Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail

Bent Dalgaard Larsen **Niels Jørgen Christensen**

Technical University of Denmark

ABSTRACT

We present a method for real-time level of detail reduction that is able to display high-complexity polygonal surface data. A compact and efficient regular grid representation is used. The method is optimized for modern, low-end consumer 3D graphics cards. We avoid sudden changes of the geometry - also known as 'popping', when reducing the geometry by exploiting the low-level hardware programmability in order to maintain interactive framerates. Terrain models are repolygonized in order to minimizing the visible error. Furthermore, the method minimizes CPU usage during rendering and requires minimal pre-processing. We believe that this is the first time that a smooth level of detail has been implemented in commodity hardware.

Keywords: terrain, viewing algorithms, frame-to-frame coherence, multiresolution modelling, continuous level of detail

1 Introduction

Height field terrain rendering and editing is an important aspect of GIS, outdoor virtual reality applications such as flight simulators and 3D-games. Such scenes may contain thousands of polygons and although modern graphics cards allow the display of many thousands of polygons at real-time framerates, many applications have models with geometric complexities that, by far, exceed the real-time capabilities. In the future, graphics cards will be able to display more and more polygons per second, but on the other hand the demand for using more complex models will also rise, and this gap between the performance of graphics cards and the desire for displaying more complex models is not likely to disappear in the foreseeable future.

In order to reduce the number of polygons to be rendered and thus achieve real-time framerate many research papers have dealt with different level of detail (LOD) algorithms and aggressive frustum culling.

The main focus has been to minimize the total number of polygons displayed on the screen at any point in time. Famous methods for terrain rendering are the ROAM method [Duchaineau97] and the level of detail algorithm introduced by Lindstrom et al. at SIGGRAPH '96 [Lindstrom96]. This method operates on a regularly spaced height-map and merges triangles based on the visible error in screen-space. The method cleverly avoids T-meshes and cracks in the surface by propagating triangle splits and merges in the height-map. In [Röttger98] the method originally developed in [Lindstrom96] was extended with a rapid geomorphing algorithm in order to avoid vertex popping. Hoppe also applied geomorphing to terrains in [Hoppe98]. This geomorphing method was implemented in software only.

Another method called Geometrical MipMapping that is highly optimized for modern graphics cards was recently introduced by de Boor [deBoor2000] which is very similar to [Lindstrom95]. This method divides the height-map into smaller tiles and creates a number of detail levels for each tile. Based on an approximated screen-space error, a switch between the different detail levels is made. When switching between detail levels a sudden change in the height-map (vertex popping) will occur, which will be noticeable to the viewer. In this article we will propose an algorithm for to solve this problem, as the geomorphing method

Journal of WSCG, Vol.11, No.1, ISSN 1213-6972
WSCG'2003, February 3-7, 2003, Plzen, Czech Republic.
Copyright UNION Agency – Science Press

Triangle Rendering method	Triangles per second
Individual triangles	3.5 M
Connected (strips and fans)	10.5 M
Connected in display lists	24.5 M

Table 1: Million triangles rendered per second on a GeForce 2 using different rendering methods. (with light and texture disabled)

proposed in [Röttger98] and [Hoppe98] does not apply to Geometrical MipMapping.
Furthermore, we will address the problem of exploiting the capabilities of 3D graphics cards. Because of the architecture in modern graphics cards, it is not always optimal to send as few polygons as possible to the hardware in the graphics cards. A far better approach is to create fixed chunks of homogeneous geometry that are rarely modified [El-Sana2000] (see Section 2.5 for a more in-depth explanation of what a chunk is). Using this approach it is possible to render as many as 7 times the number of triangles per second, compared to rendering individual polygons (see Table 1). Another very important issue is that rendering chunks of geometry is likely to be handled asynchronously by the graphics hardware thus removing the load from the CPU.

2 The Algorithm

A terrain can be defined in several ways. First of all it can be defined as an ordinary mesh also known as triangulated irregular networks (TINs). This method does not put any restriction on the terrain, and has been used by e.g. [Hoppe98] and [DeFloriani2000].
Another method is to define the terrain as a height field, which is a grid that is equally spaced in the x and z directions. The y value is used as the height information. This method puts more restrictions on the definition of the terrain. Nevertheless, it is often the method of choice for several reasons. Some of these properties are:

- Easy generation of height-maps with many algorithms already developed.
- Easy collision detection because the intersection between a ray and a height-map can be done in O(1).
- Fast and easy view-frustum culling because the height-map is suited for generating a quad-tree structure that is relatively simple to cull using a view-frustum.

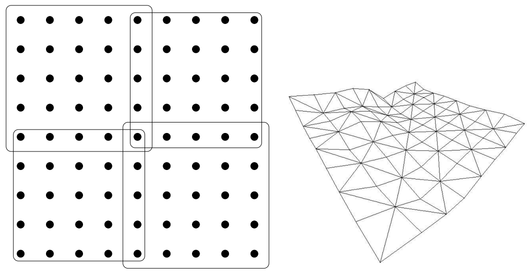
Figure 1: A terrain of 9x9 height values (left) and the 3D representation(right)

Thus, we will define our terrain as a uniformly gridded height field and use a quad-tree structure. Many others have used that approach, e.g. [Lindstrom96], [Duchaineau97] and [Röttger98].
The initial height field is a surface that consists of N by M regularly spaced grid points. Each of these grid points has a height assigned to it. First we define a level of subdivision which describes how many elements the height field should be divided into. Each of these elements we hereafter refer to as a tile. The tiles are located as leaves in the quad-tree data structure. This structure is built as a preprocess. This approach is also used by [Reddy99], [Lindstrom96] and [Röttger98]. The tiles must be regularly distributed over the entire height field and must contain $2^w + 1$ by $2^w + 1$ vertices. The tiles have to share vertices with neighbouring tiles in all directions in order to avoid gaps in the terrain. A height field of 9 by 9 will thus produce 4 tiles if the tile size is chosen to be 5 by 5 (see Figure 1). For optimal performance these tiles could be inserted into a quad tree for fast culling and spatial queries. In Figure 2 triangles have been created from the height field both in the initial resolution and a lower resolution tile which is one level higher. The difference in the number of polygons between two levels is a factor of 4. We define the level with the highest level of detail to be level 0 and the next level with 4 times fewer polygons to be level 1. The number of polygons in a level consequently sums up to $2^{2(w-l)+1}$ where the tile size is $2^w + 1$ by $2^w + 1$, and l is the level.
The basic idea for the reduction of the complexity of

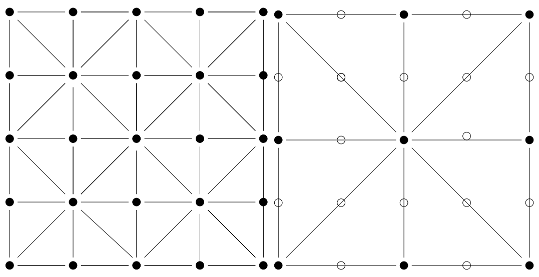
Figure 2: Level 0 (left) and Level 1(right)

the height field is to display all tiles at an appropriate level. Calculating the visible difference between the current level of the tile and a lower resolution tile generates a screen space error. If this error is smaller than a certain threshold, then the algorithm will render the scene with the lower resolution.
This is the basic idea but there are certain problems that need to be addressed when using this approach. The problems are:

- Choosing the level of detail. The level of detail has to be chosen in an appropriate way in order to minimize the visible error introduced by rendering the tile at a lower resolution. The visible error as seen on the screen should be calculated.
- Avoiding T-vertices and cracks. If two different levels are rendered next to each other, T-vertices and cracks in the polygonal mesh will occur.
- Making a smooth transition between different levels of detail. When switching directly from one level to another an artefact known as 'popping' will occur. This has to be avoided.

Solutions to each of these problems will be described in the following sections.
A height field made up of evenly distributed grid points can be triangulated in several ways. The reason for this is that a quad can be triangulated in two ways. Our triangulation scheme uses the binary right-angled triangle method, sometimes referred to as RTIN, bintree, or longest edge bisection [Lindstrom2001] [Duchaineau97].
It is noted that our triangulation is different from the method proposed in [deBoor2000]. We have chosen to triangulate the surface differently because we want to avoid long and thin triangles when connecting tiles of different levels. Furthermore the proposed structure in [deBoor2000] also needs modification when more than one out of four neighbouring tiles are rendered using a different resolution. An issue that is not described in the paper.

2.1 Choosing the level of detail

Perspective projection causes distant polygons to be rendered smaller than polygons close to the viewer. As the distance becomes greater the difference in pixels when rendering the tile at two successive levels becomes one pixel. Therefore it will be safe to switch to a higher level when a certain distance is reached. Although unsafe, it is desirable to switch to higher levels of detail, even when the difference in pixels is larger than one in order to minimize the number of polygons

Level	Reduction percentage (%)
0	0.00 %
1	75.00 %
2	93.75 %
3	98.44 %
4	99.61 %

Table 2: Reduction in number of polygons rendered calculated for different level of details.

rendered. Several options for measuring the visual difference between two levels are natural choices. Two obvious choices would be either a certain number of pixels or a fixed percentage of the screen size.
[Lindstrom95] explains that their experience is that a threshold of up to 4 pixels can be used without significant loss of image quality. In [deBoor2000] a threshold value of 6 pixels is suggested. These values are not directly comparable to our solution since we are morphing smoothly between successive levels of detail and it is therefore likely that we can use a larger threshold value without loosing significant image quality because the visually disturbing artefact known as popping is avoided.
Both [Lindstrom95] and [deBoor2000] describe methods for selecting level of details given a certain error bound. The error bounds are based on a maximum height difference between two successive level of details as shown in Figure 3.
We have chosen to implement the method described in [deBoor2000] and we will not describe that method further in this paper. We have chosen that method for ease of implementation. The number of polygons ren-

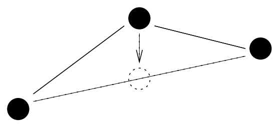

Figure 3: Error introduced when switching to a higher level of detail

dered is reduced by a factor of 4 between two levels. The reduction at each level is therefore easily calculated (see Table 2). An important property to note is that by far the greatest reduction in the number of rendered polygons is archived between levels 0,1 and 2.

2.2 Avoiding T-vertices and cracks

When two elements meet each other, the polygon edge length of the adjacent lower resolution tile will be a factor of 2^p higher than the polygons of the

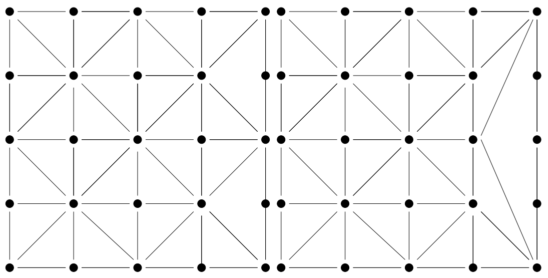

Figure 4: A tile that has a neighbouring tile of a lower resolution to the right. The difference in level is one (left) and two (right)

higher resolution, where p is the level difference between adjacent tiles. This will cause both cracks and T-vertices, that is a source of visual artefacts even when the polygons are aligned. This would occur if the two tiles in Figure 2 were joined together. In order to avoid this, it is necessary to modify the geometry of one of two adjacent tiles slightly when these are rendered next to each other at a different level of detail. We have chosen to always modify the tile with the lower resolution of two neighbouring tiles of uneven level of detail. This modification is illustrated in Figure 4 (left). The method works by doubling the size of the triangles that are adjacent to the larger tile. It is necessary to extend the quad-tree with pointers to adjacent quad-tree nodes in order to create the right triangulations of the tiles. When this method is used, it is always possible to connect two elements of different resolutions and at the same time avoid cracks and T-vertices. We have chosen only to allow the level of detail resolution to differ by a value of one between neighbouring tiles although, as indicated in Figure 4 (right), our method does not demand this restriction.

2.3 Morphing between detail levels

We will now describe our method for removing popping artifacts when switching between detail levels.
One solution for making the switch between two successive levels of detail negligible, is to set the maximum screen space error to one. But this will have the effect that most of the tiles will be rendered using a very high resolution and thus almost no polygon reduction will take place. Choosing a higher maximum screen space error will reduce the number of polygons much more but on the other hand a sudden change will happen when a tile is switched from one level to another. In the following, we will describe a method for making a smooth morph between two different levels. We will use a tile size of 3 x 3 for illustration purposes, but for any practical purposes it is advised to use a tile size of at least 9 x 9, 17 x 17 or 33 x 33 (see Section 2.5 for comments on this issue).

Morph calculations
$A = a$
$B = v\frac{(a+c)}{2} + (1-v)b$
$C = c$
$D = v\frac{(a+g)}{2} + (1-v)d$
$E = v\frac{(a+i)}{2} + (1-v)e$
$F = v\frac{(c+i)}{2} + (1-v)f$
$G = g$
$H = v\frac{(g+i)}{2} + (1-v)h$
$I = i$

Table 3: Morph calculations where v is the morphing variable in the interval [0;1]

When morphing from a higher resolution to a lower resolution the values b, d, e, f, h are linearly interpolated between their original position and the values B, D, E, F, H respectively. The calculation of the values in Figure 5 is shown in Table 3. When the linear interpolation is complete, the higher resolution tile will look exactly like the lower resolution tile and the simplified lower resolution tile may now replace the geometry. Morphing from a lower resolution to a higher resolution is similar but the procedure must be inverted. The very first thing that happens is that the tile is rendered at the higher resolution, but geometrically it is identical to the lower resolution tile. This is achieved by setting the morph variable v in the equations in Table 3 to be 1.

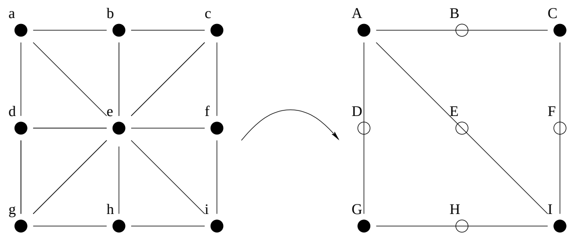

Figure 5: Before and after a morph

When morphing between two levels of detail it is not enough to morph one tile at a time since any tile can share a number of edges with the neighbouring tiles. Therefore, the border areas must be modified if the neighbouring tiles are rendered at a different level. When the level of a tile is changed, all neighbouring tiles are examined, as they may have to be modified in order to avoid T-vertices and cracks. In our current implementation, we have restricted neighbouring tiles to differ by at most one level. In the following, we will explain the algorithm we have developed in order to avoid T-vertices and cracks between two tiles.

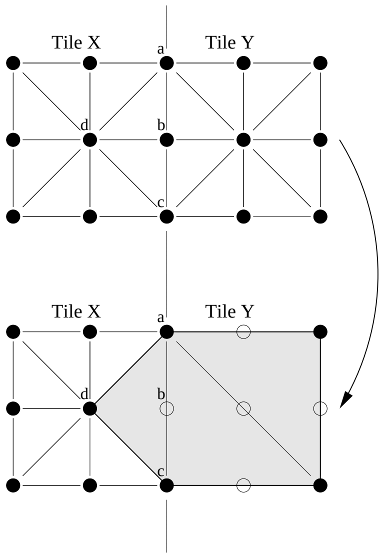

Figure 6: *Initially*: Both tile X and tile Y are rendered at the same level. *After*: Tile Y is rendered one level higher. *Border morph description*: The point b is linearly interpolated between its original value and $\frac{(a+c)}{2}$. When the morph is completed, tile X is modified by removing the triangles $\triangle dab$ and $\triangle dbc$ and adding the triangle $\triangle dac$. The shaded area is the area affected by the morphing.

Level	X morph direction	Should Y morph?
$X = Y$	up	yes
$X = Y$	down	no
$X > Y$	down	yes
$X < Y$	up	no

Table 4: Rules to determine whether the neighbouring region of Y should morph when X is morphing

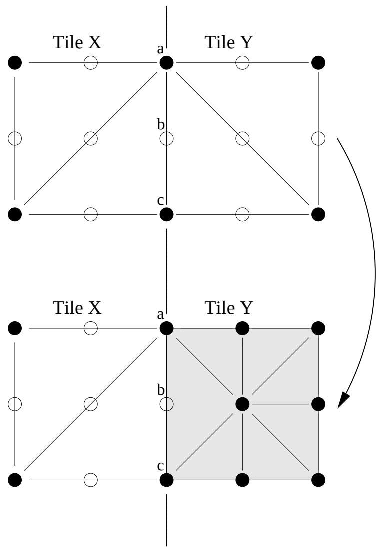

Figure 7: *Initially*: Both tile X and tile Y are rendered at the same level. *After*: Tile Y is rendered one level lower. *Border morph description*: The morphing in tile Y will take place without affecting Tile X. The shaded area is the area affected by the morphing.

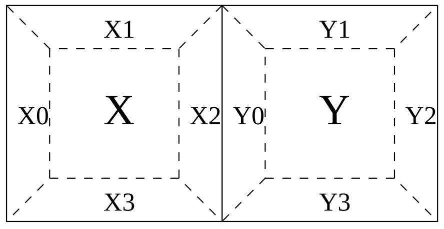

Figure 8: Morph affected regions

In Figure 6 it is shown that under some circumstances the neighbouring tile is affected, and in Figure 7 it is shown that under other circumstances the neighbour is not affected:

All tiles have four neighbours, except when the tile is located on the edge of the height field, in which case it has two or three neighbours. All neighbours have to be examined individually in order to find out whether their border region should be modified and morphed along with the tile that is changing level. In Figure 8 two tiles are shown. The regions that can be affected by a neighbour are marked by the numbers $0 - 3$. The modified region of a neighbouring tile is easily shown to be $(q + 2) mod 4$, where q is the label of the region. The rules that determine whether the region should be morphed are listed in Table 4.

All tiles are created using geometry chunks, as described earlier, which implies that the geometry data may be cached on the graphics card. As seen in Table 1 this is much faster than rendering individual polygons. The actual calculation of the morph can therefore be calculated on the graphics card. For that purpose a vertex-program is used. A vertex-program is a low level program which can be executed directly in the graphics hardware. Vertex programs were introduced by Lindholm et al. [Lindholm2001]. A vertex-program has many uses, but here we exploit its capabilities for modifying the position of a vertex. This vertex modification could just as well be made in software, but the advantage of using the hardware in the graphics card for this purpose is that it is optimized for the 3D math. Furthermore, a vertex program does not put any load on the CPU because it strictly runs

```
# Variables:
# v[OPOS] = vertex1 position
# v[NRML] = vertex2 position
# v[WGHT] = weight
#
# The function:
# R0 = weight*vertex1 + (1-weight)*vertex2
#
# The actual code:
ADD R0.x, c[4].x, v[WGHT].x;
MUL R1, v[WGHT].x, v[OPOS];
MAD R0, R1, R0.x, v[NRML];
```

Table 5: OpenGL Vertex Program

on the graphics card (on newer graphics cards such as GeForce3, ATI Radeon 8500 or better). Another advantage is that a vertex program can modify the geometry located in the memory of the graphics card, which in our case is very important, as we want to have all geometry located on the graphics card. Thus, software morphing will not be possible, and vertex programs are essential for being able to morph the geometry.
The program used in our implementation is rather simple since the only functionality of the program is to interpolate between two vertex coordinates. The code for interpolating between two vertices is shown in Table 5. When calculating the lighting it is also necessary to use the normals and these have to be interpolated in a similar way. But when interpolating normals it may be necessary to normalize after the interpolation, as a linear interpolation between two vectors does not preserve the length. A normalization on current hardware requires 3 instructions and therefore 3 clock cycles as all instructions are currently implemented so as to only require one clock cycle. It is very likely that a normalization will be implemented as a single instruction on the graphics cards in the future.

As previously described the morphing is triggered either when the screen error becomes too large and a higher resolution needs to be rendered, or when it is safe to switch to a lower resolution. The morph is basically an animation and there are several methods for controlling the timing of the animation. The options we have considered are:

- *Time controlled*. The animation is purely controlled by timing and the duration of the animation is set to a certain number of milliseconds.
- *Framerate controlled*. The animation is set to last a ceratin number of frames.
- *User speed controlled*. The speed of the animation is set to be a function of the movement of the user.

In [Hoppe98] the geomorphs are scheduled to last one second.
We have chosen to make our morph animation *user speed controlled*. The advantage of using this approach is that the terrain does not animate when the user is not moving, and when the user moves quickly it seems more natural to let the terrain change more quickly. Furthermore, the triggering of a switch between different levels of detail only occurs as the user moves a certain distance.

2.4 Tile Considerations

As mentioned earlier, the tile must be of size $2^w + 1$ by $2^w + 1$. The question is how to choose w in order to get the optimal performance. Some arguments for using a large value for w are:

- The larger the tiles, the fewer calls to the API are necessary.
- Using larger tiles makes the quad-tree smaller and thus faster to traverse.

Some of the arguments for using a smaller value for w are:

- Tiles can be rendered at a higher level when using a smaller tile size. Especially if the terrain is very rough.
- It is faster to regenerate the triangulation of a smaller tile, and the framerate will therefore not differ much from frame to frame.
- Visually it is more pleasing that only a smaller area of the terrain is morphing.

It is therefore clear that the choice of tile size depends on both the structure of the terrain and the capabilities of the CPU and graphics hardware. It is suggested by [Corpes2001] that all mutations of the detail levels are precalculated. We have tested how much memory we could use in display lists before we experienced a performance drop. As seen in Table 6, a performance drop occurs when using between 3 and 4 Megabyte of display lists (the number of vertex lists was shown to be irrelevant). This suggests that it is not appropriate to precalculate all mutations and pre-load these onto the memory of the graphics card when visualizing large terrains.
We consider a tile to be made of a collection of geometry - a chunk. This chunk can be either a display-list or a vertex-array in OpenGL. In DirectX a chunk would instead be created using a locked Vertex Buffer.

One disadvantage of using display lists is that it is not possible to modify the geometry after the list has been created. This is possible using vertex-arrays, but display lists are currently faster. Our method requires that a neighbouring tile may have to be slightly modified during a morph. We have therefore chosen to divide our tile into several display lists in order to avoid a complete regeneration during a morph. In this way we achieve the fastest polygon rendering with only minimal regeneration of display lists.

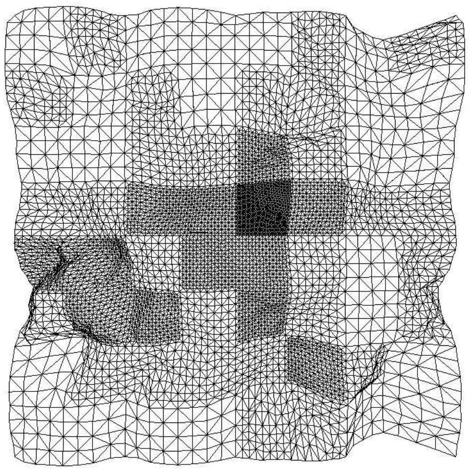
Figure 9: A terrain rendered in wireframe seen from above. The viewer is located at the center of the terrain.

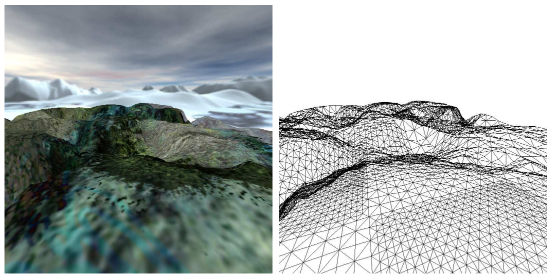
Figure 10: A simple terrain with a background (left) and rendered using wireframe

3 Results

We have implemented our terrain-rendering algorithm using the OpenGL API. Since vertex programs currently only exist as a vendor specific extension to OpenGL we used the NVidia API. We have tested the system on a Windows PC P3 800 Mhz with an NVidia

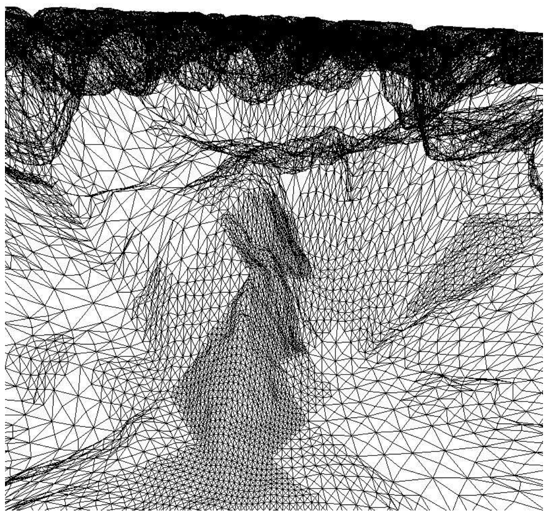
Figure 11: A 1025 by 1025 terrain rendered using a tile size of 17.

Memory used	Triangles displayed
0.5 MB	24.5 M
1.0 MB	24.5 M
3.0 MB	24.5 M
4.0 MB	20.6 M
6.0 MB	16.0 M
12.0 MB	13.7 M

Table 6: Timings for memory used for display lists compared to number of triangles displayed per second measured in millions.

GeForce 3 graphics card. We have chosen to create a predefined path and to use this path for flying through the landscape while recording the framerates. Some results can be seen in Table 7. It is noted that there is no significant difference in the framerates with or without morphing which indicates that the morphing feature does not cause a performance penalty when vertex programs are implemented in hardware. The same mesh was used in different resolutions meaning that the small height-map was very rough and the large one fairly smooth. It is noted that when using the rough height-map it is beneficial to use a small tile size, while the opposite is true when using a smooth height-map.
One of the more costly operations is the creation of the geometry chunks. This can be a problem if by coincidence many geometry chunks have to be regenerated in the same frame. Our solution was to make a queue, and only allow *one* geometry chunk to be resubmitted per frame. This is actually not very restrictive since the expected number of initiated morphs per second is very low when the observer moves with a moderate speed. This is more an insurance in order to avoid worst case

Terrain size	Tile size	With morph	No morph
513x513	17x17	66.25 fps.	67.07 fps.
513x513	33x33	39.49 fps.	38.17 fps.
1025x1025	17x17	28.31 fps.	28.51 fps.
1025x1025	33x33	39.71 fps.	38.12 fps.
2049x2049	17x17	8.65 fps.	7.66 fps.
2049x2049	33x33	18.59 fps.	18.05 fps.

Table 7: Timings for terrain rendered.

behaviour, where by coincidence a very large number of tiles initiate a morph at exactly the same frame.

4 Conclusion and Future Work

Though we find the approach very promising there is space for improvements in the future. The error metric is not so critical in our algorithm as in other algorithms, but so far we have used a very crude one from the literature and therefore the error metric should probably be re-evaluated. Furthermore, as the viewer changes position, the number of polygons rendered per frame may fluctuate significantly. The number of polygons is determined by the structure of the height field and it is thus not possible to predict the number of polygons to render. In real-time applications it is often very important to have a fixed framerate which the application is not allowed to drop below. This approach has been implemented in many other terrain algorithms e.g. [Duchaineau97] and [Röttger98]. In order to achieve this, it is necessary to modify the algorithm for choosing the level of detail so that the allowed pixel error is dependent on the current number of rendered polygons. Although graphics hardware is not very sensitive to rendering a few thousand triangles more or less.

5 Acknowledgement

A special thank goes to Martin Reddy for his invaluable input and corrections to the paper. The authors would also like to thank Kasper Høy Nielsen, Andreas Bærentzen and Michael Arneborg Eriksen for helpful comments and for proof-reading. This work was supported in part by the STVF project DMM and the Nordunit2 project NETGL.

REFERENCES

[deBoor2000] de Boer, W. H. *Fast Terrain Rendering Using Geometrical MipMapping*, unpublished and only available at http://www.flipcode.com/tutorials/geomipmaps.pdf

[Lindstrom2001] Lindstrom, P. and Pascucci, V. *Visualization of Large Terrains Made Easy*, Proceedings of Visualization 2001. pp. 363-370.

[Lindstrom96] Lindstrom, P. and Koller D. and Ribarsky, W. and Hodges, L. F. and Faust, N. and Turner, G. A. *Real-Time, Continuous Level of Detail Rendering of Height Fields*, Proceedings of ACM SIGGRAPH 96, August 1996, pp. 109-118.

[Lindstrom95] Lindstrom, P. and Koller, D. and Hodges, L. F. and Ribarsky, W. and Faust, N. and Turner, G. *Level-of-Detail Management for Real-time Rendering of Phototextured Terrain*, Technical report GIT-GVU-95-06, January 1995.

[Lindholm2001] Lindholm, E and Kilgard, M. and Turner, H. M. *A User-Programmable Vertex Engine*, Proceedings of ACM SIGGRAPH 2001, August 2001, pp. 149-158.

[Röttger98] Röttger, S. and Heidrich, W. and Slusallek, P. and Seidel, H. P. *Real-Time Generation of Continuous Levels of Detail for Height Fields*, V. Skala, editor, Proceedings of WSCG '98, pages 315-322, 1998

[Corpes2001] Corpes, G., *Procedural Landscapes*, presentation at GDC 2001

[El-Sana2000] El-Sana, J. and Evans, F. and Kalaiah, A. and Varshney, A. and Skiena, S. and Azanli, E. *Efficiently Computing and Updating Triangle Strips for Real-Time Rendering*, Computer-Aided Design Vol. 32, No. 13, Nov 2000, pp 753-772.

[Hoppe98] Hoppe, H. *Smooth view-dependent level-of-detail control and its application to terrain rendering.* IEEE Visualization 1998, October 1998, pages 35-42.

[Reddy99] Reddy, M. and Leclerc, Y. G. and Iverson, L. and Bletter, N. *TerraVision II: Visualizing Massive Terrain Databases in VRML.* IEEE Computer Graphics and Applications. vol. 19(2). 1999. pp. 30-38.

[Leclerc94] Leclerc, Y. G. and Lau, S. Q. *TerraVision: A Terrain Visualization System.* Technical Report Technical Report 540. SRI International. Menlo Park, CA. April 1994.

[Duchaineau97] Duchaineau, M. and Wolinsky, M. and Sigeti, D. E. and Miller, M. C. and Aldrich, C. and Mineev-Weinstein, M. B. *ROAMing Terrain: Real-time Optimally Adapting Meshes.* Proceedings of Visualization 1997. pp. 81-88.

[DeFloriani2000] DeFloriani, L. and Magillo, P. and Puppo, E. *VARIANT: A System for Terrain Modeling at Variable Resolution.* GeoInformatica. vol. 4(3). 2000. pp. 287-315.