Procedural Methods: Multifractal Terrain

Application Report

Cameron McPherson - 1401373

General Features:

This report will present a 3D graphics application allowing exploration of a procedurally generated mountain terrain. The program is implemented in DirectX 11 with a base framework written and provided by Dr Paul Robertson of Abertay University. The terrain generation is utilising a ridged multifractal algorithm which makes use of Ken Perlin's Improved Noise [1].

The user may explore the environment using the controllable camera, or edit several key application settings using the overlaid GUI.

User Interaction:

Camera:

The user may control the world camera using their keyboard. The WASD keys move forward, left, down and back. E moves the camera up, Q moves it down and the arrow keys control the camera's rotation.

Settings:

The overlaid GUI is built using the ImGui library [2] and provides the user with key information and settings to alter the application visually and functionally. Including:

- Current frame rate and information regarding the application's frustum culling.
- Toggling manual/automatic tessellation optimisation.
- Manually altering the min/max range of the dynamic tessellation.
- Toggling wireframe mode (or use spacebar).
- Toggling the display of the world light's depth map.
- Input parameters for the multifractal terrain algorithm.
- Toggling the atmospheric perspective post-processing effect

Implementation:

Structure:

The structure of the application is based around the App1 class. This class inherits from the BaseApplication class in the DXFramework and is responsible for setting up the program window, contains all shader classes, and declares scene objects like cameras, lights and geometry.

[1] – Perlin, K. (n.d.). *Perlin Improved Noise*. [online] Mrl.nyu.edu. Available at: http://mrl.nyu.edu/~perlin/noise/

[2] - Cornut, O. (2017). *ImGui*. [online] GitHub. Available at: https://github.com/ocornut/imgui

This class also defines the main application loop, sending geometry for rendering by the GPU each frame. The main structure of the program loop is as follows:

```
1     Handle User Input
2     Call new DX11 frame
3     Render
      3.1   Render depth map to texture
      3.2   Render Scene
                  3.2.1    Render skybox without Z buffer
                  3.2.2    Update camera frustum
                  3.2.3    Render terrain in view with texture
                           and lighting
      3.3   Check optimisation mode
      3.4   Draw graphics UI
4     End frame
```

The application is designed with a very object oriented approach. Each set of shaders have their own class defining their initialisation, parameter setting and the sending of data to the GPU. Each portion of the application is contained within its own class to maintain a structured, extendable and readable form.

Procedural Generation:

The procedural content in the application is a Perlin Noise based ridged multifractal algorithm that offsets the height of a tessellated grid to produce realistic procedural terrain.

While simply applying a noise function to the terrain would, strictly speaking, be procedurally generated, the result is in general repetitive, uninteresting and lacks the apparent randomness of the natural world. We can benefit by using noise as a *basis function* for a more complex fractal algorithm. A fractal is a mathematical concept which creates complexity by applying the same repeating function at various scales. Fractals are "self-similar" by nature, which means they are great for modelling various natural occurrences such as mountains, water and trees. By layering fractals on top of Perlin's Improved Noise, we can begin to see a more diverse, visually pleasing landscape emerge. This process of adding layers of complexity at varying scales is known as fractional Brownian motion (fBm).

However, fBm in its simplest form (also described as "plasma") can still fall a bit short in modelling a real mountainous terrain. The typical fBm algorithm is in nature homogeneous, meaning that when applied to a mountain terrain, the "roughness" will remain constant everywhere. Naturally occurring fractals are rarely so consistent, and mountains often exhibit smooth valleys at low areas and rocky outcrops at their peaks. This is the reason behind using a multifractal algorithm for the terrain heights.

Multifractals are heterogeneous, producing results of varying roughness based on location. At a mathematical level, multifractals essentially alter the fractal dimension at each stage. This produces far more realistic and interesting terrain.

Figure 1 - Terrain Generation using Ridged Multifractal Algorithm

The specific multifractal function[3] used in this application is a "ridged" multifractal written by Kenton Musgrave. It is similar to Perlin's turbulence function, in that it ignores negative results of the noise function by taking the absolute value. It flips the final signal upside-down to cause the algorithm's creases to stand up as ridges. This creates complex looking alpine terrain, with sharp crests and peaks.

This solution provided a realistic and visually interesting terrain model, but does present a couple of routes to improve. Currently, the terrain is contained within a specific range of height values, meaning that there are no stand out features. The algorithm outputs values between 0 and 2, which are then scaled by a set value. By grouping zones of the terrain and scaling the multifractal outputs based on a zone height, the terrain could be allowed to differ further over large scales.

The complexity of the application's landscape is at a believable level, however there are alternative methods in multifractal algorithms that can improve the apparent

[3] Ebert, D. (2003). *Texturing and Modelling: A Procedural Approach*. 1st ed. San Diego: Elsevier Science, pp.504-505.

intricacy of its appearance. Currently, the algorithm uses an *additive cascade,* meaning its frequency updates at the same variance every iteration. There has been research into using *multiplicative cascade* for terrain generation, which multiplies by the next highest frequency each step, as opposed to adding. Therefore the frequency would experience different scaling at each stage, introducing more complexity.

Normal Calculation:

In order for the terrain to be correctly lit, vertex normals must be determined after the procedural generation is complete. This means that pre-loaded solutions such as normal maps are not suitable. The normal for each vertex is therefore calculated mathematically once the terrain generation is complete.

The normals are calculated using the central differences method, which samples the multifractal function left, right, up and down from the vertex at a given step, builds four vectors to these points from the vertex, and generates surface normals for the faces they create. The average normal of these faces is taken as the vertex normal.

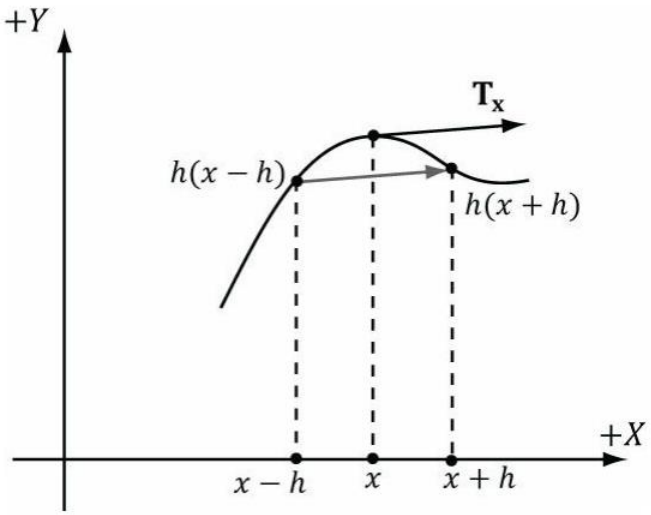

Figure 2 - Central Differences Method[4]

This method is accurate enough to produce realistic lighting effects at run time, however it is expensive, as the shaders must sample the multifractal function another four times per vertex to calculate the normals. This can lead to dropped framerates on machines with older GPUs. If the terrain was to be set on start-up and not modified, these normals could be calculated once as the application loads, saved to a texture and simply sampled by the pixel shader per-frame. This would benefit performance greatly at the expense of being unable to edit the parameters of the multifractal algorithm in real-time.

[4] Luna, F. (2012). *Introduction to 3D game programming with DirectX 11*. 1st ed. Dulles, VA: Mercury Learning and Information, p.528.

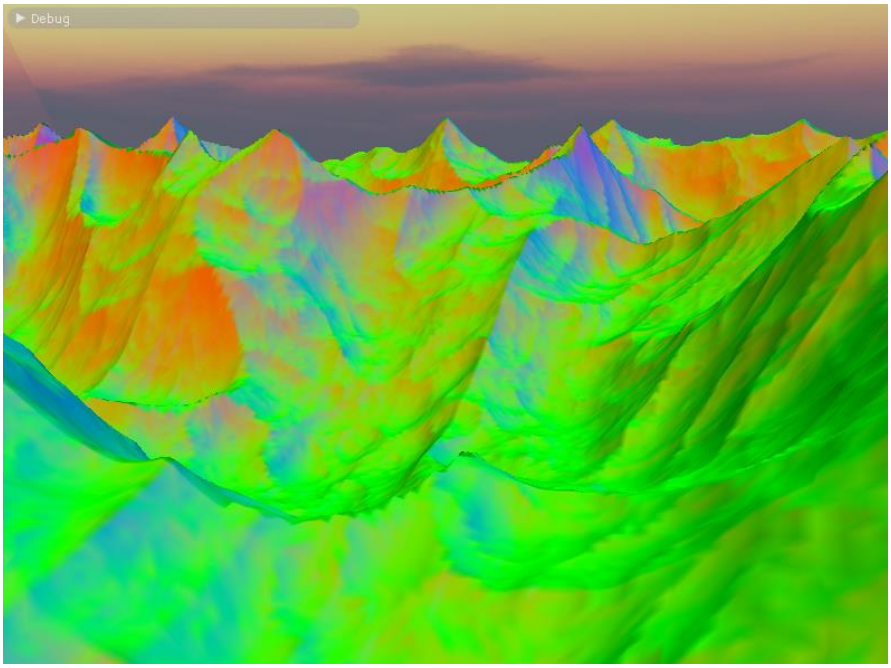

Figure 4- Visual Representation of Normals

Texturing:

To texture the terrain realistically, the application makes use of the calculated normals to decide between snow and rock based on the slope of the geometry. Steep slopes are textured with rock, while flatter surfaces use the snow texture. This is simple and cheap, and reflects the real world quite well. The method for slope-based texturing is based on an example hosted by RasterTek[5].

This method looks great with the tessellation at high detail, however suffers when the detail is low as there are less vertices populating the geometry. Again, this problem becomes easier to solve once the terrain is decided upon and not variable during run-time. Each terrain patch could store a high detailed texture map once the initial procedural generation is complete. Then regardless of the level of detail of the geometry, every part of the terrain will have high resolution texturing.

Lighting:

The scene is lit by a single point light off one end of the finite terrain area. The light emits in all directions with no attenuation to mimic the light of the sun. When just interacting with the generated normals, the lighting looks good. However, it was nice to try and implement shadows as well, having the tall peaks casting darkness onto valleys. Therefore, it was necessary to implement a shadow map pass into the render loop. This takes the terrain geometry and renders only the depth information from the light's point of view into a high-resolution texture.

[5] Rastertek.com. (n.d.). *Tutorial 14: Slope Based Texturing*. [online] Available at: http://rastertek.com/tertut14.html

Then, when it comes to lighting the terrain in the render pass, the shaders sample the depth map to test whether the current pixel is occluded by other geometry, if so the pixel is not lit.

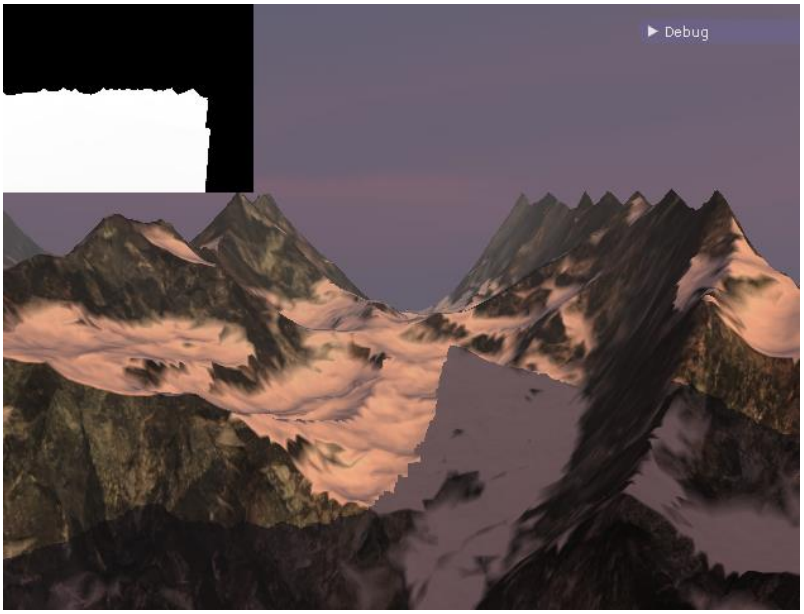

Figure 5 - Depth Map Shown with Shadows

Post Processing:

When expanding the scale of the terrain and generating the landscape over increasing distances, it became clear that post processing was necessary to mimic the atmospheric effects of distance in the real world. At distance, colours of objects become cooler and desaturated because of fog and atmospheric effects. This effect is replicated in the application by blending output colours based on distance from the camera.

```
if(camPadding == 1.0f)
{
	float distToEye = distance(input.position3D, camPos);

	float fogAmount = saturate((distToEye - fogNear) / fogFar);
	outcolor = lerp(outcolor, fogColour, fogAmount);
}
```

The result makes it far easier for the eye to distinguish distance, and adds a great level of depth to the application. This implementation is based on Frank Luna's example [6.0].

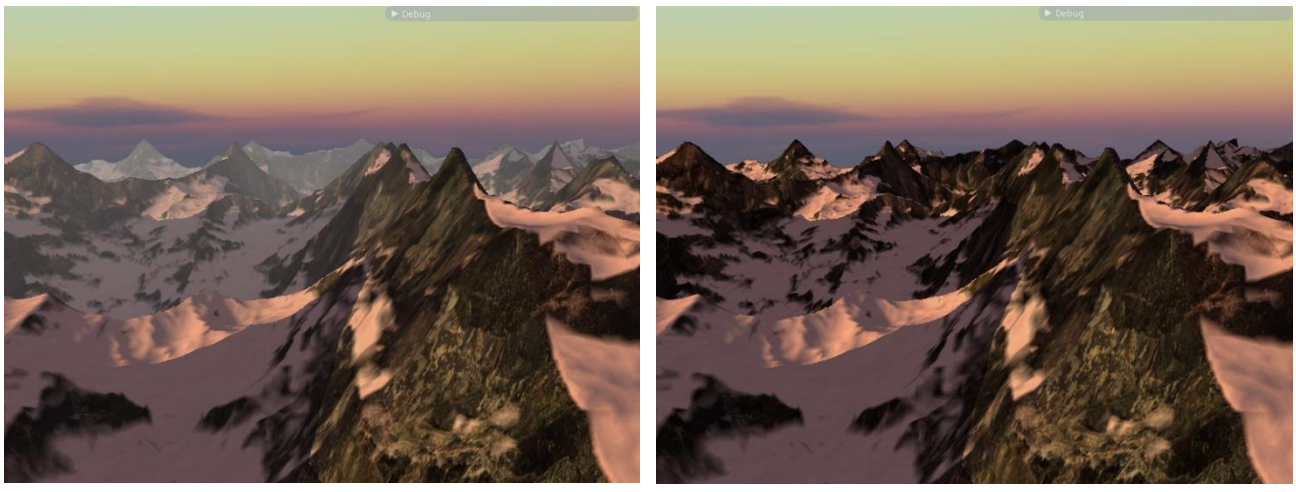

Figure 6 - Atmospheric Effects On (Left) and Off (Right)

Optimisation

Dynamic Tessellation:

When exploring an environment, the user is unlikely to care as much about distant objects as those near to them. It is therefore inefficient to render everything at the same quality regardless of distance to the camera. In the application, the level of tessellation of each terrain patch is scaled based on its distance from the camera. This renders close-by terrain at high detail, and saves stress on the GPU by not adding as many vertices at distance. The algorithm (based on Frank Luna's example[6.1]) is located in the hull shader, taking the camera's position and vertex position and determining a tessellation level based on the distance between them. The range of distance between which the tessellation values are varied can be edited in the GUI. This effect is best observed by enabling wireframe mode.

This can produce some artefacts in the terrain texturing when moving the camera, as the vertices used for determining which texture is used are moved, added and destroyed. This is less noticeable at high detail levels.

Frustum Culling:

Clearly, anything that is not currently visible by the camera but still rendered by the GPU is wasted effort. So, it was necessary to improve the application's performance by implementing frustum culling. The terrain patches are offset once to their correct heights at runtime, then have bounding boxes defined and stored. Then, by assigning the scene's camera a view frustum that is updated every frame, each patch's bounding box can be tested to determine if it is in view. If not, it is not sent to the GPU for rendering. The class that was created to calculate and store the camera's frustum is based on an example from RasterTek[7].

Target Frame Rate:

Not all computers are born equal, and different hardware performs better or worse in graphics applications. In order to give every user an acceptable level of performance and detail, a target frame rate (30 fps) was defined for the application.

By activating "automatic" optimisation mode via the GUI, the application will automatically scale the tessellation range values to attempt to achieve 30fps. If the user's machine is achieving greater frame rates, it will increase detail at range, and vice versa. Choose "manual" mode to alter detail levels without interference.

Testing

These methods help to increase the frame rate in the application across differing hardware, and aim to make it accessible to machines running older graphics cards and processors. Below is the test data of frame rates across differing machines and settings.

[6.0] and [6.1] Luna, F. (2012). *Introduction to 3D game programming with DirectX 11*. 1st ed. Dulles, VA: Mercury Learning and Information, p.327 and p.526 respectively.
[7] Rastertek.com. (n.d.). *Tutorial 16: Frustum Culling*. [online] Available at: http://rastertek.com/dx11tut16.html

Machine 1 – GPU: nVidia GT 640 2GB DDR3 **CPU:** Intel Xeon E3-1245

Machine 2 – GPU: nVidia GTX 960 2GB GDDR5 **CPU:** Intel i5 4460

Machine 3 – GPU: nVidia GTX 1070 8GB GDDR5 **CPU:** Intel i7 7820HK

Frustum Culling:

Figure 7 - No Frustum Culling at High Detail (~6.5 fps)

Figure 8 - Frustum Culling at Same Detail (~11.1 fps)

As shown in Fig. 7 and 8, enabling frustum culling on Machine 1 while at high detail levels almost doubles the frame rate across the application. Machine 2 saw roughly the same benefit while Machine 3 saw ~22 FPS with no culling at full detail, and 40 FPS with culling enabled. It is clear that this strategy greatly improves performance across all hardware.

Target Frame Rate:

The below table outlines the test results on each machine when automatic optimisation mode was turned on while at the default detail values. The "near" tessellation value represents the distance from the camera to which the maximum detail will be applied, while the "far" value represents the distance beyond which the minimum is applied. Between these two distances the detail level is blended.

	30 FPS reached?	**FPS at default (near 150, far 300)**	**Near detail range @ target FPS**	**Far detail range @ target FPS**
Machine 1	Yes	12 FPS	30	180
Machine 2	Yes	45 FPS	200	350
Machine 3	Yes	60 FPS	410	500

We can see that the automatic optimisation mode helps hardware at all levels to reach their maximum detail level while maintaining an acceptable frame rate.

Critical Analysis/Next Steps

This application demonstrates the power of procedural generation when modelling vast and complex environments. The alternative would be to manually model each rock, ridge and valley by hand in 3D and load it in, which is both entirely inefficient as well as extremely expensive on the hardware. Generating terrain procedurally allows us to create complex, interesting worlds without passing huge model data to the GPU, or worry about modelling the terrain ourselves. The process takes work away from the developer and gives it to the hardware, speeding up development and reducing stress on memory and processing.

However, taking the procedural approach presents some problems of its own, with features such as lighting and texturing complicated by the fact that the shape of your terrain is unknown before run-time. There are steps to take to get around this as outlined in the relevant sections above, but this application reflects the desire to exhibit the procedural algorithm primarily. Allowing the user to experiment with the parameters of the multifractal was prioritised over hyper-realistic lighting and texturing. When implementing such a terrain in a specific product such as a game however, these parameters can be set to a desirable value and left there, allowing normals and texturing to be pre-calculated.

While the implementation of the procedural algorithm behaves as intended, and the landscape generated is interesting and complex, the performance of the application is still slightly prohibiting in terms of hardware required. In order for the landscape to be extended beyond finite boundaries and the application to run on most modern machines quickly, more needs to be done in optimisation. For example, taking the concept of frustum culling to the next step and implementing occlusion culling would greatly increase the performance of the program. Not only would the parts of terrain out with the camera's frustum not be rendered, but geometry within the frustum that is occluded by closer terrain would also be left out. This would further reduce the stress on the GPU and allow the terrain to be extended to an infinite, scrolling landscape.

Overall, this project serves as a great example of how procedural generation can facilitate the creation of extremely complex geometry, and specifically the power of fractals in modelling self-similar natural phenomena in believable and efficient ways. As hardware and research continue to develop we will only see better and more effective techniques in procedural graphics and, in turn, ever more immersive and impressive games, movies and software.