

# Hardware Based Terrain Clipmapping

Alex Holkner

October 14, 2004

## Abstract

This paper describes a technique for rendering large terrains from heightmap data in real-time. A level-of-detail algorithm is employed which is position- but not view-dependent. The latest generation of programmable GPUs enables the implementation to be almost entirely in hardware, leaving the CPU relatively idle and open to more demanding physics or A.I. calculations, for example.

The algorithm has been implemented on current hardware, and, with the exception of some driver-related bugs, is shown to produce photo-realistic terrain images at interactive framerates.

## 1 Introduction

Terrain rendering typically raises several problems which must be addressed:

- Modern graphics hardware is unable to display a high-resolution terrain mesh of reasonable size. The mesh must be simplified yet retain its apparent fidelity.
- Even a simplified mesh requires a large amount of bandwidth between system memory and the video card, and may tie up the CPU for a substantial amount of time.
- Terrains have complex lighting and shadowing characteristics which cannot be calculated in real-time.

Terrain simplification algorithms fall into two broad categories: view-dependent and view-independent. View-independent algorithms attempt to find large regions of essentially co-planar polygons, and reduce their number. This can be a pre-processing step and is thus not limited by any real-time constraints. The disadvantage of these algorithms is that there is only so much degree to which they can simplify a terrain before it loses substantial detail, and there remains a challenge of rendering very large terrains. A valuable survey of existing algorithms using this technique is presented by Garland and Heckbert [GH95].

A view-dependent algorithm will simplify a mesh based on the viewer's current position and orientation, ideally such that details in the foreground are rendered with more polygons than those in the distance. These algorithms have been the topic of much research in recent decades, and typically involve

using a quad-tree-like data structure for both culling and continuous level-of-detail (LOD). A popular example of this technique is the ROAM algorithm [DWS<sup>+</sup>97], later improved by Roettger et al [RHS98].

Most continuous LOD algorithms require triangles or other polygons to be constantly split and merged, making it impossible retain a set of vertices in video memory — they suffer from their inability to be accelerated in hardware.

The technique presented in this paper uses the clipmapping algorithm presented by Lossasso and Hoppe [LH04]. While the method of clipmapping is unchanged, the implementation is done almost entirely with GPU shaders, which presents its own advantages and challenges.

## 2 Terrain Clipmapping Fundamentals

Terrain data is initially supplied as some sort of regular-grid bitmap, or heightmap  $H_0$ . During a pre-processing stage, this heightmap is resampled at half the width and height, then again at one quarter the width and height, and so on until the desired number of “coarse” heightmaps  $[H_0 \dots H_{N-1}]$  have been generated. Each of these heightmaps is essentially a view of the terrain from a further distance, and should accurately reflect the features of the terrain at that resolution.

Although the resampling stage is essentially an image operation, it is important to consider the nature of the data being manipulated. Convoluting the source heightmap with a point- or box-filter will result in unsatisfactory results, with large mountains being “flattened” and fine detail also being smoothed out. This implementation uses a sinc function with Blackman window as the filter kernel; this does a much better job at preserving the peaks and troughs of a terrain.

During the rendering phase, a number  $N$  of “clipmaps” are drawn, initially centered at the viewer’s position, and co-planar with the ground. A clipmap  $C_i$  of level  $i$  has twice the width and twice the height of the clipmap  $C_{i-1}$ , but the same number of vertices. The number of vertices along one edge of a clipmap is referred to as the “clip-size”, and must be a power of two. The ratio of the distance between vertices in a clipmap and the top-level (smallest) clipmap is called the “step”. Note that  $\text{step}(C_i)/\text{step}(C_{i-1}) = 2$

Each of the clipmaps undergo two clipping operations: one on the outer edge and one against the inside. The outer clip prevents the clipmap from extending beyond a distance  $w$  equal to half its extent from the viewer’s position. In effect, this clip prevents unnecessary detail being rendered that the viewer won’t be able to perceive when the move away from the center of a clipmap, as well as allowing the toroidal memory access described later.

Each clipmap  $C_i$  for  $i > 0$  is also clipped from the inside against the extent of clipmap  $C_{i-1}$ . This prevents overdrawing two clipmaps with vertices in the same position. All clipping distances are rounded to a multiple of the larger clipmap’s step size.

For each vertex in clipmap  $C_i$ , its height (displacement from the ground along the Y-axis) is determined via a lookup into heightmap  $H_i$ , the offset being given by the vertex’s  $(x, z)$  coordinates divided by the step of the clipmap.

Neighbouring vertices are easily polygonated with triangle strips by traversing the rows or columns of a clipmap.

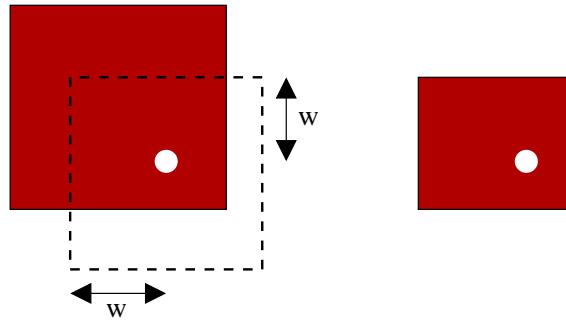


Figure 1: The outer clip operation. The dashed square shows the clipping region for a clipmap (showed in red), derived from a square of size  $2w$  centered around the viewer (shown as a white dot).

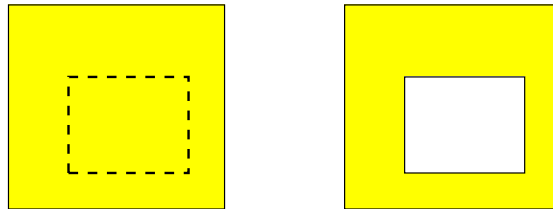


Figure 2: The inner clip operation. The yellow region is clipmap  $C_i$ , and the dashed rectangle shows the extent of clipmap  $C_{i-1}$ .

As the viewer moves from their original position, clipmaps will gradually be clipped on their outer edge while their opposite edge remains in the same position. At some point the origin of a clipmap must be moved to the viewer's location again, and its height data updated. This can be accomplished efficiently with a toroidal memory structure, in which an access into an array of size  $M$  to index  $j + kM$  gives the value at index  $j$ .

Thus, when a clipmap is moved, the height data within it need not be completely updated; instead a constant offset is applied to the vertices'  $(x, z)$  coordinates to compensate for the displacement of the clipmap. Only the invalidated region need then be updated. This is the main benefit and feature of clipmapping – that large heightmaps can be traversed over with relatively small clipmaps and with only a minimal amount of data transfer.

### 3 GPU-based Terrain Clipmapping

The above algorithm, if implemented as written, is not at all efficient, in that despite the many savings in data transfer attributed to clipmapping, the entire set of vertices for each clipmap must be sent to the video card each frame. A far more efficient solution is to implement the clipmapping procedure using the video card's programmable hardware, and transfer data only to replace the invalidated regions of memory.

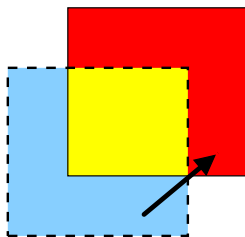


Figure 3: Demonstration of toroidal memory updates. The viewer has moved from the center of the dashed rectangle to the center of the solid rectangle (in the direction of the arrow). The yellow region is memory that can remain intact without changing. The blue and red areas are actually represented by the same regions of memory, due to toroidal access. Hence, the red area is invalidated and must be refilled.

Each clipmap is associated with a display list and a texture. The display list contains the unclipped triangle strips of the clipmap at zero-displacement (ground level). The texture has equal dimension to the clip-size and contains heightmap data for each vertex in the clip map.

Vertices from the display list are clipped with a vertex shader. The same vertex shader also displaces each vertex along the  $y$ -axis according to a lookup in the clipmap's texture.

Textures support toroidal read access natively by setting the `GL_TEXTURE_WRAP_*` parameters to `GL_REPEAT`. Unfortunately writes must be handled specially — update regions which cross over the toroid edge must be split into two normalized rectangles.

When a clipmap is updated, the display list is displaced, thereby shifting all the vertices implicitly. The vertex shader calculates texture coordinates from the vertex  $(x, z)$  coordinates and the clipmap step. Invalidated regions of the texture are updated with `glTexSubImage2D()`, meaning only the invalidated portions need to be transferred over the video bus.

## 4 Vertex Geomorphing

Due to the difference in resolutions between clipmaps, “holes” will appear in the terrain at the boundary between two clipmaps where the odd-indexed vertices of the higher-resolution clipmap differ significantly from the equivalent position in the lower-resolution one.

Odd-indexed vertices along the outer edge of each clipmap are thus displaced to the linear interpolation between the height of the vertex on either side. This eliminates gaps in the terrain, but leads to a noticeable “popping” as the clipmaps move over the terrain and different vertices are shifting position.

The solution is “geomorphing,” where vertices are gradually moved from their native “correct” positions into the linearly-interpolated “coarse” position or vice versa as the viewer approaches or recedes. A transition region is defined as a border around the inside of the outer edge of a clipmap. Vertices are given an  $\alpha$  value ranging from 0 to 1 depending on their position within the transition

region (where a vertex on the edge is assigned  $\alpha = 1$  and a vertex far away from the edge is assigned  $\alpha = 0$ ). The displacement of a vertex within this transition region is then the linear interpolation between the correct position and the coarse position of weight  $\alpha$ .

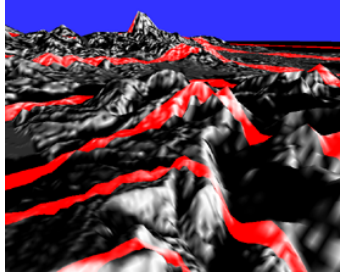


Figure 4: Transition regions highlighted in red

The width of the transition region is selectable, but in practice values between 5 and 10 produce ideal results, with no noticeable popping and adequate detail remaining in the inner clipmap. The geomorphing is performed within the vertex shader and adds negligible computational overhead.

## 5 Lightmapping

The standard OpenGL lighting model is not sufficient for accurate terrain rendering, as much of our perception of height and detail depends on shadows. Instead, a lightmap is pre-computed with a simple radiosity algorithm and applied to the terrain in real-time.

The lightmap is generated per-pixel from the detailed heightmap. For each pixel, a ray is traced from the point on the terrain to the position of the sun. If the ray intersects any terrain the point is in shadow, otherwise it is illuminated according to the diffuse component given by the standard Lambertian lighting model. An epsilon value is used for the ray intersection to allow for soft shadows.

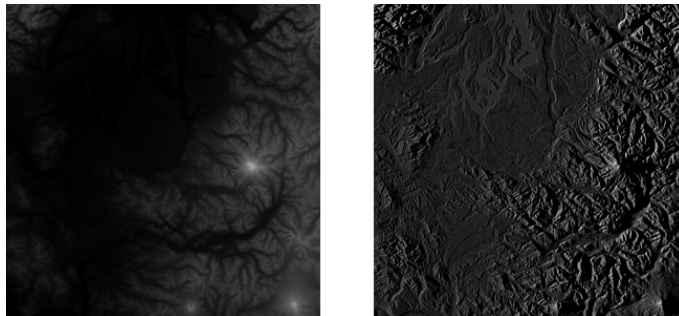


Figure 5: A heightmap [left] and its generated lightmap [right]

Lightmaps are resampled with the same sinc-Blackman filter as the heightmaps to produce a coarse lightmap for each heightmap level. Each clipmap is then

textured with the equivalent lightmap texture using the same texture coordinates as for the heightmap. A fragment shader is used to perform a bilinear filter on the lightmap texture to prevent aliasing.

Distinctions between the clipmaps can be perceived where lightmaps touch at two different clipmaps, due to their difference in resolution. This is solved by performing an additional bilinear filter within the transition region of each clipmap and interpolating the light value according to the same  $\alpha$  value.

Lightmaps need not be the same size as the heightmap. Excellent results have been obtained by setting all but the top-most clipmap to use a lightmap twice the size of the heightmap, effectively mapping two texels of light data to each vertex. This is an inexpensive way to increase the perceived resolution of a terrain, and additionally lessens the impact of transitions between clipmaps.

## 6 Implementation Issues

This project encountered severe setbacks due to the immaturity of the technology. The GPU-based clipmapping requires the vertex shader to perform texture-fetches, an operation only supported on shader level 3 devices such as the nVidia 6800. This card was introduced one month into the project and adequate documentation is still unavailable. Some of the specific problems encountered and their solutions are outlined below:

- The vertex and fragment shaders were originally written in GLSL as this seemed to be the most promising language with future support. The Linux 6800 driver does not currently support all of the language features however, and would crash regularly with valid input.

The vertex and fragment shaders were rewritten in Cg, which has a larger development base and more mature core within the Linux driver. The application code was carefully modularised to allow different shader engines or languages to be used with no substantial changes to the codebase. Currently one additional shader engine is supported — one which does all clipmapping on the CPU, and is useful for debugging.

- The documentation for Cg and, specifically the OpenGL bindings to Cg, is incomplete and inaccurate in places. The use of OpenGL in preference to DirectX is a recurring theme in the problems encountered in the project — there is not nearly the same support available.

One omission in particular, the allowable internal texture formats for vertex texture access, cost a two to three week delay in development.

- There is a bug in the Linux driver within `glTexSubImage2D()` whereby under certain conditions the driver will either hang, throw a floating-point exception or segfault. A significant proportion of development time was spent attempting to isolate and work around this bug. The current solution involves maintaining a cache of each clipmap in system memory and performing region updates with a drop-in replacement for `glTexSubImage2D()`. Textures are then transferred in full with `glTexImage2D()`. This has a substantial impact on the performance of the algorithm.

- A call to certain Cg-related functions will clear the OpenGL error flag. This is, again, a bug in the Linux driver which, while not at all severe on its own, caused significant delays in tracking down unrelated program errors.
- There is an as yet unresolved error in which the program will terminate with the message “Killed” during its heightmap resampling phase. This is an intermittent failure and tends to happen more often when resampling large images. This is a suspected bug in Fedora, given that no exception is raised.

There is one outstanding glitch in the rendering process, in which the corner of a clipmap is not correctly interpolated for a frame, but is resolved in the next.

## 7 Results

The following framerates were collected from the Sutherland lab using the 6800 shader code. Two sets of results are recorded: one with the `glTexSubImage2D()` bug workaround which severely impedes performance but is robust, and the second set without the workaround which represents ideal results expected when the driver is patched.

Note that the same framerates will be recorded regardless of the size of the terrain, due to the upper bound placed on the number of vertices rendered at once.

Clip Size	Clip Levels						
	1	2	3	4	5	6	7
64	256	145	118	83	60	25	17
128	215	91	55	26	16	7	5
256	82	27	16	7	5	2	1

Table 1: Frames per second for varying clip size and clip levels using terrain of size 1024x1024 with nVidia driver bug workaround

Clip Size	Clip Levels						
	1	2	3	4	5	6	7
64	445	193	122	92	76	64	56
128	278	108	71	56	46	38	34
256	102	50	35	25	21	18	16

Table 2: Frames per second for varying clip size and clip levels using terrain of size 1024x1024 without workaround (program hangs intermittently).

## 8 Future Directions

The biggest limiting factor (besides the crippling driver bugs) at the moment is the precomputation step involving multiple resamples of the heightmap and

lightmap. While this takes only a few seconds for the small terrain listed above, and two to three minutes for the larger terrain, it is completely infeasible to expect the computation to be performed even once for, say, a 16384x16384 terrain or larger. Calculating the lightmap is also similarly prohibitive.

Ideally the program should allow a user to load an arbitrarily sized terrain and begin traversing it immediately. This would require both a disk-caching mechanism, to load blocks of terrain off disk before they are required, and a method to resample heightmaps either in real-time or in a background thread (operating on cached disk blocks). The lightmap would either need to be similarly treated, or replaced with a dynamic lighting scheme. These are formidable hurdles that would require considerable research to overcome.

## 9 Conclusion

Terrain clipmapping is a new area of research first investigated earlier this year by Losasso and Hoppe. This paper shows an effective technique for implementing the significant portion of the algorithm within GPU programmable hardware, freeing up the CPU and system bus for other tasks. In addition, a sinc-Blackman filter is used for heightmap and lightmap resampling, which reduces the transitions between clipmaps.

The current implementation is severely hampered by defects in the current driver, however it is hoped these issues will be resolved in the near future and hardware terrain clipmapping will become a common practice for terrain visualisation and gaming.

## References

- [DWS<sup>+</sup>97] Mark A. Duchaineau, Murray Wolinsky, David E. Sigesti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, 1997.
- [GH95] M. Garland and P. Heckbert. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, Sept. 1995.
- [LH04] F. Lossao and H. Hoppe. Geometry clipmaps: Terrain rendering using nested regular grids. *ACM SIGGRAPH*, pages 769–776, 2004.
- [RHS98] S. Roettger, W. Heidrich, and P. Slussallek. Real-time generation of continuous levels of detail for height fields. In *Proc. 6th Int. Conf. in Central Europe on Computer Graphics and Visualisation*, pages 315–322, 1998.