Rendering Mars Environment using VBO based Geometry Clipmapping

Eve Powell
empowell@uncc.edu
Ph. D. Student
UNC Charlotte

Kirk Fernandez
24kirk@gmail.com
MS student
UNC Charlotte

Abstract

In this paper we present our attempts to use Geometric clipmapping and compression techniques to create a usable Mars simulation environment. The Mars height data was pulled from the MOLA dataset and took up 2 GB of storage. In this project, we looked at various ways to cleverly reduce the size of the data without introducing too much error and then looked at ways to incorporate compression techniques into the Geometry clipmapping solution. Geometry clipmapping proved to be more effective that rendering through OpenGL and using the VBO to render all of the data at once, even with the VBO being constantly updated in the clipmapping solution.

Keywords: Level of Detail control, LOD, simulation, visualization, terrain compression

Introduction

One of the biggest difficulties of terrain rendering is displaying the terrain at real time frame rates. Rendering large amounts of data at once will negatively impact frame rates, sometimes to the point where the visualization cannot be used for interactive application i.e. Simulations and Games. Among the most critical properties of terrain rendering are: quick adaptive triangulation, limited geometry error and error management, and proper level of detailing. All of these properties are addressing in our implementation of geometry clipmapping.

Based on some observations of terrain rendering solutions presented in class, we took the most interest in geometry clipmapping because of the promise of efficient and effective level of detail it had over its counterparts. Geometry clipmapping is based on the idea of storing a terrain pyramid, or nested rectangular regions, in a cache in a way that the nested information is centered around the viewer [Losasso 1994]. These grids represent filtered versions of the terrain at power of two resolutions. In our case, as was the initial case of geometry clipmapping, the information is stored as vertex buffers in Graphics card memory.

In this paper, we present the following contributions:

- An implementation of the geometry clipmapping algorithm to render a Mars simulation. We will present our approach at this implementation, the successes and failures of the implementation, and a comparison of our results with other clipmapping papers.
- An implementation of terrain compression. We compressed the 2GB data of Mars to fewer than 200 MB. We discuss ways that terrain compression was implemented into our approach.
- An analysis of terrain rendering using clipmapping against more canon approaches to clipmapping, primarily brute force methods.

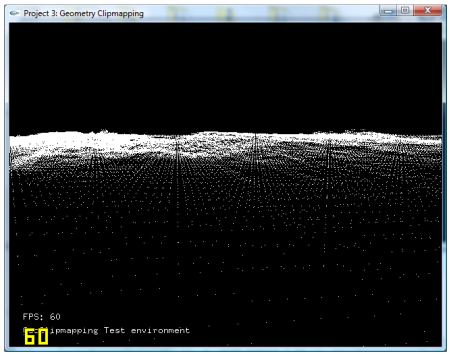

Figure 1 : Geometry clipmap rendering of Mars terrain, using our clipmapping solution.

Related Work

There are currently many approaches to terrain rendering of large data sets.

Brute-force: The easiest approach would be a brute force method, in which the programmer would create a brute-force triangulation of size n x n, n being the resolution of the heightmap data set. The vertices of the resulting triangulation would then be adjusted in the y direction of each corresponding element in the height map [Bretell 2005]. The problem with such a method is that having all of the terrain reside in memory at once will greatly reduce the performance of the simulation [Losasso 1994]. Furthermore, this method becomes impossible as the terrain data approaches and surpasses the size of memory. Typically, this method will not be attempted for rendering a significantly large terrain.

Quadtree-based Terrain synthesis: In a tree-like data structure, the terrain is partitioned into square tiled blocks that are triangulated at different resolutions [Losasso 94]. These types of algorithms can be efficient with respect to triangulation and level of detailing. The primary drawback to quad tree implementations, however, would be the increased size of the resulting rendered mesh [Parajola 2002]. Geometric Clipmapping outperforms Quadtrees in this regard.

TIN aka triangulated irregular networks: In opposition to brute method approaches, triangulated irregular meshes are typically products of irregular nodes of three dimensional coordinates arranged in a network of non-overlapping triangles[Cohen-Or 1996]. TIN generation requires methods of cleverly choosing points to input into the network in order to best capture the topographical data. They are usually based on the Delaunay triangulation. Lossaso and Hoppes observe that while these meshes tend to produce the best approximation for a given number of faces, they require the tracking of alternate data, which could prove problematic [Lossaso 1994].

Implementation

In this section we shall go over the implementation and design of the GeoClipmapping application. Development was done as a C++ application. We used GLUT and Glee libraries on top of OpenGL. Due to time constraints, the team was split to work on separate parts of the application, to limit the learning curve on the technically challenging parts of the application.

The application required the generation of a simulation engine, a compression\decompression engine, and a geometry clipmap data structure. Each of these components will be explained in the following sections.

Implementing Clipmapping

About the Pyramid

With geometry clipmapping, terrain pyramids are typically described as “windows” into the elevation. These hollowed frames are positioned dependant to the user-controlled parameters. For our purposes, this parameter would be the user viewpoint. Based on update constraints, each window would be sent to a separate structure for rendering [Kaba 1993]. Rendering in this way can be sped up given the correct conditions as one can send the terrain to a separate vertex buffer object, as is the case with VBO based geometry clipmapping, or a completely different processor, as can be seen in the advanced GPU based geometry clipmapping technique.

Hoppes proposes that the terrain pyramid structure be broken down into the following design elements. We decided to stick with the original Hoppes design for the development of our terrain pyramid.

Levels. A terrain pyramid shall contain m levels. There is a nested clipmap located in every level of the terrain pyramid. Each level shall contain the same amount of data, but should represent a power level of two greater resolutions than the proceeding level. For level zero, the clipmap shall contain a one to one correspondence to the original heightmap, it being the finest level. Proceeding clipmap levels will contain courser resolutions.

Clip Regions. A clip region is the world extent of **the n x n grid data at that level.**

Active Regions. An active region is the extent meant to be render at that level. When the viewer moves, when incrementally update the clipregion to match the active region.

Render Region. A render region is the hollowed region that is indexed and sent to a VBO for a level of the terrain [Losasso 1994].

```
struct RenderRegion
{
    GLuint drawRenderRegion[2*(SIZE*SIZE)-2];
    Vector2 top; //top left position of the top render region
                 //triangle strips until
    Vector2 bottom; //
    Vector2 left; //stop at (pos *2) + 2
    Vector2 right;
};

struct TerrainPyramid
{
    //Terrain Pyramid of Clipmap information
    //of size m

    ClipRegion clipRegions[NUM_LEVELS];
    ActiveRegion activeRegions[NUM_LEVELS + 1];
    RenderRegion renderRegions[NUM_LEVELS]; //position NUM_REGIONS
                                            //is always empty
};
```

Figure 2 : C++ code snapshot of Terrain Pyramid structures.

For our implementation we constructed these regions in two distinct ways. For testing, we used terrain sampling on a terrain that was small enough to fit into memory. This method was reminiscent of the technique used in Bretell's work [2005]. In this way, we could test the functionality of the clipmapping without worrying about yet incorporating the decompression techniques. In our final implementation, we selectively pulled decompressed heightmaps into the clipmap structure based on position.

Toroidal Arrays

Toroidal Arrays was critical in the incremental update of the active regions. Toroidal arrays are arrays that use wraparound addressing, which allows for incrementally updating the array. Should one forego toroidal mapping the clip regions would have to be completely re-written in each frame, significantly reducing the performance of clipmapping. With toroidal arrays, the positions in the heightmap data maintain a perfect mapping to you clipmap, constraining each heightmap element to map to exactly one position in your clipmap no matter where the user is positioned on the terrain.

In our representation we used a simple mapping function. For each vertex*(x,y,z)* the vertex shall be mapped to the clipmap_position*(x/gs mod n, x/gs mod n*).

Creating Indices

The clipmapping solution, as I discovered, becomes increasing complex at this stage of the implementation. At each update of the clip region, the program must also update the index buffer to generate the hollowed frame to be rendered [Bretell 2005].

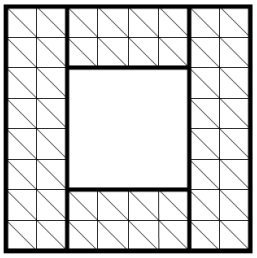

Figure 3: Example of resulting indexing array [Losasso 1994].

The algorithm itself uses the toroidally mapped vertex positions to tessellate the region and it must produce this result in such a way that the produced triangles wrap around the internal square of the preceding level. This is where development of our solution slowed down greatly.

Compute Desired Active Regions

The most important part of finding a desired active region is finding the centerpoint of that region in terrain world space [Lossaso 1994]. Once this element is found, we use a simple algorithm based on the suggestion of Lossaso and Hoppe to generate an active region. This component turned out to be the single most essential method in the clipmapping engine. Until the last days of development, our engine updated the entire array by using the ComputeDAR() method, foregoing the incremental updates.

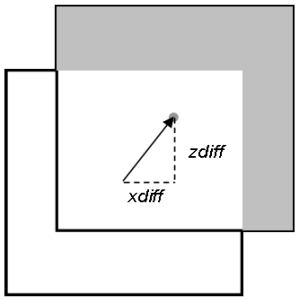

Figure 4 : Result of computeDAR() method after moving in an upward/left direction [Losasso 1994]. The algorithm only updates the greyed out L-shaped region.

Other elements of the standard geometry clipmap were not present because of time constraints.

We did not implement an algorithm the supplied for continuity between levels, though a simple solution would have been trivial to implement. The two primary papers that we used as references suggested using a simple shader program to merge boundaries together [Lossaso 1994]. One could have also used something as simple as border stitching to complete the task.

Frustum culling is also recommended but was avoided because the results would have skewed our analysis and because of time constraints.

Another feature suggested but not added was dealing with a finite heightmap [Bretell 2005]. Should the user reach the edge of the world, there was no handling of that condition. This is one of the most critical future works of our solution, as not addressing this condition causes our solution to crash and exit prematurely.

Compression

Compression is a pre-processing step that is done to fit the entire terrain in memory. It takes a very long time and hence saved the compressed files to avoid doing it every time. The program just reads in the compressed data and stores it in memory. This data is decompressed as and when needed.

As far as the size of the data goes heightmap information can very easily be compressed with not much loss of quality. We can easily compress by a factor of 100 using standard compression techniques. Thus we can accommodate fairly huge heightmaps in main memory.

When it comes down to increasing the frame rate we want to pass as little information to the GPU as possible. There are several ways in which we can do this. The technique used here is a level of detail scheme. We exploit the fact that the detailed information rendered at a distance would not be perceived that well as it would be when much closer to the user. So we render the heightmap in nested regular grids with the detail progressively decreasing as the distance from the viewer increases. Another way to reduce information sent to the GPU as far as terrains are concerned, is view frustrum culling.

Both LOD and compression are tackled together by generating an image pyramid [Losasso 2004]. The image pyramid serves as the LOD's and it also facilitates compression. The basic idea is store the difference between an image in the image pyramid and the upsampled version of its coarser image. This difference is call a residual and residuals are very easy to compress because they are very small to begin with and small changes in their values will not affect the quality that much. Hence they are a good target for compression. So what is done is the coarsest version of the image pyramid is stored along with all the compressed residuals.

Decompression has to be done in real time and only the region of interest has to be decompressed. Both of these are tackled by breaking up the heightmap into blocks. Thus only the required blocks can be decompressed and also the processing is done much faster.

Region of Interest (ROI) queries

The decompress function has to support region of interest queries and hence I compressed the heightmap in the form of blocks, thus the region of interest can be specified as a block.

Generating blocks also helped in the pre-processing as it would be extremely difficult to work with the entire heightmap all at once, since it's so big.

Generating the Compressed Heightmap Blocks

I used matlab for a lot of pre-processing because the data was so huge and the compression involved many steps. The heightmap was originally available in pieces, I stitched together the pieces and generated equal size blocks with the filename as the coordinates of the block and the level. For example, "0 0 level0", would refer to block position (0,0) with a finest level of detail.

Generating the Residuals

When generating the residuals of the compressed data, I followed this method:

- First an image pyramid is generated for each block.
- Each block is then upsampled and residuals i.e. the difference between the original and the upsampled version are calculated [Losasso 2004].

Compressing the Residuals

The residuals already had long streams of zero's so they were very good candidates for RLE. In this implementation the last bit is sacrificed hence drastically increasing the runs of zeros. RLE is then performed and the coarsest image along with the compressed residuals is stored.

Compressing ratio

During development it was noticed that the compression ratio was highly dependent on the data. In most cases, more detail meant less compression. With the compression set to the minimum I achieved tremendous compression on some blocks and minimal compression on others, thus the compressed blocks varied from 3KB to 300 KB per 1MB of data.

In this situation, it was possible for compression factor could be set to vary according to the data.

Table 1: A comparison of our compression versus the papers compression

	Uncompressed data	Compressed data	Compression Ratio
F.Losasso, H. Hoppe	40 GB	355 MB	100
Our Algorithm	2 GB	79 MB	26

Decompression

The compressed files are stored in a 2D array of character streams, thus fitting the entire heightmap into main memory. The Decompress function takes as arguments the coordinate of the block to decompress and reads in the data from the 2D array and decompresses it on the fly. Only the region of interest is decompressed this way.

- The compressed files are saved with the coordinates in the filename for example "0 0 compress", thus a loop is used to pull in all the compressed data.
- This data sits in memory and is decompressed as and when needed.
- World coordinates are translated into block coordinates using the modulus operator (ex. Terrain_coordinates*(x%512, y%512))*

Discussion

We tested our work on a computer present in the Games + Learning Lab and one in the Visualization Lab. On these computers, we ran our test data (the smaller bitmap file) using geometry clipmapping and other canon methods of terrain rendering.

Table 2: Information about the two testing machines on which we ran our analysis

Test computer one: Computer Name: RAGNAROK Processor: Intel Core2 DUO 2@2.13GHz RAM: 2GB OS: 32-bit Windows Vista GPU: NVIDIA GeForce 7900 GS
Test computer two: Computer Name: NAMELESS Processor: Intel Pentium 4 RAM: 2GB OS: 32-bit Windows Vista GPU NVIDIA GeForce 6800

Our clipmapping algorithm showed significant performance gains over the more conventional methods, leading us to believe that for significantly large datasets, the performance gains of geometry clipmapping would be well worth the development time of the other significantly less complicated solutions.

Table 3: Performance of geometry clipmaps against canon rendering methods. Clipmaps are tested at with 5 levels of detail. Terrain data sample was 675x675

GL draw primitives	8 FPS
Only VBO's	150 FPS
GeoClipmapping Solution (n = 16)	+450 FPS
GeoClipmapping Solution (n = 32)	+400 FPS
GeoClipmapping Solution (n = 64)	+300 FPS

Using a “draw primitives” method in OpenGL is by far the easiest method in which to pull terrain data from a bitmap. Unfortunately, even with a relatively small terrain of 675 x 675 the frame rate was still unusable for interactive rendering of terrain.

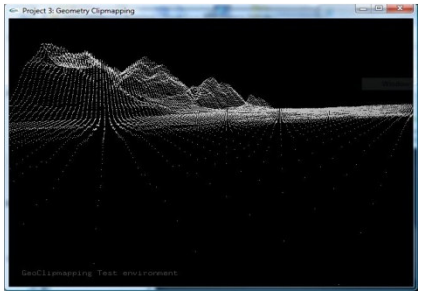

Figure 5: Test terrain rendered using OpenGL draw commands.

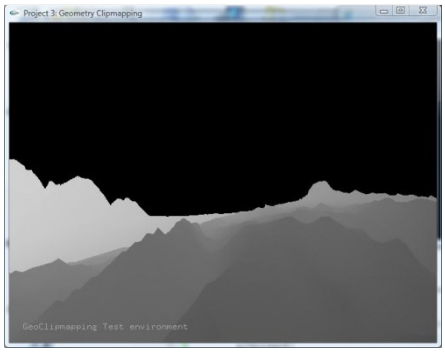

Figure 6: Terrain sample data rendered using VBO's.

When testing by pulling sample terrain into one vertex buffer, we achieved very significant performance gains, enough for a decent interactive visualization. These methods however, did not support a terrain as large as the terrain used in our final tests. An advantage, however, over the geometry clipmapping method was that the vertex buffer did not have to be updated after the initial bind, greatly increasing the performance of the algorithm over clip our initial implementation of clipmapping. Our initial implementation used our ComputeDAR() method to bind new data to our multiple vertex buffers every frame. Apparently, the key to efficiency with geometry clipmapping is optimizing your application to handle the demands of your solution. In a case where the clipmap information does not update that often, one should ensure that their algorithm selectively decides when to update the vertex and index buffers.

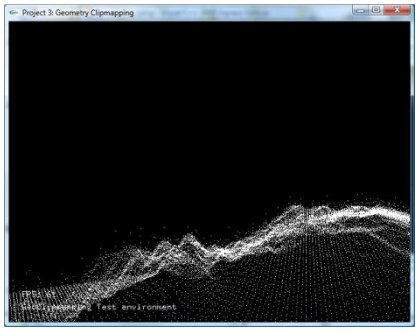

Figure 7: Initial development shot. Terrain rendered using stacked points at different resolutions of the sample terrain data.

Our clipmapping project achieved better rendering times than the preceding algorithms. Unfortunately, our project was significantly buggier and increased development time by at least a factor of 100. The algorithm efficiency and development time ratio may have been largely affected due to inexperience with advanced graphics but the results do imply that one may want to consider their own time situation and the resulting increase in performance prior to choosing a rendering solution.

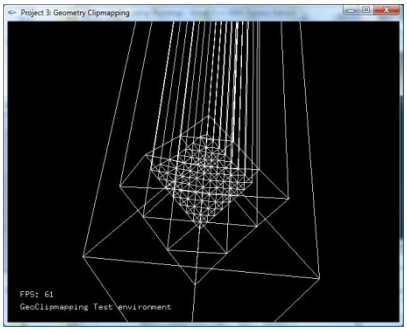

Figure 8: An example of complications with create index buffer method.

There is little comparison that can be made to our algorithm against preexisting geometry clipmap solutions. This is largely due to the fact that our solution is less complete. In the case of the Lossasso and Hoppe solution, our solution was missing tessellation, geometry clipmap updating, textures, and frustum culling.

In the case of the Bretell solution, our comparable solution did not contain continuity between levels

and we could not cleanly tested terrain tessellation. We still did a comparison; however, our results were inconclusive because of the drastic limitations of our solution.

Future Work

Most essential to continued development would be debugging our current solution so as to retrieve more stable and conclusive data. In addition, in order to compare to pre-existing geometry clipmap solutions, the must be able to create continuity between clipmap levels, create terrain normals, and use some type of texturing solution. For our solution, we intended to right a shader program to procedurally texture our terrain. In our future work, these requirements will be implemented into our existing solution.

Also, because our solution is intended to be used as a 3d simulation environment, we intend to add realism to our solution by adding a skybox and terrain lighting.

We also intend to look for effective debugging solutions for graphics programming. Our ignorance of such solutions significantly slowed our development process.

There is a lot of room for further compression, some of the ideas that could be implemented are:

- DCT and quantization instead of sacrificing 1 bit will increase the runs of zeros a lot more with less loss of quality
- Huffman coding will provide a tremendous increase in the compression ratio, because the data values are mostly similar
- Since the values of the residuals are all below 32 they can be stored in 4 bits instead of 8

Adaptive Compression

As mentioned earlier the compression ratio varies from a factor of 300 to a factor of 3 so the compression ratio is highly dependent on the data itself. Because the detailed blocks were getting distorted with a higher compression factor I had to keep the compression factor to a minimum, thus the areas more susceptible to compression were missing out. So there would be huge compression gains just by setting the compression factor to vary according to the data. This could be done in many ways including just summing up residual values which is a good indicator of detail. Also the whole process would not take up much processing time thus not affecting the frame rate.

Conclusion

We have presented our implementation of geometry clipmapping and have shown its advantages and disadvantages over canon methods. We've also displayed how our methods compared to existing geometric methods. It can be concluded that current geometry clipmapping algorithms have performance levels at or above our algorithm. This is especially so for the most recent GPU based geometry clipmapping solutions.

In retrospect, more time could have been spent logging problems and challenges so that updates between members would have been more valuable. Perhaps using a bug tracking system would have helped to speed up development. We look forward to, in coming solutions, using better software engineering processes to adequately speed up development.

Acknowledgement

Thank you to Dr. Zachary Wartell and Dr. K.R. Subramanian for their consultation and lecturing service. We also thank NSF for their contributions.

Sources

Bretell, Nick. "Terrain Rendering Using Geometry Clipmaps." http://www.cosc.canterbury.ac.nz/research/reports/HonsReps/2005/hons_0502.pdf. November 2005.

Cohen-Or, Daniel and Yishay Levanoni. "Temporal continuity of levels of detail in Delaunay triangulated terrain." IEEE Visualization '96. Pages: 37 – 42. 1996.

Kaba, James and Joseph Peters. "A Pyramid-based Approach to Interactive Terrain Visualization."

Proceedings of the 1993 symposium on Parallel rendering. 1993.

Losasso, Frank and Hugues Hoppe. "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids." SIGGRAPH 2004, 769-776. 2004.

Pajarola, Renato. "Overview of Quadtree-based Terrain Triangulation and Visualization." UCI-ICS Technical Report No. 02-01. January 2002.

Rose, Robert. "Terrain Rendering using Geometry Clipmaps."
http://www.robertwrose.com/cg/terrain/rose-geoclipmaps.pdf. June 2005.