REAL-TIME TERRAIN RENDERING WITH INCREMENTAL LOADING FOR INTERACTIVE TERRAIN MODELLING

Simon van den Hurk and Wallace Yuen and Burkhard C. Wünsche
Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand
{svan071, wyue013}@aucklanduni.ac.nz, burkhard@cs.auckland.ac.nz

Keywords: terrain rendering, terrain modelling, sketch-based interfaces, GPU-acceleration

Abstract: Terrains are an essential part of virtual environments in computer games, movies, visual impact studies, architecture, urban design and archaeology. In order to render large terrains in real-time multi-resolution representations are used. The underlying data structures are usually static and do not allow modification of the terrain in real-time. In recent years an increasing number of applications have been developed requiring interactive terrain modelling. In this paper we describe a real-time geometric clipmapping terrain rendering technique for large terrains which allows incremental updates of the underlying data structure. We have combined the method with an interactive sketch-based terrain modelling technique. The clipmap data structure is updated during runtime to synchronise the terrain visualization with changes to the underlying digital elevation map. Tests and examples demonstrate the advantages of our method over traditional approaches. Disadvantages and limitations are discussed and potential solutions for future work are presented.

1 Introduction

Terrains are an essential part of virtual environments in computer games, movies, visual impact studies, architecture, urban design and archaeology (Keymer et al., 2009).

Terrain data sets can be very big and in order to achieve real-time rendering multi-resolution representations are necessary. A popular representation is to use a regular grid of height values, also called Digital Elevation Map (DEM), where grid points are connected by triangles. A multi-resolution representation represents the same height field using different layers with a decreasing number of triangles with increasing size. When rendering the terrain regions close to the view point are represented in high resolution, and regions far away in low resolution, so that the size of triangles projected onto the view plane is approximately constant for the entire terrain.

In most applications the underlying multi-resolution terrain data structures are static and do not allow modification of the terrain in real-time. In recent years an increasing number of applications have been developed requiring interactive terrain modelling, e.g. in architecture, geology, and archaeology (Keymer et al., 2009). We are especially interested in interactive terrain modelling for use in our LifeSketch project, a sketch-based tool for rapid prototyping of animated 3D environments (Yang and Wünsche, 2010; Xie and Wünsche, 2010; Guan and Wünsche, 2011; Olsen et al., 2011).

In this paper we describe a real-time geometric clipmapping terrain rendering technique for large terrains which allows incremental updates of the underlying data structure (Losasso and Hoppe, 2004).

Section 2 reviews existing terrain rendering techniques. Section 3 introduces the geometric clipmapping algorithm, which forms the foundation of our proposed technique. Section 4 presents our solution including data structures, methods for interactively updating the terrain during modelling, and optimizations for increasing the rendering speed. Section 5 evaluates the performance of our solution and discusses advantages and disadvantages of it. We conclude this paper in section 6 and give a short overview of future work in section 7.
2 Literature Review

2.1 Terrain Rendering

Terrain rendering techniques have been extensively studied since the mid 1990s and can be categorised into methods using tile based data structures, quadtree based triangle hierarchies and out-of-core approaches such as the geometry clipmaps approach used in this project.

Initial work by Lindstrom and Pascucci improved computation time by reducing the level of detail in drawn triangles as the terrain moves further into the distance of the current viewpoint (Lindstrom et al., 1995). An alternative approach by Duchaineau et al. named ROAM uses priority queues to perform merge and split operations upon the base terrain data (Duchaineau et al., 1997). Such a technique requires a large amount of recalculation as the user rotates upon the spot and the triangulation of the terrain must be recalculated.

De Boer recognised that rendering speed can be improved dramatically by using graphics hardware, which requires different data structures and algorithms. The resulting Geometrical MipMapping (de Boer, 2000), divides the terrain in multiple chunks of different levels of detail. These chunks are then updated as the user moves throughout the simulation. Additional work involving out of core terrain rendering (Schneider and Westermann, 2006) uses tiles and a nested mesh hierarchy to avoid mesh re-triangulation.

The geometry clipmaps technique borrows from the concept of texture clipmaps (Tanner et al., 1998) to create layers with increasing level-of-detail (Losasso and Hoppe, 2004). Further work by Lindstrom et al. emphasizes the efficiency gained through carefully designed data structures rather than focusing on modification of a mesh as done for the ROAM algorithm (Lindstrom and Pascucci, 2001). The authors describe a general approach to out of core rendering to provide a framework for future work (Lindstrom and Pascucci, 2002).

Specific work with regards to culling is undertaken by Hesse et al. who provide an analysis of different approaches using the ROAM algorithm as a basis (Hesse and Gavrilova, 2003).

Work by Scheib et al. uses a quadtree subdivision approach which also includes modifications allowing level of detail control as well as data approximation while maintaining a low error metric.

A comprehensive survey of terrain rendering techniques is presented in (Pajarola and Gobbetti, 2007).

2.2 Terrain Editing

Early work on terrain editing includes multi-resolution detail patches devised by (He et al., 2002). (Atlan and Garland, 2006) modify the terrain in real-time by specifying editing strokes, in which quadtree hierarchy is used to represent the heightmap for multi-resolution editing.

(Bhattacharjee et al., 2008) improve performance by utilising the GPU for rendering and editing the terrain simultaneously. The authors use a fragment shader to operate on each height value, and every time the terrain undergoes deformation or modification, the parameters of the actual process are parsed into the shader, which could be a result of simulation of terrain dynamics or direct editing performed by user using input devices such as mouse.

(Dan et al., 2009) discuss various ways of terrain editing, including geometric editing and texture editing. Similarly to (Bhattacharjee et al., 2008), they also modify the height point, but add natural looking variations by defining new height values using an outer and inner radius and a parameter determining the radius. We add to this work by providing an incremental update technique which enables interactive terrain modelling by minimizing data transfer and data structure updates in GPU memory.

3 Geometry Clipmaps

Geometry Clipmaps is a level-of-detail GPU-based terrain rendering technique. It uses multiple representations of the same terrain at different resolutions to increase the efficiency of the rendering process. The clipmaps data structure is broken into several layers, each of which contains a higher resolution than the layer below it. These layers are arranged by centering them about the viewpoint and then rendering them.

The terrain data is stored in a vertex buffer object and updated as the viewpoint moves throughout the simulation. This grid of data is stored using an offset into the vertex buffer object to allow for a toroidal access. By using a toroidal index the vertex buffer object is able to only update a single column or row rather than moving all rows or columns within the vertex buffer object to a new location.

The update process examines each level in the clipmap, starting from the lowest resolution and iterating to the highest. Each level uses the viewpoint of the camera to determine an active region of the clipmap. If this active region differs from the previous update’s active region, then the vertex buffer object is
updated, and the data will be synchronised with the terrain data that is stored within the RAM memory.

When the terrain is rendered, each clipmap layer renders a ring section within which the next clipmap layer is rendered. This creates a high resolution terrain surrounding the users viewpoint. The resolution decreases with increasing distance from the viewpoint. Figure 1 shows the same terrain rendered in two different resolutions.

The most important aspect of the clipmaps data structure is the way clipmap layers are blended to create smooth transitions between them. The vertex information of the terrain stored by each clipmap layer includes not only the x, y, and z coordinates of each point in the terrain, but also an additional channel containing the height of the parent layer at this point. During the render method, an alpha value is calculated to interpolate between the vertex height and the parent vertex height. The alpha value approaches one for the vertices that are closer to edge of the clipmap, therefore aligning the edge of the clipmap with the edge of its parent clipmap layer.

Figure 1: An example of the same terrain rendered using a low resolution (left) and a high resolution (right). In interactive applications the resolution of a section of the terrain will decrease with its distance to the current viewpoint.

4 Design

In order to interactively model terrains we need a technique with suitable data structures which can be modified to support interactive updates, and with as few constraints as possible when using it. After careful analysis we chose the Geometric clipmap algorithm.

A major advantage is that the algorithm performs well when viewed from a top down perspective, so that the rendering rate can be sustained while the detail of the terrain is rendered (Asirvatham and Hoppe, 2005), which is important during modelling to add and view terrain structures such as mountain ranges and rivers.

The algorithm also loads large sections of terrain, allowing for a fast rotation of the viewpoint without a large computational requirement. The algorithm merges well with the sampling technique that the sketch based input provides. The geometry clipmaps algorithm uses multiple levels of detail and the sketch based input can provide these different levels by sampling the contours at different resolutions. Real-time incremental updating of the terrain is achieved by computing small clipmap sections fitting into multiple resolution representations and as such minimizing the amount of data passed between the RAM and graphics card memory.

4.1 Data Structures

4.1.1 The TerrainSurface interface

The data for the surface of the terrain will be stored within the RAM memory during the runtime of the program. In order to access the terrain data and provide it to the clipmaps data structure, we wrote the TerrainSurface interface. Figure 2 shows a UML class with a listing of this interface.

The clipmaps algorithm requires different representations of the same terrain at different resolutions. The desired representations are obtained by calling the methods within this interface with a Dimension parameter. The terrain data is stored in RAM using a one-dimensional array of floats. This packed data is then accessed using offsets within the data. A data stride value is required to determine the byte offset between vertex and normal data. The interface describes the methods providing access to the different types of data required by the clipmaps algorithm:

- Vertex information.
- Normal Information.
- Parent Normal Information.
- Offsets to the position of the above information within the single dimensional array.

By using offsets into the one dimensional array the interface provides flexibility for implementations which already contain a set structure for the order of the data stored in the packed array.

4.1.2 Clipmaps classes

The implementation of the clipmaps algorithm is split into two separate classes: Clipmaps and Clipmap. The Clipmaps class provides the interaction with the clipmaps concept and is the class instantiated by the
end user. This class manages the updating of all levels of the clipmap data structure, as well as providing simple method calls to render the entire data structure.

The Clipmap class provides implementation for a single layer of the clipmaps data structure. Each layer of the clipmap is stored in a series of vertex buffer objects within the memory of the graphics card. This class provides core methods to create these vertex buffer objects, and to update them as the viewpoint is moved throughout the scene. It also provides methods to render the terrain and perform the frustum culling to improve the efficiency.

For this specific project there are some notable changes from the original geometry clipmaps implementation. The most important is that no compression is done upon the terrain data stored in RAM memory. This is because the compression algorithm used within the original paper is quite quite slow and updating the terrain and recompressing it would prevent interactive framrates.

Another modification is that the lowest level of the clipmap data structure has been changed to always render, regardless of the camera position. In the original implementation the entirety of the terrain is only rendered when the viewpoint is in the centre of the terrain. By enforcing the lowest level to always be drawn, the entire terrain is therefore always visible at the lowest resolution regardless of the viewpoint.

Lastly, this implementation allows non-square shaped clipmaps to be defined, so that the size of the clipmaps can be rectangular such as 256 x 512.

4.2 Incremental Updates

An interactive terrain modelling system must show any modifications to the terrain in real-time. The visual feedback will assist the users with editing the terrain. In order to achieve this, only small rectangular sections enclosing the modified regions are updated.

4.2.1 Clipmaps Section Update

Updates of the terrain data will change the information stored in RAM memory that is interacted with through the TerrainSurface interface. After a section of the terrain stored in RAM memory is updated, the corresponding data within the graphics card memory must also be updated. This functionality is provided through an UpdateSection method within the Clipmaps class. This method works by defining a rectangular region that is to be updated and a Dimension variable which specifies at which resolution the rectangular region is specified.

Algorithm 4.1: UpdateSection(x, y, width, height, updateDimension)

for i ← 0 to clipmapsStack.size − 1 do
    clipmap ← clipmapsStack[i]
    surfaceResolution ← clipmap.surfaceResolution
    if heights ← clipmap.surface.getHeight(surfaceResolution)
        rect ← getClipmapUpdateRegion(x, y, width, height, clipmap)
        clipmapRegion ← convertSurf UCSToClipmapCoordinates(rect)
        updateVB0f(clipmapRegion, heights, rect)

As shown in algorithm 4.1, the method iterates through all clipmap layers in the clipmap stack and determines the appropriate rectangular region to be updated by scaling the co-ordinates for each level depending on the surface resolution. The method then performs a standard update for this region, and lastly
synchronises the data on the graphics card with the newly updated terrain data in the RAM memory. By using this method, the amount of data required to be transferred between the RAM memory and the graphics card memory is reduced to a minimum, reducing rendering time and saving bandwidth for other applications.

4.3 Shaders

The clipmaps implementation requires specific shaders to correctly render the desired terrain. These shaders were written in the OpenGL Shading Language (GLSL).

4.3.1 Vertex Shader

The vertex shader takes two additional parameters viewCoord and activeRegionSize, which are the same for every rendered vertex. These parameters are both of type vec2 and declared uniform as they do not change between each rendered vertex. viewCoord specifies the position of the camera and is used to determine around which point the alpha blending is centered. The other parameter activeRegionSize specifies the width and height of the active region which is to be drawn. Combining these two variables, the shader is able to calculate the alpha value required to blend between the height value of this clipmap and its parent. This is done using the formula specified by (Losasso and Hoppe, 2004):

\[
\alpha_x = \min(\max(\frac{|x - v_{l|x}| - (v_{max} - v_{min} - w - 1)}{w}, 0), 1)
\]

This formula calculates the difference between the vertex position \( v \) and the position of the camera within this layer \( v_{l|x} \), and then subtracts half of the region size and the blend width \( w \). A blend width of 10 was found to produce suitable results, as lower blend width values tend to produce less smooth results between two clipmaps, and higher blend width values require unnecessarily more calculations. This calculated value is then divided by the blend width \( w \) and finally the result is clamped to be within a range of zero to one. A similar formula is also used for \( \alpha_y \), and the final alpha value is calculated as the maximum of \( \alpha_x \) and \( \alpha_y \). Using this calculated alpha value the blended height can now be determined. This is done using a linear interpolation between the height value of the current clipmap and the height value of the parent clipmap stored in the fourth channel of the position vertex \text{gl\_Position}. The formula to calculate the final height value is:

\[
\text{blendedHeight} = (1 - \alpha) \text{height}^l + \alpha \times \text{height}^{l+1}
\]

To avoid slight rounding errors with the final blended height the alpha value was rounded up to 1 if value was close to that number. This ensures the border vertices are completely rendered using the parent height and as such provide a seamless integration with the surrounding clipmap level.

The shaders also calculate the lighting and final colour of the pixel, which requires blending normals so that they correspond to the blended height values. Normal blending uses the \( \alpha \) value for blending heights in order to linearly interpolate between the normal of this layer and the normal of the parent layer:

\[
\text{blendedNormal} = (1 - \alpha) \text{normal}^l + \alpha \times \text{normal}^{l+1}
\]

This blended normal must then be normalized in order to ensure the lighting is correctly calculated. The debug version of the vertex shader replaces the final pixel colour with a colour representing the value of the alpha value used for blending. A screenshot of this debug vertex shader can be seen in figure 3.

Figure 3: The Debug version of the shader enables the debugging of alpha values. Red indicates the height of the clipmap, while green represents the use of a blended height. The blue strip uses only the parent height value to ensure there are no gaps between the different clipmap levels.

4.3.2 Fragment Shader

The fragment shader is used to calculate the final colour of the pixel fragment that is to be drawn to the screen. The intensity of the fragment must be calculated and then merged with the colour value from the texture. When calculating the intensity the normal must be normalized once again. This is because this normal value is a linear interpolation between two of the normals provided with vertices. The linear interpolation does not guarantee a normalized vector and so this normalization must be performed manually.
Combining these two values produces the final colour that is to be rendered to the screen. The debug version does not need to calculate the intensity as the fragment colour only depends on the colour representing the alpha value.

### 4.4 Rendering Optimisations

#### 4.4.1 Active Regions

The active region defines the section of the clipmap level that is to be rendered to the screen. The dimensions of this active region must lie within the clipmap. During the update method of the clipmap, the active region is recalculated if the position of the viewpoint has moved. To ensure that the rendering of the clipmap aligns with the parent layer, the active region must be enlarged so that its vertices are shared by the current and the parent layer. For example consider a clipmap with two layers, 256x256 and 512x512. If the viewpoint would create an active region starting at coordinates (501, 0) then this must be rounded to 500 as the point 500 on the high resolution layer aligns with the point 250 on the lower resolution layer. Similarly if the point lies outside the clipmap, (540, 0), then it must be clamped to (512,0) to lie within the clipmap.

#### 4.4.2 Clipmap Segments

The clipmap layer which defines the finest resolution is drawn as a single rectangular block using triangle strips. All other layers are a ring shape, with a hole in the centre where the next clipmap layer is drawn. In order to render this ring shape, the vertex data must be drawn as several rectangular segments. In the original geometry clipmaps paper Losasso and Hoppe use four segments (Losasso and Hoppe, 2004) as indicated in the left hand side of figure 4. Our implementation uses eight segments as illustrated in the right hand side of figure 4. The higher number of segments reduces computations during frustum culling.

#### 4.4.3 Frustum Culling

The frustum culling process reduces the amount of computation required in the rendering process. This is done by determining which sections of the clipmap need to be drawn given the current orientation and position of the viewpoint. Two approaches were tried in order to perform this frustum culling.

The first was to project the points of each of the eight segments in the clipmap onto a horizontal x-z plane. Then the points of the view frustum were also projected onto this plane. Following this, the points of each region were tested for containment within the oriented bounding box of the axis projected frustum. If at least one of the points was contained, then the region was considered necessary for rendering. This method proved to be the least successful due to the nature of the oriented bounding box. The calculated bounding box would often include many points within the region, which were not required for the rendering process. This over inclusion of points displaced any advantage that might have been gained by using the efficient containment detection provided by the bounding box.

The second implemented technique for frustum culling approximated the viewing frustum using six planes. The bounds of each region segment were then tested for containment within these six planes. This proved to be efficient as it only requires a simple distance calculation to determine upon which side the plane a point lies. Furthermore this method determines containment within the frustum far more accurately than using bounding boxes. If any of the segment points was found to be within the view frustum, then that segment of the clipmap is rendered.

![Figure 4: A clipmap layer divided into four (left) and eight (right) segments. The orange section is the next layer. The structure on the left is used in (Losasso and Hoppe, 2004), the structure of the right is used by us.](image-url)

![Figure 5: An aerial view of the clipmaps algorithm being run with the culled segments removed from the rendering process.](image-url)
4.4.4 Viewpoint-Based Culling for Interactive Modelling

In the original geometry clipmaps algorithm the high resolution clipmap levels surround the camera position. This allows a user to see high levels of detail in the terrain in their immediate vicinity. It also guarantees a high rendering speed should the user rotate rapidly or wish to quickly look in the opposite direction. The original motivation for this design comes from flight simulators where the user frequently looks out of varying viewports of the cockpit.

When providing high speed rendering for interactive terrain modelling, the user requirements vary slightly. The user is likely to view the terrain from a top down position and look down upon a section of the terrain that they are currently editing. The camera might not be fixed at a point of interest, but instead rotate around a point of interest which is represented in the highest resolution as indicated in figure 6.

Figure 6: A diagram displaying a camera which rotates around a point, which is the centre of a high resolution clipmap. An example of this concept is shown in figure 7. The high resolution terrain is centered about the position that the camera is facing. This allows the user to view a part of the terrain in full resolution.

The frustum culling that is performed by the clipmap levels to determine which segments should be rendered requires still the original position of the camera. Using the point upon which the camera rotates would not provide a correct rendering output.

It is important to note that this modification contains a flaw. Should the camera be sufficiently far away from the central rotation position and then tilt to a low angle such that the camera skims across the terrain with its viewpoint then the high resolution clipmaps will lie far away from the camera position and the low resolution terrain will be visible to the user. A possible method to avoid this situation would be to interpolate between the point of rotation and the camera position depending on the angle between the camera and the rotate position and the plane that the terrain resides upon.

5 Results

5.1 Terrain Rendering Performance

The performance of the clipmaps implementation was measured by testing multiple resolutions with two base sizes. These tests were run upon an ATI Radeon HD 4800 graphics card. In table 1 the program was run with the number of layers ranging from a single layer to five layers. Each layer has twice the number of vertices of the previous layer.

In table 2 the same graphics card is used to test the same data set with a starting resolution of 512x512. This increased resolution causes a major drop in the frame rate, as was expected due to the complete rendering of the lowest resolution regardless of the viewpoint.

5.2 Terrain Editing Performance

The performance of the incremental updates were measured by using multiple resolutions with varying
<table>
<thead>
<tr>
<th>#Layers</th>
<th>Resolution Range</th>
<th>Frames Per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>512x512</td>
<td>135 fps</td>
</tr>
<tr>
<td>2</td>
<td>512x512 - 1024x1024</td>
<td>70 fps</td>
</tr>
<tr>
<td>3</td>
<td>512x512 - 2048x2048</td>
<td>43 fps</td>
</tr>
<tr>
<td>4</td>
<td>512x512 - 4096x4096</td>
<td>31 fps</td>
</tr>
</tbody>
</table>

Table 2: Frame rates for varying resolutions when the base size is 512x512.

<table>
<thead>
<tr>
<th>Clipmap Update Size</th>
<th>Frames Per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>257 x 257 (entire clipmap)</td>
<td>38.63 fps</td>
</tr>
<tr>
<td>129 x 129</td>
<td>39.82 fps</td>
</tr>
<tr>
<td>65 x 65</td>
<td>42.67 fps</td>
</tr>
<tr>
<td>40 x 40</td>
<td>45.77 fps</td>
</tr>
<tr>
<td>20 x 20</td>
<td>55.95 fps</td>
</tr>
</tbody>
</table>

Table 3: Frame rates for differently sizedclipmaps update sections with clipmap size of 257x257 with 5 layers of resolution surfaces from 257x257 to 4097x4097.

clipmap size. These tests were run upon a Nvidia GeForce GT 330M 512mb graphics card. Each test changed the height of a 20 x 20 area of pixels in 60 fps using the glutIdleFunction to control the frequency. Tables 3–5 show the results of averaging 10 tests for each of the specified parameters. Note that each test starts with updating the entire terrain into each clipmap, and then slowly reduces the amount of data to be updated. We compare the performance between incremental to no incremental update, and also discuss how the performance changes when the size of the terrain data that needs to be updated changes.

Table 3 shows that processing the full active region of the clipmap in the graphics card results in a frame rate of 38.63 fps. By reducing the update region this can be improved to 55.95 fps, which is a 44.8% improvement.

Table 4 shows the results for an increased clipmap size. The results demonstrate that the increased clipmap size reduces the efficiency of the rendering process. The frame rates are reduced compared to the previous experiment. Processing the full active region of the clipmap in the graphics card results in a frame rate of 29.80 fps. By reducing the update region this can be improved to 41.18 fps, which is a 38.19% improvement.

Table 4: Frame rates for differently sized clipmaps update sections with clipmap size of 257x513 with 5 layers of resolution surfaces from 257x257 to 4097x4097.

For the last test we reduced the number of clipmap layers to three, including 513x513, 1025x1025, and 2049x2049, and the size of the clipmap is increased to 513x513. Updating the entire active region of the clipmap results in a frame rate of 29.96 fps. Changing the update region of the clipmap to the exact area of the updated pixels (20x20), increases the frame rate to around 42.83 fps, which is a 42.96% increase from re-syncing the entire active region of the clipmap.

As various tests showed, the performance of the rendering process increases when only relevant parts of the clipmaps sections are updated and synchronized with the graphics card. This is down to the fact that the number of bytes transferred between the RAM memory and the graphics memory is reduced greatly when the update region is reduced to the area whose height values are modified.

The number of bytes is dependent upon the update width and height of the clipmap region, which can be set to a lower dimension to increase the overall performance as the tests have shown, using various sizes of the active region of clipmaps. Datastride is the number of bytes storing the x, y, z co-ordinates of vertices, and their respective normal. Furthermore, it also stores a parent height value for a vertex, and the parent pixel’s normal for interpolation. Therefore, the total of bytes involved in the datastride is 10 bytes, which is then multiplied by four since the data is stored as floats. From this equation, we can find the total number of bytes transferred from the RAM memory to the graphics card memory.

To explain how the amount of data transfer affects the overall rendering process performance, refer to the first test in table 1. The total number of bytes transferred for a full clipmap is 2.52 MByte, and since in our case a terrain has 5 clipmaps this results in 12.5 MByte of data per frame. Furthermore, because the updates are set to execute in 60 fps in all the tests we performed, the necessary data transfer is 750 MByte/s. This significantly reduces rendering performance and leaves little bandwidth for other tasks. In contrast, if only a subsection of the active region of
the clipmap is used, then the amount of data transfer from the RAM memory to the graphics memory can be reduced dramatically. In the case of only the actual areas of heights changed, which is 20x20, the total amount of data that gets transferred per frame is 78.13 KByte. Therefore, the transfer rate becomes 4.58 MByte/s, which explains why the rendering process performance increases after reducing the area of the clipmap update section.

With this result, we have demonstrated how using subsections of the active region of clipmaps reduces the bottleneck for updating the terrain in real-time. This makes the modified geometry clipmaps algorithm suitable for interactive terrain rendering. We have used the algorithm for sketch-based terrain modelling, e.g., the modelling of rivers and lakes using sketch input as demonstrated in figures 8 and 9.

5.3 Suitability for Sketch Based Input

5.3.1 Advantages

Our algorithm removes the restriction of having fixed sized clipmaps, thus allowing the shape of the clipmaps to be rectangular, and allowing the difference in resolution between clipmaps to be a multiple of two, rather than exactly double. By removing the restriction of the clipmaps being square we allow for the creation of terrain in a rectangular grid, which may be desired by the user. Furthermore the flexibility of the resolution allows for differences in clipmap size, which can aid the design of the framework in ensuring a smooth frame rate while maintaining a high level of detail.

This paper also reduces the size of the update section in the active region of the clipmaps, where its results were tested in the result section. As the results showed, this approach allows real time modification done to the terrain surface.

5.3.2 Disadvantages

A current disadvantage that may arise is if the time taken to calculate an updated region is too large, the terrain in the RAM will be partially updated and not synchronised with the graphics card. If this is the case and the user moves the viewpoint towards this partially updated region, then the clipmap class will begin to synchronise the data with the partially updated region and errors could occur. If this situation does occur there are a number of possible solutions that could be used.

Firstly the updated region could be broken into several smaller parts and updated over a number of time steps. Secondly the viewpoint control could be restricted until the updated region is calculated. Note that only the position of the camera needs to be restricted and the rotation and orientation of the camera could still move about freely. Finally the update could be broken into two parts, firstly updating the positions and synchronising this data, then calculating the normals secondly. The normal calculation will most likely be the most computationally time consuming of the data that needs to be updated and delaying it should allow for a faster update process.

5.4 Limitations

There is one main limitation that the clipmaps implementation currently has. By using a one dimensional array the clipmaps implementation is limited by the maximum size that a one dimensional array can support in the compiler, which is
0x7FFFFFFF(2147483647) bytes. The implementation uses 10 floats, each requiring four bytes, for each terrain vertex: three to represent position, one for parent height, three for the normal and three for the parent normals. Hence the maximum terrain size is currently 53687091 data points in a one dimensional array, which is equivalent to a clipmap level size of 7327x7327 vertices.

The number of data points can be increased by removing a channel from the normals, and then assuming this channel is one. For example, the z axis of the normal can be discarded, then the shader could be written to assume a z value of one. This would force further calculation into determining the values of the normals, which is already a slower part of the process.

It is important to note that a larger terrain could simply be represented by multiple instances of the Clipmaps class. Because the lowest resolution of the clipmap is always drawn, multiple clipmaps classes placed next to each other would render correctly. This approach would increase the number of calls to the graphics card when the viewpoint lies near the edges of multiple clipmap stacks, but warrants investigation if extremely large datasets are required for the application. Such an approach would then cause the size of the memory on the graphics card to be the bottleneck as well and a combination with an out-of-core terrain rendering techniques would be necessary to resolve these limitations.

6 Conclusion

An implementation of the geometry clipmaps data structure has been developed, which allows incremental updates to the terrain at run-time in order to enable interactive editing of large terrains. The update regions can be an arbitrarily sized making the technique suitable for applications requiring constant small changes, such as sketching a river, and for large changes, such as inserting a new mountain range. We extended the underlying Geometry Clipmap approach to allow arbitrary view points, including a birds-eye perspective, which is useful for terrain editing. Our results demonstrate that terrains can be edited and rendered at interactive frame rates in high resolutions. The main limitation is the maximum allowed terrain size due to compiler restrictions and GPU memory limitations. This could be overcome by employing concepts from out-of-core terrain rendering techniques.

7 Future Work

Currently we texture terrains with a 1D texture for illustration purposes, which is sufficient in sketch-based modelling applications where no texture image exist. In more general applications we would like to use 2D textures, including multi-texturing in order to combine large-scale texture variations with terrain details. Such an extension would require the following steps: Firstly the TerrainSurface interface would require additional methods which describe the offsets into the float data which point to the coordinate information about the two dimensional texture. The clipmap code would then require modification to pass the texture coordinate information to the shaders. These shadars would then have to index the 2D texture maps and blend the texture values appropriately.

REFERENCES


