GPU ENHANCED GLOBAL TERRAIN RENDERING

by

Marcus Craig Tyler

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in the
Department of Computer Science

Charlotte

2007

Approved by:

____

Dr. Kalpathi R. Subramanian

____

Dr. Zachary Wartell

____

Dr. Aidong Lu

©2007
Marcus Craig Tyler
ALL RIGHTS RESERVED

ABSTRACT

MARCUS CRAIG TYLER. Terrain rendering utilizing level of detail techniques optimized to make use of modern GPU's allowing for real-time rendering of global terrain data. (Under the direction of DR. K.R. SUBRAMANIAN and DR. ZACHARY WARTELL)

With emphasis being placed on high fidelity terrain rendering, a majority of research has been in the area of continuous level of detail algorithms. Such algorithms constantly change the topology of the landscape and make it difficult for graphics processors to cache this data efficiently. These algorithms also place considerable amounts of processing on the CPU to calculate the vertices to display every frame.

The methods presented in this paper go back to traditional discrete level of detail calculations which allow for better caching of data for blocks of terrain. We present an efficient algorithm for generating and storing this information for quick access at run-time for very large datasets. We also present a method for pre-computing geomorph information thus reducing computation time as well as data throughput for reducing popping artifacts. We show that focusing on efficient data transfer on modern hardware can produce interactive rates while reducing run-time computation on the CPU.

TABLE OF CONTENTS

LIST OF FIGURES vi

CHAPTER 1: INTRODUCTION 1

CHAPTER 2: TERRAIN RENDERING 4

2.1. Terrain Data 4

2.2. Mesh Representations 5

2.3. Levels of Detail 9

2.4. GeoCoordinates 18

CHAPTER 3: GPU OPTIMIZED MESH 20

3.1. Vertex Buffer Objects 21

3.2. Programmable Shaders 21

3.3. Progressive Levels of Detail 22

3.4. Refinement Algorithm Details 26

3.5. Data Storage 29

3.6. Preprocessed Geomorphs 31

CHAPTER 4: GLOBAL TERRAIN 34

CHAPTER 5: ANALYSIS 36

5.1. Baseline Performance 36

5.2. Progressive Vertices 38

CHAPTER 6: CONCLUSIONS 43

CHAPTER 7: FUTURE WORK 45

7.1. Crack Mending 45

7.2. Better error metrics 45

7.3. Cliffs and Overhangs 45

BIBLIOGRAPHY 47

APPENDIX A 49

LIST OF FIGURES

FIGURE 1: Triangulated Irregular Network (TIN) 6

FIGURE 2: Three levels of a quad-tree using longest edge bisection. 7

FIGURE 3: First 4 levels of a bin-tree 8

FIGURE 4: Relationship of a bin-tree node and its neighbors 8

FIGURE 5: Terrain with higher level of detail placed on the users focal point 11

FIGURE 6: Triangle fan being tested for decimation. 13

FIGURE 7: Cracks formed in a quad-tree with varying levels of detail between neighbors. 17

FIGURE 8: Resolving T-Junctions in binary trees through recursive splits. 18

FIGURE 9: The second level of detail uses all the vertex information from the first plus the extra vertices required for the new detail. 23

FIGURE 10: Recursively traversing the quad-tree in SW, NW, NE, SE causes reordering of indices from one level to the next 25

FIGURE 11: a) Triangle fans with recursive refinement and b) restricted refinement 28

FIGURE 12: Π-Order space filling curve 29

FIGURE 13: Coarse LOD (left) showing a point being added to fine LOD (right). 31

CHAPTER 1: INTRODUCTION

As computer hardware changes so do the programs that run on them. It is a constant battle to optimize the bottlenecks presented by hardware. An optimal algorithm one year may not be the best the next due to improvements in processing power or even completely new components designed to ease the burden from the CPU. Such a shift has happened in recent years with modern graphics processors and their interfaces.

Algorithms to efficiently render large areas of terrain have been in development for many years. During that time computer hardware has changed drastically placing more power within graphics hardware. In the early developments of terrain algorithms emphasis was placed on reducing the number of polygons rendered on the GPU as it, and the pipe to it, were clearly the bottleneck. Today that bottleneck has been greatly reduced through better processors and faster throughput to them.

Still yet, when working with millions of primitives even these vast improvements are not enough for us to throw caution to the wind. We must still take care in our data structures and algorithms to be as efficient as possible and not over tax the system so as to not slow other rendering systems . Using a mixture of traditional level of detail algorithms with modern graphics hardware in consideration we propose a modern approach to a long studied problem – efficient rendering of global terrain.

In this work we focus on the problems facing terrain rendering such as handling the large amounts of data, levels of detail techniques, cracks between level of detail as well as more modern techniques such as programmable GPUs and how they can be used to morph between levels of detail. By taking advantage of the extra processing power of the GPU, we can achieve better frame rates over other implementations that put a tremendous amount of load on the CPU. While previous techniques strive for better accuracy by finely tuning triangle counts at run-time, we focus on throughput speed and take advantage of the GPU's ability to handle the extra vertex data to compensate for the lack of finely tuned triangle counts.

After covering techniques used in terrain rendering we then present our solution to these problems. The contributions of this thesis are as follows:

1. *Implementation of pre-processed chunked discrete levels of detail*. By creating discrete levels of detail offline the amount of runtime computation is greatly reduced. We offset lack of granularity and thus precision by allowing for more vertices to be sent.
2. *GPU enhanced data structures for more efficient use of the GPU pipeline*. Since we will be sending more vertices the data needs to be optimized for faster throughput to the GPU. We present a progressive method for sending levels of detail as it is needed within a given block of terrain.
3. *Pre-processed height differentials for use in morphing between levels of detail to reduce popping of geometry*. This also saves computation time on the CPU and

GPU. Since we are already processing the height field for LOD information, it is a logical next step to also process offsets to assist in morphing between levels.

4. *Optimized triangle fan generation for reducing the triangle count required to render a terrain mesh without sacrificing geometric fidelity*.
5. *A simple algorithm for eliminating T-Junctions between neighboring quad-nodes within a block of terrain*. We eliminate the need to recurse down neighboring quad nodes in order to fill cracks when placing higher detail near quad boundaries.

CHAPTER 2: TERRAIN RENDERING

This chapter reviews issues involved in rendering global terrain data and previous work on how these issues have been addressed. Rendering such large amounts of data requires techniques in data storage, mesh representation, decimation and data precision when working with such large units of measure. Following this chapter is our approach to these issues and details on its implementation.

2.1. TERRAIN DATA

Terrain data has some unique qualities that help make creating triangle meshes easier than other 3D geometry such as solid models. The methods for acquiring terrain data usually involve techniques that only allow for single height values per sample point along a two dimensional plane. This greatly simplifies the way data can be stored as well as how we can represent the data as a three dimensional mesh. Since only a single height value is stored for each sample point, it cannot represent any form of cliff or overhang.

With only a single height value for each point along a plane, we can store this information in a two dimensional array of height values. Each dimension of the array can then be translated in to coordinates along the X-Y plane and the height value along the Z axis. This representation fits nicely with a grayscale image where each pixel value is the height offset from sea level with darker values representing lower values. This representation makes it easier for human consumption when

working with the raw data while been an efficient storage mechanism. The GeoTIFF standard (1) defines a file format for storing height values in a grey scale image as well as meta-data defining the geo-coordinates of the block of data described in the file. This meta-data is used to place the block of terrain in the appropriate location and scale.

2.2. MESH REPRESENTATIONS

There are two ways in which we can represent terrain data in a 3D mesh. The most straightforward is to create a regularly spaced grid of triangles where each vertex represents a sample point in the dataset. Reading in height values from our heightmap file and using them as the Y value in a 3D mesh, we can construct rendering of the terrain. This is a very popular method used by most of the recent work in terrain rendering. While easy to implement this approach is not the most efficient when one takes into consideration that some areas of the terrain do not change in height as much as others and can therefore be represented with fewer triangles. Another approach is to allow variable spacing between vertices allowing for irregular placement where flatter areas of the terrain can be represented with fewer triangles (2).

Triangulated irregular networks (TINs) can more efficiently represent a terrain mesh by limiting vertex placement to areas of high curvature. Large flat areas can be reduced to just a few triangles thus saving potentially large amounts of memory. At worst TINs place vertices at the same sample points as an regular grid method. While efficient at representing triangles, TINs are difficult to generate. Few earlier

works, such as (2), used TIN's to reduce the number of triangles, but with current hardware the emphasis on fewer triangles is not as important as the speed of the algorithm to generate an approximation.

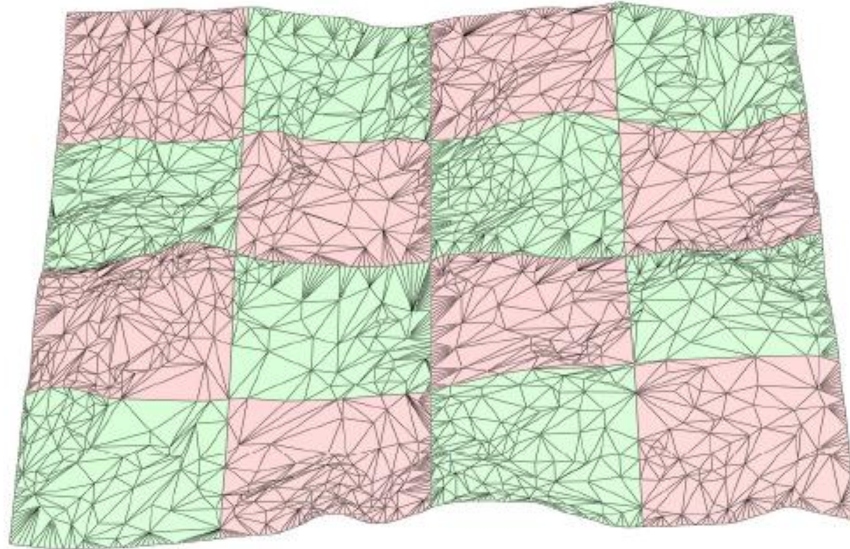


FIGURE 1: Triangulated Irregular Network (TIN) (2)

Regular grids on the other hand are very easy to generate and allow for easier manipulation. Due to the regular nature of the mesh, the data can be stored as single height values as opposed to TINs which require the X and Y components to be stored as well. Finding neighboring points in a regular mesh is trivial while the same function within a TIN requires a bit more work (2).

2.2.1. QUAD-TREES

In order to quickly navigate a mesh of data we must store the data in a manner that allows us to quickly find vertices. One such method is to store the data in a tree structure by splitting the grid into four quadrants where each quadrant is again split

recursively forming what is known as a quad-tree (3). The root of this quad-tree is the center point of a square grid whose dimensions are $2^n + 1$. Using this dimensioning allows for even partitioning of the grid into quadrants without creating nodes between sample points. To find the children of a node, the current quadrant is split into four equal quads where the center of each quad represents the child node. Successive levels can be generated by continually subdividing each quadrant in the same manner.

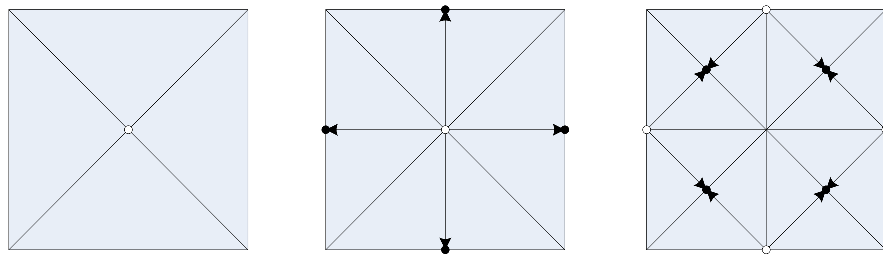


FIGURE 2: Three levels of a quad-tree using longest edge bisection.

Lindstrom et. al. used a variation on this quad-tree called a restricted quad-tree that splits the quad based on the longest edge (4). As can be seen in FIGURE 2 the children of a root node fall on the longest edge, or diagonal, of each quadrant. The tree is constructed using a Directed Acyclic Graph to navigate through the tree quickly. (5) uses a similar quad-tree with a variation that places the child nodes at the center of each quadrant. These methods have the benefit of being able to use index operations and recursion to navigate the tree rather than pointers.

2.2.2. BIN-TREES

Duchaineau et. al. introduced a binary partitioning method of a square mesh in (6) by splitting the quad down its diagonal to form two top level nodes in a binary tree. Each level of the tree is formed by splitting the current triangle node from its apex to the midpoint of the base edge. The first four levels of such a bin-tree can be seen in FIGURE 3. Navigating this tree is exactly like navigating a normal binary tree and can therefore make use of index navigation as opposed to allocating memory for pointers to nodes.

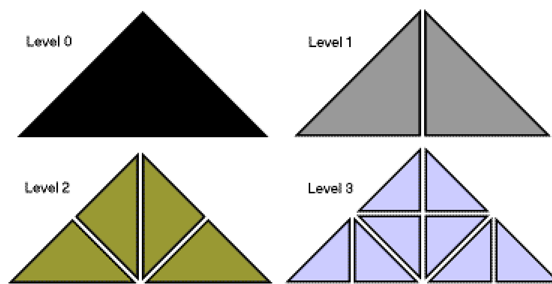


FIGURE 3: First 4 levels of a bin-tree (6)

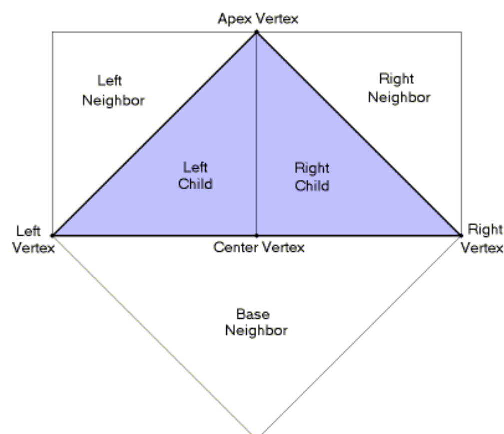


FIGURE 4: Relationship of a bin-tree node and its neighbors (6)

2.3. LEVELS OF DETAIL

Now with a basis for what makes up a terrain mesh, we can look at how to limit the number of vertices and triangles that are rendered in any given frame. We've already covered one method by using TINs to pare the mesh down to a minimum number of triangles to represent the terrain accurately. In this section we will cover another technique that takes into consideration other error metrics for reducing the number of vertices, such as distance to the user. Level of detail methods use different methods to determine the fewest number of vertices required to render a scene such that there is no perceptual loss in quality to the user. In the following sections we will discuss how one goes about determining which vertices to display in any given frame.

2.3.1. DISCRETE LODS

The first LOD techniques used were based on discrete levels of pre-computed detail in an offline process which were then selected for rendering at runtime based on some error threshold. Selected for its simplicity, discrete level of detail calculations require little runtime processing making them ideal when CPU processing time is limited. The trade-off however is steep. Sacrificing CPU time for vertex count, one must either sacrifice quality by sending lower detail blocks than needed or select a higher level needed and fill the memory pipeline to the GPU bringing frame rates down.

Until recent advances in GPU technology (7), discrete level of detail algorithms had been put to the way side in favor of continuous levels of detail which create

more precise mesh representations thus reducing the number of primitives required while maximizing visual fidelity. With finer control over how data is stored on the graphics processor, discrete methods are beginning to make an appearance again (8). A few techniques such as (9) (10) even use virtual mip-mapped textures as input for vertex data making use of texture throughput and caching to minimize transferring of data.

2.3.2. CONTINUOUS LODS

As stated continuous level of detail algorithms give much finer control over how geometry is refined. Instead of a set number of levels, continuous LOD methods calculate detail in a progressive manner allowing for a wider variation of detail. Since its first use in terrain rendering (4) continuous LOD has been a very popular method when screen space accuracy is of utmost concern. The trade-off for continuous LODs is CPU processing time. At the time when these methods were popularized graphics hardware was not capable of handling vast amounts of primitives. To make CLODs real-time, importance is placed on optimizing the algorithm so that it is as efficient as possible. Quickly eliminating large portions of geometry is key and has been the focus of most of the research in this area.

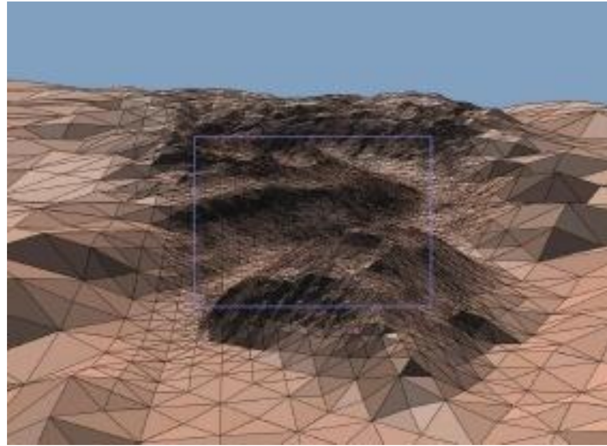


FIGURE 5: Terrain with higher level of detail placed on the users focal point (4)

2.3.3. ERROR METRICS

The goal of any LOD technique is to remove vertices from a mesh without degrading the visual quality. Simplifying a piece of geometry changes its appearance. As vertices are removed we must approximate the actual geometry. Even with the best algorithms some error is inevitable so we must minimize it. In order to minimize error we must first measure it. The practice of measuring error is the hope that more accurate error measurements lead to better refinement algorithms to reduce that error. The following sections explore two popular techniques used to calculate error which will then be used to reduce visual errors when rendering the scene.

SCREEN SPACE ERROR. When referring to error in approximating a piece of terrain, we are talking about the difference in the images produced by the actual geometry versus the image rendered with the approximation (11). This approach to measuring error is used by a large portion of the papers on the topic of terrain

rendering as it is ultimately how a user will judge the quality of the algorithm used to approximate the scene. Screen space error is ideally suited for continuous level of detail algorithms such as those discussed in 2.3.1.

The issue with using a screen space error is that it is traditionally a run-time error metric. This means to use the error metric you must take into consideration the viewing parameters. Any approximations based on this metric must also be performed at run-time. While this may be suitable for continuous level of detail methods, it is not useful for determining approximations in a pre-process for computing discrete levels of detail. As always there is a tradeoff between precision and speed. But in some cases, as we will discuss in the next section, brute force can help make up the difference.

GEOMETRIC ERROR. Geometric error measures distances within the actual model geometry before rendering (11). Measuring error in this way has the potential for displaying more artifacts than screen space error due to the fact that it is not based on runtime information from the viewing orientation. Geometric error only takes into consideration the features of the geometry regardless of where it will be viewed from. The advantage of using geometric error is that it is easier to implement and can be determined prior to any rendering in a pre-processing step.

To overcome any artifacts due to inaccuracies of using geometric error, generally the implementing algorithm takes a very conservative approach as to when to refine a piece of geometry (8). While this generates more polygons than necessary, using modern GPU's and better throughput management of data, this

latency can be minimized and in some cases eliminated through caching. Through the use of Vertex Buffer Objects we can tell the GPU to hold certain blocks of data in memory so the data does not have to be resent each frame. The static nature of terrain data along with predictable movement over the terrain allow this type of strategy to work.

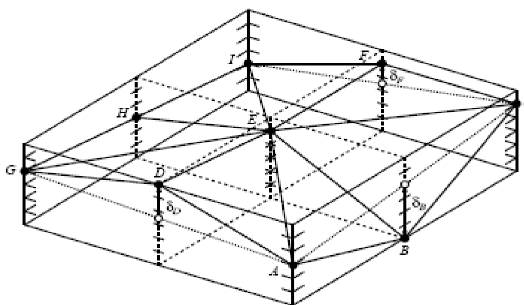


FIGURE 6: Triangle fan being tested for decimation (4)

Calculating geometric error is generally straightforward. To determine when a vertex needs to be rendered a similar approach to (4)'s fine grained simplification can be used. FIGURE 6 shows a triangle fan to be tested for decimation in this algorithm. To determine if vertex B is a candidate for removal the distance from the actual point B to a point on the line AC is calculated. By measuring the distance between these points the error introduced by removing the vertex can be determined. If the difference falls within the current LOD threshold then the vertex can be rejected. For very flat areas this can reject a large portion of vertices that will never be rendered or even stored as in the case of pre-processing the data (8).

2.3.4. REFINEMENT TECHNIQUES

There are two different ways of looking at creating different levels of detail for a terrain mesh. Either traverse the tree in a bottom-up fashion where the highest level of details is progressively coarsened until the desired error threshold is met, or traverse the tree top-down going from coarse to fine in a similar manner. The following sections explore the trade-offs between the two and how they can be used together depending on the type of refinement required.

BOTTOM-UP. Since Lindstrom first used a bottom-up approach in (4) it has rarely been used on its own due to its shortcomings. By starting with the finest level of detail any algorithm that uses this approach is immediately dependent on the size of the sampled data. Not only is the memory requirement a hindrance, but the time to process every node recursively in this manner can be too costly. Another limiting feature of starting from the highest level of detail in real-time rendering is fitting the algorithm within a time constraint. Most real-time systems require certain subsystems to complete within a set amount of time, however if one were to stop processing from a bottom-up approach the number of vertices may still be too large to render in real-time.

TOP-DOWN. Resolving the issues of the bottom-up approach is simple. By traversing the tree from coarse to fine levels of detail, the refinement algorithm is no longer dependent on the size of the mesh. Once a certain error threshold is met, the refinement can stop thus chopping off potentially large portions of data. Most of the algorithms referenced in this paper use some form of top-down algorithm. Some

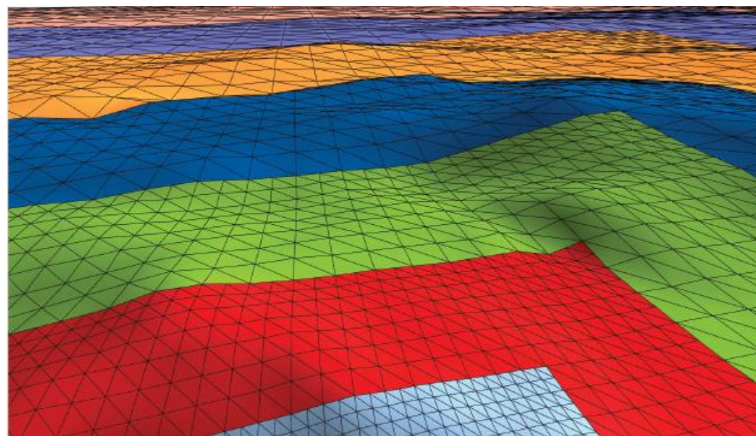
even use a combination of both as in (6) where a bottom-up approach is used to refine a mesh from a previously rendered scene taking advantage of frame to frame coherence.

Taking a top down approach can also benefit from having a time budget whereas a bottom-up cannot (6). If the system demands the refinement algorithm finish its processing within a certain time constraint, a top-down approach simply stops processing after the allotted time. If the algorithm was not finished, the rendered scene will suffer from larger errors, but will remain responsive. Using a bottom-up approach on the other hand would end up sending more vertices than required and most likely bring the rendering system to a crawl. In the end one must weigh the advantages of performance over the quality of the rendered scene.

FRAME COHERENCE. An important consideration in continuous level of detail calculation is to take into account the previous frames information as introduced in (6). A convenient trait of terrain rendering is the fact that the level of detail does not change drastically from one frame to the next. By taking advantage of this, one can start with the previous LOD information and refine it to meet the requirements of the current frame. In their paper (6) actually use a bottom-up approach to frame coherence in their dual-queue ROAM implementation. This saves a considerable amount of time especially when little change is required for large branches of the terrain tree structure.

CLIPMAPS. One refinement technique that deserves special attention is the use of what Losasso and Hoppe term Geometry Clipmaps (12). In this technique nested

grids of discrete levels of detail are placed around the user's position. With the finest level of detail placed directly below the user's viewpoint progressively coarser levels of detail grids are placed as the viewing distance increases. This algorithm was further refined in (9) to take advantage of recent GPU improvements in the form of programmable shaders. In this method the terrain data is sent to the GPU as a set of textures and is then used to offset a static set of vertices in the vertex shader.



2.3.5. CRACKS & T-JUNCTIONS

A point of major concern for all algorithms in level of detail calculation is resolving cracks that can form between areas of different levels of detail. Quad-tree algorithms such as (13) must take special care to avoid cracks between neighboring quads as illustrate in FIGURE 7. The extra vertices along the edges of neighboring quads can cause T-Junctions which are subject to round-off errors causing jittering effects. This can occur even where the extra vertex happens to fall on the line of the neighboring quad. In the worst case a crack appears causing background colors to bleed through.

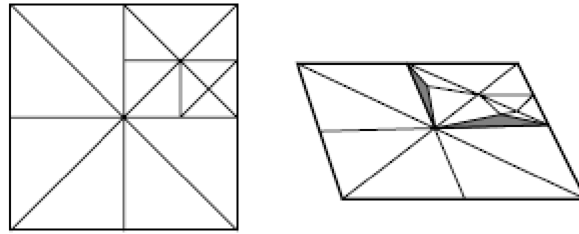


FIGURE 7: Cracks formed in a quad-tree with varying levels of detail between neighbors.

A rather simple solution to the problem is to avoid simplification around the edges of quad blocks (2). This creates an unnecessarily large number of triangles around the edges of the quad blocks. A more widely used solution is to gradually blend vertices from one level of detail to the next (14). This requires a significant amount of work at runtime to determine the level of detail of neighboring quads. Hoppe (9), while not using a quad approach, uses this blending approach in his algorithm, but offloads the blending to the GPU thus saving CPU cycles from calculating this information

Binary trees suffer from the same issues as quad-trees with respect to cracks. The solution for binary trees is significantly easier though and is generally the reason for using such a data structure. By recursively traversing the tree each neighbor is tested to ensure it is at most one level of detail away from the current node (6). If it is not then it is visited and split. If it is more than one level away then its neighbors will have to be visited recursively until the condition of one level difference is satisfied. FIGURE 8 shows how a triangle to be split (top left in blue) checks its neighbors and recursively refines as necessary. Note this procedure can

have wide ripple effects throughout the surrounding mesh even though it did not meet the given error threshold.

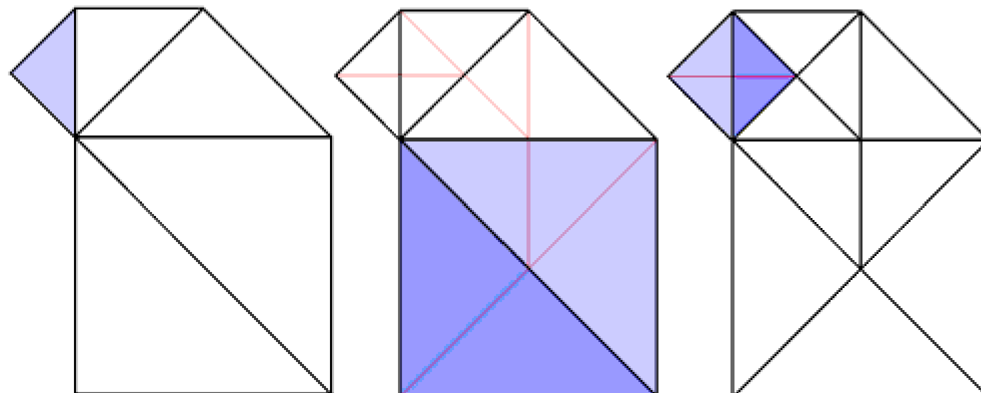


FIGURE 8: Resolving T-Junctions in binary trees through recursive splits. (6)

An interesting and efficient solution to the resolving varying levels of detail between neighboring geometry is to use a tile approach (15). Using variations of a tile where the edges of the tile either fits to a tile with equal LOD or steps down to the next LOD. Since vertex data is static only the index information needs to change which can be pre-computed. This approach obviously limits neighboring tiles to be only one level away from its neighbor, but a restriction easily adhered to.

2.4. GEOCOORDINATES

Over the years many standards have been introduced for estimating the dimensions of the earth. The problem with most of these calculations is they are based on a perfect ellipsoidal earth. When considering the dimensions of the earth with respect to sea-level one finds there are slight perturbations due to varying densities of the earth's core. The solution provided in the WGS-84 standard

addresses these issues through what it calls spheroids. Starting with an ellipsoid that is fatter around the equator and shorter at the poles, the WGS-84 defines the base sea-level for the planet as a whole. It further refines this measurement through a set of spheroids which define variations in sea-level measured in an offset from the base ellipsoid. These variations measure in 10's of meters and therefore, depending on the accuracy one is looking for, may not lend much to the overall precision and in most terrain algorithms is ignored.

GeoCoordinates are measured in degrees which do not work well for defining positions of vertices for a graphics rendering system. We must convert these coordinates into XYZ triples for use in vertex data. This can be accomplished by treating the center of the earth as the origin in a format called Earth Centered Earth Fixed. With the center of mass as the origin the z-axis points towards the north pole leaving the XY plane aligned with the equator. Now the X axis is placed at the prime meridian line put the Y axis 90° to the east of the x-axis.

CHAPTER 3: GPU OPTIMIZED MESH

Before programmable GPU's came along all data to render a scene had to be passed to the GPU every frame. Optimizations in the form of display lists were introduced, but this still required the compiled list to be sent across the pipe to the GPU each frame. It was up to the GPU to determine what to cache and for how long. When working with large datasets, such as rendering the earth, the amount of time pushing data across to the GPU becomes the bottleneck. With the introduction of Vertex Buffer Object the programmer has more control over the data that is cached on the GPU and how long it should stay in memory. This is especially helpful for terrain rendering since there is a large amount of vertex information to render which changes little from frame to frame. Once the terrain data has been pushed to the GPU there is no need to change this information which allows us to tell the GPU it is not volatile, thus improving cache performance.

In this chapter we introduce our approach to solving the problems facing terrain rendering as detailed in the previous chapter. We implement a discrete level of detail algorithm which departs from recent research in terrain rendering. To resolve past issues with this method we must introduce improvements that overcome these issues. These techniques utilize improvements to the graphics pipeline in the form of vertex buffer objects, updatable cached GPU memory as well as data and triangle stripping optimizations.

3.1. VERTEX BUFFER OBJECTS

Introduced in the OpenGL 1.5 specification, Vertex Buffer Objects allow the programmer to allocate a region of memory for use by the program and specify how it will be accessed. Allocating a region of memory tells the GPU to prepare a block of GPU memory for use and return a handle so the program can access it. Specifying a usage flag tells the GPU how the block of data will be used which lets the GPU optimize how that block of memory is cached. Access flags also assist in this decision by telling the GPU the program will only read from the memory or if it will need write access to it as well.

An important feature of Vertex Buffer Objects for our purposes is the ability to modify a block of memory preallocated on the GPU. By using glMapBuffer calls we can update a region of the memory or all of an allocated block. When updating a portion of the allocated memory, using glBufferSubData offers even better performance, allowing access to a subset of the allocated memory. Section 3.3 goes into further detail on how Vertex Buffer Objects can be used to optimize terrain rendering.

3.2. PROGRAMMABLE SHADERS

There are occasions where we might need to modify the positions of vertices slightly after it has been sent to the GPU. Constantly updating these values would negate any benefit we hope to gain by using Vertex Buffer Objects. Programmable shaders allow us to make changes to a vertex or fragment without altering the

vertex information stored in GPU memory. Shaders allow the programmer to pass in data that can be used in calculating new positions of vertices. These variables can be sent to all vertices or individual vertices, depending on the needs of the shader program. This saves us from having to resend large chunks of coordinates for changed vertices.

Programmable shaders are also a great place to get an extra boost in performance by offloading calculations that can be parallelized. Modern GPUs include multiple processors that can run shader programs in parallel. Section 3.6 goes into detail on how Shaders can be used to help speed up terrain rendering as well as overcome performance issues by updating vertex positions whenever a new level of detail comes into view.

3.3. PROGRESSIVE LEVELS OF DETAIL

Because we are using a discrete LOD method we must take care to organize our data so that it can be efficiently sent to the graphics processor. We accomplish this by progressively sending vertex data as it is needed. In order to do this we must utilize modern GPU's vertex buffer objects. To take advantage of Vertex Buffer Object's feature that allows the calling program to update blocks of vertex data in GPU memory, we must devise an algorithm that sends vertices to the GPU progressively between frames and not all at once in one frame. We can do this by using LOD techniques where only a subset of the vertex data is needed to render objects further away and progressively adding more vertices as they come closer to the viewer. We can accomplish this by ordering the vertex data in such a way that

the vertices are ordered by level of detail and each level of detail makes use of the vertex data from the previous level.

FIGURE 9 shows how the vertices are laid out in memory from one level of detail to the next. Notice that the first level of detail is a subset of the second level of detail. This layout will allow us to progressively send more vertices to the GPU as they are needed. This means that to render the second level of detail we only send the vertices not already on the GPU which were sent for the first level of detail. The only thing that must change is the index information for creating triangles for the vertices. The following section goes into detail on how we can traverse the mesh and reorder the vertices in this manner.

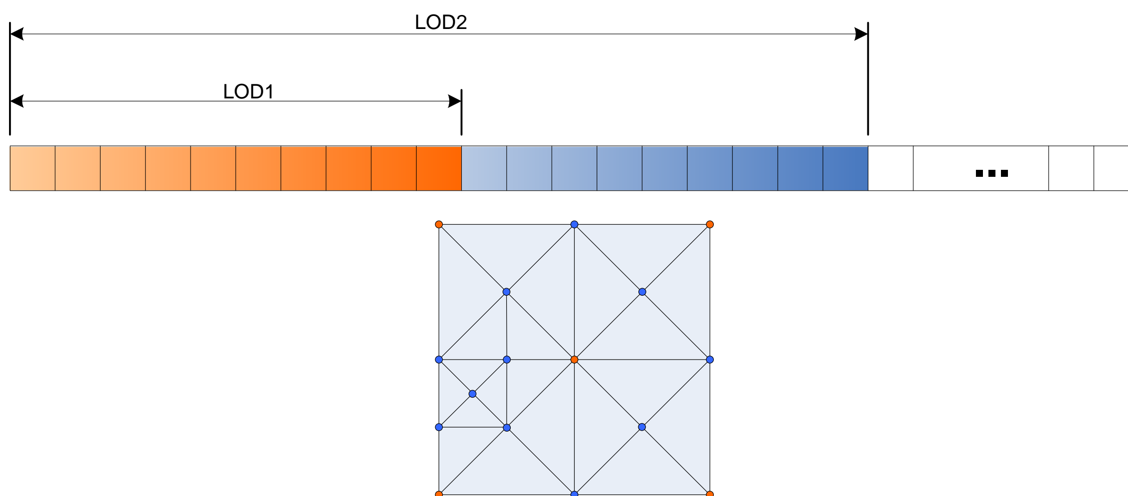


FIGURE 9: The second level of detail uses all the vertex information from the first plus the extra vertices required for the new detail.

Once we have the vertex data ordered in a way that can be progressively sent to the GPU we can then make use of the Vertex Buffer Objects ability to update blocks

of memory on the GPU. By using OpenGL's glBufferSubData call we can map to a section of memory on the GPU and update the block of memory already allocated. This allows us to pass in a subset of vertices for coarse levels of detail and later pass in the vertices for finer levels of detail if or when they are needed.

At the first level of detail we allocate enough room on the GPU for the entire size of the array of vertices. This does not actually send any data and therefore does not use any bandwidth to the GPU. This call tells the GPU to reserve a section of memory for use later. Once the memory has been allocated vertices can be sent to specified locations within that buffer. When a block of terrain first comes into view we will usually only have to send a handful of vertices since the block will be a coarse representation. When the viewing position gets closer, and more detail is required, we only send the subset of vertices not already in the first level of detail. As the block goes back out of view, the memory that was allocated on the GPU is released. If the viewer never gets close enough to trigger the highest level of detail, then those vertices are never sent saving much needed bandwidth.

When rendering finer levels of detail this method greatly improves the bandwidth used to send vertices. Using a block of terrain with 257x257 set of vertices at its finest level we would be sending 65k bytes of data. With progressively sending vertices in this manner, by the time we need the finest level of detail we could potentially have already sent in half of the data, depending on the error thresholds used. See section 5.2 for details on throughput savings by using progressive meshes method.

According to the NVIDIA white paper on using Vertex Buffer Objects (7), their GeForce GPU can take advantage of indices that fit within a 16 bit integer instead of a 32 bit integer. This is a 2x savings that could greatly speed up passing index information to the GPU. Unfortunately due to how our algorithm requires a grid dimension of 2^n+1 this does not allow our index information to fit nicely into a 16 bit integer. Using a dimension of 257 (2^8+1) this gives us an index size of 66,049 which just misses the 65,536 range of a 16 bit integer. Stepping down to a dimension of 129 (2^7+1) we only need a range of 16,641 requiring only 15 bits of precision.

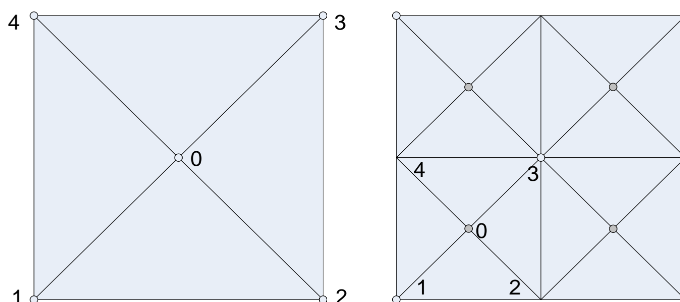


FIGURE 10: Recursively traversing the quad-tree in SW, NW, NE, SE causes reordering of indices from one level to the next

While progressive meshing in this manner saves bandwidth for sending vertices, it does not allow the same savings with indices. FIGURE 10 illustrates how each level of detail requires its own indexing. As illustrated the second level of detail completely reorders the index information in such a way that makes it impossible to add index information as we did with the vertices. Since an index is represented as a single 4byte integer, the amount of bandwidth required is still considerably less than sending 3*4bytes of data for each vertex every frame. It is possible to cache one

level of index information just as we do with vertices, so if the level of detail threshold never changes no index information is required to be sent after the initial transfer.

3.4. REFINEMENT ALGORITHM DETAILS

The algorithm to order the vertices as described in the previous section uses a quad-tree representation of the height field where the top node of the tree is the vertex at the center of the mesh. Dividing the data in this manner requires the dimensions of the array be $2^{n-1} + 1$ where n is the number of levels in the quad-tree. So for a 9 level quad-tree we can cover a 257x257 array of terrain data. To quickly access the nodes in the quad-tree we can use a point's offset into the array and calculate its children by finding the distance to each child's index. The South West and North East children will always be $\frac{n-1}{2^{lvl}}$ points away from the parent where *n* is the number of vertices in the array, and *lvl* is the current level of the child in the tree in relation to the root node. The South East and North East child can be obtained from an offset of $\frac{width-1}{2^{lvl}}$ from the SW and NE nodes where *width* is the width of the overall array size (257 in our example).

Using these offsets we can quickly navigate the quad-tree to determine which vertices need to be decimated. We use a top down method similar to (8) where we navigate from the root node and traverse down the tree. As we traverse the tree we check each edge node of a quad to see which vertices need to be activated. We keep an active node array and increment the node index in that array. As we pop back up

the tree we keep track of the depth of the lowest active node in that branch. This depth value will help us navigate the active node to determine which nodes are leaf nodes and which are parent nodes. This will be used in constructing our triangle fans discussed later in this chapter.

ELIMINATING T-JUNCTIONS. As discussed in the previous chapter, T-Junctions are the points in a mesh where two neighboring levels of detail contain different active vertices along their shared edge. This causes a crack to appear where the extra detail is rendered with the additional vertex. Care must be taken to avoid such situations when using a Top-Down approach such as ours.

While we use a similar algorithm as (8) for navigating quad-trees, we differ in how we resolve T-Junctions. When resolving cracks within a block their solution is to revisit neighboring areas and progressively decrease the detail for each neighboring block. FIGURE 11a illustrates this refinement by showing the lines that are required to mend cracks that will occur due to the differing levels of refinement between two different quads. The darkened area is a child of the north west quad that has been refined, while its neighbor has not. The dotted lines show how the algorithm in (8) resolves this by further refining the neighbors. Each neighbor is refined to one higher level than the one before it until further refinement is no longer necessary. Notice this refinement can cross many boundaries. The addition of a single point in the NW quadrant has a ripple effect that goes all the way to the SE quadrant.

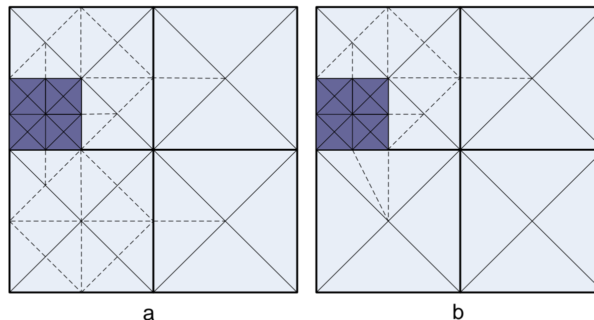


FIGURE 11: a) Triangle fans with recursive refinement and b) restricted refinement

Our approach is to limit the refinement to just the neighbors of the block with higher detail. This is accomplished by adding points to the triangle fan already in the active array around the quads edges. Whenever we find a node in the active array with a depth of 2 we know we have a parent node with just one more level of leaf nodes. At this point we can use the parent node as the center point of the triangle fan and sweep the edge of the quad in a counter-clockwise direction, adding any active nodes we find.

As shown in FIGURE 11b the number of additional triangles is greatly reduced over the algorithm used in (8), even for this simple example. This also simplifies the refinement algorithm in that only immediate neighbors are affected by refinement so there is no complex recursive refinement in contrast to (6) and (8).

TRIANGLE FANS. Schneider and Westermann (8) point out that the use of triangle fans increases cache hits if rendered using a Π-Order space filling curve. They claim a worst case cache hit of 25% using a fan size of 5, thus increasing the fill rate on the GPU. Using additional points along each fan, as we do, can only increase the caching potential of our algorithm.

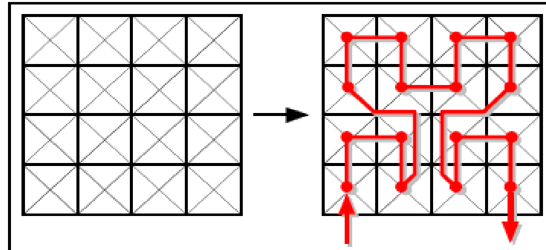


FIGURE 12: Π-Order space filling curve (8)

However, using triangle fans implies repeated calls to glDrawElements for each fan. To render a triangle fan one must make a call to glDrawElements specifying the vertex/index information as well as the triangle fan size. The overhead of making a call for potentially 100's of thousands of fans can quickly degrade performance. A recent extension to the OpenGL specification from NVIDIA helps eliminate the problem of repeated glDrawElement calls. Through the use of the primitive restart extension (16), an index can be specified that will be used as a restart indicator for the GPU. By specifying a restart index of zero, all one needs to do is place a index of zero at the end of each fan in the index array. Now all the triangle fans in the index array can be rendered in one call to glDrawElements with TRIANGLE_FAN as the element type and an array of all indices including the restart index throughout.

3.5. DATA STORAGE

When working with such large amounts of data it is important to pick a data storage scheme that allows for quick retrieval of data. As terrain blocks come into view their data needs to be loaded into memory and prepared for viewing. File IO is notoriously slow due to relatively slow access times to hard disks. To help alleviate these slower access times File IO is usually performed in a separate thread so as to

not slow down rendering of each frame. As the data becomes available it is then sent to the GPU for rendering. This does cause some lag but should be unnoticeable if the data for a block is retrieved before it is needed. This has been handled before (17) by predictive fetching based on where the user is expected to go.

It is also wise for us to keep file sizes as small as possible. With terrain data being very regular we can exploit this by only storing height values for each sample point. However, because we have reordered the vertices to progressively send to the GPU, their position in the data structure no longer correlates to their coordinates in 3D space. We are therefore required to store an extra 4 Bytes representing the vertex's offset into the grid (a smaller index could be used depending on the size of the mesh used). Upon loading the data for rendering, determining the X-Y coordinates is a simple translation of the index based on the dimensions of the grid.

To further alleviate file IO times we also compress the data using standard compression algorithms. The idea is that the time it takes to load a 1M file is much longer than the time it takes to load a .5M file and uncompress it. The trick is to find the compression ratio and file chunk size that optimizes this savings. If the uncompressed file is too small, then little gains will be made from compressing the data and will take longer to load and uncompress than simply loading the original uncompressed file.

3.6. PREPROCESSED GEOMORPHS

Geomorphs are a great way to eliminate popping from one level of detail to another. The problem comes in determining where a vertex was in a previous frame. Finding this point requires knowing which vertices were rendered in a previous LOD and where the new vertex was within that LOD. The red dot in FIGURE 13 shows a vertex being added by the next finer level of detail. The right image shows the new level of detail and the vertex in question. The image on the left shows the previous level of detail and the point where the new vertex would have been had it been rendered. In order to find this point on the previous level of detail we must first find the triangle it was contained within.

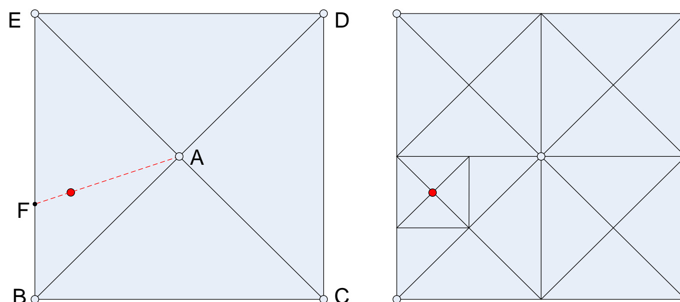


FIGURE 13: Coarse LOD (left) showing a point being added to fine LOD (right).

Previous attempts (8) at finding this information put most of the work on the GPU. The problem with this approach is the amount of data needed to perform this calculation as well as the time spent calculating the previous position. Schneider and Westermann (8) came up with a novel approach to eliminate any branching normally required to determine this position by using height values as control points in a piecewise linear interpolation. There is no data presented on the amount

of processing time required for this approach, but it is safe to assume it is not insignificant.

The method presented here takes a much more simplistic approach and offloads the processing of the morph values to the pre-processing phase. Since we are already pre-processing the terrain blocks for level of detail, we can also process the positions of each vertex from one level of detail to the next. (9) uses a similar technique of determining the morph points before sending the vertex data to the GPU. They conclude that this saves bandwidth required to send the previous vertex points as well as the extra lookups required to find those locations in the texture. Our situation is complicated by the fact that our vertex array is not stored in an order that would easily translate into X-Y lookups and therefore requires this information to be passed along with the height values.

Using the example in FIGURE 13 we can see that the finer level of detail contains 13 additional points that we will need to store transitions for. This method requires an additional height value (4 Bytes) stored for each vertex. In a 257x257 grid this accounts for at most an additional 264K bytes of data. This additional disk space is a tradeoff between the amount of processing time on the GPU for evaluating previous positions and the time to load the extra data.

The algorithm to generate these previous height values is fairly straightforward when tied to the existing algorithm for calculating each level of detail. When a vertex is activated for the first time, the previous height value is calculated since it was obviously not in the vertex array in previous LODs. To find this previous value we

need to find the triangle it was in on the previous LOD. This triangle consists of the previous parent node and its two leaf nodes on either side of the intersection point along the previous quad. This can be seen in FIGURE 13 by ΔAEB with the intersection point F.

Determining the previous parent is a matter of accessing the previous LOD's active node matrix and looking for the parent node with a depth value of 2, meaning it was an active node with leaf nodes active. From this previous parent node we calculate the line through the new point and find where it intersects with the boundary of the parent's quad edges. Finding the surrounding leaf nodes is a matter of searching in either direction of the intersection point F. Finally, to find the new point within the triangle we simply interpolate between the three points. There is a special case where the point F lies on a line along a previously rendered line. In this case we only have to interpolate along a single line.

CHAPTER 4: GLOBAL TERRAIN

We use the WGS-84 as described in chapter 2 for calculating global coordinates. However we do not consider the variations in sea level based on spheroids. It is not our intention to render the globe to this accuracy as it will lend little to the visual perception of the terrain. If we were concerned with calculations based on sea level then spheroids would have to be considered. This can easily be incorporated at no extra cost to runtime processing as it can be handled in the preprocessing of the data. Since our major concern is visual fidelity and not actualities, we also make generalities when placing individual Terrain Blocks and therefore the vertices within them. The globe is divided into 10 degree increments along the latitude and longitude giving us a 36 x 18 grid of Spheroid Blocks. The Spheroid Block coordinates are converted to ECEF coordinates and stored as the boundaries for the block.

A spheroid block is made up of many terrain blocks all aligned and translated to the spheroid blocks coordinate system. Based on the resolution one wants to achieve the number of terrain blocks mapped to a spheroid block can change. To achieve 30 meter accuracy (1 arc second) we would need to divide each spheroid block into 120 terrain blocks. Once terrain blocks are assigned a position within a spheroid block their local coordinate systems are translated and rotated to align with the plane of the spheroid block. This aligns the terrain block in a way that puts its Y value perpendicular to the surface of the sphere. This greatly simplifies matters in that we do not have to convert each value to ECEF coordinates.

Using the spheroid block as the basis for our terrain blocks has the added advantage of allowing more accuracy in the height detail of each block. If we were to translate each terrain block based on its ECEF coordinate this would leave little room in a 32bit float for rendering detail in the terrain height. Care must be taken though when working with such large floating point numbers. Precision errors crop up very easily through something as innocuous as a matrix multiply. To minimize floating point precision issues the rotation matrix for the spheroid block is calculated based on its Right, Up and LookAt vectors and placed in a matrix where it is later multiplied directly against the model-view matrix.

To further complicate matters we also run into precision errors when working with OpenGL's viewing frustum. Using the WGS-84 measurements for the major axis of the earth we find the diameter to be over 12,000 kilometers. This makes it difficult to set a viewing frustum that works when viewing the entire planet as well as viewing ground detail.

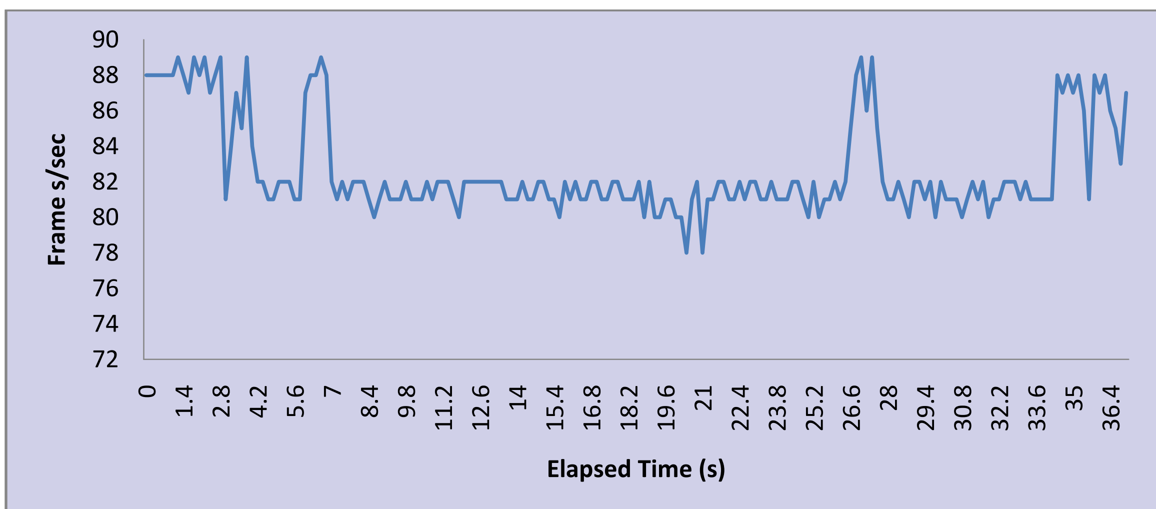
To fix this problem one cannot simply scale everything to fit within a global viewing frustum. This would remove precision when viewing ground detail. To solve the problem one must employ a sliding scale. When viewing the entire planet everything can be scaled down to fit within a smaller range. This requires the vertex information as well as the viewing planes to be scaled.

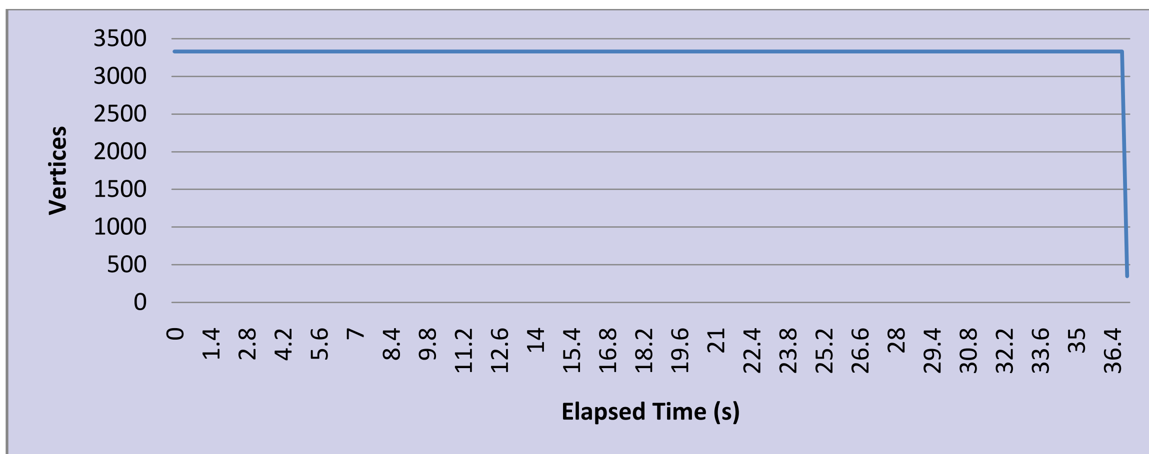
CHAPTER 5: ANALYSIS

The following is an analysis of data taken from the system with various algorithms. In each section we show the frames per second followed by the vertex count for that time. These measurements were taken with a debug version of the nvidia drivers and therefore introduce a minor amount of overhead. While minimal we are mostly concerned with relative performance between the different algorithms.

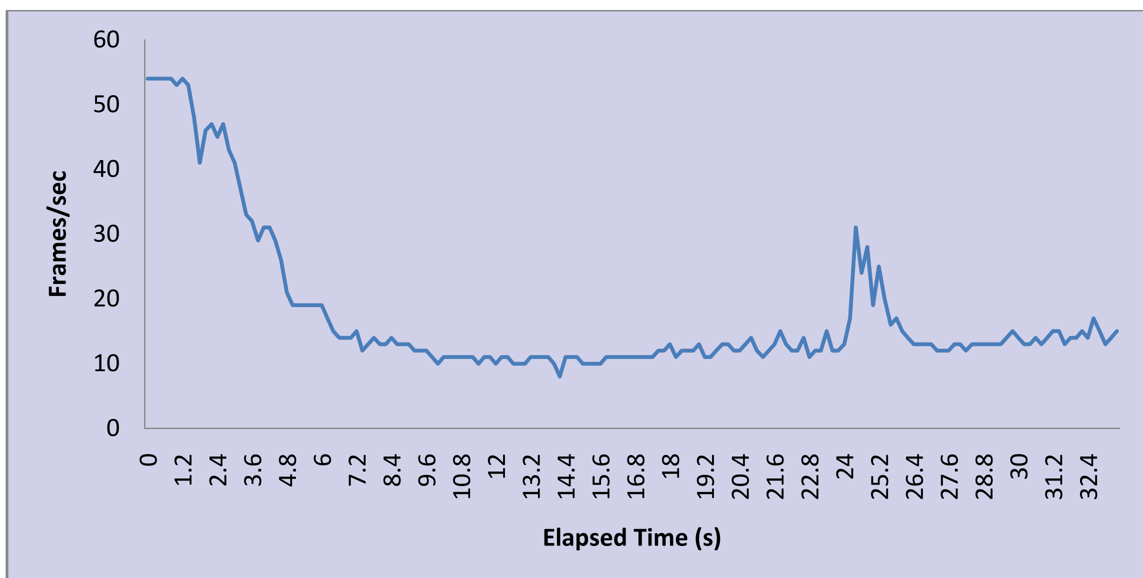
5.1. BASELINE PERFORMANCE

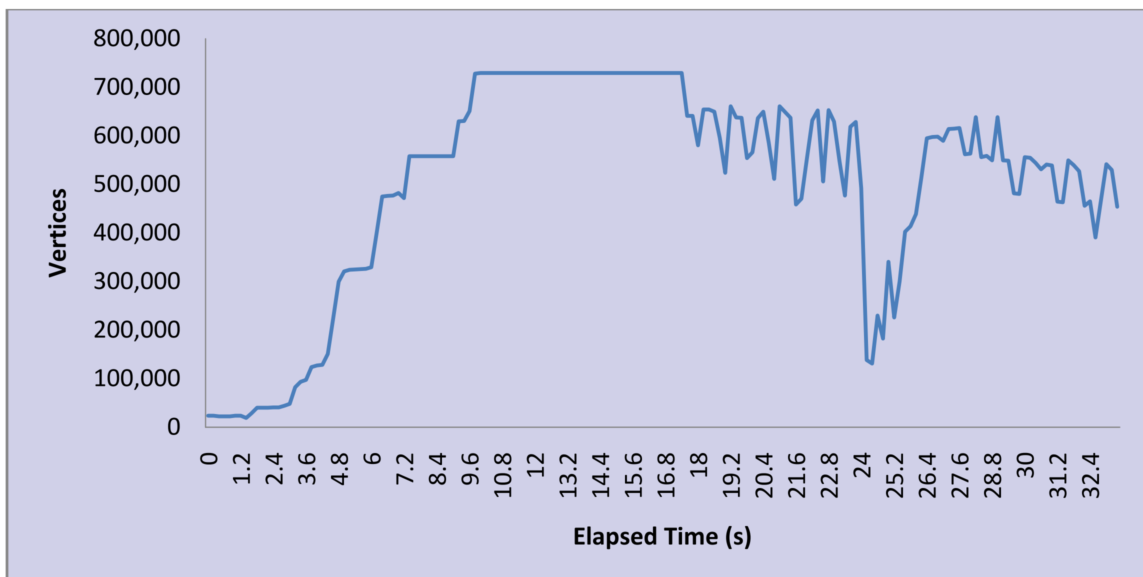
To get a baseline of how the system performs, we ran through the environment at the highest level so no detail was shown other than the overall globe. The following baseline graphs show the system can sustain 80 frames per second with the minimum number of vertices to render a 10 deg lat/lon separation consisting of approximately 3300 vertices.





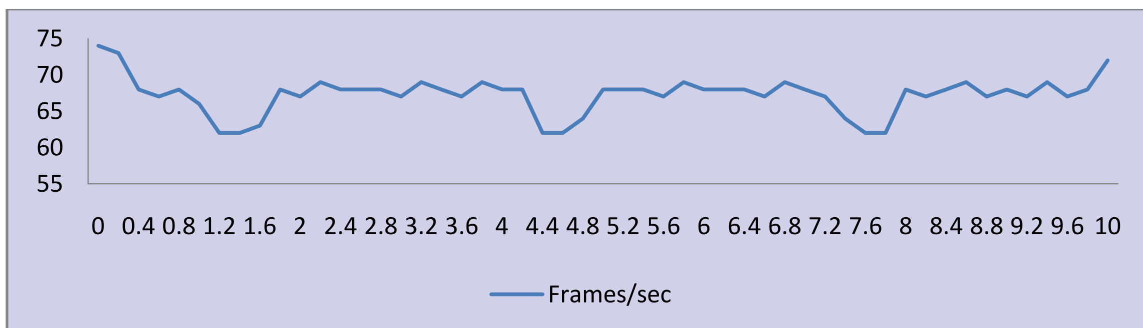
For the first comparison individual triangle fans were used. Each fan was rendered in a separate call to glDrawElements. On average a fan consists of approximately 8 vertices. As the graph below show using this method can barely sustain real-time rates with a vertex count of 700k vertices. The occasional dips and peaks are areas of transition from one block on the sphere to the next. This is when the detail of the previous block has been culled by the view frustum and the next block has not come into view yet.

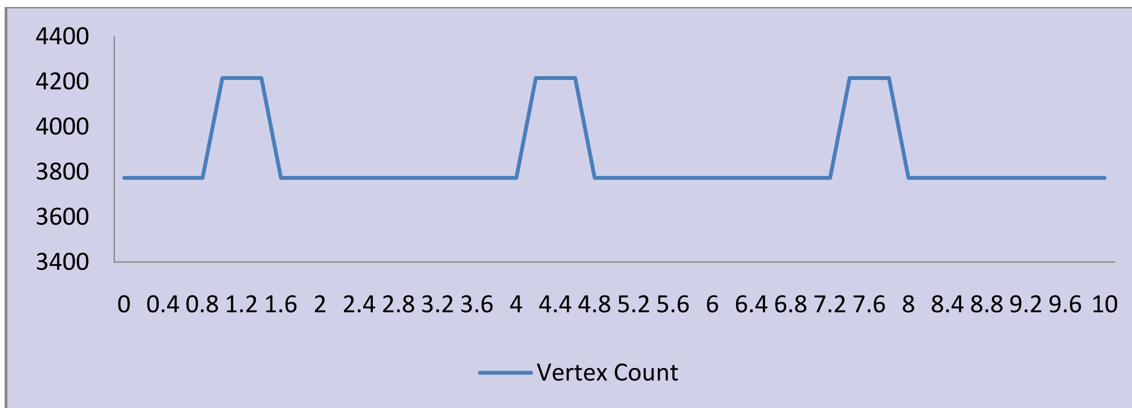


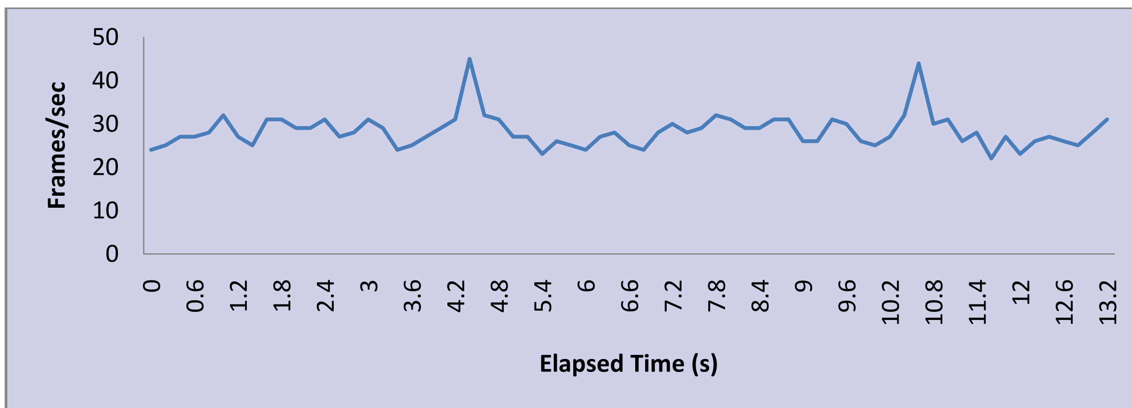


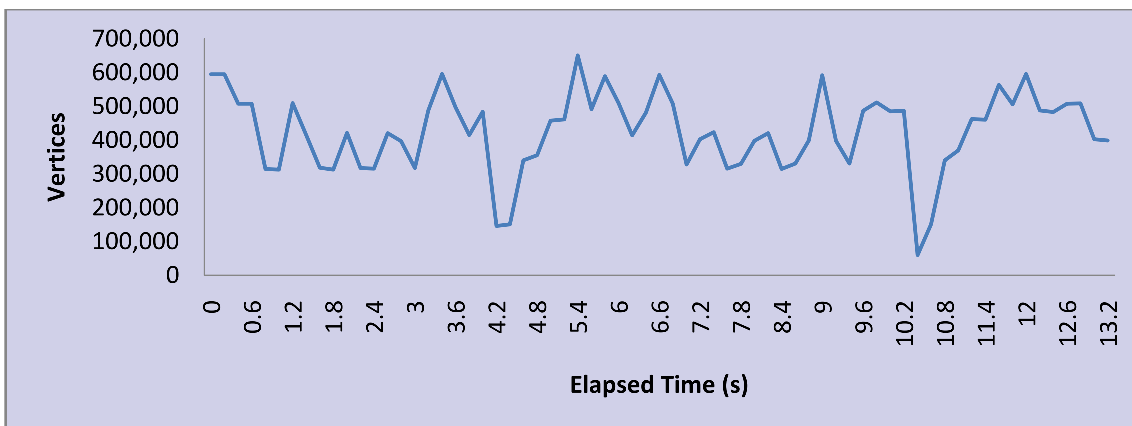
5.2. PROGRESSIVE VERTICES

With a baseline for how the system performs using traditional rendering methods, we now look at how it performs using GPU optimizations. The first comparison is with vertex data cached on the GPU in one shot when it is loaded from the file system. The data in this sample is not progressively sent, but completely loaded as it becomes available. As can be seen the average frames per second is well above that of repeated calls to glDrawElements.



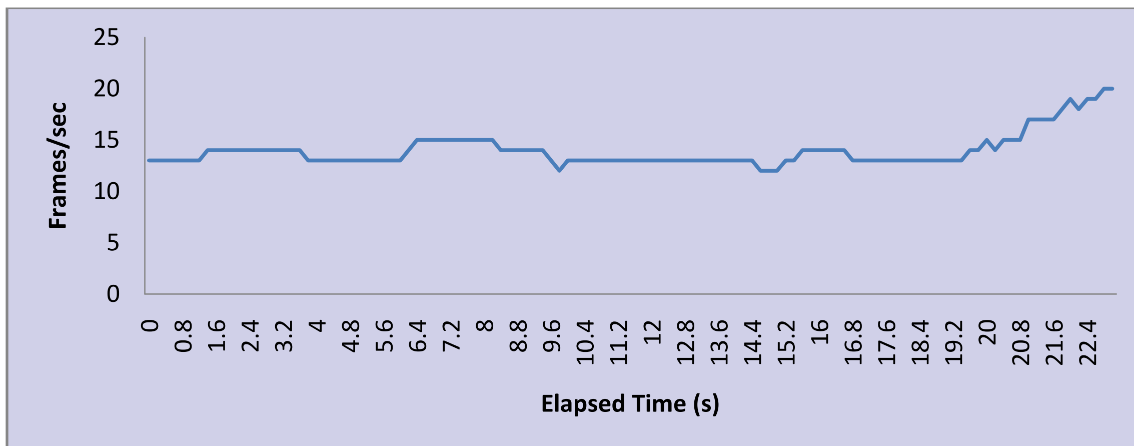


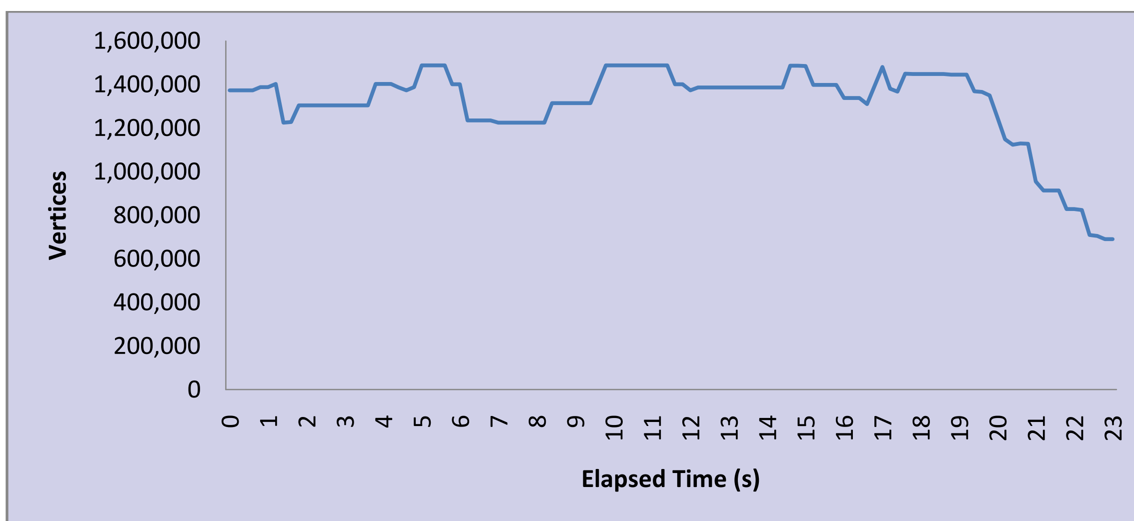




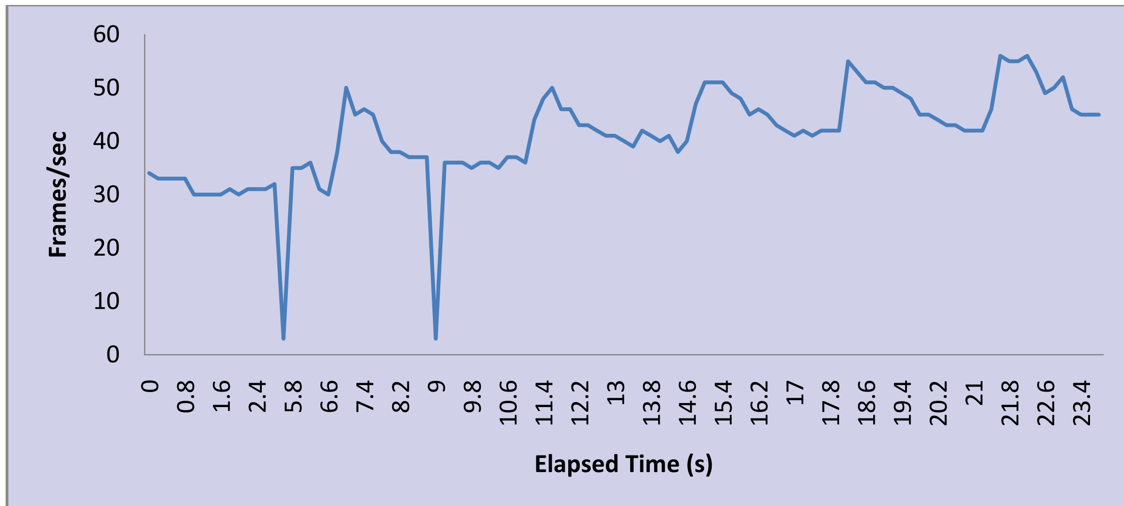
In an attempt to see how far we could push the system the draw distances were increased to bring more vertices into the scene by rendering higher detail for blocks further away. Using VBO's nearly doubles the performance of the previous sections

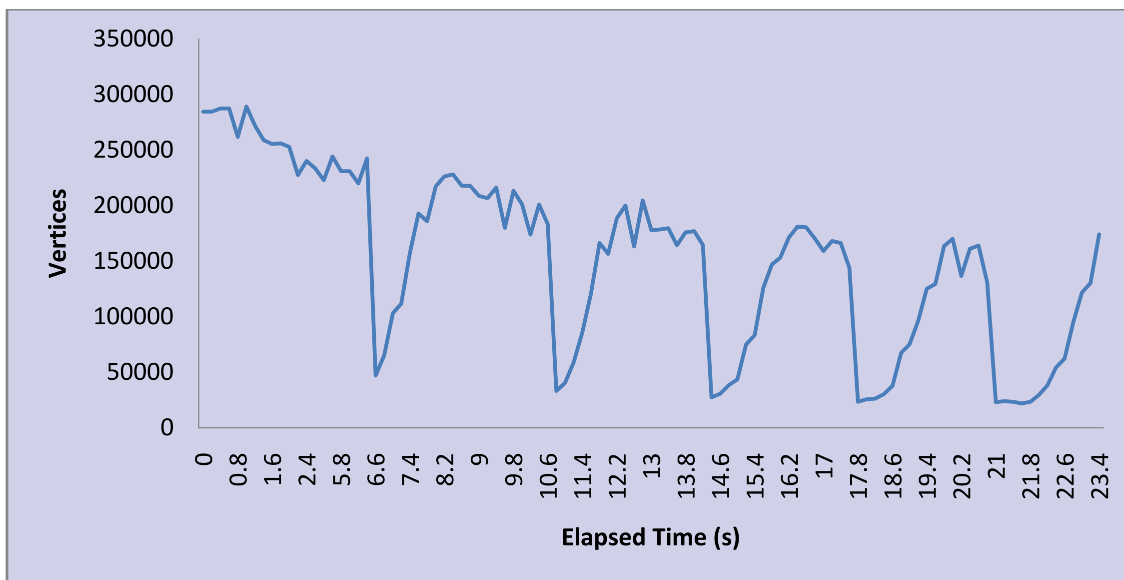
repeated calls to glDrawElements. Notice too the frame rate is much steadier. We attribute this to caching of data on the GPU through the use of VBO's.



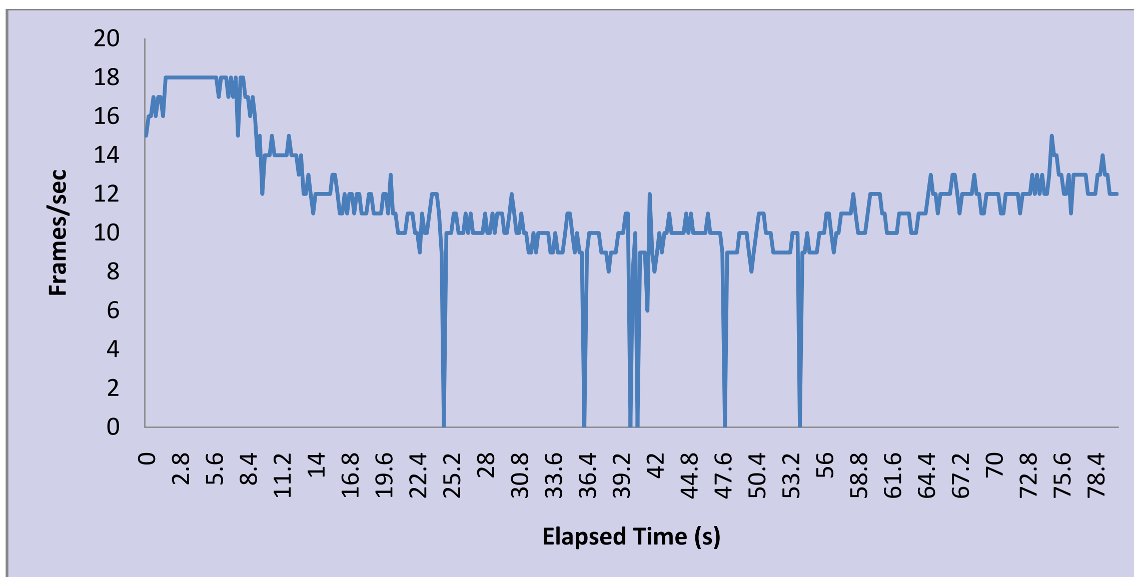


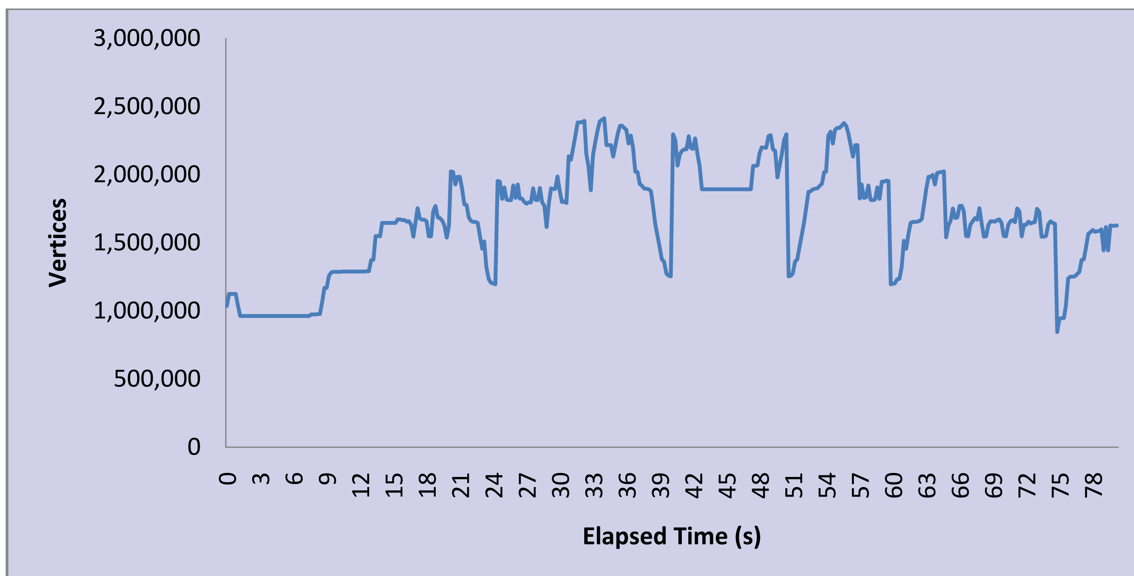
Finally we show how the system performs using VBO's and sending vertex data progressively to the GPU as higher detail is needed for a particular block of terrain. The large valleys are areas where we changed altitude to load a fresh block of data. Clearing out the vertex cache shows how the system can stream the data back in to the GPU with a minimal amount of a performance hit.





The final graphs show the system using progressive mesh data under extreme load. We can maintain around 10 frames per second with approximately 2 million vertices. Note the graphs shown use a fairly quick movement through the environment and therefore have a high rate of introducing new vertices. This is the reason for the erratic nature of the graphs. In a scenario where the user is walking through the environment, frame rates will be steadier since the vertex information is cached and is not being resent to the GPU.


Frames/sec
20
18
16
14
12
10
8
6
4
2
0
0
2.8
5.6
8.4
11.2
14
16.8
19.6
22.4
25.2
28
30.8
33.6
36.4
39.2
42
44.8
47.6
50.4
53.2
56
58.8
61.6
64.4
67.2
70
72.8
75.6
78.4
Elapsed Time (s)


Vertices
3,000,000
2,500,000
2,000,000
1,500,000
1,000,000
500,000
0
0
3
6
9
12
15
18
21
24
27
30
33
36
39
42
45
48
51
54
57
60
63
66
69
72
75
78
Elapsed Time (s)

CHAPTER 6: CONCLUSIONS

The graphs in the analysis section clearly show improvement of using a GPU based technique over immediate mode type rendering and even calls to glDrawElements for small fan arrays. The differences between progressive meshes and vertex arrays however are not as great. This can be attributed to the choice in sizes of each level of detail. Where the vertex arrays send all the vertices for a given level of detail, the progressive VBO's only send what wasn't already sent in the previous level of detail. The savings is greatly dependent on the difference in number of vertices from one level to the next.

It should also be noted that the method for telling the GPU how the vertex data is going to be used is only a suggestion to the GPU. This means that if there is insufficient room for incoming data, some of the allocated memory for a VBO could be used temporarily. This would require the vertex data to be sent to the GPU again. Vertex arrays can also take advantage of the same caching, it is just not user specified. This is another reason for the subtle difference in performance between the two methods.

The GPU was once the bottleneck for rendering large million vertex mesh structures. We have presented evidence that the GPU is capable of handling large amounts of data and maintain interactive rates. Through intelligent processing of terrain data, we can organize it in a manner that optimizes the graphics pipeline for better throughput and caching. By offloading CPU cycles to a preprocessing step we are able to free the CPU for other processing tasks. This increases that amount of

data required to render accurate terrains, but as we have shown, this increase is manageable and capable of interactive rates.

CHAPTER 7: FUTURE WORK

Due to time constraints several features were not implemented that would make a more complete rendering system for viewing global terrain. The following is a list of the more important features to be addressed.

7.1. CRACK MENDING

While considerable effort was taken to avoid T-Junctions and cracks between varying levels of detail within blocks of terrain, there remain cracks between spheroid blocks. The intended solution is to mend these cracks with extra triangle fans between these blocks. These areas can be minimized by passing edge information to border blocks where vertices can be blended in the vertex shader in a similar fashion used with GPU Clipmaps (9).

7.2. BETTER ERROR METRICS

The focus of this thesis was on enhancements of data structures and algorithms for use with programmable GPU hardware therefore a simple error metric was used to create the discrete levels of detail. This error metric should take into consideration some form of standard deviation across a block of terrain to come up with the error thresholds used to reject vertices at each level.

7.3. CLIFFS AND OVERHANGS

One of the initial goals of this work was to incorporate a method for allowing easy manipulation of terrain data for generating cliffs, overhangs and caves.

Typically terrain data comes in the form of a heightmap which specifies a single height for every point. By using layers of height maps one could create overhangs. If two height maps were to trace a curved spline marking the center, a simplistic tunnel could be generated with one height map representing the floor and the second the ceiling. Intersecting splines could be used to create a network of tunnels.

BIBLIOGRAPHY

1. GeoTIFF Specification. *RemoteSensing.org.* [Online] 2000. http://www.remotesensing.org/geotiff/spec/geotiffhome.html.

2. *Smooth view-dependent level-of-detail control and its application to terrain rendering.* **Hoppe, H.** 1998, IEEE Visualization, pp. 35-42.

3. *Accurate triangulations of deformed, intersecting surfaces.* **Herzen, B. and Barr, A.** 4, s.l. : SIGGRAPH, 1987, Vol. 21, pp. 103-110.

4. *Real-time, continuous level of detail rendering of height fields.* **Lindstrom, P., et al.** 1996, ACM SIGGRAPH, pp. 109-118.

5. *Large scale terrain visualization using the restricted quadtree triangulation.* **Pajarola, R.** 1998, IEEE Visualization, pp. 19-26.

6. *ROAMing Terrain: Real-time optimally adapting meshes.* **Duchaineau, M., et al.** 1997, IEEE Visualization, pp. 81-88.

7. **nvidia.** Using Vertex Buffer Objects. [Online] 10 16, 2003. http://developer.nvidia.com/object/using_VBOs.html.

8. *GPU-Friendly High-Quality Terrain Rendering.* **Schneider, J. and Westermann, R.** s.l. : Journal of WSCG, 2006, Vol. 14. ISSN 1213-6972.

9. **Asirvatham, A. and Hoppe, H.** Terrain rendering using GPU-based geometry clipmaps. [book auth.] M. Pharr and R. Fernando. *GPU Gems 2.* s.l. : Addison-Wesley, 2005.

10. *Terrain Rendering using Spherical Clipmaps.* **Clasen, M. and Hege, H.** 2006, EuroVis.

11. **Luebke, D and al, et.** Level of Detail for 3D Graphics. s.l. : Morgan Kaufmann, 2003.

12. *Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids.* **Losasso, F. and Hoppe, H.** s.l. : SIGGRAPH, 2004, pp. 769-776.

13. *Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail.* **Larsen, B. and Christensen, N.** 1, s.l. : Journal of WSCG, 2003, Vol. 11.

14. *Planet-sized batched dynamic adaptive meshes (P-BDAM).* **Cignoni, P., et al.** 2003, IEEE Visualization.

15. **Snook, Greg.** Simplified Terrain Using Interlocking Tiles. [book auth.] Mark DeLoura editor. *Game Programming Gems 2.* s.l. : Charles River Media, 2001, pp. 377-383.

16. **nvidia.** NV_primitive_restart. *OpenGL.org.* [Online] 2002. http://www.opengl.org/registry/specs/NV/primitive_restart.txt.

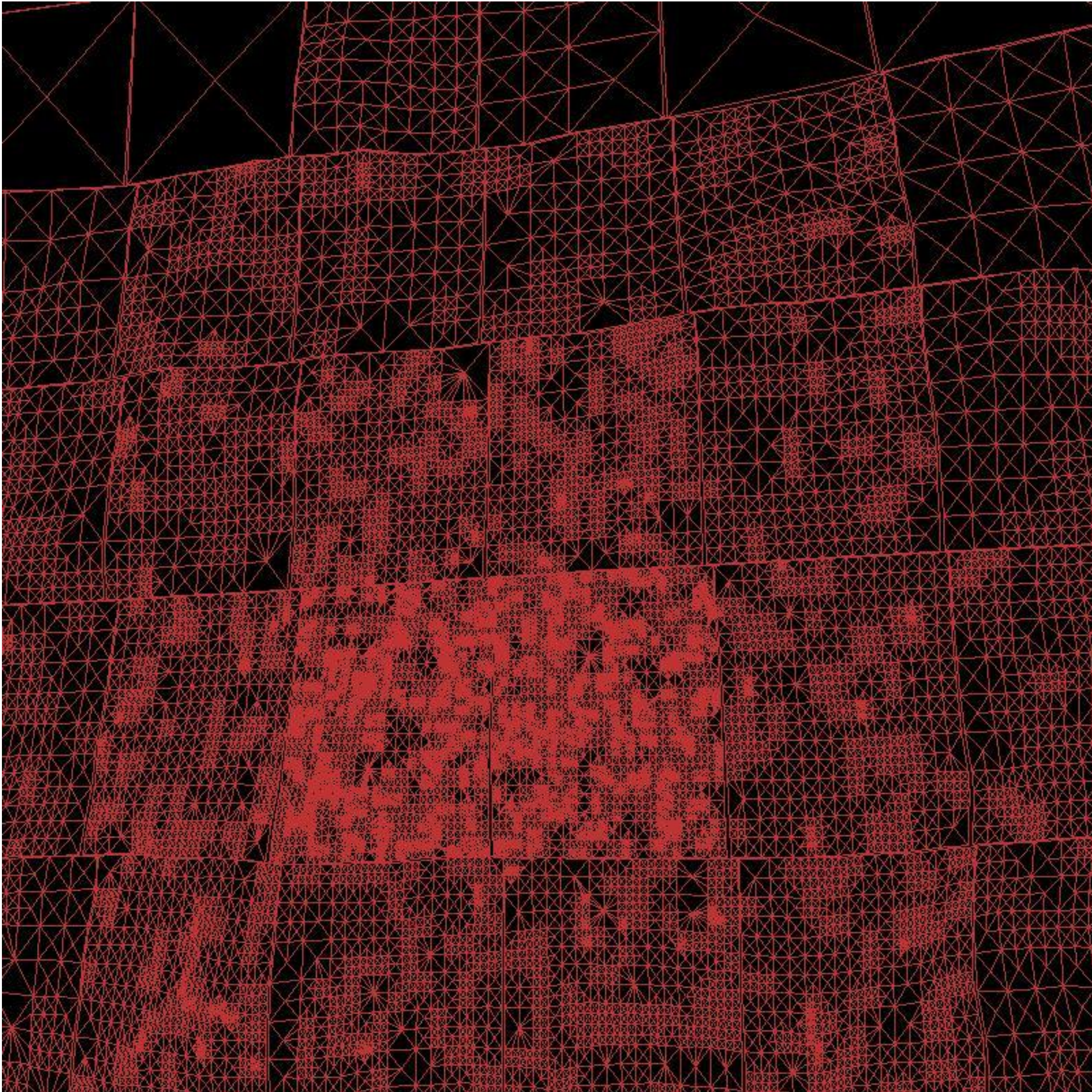
17. *Terrain simplification simplified: A general framework for view-dependent out-of-core visualization.* **Lindstrom, P. and Pascucci, B.** 2002, IEEE TVCG 8(3), pp. 239-254.

18. *The clipmap: A virtual mipmap.* **Tanner, C., Migdal, C. and Jones, M.** 1998, ACM SIGGRAPH, pp. 151-158.

19. *Quadtin: Quadtree based triangulated irregular networks.* **Pajarola, R., Antonijuan, M. and Lario, R.** 2002, IEEE Visualization, pp. 395-402.

20. *Real-Time Generation of Continuous Levels of Detail for Height Fields.* **Rottger, S., et al.** 98, WSCG, pp. 315-322.

APPENDIX A



A screen-shot of the application showing varying levels of detail