Real-Time Visualization of Large Textured Terrains

Anders Brodersen*
University of Aarhus

Abstract

In this paper, we present a framework for real-time rendering of large scale terrains with texture maps larger than what the graphics hardware can display in a single texture. The presented system is compact and efficient, yet very simple and easy to implement.

Keywords: terrain rendering, level of detail, texturing

1 Introduction

Real-time visualization of large terrains has been an active area of research for more than a decade. In the past few years, as a natural result of the constantly increasing capabilities of modern graphics hardware, the focus has turned from CPU intensive algorithms, where level of detail is determined per triangle, to the more GPU intensive algorithms, where the level of detail is determined for a set of triangles at a time. The result is a much simpler and faster level of detail calculation at the cost of a somewhat increased triangle count.

In most cases, however, the important issue of how to add a texture map to these large terrains have been ignored. The problem arises because modern graphics hardware is limited to displaying textures of sizes up to 2048×2048 or 4096×4096, which for most applications (particularly games) is more than sufficient. However, for a terrain covering an area of 80km^2, such a texture would only provide one texel per 20 or 40 meters!

In this paper, we demonstrate how a geometric mipmap based terrain engine can be adapted to efficiently render large scale terrains and at the same time allow textures larger than what the graphics hardware is capable of displaying using a single texture. By combining carefully designed data structures with the use of vertex programs, the proposed system requires less than 2.5 bytes per vertex.

2 Related Work

Algorithms for interactive rendering of height fields are typically divided into two categories: Algorithms that take advantage of the regular structure of the datasets to create an efficient hierarchical representation of the terrains, which is then used for run-time by progressive mesh refinement or simplification[Duchaineau et al. 1997; Lindstrom and Pascucci 2002]; and algorithms based on a more general unconstrained triangulation of the terrain, such as the BDAM algorithm[Cignoni et al. 2003] and the view dependent progressive meshes presented by Hoppe[Hoppe 1998].

*rip@daimi.au.dk, Åbogade 34, 8200 Århus N, Denmark

Inspired by the massive evolution in consumer graphics hardware, a new group of algorithms has started to appear. Taking advantage of the highly increased triangle throughput of modern graphics hardware, existing algorithms have been modified[Levenberg 2002; Cignoni et al. 2003] and new algorithms have been invented[Losasso and Hoppe 2004; de Boer 2000]. What these new algorithms have in common is that they utilize far simpler algorithms which, at the cost of rendering more triangles than required to achieve the desired mesh quality, has a much lower CPU overhead. In other words, most of the work is shifted from the CPU to the GPU.

Although texture mapping is an important aspect of terrain rendering, most papers touch only very briefly the subject. The standard approach for the systems that do handle large textures is to partition the texture into tiles, binding each tile to a certain part of the terrain[Hoppe 1998]. In some cases the textures are arranged into a pyramidal structure to facilitate texture level of detail along with the geometric level of detail[Döllner et al. 2000; Cignoni et al. 2003; Hua et al. 2004]. In all cases, the texture handling is tightly coupled with the geometrical level of detail algorithm; the only truly general approach is the clip-map[Tanner et al. 1998], which requires special hardware. The approach presented in this paper is also based on texture tiling, but with texture level of detail currently limited to standard hardware controlled mipmapping. Extending this system with texture management as in [Döllner et al. 2000] and [Cignoni et al. 2003] can easily be done with no significant changes to the rest of the system.

3 Data Structures and Memory Layout

The terrain engine presented here has been implemented as part of a commercial program, requiring more than just fast rendering of the terrain. One requirement is the ability to render the terrain with a texture larger than the textures displayable by current hardware. Another requirement is that preprocessing of the data should be limited to no more than a few minutes.

In designing a system that satisfies all of our requirements, we have turned to the GeoMipMap algorithm[de Boer 2000], where the terrain is divided into smaller patches, called GeoMipMaps, of size $(2^n + 1) \times (2^n + 1)$. The original GeoMipMap algorithm is clearly designed for smaller scale terrains used in games, but we have extended it to be suitable also for rendering larger terrains.

Our system uses a 3 level data structure:

- At the bottom level we have a 2D array of *GeoMipMap* structures, containing the heights of the vertices in that particular patch as well as some information needed for the level of detail algorithm. The class declaration is listed in figure 1.
- At the mid level, we have a 2D array of the *MapBlock* structure. This structure allows us to control the texture mapping. Each MapBlock controls $n \times m$ GeoMipMaps as well as one or more textures, used for decorating all of the controlled GeoMipMaps. Like [Döllner

```
class GeoMipMap
{
   //Current level of detail
   unsigned short LoD;
   //delta max for each detail level
   unsigned short *deltaMax;
   unsigned short *heights;
   VboElement *vboElm;
   //Bounding box
   unsigned short ymin, ymax;
};
```

Figure 1: Class Declaration of GeoMipMap. The VboElement entry is described in section 5.

et al. 2000], textures can be combined using any of the blending operators supported by OpenGL.

- Finally, at the top level we have a single *Terrain* structure, which forms the interface between the terrain engine and the rest of the system. The Terrain structure also holds all data that can be reused between different MapBlocks or GeoMipMaps.

Storing the heights in the GeoMipMaps instead of having one large 2-dimensional array means that heights along the shared edge of two GeoMipMaps need to be stored in both GeoMipMaps. However, as we will see in the next sections, the benefits of this layout makes this a very small price to pay.

4 Level of Detail

The level of detail algorithm used in our system is based on the geometrical mipmapping algorithm presented by de Boer[de Boer 2000]. We follow the convention from [de Boer 2000] that the y coordinate of our vertices represents the height of the terrain.

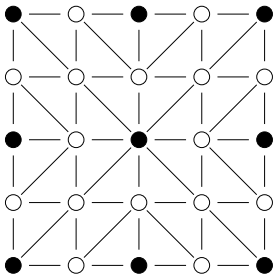

Figure 2: Mesh layout for a map size of 5. The black circles are the vertices used for lower detail level mesh (level 1). Both the white and black circles are used for the highest resolution mesh (level 0).

The terrain is subdivided into a number of smaller patches, called GeoMipMaps, of size $(2^n+1)\times(2^n+1)$ samples (typically either 17×17 or 33×33), which is then rendered at full resolution, every second vertex only, every fourth vertex only, etc., depending on the desired level of detail. In other words, the desired level of detail is determined for the entire GeoMipMap, making the refinement process both simple and efficient. Changing from one GeoMipMap level to the next simply amounts to removing every second vertex in both directions, thus reducing the number of vertices from $(2^n+1)\times(2^n+1)$ to $(2^{n-1}+1)\times(2^{n-1}+1)$, which is depicted in figure 2. At creation time we calculate, for each level of each GeoMipMap, the maximum geometrical error caused by changing from level 0 (highest resolution) to that level. This is the largest vertical distance between a vertex in the original mesh and the triangulation of the current level. Selection is then done at run-time, given the current view parameters and the world space bounding box of a GeoMipMap, by calculating the maximal geometrical error allowed inside that bounding box, with respect to a user supplied threshold. This value is then compared to the error values for the GeoMipMap, and the lowest detail level with a maximum error below this value is chosen to be rendered.

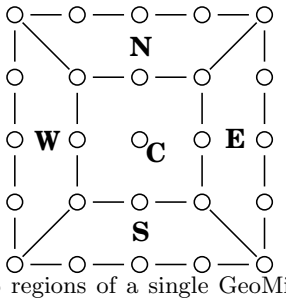

Figure 3: The 5 regions of a single GeoMipMap implicitly defined to simplify the task of avoiding cracks in the mesh. Note that when no more than 3x3 vertices remain, the center region is empty.

One last issue with the level of detail algorithm is how T-vertices and the resulting cracks in the mesh are avoided. We divide each GeoMipMap into 5 separate regions, see figure 3, where tessellation of the center region is based entirely on the currently selected level of detail. The tessellation of the four border regions are based on the currently selected level of detail, and the level of detail of the neighbor GeoMipMap sharing an edge with that region. If the neighbor is at a higher resolution, we add the missing vertices to the shared edge to ensure a consistent tessellation between GeoMipMaps. This is demonstrated in figure 4. This is the opposite approach of [de Boer 2000] and [Larsen and Christensen 2003], where vertices are removed rather than added. Removing vertices instead of adding them results in fewer triangles to render, but at the cost of removing triangles with a potential geometrical error larger than the calculated maximum. We believe that providing a tessellation of a (potentially) lower quality than implied by the user specified threshold is an ill design choice, and therefore prefer the slightly increased triangle count.

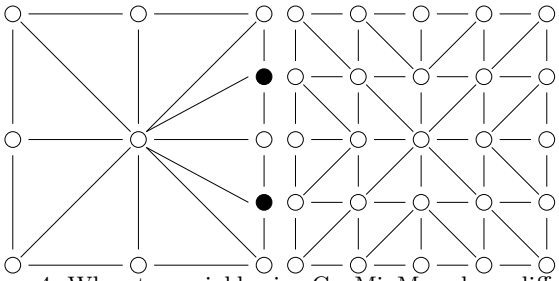

Figure 4: When two neighboring GeoMipMaps have different resolution, extra vertices (black dots) are added to the lower resolution GeoMipMaps representation of the shared edge, making the two edges identical.

4.1 Keeping Memory Usage at a Minimum

An important benefit from using a regular grid height map for terrain visualization is that the x and z coordinates can easily be calculated at runtime, and thus need not be stored. To avoid recalculating the x and z coordinate whenever we need to draw a triangle, we take advantage of the fact that

each map has its own copy of the y coordinates making up that map. By precalculating the x and z coordinates of *one* map, we can reuse these coordinates for all other maps, by simply translating the vertices to the position of the new map relative to the original map.

In practice, this means that we store the x and z coordinates of the lower left block in one array, and the y coordinates for each of the $n \times m$ maps in separate arrays. When rendering, we pass in the x and z coordinates using the 0th OpenGL vertex attribute array, and the y values using the 1th vertex attribute array. We then use a simple vertex program to assemble this into a full vertex.

Using the GeoMipMap structure listed in figure 1, the amount of memory needed to store a single GeoMipMap of size 17×17 is no more than 604 bytes. The MapBlock structure is a lightweight structure containing nothing more than a compressed bounding box (2 shorts), two indices, one or more texture ids and a list of visible maps, thus it contributes little to the overall memory consumption. As the top level Terrain structure is never instantiated more than once, even with all the indices stored (see section 5), the total memory consumption for even moderate size terrains[1] is less than 2.5 bytes per sample.

5 Efficient Rendering

Vertex Buffer Object Management In order to achieve high frame rates, it is important that we have all the needed vertex data in graphics memory. However due to the large memory requirements of the textures combined with the additional use of graphics resources by the application itself (that is, textures and vertex data for other geometry displayed along with the terrain), it is equally important that we do not waste resources on parts of the terrain that are not drawn.

For this reason we have implemented a manager for OpenGL vertex buffer objects. Each GeoMipMap stores a pointer to a VBOElement, which is a wrapper around an OpenGL vertex buffer object (VBO). VBOElements are managed using a standard *least recently used* approach. Whenever we are about to render a GeoMipMap with no VBOElement assigned, we revoke the least recently used, and assign it to the GeoMipMap.

An important detail for this to work properly is that no VBO is allowed to be used more than once per frame. Should a VBO be used twice in the same frame, then the least recently used sharing approach will often cause most if not all VBOs to be updated, effectively killing performance. To avoid this, we keep track of how many VBOs are used each frame. If at any one time we have used all VBOs in our queue, and a GeoMipMap makes a request for a VBOElement, a new one is immediately created instead, growing the queue to fit the current requirements. If the number of VBOElements in use is less than the current size of the queue, we slowly shrink the queue[2] in order to reclaim graphics resources when possible.

Pre-calculated indices For efficiency, we pre-calculate the indices for all possible level of detail configurations for a GeoMipMap and its 4 neighbors, which for GeoMipMaps of size 17×17 results in 354 different sets of indices. This takes up 88896 bytes of memory (58718 after being converted to tristrips), which are turned into triangle strips, and stored in a single VBO. A more memory efficient approach would be to separately render each of the sections of the five section GeoMipMap layout presented in section 3. This reduces the total memory requirement for the (stripified) indices, to a mere 4338 bytes (for a 17x17 GeoMipMap) but at the cost of having five draw calls per GeoMipMap instead of one. Using maps of size 17×17, we have seen a performance increase of up to 7% when using only one draw call and therefore recommend using that approach. However, for map sizes larger than 17×17, the memory needed to store the indices may become a problem, and drawing the five regions separately may then be advisable.

[1]The larger the terrain, the smaller is the influence of the single Terrain structure

[2]By removing at most one element per frame

```
RENDER-TERRAIN()
 1  visibleMapBlocks.clear();
 2  for each mb in mapBlocks
 3  do if mb is visible
 4        then visibleMapBlocks.add(mb);
 5             for each gmm in mb.geoMipMaps
 6             do if gmm is visible
 7                   then mb.visibleGeoMipMaps.add(gmm);
 8                        gmm.calculateDesiredLoD();
 9  for each mb in visibleMapBlocks
10  do mb.setupTextures();
11     mb.updateVertexProgramState();
12     for each gmm in mb.visibleGeoMipMaps
13     do gmm.getNeighbourLoDs(&n,&s,&e,&w);
14        gmm.setupVertexArrays(n,s,e,w);
15        gmm.updateVertexProgramState();
16        renderPatch();
```

Figure 5: Pseudo-code describing the rendering procedure.

Frustum Culling View frustum culling in performed using the optimized two point axis-aligned bounding box/plane intersection test with masking by Assarsson and Möller[Assarsson and Möller 2000]. Culling is performed on the 3 level layout presented in section 3. Because the bounding boxed of all three levels are aligned to the same coordinate system, the n-vertices and p-vertices are the same for all structures, and therefore need only be found once per frame. As a result, frustum culling using the existing data structures is very fast, without introducing any additional data structures.

Texture Handling Handling textures of arbitrary size is done by cutting the textures in to smaller sub-textures, each sub-texture being small enough to be displayable by the graphics hardware. The subdivision of the texture is done in a way that each sub-texture fits exactly $n \times m$ GeoMipMaps. Each sub-texture is then assigned to a MapBlock controlling exactly the $n \times m$ GeoMipMaps covered by the sub-texture. During the cull phase, each visible GeoMipMap is added to a visibility list of the corresponding MapBlock. The visible GeoMipMaps are then rendered, one MapBlock at a time, thus requiring the textures of that MapBlock to be bound only once per frame.

Splitting textures is done in an offline step for image textures that are independent of the actual terrain, while textures, such as light maps, that are tightly coupled to the geometry are generated and subdivided at runtime.

An outline of the rendering loop, including culling, level of detail calculations etc. is depicted in figure 5.

6 Results

The implementation has been tested on a laptop computer powered by an Intel Pentium M 1.5GHz processor with one

gigabyte of memory and an nVidia GeForceFx 5650 Go chip with 128MB dedicated graphics memory.

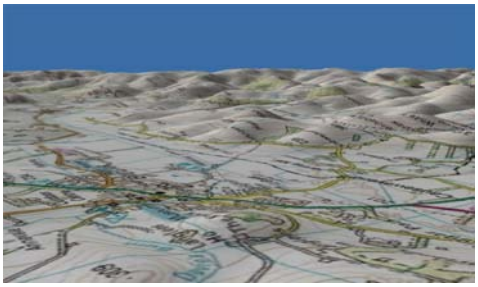

Figure 6: View of the Broad-Law terrain

Our test terrain is an area in Scotland known as Broad Law. The terrain is made up of 8193×8193 height samples, and rendered with a 8192×8192 RGB image tiled into 8×8 textures of size 1024×1024 and a 1024×1024 light map also tiled into 8×8 sub textures, see figure 6.

With a screen space error, τ, of one and a GeoMipMap size of 17×17, the terrain is rendered at on average 70 frames per second at a rate of up to 35MΔ/sec.

It is our experience that the best performance is obtained when using GeoMipMaps of size 17×17, at which point the balance between the number of draw calls and number of unnecessary triangles drawn seems to be optimal. This correlates with the results obtained by Larsen and Christensen[Larsen and Christensen 2003].

Figure 7 shows a wire frame rendering of the terrain from figure 6, with a screen space error of 3 pixels.

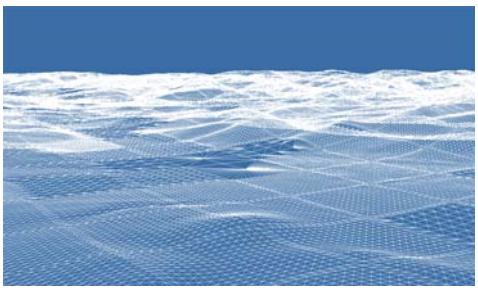

Figure 7: Wire frame rendering of terrain with $\tau = 3$ pixel.

7 Future Work

We have two issues that we would like to address in the future: Improved texture management and better support for terrains larger than what fits in main memory.

To handle textures more efficiently, we plan to investigate a system inspired by those described in [Döllner et al. 2000; Cignoni et al. 2003], although in a slightly simplified version. This entails keeping the texturing system as is, but in the cull phase we will ensure that only the textures that are actually needed or are likely to be needed within the next few frames, reside in texture memory, and that unneeded levels of the mipmap remain unspecified.

As for terrains larger than what fits in main memory, the first step is to implement the compression scheme similar to that of [Losasso and Hoppe 2004]. Another option is to use memory mapped files and then leave the rest to the operating system, as done f.ex. in [Lindstrom and Pascucci 2002].

8 Conclusion

We have presented a framework for real-time rendering of large terrains with texture maps larger than what the graphics hardware can display in a single texture. By carefully designing the data layout, and using vertex programs to allow reusing of as much data as possible, less than 2.5 bytes is stored in memory per vertex.

Finally, the presented system is simple, efficient and easy to implement.

9 Acknowledgments

This work was done in collaboration with 43D (www.43d.com). A special thanks goes to Peter Ørbæk of 43D for a lot of invaluable help and discussions. Also a great thanks to Ken Museth and Christina Brodersen for proof reading and comments to the paper.

References

Assarsson, U., and Möller, T. 2000. Optimized view frustum culling algorithms for bounding boxes. *J. Graph. Tools 5*, 1, 9–22.

Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., and Scopigno, R. 2003. BDAM — Batched Dynamic Adaptive Meshes for high performance terrain visualization. *Computer Graphics Forum 22*, 3 (Sept.), 505–514.

de Boer, W. H., 2000. Fast terrain rendering using geometrical mipmapping. http://www.flipcode.com/articles/article_geomipmaps.pdf.

Döllner, J., Baumann, K., and Hinrichs, K. 2000. Texturing techniques for terrain visualization. In *Proc. of the 11th Ann. IEEE Visualization Conference (Vis) 2000*, 227–234.

Duchaineau, M. A., Wolinsky, M., Sigeti, D. E., Miller, M. C., Aldrich, C., and Mineev-Weinstein, M. B. 1997. ROAMing terrain: Real-time optimally adapting meshes. In *IEEE Visualization ´97*, R. Yagel and H. Hagen, Eds., IEEE, 81–88.

Eberly, D. H. 2000. *3D Game Engine Design*. Morgan Kaufmann. ISBN 1558605932.

Hoppe, H. 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, IEEE Computer Society Press, Los Alamitos, CA, USA, 35–42.

Hua, W., Zhang, H., Lu, Y., Bao, H., and Peng, Q. 2004. Huge texture mapping for real-time visualization of large-scale terrain. In *VRST '04: Proceedings of the ACM symposium on Virtual reality software and technology*, ACM Press, New York, NY, USA, 154–157.

Larsen, B. D., and Christensen, N. J. 2003. Real-time terrain rendering using smooth hardware optimized level of detail. In *Journal of WSCG, Vol.11, No.1*, EuroGraphics.

Levenberg, J. 2002. Fast view-dependent level-of-detail rendering using cached geometry. In *VIS '02: Proceedings of the conference on Visualization '02*, IEEE Computer Society, Washington, DC, USA.

Lindstrom, P., and Pascucci, V. 2002. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics 8*, 3, 239–254.

Losasso, F., and Hoppe, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics 23*, 3 (Aug.), 769–776.

Tanner, C. C., Migdal, C. J., and Jones, M. T. 1998. The clipmap: A virtual mipmap. In *SIGGRAPH 98 Conference Proceedings*, Addison Wesley, M. Cohen, Ed., Annual Conference Series, ACM SIGGRAPH, 151–158. ISBN 0-89791-999-8.