Ray Tracing Height Fields

Huamin Qu* Feng Qiu* Nan Zhang* Arie Kaufman* Ming Wan †

*Center for Visual Computing (CVC) and Department of Computer Science
State University of New York at Stony Brook, Stony Brook, NY 11794-4400
†The Boeing Company, P.O. Box 3707, M/C 7L-40, Seattle, WA 98124-2207

Abstract

We present a novel surface reconstruction algorithm which can directly reconstruct surfaces with different levels of smoothness in one framework from height fields using 3D discrete grid ray tracing. Our algorithm exploits the 2.5D nature of the elevation data and the regularity of the rectangular grid from which the height field surface is sampled. Based on this reconstruction method, we also develop a hybrid rendering method which has the features of both rasterization and ray tracing. This hybrid method is designed to take advantage of GPUs newly available flexibility and processing power.

Keywords: depth image, height field, hardware acceleration, layered depth image, ray tracing, terrain rendering.

1 Introduction

Terrain rendering has many important applications, such as flight simulation, battlefield visualization, mission planning, and GIS. Terrain data usually come in the form of two complementing datasets: a color or texture image and an elevation map (i.e., height field). There are two approaches to render terrain: by rasterization or by ray tracing. Rasterization of triangle meshes constructed from elevation maps is currently more popular. It achieves fast rendering speed and high image quality by using state-of-the-art graphics hardware and various level-of-detail techniques [3, 5].

However, ray tracing has some advantages over rasterization. First, terrain models for ray tracing are usually more compact. A typical terrain model for ray tracing is simply an elevation map, which requires less storage space and consumes less memory than a triangle mesh. Second, some advantages of ray tracing, such as built-in occlusion culling, logarithmic complexity, and trivial parallel scalability, make the ray tracing approach a promising alternative for rasterization approach when the size of the terrain is very large [13]. More importantly, ray tracing provides more flexibility than hardware rendering. For example, ray tracing allows us to operate directly on the image/Z-buffer to render special effects such as terrain with underground bunkers, terrain with shadows, and flythrough with a fish-eye view. In addition, it is easy to incorporate clouds, haze, flames, and other amorphous phenomena into the scene by ray tracing. Ray tracing can also be used to fill in holes in image-based rendering.

Most ray tracing height field papers [2, 4, 8, 10] focused on fast ray traversal algorithms and antialiasing methods. Surface reconstruction for height fields by ray tracing has not been well studied in the literature. Typically, terrain surfaces are either piecewise height planes [2] which may not provide satisfying image quality, or triangle meshes [8, 9] rendered by conventional ray-triangle intersection algorithms which did not exploit the regularity of the height field data. In this paper, we develop a novel surface reconstruction algorithm which can directly reconstruct surfaces with different levels of smoothness in one framework from height fields using ray tracing.

In recent years, the traditional graphics processing unit (GPU) evolved from a fixed pipeline only for rendering polygons with texture maps to a flexible pipeline with programmable vertex and fragment stages. The design of fragment processor is highly optimized for vector instruction. Thus, GPU is capable of processing fragments much faster than CPU. In next generation GPU, the fragment stage will support more complex operations. How to fully exploit the GPUs newly available flexibility and processing power for terrain rendering is still an open research problem. Based on our novel surface reconstruction method, we develop a hybrid terrain rendering method which has the features of both rasterization and ray tracing. This algorithm is especially designed for the next generation graphics hardware.

*Email: {huamin|qfeng|nanzhang|ari}@cs.sunysb.edu
†Email: ming.wan@boeing.com

2 3D Discrete Grid Ray Tracing

Our ray tracing algorithm was inspired by several methods [1, 2, 8]. We extend and integrate these methods and propose a complete 3D discrete grid ray tracing algorithm. In our algorithm, the sampling points along a ray are always on the boundaries of cells. This is similar to Musgrave's approach [8]. However, Musgrave used a modified Bresenham DDA to traverse a 2D array of height values. We instead use a fast voxel traversal algorithm developed by Amanatides and Woo [1] to compute the coordinates of sampling points. They noticed that these irregular sampling points actually consist of two sets of evenly spaced points. Therefore, the coordinates can be computed efficiently by additions and comparisons. We also use a multiresolution ray-casting method proposed by Cohen-Or et al. [2] for antialiasing. During preprocessing, a mipmap of the elevation map is built. The traversal algorithm switches to a lower resolution at the distance where the footprint of a voxel in the image is narrower than that of a pixel.

The vertical ray coherence [2, 4, 14] in terrain has been widely exploited to accelerate the ray traversal by advancing the starting points of ray tracing. However, the vertical-ray-coherence does not exist in the vertical scanlines of the final image if the camera rotates around the view direction. Currently available techniques to handle camera rolling either need an expensive warping [4] or accesses pixels of the final images in an irregular way [14]. Instead, we devise a new algorithm to tackle the camera rolling problem. As long as the camera does not translate, visibility of a scene does not change. Therefore, texture-mapping can be used to handle camera rolling. We first render an image without camera rolling. Then, we use this image as a texture for texture mapping by the graphics hardware to compensate for camera rolling. This solution is simple and efficient.

3 Surface Reconstruction

Fig. 1 shows the framework of our surface reconstruction method. In order to simplify the presentation, we first introduce some notation. Let Z_1, Z_2, Z_3, Z_4 be the heights of the four grid points of a cell in the elevation map. We also use Z_1, Z_2, Z_3, Z_4 to refer to the actual sampling points of the terrain surface. For each cell, there are two intersection points between the ray and the cell boundaries: one entry and one exit point. Let P_1 be the entry point and P_2 be the exit point. Fig. 1a shows a possible situation of a ray passing through a cell. To simplify the presentation, we will only demonstrate our algorithm for this situation. Extension to another situation where the ray exits through the adjacent face of the cell is straightforward. Let Q_1 and Q_2 be the projections of P_1 and P_2 on lines Z_1Z_2 and Z_3Z_4 , respectively, along the height direction. Therefore, P_1Q_1 and P_2Q_2 are the offsets of P_1 and P_2, respectively, to the terrain surface along the height direction. If the heights of point P_1 or P_2 are below any heights of Z_1, Z_2, Z_3, and Z_4, the ray possibly hits the terrain surface in this cell. Then, we compute the offsets of P_1 and P_2. Fig. 1b shows how to compute the offset of a sampling point on the boundary plane of a cell. If the offsets of two sampling points in a cell have different signs, the ray hits the terrain surface in that cell.

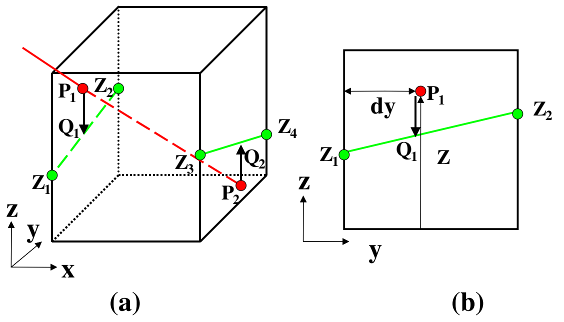


Figure 1. (a) A case of a ray passing through a cell. (b) Computation of the offset of a sampling point to the terrain surface along the height direction: $P_1Q_1 = Z_1 + dy(Z_2 - Z_1) - Z$**.**

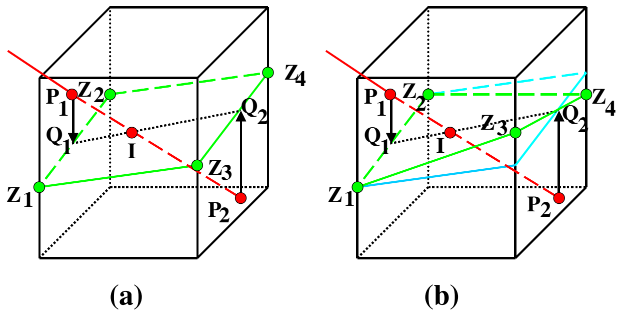


Figure 2. Linear approximation planes: (a) Z_1, Z_2, Z_3, Z_4 **are coplanar. (b)** Z_1, Z_2, Z_3, Z_4 **are not coplanar.**

3.1 Linear Approximation Planes

The first kind of 3D terrain surface which can be constructed in our framework is a linear approximation plane. Fig. 2 shows linear approximation planes. If the offsets of P_1 and P_2 to the terrain surface have different signs, the ray hits the terrain surface in this cell. Because the offset of the terrain surface is 0, we can compute the intersection point I by linear interpolation. Suppose the offset of P_1 is d_1 and the offset of P_2 is d_2. Then, the intersection point is:

$$I = P_1 + \frac{d_1}{(d_1 - d_2)}(P_2 - P_1) \tag{1}$$

If Z_1, Z_2, Z_3, Z_4 are coplanar, we can prove that our method actually reconstructs the exact plane spanned by these four points (see Fig. 2a). Suppose I is the intersection point of the ray and the plane. It is easy to see that triangle P_1Q_1I and triangle P_2Q_2I are congruent. Therefore, $d_1/d_2 = t_1/t_2$, where t_1 and t_2 are the distances of points P_1 and P_2, respectively, to point I. Therefore, I is the exact intersection point computed by our method using Eq. 1. If Z_1, Z_2, Z_3, Z_4 are not coplanar, our method actually uses a plane passing through points Q_1 and Q_2 to approximate the terrain surface for this ray (see Fig. 2b).

3.2 Triangle Mesh

We can also reconstruct a triangle mesh in our framework. Fig. 3a shows our method for triangle meshes. If the heights of point P_1 or P_2 are below any heights of Z_1, Z_2, Z_3, and Z_4, then the ray possibly hits the triangle mesh. There are two triangles in a cell, triangle $Z_1Z_2Z_3$ and triangle $Z_2Z_3Z_4$. We first compute the intersection point of the ray and triangle $Z_1Z_2Z_3$. In order to do this, we can move point Z_4 to Z_5 so that Z_1, Z_2, Z_3, Z_5 are coplanar. It is easy to see that $Z_5 = Z_3 + (Z_2 - Z_1)$. Then, we can use the method introduced in the previous section to compute the intersection point. If the intersection point is indeed inside triangle $Z_1Z_2Z_3$, then we find the real intersection point of the ray and the triangle mesh. Otherwise, we repeat the process for triangle $Z_2Z_3Z_4$.

The division used to compute the intersection point in Eq. 1 is an expensive operation. Therefore, it is desirable to find a simple way to test if the intersection point is inside the triangle without explicitly computing the intersection point. Fig. 3b shows our method. Let C be the central point of line Q_1Q_2. If the intersection point I is inside triangle $Z_1Z_2Z_3$, then

$$\frac{P_1Q_1}{P_2Q_2} = \frac{Q_1I}{Q_2I} \leq \frac{Q_1C}{Q_2C} \tag{2}$$

We project line P_1P_2 onto the base plane of the terrain. Let a be the projection of line Q_1C, and b the projection of line Q_2C. Let $dy1$ be the projection of Q_1Z_2 and $dy2$ the projection of Q_2Z_3. Then,

$$\frac{Q_1C}{Q_2C} = \frac{a}{b} = \frac{dy1}{dy2} \tag{3}$$

Therefore, if $P_1Q_1 \times dy2 \leq P_2Q_2 \times dy1$, the intersection point is inside the triangle. Our algorithm is tailored for height fields and is more efficient than general ray-triangle

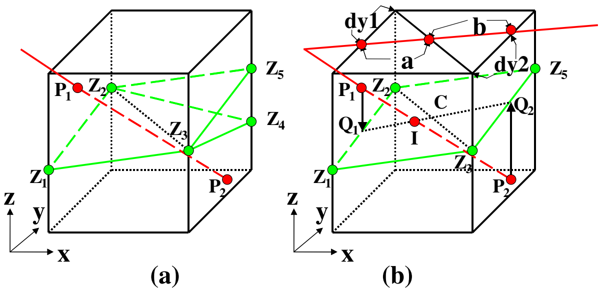


Figure 3. Triangle mesh: (a) Z_4 is moved to Z_5 so that Z_1, Z_2, Z_3, Z_5 are coplanar. (b) A simple way to test if an intersection point is inside the triangle by projecting a ray onto the xy plane.

intersection algorithms. It does not need the normal of a triangle. Unlike Musgrave's algorithm [8], our method can test if there is an intersection point of a ray and a triangle without actually computing the intersection point and the test is simpler than that of Musgrave's algorithm.

3.3 Bilinear Surface

We can also reconstruct a bilinear surface. Fig. 4 shows the bilinear surface. For any sampling point P_0 inside a cell, we use a plane which passes through point P_0 and is parallel to the boundary planes of the cell to cut the cell, we get two intersection points of the plane with lines Z_1Z_3 and Z_2Z_4. Let Z_5 and Z_6 be the intersection points. Then, the offset of P_0 to the terrain surface can be computed by using the formula $Z - Z_5 + dy \times (Z_6 - Z_5)$, with Z as the height of P_0 and dy as the distance of P_0 to the boundary plane of the cell which passes through points Z_1 and Z_3.

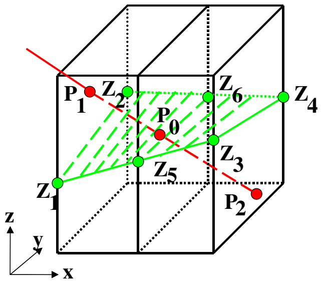


Figure 4. Bilinear surface.

The terrain surface consists of samples whose offsets are 0. If the offsets of points P_1 and P_2 have different signs, then the intersection point is bracketed in the interval [P_1, P_2]. We use the bisection method to compute the intersection point. We compute the offset of the interval's midpoint and examine its sign. The midpoint is then used to replace whichever limit has the same sign. After each iteration, the bound containing the intersection point decreases by a factor of two. At the last iteration, the midpoint of the interval is treated as the intersection point.

If the offsets of points P_1 and P_2 have the same sign, but the height of P_1 or P_2 is below any heights of Z_1, Z_2, Z_3, and Z_4, then it is still possible that there are intersection points of the ray and the bilinear surface. In this case, we use small intervals to sample the ray from P_1 to P_2. At each sampling point, we examine its offset. If the offset of a sampling point is equal to or larger than 0, this sampling point is treated as the intersection point. This can be done efficiently by incrementally computing the position of sampling point P_0 and the heights of points Z_5 and Z_6 (Fig. 4).

4 Hybrid Height Field Rendering Method

In this section, we focus on rendering the triangle mesh of Section 3.2 with the next generation programmable GPU. As shown in Fig. 3, we only need to compute two intersection points, P_1 and P_2, of each ray with the cell boundaries, where P_2 is the first intersection point which is below the terrain surface and P_1 is the last one which is above the terrain surface. P_1 and P_2 can be computed efficiently using rasterization and fragment processors of the next generation programmable GPU.

All the cell boundaries can be combined into $2n$ rectangles for an $n \times n$ height field. When projecting all these rectangles onto the screen, each fragment is an intersection point of the ray and a cell's boundary plane. Each vertex of the rectangle is assigned a 3D texture coordinate relative to the bounding box of the height field. The r texture coordinate of fragment P is the distance from the fragment center to the base plane of the height field. In the fragment program, the s and q texture coordinates are used to retrieve the height of Q from the elevation texture, where line PQ is perpendicular to the base plane and Q is on the terrain surface. If P is higher than Q, the fragment is discarded. Otherwise, we find an intersection point which is below the terrain surface. The fragment with the s and q texture coordinates, length of PQ, and depth value is written to the floating point pixel buffer without loss of precision. The depth test is used to get the fragment nearest to the view point. Thus, after the first pass, we can get the first sampling point P_2 which is below the terrain surface. The ray must intersect with the triangle in the cell containing P_2.

In the second pass, only one rectangle covering the whole screen is projected. In the fragment program, traditional ray-rectangle intersection method is used to compute all intersection points of the ray with the cell boundary, resulting in P_1, the nearest one to P_2. Then, the position of the intersection point is interpolated from the attributes of P_1 and P_2 and color value is retrieved from texture [11].

This algorithm has the following features:
1) It is a hybrid algorithm which has the features of both rasterization and ray tracing. Unlike traditional projection methods, it dramatically reduces the number of geometric primitives to be rasterized. In our algorithm, only $O(n)$ rectangles are rendered, while traditional triangle mesh methods need to render $O(n^2)$ triangles. Unlike conventional ray tracing methods, it avoids the expensive ray traversal.
2) It takes full advantage of next generation graphics hardware. Except for the rasterization program and texture mapping which have been widely used in practice, next generation graphics hardware also provides powerful fragment processors. Traditional triangle mesh methods do not take advantage of this. Our algorithm lends itself to trivial parallelism much like ray tracing methods. Thus, it can take advantage of the newly available fragment programs.

5 Experimental Results

We demonstrate our algorithm by rendering two actual terrains: a 512 × 512 southern California terrain (see Fig. 5b) and a 4096 × 2048 Grand Canyon terrain (see Fig. 5c). Each terrain model consists of an elevation map and a corresponding aerial photographic map of the same resolution. The image resolution is 500 × 400.

Table 1. Time to reconstruct different surfaces.

Surfaces	Small Terrain		Large Terrain	
	SGI	PC	SGI	PC
Height Plane	0.12	0.14	0.15	0.16
Linear App. Plane	0.18	0.19	0.21	0.22
Triangle Mesh	0.24	0.26	0.28	0.30
Bilinear Surface	0.25	0.29	0.30	0.33

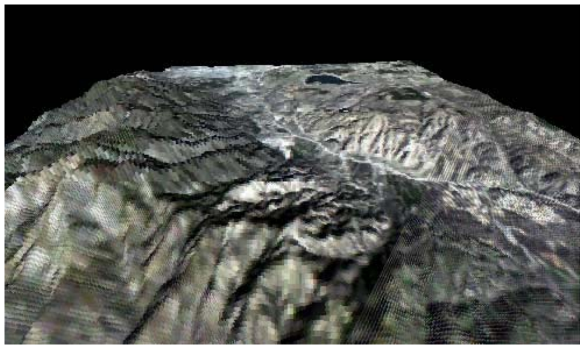
Table 1 compares the time to render four different terrain surfaces (height planes, linear approximation planes, triangle meshes, bilinear surfaces) on two hardware configurations: SGI Power Challenge (R10K processor, 194 MHZ, 4GB RAM) and Intel Pentium III (933 MHz, 384MB RAM). Only one processor is used on the SGI workstation. The height planes are piecewise constant height planes (cf. [2]). All rendering times are in seconds. These are the average times to render a frame. We use the same multiresolution method for our 3D discrete grid ray casting as that proposed by Cohen-Or et al. [2]. Table 2 shows the times of rendering the California terrain, which are broken down into four major rendering tasks. The rendering time was measured on the SGI Power Challenge. Of the four major rendering tasks, the times to compute the ray parameters (ray directions, start points, etc), the intersection points of rays and surfaces, and the color are unrelated to the size of the terrain. Only the time to skip empty space depends on the resolution of the elevation data. However, no matter how large the terrain is, we can always use multi-resolution techniques to accelerate ray traversal [2]. The focus of this paper is on surface reconstruction so we did not experiment with very large terrains which need sophisticated cache and memory management.

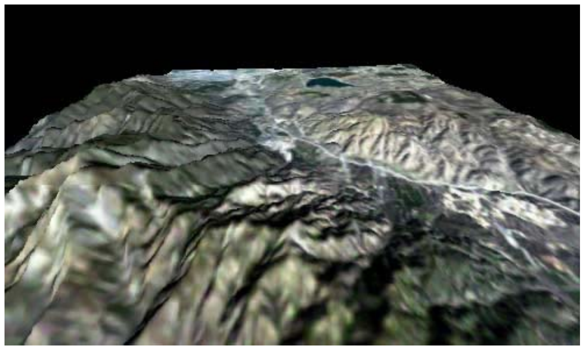
Table 2. Rendering times broken down into computing ray parameters (RP), skipping empty space (ES), computing intersection point (IP), and interpolating color (IC).

Surfaces	RP	ES	IP	IC	Total
Height Plane	0.06	0.04	0.00	0.02	0.12
Linear App. Plane	0.06	0.06	0.03	0.03	0.18
Triangle Mesh	0.06	0.07	0.07	0.04	0.24
Bilinear Surface	0.06	0.07	0.08	0.04	0.25

We also compare our algorithm with previous work. Compared to the 2.5D ray casting algorithm proposed by Cohen-Or et al. [2], our algorithm can provide better image quality. Fig. 5a shows an image rendered by the 2.5D ray casting algorithm. At the near area, the voxel size is bigger than the pixel size. Therefore, box aliasing is visible. Fig. 5b shows the image rendered using our algorithm with linear approximation planes. Comparing Fig. 5a and Fig. 5b, we can see that the image quality is improved by our algorithm. Compared to the 3D voxel-based terrain rendering method proposed by Wan et al. [14], our algorithm can provide the same quality images, use much less storage and memory, and render much faster. For comparison, it takes 0.92 sec to render the image of Fig. 5b using Wan et al.'s method on the SGI Power Challenge [14]. We also compare the image quality of the three kinds of surfaces reconstructed by our algorithm. Even though the differences are clear for shaded surfaces, they may not be very noticeable for texture mapped terrain surfaces.

We have implemented the hybrid rendering algorithm on the software simulator of Nvidia GeforceFX. The rendering cost of our algorithm includes two parts. One is the cost of rectangle rasterization. The other is the cost of the fragment program. Assume the resolution of the height field is $n \times n$. In our algorithm, only about $4n$ rectangles are rendered. Commodity graphics hardware can render more than 1M rectangles per second. Thus, the cost of the rasterization can be ignored and the total cost is determined by the efficiency of the fragment program and the number of fragments generated. In our implementation, the fragment program in the first pass includes 4 instructions and in the second pass less than 100 instructions. Therefore, the total


(a)


(b)

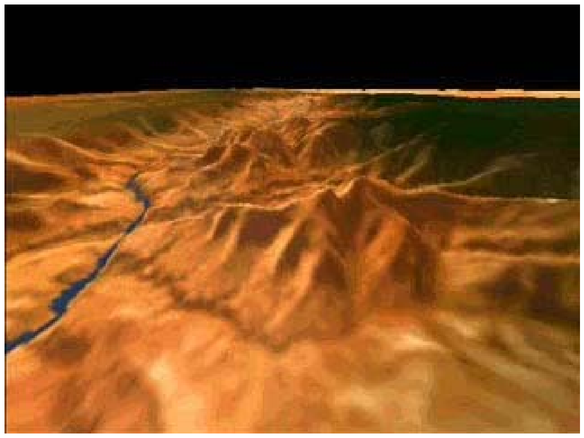

(c)

Figure 5. (a) California terrain rendered by 2.5D ray casting algorithm. (b) California terrain rendered by our algorithm using linear approximation planes. (c) Grand Canyon terrain rendered by our algorithm using linear approximation planes.

number of fragment instructions executed in one frame is less than $4f_1 + 100f_2$, where f_1 and f_2 are the number of fragments generated in each pass. With the screen resolution of 640×480, f_2 is approximately $30K$. The total number of fragments generated by our algorithm in the first pass with the Grand Canyon terrain is about $10M$. Thus, the total number of fragment instructions executed for one frame is $70M$. For the GeforceFX card which has 8 fragment processors running at 500MHz, it can execute about $4G$ fragment instructions per second. Thus, we expect the hardware to render about 60 frames per second. This is much faster than the CPU version of our algorithm.

6 Conclusions

Our contributions in this paper are:
1) A complete 3D discrete grid ray tracing height field method which extends and integrates the best parts of several available methods. We also present a novel algorithm using texture mapping to deal with the camera rolling problem so the vertical coherence in the terrain can still be exploited to accelerate ray traversal.
2) A new framework which can reconstruct three kinds of surfaces with different levels of smoothness. Especially, we present an efficient ray-triangle intersection algorithm which can test if there is an intersection point of a ray and a triangle without really computing the intersection point.
3) A hybrid height field rendering method designed for next generation graphics hardware.

Our algorithm has also important applications in image-based rendering. Recently, depth images [7] and layered depth images (LDIs) [12] have become popular modeling and rendering primitives. Many used orthographic depth images and LDIs to model objects. Our method can be directly used to render orthographic depth images. Layered depth images can be decomposed into multiple depth images [6]. Our method is suitable for rendering terrains populated with objects. Because objects can be modeled by orthographic depth or layered depth images, which are de facto height fields, we plan to build a flythrough or walkthrough system which consists only of height field models. We also plan to improve the memory coherence for our method and optimize the hybrid rendering algorithm.

Acknowledgments

This work has been supported by ONR grant N000140110034. The Grand Canyon terrain data was from The U.S. Geological Survey (USGS), with processing by Chad McCabe of Microsoft Geography Product Unit.

References

[1] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. *Eurographics '87*, pages 3–10, Aug. 1987.

[2] D. Cohen-Or, E. Rich, U. Lerner, and V. Shenkar. A real-time photo-realistic visual flythrough. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):255–265, Sept. 1996.

[3] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. *IEEE Visualization'98*, pages 35–42, 1998.

[4] C.-H. Lee and Y. G. Shin. An efficient ray tracing method for terrain rendering. *Pacific Graphics '95*, pages 180–193, Aug. 1995.

[5] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. *Proceedings of SIGGRAPH 1996*, pages 109–118, 1996.

[6] D. Lischinski and A. Rappoport. Image-based rendering for non-diffuse synthetic scenes. *Eurographics Rendering Workshop 1998*, pages 301–314, Jun. 1998.

[7] L. McMillan and G. Bishop. Head-tracked stereoscopic display using image warping. *Proceedings SPIE*, 2409:21–30, Feb 1995.

[8] F. K. Musgrave. Grid tracing: Fast ray tracing for height fields. Technical Report YALEU/DCS/RR-639, Yale University, Dept. of Computer Science Research, 1988.

[9] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. *Proceedings of SIGGRAPH 1989*, pages 41–50, 1989.

[10] D. W. Paglieroni and S. M. Petersen. Height distributional distance transform methods for height field ray tracing. *ACM Transactions on Graphics*, 13(4):376–399, Oct. 1994.

[11] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *Proceedings of SIGGRAPH 2002*, pages 703–712, 2002.

[12] J. Shade, S. J. Gortler, L. He, and R. Szeliski. Layered depth images. *Proceedings of SIGGRAPH 1998*, pages 231–242, July 1998.

[13] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3):153–164, 2001.

[14] M. Wan, H. Qu, and A. Kaufman. Virtual flythrough over a voxel-based terrain. *IEEE Conference on Virtual Reality (VR-99)*, pages 53–60, 1999.