Implementing Ray Casting in Tetrahedral Meshes with Programmable Graphics Hardware (Technical Report)

Martin Kraus, Thomas Ertl*

March 28, 2002

1 Introduction

Although cell-projection, e.g., [3, 2], and resampling, e.g., [4, 5] are the most popular approaches to hardware-assisted volume rendering of tetrahedral meshes, they are not the only ones; for example, Westermann and Ertl [6] suggested a hardware-assisted technique for ray casting in tetrahedral meshes.

In this technical report another hardware-assisted implementation of the ray casting algorithm is sketched. However, it employs programmable pixel shading to perform basically all computations in the graphics hardware and thereby avoids any significant data transfer between the graphics subsystem and the main memory. Although this approach was neither implemented yet, nor is it likely to perform as well as the mentioned methods on today's graphics hardware, it has the potential of overcoming bandwith limitations on future graphics hardware.

2 Hardware-Based Ray Casting in Tetrahedral Meshes

The basic principle of a ray casting implementation based on programmable pixel shading is to trace all viewing rays from front to back at the same time as illustrated in Figure 1a. Each of these viewing rays corresponds to one pixel of the frame buffer. In an initialization step, the first intersection of each ray with the mesh is computed. After this, the rays are propagated one cell in each of the following passes. Note that one of these passes will usually take several actual rendering passes because of limitations of the pixel shading hardware. As pixel shading programs may usually not access the frame buffer, several two-dimensional texture maps of the dimensions of the frame buffer are necessary to store the intermediate information about intersections of rays with cells of the mesh. Each texel of these texture maps corresponds to exactly one

*Visualization and Interactive Systems Group, IfI, University of Stuttgart, Breitwiesenstr. 20-22, 70565 Stuttgart, Germany. E-mail: {Martin.Kraus | Thomas.Ertl}@informatik.uni-stuttgart.de .

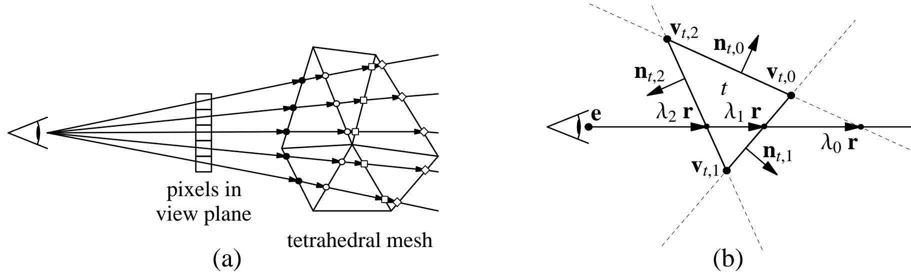

Figure 1: Hardware-assisted ray casting: (a) For each pixel one viewing ray is traced, which stops at all intersected cell boundaries. The initial intersections are marked with dots (•), further intersections with circles (∘), squares (□), and diamonds (⋄). (b) Intersections of a viewing ray with a cell's boundary.

pixel of the frame buffer and, therefore, to exactly one of the viewing rays. Thus, the algorithm will work as follows:

1. Clear the frame buffer and initialize the texture maps that contain the information about intersections. The initial values are determined by the first intersection of the viewing ray associated with each texel and the cells of the mesh.
2. Update each of the mentioned texture maps by rendering one rectangle of the same dimensions into the texture memory. In the rasterization of each texel, compute the next intersection point of the ray corresponding to the texel with a cell's boundary. Duplicate the previous intersection point if there are no further intersections.
3. Render another rectangle of the same dimensions into the frame buffer. This time, compute the volume rendering integral between the previous intersection point and the point computed in step 2 and blend the resulting color into the frame buffer.
4. Continue with step 2 unless a specified time limit has been reached.

Step 1 may be implemented using a rasterization of the visible boundary faces of the mesh. As there are usually far less boundary faces than cells in a mesh, this step is not time critical and may also be performed in software.

The computation of the volume rendering integral in step 3 may be performed as discussed in [2]. However, the blending has to be adapted from back-to-front to front-to-back blending. In the notation of [1] the accumulated associated color $\tilde{C}'_i$ and opacity α'_i after i passes with front-to-back blending is given by

$$\tilde{C}'_i = \tilde{C}'_{i-1} + (1-\alpha'_{i-1})\tilde{C}_i \qquad \text{and} \qquad \alpha'_i = 1-(1-\alpha'_{i-1})(1-\alpha_i),$$

where $\tilde{C}_i$ and α_i denote the associated color and opacity of the segment of the volume rendering integral in the i-th pass.

The stop condition in step 4 guarantees an almost constant frame rate and is easy to evaluate; however, the algorithm might not render the whole mesh. In order to guarantee the rendering of the complete mesh, one has to test whether any new intersection points have been computed in step 2.

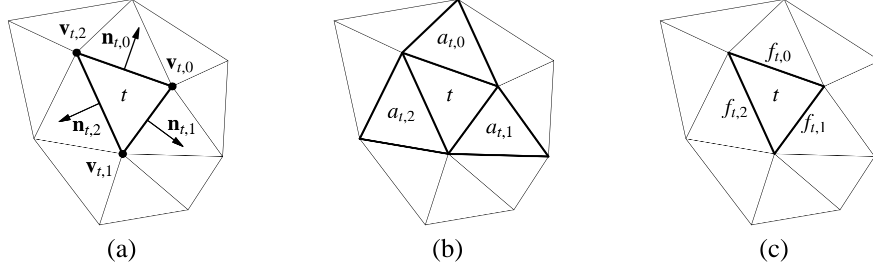

Figure 2: Nomenclature in this paper: (a) The vertex $\mathbf{v}_{t,i}$ is in and the face normal $\mathbf{n}_{t,i}$ is perpendicular to the i-th face of cell t. For tetrahedral cells i is 0, 1, 2, or 3. (b) The neighbor $a_{t,i}$ of cell t shares the i-th face. (c) Face indices $f_{t,i}$ of t's neighbors: The i-th face of t corresponds to the $f_{t,i}$-th face of t's neighbor $a_{t,i}$.

In order to describe the computations of step 2, a few notations have to be introduced; see also Figure 2. Tetrahedral cells of a mesh consisting of n cells will be identified by an integer index from 0 to $n-1$; often this index will be called t. Each tetrahedron t has four faces, the normal vectors of which are denoted by $\mathbf{n}_{t,i}$ where i specifies the face and is 0, 1, 2, or 3. Note that normal vectors will always point to the outside of their cell. Each tetrahedron t also defines four vertices $\mathbf{v}_{t,i}$. (See Figure 2a.) In contrast to the standard convention vertex $\mathbf{v}_{t,i}$ is part of the i-th face. As indicated in Figure 2b, the neighbor of a tetrahedron t that shares the i-th face is denoted by $a_{t,i}$. The index of the face of $a_{t,i}$ that corresponds to the i-th face of t is denoted by $f_{t,i}$; see Figure 2c.

The scalar field value $s(\mathbf{x})$ at a point $\mathbf{x}$ is linearly interpolated from the scalar values at the vertices of the mesh; thus, $s(\mathbf{x})$ within one tetrahedral cell t is a linear function and can be computed as

$$s_t(\mathbf{x}) = \mathbf{g}_t \cdot (\mathbf{x} - \mathbf{x}_0) + s_t(\mathbf{x}_0) = \mathbf{g}_t \cdot \mathbf{x} + (-\mathbf{g}_t \cdot \mathbf{x}_0 + s_t(\mathbf{x}_0)),$$

where $\mathbf{g}_t$ is the gradient of $s_t(\mathbf{x})$ and $\mathbf{x}_0$ is any point, e.g. one of the vertices. Therefore, $s_t(\mathbf{x})$ may be specified by the vector $\mathbf{g}_t$ and the scalar $\dot{g}_t = -\mathbf{g}_t \cdot \mathbf{x}_0 + s_t(\mathbf{x}_0)$.

As indicated in Figure 1b, $\mathbf{e}$ denotes the eye point and the direction of a viewing ray is specified by a normalized vector $\mathbf{r}$. Any point $\mathbf{x}$ on a viewing ray will be identified by the factor λ such that $\mathbf{x} = \mathbf{e} + \lambda\mathbf{r}$.

All data have to be stored in texture maps in order to access them in a hardware-based pixel shading program. Table 1 summarizes one possibility to organize these texture maps. Note that the cell indices are encoded in two texture coordinates and two color components as their range will usually exceed the range of a single texture coordinate or color component. Apart from the five constant texture maps for vertices, face normals, neighbor data, linear functions, and normalized directions of the viewing rays

data in texture map	texture coordinates			texture data			
	u	v	w	r	g	b	α
vertices	t		i	$\mathbf{v}_{t,i}$			—
face normals	t		i	$\mathbf{n}_{t,i}$			—
neighbor data	t		i	$a_{t,i}$		$f_{t,i}$	—
$s_t(\mathbf{x})$	t		i	$\mathbf{g}_t$			$\dot{g}_t$
ray directions	raster pos.		—	$\mathbf{r}$ for this ray			—
entered cells	raster pos.		—	t		i	—
intersection points	raster pos.		—	ray factor λ			$s(\mathbf{e}+\lambda\mathbf{r})$

Table 1: Summary of the texture maps described in the main text.

associated with the pixels of the frame buffer, there are two texture maps describing the intersections of viewing rays with the boundaries of cells: One texture map specifies the most recently entered cell for each viewing ray indexed by the raster position of the corresponding pixel, and similarly another texture map specifies the position of and the scalar value at the last intersection point. This information is necessary in order to compute the segment of the volume rendering integral between two successive intersection points in step 3 of the algorithm. In fact, the latter texture map has to be duplicated as the algorithm needs to access the data of the new and the previous intersection point.

In step 2 of the algorithm the next intersection of each viewing ray has to be determined; thus, the two texture maps specifying the entered cells and the intersection points have to be recomputed. The next intersection point is the exit point of the cell entered at the previous intersection point. This point may be determined by computing all intersection points of the ray with the four faces of the entered cell and choosing the intersection point that is closest to the eye point but not on a face that is visible from the eye point.

With the eye point $\mathbf{e}$ and the normalized direction $\mathbf{r}$ of the viewing ray (see Figure 1b), the four intersection points with the faces of cell t are $\mathbf{e}+\lambda_i\mathbf{r}$ with $0 \leq i < 4$ and

$$\lambda_i = \frac{(\mathbf{v}_{t,i}-\mathbf{e})\cdot\mathbf{n}_{t,i}}{\mathbf{r}\cdot\mathbf{n}_{t,i}}.$$

This equation is easily implemented with pixel shading operations as three-dimensional vector operations are usually well supported. A face is visible, if the denominator in the previous equation is negative; thus, this test comes almost for free. If λ_i is set to an appropriately large number for all visible faces, $\min\{\lambda_i | 0 \leq i < 4\}$ will identify the exit point. Determining the minimum of four numbers is usually less well supported by pixel shading hardware, but it may be implemented with the help of a sequence of comparisons.

Once the minimum λ_i and its face i are identified, the intersection point $\mathbf{x}$ may be computed as $\mathbf{x} = \mathbf{e}+\lambda_i\mathbf{r}$ and the value of the scalar field at that point is $s(\mathbf{x}) = \mathbf{g}_t\cdot\mathbf{x}+\dot{g}_t$. For each intersection point, λ_i and $s(\mathbf{x})$ is stored as this information is required in step 3 of the algorithm to compute the volume rendering integral between two successive

intersection points.

The cell entered at $\mathbf{x}$ is given by $a_{t,i}$ and the new face index in this cell is $f_{t,i}$. The values of $a_{t,i}$ and $f_{t,i}$ will also indicate whether $\mathbf{x}$ is on the boundary of the mesh, as there is no neighbor in this case. Once a viewing ray leaves the mesh, the intersection point should be kept constant because constant intersection points will result in segments of length zero, which will not contribute to the volume rendering integral. However, this procedure is only useful for convex meshes as there may be re-entries if the mesh is non-convex.

3 Discussion and Conclusions

As each pass of the algorithm requires the update of two texture maps and one update of the frame buffer, there are at least three rendering passes per iteration of the algorithm. On today's graphics hardware further rendering passes and additional texture maps for intermediate results are necessary because the maximum number of operations per pixel shading program is rather small. Another disadvantage of this method is the need to rasterize texels and pixels, although their viewing rays have already left the mesh. However, it might be possible to use early depth-tests in order to minimize the associated performance penalty.

The main advantage of this approach is that the complete mesh data is stored in the texture memory. Therefore, there is almost no data transfer between the main memory and the graphics subsystem provided that there is enough texture memory.

This algorithm may also serve as a (benchmark) test for programmable pixel shading hardware and pixel shading languages, as it is an application of programmable pixel shading to an actual problem of computer visualization and it relies on several aspects of programmable pixel shading (including dependent textures, three-dimensional texture maps, vector operations, and rendering to texture memory).

References

[1] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Proceedings Eurographics/SIGGRAPH Workshop on Graphics Hardware '01*, pages 9-16, 2001.

[2] S. Röttger, M. Kraus, and T. Ertl. Hardware-Accelerated Volume And Isosurface Rendering. In *Proceedings IEEE Visualization '00*, pages 109-116, 2000.

[3] P. Shirley and A. Tuchman. A Polygonal Approximation To Direct Scalar Volume Rendering. *ACM Computer Graphics (San Diego Workshop on Volume Visualization)*, 24(5):63-70, 1990.

[4] M. Weiler and T. Ertl. Hardware-Software-Balanced Resampling for the Interactive Visualization of Unstructured Grids. In *Proceedings IEEE Visualization 2001*, pages 199-206, 2001.

[5] R. Westermann. The Rendering of Unstructured Meshes Revisited. In *Proceedings EUROGRAPHICS/IEEE Symposium on Visualization 2001*, 2001.

[6] R. Westermann and T. Ertl. The VSBUFFER: Visibility Ordering Unstructured Volume Primitives By Polygon Drawing. In *Proceedings IEEE Visualization '97*, pages 35-43, 1999.

[7] R. Westermann and T. Ertl. Efficiently Using Graphics Hardware In Volume Rendering Applications. *ACM Computer Graphics (Proceedings SIGGRAPH '98)*, pages 169-177, 1998.