi/10.1111/cgf.12820/full
[Accessed 6 April 2017]

[12]: Perlin, K. (2002). *Improved Noise reference implementation.* [Online]
Available from: http://mrl.nyu.edu/~perlin/noise/
[Accessed 6 April 2017]

[13]: PCGamesN. (2017). *The best Minecraft seeds*. [Online]
Available from: https://www.pcgamesn.com/minecraft/30-best-minecraft-seeds
[Accessed 6 April 2017]

[14]: Dwarf Fortress Wiki, Last modified 2016. *Dwarf Fortress Wiki: Biome*. [Online] Available from: http://dwarffortresswiki.org/index.php/DF2014:Biome [Accessed 14 February 2017].

[15]: Ulrich, T. (2000). *Continuous LOD Terrain Meshing Using Adaptive Quadtrees*. [Online] Available from: http://www.gamasutra.com/view/feature/131841/continuous_lod_terrain_meshing_.php [Accessed 6 April 2017]