

Hardware-Based Ray Casting for Tetrahedral Meshes

Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl*
Visualization and Interactive Systems Group, University of Stuttgart, Germany

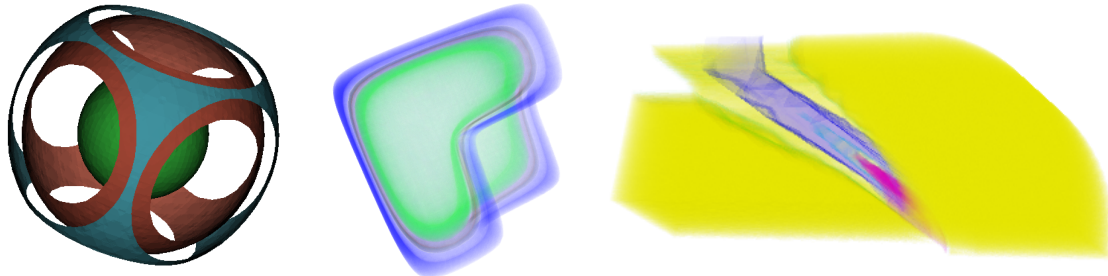


Figure 1: All images show tetrahedral meshes consisting of 125K to 190K cells rendered with our hardware-based ray casting algorithm. The algorithm exploits the programmable fragment unit of the ATI Radeon 9700 graphics chip and runs at several frames per second in a 512×512 viewport. The left image shows multiple shaded isosurfaces, the middle and right images are rendered with a full density-emitter model.

Abstract

We present the first implementation of a volume ray casting algorithm for tetrahedral meshes running on off-the-shelf programmable graphics hardware. Our implementation avoids the memory transfer bottleneck of the graphics bus since the complete mesh data is stored in the local memory of the graphics adapter and all computations, in particular ray traversal and ray integration, are performed by the graphics processing unit. Analogously to other ray casting algorithms, our algorithm does not require an expensive cell sorting. Provided that the graphics adapter offers enough texture memory, our implementation performs comparable to the fastest published volume rendering algorithms for unstructured meshes.

Our approach works with cyclic and/or non-convex meshes and supports early ray termination. Accurate ray integration is guaranteed by applying pre-integrated volume rendering. In order to achieve almost interactive modifications of transfer functions, we propose a new method for computing three-dimensional pre-integration tables.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and Framebuffer Operations, Display Algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, Shading, Shadowing, Texture, Raytracing

*VIS, Universität Stuttgart, Universitätsstr. 38, 70569 Stuttgart, Germany. E-mail: {Manfred.Weiler|Martin.Kraus|Thomas.Ertl}@vis.uni-stuttgart.de, merzms@gmx.de.

Keywords: ray casting, pixel shading, programmable graphics hardware, cell projection, tetrahedral meshes, unstructured meshes, volume visualization, pre-integrated volume rendering

1 Introduction

The evolution of computer graphics hardware has always been one of the predominant factors for the design of algorithms for computer graphics in general and computer visualization in particular. For example, the steady increase in memory of graphics subsystems favored frame buffer and texturing techniques over alternative approaches. Nowadays, modern graphics hardware is additionally characterized by a highly parallel architecture, floating-point arithmetic, and an increasingly flexible programmability. Provided that these features are efficiently exploited, many algorithms can be implemented completely in graphics hardware, thereby reducing the communication between the CPU and the graphics adapter to the bare minimum and running graphics hardware at full capacity. Therefore, the possibility to efficiently exploit these features is a more and more important trait of visualization algorithms. In order to achieve this goal, there are two particularly important requirements for the design of an algorithm: Firstly, a parallel implementation should be straightforward; and secondly, the algorithm should not require random memory writes. The latter requirement is often expressed by a “streaming processor model”; see, for example, [Purcell et al. 2002].

With respect to direct volume visualization of tetrahedral meshes, it is worth noting that there appears to be no working implementation of a hardware-based algorithm that fulfills these two requirements. For example, cell projection with non-commutative blending requires cell sorting, but a parallel implementation of cell sorting in graphics hardware has not been published yet. With the R-buffer architecture [Wittenbrink 2001; King et al. 2001] order-independent cell projection could be achieved. Unfortunately, it has not been built yet.

On the other hand, ray casting was recently implemented for uniform meshes in programmable graphics hardware [Röttger et al. 2003]. Moreover, ray casting was also successfully employed

for view-independent projection of single tetrahedra [Weiler et al. 2002; Weiler et al. 2003]. Therefore, our approach is to design and implement a complete ray caster for tetrahedral meshes in programmable graphics hardware. Obviously, all pixels can be processed in parallel and—as we will demonstrate—no random memory writes are required. Similarly to other ray casting approaches, cyclic meshes do not pose any particular problems and the performance can benefit from early ray termination techniques.

Thus, ray casting appears to be an extremely attractive algorithm for direct volume visualization of tetrahedral meshes on today’s and future graphics hardware. Unfortunately, limitations of texture memory resources and the particular problems posed by complex non-convex meshes will often diminish the advantages of our approach. Therefore, instead of suggesting that ray casting will replace cell projection as the algorithm of choice for hardware-assisted direct volume visualization of unstructured meshes, we want to clarify under which circumstances ray casting is a serious competitor on modern graphics hardware.

Before presenting our ray casting algorithm for tetrahedral meshes in Section 3, we discuss previous and related work in Section 2. Section 4 introduces *incremental pre-integration*, an algorithm that allows us to employ pre-integrated volume rendering and still modify transfer functions at almost interactive rates. Implementation issues of our ray casting approach are discussed in Section 5, while results are presented in Section 6.

2 Previous Work

Our ray casting algorithm is mainly based on the algorithm published in [Garrity 1990], which—for each viewing ray—traverses a mesh by following links between cell neighbors. However, in order to compute re-entries of viewing rays in non-convex meshes, we employ a concept published in [Williams 1992], which we refer to as the convexification of non-convex meshes. Ray integration within individual cells is performed analogously to the hardware-based, view-independent cell projection published in [Weiler et al. 2002; Weiler et al. 2003]; in particular, we also employ pre-integrated volume rendering as suggested in [Röttger et al. 2000].

Several of our modifications of Garrity’s ray casting algorithm that were necessary for an implementation in programmable graphics hardware were suggested by the ray tracing algorithm in [Purcell et al. 2002] and the ray casting algorithm for uniform meshes published in [Röttger et al. 2003]. Preliminary versions of our approach were described in [Kraus and Ertl 2002] and [Weiler et al. 2003].

3 Parallel Ray Casting

The fundament of our ray casting algorithm for programmable graphics hardware is a ray propagation approach similar to [Garrity 1990]; see Figure 2. Each viewing ray is propagated front to back from cell to cell until the whole mesh has been traversed. The ray starts from its first intersection with the mesh, which is determined during an initialization phase. The traversal is performed in passes; in each pass the color and opacity contribution of a pixel’s current cell is computed and accumulated in the frame buffer.

A software solution typically processes the pixels and, therefore, the viewing rays successively. However, ray casting algorithms can be parallelized easily as the computations performed for each pixel are independent. This paves the way for an implementation on programmable graphics hardware, where each fragment is processed individually.

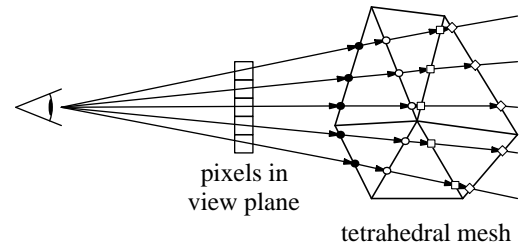


Figure 2: *Ray propagation: For each pixel one viewing ray is traced, which stops at all intersected cell faces. The initial intersections are marked with dots (●), further intersections with circles (○), squares (◻), and diamonds (◊).*

3.1 Overview

We employ fragment programs to perform all computations of the ray propagation in graphics hardware. A screen-sized rectangle is rendered to perform one propagation step for each pixel and, thus, for each viewing ray at the same time. With each rendered rectangle each viewing ray will proceed one cell through the tetrahedral mesh.

This strategy requires that intermediate information on intersections of rays with cells of the mesh is communicated between successive rendering passes. This is accomplished with the help of several two-dimensional RGBA textures of the dimensions of the frame buffer that are read and updated in every pass. The textures contain the current intersection point of the ray with the face of a cell and the index of the cell the ray is about to enter through this face.

In order to be able to compute all required information for a rendering pass, the fragment program requires access to the mesh data. Therefore, we store vertex data, face normals, and neighbor data in several texture maps, which not only enables random access by the fragment program but also allows for the data to reside in the local memory of the graphics adapter; thus, our algorithm has very low bandwidth requirements.

For each viewing ray, the algorithm basically performs the following steps:

1. initialization
2. while still within the mesh:
 - (a) compute exit point for current cell
 - (b) determine scalar value at exit point
 - (c) compute ray integral within current cell
 - (d) blend to frame buffer
 - (e) proceed to adjacent cell through exit point

We start by clearing the frame buffer and by initializing the first intersection of the viewing ray, which is an intersection with one of the boundary faces of the mesh. This may be implemented using a rasterization of the visible boundary faces. However, it may also be performed in software as there are usually far less boundary faces than cells in a mesh, and thus, this step is not time critical. The remaining steps can be divided into ray integration and ray traversal issues. We will describe both parts after a brief introduction to the nomenclature used in this paper.

3.2 Nomenclature

In order to describe the computations of one rendering pass, a few notations have to be introduced; see also Figure 3. Tetrahedral cells of a mesh consisting of n cells are identified by an integer index from 0 to $n-1$; often this index is 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 always point to the outside of their cell. Each tetrahedron t also defines four vertices $\mathbf{v}_{t,i}$, where vertex $\mathbf{v}_{t,i}$ is opposite to the i -th face; see Figure 3a. As indicated in Figure 3b, 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 3c.

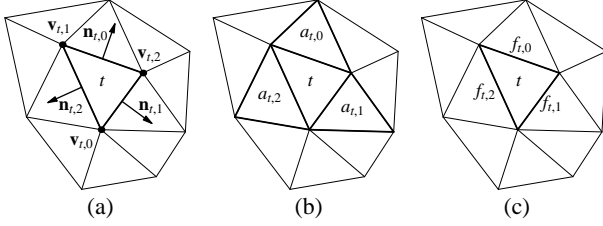


Figure 3: Nomenclature in this paper: (a) The vertex $\mathbf{v}_{t,i}$ is opposite to and the face normal $\mathbf{n}_{t,i}$ is perpendicular to the i -th face of cell t . For tetrahedral cells i is 0, 1, 2, or 3. (b) The neighbor $a_{t,i}$ of cell t shares the i -th face. (c) Face indices $f_{t,i}$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

3.3 Ray Integration

Ray integration within single cells is the main task of one rendering pass of our ray casting algorithm. Given the current cell for each viewing ray we have to compute the color and opacity contribution for the ray segment inside the current cell, which has to be blended into the frame buffer. Note that we apply the pre-integrated classification approach published in [Röttger et al. 2000], which stores pre-integrated color and opacity values in a texture map, and uses the scalar value at the entry point, the scalar value at the exit point, and the length of the ray segment as parameters for the lookup into this texture.

The entry point and its scalar value can be read easily from the textures communicating intermediate information between rendering passes as introduced in Section 3.1. These textures also identify the cell that the ray entered at this point. We determine the corresponding exit point by computing three intersection points of the ray with the faces of the entered cell and choosing the intersection

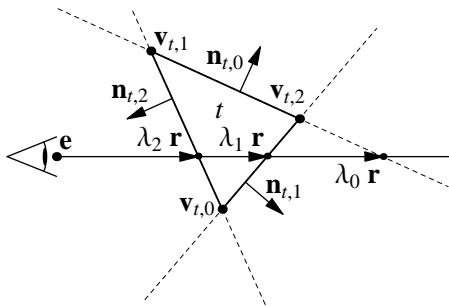


Figure 4: Intersections of a viewing ray with the faces of a cell.

point that is closest to the eye point but not on a face that is visible from the eye point. Note that it is not necessary to consider the intersection with the entering face.

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

$$\lambda_i = \frac{(\mathbf{v} - \mathbf{e}) \cdot \mathbf{n}_{t,i}}{\mathbf{r} \cdot \mathbf{n}_{t,i}}, \quad \text{where } \mathbf{v} := \mathbf{v}_{t,3-i}. \quad (1)$$

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, $\min\{\lambda_i | 0 \leq i < 4 \wedge i \neq j\}$ identifies the exit point. Determining the minimum of three numbers is usually less well supported by pixel shading hardware, but it may be implemented with the help of a sequence of conditional set operations. (An alternative solution based on texture mapping is discussed in [Weiler et al. 2002].)

Once the minimum λ_i and its face i are identified, the intersection point \mathbf{x} may be computed as $\mathbf{x} = \mathbf{e} + \lambda_i \mathbf{r}$. The scalar field value $s(\mathbf{x})$ at a point \mathbf{x} can be computed as

$$s(\mathbf{x}) = \mathbf{g}_t \cdot (\mathbf{x} - \mathbf{x}_0) + s(\mathbf{x}_0) = \mathbf{g}_t \cdot \mathbf{x} + (-\mathbf{g}_t \cdot \mathbf{x}_0 + s(\mathbf{x}_0)), \quad (2)$$

where \mathbf{g}_t is the gradient of the scalar field, which is constant for a cell, and \mathbf{x}_0 is any point in cell t , e.g., one of the vertices. Note that Equation (2) implies that we only have to store one vector \mathbf{g}_t and one scalar $\hat{g}_t = -\mathbf{g}_t \cdot \mathbf{x}_0 + s(\mathbf{x}_0)$ for each cell in order to be able to compute the scalar field value at any point within the cell. Moreover, this reconstruction of scalar values involves only one dot product and one scalar addition, which can be performed much more efficiently than a full linear interpolation from the scalar values at the vertices of the mesh since no computation of interpolation weights is involved.

With the scalar values of the entry point and the exit point and the length of the ray segment, which can be computed as the distance between the coordinates of these points, we are now able to evaluate the ray integral by performing a lookup in the pre-integration texture map. Note that the length has to be divided by the maximum edge length of the mesh in order to serve as a texture coordinate in the range of $[0 \dots 1]$. This dependent lookup can be implemented by only one additional fragment program operation.

As demonstrated in [Weiler et al. 2002] this lookup supports different optical models including emission, absorption, maximum intensity projection, and the rendering of multiple flat shaded iso-surfaces as well as arbitrary transfer functions, which only affect the generation of the texture map. In contrast to [Weiler et al. 2002], however, our ray casting approach naturally allows for optical models requiring non-commutative blending, in particular the full density-emitter model [Williams and Max 1992] that is commonly applied in volume rendering applications (see Figure 5). The cell traversal ensures the processing of the ray segments in correct visibility order. Therefore, in contrast to cell projection based algorithms no expensive cell sorting is required and visibility cycles are correctly rendered.

In a last step, the color contribution of the ray segment has to be written to the frame buffer. As the blending has to be performed front-to-back, the accumulated associated (i.e., premultiplied) color \tilde{C}_k^t and opacity α_k^t after k passes is given by

$$\tilde{C}_k^t = \tilde{C}_{k-1}^t + (1 - \alpha_{k-1}^t) \tilde{C}_k^t \quad \text{and} \quad \alpha_k^t = \alpha_{k-1}^t + (1 - \alpha_{k-1}^t) \alpha_k^t,$$

where \tilde{C}_k^t and α_k^t denote the associated color and the opacity of the ray segment processed in the k -th pass.

3.4 Cell Traversal

As mentioned above, the traversal of the viewing rays through the tetrahedral mesh is performed by successively rendering screen-sized rectangles. Each rendering pass progresses one layer of cells of the mesh. The traversal is controlled by a current cell index per viewing ray that is stored in intermediate textures of the same size as the frame buffer.

The fragment program applied in each rendering pass reads the current index and updates each texel of the texture with the index of the cell adjacent to the face through which the viewing ray leaves the current cell. This index is given by $a_{t,i}$ for the current cell t with i being the index of the minimum intersection λ_i computed during ray integration.

Note that for boundary cells this index is -1 ; thus, we can easily determine whether a viewing ray has left the mesh and the ray processing for a particular pixel should end. In this case the new exit point should be set to the old entry point resulting in a zero length ray segment without effect on the frame buffer content. A more efficient solution which exploits the early z-test of modern graphics adapters is presented in Section 5.2

However, this procedure is only useful for convex meshes as there may be re-entries if the mesh is non-convex. We convert non-convex meshes into convex-meshes by filling the empty space between the boundary of the mesh and a convex hull of the mesh with imaginary cells as proposed in [Williams 1992]. In the rendering pass we have to take special care of imaginary cells since these cells must not contribute to the color and opacity of the ray. However, we have to compute the exit point and the next cell in the same way as for ordinary cells.

The ray traversal has to continue until all rays have left the mesh. An appropriate test can be performed in software by evaluating the component of the intermediate textures specifying the current cell. However, this requires an expensive read back from the graphics adapter. Fortunately, modern graphics adapters can provide feedback, e.g., via the occlusion query functionality, which can be exploited to evaluate the stop condition in hardware as demonstrated in Section 5.2.

Furthermore, the current cell index allows us to easily incorporate techniques for early ray termination, which has the potential of greatly improving rendering performance due to the reduced rasterization. We simply have to set the current cell in the intermediate texture to -1 , if the accumulated opacity is sufficiently high.



Figure 5: A non-convex FE-dataset consisting of 124 K tetrahedra is rendered with a three-dimensional pre-integration lookup table implementing a volume density-emitter model.

4 Incremental Pre-Integration

As mentioned in Section 3.3, the color and opacity contribution of each ray segment is determined by a pre-integrated texture lookup as suggested in [Röttger et al. 2000].

We employ transfer functions $\tau(s)$ and $c(s)$, which depend on scalar values $s = s(\mathbf{x})$ for any point \mathbf{x} . $\tau(s)$ specifies extinction coefficients and color densities (per unit length) while emitted colors are specified by $c(s)$. Thus, the associated color \tilde{C} and opacity α of a ray segment within a tetrahedron are

$$\tilde{C} = \int_0^1 \tau((1-\omega)s_f + \omega s_b) c((1-\omega)s_f + \omega s_b) l \times \exp\left(-\int_0^\omega \tau((1-\omega')s_f + \omega's_b) l d\omega'\right) d\omega \quad (3)$$

and

$$\alpha = 1 - \exp\left(-\int_0^1 \tau((1-\omega)s_f + \omega s_b) l d\omega\right) \quad (4)$$

with the scalar value s_f at the entry point, the scalar value s_b at the exit point, and the length l of the ray segment. Apart from the transfer functions $\tau(s)$ and $c(s)$, \tilde{C} and α depend only on s_f , s_b , and l ; thus, one three-dimensional texture is sufficient for a table lookup of these integrals. As this lookup table has to be updated whenever the transfer functions $\tau(s)$ or $c(s)$ are modified, an efficient evaluation of Equations (3) and (4) is critical for many volume visualization applications.

Here we present *incremental pre-integration*, which is one of the most efficient acceleration techniques for the computation of lookup tables for $\tilde{C} = \tilde{C}(s_f, s_b, l)$ and $\alpha = \alpha(s_f, s_b, l)$ according to Equations (3) and (4). Variants of this technique were developed independently by several researchers [Guthe 2002] but, to our knowledge, have not been published previously.

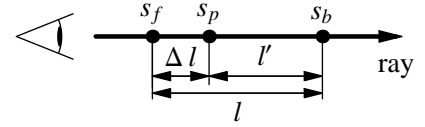


Figure 6: Splitting of a ray segment of length l into two parts of length Δl and l' , respectively.

Provided that a lookup table has been calculated for all entries with lengths less or equal l' , the entries for the next length $l = l' + \Delta l$ can be calculated by splitting the integrals into one part of length Δl and one part of length l' ; see Figure 6. The scalar value s_p at the split point is interpolated linearly between s_f and s_b , i.e., $s_p = (l' s_f + \Delta l s_b) / (l' + \Delta l)$. As the integrals for these parts are already covered by the lookup table, the evaluation is reduced to interpolations of tabulated integrals and a blending operation. More specifically, $\tilde{C}(s_f, s_b, l)$ and $\alpha(s_f, s_b, l)$ are given by

$$\tilde{C}(s_f, s_b, l' + \Delta l) = \tilde{C}(s_f, s_p, \Delta l) + (1 - \alpha(s_f, s_p, \Delta l)) \tilde{C}(s_p, s_b, l'), \quad (5)$$

$$\alpha(s_f, s_b, l' + \Delta l) = \alpha(s_f, s_p, \Delta l) + (1 - \alpha(s_f, s_p, \Delta l)) \alpha(s_p, s_b, l'). \quad (6)$$

Note that $\tilde{C}(s_f, s_b, 0)$ and $\alpha(s_f, s_b, 0)$ are 0 for all s_f and s_b , and that $\tilde{C}(s_f, s_b, \Delta l)$ and $\alpha(s_f, s_b, \Delta l)$ have to be calculated by a numeric evaluation of the integrals in Equations (3) and (4).

The interpolation of tabulated integrals for the table lookups in Equations (5) and (6) introduces an error depending on the chosen resolution of the table. Note that the errors associated with the lookups for $\tilde{C}(s_f, s_p, \Delta l)$ and $\alpha(s_f, s_p, \Delta l)$ are not critical since these errors cannot accumulate. However, the errors associated with the lookups for $\tilde{C}(s_p, s_b, l')$ and $\alpha(s_p, s_b, l')$ are potentially accumulating. In practice, this error accumulation is prevented by the blending operations in Equations (5) and (6) due to the chosen order of the two parts of length Δl and l' , respectively.

5 Implementation Issues

In the description of our parallel ray casting algorithm for fragment shading hardware in Section 3 we tried to avoid most of the distracting hardware-specific details. This section will deal with issues of an implementation on an ATI Radeon 9700 with DirectX 9 and the Pixel Shader 2.0 shading language.

The feature set of Pixel Shaders 2.0 allows for a quite straightforward implementation of the operations described in Section 3, since several restrictions that applied to the previous version—Pixel Shaders 1.4—have been relaxed, in particular the number of different texture maps that can be addressed (16 in Pixel Shader 2.0), and the number of per-fragment operations provided per pass (32 texture addressing instructions and 64 color blending/arithmetical instructions). Moreover, the current pixel shader API does no longer restrict the interleaving of texture and arithmetical operations and supports floating point precision throughout all steps of the geometry and rasterization pipeline. We will, therefore, not present particular pixel shader code in this paper.

The essential feature of DirectX 9 for our algorithm is the support of textures with full 32 bit floating point precision since these, for the first time, allow us to store the complete mesh in texture maps, i.e., in the local memory of the graphics adapter, without losing accuracy. They also allow us to randomly access the data from a fragment program. However, several restrictions—e.g., only nearest neighbor sampling and no blending—apply to floating point textures depending on the actual graphics hardware and, therefore, had impact on our implementation.

5.1 Data Structures

All data structures required by our hardware-based ray caster, except for the pre-integration table, are implemented either as two-dimensional or three-dimensional floating point RGBA texture maps. The pre-integration table is stored in an 8 bit RGBA texture since smooth ray integration depends on the trilinear interpolation. All other textures are mainly used for indexed lookup; thus, the lack of linear interpolation for floating point textures on the Radeon 9700 is actually no problem here. Table 1 provides an overview of the required textures. Note that the cell indices are encoded in two texture coordinates and two color components as their range might easily exceed the range of a single texture coordinate.

data in texture	texture coords			texture data			
	u	v	w	r	g	b	α
vertices	t	i		$\mathbf{v}_{t,i}$			—
face normals	t	i		$\mathbf{n}_{t,i}$			$f_{t,i}$
neighbor data	t	i		$a_{t,i}$	—	—	—
scalar data	t	—		\mathbf{g}_t			\hat{g}_t
current cell	raster pos	—		t	j	—	—
intersect. pt.	raster pos	—		—	—	$\hat{\lambda}$	$s(\mathbf{e} + \hat{\lambda}\mathbf{r})$
color, opacity	raster pos	—		r	g	b	α

Table 1: Summary of the textures described in the main text.

The lower index of a cell is the remainder of the cell index divided by the v -size of the texture, the upper index is the integer part of the division result. The texture maps can be divided into constant textures storing mesh data and intermediate textures that communicate data between different rendering passes.

Mesh data consists of three three-dimensional textures for vertices, face normals, and neighbor data; and one two-dimensional texture for scalar data reconstruction (see Section 3.3). The normal texture additionally stores the cell type—imaginary or not—as the sign bit in the alpha channel of each texel. The textures are accessed via the cell index specified in the first and second texture coordinate. For three-dimensional textures the w -coordinate specifies the index of the vertex, face, or neighbor, respectively. Thus, the third dimension of all three-dimensional textures is fixed at a size of four since each tetrahedron specifies four vertices, four faces, and four neighbors. The u - and v -size has to be adapted to the size of the mesh.

The traversal data structures consist of three two-dimensional RGBA floating point textures. They are accessed by the raster position of the corresponding pixel. Unfortunately—unlike the OpenGL `ARB_fragment_program` extension—Pixel Shaders 2.0 do not automatically provide the raster position of a fragment; thus, we have to specify the raster position of the four vertices of the rendered rectangle as texture coordinates such that they are interpolated for each fragment.

The first traversal texture specifies the most recently entered cell for each viewing ray; the second texture specifies the position of and the scalar value at the last intersection point of each viewing ray with the faces of cells. Note that in contrast to the description in Section 3 we are not able to use the frame buffer for accumulating the color and opacity of the integrated ray segments since there is no floating point format available for the frame buffer. A precision of 8 bits, however, would lead to blending artifacts; in particular, as the contribution of a single ray segment usually is very small. Furthermore, the Radeon 9700 does not support simultaneous rendering to textures and the frame buffer. Thus, we employ an additional floating point texture for color and opacity accumulation.

All these textures are read and updated during each rendering pass. Therefore, we in fact require two sets of traversal data structures, i.e., two sets of textures, since current graphics adapters do not allow to read from and write into a texture at the same time. We apply ping-pong rendering by binding one set of textures for reading and specifying the second set as multiple render targets for writing and exchange the role of the texture sets after every rendering pass.

In order to access the normalized ray direction within the fragment program, we decided not to use a texture map because of the additional memory overhead. Instead we compute the direction in each rendering pass on-the-fly. The additional computational overhead can be neglected since we exploit the up-to-now hardly stressed geometry unit by computing the ray direction for each vertex within a vertex shader and add only one additional operation to the pixel shader that normalizes the interpolated ray directions.

5.2 Rasterization Loop

Terminating the rendering loop comprises two issues. First, for each fragment we have to detect whether the corresponding viewing ray has left the mesh. In this case ideally no more fragment operations should be spent on the corresponding pixel. Secondly, we must determine whether *all* viewing rays have left the mesh, i.e., whether we can stop the rendering.

For the first task we apply the early z-test provided by the Radeon 9700 graphics adapter [Rieger 2002]. This additional z-test is applied before the shading of the fragment; thus, fragment processing is stopped before any pixel shader is invoked. This solution

is much more efficient than using the pixel shader “kill” operation.

Exploiting the early z-test for detecting fully processed viewing rays is accomplished by checking within the pixel shader, whether there is no next cell, i.e., whether the neighbor index $a_{t,j}$ is -1 with i being the index of the face through which the view ray leaves the current cell t . In this case, the z-value of the corresponding pixel is altered to z_{near} which prevents further updates of the pixel color. However, this cannot be performed within the regular ray processing pixel shader, since the early z-test is automatically disabled for pixel shaders modifying the z-value [Riguer 2002]. Thus, we apply a special z-update pass, which reads from the source textures but only writes to the z-buffer. As mentioned above, we update the z-value of each pixel with a corresponding current cell index of -1 to z_{near} ; otherwise, i.e., if there is a next cell the pixel’s z-value is not updated. The latter is accomplished by exploiting the z-test in the z-update pass since we assign a value of z_{far} to the corresponding fragments, which are therefore discarded.

Since this additional pass introduces some overhead in terms of rasterization and pixel shader switches, z-update passes are not applied after every rendering pass. We experienced the greatest performance improvements with one z-update pass about every 14 rendering passes. Pixels not covered by the silhouette of the tetrahedral mesh are already locked by an additional z-update pass right after the rendering pass determining the first hit of each ray.

Note that if a cell index of -1 has been read, the shader has to copy the accumulated color and opacity from the source texture to the destination texture because the destination texture may still contain the results from the last but one ray integration step. However, it is not guaranteed that the copying takes place because the pixel might already be locked by the z-buffer. Therefore, in order to ensure a correct final image, we apply z-update passes only after odd rendering passes, i.e., after rendering into the first set of textures and we also expect the final result there. This is obviously correct for rays that require an odd number of rendering passes. The final result for rays requiring an even number of passes will eventually be written into the second set of textures. However, since the rendering loop will be terminated after odd passes only, at least one additional pass will be applied copying the result into the first set of intermediate textures.

In order to determine whether all rays have been fully processed, we employ the DirectX 9 occlusion query. With every z-update pass an occlusion query (`D3DQUERYTYPE_OCCLUSION`) is initiated returning the number of pixels that passed the z-test. As only fragments corresponding to viewing rays which left the mesh since the last z-update pass will pass the z-test, the result exactly reflects the number of viewing rays that have recently become finished. Accumulating the results of the occlusion queries allows us to stop the rendering as soon as the sum reaches the number of pixels covered by the projected mesh. We determine the latter with an additional occlusion query for the initial rendering pass that is used for detecting the first hit of the viewing rays with the boundary of the tetrahedral mesh.

Note that the asynchronous delivery of the occlusion query result leads to some additional rendering passes. However, we found this effect neglectable compared to the delay caused by waiting for the result.

6 Results

We have tested our implementation on a PC with an Athlon XP 1900+ processor and 512 MB RAM running Windows XP. However, the influence of the CPU is neglectable since our approach moves most of the computation to the graphics processor. As graphics adapter an ATI Radeon 9700 Pro with 128 MB of local memory was employed. The implementation was built on top of DirectX 9.

	total no. cells	imaginary	fps	tets/sec
Bluntfin (Fig. 1)	190 852	3 534	3.37	643 K
Heat-sink (Fig. 9)	124 152	2 484	2.27	282 K
Cylinder (Fig. 9)	203 460	1 860	1.94	395 K
Sphere (Fig. 1)	148 955	—	5.13	764 K

Table 2: Performance of our hardware-based ray casting algorithm for different datasets using a Radeon 9700 Pro graphics adapter. The cell numbers include imaginary cells from the convexification process. The referenced figures demonstrate the rendering mode and the employed transfer functions.

Table 2 shows the performance of our hardware-based ray casting algorithm for different datasets presented in the paper. Sphere, heat-sink, and bluntfin denote the datasets of Figure 1, whereas the cylinder dataset is shown in Figure 9. Note that the performance is merely affected by the optical model except for the full density-emitter model shown in the middle and right image of Figure 1 since the required three-dimensional dependent texture lookup is more expensive than the two-dimensional lookup used for the other optical models. However, as our ray casting approach includes early ray termination the performance is influenced by the selected transfer function since the traversal terminates if the accumulated opacity is sufficiently high. The table, therefore, contains references to the corresponding images.

The performance numbers in Table 2 represent average values for varying view directions. They have been acquired by an animation where the datasets are rotated simultaneously about the local x-axis and y-axis with different angular velocities for each axis. The actual frame rate, or frame time respectively, almost directly corresponds to the number of calls to the ray traversal shader, which is influenced by the number of pixels covered by the projected dataset and the number of passes required for each pixel. The latter depends on the dataset itself and the visualization parameters that influence the effectiveness of the early ray termination. On average our implementation was able to perform about 15 million traversal shader calls per second on the test system. A typical variation of the frame time is presented in Figure 7. It corresponds to an animation sequence for the heat-sink dataset of Figure 9.

The effect of early ray termination is illustrated in Figure 8, which shows the number of fragments processed in each rendering

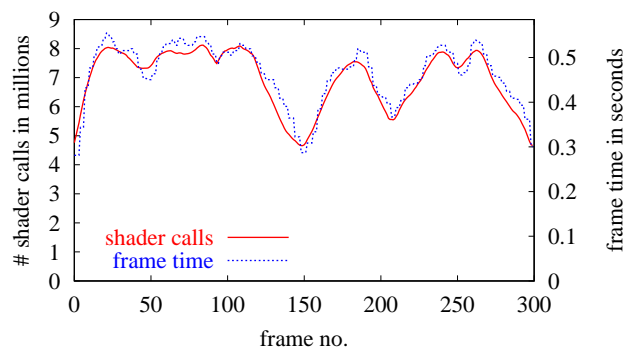


Figure 7: The figure shows the variation of the frame times for varying view directions on the heat-sink dataset rendered on a 512×512 viewport. The corresponding image is shown in Figure 9. The curve shows good correspondance between the frame time and the total number of traversal operations. Note that the minimum and maximum of the frame times correspond to 1.80 fps and 3.55 fps respectively.

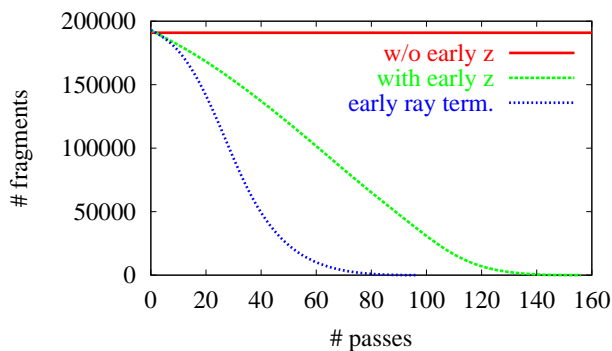


Figure 8: The figure shows statistics for the heat-sink dataset rendered on a 512×512 viewport. The corresponding image is shown in Figure 9. The curves reflect the number of fragments processed in each rendering pass. As can be seen from the area below the curves, exploiting the early z-test significantly reduces the total amount of rasterized fragments.

pass for the heat-sink dataset in Figure 9. Without early ray termination, for this particular set of viewing parameters, our ray casting algorithm requires 156 rendering passes to finish (green line). Thereby, compared to the brute-force fragment processing without an early-z test that discards rays that have left the mesh (red line), the load on the fragment processor is reduced by a factor of 2. When early ray termination is enabled the total fragment load is further reduced by a factor of 2 and the rendering completes after 96 passes (blue line).

Despite the advantages from early ray termination, currently, our ray casting algorithm can in many situations be outperformed by optimized cell projection algorithms, e.g., as reported by [Guthe et al. 2003]. This is in particular the case for large viewports, since our algorithm is highly output sensitive. However, cell projection approaches, especially for non-commutative optical models, are limited by the visibility sorting, which already prevents the current graphics adapters from running at full capacity, due to limited CPU resources and graphics bus bandwidth. Thus, projecting the development of CPUs and graphics adapters during the last years our ray casting approach, which is only limited by the rasterization performance of the graphics adapter, may soon catch up.

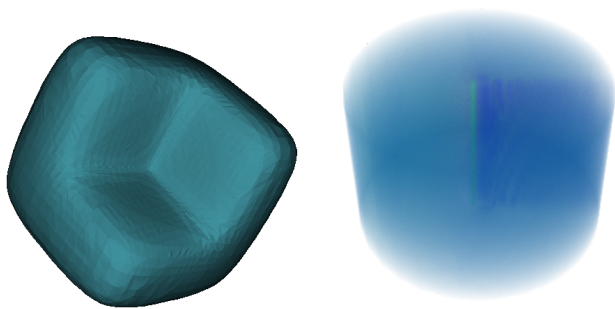


Figure 9: The left image shows an isosurface rendering of the heat-sink dataset. In the right image the cylinder dataset is presented. The hole in the middle has been filled with imaginary cells in order to allow for re-entries during the cell traversal. Both datasets are rendered on a 512×512 viewport.

One of the main advantages of our approach is the low bandwidth requirement for the graphics bus that allows the graphics processor to run at full capacity. This, however, limits the size of suitable datasets since interactive frame rates cannot be achieved if texture paging is necessary.

Provided that the frame buffer consumes 12.5 MB of the graphics memory (2×32 bit RGBA + 16 bit depth for a standard 1280×1024 desktop), we are able to store between 500,000 and 600,000 tetrahedra depending on the optical model used. We apply a 8 bit $128 \times 128 \times 256$ ($\hat{=}$ 16 MB) RGBA texture for the full density-emitter model since this requires a three-dimensional pre-integration table, whereas for the remaining models a two-dimensional RGBA texture ($256 \times 256 \hat{=} 0.25$ MB) is sufficient since the length of the ray segment affects the color contribution either linearly or not at all.

Unfortunately, with DirectX 9 all textures for intermediate data must share the same pixel format in order to serve as render targets, which produces quite some memory overhead since according to Table 1 we only need 9 of the 12 provided color components per texture set. Moreover, 16 bit short values would provide enough precision for cell and face indices since we only have to address 2048 different texels, the maximum dimension of a texture. Thus, the memory required for one pixel in the output image is 2×48 bytes according to the following summary:

$2 \times$ current cell	4×4 bytes	vertices	$4 \times 4 \times 4$ bytes
$2 \times$ intersect. pt.	4×4 bytes	normals	$4 \times 4 \times 4$ bytes
$2 \times$ color, opacity	4×4 bytes	neighbors	$4 \times 2 \times 2$ bytes
		scalar data	$1 \times 4 \times 4$ bytes
$2 \times$	$48 \frac{\text{bytes}}{\text{pixel}}$	total	$160 \frac{\text{bytes}}{\text{tetra}}$

Each tetrahedron requires 160 bytes of memory. As mentioned above, most mesh data is stored in RGBA floating point textures; however, neighbor data only requires a two-component 16 bit texture.

Note that limited fragment operations have prohibited further optimizations, e.g., data packing, storing only one instance of each vertex and referencing the vertices of each cell by an index in this texture, encoding normals with two components and one sign bit, or using a single 32 bit value for cell indices and face index restoring the individual components in the pixel shader on-the-fly.

These arrangements would have decreased the memory consumption to 16 bytes/vertex, 64 bytes/tetra, and 2×32 bytes/pixel allowing for 1,280,000 to 1,530,000 tetrahedra on a 128 MB graphics adapter without loss of accuracy.

The incremental pre-integration was tested for a lookup table of $128 \times 128 \times 256$ on an Athlon XP 1900+. Computing each entry individually took about 128 sec., where the transfer function was sampled with the Nyquist frequency, but the number of integration steps was not increased with the length of each viewing ray segment. With our incremental pre-integration approach the computation took only 1.5 sec. to complete. In this case we perform an explicit integration only for the first slice of the texture corresponding to $\frac{1}{256}$ of the maximum segment length. Combining two such ray segments during the incremental computation is equivalent with the numerical integration using twice as much samples along the length of the ray segment. Thus, the complete integration table is computed with a fixed sample distance along the ray segment and, therefore, the results are much more accurate. Note that the corresponding numerical integration of each entry of the integration texture with a fixed sample distance along the ray takes hours to complete.

7 Conclusions

We have presented the first implementation of a volume ray casting algorithm for tetrahedral meshes running on off-the-shelf programmable graphics hardware. In contrast to cell projection approaches our algorithm allows the graphics adapter to access all mesh data from its local memory. Thus, it can run at full capacity since it is neither limited by the bandwidth of the graphics bus nor by the CPU. Cell projection, however, is computational and bandwidth bound since the CPU has to provide sorted cells and a lot of triangles must be transferred to the graphics chip per frame. Therefore, we expect our approach to benefit more from the increase in performance of graphics hardware than cell projection algorithms.

On the other hand, cell projection approaches will benefit more from higher CPU speeds and improved graphics bus bandwidths since the rasterization is more expensive for our approach and we are—due to the greater amount of texture lookups—additionally limited by the internal memory bandwidth of the graphics adapter. Furthermore, although we can exploit early ray termination, with our approach the total number of fragments can be higher since more fragments might be generated per pixel than the depth complexity of the mesh requires. In contrast to this, cell projection guarantees to rasterize exactly the front faces of all cells.

Texture memory is a crucial limitation of our approach since all mesh data for a dataset has to fit into the local memory of the graphics adapter in order to allow for interactive visualizations. However, memory resources on graphics adapters are also increasing but may not keep up with the size of the datasets to be visualized. In summary, our approach is of particular interest for visualizing small, possibly cyclic, convex meshes with high opacities, e.g., due to opaque isosurfaces.

References

- GARRITY, M. P. 1990. Raytracing Irregular Volume Data. In *Proceedings of the 1990 Workshop on Volume Visualization*, ACM Press, 35–40.
- GUTHE, S., RÖTTGER, S., SCHIEBER, A., STRASSER, W., AND ERTL, T. 2003. High-Quality Unstructured Volume Rendering on the PC Platform. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '02*, 119–125.
- GUTHE, S. 2002. personal communication.
- KING, D., WITTENBRINK, C. M., AND WOLTERS, H. J. 2001. An Architecture for Interactive Tetrahedral Volume Rendering. In *Proceedings of the International Workshop on Volume Graphics 2001*, Springer-Verlag, K. Mueller and A. Kaufman, Eds., 163–180.
- KRAUS, M., AND ERTL, T. 2002. Implementing Ray Casting in Tetrahedral Meshes with Programmable Graphics Hardware. Tech. Rep. 1, Visualization and Interactive Systems Group at the University of Stuttgart.
- PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray Tracing on Programmable Graphics Hardware. In *Proceedings of ACM SIGGRAPH 2002*, vol. 21, 703–712.
- RIGUER, G. 2002. Performance Optimization Techniques for ATI Graphics Hardware with DirectX 9.0. available at <http://mirror.ati.com/developer/techpapers.html>.
- RÖTTGER, S., KRAUS, M., AND ERTL, T. 2000. Hardware-Accelerated Volume and Isosurface Rendering Based On Cell-Projection. In *Proceedings IEEE Visualization 2000*, ACM Press, 109–116.
- RÖTTGER, S., GUTHE, S., WEISKOPF, D., AND ERTL, T. 2003. Smart Hardware-Accelerated Volume Rendering. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '03 (to appear)*.
- WEILER, M., KRAUS, M., AND ERTL, T. 2002. Hardware-Based View-Independent Cell Projection. In *Proceedings of IEEE Symposium on Volume Visualization 2002*, IEEE, 13–22.
- WEILER, M., KRAUS, M., MERZ, M., AND ERTL, T. 2003. Hardware-Based View-Independent Cell Projection. *IEEE Transactions on Visualization and Computer Graphics (Special Issue on IEEE Visualization 2002)* 9, 2 (April-June), 163–175.
- WILLIAMS, P. L., AND MAX, N. 1992. A Volume Density Optical Model. In *ACM Computer Graphics (1992 Workshop on Volume Visualization)*, 61–68.
- WILLIAMS, P. L. 1992. Visibility Ordering Meshed Polyhedra. *ACM Transactions on Graphics* 11, 2, 103–126.
- WITTENBRINK, C. M. 2001. R-Buffer: A Pointerless A-Buffer Hardware Architecture. In *Proceedings Graphics Hardware 2001*, ACM Press, 73–80.