Applying Tessellation to Clipmap Terrain Rendering

COSC460 Research Project Report

Corey Barnard

55066487

October 17, 2014

Department of Computer Science & Software Engineering

University of Canterbury

Christchurch

New Zealand

Supervisor: Dr R. Mukundan

Abstract

Recent graphics hardware has introduced a number useful techniques that can be applied to terrain rendering. In this report, we describe some of the issues with terrain rendering, such as performance. We describe our framework for implementing terrain rendering algorithms which help to reduce implementation complexity. We explore the possibility of applying recent hardware and API features to the Geometry Clipmaps algorithm proposed in 2004. Our implementation is described, which makes use of the nested grid structure introduced in Geometry Clipmaps, as well as recent hardware tessellation features in modern GPUs. Some experiments are carried out to evaluate the new approach. Compared to the original algorithm from 2004, our implementation improves performances by 71%.

Acknowledgements

I would like to thank my supervisor, Dr R. Mukundan, for his guidance throughout the duration of this project.

Contents

1. INTRODUCTION

Terrain rendering is an area of computer graphics which covers methods of visualizing imaginary and real-world surfaces in real-time[1]. It has many applications including Geographical Information Systems (GIS) [1], flight simulation, Synthetic Vision Systems (SVS) [2] and computer games. These applications demand low latency, while requiring accurate and realistic images. Although current approaches achieve high realism, due to the large and highly-detailed nature of terrain scenes, modern approaches limit the size of the terrain. This is usually hidden from the viewer by applying fog effects to distant areas, to hide this.

1.1. Typical process

The most common family of terrain rendering algorithms is known as height-mapped-terrain, which uses an image (heightmap) to store elevation data. Heightmaps are typically created based on satellite data, which are known as digital elevation maps (DEM), or may be created by an artist using a modelling tool. Each pixel represents the elevation of a point on the surface, which is used by the terrain rendering algorithm to produce a 3D representation of the terrain. Pixels are typically 16-bit and vary from. The most common technique for achieving this is to treat the terrain as a 3D grid, where each point's elevation is offset according to the heightmap.

Some of the techniques proposed for terrain rendering are not limited to terrain visualization. The techniques have been applied to other areas [3], such as ocean rendering. The techniques are most beneficial to rendering models that cannot easily take advantage of traditional optimization techniques like, culling.

1.2. Motivation

Terrain rendering research, like many areas of computer graphics, is motivated by the desire to produce high quality 3D scenes as fast as possible. Consumer demand is one of the driving factors behind this due to the increasing expectations and demands. As hardware becomes faster and API features are introduced, new techniques become possible. Terrain rendering research investigates how these factors can be applied to produce larger, more photo-realistic terrains in real-time.

1.3. Aims and Objectives

The aim of this research was to investigate possible improvements to existing terrain rendering algorithms, focussing on utilizing recent hardware and API features. We were particularly interested in finding out whether older algorithms could benefit from any of the recently

[1] Real-time means allowing a user to freely navigate the scene while maintaining at least 30 frames per second.

introduced API features, as well as recent hardware features. We wanted to see if we could improve performance and reduce implementation complexity, while maintaining high image quality.

1.4. Report Structure

This report describes our work on improving existing terrain rendering algorithms. The report is structured as follows: Section two gives a general background into computer graphics, focussing on ideas and concepts we have used. Section three explores previous terrain rendering methods and categorizes them according to characteristic features. Section four describes our implementation details as well as design decisions. Section five contains results of experiments that were carried out to evaluate our approach. Section six provides a discussion of these results and how this relates to prior work. Section seven summarizes our research, relating it back to our initial goals. We also discuss possible directions for future work.

2. BACKGROUND

2.1. Rasterization

Rasterization is a technique for rendering a three-dimensional scene. This method treats a 3D model as a set of polygons, usually triangles, and performs projective transformation to a plane. This is the standard technique used by current graphics hardware. The Rasterization algorithm's execution time is proportional to the number of triangles submitted, although the process is highly accelerated by parallelization. It is common in computer graphics to minimize the number of polygons used to draw a model by avoiding those which are not truly necessary. This includes polygons outside the camera's view and also any polygons which are behind other objects and cannot be seen. These are effective techniques for increasing the algorithm's performance. A common metric for measuring performance of a computer graphics algorithm is frames per second (FPS). This is a count of the number of images rendered within one second. 30 FPS is typically regarded as the lower-bound for the human-eye to perceive the frames as a continuous motion.

Vertex buffers are data structures used in graphics application development. A vertex buffer is filled with some data, specified by the developer. This is typically data corresponding to a 3D model, which usually includes vertex data, normal data and texture coordinates. Data is generated on the CPU, either from a file or a procedural algorithm, which is then buffered into the GPU memory. This allows for faster access when rendering a scene. One limitation with vertex buffers is that the data cannot be modified, once it has been transferred, without the use of costly buffer function calls. For this reason, vertex buffers tend to be generated once and remain unchanged until no longer needed.

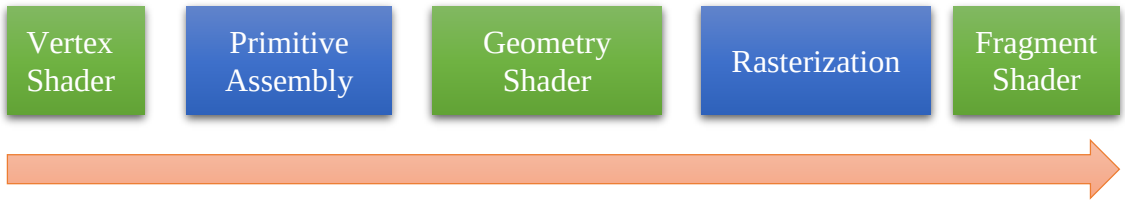

Figure 1. The graphics pipeline prior to OpenGL 4. Programmable stages are shown in green.

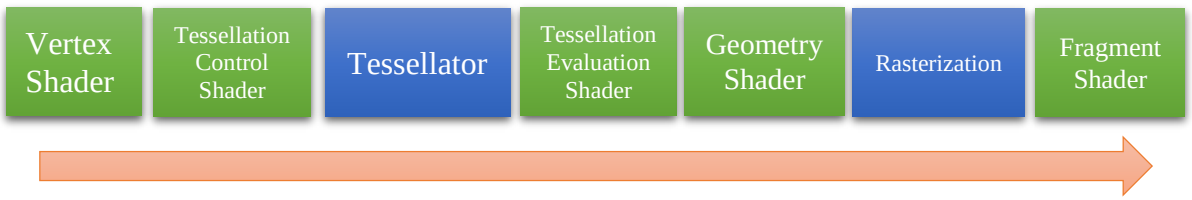

Figure 2. The recently introduced pipeline in OpenGL 4. Programmable stages are shown in green.

2.1. Level of Detail

A downside to performing rasterization-based rendering using vertex buffers is the fact that all the vertex data is stored statically. This means that, given a set of vertices within a buffer, a model will be rendered the same regardless of how close it is to the viewer. Level of detail (LOD) [3] is a technique which involves reducing the complexity of 3D model according to certain conditions. View-dependent level of detail (LOD) [4] is a technique for altering the detail needed to represent a 3D model, based on the model's location, relative to the camera. This enables some redundant operations to be avoided by rendering models in the distance at lower detail, without having an impact on the overall quality.

2.1. OpenGL

OpenGL [5] is an API designed for providing developers with the necessary tools for writing graphics applications. Early versions used a "fixed-function" pipeline which rendered images using default functions with customizable parameters. Although the fixed-function pipeline is easy to use, the approach lacks the ability to customize the rendering process. Additionally, using the fixed-function tends to be slower due to being more CPU intensive, which becomes a bottle-neck as large amounts of data is transferred to the GPU regularly.

2.1. Programmable Pipeline

The programmable pipeline is an addition, introduced in a subsequent version of OpenGL, which gives developers greater control of the rendering process. Programs, called Shaders, can be written in OpenGL Shading Language (GLSL), which are executed in place of the fixed-function pipeline. This provides developers with greater flexibility where developing graphics applications. The programmable pipeline used in previous versions is shown in Figure 1.

2.1. OpenGL 4 pipeline

OpenGL 4 is a recent API [6] which enables developers to target Direct3D 11[2] hardware. Modern Graphics Processing Units (GPU) provide features previously not available to developers. The most significant addition is the modified OpenGL pipeline, which introduces a tessellation stage, which can be seen in Figure 2. The tessellation stage is used to subdivide a given polygon and apply some transformations. This feature can be effectively applied to a 3D model to achieve dynamic LOD very easily. The tessellation stage is utilized by the developer by writing two shader programs: a tessellation control shader and a tessellation evaluation shader. The tessellation control shader is used to programmatically select an appropriate tessellation

[2] Direct3D is an alternative API to OpenGL, which is commonly used as a standard by hardware vendors.

3. RELATED WORK

3.1. Greedy approach

The greedy approach, also referred to as brute force, is a simple method of rendering terrain. This method involves rendering a grid of uniformly-sized squares and offsetting the height of each according to some data. The grid is represented as a vertex buffer and is created at the start of the application. Rendering the terrain consists of drawing each vertex within the buffer, regardless of its location. This approach, although straightforward to implement and accurately models the terrain data, offers poor performance. This is due to rendering the entire terrain at the highest detail, even where the terrain is not visible, due to the lack of LOD.

3.2. CPU-based algorithms

Early terrain rendering algorithms were mostly CPU-based. This is because the graphics pipeline at the time was mostly fixed function, meaning that the developer could not customize the steps during rendering. CPU-based terrain rendering algorithms can classified into two categories: simplification and refinement [7]. Simplification methods reduce the data set to a simplified version of the data. Refinement methods start with a small dataset and insert detail to improve the resolution. The main focus of these kinds of algorithms was finding ways to intelligently simplify or refine the data to achieve sufficient quality. Progressive meshes [8] is a general-purpose LOD technique that can be applied to terrain rendering. The method works by collapsing edges within a mesh into points and vice-versa. To render terrain, the approach refines areas based on the view frustum and projected screen-size of primitives. This method was later extended to support view-dependent LOD [9]. Real-time Optimally Adapting Meshes (ROAM) [10] achieves LOD using two factors: the camera position and direction; as well as the shape of the terrain. A hierarchical-triangle-binary-tree is used to represent the terrain. Each triangle is either split or merged with its neighbour at regular intervals according to a threshold. This is applied recursively according to the desired quality. Such operations relied heavily on the CPU, making ROAM less popular as GPU hardware became more common. Another popular CPU based algorithm is Geometry Clipmaps [11], which is described in more detail in section 3.4.

3.3. GPU-based algorithms

With the availability of programmable GPU hardware, developers could produce more complex effects, while reducing the load on the CPU. This lead to the introduction of a number of terrain rendering techniques, which took advantage of this. Geomipmapping [12] is an approach analogous to texture mipmapping [13], which stores a set of grid-based tiles, at various resolutions, inside vertex buffers on the GPU. The terrain is rendered as a grid of tiles, where each tile is chosen based on the required LOD at the point. Vertex Shaders are used to offset the height of points within each tile according to the dataset. Batched Dynamic Adaptive Meshes (BDAM) [14] is a similar method to ROAM that also uses binary trees. However, this

approach uses small mesh patches instead of triangles, which can be sent in batches to the GPU to optimize the rendering process. Seamless patches [15] is a method which subdivides the terrain surface into a grid of square patches. Each patch consists of four triangular tiles of different, predetermined resolutions. A thin strip is positioned between each tile, which provides a seamless connection between tiles of different resolutions. Terrain elevation for each patch is provided by the vertex shader. A GPU-based version of Geometry Clipmaps [16] is described in detail in section 3.4.

3.4. Geometry Clipmaps

The Geometry Clipmaps [11] algorithm is a terrain rendering technique which introduced a new approach to LOD which involved representing the terrain as a series of nested grids centred on the camera. The centre grid is rendered fully, while each surrounding grid is rendered, at a larger scale, with the inner part removed. The result is a grid structure where the inner-most sections are denser than the outer regions, shown in Figure 3. As the camera moves around the scene, the grid is shifted to stay centred at the camera and vertex positions are updated to reflect the shifted terrain.

By using nested grids, the method achieves LOD very elegantly as it is not directly based on the distance from the camera. Instead, each successive grid is rendered at twice the scale as the previous, meaning each polygon occupies twice as much space in the world. Rendering all the grids results in each polygon having approximately uniform size in screen-space. This is due to perspective projection which causes objects further from the camera to appear smaller on screen.

The algorithm has some limitations, which affect its performance. The main limitation is that the algorithm is highly CPU-dependent. A vertex buffer was used to represent each Clipmap level, meaning that the vertex data could not be modified. To update the terrain according to camera movement, a new set of data needed to be computed and buffered to the GPU to reflect the update. This buffering introduced a bottleneck in the process, as recomputing vertex data per update is a costly operation. Another limitation in the algorithm is that the actual content of the terrain did not influence the final result. This is problematic in cases where there is more detail than is actually required. An example is an area of terrain without much variation in

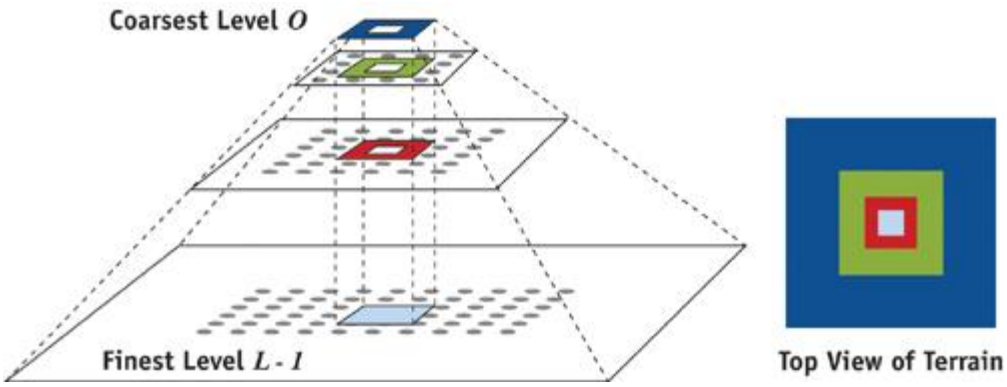

Figure 3.The nested grid structure used in Geometry Clipmaps [16].

height, such as a flat plain. If such an area is rendered with more polygons than necessary, it is a waste of time. Conversely, if an area is rendered with insufficient detail to accurately represent the terrain data, such as a mountain, then image quality can be affected.

An improved GPU-based implementation of the Geometry Clipmaps algorithm was proposed shortly after [16], which addressed the bottleneck problem with the original implementation. This was achieved by utilizing texture sampling features in the GPU. Instead of recomputing the vertex data during each update, the vertex shader stage of the rendering pipeline was used. This involved sampling a heightmap within the vertex shader to determine the height of each vertex. As the camera moved, only the texture coordinates used to sample the heightmap needed to be updated.

A comparative analysis of the algorithm was performed [17], which compared the performance of their implementation with other terrain rendering algorithms [10], [12]. The researchers concluded that their implementation could achieve highest efficiency on more recent hardware, whereas the remaining algorithms outperformed it on older graphics cards.

The original Geometry Clipmaps algorithm and the improved GPU-based version both did not consider the terrain "content". This means that areas of terrain with lots of fine details, such as mountain ranges or ridges, were rendered with the same degree of detail as those which had significantly less detail, like grassy plains. This is wasteful because an area of terrain might be overly-represented, meaning that an equivalent image could be achieved with significantly fewer polygons and would be faster. Real-world terrain data tends to have variations, regardless of the location, meaning this is only a minor concern. However, for terrain data produced by an artist, where flatter regions are more prominent, this could be a significant factor.

One difficulty that arises when using Geometry Clipmaps is the need to find a balance between performance and accurately representing the terrain data. This is done by selecting the density of each nested grid. The density should be chosen such that outer levels, which consist of larger, more spaced out polygons, are able to represent the terrain data to a sufficient degree of accuracy. Furthermore, the density should not be so high that the inner levels over-represent the data or introduce performance problems. A suggested approach is to select the density such that any given polygon, independent of its grid level, should occupy approximately one pixel when projected into screen-space.

3.5. Tessellation for Terrain

Tessellation features in modern graphics hardware allow for efficient terrain rendering. Most algorithms follow a similar concept which involves subdividing a terrain into uniform-sized tiles, which consist of a number of tessellation patches [18]–[21]. Culling is performed by selectively rendering only tiles that are within view, while tessellation provides LOD. The specification of the number of patches per tile varies among researchers. The method described by Kvalsvik Jakobsen treats each tile as a single patch [21], whereas Cantlay's approach uses an 8x8 grid of patches for each tile [18]. By using more patches per tile, a higher degree of detail can be achieved. This is because current hardware enforces a tessellation limit of 64,

meaning a given patch can only be tessellated 64 times horizontally and vertically. Increasing the number of patches in a given tile artificially increases this limit, while still allowing for culling. Alternative methods [19], [20], [22], [23] determine the tile size and number of patches using a tree structure, such as a quad-tree or bin-tree. The method proposed by Kang *et al.* treats each tile as a quad-tree [23] with maximum depth four. A pre-processing stage is performed which determines the degree of curvature at each tile, using screen-space error. This is used at run-time, in addition to the camera position, to achieve LOD by determining the quad-tree depth required for each tile. This means that tiles with deeper quad-trees are rendered with more detail. Kvalsvik Jokobsen proposed a novel technique [21], which utilizes an additional texture called a "tessellation map". This is a texture, similar to the heightmap, which influences the tessellation factor required at each point. This was shown to be effective when indicating areas requiring higher detail, like roads. The drawback to this method is the need for an additional texture sample as well as the need to provide the tessellation map. A similar technique was proposed [24] which automatically generates this texture during pre-processing. This approach generates a texture, referred to as a height acceleration map (HAM), by applying the Sobel filter to the heightmap. The Sobel filter is a technique which effectively approximates the image gradient and emphasizes edges and transitions within an image. The HAM encodes the amount of variation across the terrain surface, which is used to provide LOD by determining tessellation factors.

3.6. Raycasting

Raycasting is an alternative technique to rasterization, for rendering a 3D scene. The concept involves tracing geometric rays from the camera to the scene to find the closest object using ray-surface intersection tests. Raycasting has been applied to terrain rendering [25], [26] in recent years. These algorithms work by pre-processing the terrain into tiles and computing axis-aligned bounding volumes. Each bounding box is created such that the terrain surface, within the tile, is fully enclosed. Every frame, the tiles within the view frustum are rendered. To do this, rays are cast through all the screen-space fragments covering a tile to determine the first intersection point with the heightmap. A photo texture is sampled at the intersection point to obtain the pixel colour. This is achieved by rendering the back faces of a tile's bounding box and processing the resulting fragments using GPU-based ray casting [27] in the fragment shader. Compared to traditional terrain algorithms, ray casting-based methods tend to be less efficient than rasterization-based approaches for smaller data sets and are more complex to implement. When dealing with larger data sets, ray casting becomes more effective.

Hybrid methods, which combine rasterization and ray casting, have been proposed [28], [29], which utilize advantages of each technique. The concept involves selecting the best method to apply to each tile, per frame. A heuristic is used to estimate the time needed to render a tile for each method. The method with the lowest estimated time is used to render the tile. Results suggest that the hybrid method performs at least as well as rasterization and in some cases, better.

4. IMPLEMENTATION

In this section, we describe our implementation, which is a modified version of Geometry Clipmaps [11], combined with tessellation features and other features available in modern GPUs. Our implementation addresses some of the limitations in the original GPU-based implementation and also some of the issues in modern tessellation-based algorithms.

4.1. Development Environment

The algorithm was implemented on a PC running Windows 7 64-bit, with a 3.5GHz AMD Phenom II X4 quad-core CPU, 8GB RAM and an NVIDIA 1GB GTX560 Ti graphics card, using Microsoft Visual Studio 2013. We have used the following libraries: the OpenGL API, FreeGlut toolkit, OpenGL mathematics (GLM) and the OpenGL extension wrangler (GLEW).The algorithm implementation consists of one C++ class and five shader programs utilizing all stages of the OpenGL 4 pipeline.

4.1. Framework

A framework was implemented to reduce development time and complexity when implementing terrain algorithms. Our framework implements much of the essential foundations needed, including texture and shader management, text rendering, graphic user interface functionality and camera control.

To implement a new terrain rendering algorithm, the following steps would need to be completed: A class must be written which inherits from the base terrain class. The purpose of this class is to load required resources and perform rendering functionality. Based on the specific algorithm, several shaders must also be written.

To simplify testing, we implemented a system to automatically iterate through the available algorithms and heightmap images, run experiments for a specified time and export the results.

We attempted to implement an automatic heightmap management system. The idea behind this was to be able to subdivide a large image into smaller images and load them into the system. Then, instead of manually streaming or compressing a large texture, pixel values could be queried from the system, by coordinate, without any knowledge of which sub-texture it belonged. Internally, the system would handle all the overhead. We made some progress with this, but abandoned the concept because it could not provide texture access within a shader.

4.2. Heightmap

Our approach is heightmap-based, where pixels within an image file are used to determine the heights of each point within the terrain. We used a simple approach consisting of reading the entire terrain data and storing it on the GPU as a 32-bit texture. This imposes a limit to the size

of the data that can be used. This differs from more popular methods, which typically store the data set in RAM and copy the needed data to the GPU. We chose this approach due to its simplicity.

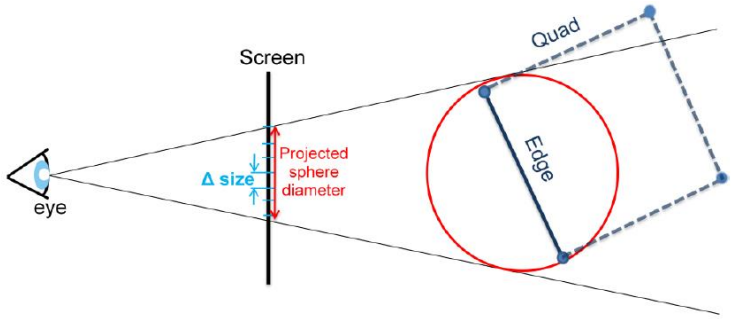

Figure 4. A technique for determining tessellation factors using the projected screen-space edge length [18].

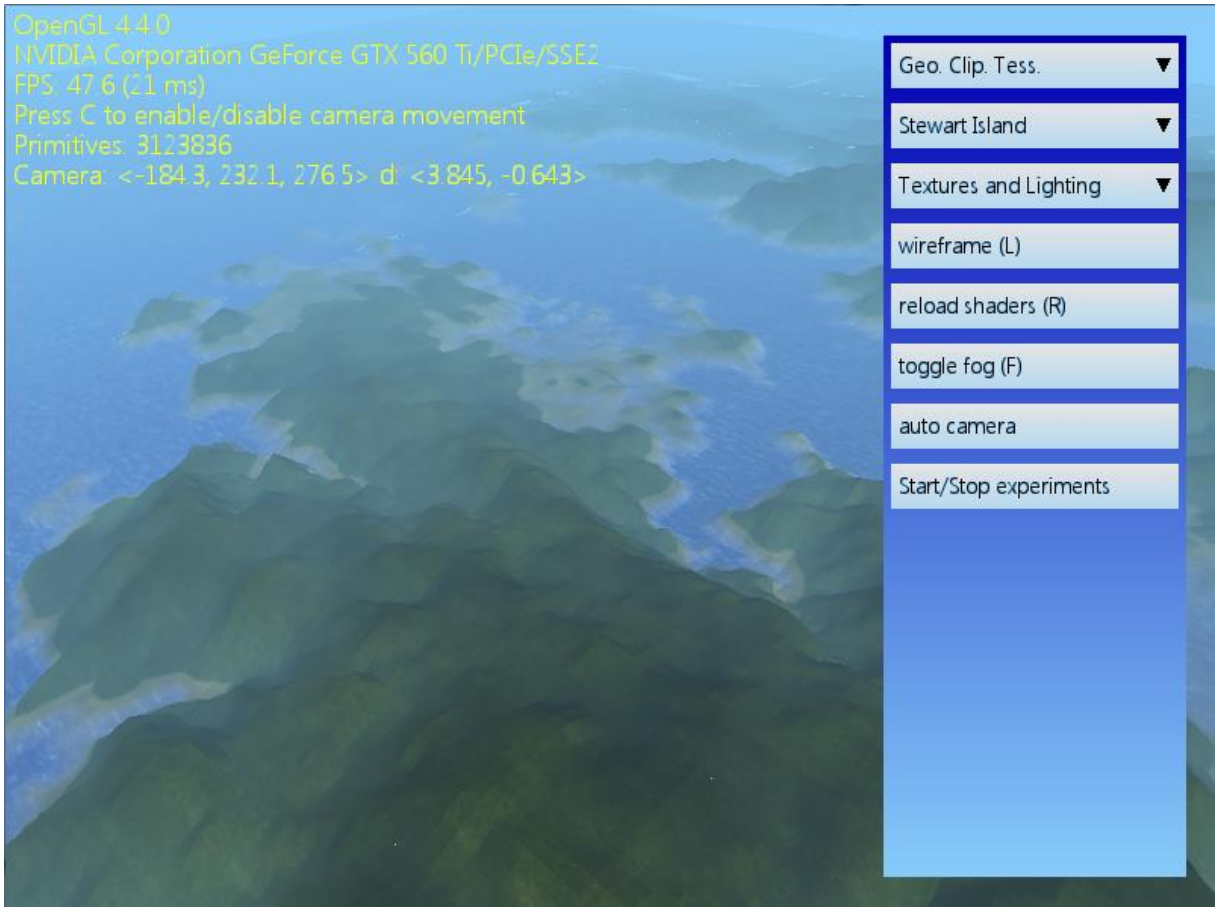

Figure 5. A screenshot of our framework running. A GUI is provided to easily switch between implemented algorithms, available heightmaps and for changing parameters.

4.3. Dynamic Level of Detail

Typical terrain rendering algorithms, based on tessellation, achieve view-dependent LOD based on the camera position by measuring the projected size in screen-space, shown in Figure 4. We use this approach to a lesser degree. Our approach inherits the nested grid-structure iconic to the Geometry Clipmaps algorithm. This passively provides view-dependent LOD due to the fact that each tile within the grid maintains approximately uniform size in screen-space.

Further LOD is achieved by selectively choosing the amount of detail to apply to each tile based on how much variation is featured in the heightmap. We use an additional texture, called a height acceleration map (HAM), which is an approach previously applied to tessellation-based terrain rendering [24]. The HAM is generated during the initialization stage after the heightmap has been loaded. We derive the HAM from the heightmap by applying a Sobel filter which approximates the image gradient. The following 3x3 matrices are applied, sequentially, to the heightmap in two passes.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

The final image is derived by averaging the result of each pass. This produces an image which approximates the gradient.

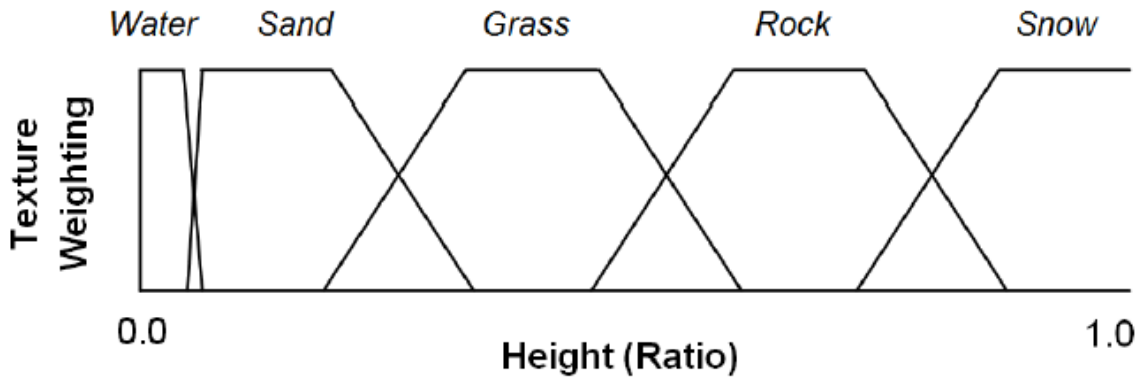

Figure 6. A method of terrain texturing which uses weights according to the terrain's elevation [30]. *Our approach uses four texture levels, instead of five.*

4.4. Clipmap Levels

The geometry for the terrain is generated during start up. This consists of four vertices defining a quad, which is repeatedly drawn as a grid across the terrain surface at different scales according to the Clipmap level. We use a different approach to rendering the various Clipmap levels. We render the entire surface as a $n \times n$ tile grid centred at the camera. For each tile we apply an offset and specify a number of patches required. This results in a surface resembling the nested grid-structure from the Geometry Clipmaps method. A part of the grid structure, illustrating the distinct tiles, is shown in Figure 7.

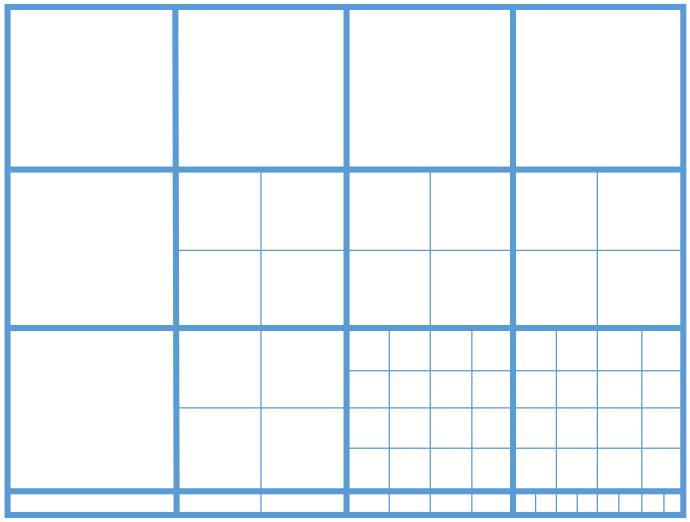

Figure 7. A section of the grid structure showing the tile arrangement. Bold lines indicate tile boundaries. Thin lines indicate tile resolution, which determines the number of patches allocated.

Algorithm 1 *Renders the set of tiles and performs culling on tiles not visible.*

```
1.  for x = 0 to gridsize then
2.      for y = 0 to gridsize then
3.          tiledirection = x,y – gridsize / 2
4.          if dotproduct(cameradirection, tiledirection) < 0.9238795 then
5.              cull current tile
6.          else
7.              xdiff = gridsize / 2 - | x – gridsize / 2 |
8.              ydiff = gridsize / 2 - | y – gridsize / 2 |
9.              patches = 2 ^ min (xdiff, ydiff)
10.             draw tile (patches, x, y)
11.         end if
12.     end for
13. end for
```

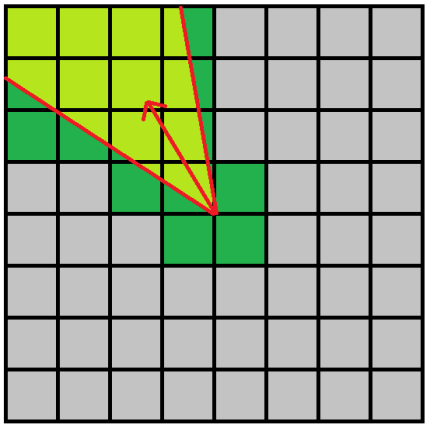

Figure 8. Our culling approach. Red lines indicate the camera view frustum. Green tiles are rendered while grey tiles are culled.

4.5. Culling

We have implemented frustum culling at the tile rendering stage, which is shown in Figure 8. Before drawing each tile, the tile is checked to see if it is visible based on the camera's position and direction in the X and Z plane. We are essentially checking whether each tile is within the view frustum, although the Y directional component is not considered. To do this, we use the camera direction and find the direction of each tile, relative to the camera, which is always between the centre tiles. The dot product of the camera direction and the tile direction results in a value between -1 and 1. We use the following expression to test whether each tile is within view:

$$v_{camera} \cdot v_{tile} < \cos\frac{\pi}{8} = \frac{\sqrt{\sqrt{2}+2}}{2}$$

where v_{camera} is the direction of the camera and v_{tile} is the direction of a given tile, relative to the camera. We have used a field-of-view (FOV) of 45 degrees, so the view expands 22.5 degrees either side of the view direction, which is why the $\pi/8$ term is used. For tiles immediately adjacent to the camera, we skip this test and render the tiles regardless. This is useful when the camera points down and would otherwise see the empty space. This procedure is detailed in **Algorithm 1**.

4.6. Texturing and Lighting

Although this was not one of our main objectives, texturing and lighting is an important aspect to achieve highly realistic renderings. To achieve a realistic rendering, we used several techniques involving texturing and lighting. Textures are used to represent the content of different areas of terrain, are applied to the terrain surface by texture mapping. We used a texturing method [30] which divides the terrain into discrete regions according to the surface elevation. A different texture is applied at different levels to reflect the type of surface expected at each height. This is done by applying a weight to each texture, which is determined by the elevation. There is an overlap at transition regions, as shown in Figure 6, producing a blended appearance between different levels and improving realism. We used four texture levels,

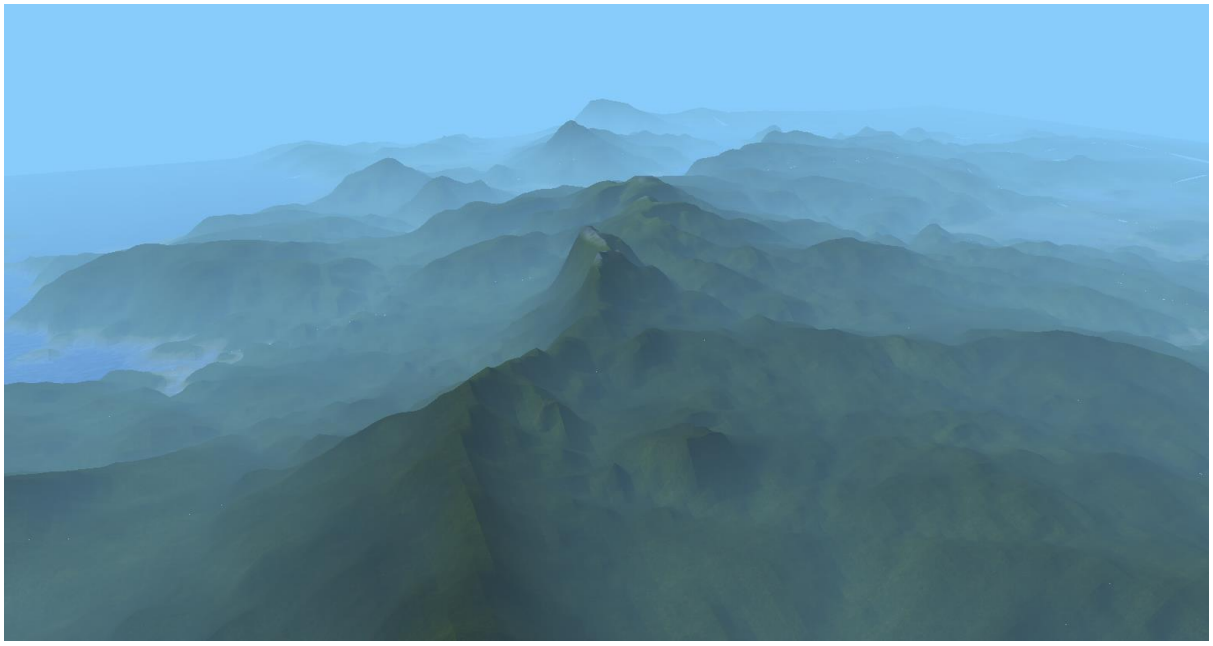

Figure 9. The rendered terrain based on our implementation, with some realistic lighting and atmospheric scattering effects.

indicating different types of surfaces on the terrain: water, sand, grass and snow. Using four textures allowed us to store texture weights in a single four-element vector.

Our approach applies a fog effect [31] which increases realism by blending distant areas into the background colour. This is an approximation to a phenomenon observed on earth due to atmospheric scattering [32]. The method is implemented in the fragment shader.

4.7. Vertex Shader

We use the instanced draw function to draw each tile. The main purpose of our vertex shader is to position the patch at the correct location within the tile. We use the `gl_InstanceID`, which ranges from zero to the number of patches indicating, to determine the offset. The following expressions are used to determine the x and y indices of a single patch.

```
xoffset = gl_InstanceID % numberofpatches;
yoffset = gl_InstanceID / numberofpatches;
```

The patch is positioned according the local offset within the tile, as well as the global offset of the tile itself. Additionally, texture coordinates are computed and output from the vertex shader.

4.8. Tessellation Control Shader

The purpose of the tessellation control shader is to determine tessellation factors to be applied by the tessellator. As we are using quad patches, we must supply four outer tessellation factors and two inner tessellation factors. We have extended an existing implementation [33], which uses the screen-space edge lengths of each side of the current quad-patch to determine the tessellation levels. This is performed by using the model-view-projection matrix to project the four corners into screen-space. We can then measure the lengths of each edge. Our implementation also takes into account the HAM. We sample one of the components of the HAM, at the given texture coordinates, from which we obtain a scalar value between zero and one. This is used, along with a constant scale factor, to derive four outer tessellation levels. The remaining two inner tessellation levels are obtained by taking the minimum of the left and right outer levels as well as the minimum of the upper and lower outer levels. We use the following equation to determine each outer tessellation level:

$$TessLevel_E = s \cdot e^{|E|} \max(HAM_E, HAM_C)$$

where E is the current edge, $|E|$ is the length of the edge in screen-space, HAM_E is the value of the HAM sampled at the neighbour patch which shares edge E, HAM_C is the value of the HAM sampled at the current patch and s is a constant scale factor.

4.9. Tessellation Evaluation Shader

The tessellation evaluation shader is used to apply a vertical offset to the tessellated vertices, according to the heightmap. After a patch is tessellated, the resulting vertices are processed by the tessellation evaluation shader. At this stage, we apply the model-view-projection matrix to

the input vertices, transforming them to clip-space. Finally, we generate the texture weights based on the elevation of each vertex.

4.10. Geometry Shader

The Geometry shader is not essential for our method, but it is used to generate normals. Normals are vectors perpendicular to an object and are useful for realistic lighting calculations. A normal map is an image where each pixel consists of three components; red, green and blue. Each pixel in the normal map represents a normal vector at that position, where the red, green and blue components denote the x, y and z directions. Normal maps are not typically supplied with a DEM, meaning it must be computed. Although there are some advantages to using a normal map, we have not used one for simplicity. Instead, we compute face-normals inside the geometry shader. This is straightforward approach involving the cross product of two edges of a face.

4.11. Implementation Limitations

We were unable to address the issue of "cracking". This is a commonly observed problem in terrain rendering, where adjacent edges are not "water-tight". This results in thin, slit-like holes in the terrain, called cracks. The main issue this presents is a reduction in image quality. The reason cracks were prevalent in our implementation is due to Geometry Clipmap's nested grid structure. Adjacent tiles of different levels are rendered at different resolutions. This means there are some points along the shared edge that only exist on one side. If those points move, holes will appear on the surface. This is a difficult problem to solve, without treating each case separately. We attempted to address this with the geometry shader, by outputting extra primitives along transition edges. Although initially promising, the approach did not work well with tessellation enabled.

5. RESULTS

5.1. Test Environment

The system used for evaluating the implemented methods was a PC running Windows 7 64-bit, with a 3.5GHz AMD Phenom II X4 quad-core CPU, 8GB RAM and an NVIDIA 1GB GTX560 Ti graphics card.

5.2. Methodology

To evaluate our implementation of Geometry Clipmaps, we performed experiments using a number of data sets. We also implemented several existing methods, which are not described in this report. These additional implementations were used for comparing performance with our algorithm. We measured the number of triangles rendered by OpenGL as well as the frame rate.

5.3. Experiment one

The first experiment was intended to compare our modified version of the Geometry Clipmaps algorithm to the two original algorithms. We obtained the heightmap data used by the researchers to generate their results for the original algorithm. To make the experiment conditions closer to the original paper, we disabled the normal generation, as well as the texturing and lighting techniques. Instead, a colour-map is sampled and applied to the rendered terrain. We also set the resolution to 1024x768, which is what the original paper used. We ran tests to see how much impact the scale factor had on performance and triangle throughput. Each test consisted of rotating around a static reference point within the scene for one minute, while observing the current frame rate and number of triangles previously rendered. This data was stored and a moving average was applied to reduce the some of the noise. Figure 14 and Figure 15 show the rendered terrain.

Figure 10 shows the results of testing over one minute, with four different values used for the scale factor. The results suggest lower scale factors achieve higher performance. However, half way through testing the largest scale factor, the performance reaches 450 FPS, which is significantly higher than any other point during testing. Figure 11 shows the throughput results. The results imply that higher scale factors lead to higher numbers of triangles rendered. During the first half of testing, however, there appears to be significant drop in throughput for the final test case.

5.1. Experiment two

Our second experiment was similar to the first. We used the same heightmap dataset and screen resolution. The scale factor was kept fixed at 64 for this experiment. We wanted to see how

much impact the different number of clip levels had on performance and throughput. To test this, we ran the same tests as in experiment one, except with a fixed scale factor. We tested

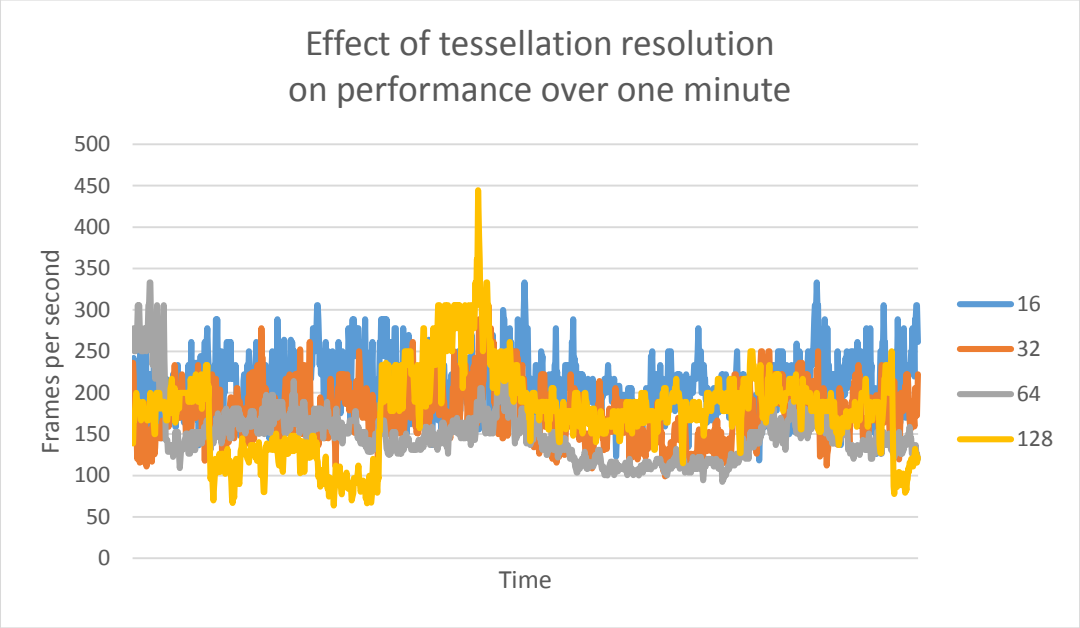

Figure 10. Performance results of testing our implementation with four different scale factors influencing the tessellation level used.

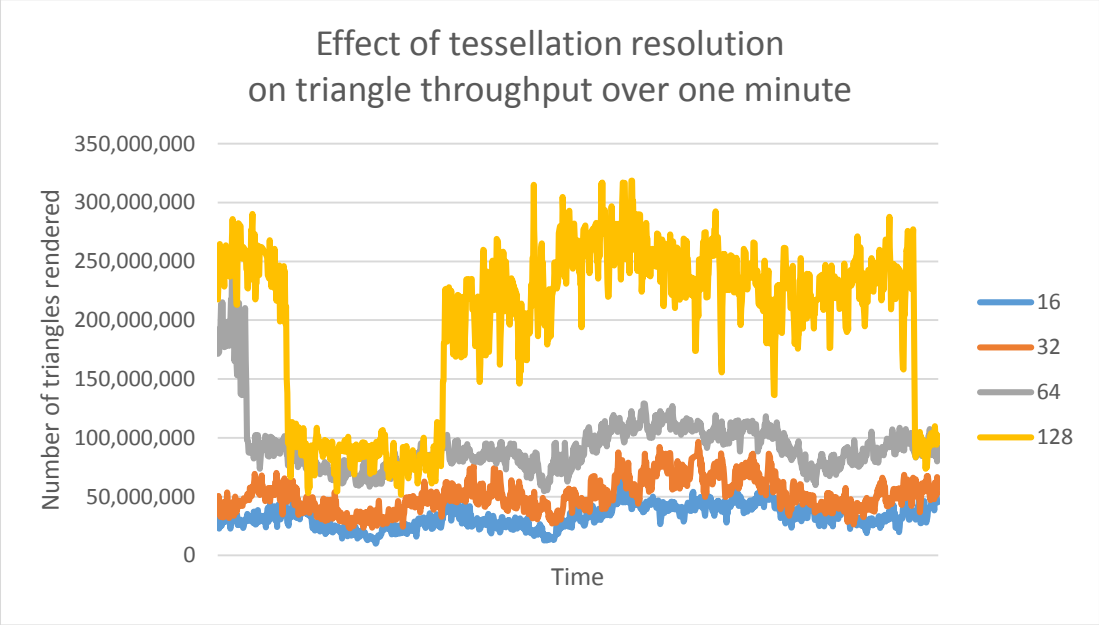

Figure 11. Throughput results of testing our implementation with four different scale factors influencing the tessellation level used.

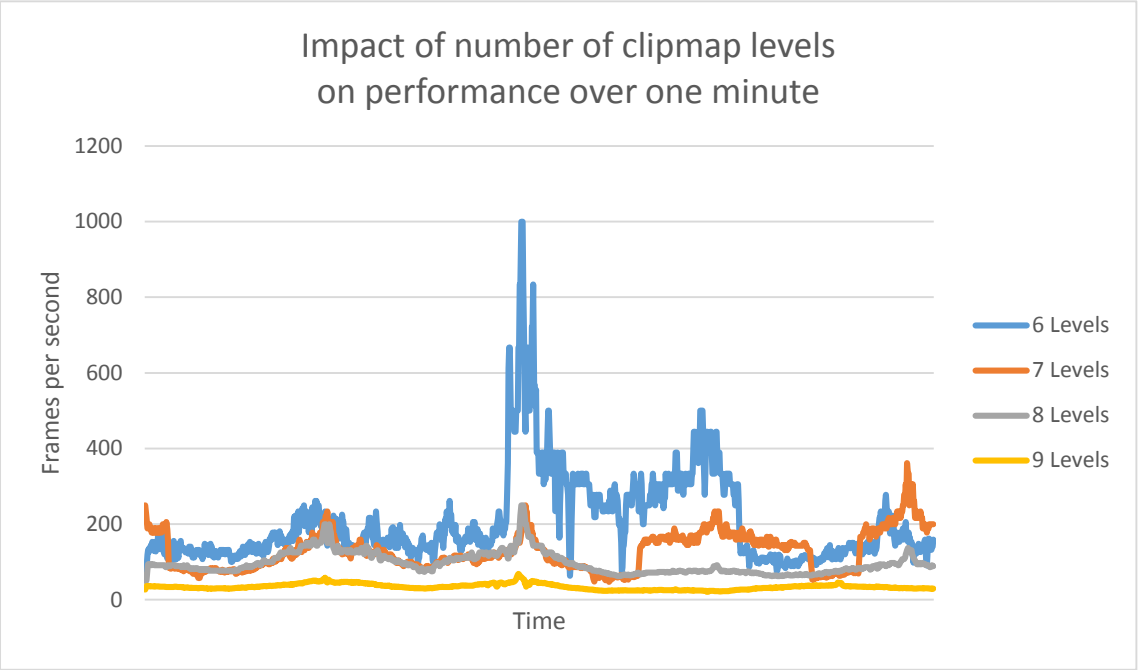

Figure 12. Performance results of testing our implementation with four different Clipmap levels.

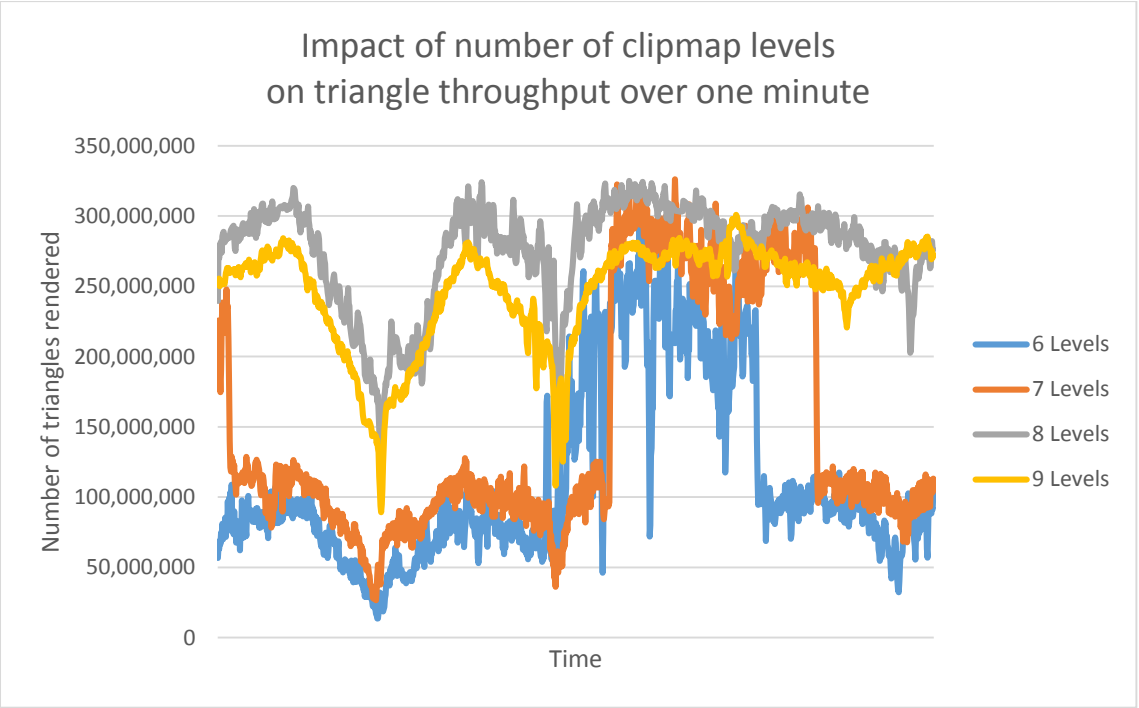

Figure 13. Throughput results of testing our implementation with four different Clipmap levels.

four clip levels. Similar to experiment one, we applied a moving average to the collected data.

Figure 12 shows performance results from testing for one minute. The results show a reduction in performance when more Clipmap levels are used. It appears that halfway into testing, the algorithm has a sharp, but brief increase in performance and is observed regardless of how many Clipmap levels are used. Figure 13 show the throughput results. The results show an increase in throughput, the more Clipmap levels are used. Using eight levels appears to result in higher throughput than using nine. There is a large gap in resulting throughput between the case of seven and eight Clipmap levels. This is observed throughout the whole test sequence, except during the third quarter where the gap narrows.

5.2. Experiment three

The final experiment involved comparing our modified Geometry Clipmaps algorithm with four of our other implementations of various terrain rendering algorithms. These were implemented in our framework, according to the available reference material. This included the research papers, which introduce them, as well as existing implementations. We compared our modified Geometry Clipmaps method with four approaches; an implementation of GPU-based geometry Clipmaps [16], two implementations of tessellation-based terrain, similar to the approach described by Cantlay [18] and an implementation of the brute-force algorithm. We tested our implementations using four different sized input datasets. For each implementation, testing lasted one minute per dataset. For these tests, lighting, texturing and other effects were enabled. Figure 17 and Figure 18 show the rendered terrain with realistic shader effects enabled.

Table 1. Throughput results of testing five implementations on various-sized input data sets.

	512x512		1k x 1k		4k x 4k		8kx 8k	
	Triangles	σ	Triangles	σ	Triangles	σ	Triangles	σ
Geo. Clip.	5.35E+08	2.62E+08	3.81E+08	79284621	3.94E+08	70120205	4.9E+08	2.24E+08
Geo. Clip. Tess.	1.19E+08	52014734	1.44E+08	44517962	1.34E+08	51112075	1.05E+08	48687946
Tess. 1	2.78E+08	11296148	2.81E+08	9173786	2.8E+08	9121148	2.8E+08	10450090
Tess. 2	1.9E+08	12083211	1.92E+08	12338100	1.9E+08	12597649	1.9E+08	12553167
Brute Force	5.19E+08	45652781	5.24E+08	37195826	5.26E+08	32217893	5.26E+08	33119596

Table 2. Performance results of testing five implementations on various-size input data sets.

	512x512		1k x 1k		4k x 4k		8kx 8k	
	FPS	σ	FPS	σ	FPS	σ	FPS	σ
Geo. Clip.	179.7088	85.9394	150.5814	73.5514	107.7136	22.2857	111.1566	19.85551
Geo. Clip. Tess.	93.89292	40.00566	114.0215	37.4254	101.3123	37.98567	82.16453	40.58455
Tess. 1	33.37147	12.29592	34.00365	12.64097	33.08605	12.48209	34.70856	12.85808
Tess. 2	86.90334	15.74642	88.32283	16.41175	86.25848	15.9824	89.62477	17.47403
Brute Force	15.47937	1.361224	15.62096	1.109064	15.6979	0.960637	15.68799	0.987523

Table 1 shows the throughput results for each method tested. We computed the mean and standard deviation for each case. The Geometry Clipmaps and brute force implementations appear to have the highest triangle counts overall, while our modified Geometry Clipmaps and the second tessellation-based implementation have the lowest. It appears that only the two Geometry Clipmap-based approaches are affected by the input dataset size. The throughput of each of the remaining three do not have a lot of variation. Table 2 shows the performance results of testing. Similar to the results of throughput, only the first two algorithms' performance appears to be dependent on the input data size. For the first implementation, the results suggest a decline in performance as the input size increases. The Geometry Clipmaps implementation gets the highest frame rate of all others for each test case, while the brute force approach consistently performs the worst.

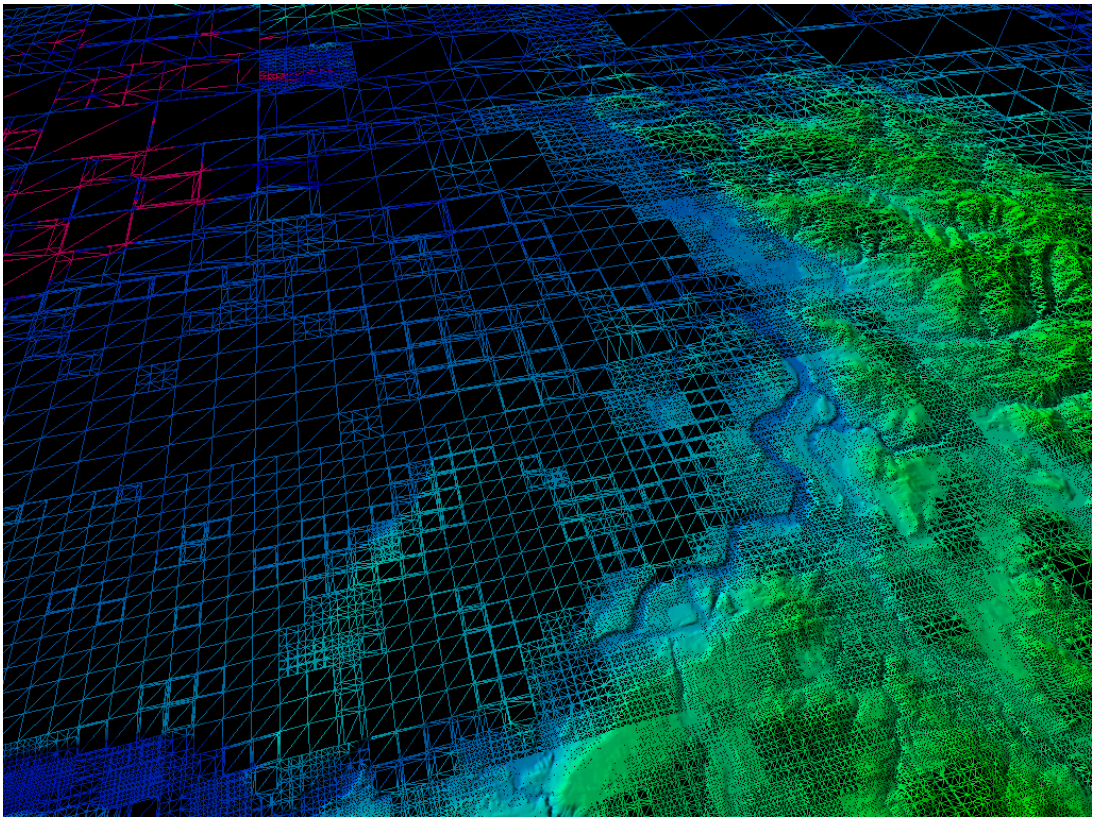

Figure 14. A wireframe view illustrating the grid layout and the curvature-based LOD.

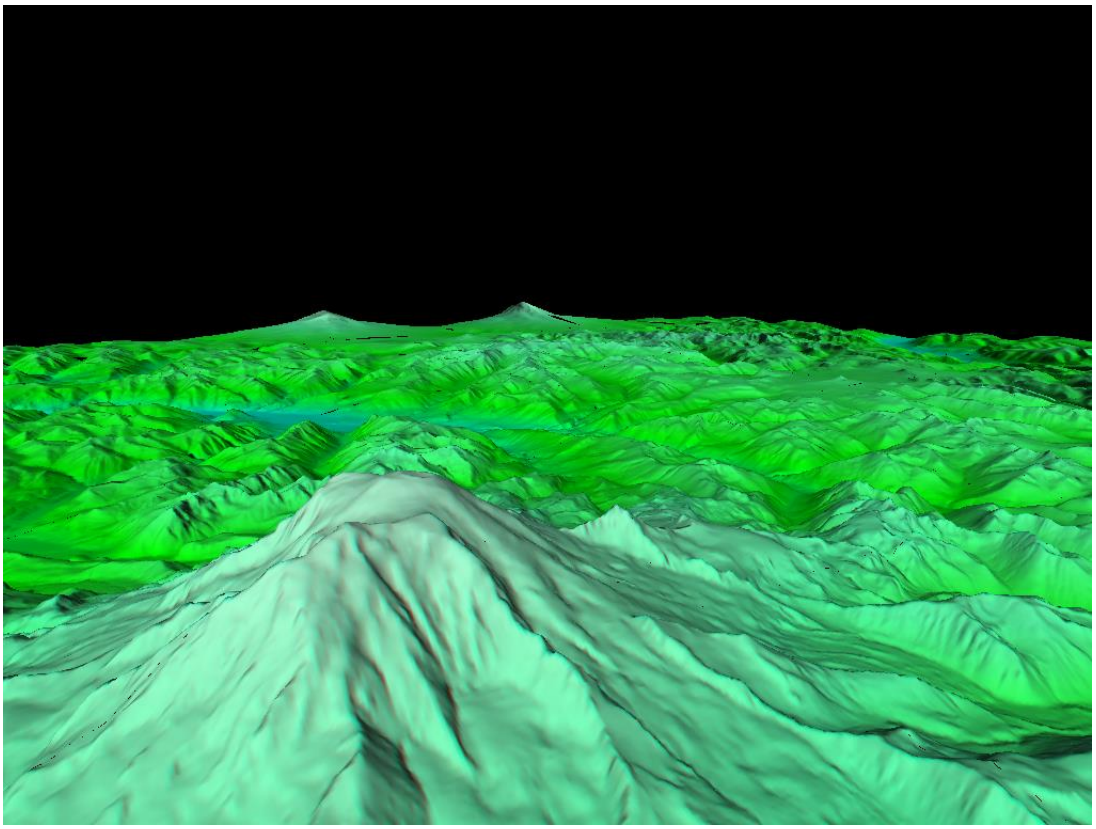

Figure 15. A view of the terrain from our modified Geometry Clipmaps implementation using the heightmap dataset used in the original paper.

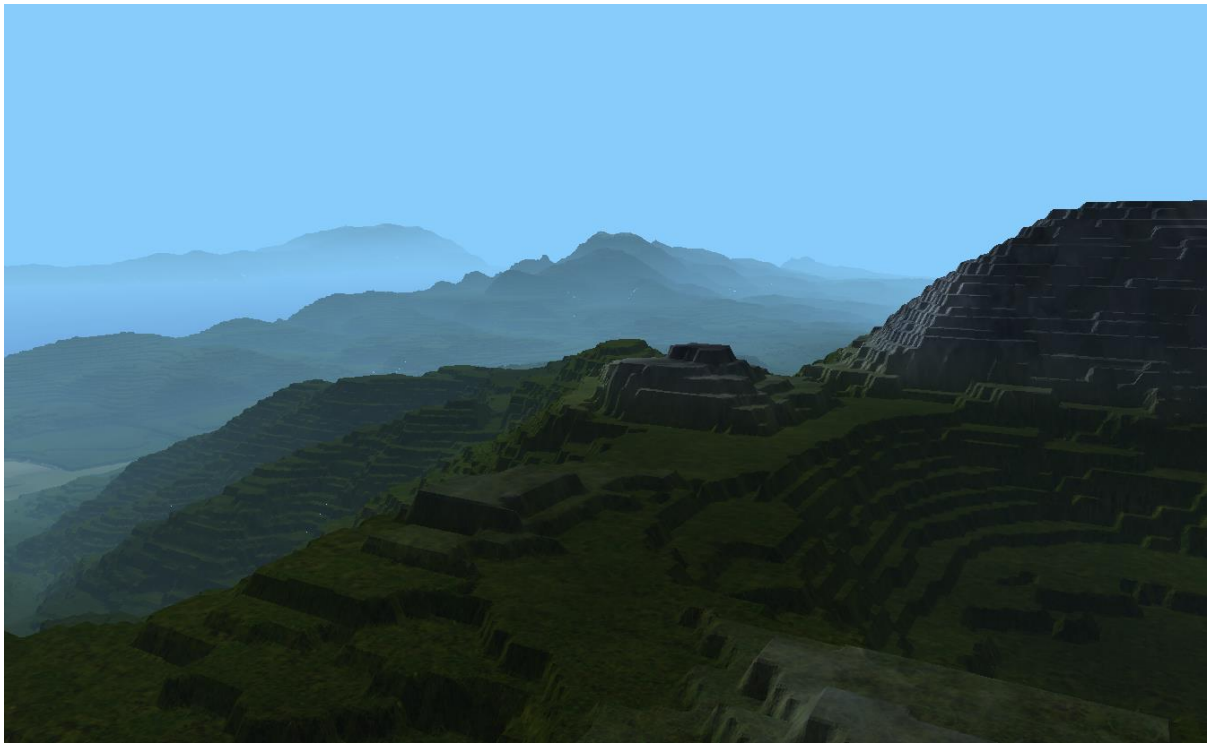

Figure 16. An 8-bit heightmap was used in this scene, which resulted in a staggered appearance, due to the low precision.

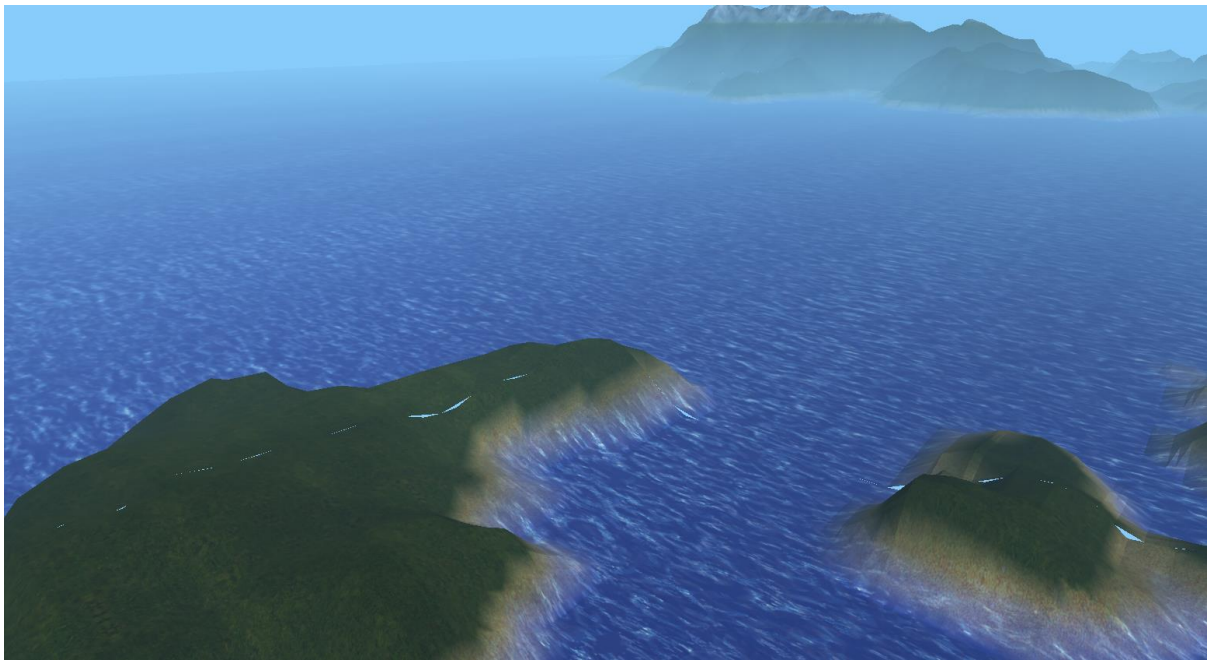

Figure 17. Terrain view illustrating the problem of "cracks" appearing between tiles.

6. DISCUSSION

6.1. Interpretation

The results of the first experiment were surprising. Our hypothesis was that using a greater tessellation level would reduce the performance. The results of the first experiment support this to an extent. However, the case where the tessellation scale factor is 128 appears to show performance roughly equivalent to using a scale factor of 32. According to the results, the scale factor 128 achieves better performance than a scale factor of 64. The exact reason for this is unclear, but may be attributed to external causes, such as background processes running on the machine.

During the final case of the first experiment, an unusual event is observed and evident in the results. During the first half of testing the case where the scale factor is 128, there is a period of approximately 15 seconds where both frame rate and throughput drop significantly. The frame rate drops to roughly 100 FPS and the throughput drops to roughly 100 million triangles. This seems to contradict the expected behaviour, where the decrease of one results in an increase of the other. We expect this because decreasing the amount of work needed results in finishing the work faster and allowing more repetitions to be performed. We can rule out the possibility of being caused by the terrain data itself, as a similar event is not observed in the other test cases. The most likely reason for this is possibly an energy-saving feature which reduces the work-load to reduce power consumption. We cannot confirm whether this is the real cause.

The results of the second experiment show an increase in the number of triangles rendered when higher Clipmap levels are used. This result is within our expectations as each Clipmap level addition results in an additional tile grid being added. We did not expect to see such a large gap between the second and third test cases. Based on the results, going from six to seven levels or eight to nine levels has little effect. However, going from seven to eight levels, resulted in roughly 150 million extra triangles being displayed. We believe this is caused by the tessellation level limit of 64. If this limit was no enforced, we would see a very high increase in throughput from eight levels to nine. When a level is added, the entire resolution effectively doubles. At eight levels, the highest resolution tile is 64, which is the limit. Therefore, when a subsequent level is added, the tiles with resolution of 64 do not receive additional patches. This means adding levels after eight have less impact.

Another observation from the second experiment is that there appears to be a period during the third quarter in which the number of triangles rendered increases dramatically for the lower level cases. We believe this is due to the terrain requiring a large amount of detail at a particular location.

In the third experiment, we observed our implementation of the original GPU-based Geometry Clipmaps algorithm performing better than our "improved" version. We expected the brute-force algorithm to perform the worst, which it did. We did not expect the Geometry Clipmaps algorithm to perform the fastest. This was most likely due to certain implementation aspects and optimizations that might have been applied to some techniques and not others. To accurately compare each algorithm, we would need to ensure they are implemented correctly.

6.2. Complexity

We aimed to reduce complexity of the implementation phase of developing a terrain rendering application. We believe this has been achieved. The framework we developed was motivated by the need to simplify the implementation stage. A lot of the aspects of terrain rendering are not implementation specific, meaning they can be generalized and reused. Without our framework, implementing an algorithm from scratch could potentially require several classes with hundreds of lines of code each. By using our framework, this is reduced to a single class and the necessary shader programs. However, this does not address specific complexity issues specific to a given algorithm.

6.3. Relating to prior work

Our modified Geometry Clipmaps algorithm was able to achieve 82.16 FPS on average with standard deviation 40.6. It was able to achieve an average throughput of 105 million triangles per second with standard deviation 48,687,946. The original paper [11] reported an average frame rate of 120 FPS and a rendering rate of 59 Million triangles per second. For the GPU-based version [16], the frame rate increased to 130 FPS and 60 million triangles per second. Although our method achieves smaller frame rate, the original papers did not apply lighting or fragment based operations, other than sampling a colour map. If we adapt our implementation by removing lighting calculations and texture sampling, we can achieve an average frame rate of 211.57 FPS. We achieve a higher frame rate than Cantlay's tessellation-based terrain algorithm [18], which runs at 102 FPS and 1920x1200 resolution.

6.4. Limitations

We do not provide a measure of LOD error. This is a value, sometimes presented alongside frame rate and triangle rate, which determines how much variation exists between the original terrain and the approximation rendered on-screen. If the error is low, then the rendered terrain will appear as it is represented. Otherwise, there may be slight deviations from the actual terrain model. Without an error-metric, it is difficult to determine whether our method provides higher image quality and accuracy.

7. CONCLUSION

A modified terrain rendering algorithm has been designed and implemented, providing good performance and realistic terrain. We have been able to improve performance compared to the original Geometry Clipmaps algorithm, and its GPU-based improvement, by up to 71%. Our method uses the nested grid structure, similar to the original algorithm. Tessellation is used to selectively apply more resolution where it is required. We determine this using a height acceleration map, which approximates the image gradient and is used to estimate how much variation in elevation exists across the surface.

Our main objective was to investigate whether older terrain rendering algorithms could benefit from recent hardware and API features. We were interested in finding out if performance can be improved and if complexity can be minimized. We were also concerned about maintaining sufficient image quality. Without a way to measure LOD error, it is difficult to determine if our method satisfies this. We believe our implementation does improve performance and address complexity.

We hope this work will benefit anyone interested in implementing terrain, extending existing methods or inventing new techniques.

7.1. Future Work

As mentioned in section 4.11, the implemented method suffers from the "cracking" problem. A potential direction for future research would explore solutions to this problem. One simple approach would be to simply "cover up" the holes using the geometry shader. Some detection method could be implemented to detect when a neighbouring edge is a different resolution. A primitive could be generated to fill the gap. Alternatively, a method similar to Seamless Patches [15] could be applied to render transition strips between edges of different resolution.

Further future work would investigate ways to utilize larger datasets. One of the original goals for this research was to apply a large, 800-megapixel[3], dataset of the planet earth, provided to the public by NASA [34]. This turns out to be quite difficult as the total, uncompressed data can be as high as 1.6GB[4], which exceeds the memory capacity of many graphics cards. Furthermore, OpenGL currently enforces an upper limit for textures to 16384x16384, meaning the data could not be represented as a single texture as traditional techniques do. Two possible

[3] The data consists of eight separate data files of 10800x10800 pixels each.

[4] Assuming a heightmap where 16 bits are used for each pixel.

ways to address this are compression and streaming [35]. Compression reduces the amount of space needed to represent the data, while streaming actively brings data in and out of memory when it is needed. These techniques offer interesting avenues for future research.

REFERENCES

[1] J. Star and J. E. Estes, *Geographic Information Systems: An Introduction*. Prentice Hall, 1990, p. 303.

[2] L. J. Prinzel and L. J. Kramer, "Synthetic Vision Systems," in *International Encyclopedia of Ergonomics and Human Factors*, 2nd ed., W. Karwowski, Ed. Talyor & Francis., 2006, pp. 1264–1271.

[3] J. H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms," *Commun. ACM*, vol. 19, no. 10, pp. 547–554, 1976.

[4] H. Hoppe, "Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering," in *Visualization*, 1998, pp. 35–42.

[5] "OpenGL," 2014. [Online]. Available: https://www.opengl.org/. [Accessed: 14-Jun-2014].

[6] M. Segal and K. Akeley, "The OpenGL Graphics System : A Specification (version 4.0)," Silicon Graphics, Inc. Available: https://www.opengl.org/registry/doc/glspec40.core.20100311.pdf, 2010.

[7] P. Cignoni, E. Puppo, and R. Scopigno, "Representation and visualization of terrain surfaces at variable resolution," *Vis. Comput.*, vol. 13, no. 5, pp. 199–217, 1997.

[8] H. Hoppe, "Progressive Meshes," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 99–108.

[9] H. Hoppe, "Smooth view-dependent level-of-detail control and its application to terrain rendering," in *Proceedings. Vis. '98*, 1998, pp. 35–42,.

[10] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein, "ROAMing terrain: Real-time Optimally Adapting Meshes," *Proceedings. Vis. '97 (Cat. No. 97CB36155)*, pp. 81–88, 1997.

[11] F. Losasso and H. Hoppe, "Geometry Clipmaps : Terrain Rendering Using Nested Regular Grids," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 769–776, 2004.

[12] W. H. De Boer, "Fast Terrain Rendering Using Geometrical MipMapping," Online article. http://www.flipcode.com/archives/article_geomipmaps.pdf, 2000.

[13] T. Akenine-Moller, E. Haines, and N. Hoffman, "Image Texturing," in *Real -Time Rendering*, 2nd ed., Natick, Massachusetts: A K Peters Ltd, 2002, pp. 133–137.

[14] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "BDAM - Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization," *Comput. Graph. Forum*, vol. 22, no. 3, pp. 505–514, Sep. 2003.

[15] Y. Livny, Z. Kogan, and J. El-Sana, “Seamless patches for GPU-based terrain rendering,” *Vis. Comput.*, vol. 25, no. 3, pp. 197–208, Mar. 2008.

[16] A. Asirvatham and H. Hoppe, “Terrain Rendering Using GPU-Based Geometry Clipmaps,” in *GPU gems 2*, 2nd ed., M. Pharr and R. Fernando, Eds. Addison-Wesley, 2005, pp. 27–46.

[17] N. Brettell, “Terrain Rendering Using Geometry Clipmaps,” Dept. Comput. Sci. Software Eng., Univ. Canterbury, Christchurch, NZ, Tech. Report, 2005.

[18] I. Cantlay, “Directx 11 Terrain Tessellation.” Nvidia, Tech. Rep., 2011.

[19] E. Yusov and M. Shevtsov, “High-Performance Terrain Rendering Using Hardware Tessellation,” *J. WSCG*, vol. 19, no. 3, pp. 85–92, 2011.

[20] X. Bonaventura, “Terrain and Ocean Rendering with Hardware Tessellation,” in *GPU Pro 2*, W. Engel, Ed. A K Peters Ltd, 2011, pp. 3–14.

[21] A. K. Jakobsen, “Tessellation based Terrain Rendering,” M.S. thesis, Dept. Comput. Inform. Sci., Norwegian Univ. Sci. Technology, Trondheim, Norway, 2012.

[22] O. Ripolles, F. Ramos, A. Puig-Centelles, and M. Chover, “Real-time tessellation of terrain on graphics hardware,” *Comput. Geosci.*, vol. 41, pp. 147–155, Apr. 2012.

[23] H. Kang, H. Jang, C.-S. Cho, and J. Han, “Multi-resolution terrain rendering with GPU tessellation,” *Vis. Comput.*, May 2014.

[24] A. Valdetaro, G. Nunes, A. Raposo, B. Feijo, and R. De Toledo, “LOD terrain rendering by local parallel processing on GPU,” in *SBGAMES 2010: Proceedings of the 9th Brazilian Symposium on Games and Digital Entertainment*, 2010, pp. 169–176.

[25] D. Feldmann and K. Hinrichs, “GPU based Single-Pass Ray Casting of Large Heightfields Using Clipmaps,” in *Proceedings of Computer Graphics International (CGI)*, 2012.

[26] C. Dick, J. Krüger, and R. Westermann, “GPU Ray-Casting for Scalable Terrain Rendering,” in *Eurographics 2009: Proceedings from the 30th Annual Conference of the European Association for Computer Graphics*, 2009.

[27] R. Marques, D. Informática, U. Minho, and L. P. Santos, “GPU Ray Casting,” in *17th Portuguese Conference on Computer Graphics*, 2009, pp. 83–91.

[28] S. Wiendl and S. C. Dick, “GPU-Aware Hybrid Terrain Rendering,” Dept. Comput. Sci., Univ. Munich, Germany, Tech. Report, 2013.

[29] C. Dick, J. Kruger, and R. Westermann, “GPU-Aware Hybrid Terrain Rendering,” *Int. J. Comput. Inf. Syst. Ind. Manag. Appl.*, vol. 3, pp. 820–827, 2011.

[30] K. Nicholson, “GPU Based Algorithms for Terrain Texturing,” Dept. Comput. Sci. Software Eng., Univ. Canterbury, Christchurch, NZ, Tech. Report, 2008.

[31] M. Nuebel, “Introduction to Different Fog Effects,” in *Introductions & Tutorials with DirectX 9*, W. Engel, Ed. Wordware Publishing, Inc, 2004, p. 151.

[32] A. T. Young, “Rayleigh scattering,” *Phys. Today*, vol. 35, no. 1, pp. 42–48, 1982.

[33] G. Sellers, R. Wright, and N. Haemel, “Tessellation Example - Terrain Rendering,” in *OpenGL SuperBible*, 6th ed., Addison-Wesley, 2013, pp. 300 – 303.

[34] “NASA’s Visible Earth. A catalog of NASA images and animations of our home planet.” [Online]. Available: http://visibleearth.nasa.gov/. [Accessed: 20-Mar-2014].

[35] M. Mittring, “Advanced virtual texture topics,” in *ACM SIGGRAPH 2008 Games*, 2008, pp. 23–51.