Terrain Rendering Using Geometry Clipmaps

November 10, 2005

Nick Brettell
Supervisor: Dr R. Mukundan

Abstract

A primary difficulty in terrain rendering is displaying realistic terrains to the user at real-time frame rates. The brute force approach is usually too complex for real-time frame rates to be achieved. Several terrain-rendering techniques have been proposed that use Level of Detail (LOD) to generate a simplified representation of a terrain. The geometry clipmap is a recently proposed approach that utilises the potential of modern graphics hardware. It stores vertex data on the graphics card, that is updated incrementally as the viewpoint moves. LOD is achieved using regular nested grids of increasing size and decreasing detail, centred around the viewpoint. We implemented the geometry clipmaps algorithm, and present aspects of our implementation, such as the maximum number of clipmap levels that are possible, and constraints on the extent of each level. We also performed a comparative analysis with other terrain-rendering techniques. Geomipmapping outperformed geometry clipmaps on a mid-range graphics card, but geometry clipmaps had the greatest rendering throughput on high-end graphics hardware.

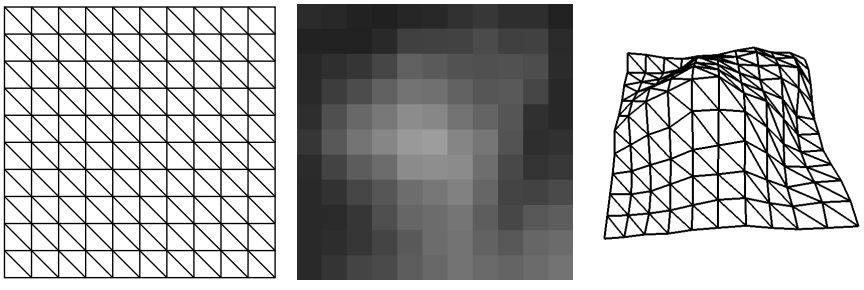
Contents

Chapter 1

Introduction

Terrain rendering is an area of computer graphics concerned with displaying terrains to the viewer at real-time frame rates. The terrains can be built from a heightmap, a 2D grid of height values, created using a terrain-generation algorithm, or from real-world data. Terrains have a number of applications: flight simulators, computer games, computer animation, virtual reality, visualisation and geographic tools, to name a few. These applications depend on the speed in which terrains are displayed to the user—for example, flight-simulators require immediate feedback to be effective and gamers crave for increased frame rates.

There are two main difficulties in rendering terrains. The first is that there may be too much detail to display a terrain at full resolution, and maintain real-time frame rates. The most straightforward, and accurate, approach to render a terrain is to use a "brute force" method, where every height value in the heightmap is used to form a terrain mesh. This technique is illustrated in Figure 1.1. Every point where lines intersect in Figure 1.1(a) has a single corresponding height value from the heightmap in Figure 1.1(b). The terrain is constructed, as in Figure 1.1(c), using the height values at each point. There is a one-to-one mapping between the heightmap values and terrain



(a) The brute-force triangulation (b) An example heightmap (c) The resulting terrain

Figure 1.1: Constructing a terrain from a heightmap using the brute-force approach.

vertices, so the terrain is rendered at maximum detail. Since the tessellation of triangles (the *triangulation*) is constant, the vertices can be cached, or the entire rendering process stored in a display list for efficient execution.

This approach is valuable for pre-rendered terrains, where the camera does not move, or follows a pre-defined path. For example, this technique might be used for computer-generated imagery (CGI), when realism is of utmost importance, and the scene can be planned to the smallest detail. However, a brute-force rendering of a terrain is not so useful for real-time terrain rendering, at least, traditionally.

A number of algorithms have been proposed that render simplified representations of terrains [7]. These algorithms increase frame rates, ideally without any noticeable change in the terrain geometry, by using Level of Detail (LOD). LOD decreases the number of polygons by providing detail only where it is necessary. For example, parts of the terrain far from the viewpoint or where the terrain is flat can be rendered with fewer polygons, without any noticeable difference in quality. The LOD *metric* determines where more or less detail is required. A number of different metrics exist, for example: the distance from the viewpoint, the area the triangle occupies on screen, or the difference between the height at the hypotenuse midpoint and the sampled height at that point from the heightmap (the *error* metric). Furthermore, metrics can be combined. For example, an effective approach is to divide the error metric by the distance of the triangle from the viewpoint [17].

The second difficulty is that, even if the terrain is rendered with the maximum detail available from a given heightmap, this can cause *aliasing*, when multiple vertices of the terrain correspond to the same pixel on screen. LOD can also prevent this, if the metric is based, directly or indirectly, on the screen-size of each triangle.

However, LOD introduces some new problems. A terrain should be *spatially* and *temporally coherent*. A terrain has spatial coherence if there are no visible *cracks* in the terrain. Unless precautions are taken, cracks appear along an edge between two adjacent regions of different levels of detail if the region of less detail cannot represent the height at a point where the region of higher detail can. A temporally coherent terrain does not rapidly change in geometry over time. This phenomenon can occur as the LOD changes, and is known as *popping*.

Originally, terrain-rendering techniques employed LOD to decrease the number of triangles that need to be rendered, thus reducing the load on the Graphics Processing Unit (GPU). However, for a suitable triangulation to be generated for a given terrain, computations are required to decide where more or less detail is needed. This increases the number of calculations performed on the computer processor (CPU), so in effect, the calculations are delegated from the GPU to the CPU. As will be discussed in Chapter 2, terrain-rendering algorithms can be broadly categorised into three classes: hierarchical techniques, triangular irregular networks and GPU-based approaches. In general, hierarchical techniques and triangular irregular networks decrease the load on the GPU, but subsequently increase the work done by the CPU.

Recently, the GPU has become rapidly more powerful, with increased processing power, greater bandwidth and reduced latency. Whereas early graphics hardware could barely process geometry, the GPU is now more effective than the CPU for some tasks. The GPU is *parallel* in structure, so is suited to executing multiple tasks at once, and is *specialised*, with special-purpose hardware for particular tasks [19]. As a result, it has

become more effective to render large numbers of triangles on the GPU, with minimal CPU computations. Performing as many operations as possible on the GPU also frees up the CPU for other computations. For example, if a terrain were to be included as part of a computer game, a GPU-based terrain-rendering algorithm would allow the CPU to be free for game logic.

The most recent graphics hardware is capable of rendering tens of millions of triangles per second—enough to make the brute force approach to terrain rendering possible in real-time for small terrains. However, we still need to make an effort to maximise the performance of the GPU, which is what "GPU-based" techniques aim to do. Although modern graphics hardware can render more triangles per second, this rendering throughput may be limited by the CPU, that has to upload the specifications of each triangle to render. This can be avoided by caching geometry on the graphics card. Furthermore, we can group triangles together in *strips* or *fans*, so fewer vertices need to be specified.

The geometry clipmap is a recently proposed GPU-based technique that looks particularly promising [16]. This approach renders regular nested grids that are centred around the viewpoint. The grids vary in size, but have the same number of vertices, resulting in smaller grids of fine detail, and larger grids of coarse detail. The majority of operations are performed on the GPU, instead of the CPU.

1.1 Motivation

The geometry clipmaps algorithm is a novel method, but its effectiveness has yet to be fully examined. Losasso and Hoppe compare the rendering rate of geometry clipmaps with two other techniques by how many million triangles they display per second [16]. However, the number of triangles is strongly dependent on the graphics card used, which is different for each implementation.

Additionally, geometry clipmaps may perform "better" than other techniques in some situations, and not others. In particular, they appear to be most effective for large terrains, due to their potential with terrain compression, but may not be the best approach if the terrain is small. Although geometry clipmaps are tailor-made for high-end graphics equipment, they may still be effective with older graphics cards.

Knowing a terrain-rendering algorithm's strengths and weaknesses is of great importance if the algorithm is to be used in practical applications. By finding where geometry clipmaps are suitable can help prevent their use in inappropriate situations, saving time and money.

The aim of our research was to perform a comprehensive comparative analysis of geometry clipmaps with other terrain rendering techniques to find when they are appropriate to use. Since no public implementation of geometry clipmaps is currently available, our first step was to implement the geometry clipmaps algorithm. It was hypothesised that geometry clipmaps will be the most efficient algorithm to render terrains of considerable size, but other techniques will be more appropriate for small terrains.

Due to the rapid growth of Graphical Information System (GIS) systems, GIS data has become increasingly easy to obtain. GIS data represents real-world objects, such

as land elevations, houses or roads, in digital format. Another objective was to render our terrains from Digital Elevation Models (DEMs), commonly part of GIS systems, that represent the topography of the earth. Such data is freely available for all of USA, thanks to the United States Geological Survey (USGS)[1]. Similar data of New Zealand can be purchased from Land Information New Zealand (LINZ)[2].

In the remainder of this report, we review existing terrain-rendering techniques in Chapter 2, and then discuss our implementation of the geometry clipmaps algorithm, various implementation issues and how they were resolved in Chapter 3. Then, we present results from a comparative analysis of geometry clipmaps with other rendering algorithms in Chapter 4, before finishing with a conclusion and potential future work in Chapter 5.

[1] These DEMs are available for download at http://edc.usgs.gov/geodata/
[2] http://www.linz.govt.nz/

Chapter 2

Background

In this chapter, we outline prior work relating to geometry clipmaps. Firstly, we introduce some algorithms that preceded the geometry clipmaps algorithm in section 2.1. In section 2.2, we review texture clipmaps, the texturing equivalent of the geometry clipmaps algorithm. Finally, we present the algorithm itself in section 2.3.

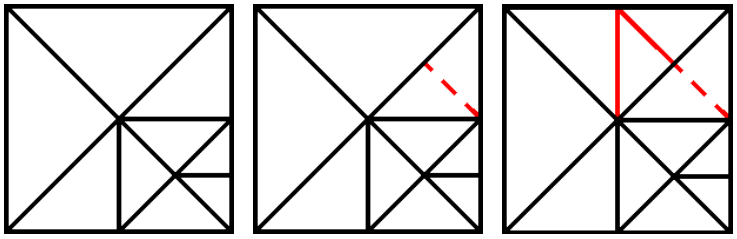
2.1 Previous Terrain-Rendering Algorithms

Although a number of techniques have been proposed for rendering terrains, we classify them into three categories: *hierarchical algorithms*, that recursively subdivide the heightmap using a common data structure; *triangular irregular meshes*, where the triangles can be of any shape and size to give the most faithful representation of the terrain; and *GPU-based* approaches, that cache vertices or triangles on the graphics card, so they can be rendered efficiently. This section discusses each of these classes, and some of the particular terrain-rendering algorithms.

2.1.1 Hierarchical Algorithms

Hierarchical terrain-rendering algorithms recursively subdivide the heightmap into a primitive shape, resulting in a hierarchy. Various shapes can be used; the only requirement being that one instance of the shape can be partitioned into s smaller copies of the same shape. Subdivisions may introduce cracks in the terrain, which can be prevented by adding or discarding vertices, or by subdividing further on an adjacent shape. Real-time Optimally Adapting Meshes (ROAM) [6, 1] and QuadTrees [22, 15] are two common examples of hierarchical techniques.

A ROAM triangulation contains only right-angled isosceles triangles. Each triangle can be *split* into two right-isosceles triangles of half the size, by dividing along a line from the apex to the midpoint of the hypotenuse. An inverse process exists where triangles are *merged*. The resulting binary tree (or *BinTree*) of triangles is stored in memory. Priority queues are used to manage which triangle is most in need of a split or merge operation.



(a) An initial ROAM triangulation

(b) A potential triangle split, with the T-junction evident

(c) The result of a force split, required to prevent a crack in the terrain

Figure 2.1: An example of a triangle force split using ROAM.

An example of a ROAM triangulation, and a potential triangle split is shown in Figures 2.1(a) and 2.1(b) respectively. However, such a split will introduce a crack in the terrain, as the height of the new vertex resulting from the split is different to the interpolated height on the top triangle with which it shares an edge. This is known as a *T-junction*, as a part of the triangulation has a 'T' shape. To prevent this T-junction, we need to perform split operations on the adjacent triangle with which it shares its hypotenuse. This is known as a *force split*. In this example, the force split results in two more splits, as shown in Figure 2.1(c).

Some variants of ROAM exist that follow a similar process, but use a different data structure. The "diamond" algorithm uses, confusingly, equilateral triangles that split into four more equilateral triangles [11]. This results in a triangle QuadTree. Meanwhile, the proposed ROAM Version 2.0 [5] uses diamonds instead of triangles to decrease the amount of memory and processing required.

QuadTrees are based on a similar idea to ROAM, but rectangles, or preferably squares, are used. Each square can be split into four squares of quarter the size, resulting in a *QuadTree* of squares. To prevent T-junctions, we enforce the constraint that no two adjacent squares differ by more than one level of detail, and when adjacent squares do differ by a single level, a single vertex is discarded.

A strength of hierarchical techniques is that they allow adaptive refinement, where a part of the terrain can be represented with more or fewer triangles as required, and this structure can change rapidly as the viewpoint moves. They also produce accurate approximations of the terrain, using considerably fewer triangles. In fact, ROAM can generate what is considered an *optimal* triangulation: the most suitable triangulation for a given heightmap, containing only right-angled isosceles triangles. Additionally, the hierarchies are naturally very easy to traverse. This not only makes rendering simple once a triangulation is generated, but it is also beneficial for view-frustum culling[1] and occlusion culling[2]. If a non-leaf triangle is out of view, all its descendants will also not be visible.

[1] View-frustum culling prevents triangles that are out of the current view frustum from being rendered, to increase the rendering efficiency.

[2] Occlusion culling prevents triangles that are hidden behind other objects from being rendered.

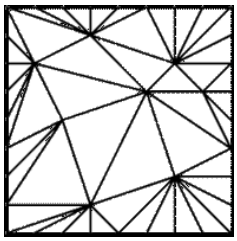


Figure 2.2: A TIN triangulation

However, hierarchical techniques have some inherent weaknesses. Adaptive refinement requires a number of CPU computations to calculate where more or less detail is necessary, and a significant amount of memory to track the current state of the triangulation. As the triangulation gets larger, more resources are consumed. On modern graphics cards, reducing the number of triangles for rendering is not the best approach; instead, we aim to maximise the number of triangles the GPU can render. Furthermore, it is difficult to form large triangle strips, which can be rendered more quickly, since the triangulation is not uniform.

2.1.2 Triangular Irregular Networks

Triangular Irregular Networks (TINs) represent the terrain as a number of triangles of different shapes and sizes. An example of an irregular triangulation is shown in Figure 2.2. Such irregular meshes give the most faithful approximation of a terrain, as an optimal triangulation for a given number of triangles can be obtained.

However, these techniques are highly CPU intensive and require a considerable amount of memory to model the mesh and keep track of refinement dependencies. In general, they consume even more resources than hierarchical techniques. Moreover, any form of geometry caching is difficult, due to the irregular structure, and triangle strips or fans may only be possible at a cost of further CPU computations. Therefore, this class of terrain-rendering algorithm is not suitable in modern systems; they do not make utilise the GPU, they require extensive use of the CPU, and use a large amount of memory. An example of a TIN algorithm where the triangulation is refined as the viewpoint changes is View Dependent Progressive Meshes (VDPM) [13].

Delaunay triangulations [3] impose a constraint on TINs: for any triangle in the triangulation, no vertex is inside the smallest circle that contains the triangle. This restricts the possible triangulations, resulting in a less accurate representation that may not be strictly optimal. However, the intent of this constraint is to prevent “sliver” triangles that often do little to improve the rendering quality.

2.1.3 GPU-based Algorithms

In the late nineties, as more graphics cards with more powerful processors were released, previous algorithms were adapted to render more triangles. With these tech-

niques, sets of triangles, known as *clusters*, *chunks* or *aggregate* triangles are cached on the graphics card and rendered together. Examples of the algorithms that used this approach are RUSTiC [21], CABTT [14] and Ulrich's Chunked LOD algorithm [24]. These algorithms use the same process and data structures as the hierarchical techniques, but where a single primitive was rendered, a cluster would be displayed instead. By caching these regions in video memory, performance would be largely unchanged, but rendering quality would be improved.

In addition to adapting previous approaches, new algorithms were introduced that were tailor-made for modern graphics hardware [2, 12]. One such algorithm is geometric mipmapping (geomipmapping) [4], a geometry equivalent of texture mipmapping. With geomipmapping, the terrain is divided up into a regular grid, called a *patch* or *tile*. Each patch can be rendered at several levels of detail; the LOD is varied by changing the space between grid lines. The tiles can be cached in vertex buffers on the graphics card for fast access, and can be rendered efficiently using triangle strips. Cracks between adjacent tiles of different levels of detail can be avoided by adding or removing vertices.

Care needs to be taken to prevent popping, a sudden change in the terrain geometry when a patch's level of detail changes. Using an appropriate metric can help, but is not usually enough for a high-quality rendering. *Geomorphing*, where vertices move gradually into place, can be used to avoid popping. This process can be performed in the vertex shader at virtually no cost [25].

The geometry clipmap is another GPU-based algorithm, but is discussed in section 2.3.

2.2 Texture Clipmaps

The texture clipmap defines a way to represent a texture of arbitrarily large size at a number of levels of detail [23]. The texture is represented as regular grids at power-of-two resolutions. Texture clipmaps support real-time rendering, largely due to incremental updates of textures using toroidal arrays. As will soon be evident, this technique inspired the geometry clipmap.

2.3 Geometry Clipmaps

Losasso and Hoppe proposed the geometry clipmap in 2004 [16]. It focussed on performing as few operations as possible on the CPU, and instead utilising the rendering throughput that is now possible on recent GPUs. It aims to form a triangulation where each triangle is approximately pixel-sized on screen. The level of detail is dependent only on the distance from the viewpoint, not the terrain geometry, resulting in a simple data structure. It provides a steady rendering rate, even when the viewpoint is moving quickly.

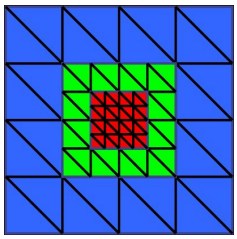


Figure 2.3: The regular nested grids of a geometry clipmap. Each clipmap level is a different colour.

2.3.1 Storing Height Values

A geometry clipmap renders a set of nested regular grids centred around the viewpoint, with small grids of high detail and large grids of low detail. Each grid contains $n \times n$ values and is called a clipmap *level*. The levels are numbered starting from $l = 0$ for the coarsest level. The distance between values at level l is the *grid spacing*, denoted g_l. An example of a clipmap of three levels, where $n = 5$, is shown in Figure 2.3, with levels $l = 0, 1$ and 2 coloured blue, green and red, respectively. The vertices in a clipmap level are stored in a vertex buffer on the graphics card. Since the grids are regular, multiple triangles can be easily combined into triangles strips, for more efficient rendering. As the viewpoint moves, the clipmap data is updated so the grids remain centred around the viewer.

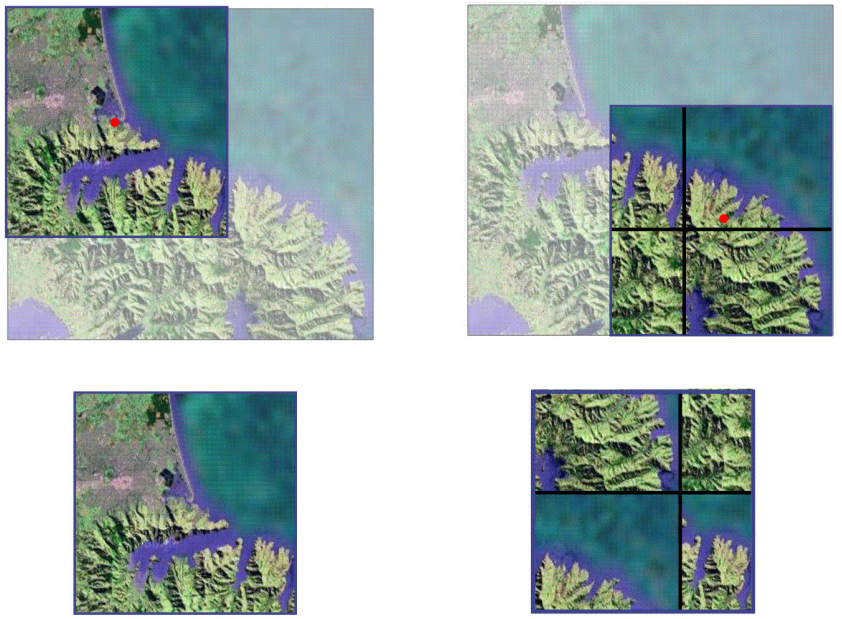
The vertices are stored as a *toroidal array* to enable *incremental updates*, where only vertices from newly visible areas are added, replacing areas that are no longer visible. Figure 2.4 shows how toroidal arrays make incremental updates possible. The heightmap and viewer position are shown, as well as the actual clipmap level data. Suppose the viewer is positioned as in Figure 2.4(a). If we move to the southeast, as shown in Figure 2.4(b), only the newly visible areas along the bottom and right edges of the heightmap need to be put in the array, and they are put in the top and left edges of the clipmap, respectively, overwriting the data that is no longer needed.

2.3.2 Clipmap Regions

For each level of the clipmap, a number of regions are defined. The *clip region* is ideally an $n \times n$ grid of data values, centred around the viewer, but may be a smaller region if considered too expensive to update. The *active region* is the $n \times n$ region centred around the viewpoint, but cropped to the available data in the clip region. The *desired* active region is the $n \times n$ region centred around the viewpoint, prior to any cropping. The *render region* is the region that will be rendered: the "hollow frame" obtained by excluding the extent of the next level's active region from the current level's active region.

Losasso and Hoppe specified four constraints on these regions. They are:

1. "clip_region$(l+1) \subseteq$ clip_region$(l) \ominus 1$, where $\ominus$ denotes erosion by a scalar



(a) Before a change in viewpoint (b) After a change in viewpoint to the southeast

Figure 2.4: An example of the data in the heightmap (top) and toroidal array (bottom) before and after a change in viewpoint. The position of the viewer in the heightmap is shown by the red dot.

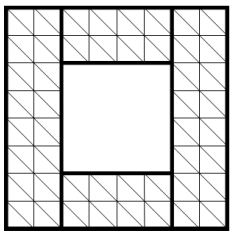


Figure 2.5: The four rectangular regions the render region is partitioned into.

distance."

2. "active_region(l) $\subseteq$ clip_region(l)"
3. "the perimeter of active_region(l) must lie on 'even' vertices"
4. "active_region($l+1$) $\subseteq$ active_region(l) $\ominus 2$"

The first of these constraints is necessary for terrain compression, which requires at least one grid unit between clip regions. The second constraint ensures that we only render data that is currently available. The third constraint is necessary for the grid at the next-coarsest level to line up. The "even" vertices at a level are those that would also be used by the next-coarsest level that has twice the grid spacing. Finally, the fourth constraint allows room for vertices near the perimeter of a level to be morphed to the height values at the next-coarsest level, to prevent cracks.

The render region is subdivided into four rectangular regions, as shown in Figure 2.5. Each of these four regions is view-frustum culled, and then rendered using triangle strips. Due to the toroidal access, the vertex indices need to be recalculated every frame, on the CPU.

2.3.3 Transition Regions

One issue with the geometry clipmaps algorithm is how to deal with the discontinuities between levels. As evidenced by Figure 2.3, along an edge between two levels, there are vertices where height values will be defined by the finer level, but not the courser level. This introduces cracks, as the height value from the finer level may not match the interpolated height from the courser level.

To eliminate these cracks, Losasso and Hoppe suggest morphing the height values near the edge of the finer level, using a vertex shader. For a vertex (x,y,z) where the height at the next-coarser level is z_c, w is the width of the transition region, (v_x^l, v_y^l) is the position of the viewpoint in the clipmap level and the bounds of the active region are (x_{min}, y_{min}) and (x_{max}, y_{max}), the suggested formula for the morphed elevation z' is:

$$z' = (1-\alpha)z + \alpha z_c$$

where $\alpha = \max(\alpha_x, \alpha_y)$,

$$\alpha_x = \min\left(\max\left(\frac{\left|x - v_x^l\right| - \left(\frac{x_{max} - x_{min}}{2} - w - 1\right)}{w}, 0\right), 1\right)$$

and similarly for α_y.

2.3.4 Other Functionality

The algorithm can also support *graceful degradation*, *compression* and *synthesis*. The clipmap is rendered from coarse-to-fine levels, and the number of updates per frame can be constrained, so if the viewer is moving rapidly, fewer updates can be performed to maintain steady frame rates. Although this results in less detail, it will not be noticeable due to the speed at which the viewer is moving. This technique is called graceful degradation.

As with images, lossy compression can be used to decrease the space required to store a terrain heightmap. Starting at the coarsest level, finer levels can be predicted, using the localised regularity of a terrain. The difference in the predicted and actual heights can be stored as the *residual*. Instead of a huge heightmap, the residuals can be kept in memory, and parts of the heightmap can be uncompressed as they are required.

As the user moves close to the terrain, newly synthesised data can be used for a higher quality rendering than the heightmap can provide. This allows the terrain to be rendered at a potentially infinite resolution.

2.3.5 Vertex Textures

Asirvatham and Hoppe wrote a chapter in GPU Gems 2 [19] on geometry clipmaps that builds on the original implementation by using vertex textures to further increase the number of computations performed on the GPU. Vertex textures [9] are a new feature in DirectX 9 Shader Model 3.0, currently supported only by the recent GeForce 6 or 7 series graphics cards. A vertex texture fetch allows vertex shaders to read data from textures. By using vertex textures instead of vertex buffers to store the vertices, all updates to the clipmap levels can be performed on the graphics card in constant time. The indices calculation can also be performed in constant time, and all other aspects of the algorithm, except decompressing the terrain, can be performed on the GPU.

Chapter 3

Implementation

Based on Losasso and Hoppe's paper [16], we have implemented our own geometry clipmaps terrain-rendering system. Our implementation is written in C++ using OpenGL, and it utilises the GLUT[1] library. The rendered terrain is based on a heightmap, represented as either a DEM, or a TGA[2]. Any such heightmap can be used.

Although our implementation is heavily based on Losasso and Hoppe's work, in this chapter we present how our implementation differs from theirs, and aspects they did not describe that required us to formulate our own approach.

The primary way in which our implementation differs from their paper is that we have not used a compressed form of the heightmap. We also do not synthesise extra detail when close to the terrain. This is principally for simplicity; our goal is to analyse the performance of the core geometry clipmaps algorithm, and compression and synthesis are largely tangential to the algorithm's performance. Using a compressed form of the terrain allows the entire heightmap to be stored in memory, avoiding hold-ups due to disk paging. We have not used heightmaps that are too big to fit in memory, so this has not been a problem.

However, not using compression has introduced some new challenges. At any non-finest level, we require height values that not only represent that vertex, but also the surrounding heights, depending on the grid spacing. Given that "heightmaps are remarkably coherent in practice" [16], we chose to just sample every ith value, where i is the grid spacing. This approach is very quick and was found to be effective, so the more computationally expensive alternative of averaging the nearby heights was deemed unnecessary.

We enforce the constraint that $n = 4k + 1$ for any $k > 1$, so that each level can be exactly centred on the next-finest level. As shown in Figure 3.1, $(n-1)$ must be divisible by 4, because every level has $n-1$ grid units, $\frac{n-1}{2}$ which must coincide with the finer level along an adjoining edge, and $\frac{n-1}{4}$ units which must be positioned on either side. In fact, $k = \frac{n-1}{4}$ is the maximum number of grid units that border each

[1] OpenGL Utility Toolkit: a library of utilities which performs functions such as window creation and key handling.

[2] A graphics file format. The colour of each pixel represents the height of the terrain at that point.

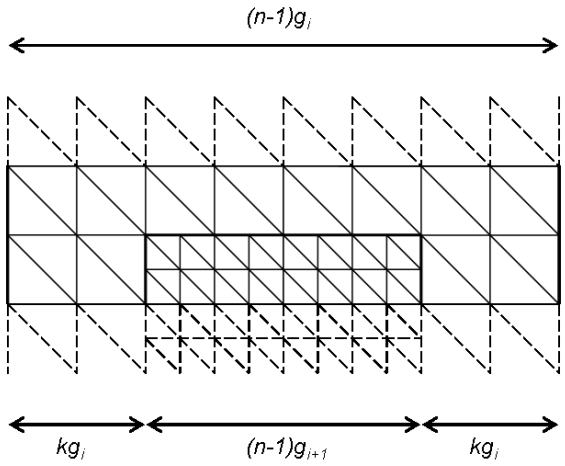


Figure 3.1: Centering clipmap level i on the next-finest level $i+1$.

clipmap level. Losasso and Hoppe's fourth constraint requires there to be at least 2, so $k > 1$.

Another key difference in our implementation is how we define the grid units. Losasso and Hoppe specify that, for the lth level, the grid spacing is $g_l = 2^{-l}$ so the desired active region is of size $ng_l \times ng_l$. For the coarsest level $l = 0$, the grid spacing is $g_0 = 1$ so the active region is at most an $n \times n$ grid. Supposing there are m levels, the finest level is $l = m-1$, with a desired active region of size $n \cdot 2^{1-m} \times n \cdot 2^{1-m}$.

We take a different approach where the grid spacing at any level is a whole number of units. For the finest level, we say the grid spacing is $g_{m-1} = 1$. For each coarser level, the grid spacing doubles, so in general $g_l = 2^{-l+m-1}$. In effect, we have just multiplied Losasso and Hoppe's grid spacing by 2^{m-1}: the inverse of their grid spacing at the finest level. The active region can still be up to $ng_l \times ng_l$ in size, so the finest level becomes $n \times n$, and the coarsest level $n \cdot 2^{m-1} \times n \cdot 2^{m-1}$.

Note also that although there are $n \times n$ height values, strictly speaking the desired active region is actually a $(n-1)g_l \times (n-1)g_l$ grid, as there are only $n-1$ gaps between the height values.

Another area that was not discussed in the original paper is how to cope with a finite terrain. One possibility would be to use some fixed height whenever outside the bounds of the heightmap—in effect, assuming the rest of the world is flat. However we did not choose this approach as it requires extra computations when updating the clipmap to see if we are outside the heightmap bounds, it produces cracks if the edges of the heightmap are not all a fixed height, and when part of a larger scene, it may not be appropriate to render outside these bounds. Instead, we only render the terrain within the area defined by the heightmap. This introduces some difficulties when the

viewpoint is near the perimeter, as discussed in section 3.4.

Finally, it should also be noted that we conform to a different coordinate system than Losasso and Hoppe. They use the *z* component for the height values, so the *z*-axis points to the sky. Our *y*-axis points "upwards", so the *y* component stores the height values.

A screenshot from our implementation can be seen in Figure 3.2. Figure 3.2(b) demonstrates how the terrain is made up of multiple clipmap levels, centred around the viewpoint.

The rest of this chapter discusses the finer details of our implementation, issues that arose, and how they were resolved. Firstly, in section 3.1 we discuss the data structures used to store and access height values, and how they the terrain geometry can be formed from them. In section 3.2, we describe how to manage the multiple clipmap levels that make up a clipmap, and in section 3.3 we present how cracks were prevented between these levels. Issues arising from a finite heightmap are discussed in section 3.4 and section 3.5 describes how view-frustum culling was implemented.

3.1 Storing and Accessing Height Values

Storing and accessing height values is an integral part of the geometry clipmaps algorithm. Vertex buffers, stored on the graphics card, are used to access data quickly. Incremental updates are possible due to toroidal arrays. The terrain geometry can be rendered by forming a sequence of vertices from the vertex buffer. This section describes how these aspects are implemented.

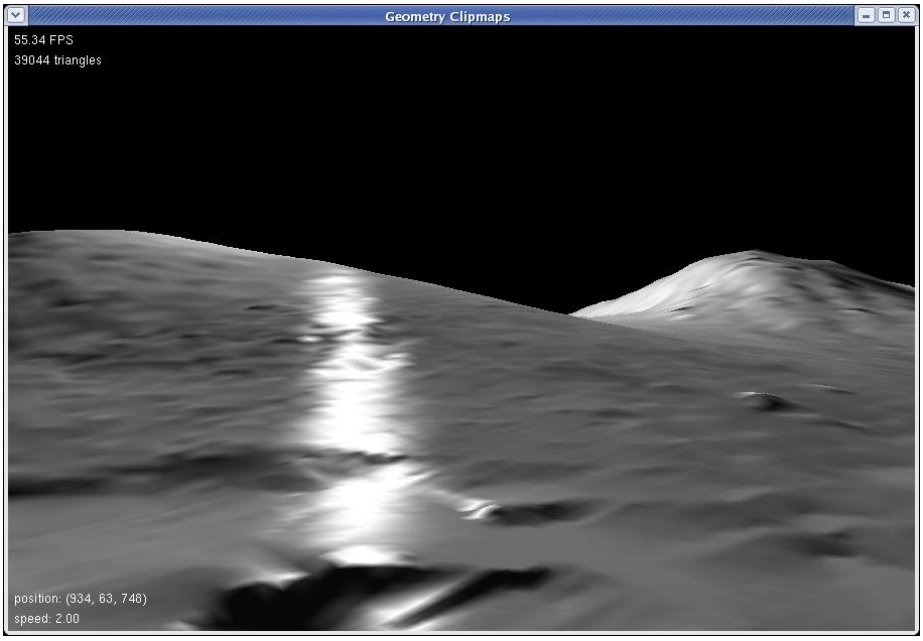
3.1.1 Vertex Buffers

Losasso and Hoppe originally used a vertex buffer to store the vertex data in each clipmap [16]. By using vertex buffers, the vertices are stored in video memory so they can be accessed quickly. In our implementation, we used the "vertex buffer object" OpenGL extension [18]. This is a "standard extension", approved by the OpenGL Architecture Review Board (ARB) and supported by most graphics cards produced in the last three years.

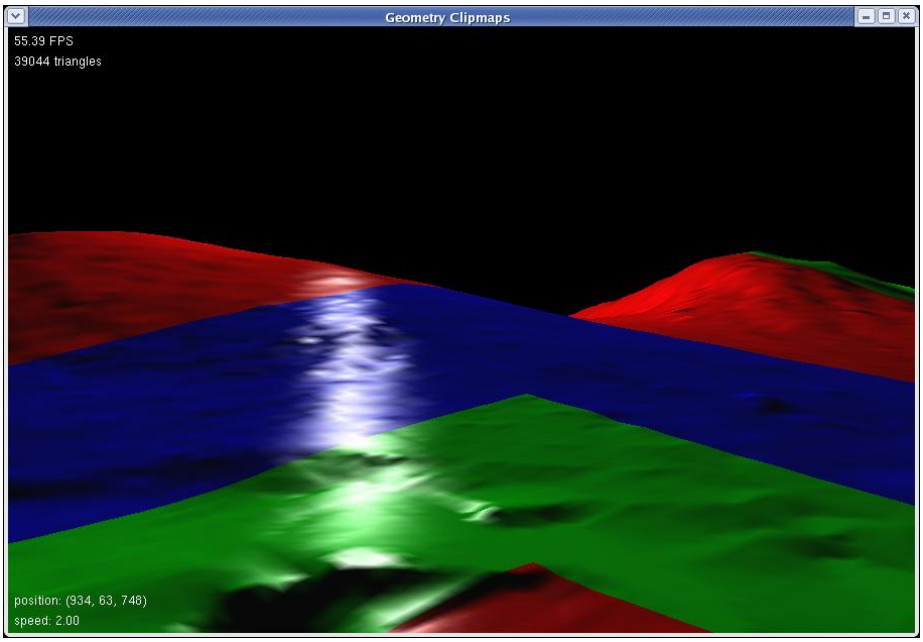
As stated in section 2.3, Asirvatham and Hoppe have since suggested the use of vertex textures instead of vertex buffers [19]. Although this would result in a performance improvement on recent graphics cards, only NVIDIA GeForce 6 series and 7 series graphics cards currently support them. We decided against using them, so we could analyse the performance of the algorithm on other hardware.

3.1.2 Toroidal Arrays

The vertex buffer is used to store the vertices in a toroidal array. A toroidal array uses wraparound addressing so it can be updated incrementally. As the viewpoint moves, only new elements are inserted into the vertex buffer. Vertices that are present in both the new and previous view do not need to be updated.



(a) White terrain



(b) Clipmap levels composing the terrain

Figure 3.2: A screenshot from our implementation. The terrain is of Hawaii. The geometry clipmap contains five levels and is of size $n = 129$.

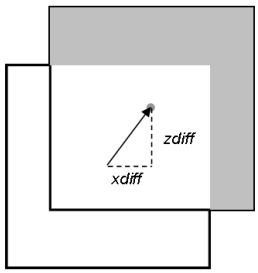


Figure 3.3: The "L-shaped" region that needs to be updated after the viewpoint moves. The grey shaded region is the data that is updated.

For our purposes, we need to find a mapping for each element into an array, that satisfies the following two properties:

1. An element i will always map to the same position in the array $f(i)$.

2. For a set of any n adjacent elements, there is a minimal perfect mapping into the array. That is, each element maps to a different position, and there are no gaps in between.

The first of these properties makes incremental updates possible, because once an element is in an array, we don't need to move it if the viewpoint changes. The second property prevents required data from being overwritten, and assures that the array is of minimum size.

At the finest level, where the grid spacing is $g_{m-1} = 1$, such a mapping is trivial. To store each of the $n \times n$ vertices, a vertex (x,y,z) is stored in the vertex buffer at $(x \bmod n, z \bmod n)$.

To update the finest clipmap level after a camera move, just the newly revealed "L-shaped" region of the terrain needs to be filled-in, as shown in Figure 3.3, overwriting the part that is no longer visible. The pseudocode that does this is shown in Algorithm 1. (`xdiff`,`zdiff`) represents the change of position in the *xz*-direction, and `xmin`, `xmax`, `zmin` and `zmax` are the bounds of the clipmap level's new active region. Lines 2 to 17 show how the clipmap level is updated if the view moves "upwards," as in Figure 3.3. First, the side region is updated, in lines 2 to 12, either to the right or left, depending on the sign of `xdiff`. Then the top strip of the upside-down "L" is updated, in lines 13 to 17. The process for "downward" movement is similar, as shown in lines 19 to 34. First the bottom strip is updated, then the side region. Note that, as the `zdiff` value is negative, the $-$`zdiff` statements will effectively be adding the absolute change in position. If `zdiff` $= 0$, the first `for`-loop will be skipped, as the terminating condition will be true immediately. This means no bottom strip is updated, as expected, since there has been no movement in the *z*-direction.

Another toroidal array can be used, also backed by a vertex buffer, to store the

Algorithm 1 Incrementally updating the finest clipmap level. `xmin`, `zmin`, `xmax`, `zmax` give the bounds of the new active region and (`xdiff`, `zdiff`) is the vector of movement.

```
1: if zdiff > 0 then
2:   for z = zmin to zmax − zdiff do
3:     if xdiff > 0 then
4:       for x = xmax − xdiff + 1 to xmax do
5:         update vertex (x, z)
6:       end for
7:     else if xdiff < 0 then
8:       for x = xmin to xmin − xdiff − 1 do
9:         update vertex (x, z)
10:      end for
11:    end if
12:  end for
13:  for z = zmax − zdiff + 1 to zmax do
14:    for x = xmin to xmax do
15:      update vertex (x, z)
16:    end for
17:  end for
18: else
19:  for z = zmin to zmin − zdiff − 1 do
20:    for x = xmin to xmax do
21:      update vertex (x, z)
22:    end for
23:  end for
24:  for z = zmin − zdiff to zmax do
25:    if xdiff > 0 then
26:      for x = xmax − xdiff + 1 to xmax do
27:        update vertex (x, z)
28:      end for
29:    else if xdiff < 0 then
30:      for x = xmin to xmin − xdiff − 1 do
31:        update vertex (x, z)
32:      end for
33:    end if
34:  end for
35: end if
```

normals at each vertex. When the viewpoint changes, the normals can be updated in the same fashion.

3.1.3 Calculating Indices

To build a surface using the elements of a toroidal array, the order of the vertices needs to be specified. A surface is formed by ordering the vertices into triangle strips. An array of indices, specifying this ordering, can be found given the eye position, taking into account which vertices in the vertex buffer correspond to the edge vertices of the clipmap level. However, this array needs to be updated whenever the viewpoint changes, as the location of the edges in the vertex buffer also changes. Additionally, when view-frustum culling is performed (see section 3.5), only a subset of the entire region needs to be rendered, and the indices array needs to reflect this. In effect, the set of vertices in the indices array implicitly represents the *render region*.

To find each of the four regions, the active region of the level, and of the next finest level, must have already been calculated. Section 3.2.2 explains how the active regions are found. Once the extent of each region has been determined, the sequence of indices can be computed, as shown in Algorithms 2 and 3. The location of the active region extents in the toroidal array (`xmodmin`, `xmodmax`, `zmodmin`, `zmodmax` $\in [0,1,\ldots,n-1]$) are found using the mapping into the toroidal array, for example: `xmodmin` $=$ `xmin` mod n. Then, the general process is to go across each "row" (that is, vary `x` from `xmodmin` to `xmodmax`) adding (`x`, `z`) and (`x`, `z`+1) to the array of indices. When `x` or `z` equals n, the values need to wraparound to 0.

Algorithm 2 Determine the sequence of indices by which we can form the region defined by (`xmodmin`, `zmodmin`) and (`xmodmax`, `zmodmax`), the location of the active region extents in the toroidal array.

```
1:  if zmodmax < zmodmin then
2:    for z = zmodmin to n − 2 do
3:      calculate_row(xmodmin, xmodmax, z, z + 1)
4:    end for
5:    calculate_row(xmodmin, xmodmax, n − 1, 0)
6:    for z = 0 to zmax − 1 do
7:      calculate_row(xmodmin, xmodmax, z, z + 1)
8:    end for
9:  else
10:   for z = zmodmin to zmodmax − 1 do
11:     calculate_row(xmodmin, xmodmax, z, z + 1)
12:   end for
13: end if
```

The triangle strips of each row are combined using four zero-area triangles. This is achieved by specifying two extra vertices to transition from one row to the next. At the end of a row, the last vertex is used for a second time, and at the beginning of the next row, the first vertex is also specified twice. As a result, we require $2n+2$ indices to

Algorithm 3 `calculate_row(int x0, int xn, int zn, int zn1)`

```
1:  add index (x0, zn) to array {extra index, repeat the first of row}
2:  if x0 ≤ xn then
3:     for x = x0 to xn do
4:        add index (x, zn) to array
5:        add index (x, zn1) to array
6:     end for
7:  else
8:     for x = x0 to n − 1 do
9:        add index (x, zn) to array
10:       add index (x, zn1) to array
11:    end for
12:    for x = 0 to xn do
13:       add index (x, zn) to array
14:       add index (x, zn1) to array
15:    end for
16: end if
17: add index (xn, zn1) to array {extra index, repeat the final index again}
```

form each row and there are $n-1$ rows, in an active region that has not been cropped, so the maximum possible number of indices for each clipmap level is:

$$(2n+2)(n-1) = 2n^2 - 2$$

Using these extra zero-area triangles allows multiple rows to be combined to form larger triangle strips, increasing the rendering efficiency.

3.2 Multiple Clipmap Levels

The previous section discussed how values are managed for the finest level of a clipmap, where the grid spacing is 1. Having multiple levels in a clipmap adds some further complications. This section discusses the difficulties, and how they were resolved.

3.2.1 Storing in Toroidal Array

Remember that at any level l, we sample every g_lth value. Thus, for any non-finest clipmap level, where the grid spacing is greater than one, the data is not tightly packed. Consider a single row of a non-finest clipmap level. Since the z-value does not change, we are only concerned with the x-values. Suppose they are:

$$a, a+g, a+2g, a+3g, \ldots, a+(n-1)g$$

where g is the grid spacing. We need to pack the n values, varying from a to $a+(n-1)g$ into an array varying from 0 to $n-1$. Each adjacent pair of vertices differs by g, so the

first step is to divide each value by g. Then the values are:

$$\frac{a}{g}, \frac{a}{g}+1, \frac{a}{g}+2, \frac{a}{g}+3, \ldots, \frac{a}{g}+(n-1)$$

Each pair of adjacent vertices now differ by one, as with the finest level where $g = 1$. For our implementation, $\frac{a}{g}$ will always be a whole number, since, when the grid spacing is g, only vertices with coordinates that are a multiple of g are sampled. All that is then required is to mod each value by n.

In the general two-dimensional case: a vertex (x,y,z) is stored in the vertex buffer at $(\frac{x}{g} \bmod n, \frac{z}{g} \bmod n)$. This satisfies the two required properties presented in section 3.1.2: a vertex (x,y,z) will always map to the same position, and there is a minimal perfect mapping for vertices in the clipmap level to the toroidal array.

Having a grid spacing greater than one also affects the update algorithm. Algorithm 1 can be modified to work with any clipmap level by incrementing the `x` or `z` variable in a `for`-loop by the grid spacing, rather than by one.

3.2.2 Active Regions

For each clipmap level, an active region is maintained to keep track of the possible extent of the region when rendered. This region may be cropped if it is too expensive to update.

The active regions are required not only for finding the extents of the current level, but also the extents of the next-finest level, as the render region needs to exclude this area. Thus, we cannot calculate the sequence of indices to render a clipmap level unless both its active region and the next-finest active region have been determined. If the next finest active region changes and the render region is not updated, the levels will not line up at the edges. In our implementation, whenever a clipmap level is updated or the camera moves, a flag is set to indicate that the indices need to be recalculated. The calculation is only actually performed just prior to rendering the terrain. This guarantees that each clipmap level's active region has been determined, and the calculation is only performed at most once per frame.

The active region can be calculated given the camera's *xz*-position, the clipmap size n and grid spacing g. Intuitively, given a camera *xz*-position of (v_x, v_z), one might specify:

$$\begin{aligned} xmin_l &= v_x - \left\lfloor \frac{n}{2} \right\rfloor \cdot g_l \\ xmax_l &= xmin_l + (n-1) \cdot g_l \end{aligned} \tag{3.1}$$

and similarly for *zmin* and *zmax*. However, care must be taken to satisfy the third constraint specified by Losasso and Hoppe [16]. This constraint states that:

> 'the perimeter of active_region(l) must lie on "even" vertices, to enable a watertight boundary with coarser level $l-1$.'

This arises because, for a clipmap level with grid spacing g_l, the next-coarsest level must have grid spacing $g_{l+1} = 2g_l$. These two levels can only be aligned in *xz* if the

perimeter of the finer level is along "even" vertices at that level. In other words, if the lth clipmap level, with grid spacing $g_l = 2^{m-l-1}$, has an active region with bounds *xmin*, *xmax*, *zmin* and *zmax*, then *xmin*, *xmax*, *zmin* and *zmax* must all be divisible by $2g_l = 2^{m-l}$.

To satisfy this constraint, we first position the centre of the clipmap level on an "even" vertex:

$$(v_{x,l}, v_{z,l}) \quad = \quad (x - (x \bmod 2g_l), z - (z \bmod 2g_l)) \tag{3.2}$$

The active region can then be found centred around this vertex, using equation 3.1. Since the centre of the clipmap level is positioned on a multiple of $2g_l$, the minimum and maximum coordinates will also be multiples of $2g_l$ since $n = 4k+1$ so $\lfloor \frac{n}{2} \rfloor$ is a multiple of 2. Additionally, this results in every grid at level l being positioned on a multiple of g_l.

The use of a finite heightmap may further constrain the possible extents of the active region, as discussed in section 3.4.1.

3.3 Continuity Between Levels

Our approach for eliminating cracks between levels was the same as described by Losasso and Hoppe [16]. A GPU vertex shader was used, implemented using the Cg shading language [8]. The excerpt from the shader that performs the transition-region morphing is shown in Program 1. The parameters to the program are `pos`, the position of the vertex, `activeMin` and `activeMax`, uniform[3] variables of type `float2` specifying the bounds of the region the current vertex is a part of, and `width`, a uniform variable specifying the width of the transition region.

Program 1 Excerpt from a vertex shader that performs transition-region morphing to prevent spatial discontinuities in the terrain.

```
//compute blend parameter alpha
float2 centre = (activeMax + activeMin)/2;
float2 alpha2 = min(max((abs(pos.xz - centre)
    - ((activeMax - activeMin) / 2 - width - 1)) / width, 0), 1);
float alpha = max(alpha2.x, alpha2.y);

//obtain morphed elevation using alpha (pos.w is nextLevel's y)
pos.y = (1-alpha) * pos.y + alpha * pos.w;
```

The vertex program is evaluated using 17 instructions, although Losasso and Hoppe claimed they cost "about 10 instructions". However, our program calculates the view position based on the active region bounds, rather than having it supplied as a uniform variable, adding a couple of instructions.

[3] A `uniform` variable indicates the initial value is provided from an environment external to the Cg program

Note that the fourth component of each vertex stores the elevation at the next-coarsest level, rather than the w component for homogeneous coordinates. In our implementation, we obtain the elevation at the next-coarsest level directly from the heightmap. If the vertex falls on an even grid point, the elevation at the next-coarsest level is the same as at the current level. Otherwise, we interpolate between the height values at the nearest even grid points.

3.4 Using a Finite Heightmap

Using a finite heightmap can cause complications as the clipmap nears the edge of the heightmap. Since we chose to limit the rendered terrain to the extent of the heightmap, an effort must be made to ensure that the constraints are still followed. When the number of clipmap levels is large and the heightmap is small, there is a maximum number of levels possible, before the coarser levels become fixed, forcing less detail upon the edges. These complications are described in more detail in this section.

3.4.1 Level Bounds

Losasso and Hoppe's fourth constraint [16] states that:

> 'active_region$(l+1) \subseteq$ active_region$(l) \ominus 2$, since the render region must be at least two grid units wide to allow a continuous transition between levels.'

To satisfy this constraint when the active region of a non-coarsest clipmap level nears the edge of a heightmap, we require the perimeter of a clipmap level to be far enough away from the perimeter of the heightmap that two grid units of each coarser level will fit. To begin with, consider the finest level. There must be room for at least two grid units of each level other than the finest, so, remembering that $g_l = 2^{m-l-1}$:

$$\begin{aligned} xmin_{m-1} &\geq 2g_{m-2} + 2g_{m-3} + \cdots + 2g_0 \\ &\geq 2^2 + 2^3 + \cdots + 2^m \\ &\geq \sum_{i=2}^{m} 2^i \\ &\geq 2(2^{m-1} - 1) \end{aligned}$$

The final step of working uses the fact we have a geometric series, with initial term 2, common ratio 2, and $m-1$ terms[4].

[4]The sum of a geometric series with n terms, scale factor a and common ratio r is $\frac{a(r^n-1)}{r-1}$.

In general, for clipmap level l, we need to leave space for a border of two grid units for each level coarser than l, as shown:

$$\begin{aligned} xmin_l &\geq 2g_{l-1} + 2g_{l-2} + \cdots + 2g_0 \\ &\geq 2^{m-l+1} + 2^{m-l+2} + \cdots + 2^m \\ &\geq \sum_{i=m-l+1}^{m} 2^i \\ &\geq 2^{m-l+1}(2^l - 1) \\ &\geq 2^{m+1}(1 - 2^{-l}) \end{aligned} \tag{3.3}$$

Here we have a geometric series with initial term 2^{m-l+1}, common ratio 2, and l terms. Note that, as expected, this gives $xmin_0 = 0$. The formula for $zmin_l$ is identical.

For the maximum bound, at the coarsest level we must have a whole number of grid units, so we can "round off" to the nearest multiple of $g_0 = 2^{m-1}$. For each finer level, there must be room for two grid units of each of the coarser levels, as for the minimum bound. If there are h_x values, since we start at zero, the maximum value will be $h_x - 1$. So, if the heightmap is $h_x \times h_z$:

$$\begin{aligned} xmax_l &\leq h_x - 1 - ((h_x - 1) \bmod 2^{m-1}) - xmin_l \\ zmax_l &\leq h_z - 1 - ((h_z - 1) \bmod 2^{m-1}) - zmin_l \end{aligned} \tag{3.4}$$

3.4.2 Maximum Number of Levels

Given the size of the clipmap n and heightmap $h_x \times h_z$, we need to impose a constraint on the maximum number of levels in the clipmap. Without such a constraint, near the edges of the terrain, the grid spacing may be unnecessarily large to fit the minimum two grid units of each level. Even worse, this may result in one or more of the finer levels being discarded entirely.

The problem arises when the desired active region of the second-most coarse clipmap level encompasses more than the entire possible extent at that level. That is, the desired active region of level 1 exceeds both bounds $xmin_1$ and $xmax_1$, or $zmin_1$ and $zmax_1$. Then, so long as the viewpoint is within the bounds of clipmap level 1, the two coarsest levels become "locked" in place, just framing the terrain.

We avoid this by enforcing the constraint that the size of the desired active region at level 1 is smaller than the maximum bounds at this level. At level 1 the grid spacing is $g_1 = 2^{m-2}$, there are $n-1$ grids, and it is bordered on two sides, by two grid units of size $g_0 = 2^{m-1}$, so the constraint is:

$$\begin{aligned} & (n-1) \times 2^{m-2} + 2 \cdot 2 \cdot 2^{m-1} && \leq \min(xmax_0, zmax_0) \\ \therefore \quad & 2^{m-2}(n - 1 + 2^3) && \leq \min(h_x, h_z) - 1 \\ \therefore \quad & 2^{m-2} && \leq \frac{\min(h_x, h_z) - 1}{n+7} \\ \therefore \quad & m && \leq \left\lfloor \log_2 \frac{\min(h_x, h_z) - 1}{n+7} \right\rfloor + 2 \end{aligned} \tag{3.5}$$

For simplicity, we ignore the fact that $xmax_0$ or $zmax_0$ will be rounded down to the nearest multiple of 2^{m-1} as in equation 3.4. If required, the constraint can be tightened to:

$$m = \left\lfloor \log_2 \frac{\min(h_x, h_z) - 1}{n+7} \right\rfloor + 2$$

for the maximum number of triangles to be used to render the terrain.

To see why having the first two clipmap levels locked is a problem, consider the case where the heightmap is two units larger than the desired active region of the finest level, that is: $(h_x, h_z) = (n+2, n+2)$. We could attempt to have two clipmap levels, but, informally, we can see that this will result in a reduction in detail, as the extents of the finer level are restricted so the coarser level can have a two grid unit width. Formally, two clipmap levels would require a $2g_1 = 4$ unit border, leaving $n-7$ units for the finest level, resulting in a total of:

$$2\frac{(n+1)(n+1)+3(n-7)(n-7)}{4} = 2n^2 - 20n + 74$$

triangles. However, having one clipmap level would result in $2(n-1)(n-1)$ triangles, which is more than $2n^2 - 20n + 74$ when $n > 4.5$, which is always the case, since $n = 4k+1$ for any $k > 1$. This is supported by equation 3.5: with the aforementioned example, $m \leq \left\lfloor \log(\frac{n+1}{n+7}) \right\rfloor + 2$, so m must equal 1 for any $n > 5$, which, again, is always true. Our approach is to always prefer a greater number of triangles, at the cost of having a clipmap of smaller area. This also results in less static clipmaps, where the two coarsest levels do not ever require their height values to be updated.

An extreme case is when we are at least $n-1$ units short of enough room for $n-1$ grid units at the coarsest level, that is: $\min(h_x, h_z) \leq (n-1)g_0 - (n-1)$. In this situation, to maintain the boundaries at each level, the finest level will always be cropped down to 0, so the terrain cannot ever be rendered at the finest level of detail. Satisfying equation 3.5 prevents this situation from occurring.

3.5 View-frustum Culling

As with any terrain-rendering algorithm, view-frustum culling can be used to increase the frame rate, since usually only a small portion of the entire terrain is visible at one time. Losasso and Hoppe briefly outlined how view-frustum culling could be performed with geometry clipmaps. We follow this process closely.

After the render region for each non-finest level is partitioned into four regions, an AABB is constructed for each region, using the minimum and maximum height values for the entire terrain. The AABB is then intersected with the view frustum, as described in section 3.5.1. If the AABB is not in view, the region does not need to be rendered. If it intersects, we can crop the region to the minimum-sized rectangular region that contains all the triangles that are inside the view frustum, using the process described in section 3.5.2, and then render it. If the AABB is in view, it is also rendered.

3.5.1 Intersecting an AABB with the View Frustum

Given a view frustum and an AABB, we need to find whether the view frustum contains the AABB, if they intersect, or if the AABB is outside the view frustum. To do this, we use the method proposed by Greene [10].

This approach compares the AABB with each plane of the view frustum. Based on the normal $\boldsymbol{n}$ of each plane, we can define the *p*-vertex as the point on the AABB that is farthest in the direction of $\boldsymbol{n}$, and the *n*-vertex as the point on the AABB that is farthest in the direction of $-\boldsymbol{n}$. These can be found quite simply, as they will always be one of the eight corners of the cuboid. For example, if $\boldsymbol{n}_x > 0$, the *x*-component of the *p*-vertex will be the maximum *x* extent of the AABB.

Next, if the *p*-vertex is behind the plane (in the opposite direction to the normal), the entire AABB is outside the view frustum. If not, we check if the *n*-vertex is in front of the plane. If it is, the entire AABB is in front of that plane. If we find it is in front of all six planes of the view frustum, the AABB is in view. Otherwise, if the *n*-vertex is behind the plane for at least one normal, and the *p*-vertex is in front of the plane for all others, then the AABB intersects with the view frustum. This process is shown in Algorithm 4.

Algorithm 4 Find if the given AABB is contained by the view frustum, if they intersect, or if the AABB is outside the view frustum. The view frustum is defined by `planes[]`.

```
1:  result = inside
2:  for i = 1 to 6 do
3:      obtain equation for planes[i]: n · x + d = 0 where n is the normal to the plane
        and x = ⟨x, y, z⟩
4:      find the p-vertex P and n-vertex N
5:      if n · P + d < 0 then
6:          return outside
7:      else if n · N + d < 0 then
8:          result = intersection
9:      end if
10: end for
11: return result
```

3.5.2 Dynamically Cropping the Rendering Region

When the AABB intersects with the view frustum, part of the terrain within the bounding box is not visible, so we may be able to reduce the extent of the region, to decrease the number of triangles that are rendered. This can result in a substantial reduction in the number of triangles for rendering, as some of the AABBs are large. For example, at the coarsest level an AABB can encompass the length of the entire terrain, before view-frustum culling.

Our pseudocode for this process is shown in Algorithm 5. When the normal of the view-frustum plane is orientated in the $+z$-direction, we find the x-value of the point that shares *y*- and *z*-values with the *P*-vertex, and would lie on the plane. If the normal

is orientated in the $+x$-direction, this gives a minimum x-value for the region, otherwise we have a maximum x-value. The process is similar when the plane is orientated in the $-z$-direction, but x and z are interchanged.

Algorithm 5 Crop the render region to the minimum size such that all triangles that are in view are within the region. `xmin`, `xmax`, `zmin` and `zmax` define the render region.

1: **for** each plane $\boldsymbol{n} \cdot \boldsymbol{x} + d = 0$ of the view frustum that insersects with the AABB, with p-vertex $P = (p_x, p_y, p_z)$ **do**
2: **if** $\boldsymbol{n}_x > 0$ **then**
3: `newxmin` = $\left\lfloor \frac{-\boldsymbol{n}_z p_z - \boldsymbol{n}_y p_y - d}{\boldsymbol{n}_x} - \varepsilon \right\rfloor$
4: **if** `newxmin` $>$ `xmin` **then**
5: `xmin` = `newxmin`
6: **end if**
7: **else if** $\boldsymbol{n}_x < 0$ **then**
8: `newxmax` = $\left\lceil \frac{-\boldsymbol{n}_z p_z - \boldsymbol{n}_y p_y - d}{\boldsymbol{n}_x} + \varepsilon \right\rceil$
9: **if** `newxmax` $<$ `xmax` **then**
10: `xmax` = `newxmax`
11: **end if**
12: **end if**
13: **if** $\boldsymbol{n}_z > 0$ **then**
14: `newzmin` = $\left\lfloor \frac{-\boldsymbol{n}_x p_x - \boldsymbol{n}_y p_y - d}{\boldsymbol{n}_z} - \varepsilon \right\rfloor$
15: **if** `newzmin` $>$ `zmin` **then**
16: `zmin` = `newzmin`
17: **end if**
18: **else if** $\boldsymbol{n}_z < 0$ **then**
19: `newzmax` = $\left\lceil \frac{-\boldsymbol{n}_x p_x - \boldsymbol{n}_y p_y - d}{\boldsymbol{n}_z} + \varepsilon \right\rceil$
20: **if** `newzmax` $<$ `zmax` **then**
21: `zmax` = `newzmax`
22: **end if**
23: **end if**
24: **end for**

Chapter 4

Analysis

We analysed the performance of the geometry clipmaps algorithm in comparison with three other terrain-rendering techniques in two experiments. The first experiment was designed to find the comparative performance of each algorithm, and the second experiment determined the effect of a larger heightmap and faster graphics card.

The first of the terrain-rendering techniques that was compared with geometry clipmaps was a simple brute-force method. The brute-force approach compiles a display list of the instructions required to render the entire terrain, and then the display list is executed each frame to render the terrain. The terrain was rendered at maximum resolution. No view-frustum culling was used.

The second technique was an optimised version of ROAM. Triangles were rendered in fans of four to five triangles, on average, and view-frustum culling was performed by intersecting a bounding sphere with the view frustum. The implementation was frame-to-frame coherent, using two priority queues to maintain which triangles should be split or merged. The LOD metric for each triangle was the error at the hypotenuse midpoint, divided by the distance from the view position.

Thirdly, we used an implementation of geomipmapping, based on that provided by Polack [20]. We adapted it to run on a Linux system and to use the same heightmap and automatic camera movement as the other algorithms. The level of detail for each patch was calculated based on the distance from the viewpoint, and the thresholds for each level were hard-coded. The implementation prevented cracks between patches of different levels of detail by omitting the odd vertices on the finer patch that do not line up with any vertex on the coarser patch. View-frustum culling was implemented to render only patches that are currently in view. The vertex data was not cached on the graphics card, and vertex geomorphing, to prevent popping, was not implemented.

4.1 Experiment One

For our first experiment, we analysed the performance of each algorithm, comparing the frame rate as the number of triangles varied. Each terrain was rendered from a heightmap of Hawaii.

4.1.1 Method

The Hawaii terrain was rendered from a DEM. 1-degree DEMs are made freely available by the USGS. These give a 1:250,000-scale representation, corresponding to one sample every 3 arc-seconds. Although the DEM has a resolution of 1201×1201, we rendered a 1025×1025 region of the terrain, as this is the largest size that all the algorithms are capable of rendering.

The analysis was performed on a computer with a NVIDIA GeForce FX 5200 graphics card, 128MB of memory and a 8x AGP port. The computer had a 2.8GHz Pentium 4 hyper-threading processor, 1GB of memory, and a 512kB cache. Each algorithm was compared by finding the average frame rate in an automatic fly-through of the terrain. The fly-through was repeated three times for each technique, and the mean frame rate found. Textures, lighting and normals were all disabled, so just the performance of the core algorithms could be compared. The scene was rendered at a 900×600 resolution.

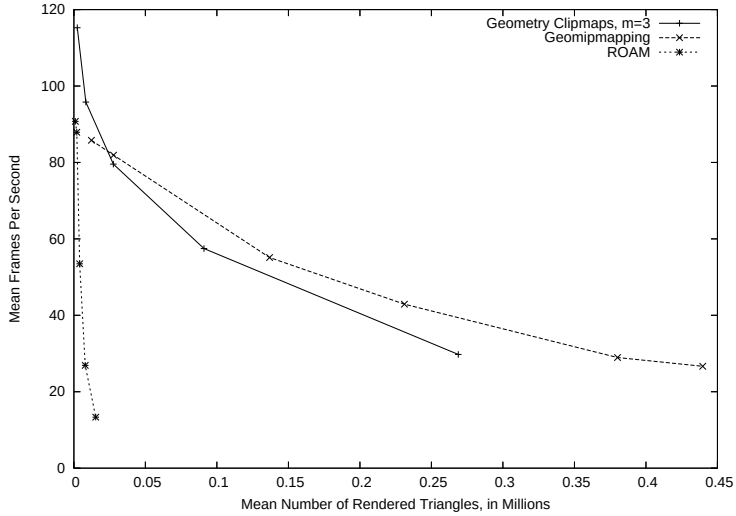
Each terrain-rendering algorithm, other than the brute force approach, can vary the number of triangles they use to render the terrain. The user can configure the number of levels m, or the size of the clipmap n with the geometry clipmaps algorithm. We varied the number of clipmap levels from $m = 2$ to 4, and roughly doubled the clipmap size each time, keeping the value odd. More specifically, the clipmap size was set to $n = 2^i + 1$ for $i = 5$ to 9. For ROAM, the number of triangles was varied by altering the value of the split or merge threshold, and for geomipmapping, the thresholds for each level of detail were changed.

We measured the number of triangles in two ways: the number of triangles actually rendered (that is, those in the view frustum) and the number of triangles that could potentially be rendered (all the triangles in the triangulation). Each approach, other than brute force, implemented some form of view-frustum culling. Even for the brute-force approach, the triangles that are not in view will be culled during the OpenGL transformation pipeline. Each triangle is transformed to the canonical view volume, and triangles that are outside this view volume will not be rendered. For this reason, we considered the number of triangles in the brute force approach to be the number in the triangulation.

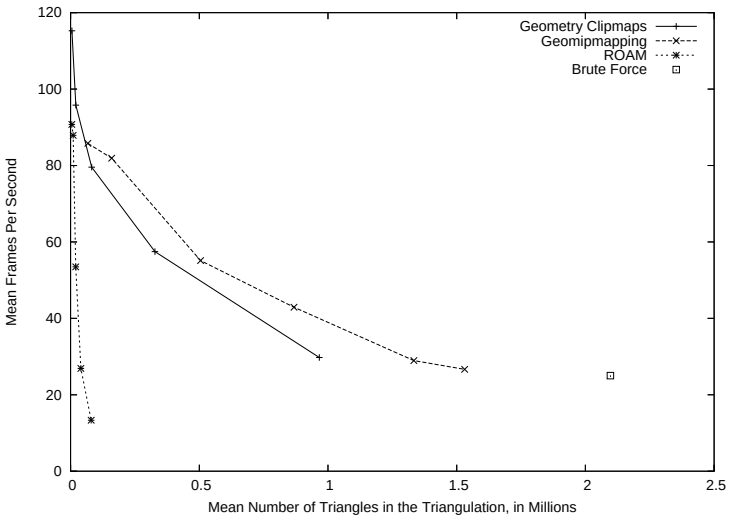
4.1.2 Results

The frame rates the algorithms achieved as the number of triangles was varied is shown in Figure 4.1. Figure 4.1(a) demonstrates how many frames were rendered per second (FPS) as the number of triangles rendered changes, and Figure 4.1(b) shows the performance as the number of triangles in each triangulation varied. Only the frame rates for geometry clipmaps with three levels are shown.

As evidenced by the graphs, geomipmapping was capable of rendering triangles at the greatest rate, closely followed by geometry clipmaps. The frame rates for all algorithms diminished as the number of triangles increased, at a decreasing rate. The behaviour of the frame rates as the number of rendered triangles, or possible triangles, varied did not differ substantially, indicating the algorithms had similar proportions of triangles in view.



(a) Frame rates as the number of triangles *rendered* varies



(b) Frame rates as the number of triangles in each triangulation varies

Figure 4.1: Performance of each algorithm in an automatic fly-through of central Hawaii.

The brute force approach rendered the entire terrain, at maximum detail, at 25FPS. This represents 52.5 million triangles per second, but a large proportion of the triangles would be culled within the OpenGL pipeline, inflating the supposed throughput. However, this can be considered an upper bound on the possible throughput for the graphics hardware, since a display list renders the triangles most efficiently.

Geomipmapping achieved the highest rendering rate of the algorithms, when 25,000 triangles or more were rendered. Rendering throughput increased as the thresholds were lowered, up to 11.7 million triangles per second when 440,000 triangles were rendered per frame. Geometry clipmaps were close behind, rendering at most ten frames fewer per second. The rendering throughput was greatest when $n = 257$: 2.2 million triangles were rendered per second on average at that size. ROAM only achieved reasonable frame rates when rendering a small number of triangles—when 12,000 triangles, or more, were rendered, the frame rate dropped below 20FPS. The rendering rate peaked at 0.2 million triangles per second, when 20,000 triangles were in the triangulation.

The algorithms rendered a similar proportion of triangles in the triangulation after view-frustum culling. Geometry clipmaps rendered an overall average of 36% of the triangles in the triangulation for $m = 2$, 3 or 4, geomipmapping averaged 24% and ROAM averaged 20%. However, the percentage of rendered triangles with geometry clipmaps decreased as the number of levels increased, with 40%, 35% and 33% for $m = 2$, 3 and 4 respectively. This is because, when the camera is pointing downwards towards the terrain, more of the finer levels are visible than the coarser levels. Increasing m just results in coarser levels being added, that will have a smaller percentage of rendered triangles.

4.2 Experiment Two

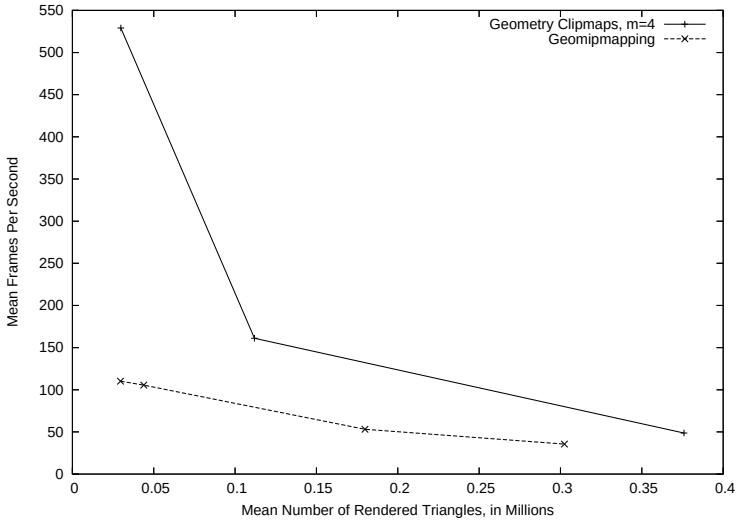
The aim of the second experiment was to find if geomipmapping was still able to render triangles more efficiently than geometry clipmaps when executed on a higher-end system.

The previous experiment was run on a NVIDIA GeForce FX 5200 graphics card, released in early 2003. However, it was released as an entry-level card, and has been observed to perform worse than a GeForce 4 MX, released a year earlier. Furthermore, the 5200 performs notoriously badly with vertex or pixel shaders, and Shader Model 2.0 in general. For this experiment, a GeForce 6 Series card was used, which performs much better with Shader Model 2.0, and thus is capable of considerably improved performance.

4.2.1 Method

For this experiment we compared the two best-performing algorithms from the first experiment: geometry clipmaps and geomipmapping.

The algorithms were executed on a computer with a 2.8GHz Pentium 4 processor supporting hyper-threading, with 1GB of memory, a 1MB cache, and a high-end graphics card—an NVIDIA 6800 with 256MB of memory, using a 16x PCI-



express port. This computer ran Windows XP, not Linux, so Windows-compatible versions of the algorithms needed to be used. Since geometry clipmaps used cross-platform OpenGL, minimal changes were required. Only OpenGL 1.1 was supported, so the call to `glDrawRangeElements()` had to be replaced with the less efficient `glDrawElements()` [26]. Polack's original geomipmapping algorithm ran on Windows, so this was modified to support the same DEM heightmap, automatic camera movement and frame-rate and number-of-triangle calculations as the geometry clipmaps algorithm.

A larger heightmap was also used, to allow each algorithm to render more triangles. We again used 1-degree DEMs from USGS, but pieced together four adjacent regions in a two-by-two grid. The geographical region was to the west of Mount Whitney in California, made up from the DEMs of Walker Lake and Mariposa, east and west. The array of DEMs gave a 2402×2402 region, of which a 2401×2401 portion was rendered.

Clipmap sizes of 129, 257 and 513 were trialled, with the number of clipmap levels varying from 3 to at most 6, depending on the clipmap size, such that equation 3.5 was satisfied. Again, each fly-through was repeated three times, and the average frame rate taken. Other aspects of the analysis were not changed from the first experiment.

4.2.2 Results

The performance of the algorithms is shown in Figure 4.2.2. Geometry clipmaps achieved markedly higher frame rates than geometric mipmapping.

	Mean frame rate (FPS)		
m	$n = 129$	$n = 257$	$n = 513$
3	588.3	182.8	49.0
4	529.2	161.1	48.6
5	507.4	155.8	—
6	493.9	—	—

Table 4.1: The effect of clipmap size and the number of clipmap levels on frame rates for a geometry clipmap.

We found that geometry clipmaps achieved its maximum rendering throughput when there were five levels, with $n = 221$. With this configuration, it rendered, on average, 22 million triangles per second.

Table 4.1 presents how the frame rate was affected by the number of levels or clipmap size. Adding more clipmap levels did not reduce the performance significantly, but increasing the clipmap size did.

4.3 Discussion

Results from our analyses suggest that the geometry clipmaps algorithm is the most efficient on recent graphics hardware, in particular a NVIDIA GeForce 6800, but the geomipmapping algorithm performed best on an older GeForce FX 5200. The ROAM algorithm had the lowest rendering throughput.

However, ROAM is not designed to maximise the number of triangles that can be rendered. Initially, it intended to reduce the number of triangles required to accurately represent a terrain, so a graphics card that was not capable of rendering many triangles at a high frame rate could still be used to render high-quality terrains. We observed that, although it did generate a triangulation that approximated the terrain well, it was inefficient if used to render a large number triangles, due to the CPU computations required for each triangle. This became progressively worse as the number of triangles increased, explaining the rapid drop in performance as the threshold was raised. We also noticed that ROAM consumed more memory than any other algorithm, in maintaining the triangles, the relationships between the triangles, the triangles metrics, and the priority queues used to determine where more or less detail was required.

Geomipmapping was surprisingly efficient, especially on the lower-end graphics hardware. That poses the question: what aspects of geomipmapping make it more efficient the geometry clipmaps on older graphics cards?

One possible reason is the time spent in the "update"[1] process of each algorithm. The update process for geomipmapping is very simple, requiring only the distance of each triangle from the viewpoint be calculated. Although geometry clipmaps support incremental updates, $O(n)$ updates still need to be performed. Additionally, the indices array needs to be recalculated each frame. In an informal trial, we found that geometry

[1] The *update* process is a common name for the procedure that updates the terrain after a change in view.

clipmaps had a greater rendering throughput when the view position did not change. This supports our theory that the updates might be more costly for geometry clipmaps than geomipmapping. However, the efficiency of the update process for a geometry clipmap can be improved using vertex textures in a GPU-based implementation.

Unlike geometry clipmaps, our implementation of geomipmapping did not store height values on the graphics card, even though the algorithm supports this. Originally, we thought this would result in an advantage to geometry clipmaps, but this may not have been the case when using the GeForce FX 5200. This graphics card's implementation of Shader Language 2.0 is notoriously bad, and it may also have an inefficient implementation of the vertex buffer object extension. Furthermore, the geometry clipmaps algorithm used a vertex shader that is likely to have resulted in a drop in frame rate with this graphics card.

Geometry clipmaps dramatically improved in performance when using the GeForce 6800, indicating the GPU was used effectively. The PCI express interface allowed vertices to be quickly uploaded to the vertex buffers on the graphics card, and the vertex shader was executed at low cost. Geomipmapping did not fare so well, but would perform better if vertex buffers were used.

Each algorithm had a similar percentage of triangles in the triangulation visible, as evidenced by the similarity of Figures 4.1(a) and 4.1(b). This is to be expected as none of the algorithms' LOD metrics took into account the viewer's orientation. Use of such a metric could increase the proportion of visible triangles, but may not be worth the extra computations.

The results of our analysis have implications for developers of applications that use terrain-rendering algorithms. The appropriate algorithm depends primarily on what hardware will be used. We suggest that when using a graphics card from the last five years or so, a GPU-based approach will be most effective. If the hardware is in the low- to mid-range, geomipmapping may be most appropriate. If more recent, and increasingly as time goes by and hardware improves, geometry clipmaps is the best choice.

Chapter 5

Conclusion

In summary, we implemented the geometry clipmaps algorithm, and compared it with three other terrain-rendering algorithms: ROAM, geomipmapping and the brute force approach.

We described how the finer details of the algorithm are implemented. Methods to incrementally update the toroidal array, and determine how to build a surface from the toroidal array, were presented. We specified some constraints required to prevent difficulties as the terrain nears the edge of a finite heightmap: the minimum and maximum possible extents at each level, and the maximum number of levels. Lastly, we presented how the render region could be cropped to the minimum required size to appear in a given view-frustum.

In our comparative analysis, we were surprised to find that geometric mipmaps outperformed geometry clipmaps on our first system, that used a GeForce FX 5200. However, when we compared the performance of geometry clipmaps with geomipmapping on a more recent graphics card, a GeForce 6800, geometry clipmaps were substantially more efficient. We also noticed that hierarchical techniques, in particular ROAM, are capable of generating accurate representations of a terrain using considerably fewer triangles, but the CPU computations required for each triangle mean they are not worthwhile on recent graphics hardware. Techniques that improve these algorithms by rendering clusters, in place of a single triangle, may result in greater rendering throughput, but a number of CPU computations will still be required. We suspect that such techniques will be inferior to geomipmapping or geometry clipmaps on modern hardware, which can be implemented almost completely on the GPU.

However, comparing terrain-rendering algorithms is a difficult task, and analysing their performance is an inexact science. The performance of each algorithm is strongly dependent on the system that is used. Different hardware may be more effective at different tasks. Implementations of the algorithms also may vary in their efficiency, depending on how well optimised they are. We tried to optimise our implementations of geometry clipmaps, ROAM and the brute force approach, as much as possible, but may not have been successful.

Nevertheless, our analysis has been of value. We have found that geometry clipmaps are a very efficient technique, if a high-performance graphics card is used that

can take advantage of the algorithm's strengths. Geomipmapping, a largely overlooked technique, demonstrated it has a lot of potential, outperforming geometry clipmaps on lower-end graphics cards. Incremental updates can still slow the geometry clipmaps algorithm down, but can be avoided by using vertex textures. In our automatic fly-through of the terrain, 36% of the possible triangles in a geometry clipmap were rendered, a number that would increase if the grids focussed their detail in front of the viewpoint, rather than right on it.

5.1 Future Work

There are numerous avenues for future study relating to geometry clipmaps.

5.1.1 Grid Structure

A problem with viewer-centred grids is that to maintain pixel-sized triangles as the field of view narrows, the clipmap size must grow. Losasso and Hoppe noted that dynamically adapting the location and size of the clipmap region could prevent this problem, but viewer-centred grids allow the viewer to rotate quickly [16]. However, for some applications, rotation is not so common, and adapting the position of the grids' centre might be an effective solution.

Informally, the number of triangles from finer levels that are visible could be maximised by centering the grids just in front of the viewer position. One way this might be achieved is by centering the lth clipmap level $\frac{ng_l}{4}$ units in front of the viewpoint, based on the view direction. Each level would have a different centre, but the grid width would remain at least 2 for $n \geq 8$. An alternative would be to have an adaptive centre, that moves as the field of view changes. Either way would result in a greater proportion of triangles being visible, assuming that it is more common to look across the terrain, than to have a birds-eye view.

5.1.2 GPU-complete Implementation

Implementations of terrain-rendering algorithms have been tending towards performing all computations on the GPU. In GPU Gems 2 [19], Asirvatham and Hoppe outlined an implementation where decompression of the heightmap is the only operation still performed on the CPU. A method to decompress the terrain geometry on the GPU is yet to be found.

5.1.3 Run-time Terrain Modification

For many applications, terrains are not static objects, but can change over time. A rendering of such a terrain needs to reflect such a change. An implementation of geometry clipmaps, that uses geometry compression, would make such a task difficult. The terrain pyramid would need to be updated whenever the terrain geometry changes, requiring a region of compressed data, at each level, to be recalculated. The clipmap itself may also need to be updated.

Bibliography

[1] BLOW, J. Terrain rendering at high levels of detail. In *Game Developers Conference Proceedings* (2000).

[2] CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. Planet-sized batched dynamic adaptive meshes (p-bdam). In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (Washington, DC, USA, 2003), IEEE Computer Society, p. 20.

[3] COHEN-OR, D., AND LEVANONI, Y. Temporal continuity of levels of detail in Delaunay triangulated terrain. In *IEEE Visualization '96 Conference Proceedings* (1996), pp. 37–42.

[4] DE BOER, W. Fast terrain rendering using geometrical mipmapping. Unpublished paper, available at http://www.flipcode.com/articles/article_geomipmaps.pdf, 2000.

[5] DUCHAINEAU, M. Roam algorithm version 2.0 — work in progress. http://www.cognigraph.com/ROAM_homepage/ROAM2/.

[6] DUCHAINEAU, M., WOLINSKY, M., SIGETI, D., MILLER, M., ALDRICH, C., AND MINEEV-WEINSTEIN, M. ROAMing terrain: Real-time optimally adapting meshes. In *IEEE Visualization '97 Conference Proceedings* (1997), pp. 81–88.

[7] FAN, M., TANG, M., AND DONG, J. A review of real-time terrain rendering techniques. In *The 8th International Conference on Computer Supported Cooperative Work in Design Proceedings* (2003), pp. 685–691.

[8] FERNANDO, R., AND KILGARD, M. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional, 2003.

[9] GERASIMOV, P., FERNANDO, R., AND GREEN, S. *Shader Model 3.0: Using Vertex Textures*. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, June 2004. Available at http://developer.nvidia.com/object/using_vertex_textures.html.

[10] GREENE, N. *Graphics Gems IV*. Heckbert, 1994, ch. Detecting Intersection of a Rectangular Solid and a Convex Polyhedron, pp. 74–82.

[11] HAKL, H., AND ZIJL, L. V. Diamond terrain algorithm: Continuous levels of detail for height fields. *South African Computer Journal* (2002).

[12] HILL, D. An efficient, hardware-accelerated, level-of-detail rendering technique for large terrains. Master's thesis, University of Toronto, 2002.

[13] HOPPE, H. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization '98 Conference Proceedings* (1998), pp. 35–42.

[14] LEVENBERG, J. Fast view-dependent level-of-detail rendering using cached geometry. In *IEEE Visualization '02 Conference Proceedings* (2002).

[15] LINDSTROM, P., KOLLER, D., RIBARSKY, W., HODGES, L., FAUST, N., AND TURNER, G. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), pp. 109–118.

[16] LOSASSO, F., AND HOPPE, H. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics* (2004), 769–776.

[17] MCNALLY, S. The Tread Marks engine. In *Game Developers Conference Proceedings* (2000).

[18] NVIDIA CORPORATION. *Using Vertex Buffer Objects*. 2701 San Tomas Expressway, Santa Clara, CA 95050, October 2003.

[19] PHARR, M., Ed. *GPU Gems 2*. Addison-Wesley, 2005.

[20] POLACK, T. *Focus on 3D Terrain Programming*. Premier Press, 2003.

[21] POMERANZ, A. ROAM using surface triangle clusters (RUSTiC). Master's thesis, University of California at Davis, 1998.

[22] RÖTTGER, S., HEIDRICH, W., SLUSALLEK, P., AND SEIDEL, H. P. Real-time generation of continuous levels of detail for height fields. In *Proceeings of the 6th International Conference in Central Europe on Computer Graphics and Visualization* (1997), pp. 315–322.

[23] TANNER, C. The clipmap: a virtual mipmap. In *Proceedings of the 25th annual conference on Computer Graphics and interactive techniques* (1998), pp. 151–158.

[24] ULRICH, T. Rendering massive terrains using chunked level of detail control. Draft, from "Super-size it! Scaling up to Massive Virtual Worlds" course at SIGGRAPH '02, 2002.

[25] WAGNER, D. *ShaderX2: Shader Programming Tips & Tricks with DirectX 9*. Wordware Publishing, 2004.

[26] WOO, M., NEIDER, J., DAVIS, T., AND SHREINER, D. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*, third ed. Addison Wesley, 1999.