GPU-based Tiled Ray Casting using Depth Peeling

Fábio F. Bernardon, Christian A. Pagot, João L. D. Comba, Cláudio T. Silva

UUSCI-2004-006

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

July 26, 2004

Abstract:

In this paper we propose several significant improvements to the hardware ray casting algorithm for unstructured meshes proposed byWeiler et al [14]. In their work, ray casting computation is entirely performed in the GPU by advancing intersections against the mesh while evaluating the volume rendering integral. Our contributions can be divided into three categories. First, we propose an alternate representation for mesh data in 2D textures that is more compact and efficient, compared to the 3D textures used in the original work. Second, we use a tile-based subdivision of the screen that allows computation to proceed only at places where it is required, thus reducing fragment processing in the GPU. Finally, we do not introduce imaginary cells that fill space caused by non-convexities of the mesh. Instead, we use a depth-peeling approach that captures when rays re-enter the mesh, which is much more general and does not require a convexification algorithm.

We report results on an ATI 9700 Pro, the same hardware used by Weiler et al in their work. Due to the use of the 2D textures and the tiling, our technique is actually much faster than their work, while at the same time being more general, since it can render true non-convex meshes, as compared to their work, which is limited to convex (or *convexified*) ones. On the Blunt Fin, our code renders between 400 Ktet/sec to 1.3 Mtet/sec.

GPU-based Tiled Ray Casting using Depth Peeling [*]

Fábio F. Bernardon UFRGS | Christian A. Pagot UFRGS | João L. D. Comba UFRGS | Cláudio T. Silva University of Utah

Abstract

In this paper we propose several significant improvements to the hardware ray casting algorithm for unstructured meshes proposed by Weiler et al [14]. In their work, ray casting computation is entirely performed in the GPU by advancing intersections against the mesh while evaluating the volume rendering integral. Our contributions can be divided into three categories. First, we propose an alternate representation for mesh data in 2D textures that is more compact and efficient, compared to the 3D textures used in the original work. Second, we use a tile-based subdivision of the screen that allows computation to proceed only at places where it is required, thus reducing fragment processing in the GPU. Finally, we do not introduce imaginary cells that fill space caused by non-convexities of the mesh. Instead, we use a depth-peeling approach that captures when rays re-enter the mesh, which is much more general and does not require a convexification algorithm.

We report results on an ATI 9700 Pro, the same hardware used by Weiler et al in their work. Due to the use of the 2D textures and the tiling, our technique is actually much faster than their work, while at the same time being more general, since it can render true non-convex meshes, as compared to their work, which is limited to convex (or *convexified*) ones. On the Blunt Fin, our code renders between 400 Ktet/sec to 1.3 Mtet/sec.

1 Introduction

The advent of modern GPUs with their high degree of parallelism is causing major shifts in the efficiency of algorithms in computer graphics and scientific visualization. Techniques that were previous seen as too slow for mainstream use, are now being considered practical. This is starting to be the case with ray tracing [11]. GPU-based implementation of these algorithms is often non-trivial, because the same features that make the GPUs so fast, also cause them to be hard to target for algorithms. Trying to exploit the high levels of bandwidth and float-point performance available turns out to be quite hard, since the general *flow* of the algorithms has to be substantially modified to enable efficient execution.

In this paper, we are particularly interested in the volume rendering of unstructured meshes. Unstructured grids are extensively used in modern computational science and, thus, play an important role in scientific computing. They come in many types, and one of the most general types are non-convex meshes, which may contain voids and cavities. The lack of convexity presents a problem for several algorithms, often causing performance issues [2]. One of the earliest techniques for rendering unstructured meshes was the original ray casting algorithm proposed by Garrity [6]. This technique was based on exploiting the intrinsic connectivity available on the mesh to optimize ray traversal, by tracking the entrance and exit points of a ray from cell to cell. One difficulty was the fact that some meshes are non-convex, and a ray can get in and out of the mesh multiple times. Garrity uses a spatial data structure to optimize this operation. This

[*] `{fabiofb,capagot,comba}@inf.ufrgs.br; csilva@cs.utah.edu`

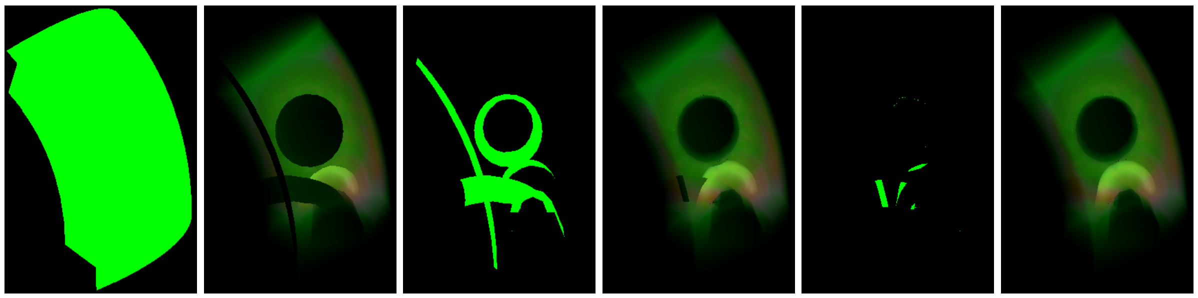


Figure 1: Volume Rendering using Depth Peeling

work was further improved by Bunyk et al [1]. Instead of building a hierarchical spatial data structure of the boundary cells, they simply determine for each pixel in the screen, the fragments of the front boundary faces that project on any given pixel, and order those fragments in a front-to-back order. During rendering, each time a ray exits the mesh, Bunyk uses that information for determining whether the ray will re-enter the mesh, and where (i.e., the particular cell). Bunyk's approach turns out to be simpler, faster, and more robust to floating-point error than the Garrity's original approach.

Weiler et al [14] proposes a hardware-based technique based on the work of Garrity. They show how to store the unstructured mesh directly in texture memory using 3D textures. They also show how to store all the information necessary for actually tracing the rays using 3D textures. One of the key features of their technique is the fact that all the processing is GPU based, and there is no need for information to be send back and forth between the CPU and GPU. Since GPU performance growth outpaces the CPU's, this is often a desirable property. One of the shortcomings of their approach, though, is that their GPU-based implementation can only handle convex meshes, i.e., it is not able to find the re-entry of rays that leave the mesh only temporarily. In their paper, they use a solution originally by Williams [16] that calls for the convexification of the mesh, and the markings of exterior cells as *imaginary* so they can be ignored during rendering. Unfortunately, convexification of meshes has many unresolved issues that make it a hard problem. We point the reader to Comba et al [2] for details. For the experimental results, Weiler *manually* convexified the meshes. Ideally, one would like a completely automatic procedure.

In this paper, we propose a novel GPU-based algorithm for rendering unstructured meshes. Our work builds on the previous work of Weiler et al [14], but improves it in several significant ways. Our rendering algorithm is based on the work of Bunyk et al [1]. This provides a better match for a hardware implementation, since we can use depth peeling [9] (see figure 1) to generate the ordered sequence of fragments for the front boundary faces, and depth peeling can be efficiently implemented in hardware [3, 7]. Our technique shares with Weiler et al. the fact that all the processing is GPU-based, but it subsumes their technique in that we can handle general non-convex meshes. As can be seen below, we have also substantially optimized the storage of data as to allow the use of 2D textures, leading to subtantially improved rendering rates.

Our paper is organized as follows. We briefly review previous work in section 2. In section 3 we outline the tiled ray casting algorithm, and describe its GPU implementation in the following section. Section 5 presents results obtained using several datasets, followed by a discussion section that evaluates the results. Conclusions are presented at the end.

2 Related Work

Our work is inspired on the hardware ray casting algorithm proposed by Weiler et al [14], which was based on the original ray casting algorithm proposed by [6]. Weiler presented the first implementation of a volume ray casting algorithm for unstructured tetrahedral meshes entirely based on commodity hardware graphics. The tetrahedral mesh data and pre-integrated tables are packed inside textures, improving its performance, as the bottleneck caused by data transfer data between GPU and CPU is avoided. Another positive point of the algorithm is that it takes advantage of the parallel nature of the ray casting algorithm, keeping the GPU in charge of all the data processing steps (ray traversal and ray integration). The algorithm is capable of rendering convex meshes. Non-convex meshes must be convexificated first. This convexification is made through an expensive and non-trivial pre-processing step, where imaginary cells are added to the empty spaces between the boundary of the mesh and a convex hull of the mesh. (After we finished our work in late March 2004, we learned that concurrently with us, Weiler used a similar depth-peeling approach – described later – to dealing with non-convex meshes; their work will be reported in an upcoming paper [15]. The preliminary version of their paper we had access to reports substantially slower running times than ours. Part of the rendering time disparate potentially is a result of the fact that they implement the depth-peeling quite differently, in particular, there is no use of tiling, and the implementation is not done in a shader, but using the standard pipeline. Also, they use a more complex data structure required by their mesh compression scheme.) Mesh processing is made through several ping-pong rendering steps, where the GPU advances over the mesh, integrating all the cells found inside it. The algorithm takes advantage of the early-z test to avoid the cost of computations related to rays that miss the mesh, or that traversed it completely. As the mesh data (vertices, normals, faces, etc.) is kept inside textures, the maximum size of a mesh is constrained by the amount of video memory available. It means that special care must be taken when choosing how to layout the data inside textures. Our method presents a new layout for packing the data inside textures that can save up to 30% of GPU memory, allowing the storage of bigger data sets.

The early z-test feature, found on almost all current consumer graphics hardware, was also used by Krueger and Westermann [8] in a proposal of two acceleration techniques for volume rendering, including an efficient ray casting approach. Through this feature they were able to improve volume rendering performance by avoinding the computations related to rays that miss the volume, or rays that have generated enough opacity during the traversal. Another approach for ray casting regular grids is contained in Roettger et al [12].

In some circumstances image tiling becomes a very effective way to improve performance, taking advantage of memory coherence and better cache usage. Farias, Mitchell and Silva [4] proposed a ZSWEEP algorithm to perform cell projection for the rendering of unstructured data sets. Later Farias and Silva reviewed the ZSWEEP algorithm, and proposed an image-based task partitioning scheme for the algorithm [5], making it suitable for distributed shared-memory machines. In this new scheme the screen was divided into several tiles, that could be dynamically sent to several processors, allowing for parallel processing. Detailed analysis of the tile-based version of the algorithm showed improvements in memory coherence and an increase in the rendering performance, even on single-processor machines. Another example of performance improvement due to the use of tiling was demonstrated by Krishnan, Silva and Wei [7]. They proposed a hardware-based visibility ordering algorithm, capable of computing the visibility order of an acyclic set of geometric primitives through the attribution of layer numbers to those primitives. Primitives that have the same layer number don’t obstruct each other. Those ordering layers are obtained through several rendering passes, with the help of depth and stencil buffers. A more efficient version of the algorithm, that takes advantage of the memory coherence through the use of tiling, is also presented.

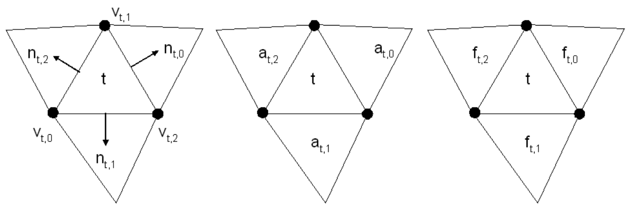


Figure 2: Mesh Notation

Nagy and Klein [10] developed a new approach for the visualization of the n-th iso-layer of a volumetric data set based on the depth peeling technique proposed by [9] and later implemented in hardware by [3]. The speed is the main advantage of this algorithm, since only on pass is required for the visualization of any number of layers. As the method is conceived to render interior and exterior iso-surfaces, it is not capable to integrate through the volumetric data set.

3 Tiled Ray Casting Algorithm

In this section we describe the main issues in the design of the ray casting algorithm, without going into the details of its GPU implementation, discussed in the following section.

3.1 Notation

Mesh data can be referenced using the following notation. A mesh M is composed of n_t, n_v tetrahedral cells (tetra) and vertices. Each tetra can be referenced by an index (t_i). For each tetra, f_i represents the $i^{th}(i = 0..3)$ face of a tetrahedra, $n_{t,i}$ represents the normal of the i^{th} face (always pointing outside), the vertex opposed to i^{th} face is referred to as $v_{t,i}$, and $a_{t,i}$ the neighbor to the face i neighbor index. (Figure 2).

The parametric equation of a line will be used to describe casting rays, using the origin of the ray (eye or e) and a normalized direction (ray or r). A point x over a given ray is computed using the equation $x = e + \lambda r$. Important to the ray casting algorithm are two coordinate systems that will be used for intersection calculations. The Object Coordinate System (OCS) represents the space where the mesh is defined, and the world coordinate system (WCS) the space obtained after applying geometric transformations to the mesh. Rays defined in the OCS (WCS) will be defined by an eye and ray referred to as e_{ocs} (e_{wcs}) and r_{ocs} (r_{wcs}) respectively (Figure 3).

3.2 Ray-Mesh Intersections

The ray casting algorithm is designed to explore the parallelism of modern GPUs. For a given screen resolution, rays are defined for each pixel from the eye into the center of the pixel. The first intersection of each ray against the mesh is computed for all rays. The algorithm proceeds by simultaneously computing the point that each ray leaves the tetra, and advancing one intersection at a time. The process ends when all rays leave the mesh.

A ray-tetra intersection can be done by computing four intersections of a given ray against the supporting planes of tetra faces [13]. Each parametric intersection λ can be computed using the following equation:

$$\lambda_i = \frac{(v - e) \cdot n_{t,i}}{r \cdot n_{t,i}} \quad where\ v\ =\ v_{t,3-i} \tag{1}$$

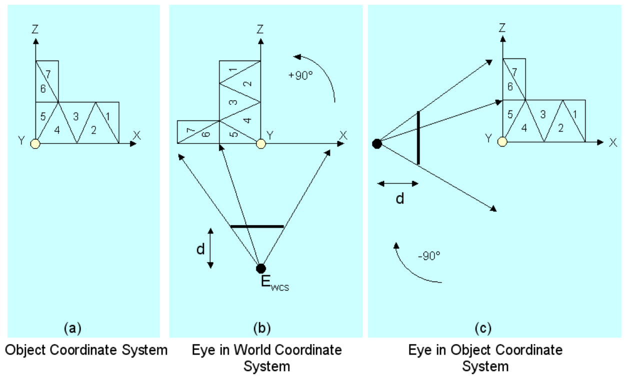

Z
X
Y
7
6
5
4
3
2
1
+90°
d
E_wcs
-90°
(a)
Object Coordinate System
(b)
Eye in World Coordinate System
(c)
Eye in Object Coordinate System

Figure 3: Coordinate Systems used in the ray casting algorithm (a) object coordinate system (b) world coordinate system and (c) eye transformed to object coordinate system

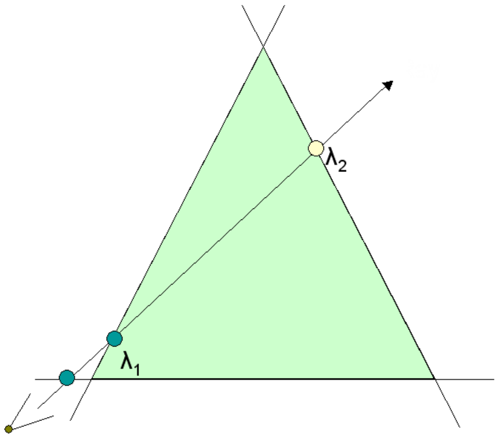


Figure 4: Entry and Exit Points Computation

Only positive λ values need to be considered. If the denominator is negative (positive), it means (using our normal convention) that the ray is entering (leaving) the halfspace of a given mesh. For the first intersection calculation, the maximum entering λ_1 defines the point that the ray enters the tetra. Similarly, the minimum leaving λ_2 defines the exit point. If $\lambda_1 > \lambda_2$ then no intersection exists. An example for a convex region is showed in figure 4.

Since the algorithm proceeds incrementally, computing the exit point of a tetra also requires computing the intersection against all faces, including the entering point. Distinguishing between them is done using the denominator rule described above. Weiler [14] suggests discarding the entering point by checking it against a face index that is explicitly stored, but this is not necessary.

Important to evaluating the volume rendering equation is the value of the scalar value at intersection points. An interpolation procedure of the scalar values at the vertices uses the gradient of the scalar field g_t and a reference point x_0 (which can be one of the vertices). The scalar field value $s(x)$ of a point x at a tetra t is computed as:

$$s(x) = g_t \cdot (x - x_0) + s(x_0) \tag{2}$$

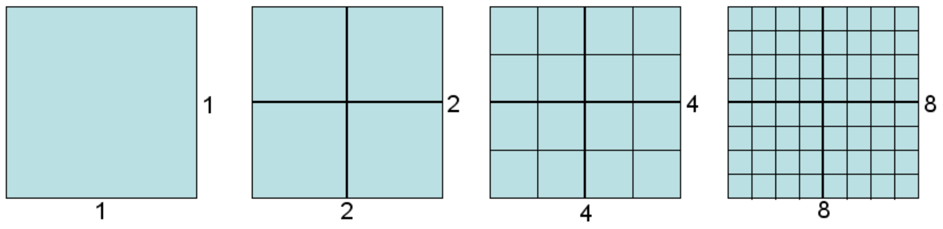


Figure 5: Fixed-size tile partitioning

The distances between the entry and exit point (normalized using the length of the greatest edge of the mesh), along the values of the scalar fields of both points, are used as indexes into a pre-integrated 3D lookup table that returns the color (r,g,b) and opacity (α) contributions. An alternate 2D LUT table can be used instead if we assume a fixed distance value. The returned color (C'_k) is composed using the front-to-back composition formula:

$$C_k = C'_{k-1} + (1-\alpha_{k-1})C_k \tag{3}$$

3.3 Tile Partitioning

Spatial coherence is used in many different ways to accelerate ray casting queries. Several calculations performed by the algorithm proposed by Weiler [14] can be made more efficient if the image screen is partitioned into disjoint tiles, each one performing independent intersection calculations. Since different regions on the screen might have different computation needs, subdiving the screen into tiles can improve this process in several ways. For an area that does not require any computation at all (i.e. no intersection between rays and the mesh), no ray casting calculations are performed. Even if all areas initially require computation, it is likely that the number of iterations until all the rays leave the mesh is different among different tiles. Therefore, allowing each tile to work independently allows a variable number of intersection passes, which better approximates the minimum required number of passes. This also has the effect of issuing occlusion queries at smaller regions, and reducing significantly the number of processed fragments.

The optimal tiling decomposition of the image screen is the one that minimizes unnecessary ray-intersection calculations. The simplest tiling scheme is to decompose the image screen into fixed-size tiles, and we used this approach in this work (Figure 5).

3.4 Depth Peeling

One of the problems of the ray casting algorithm as described above is that it only works for convex meshes. For non-convex meshes, it requires computing the point that each ray re-enters the mesh. The solution to this problem proposed in [14] involves filling up the space created by non-convexities with empty cells, and re-entry is computed by following rays through empty cells (without accumating color and opacity values) until a mesh tetra is found. This requires a convexification algorithm that is not trivial, and often manual convexification is used.

A more general solution is to use a depth peeling approach (Figure 6). Since only entry and exit points are important, only the boundary faces of the tetrahedral mesh need to be considered. The algorithm produces *n* layers of visible faces (back-face culling is used to remove non-visible faces). The first layer corresponds to the same set of visible faces. Subsequent layers contain the next visible face, which can be computed using the information of the previous layer.

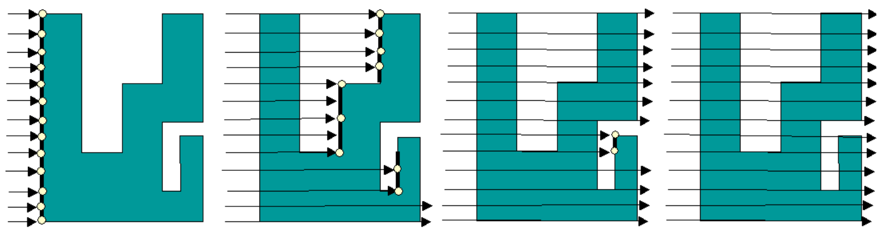


Figure 6: Depth Peeling Captures Non-Convexities

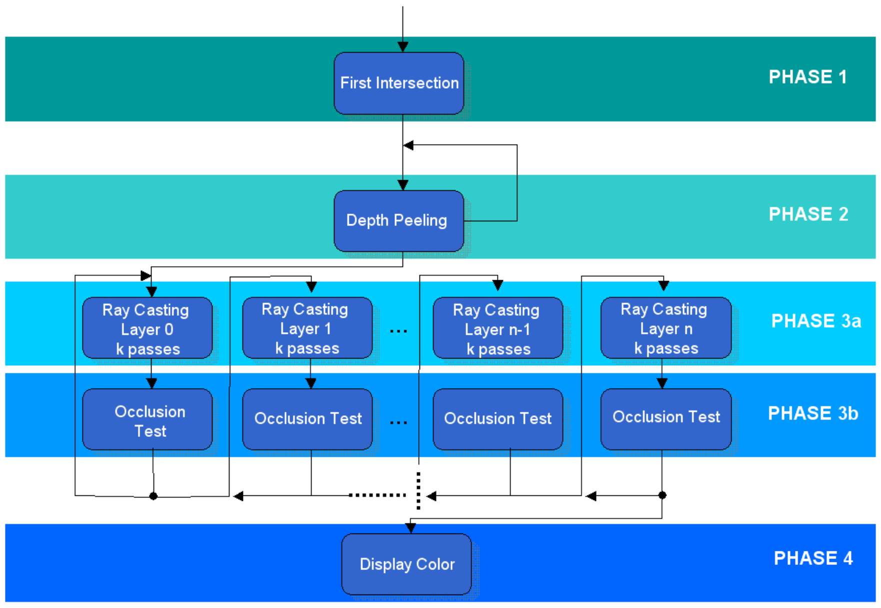


Figure 7: Tiled Ray Casting System View

The ray-casting algorithm uses the information stored in all layers to capture missing non-convexities. Starting from the first layer, computation proceeds as before until all rays leave the mesh. If the next layer contains visible faces, a second pass using as starting points the information stored in the second layer is issued. This process is repeated until all rays leave the mesh and the following layer does not contain visible faces.

4 GPU Implementation

The implementation of the tiled ray-casting algorithm in the GPU requires the execution of several fragment shaders, and a global view of the complete system is given in Figure 7. The system can be divided into four different phases: first intersection calculation, depth peeling, ray-casting/occlusion test and final display. Before describing each phase in detail, we review the storage of data structures into GPU memory.

4.1 Storage

Two different types of data structures need to be stored into the GPU memory: mesh and traversal data. The first type corresponds to the mesh information for all tetrahedron: vertices, normals, face adjacencies and scalar values . In addition, the ray-casting algorithm requires at each computation to recover the cell that the ray is currently located, as well as the intersection point and scalar field at this point. The proposal described in Weiler [14] requires 160*B* per tetrahedron. Mesh data is stored using 3 floating-point 3D-textures (vertices, normals and face adjacencies) and 1 floating-point 2D-texture (scalar data). Access to 3D textures uses an index that is composed of two components describing the tetrahedral index (separated because current GPUs limit index of 2D textures to smaller values) and a third index that specifies which vertex (or normal, or face adjacency) is to be retrieved.

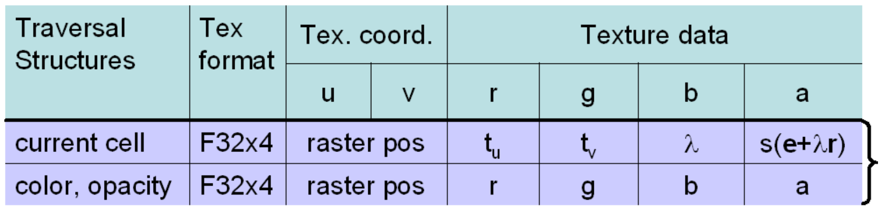
The traversal data is updated using a fragment shader that uses the Multiple Render Target (MRT) capabilities to write into 3 floating-point 2D textures with screen dimensions (intersection, current cell and color data). Color data is required because the GPU used (ATI 9700) does not allow writing to the frame-buffer while using MRTs.

One of the problems of their approach is that several texture components were unused. We reviewed their solution and improved in many points. First, we store mesh data into 2D textures instead of 3D textures. This is very important, since in this graphics generation 3D texture are slower than 2D textures. Since there is no true branching in the fragment shaders of current GPUs (ATI 9700 class hardware), an intersection calculation has to fetch all vertices and normals of a given tetra. By storing them consecutivelly into texture, there is no need to have a third index (a simple displacement in texture indices is used to get the next value). We also re-arranged data information and no texture components are left unused. As a result, our approach only requires 3 two-dimensional structures (Figure 8), and each tetra requires 144*B* (a 10% economy).

Mesh Data	Tex format	Tex. coord.		Texture data				
		u	v	r	g	b	a	
vertices	F32x4	t_u	t_v	$v_{t,0.x}$	$v_{t,0.y}$	$v_{t,0.z}$	$v_{t,3.x}$	
vertices	F32x4	t_u	t_{v+dv}	$v_{t,1.x}$	$v_{t,1.y}$	$v_{t,1.z}$	$v_{t,3.y}$	48B
vertices	F32x4	t_u	t_{v+2dv}	$v_{t,2.x}$	$v_{t,2.y}$	$v_{t,2.z}$	$v_{t,3.z}$	
face normals	F32x4	t_u	t_v	$n_{t,0.x}$	$n_{t,0.y}$	$n_{t,0.z}$	$n_{t,3.x}$	
face normals	F32x4	t_u	t_{v+dv}	$n_{t,1.x}$	$n_{t,1.y}$	$n_{t,1.z}$	$n_{t,3.y}$	48B
face normals	F32x4	t_u	t_{v+2dv}	$n_{t,2.x}$	$n_{t,2.y}$	$n_{t,2.z}$	$n_{t,3.z}$	
neighbor data	F32x4	t_u	t_v	$a_{t,0}$		$a_{t,1}$		32B
neighbor data	F32x4	t_u	t_{v+dv}	$a_{t,2}$		$a_{t,3}$		
scalar data	F32x4	t_u	t_{v+2dv}	$g_{t,x}$	$g_{t,y}$	$g_{t,z}$	s_t	16B
								144B

Figure 8: Mesh data stored into textures

The traversal data is also more compact since removing the storage of the third index leads to only 2 traversal structures: (1) current cell/intersection and (2) color (Figure 9).



Traversal Structures	Tex format	Tex. coord.		Texture data			
		u	v	r	g	b	a
current cell	F32x4	raster pos		t_u	t_v	λ	$s(\mathbf{e}+\lambda\mathbf{r})$
color, opacity	F32x4	raster pos		r	g	b	a

Figure 9: Traversal data stored into textures

4.2 Shaders

In this section we describe the different shaders used to implement the tiled ray casting algorithm. For complete details, we refer the reader to the source code of the shaders, which is attached as an appendix. The source code of the complete renderer is available from the authors.

Phase 1 - Computing First Intersection The first shader is responsible for initializing the traversal structures with the information of the first intersection of rays against mesh faces. This can be accomplished by rendering only the boundary faces, using special vertex and fragment shaders that properly update the traversal structures.

Each boundary face sent through the graphics pipeline is setup to contain at each vertex the tetra index where it is defined (coded in the two components t_u and t_v). Since we draw these faces with Z-buffer enabled, this information is only updated in the traversal structure for the first visible face along the ray associated with each fragment.

The update of the parametric value λ and scalar field values requires an additional intersection computation in the fragment shader. Unlike the intersection computed during rasterization that happens in WCS, the λ intersection computation happens in OCS (no geometric transformations are applied to mesh vertices). Both intersections results must match. This is accomplished by changin eye and ray direction in the vertex shader into OCS by applying the inverse of the modeling transformation. In the fragment shader, mesh data is retrieved from textures (vertices, normals, face adjacencies and scalar data). The ray direction (r_{ocs}) and eye (e_{ocs}) are used to compute the first intersection as described before. The scalar field is evaluated at this time and both values are stored into the traversal structure along the tetra indexes.

Phase 2 - Computing Depth Peeling Levels The second phase consists of computing additional depth peeling levels. Most of the computation is similar to what is performed in the first intersection shader, and for this reason, they use the same vertex shader.

The difference lies in the fragment shader, and how successive layers are generated. Since current GPUs do not have the ability to read and write into the same texture, a multi-pass approach that computes one layer at each pass is used. The fragment shader is essentially the same described in phase 1, with an additional test that discards the previous layer of visible faces. This is accomplished by using the information generated for the previous layer as input for the next level computation. For each candidate λ, its value is compared to the one stored in the previous layer. If the value is smaller or equal, the fragment is discarded. As a result, only the next visibile information will be stored.

It is important to notice that back-face culling must be turned ON in this process, and a bias value must be used in the comparison test to reduce artifacts due to numerical calculations.

Phase 3a - Ray-Casting The ray casting shader is responsible for advancing for each ray one intersection against the mesh, and this process is repeated through multiple passes until all rays leave the mesh. Since the result of each pass needs to be used in a subsequent pass, and that no read-write textures are available in current GPUs, a pair of traversal textures are used and switched at each pass (called ping-pong rendering).

For each pass, computation is forced to be performed for all fragments in the screen by rendering one screen-aligned rectangle. If multiple tiles are being used, multiple rectangles corresponding to each individual subregion are rendered.

The information generated in the first intersection shader is used as input to the first ray casting pass. The computation for each fragment in a given pass uses the current intersection position and tetra to compute the next intersection along the ray direction. Both entry and exit points (and their distance) are used to retrieve pre-computed color information and are accumulate into the color value as described in Equation 3. Like the first intersection and depth peeling shaders, this computation is performed in the OCS. However, the intersection test looks for the minimum λ (instead of the maximum). In addition, the adjacent tetra at the intersection point must be computed and stored in the traversal structures. If there is none, a special index is stored in the traversal structure, indicating that the ray has left the mesh.

This process is executed until all rays leave the mesh, which is detected in a separate occlusion pass described below. Once the first layer of rays leave the mesh, the next depth peeling layer is recovered and computation restarts in the first pass, but using this layer information as starting point (only that the color buffer is not reset). Computation ends when the next depth peeling layer is empty (no visible faces).

Phase 3b - Z-update and Occlusion Test Ray casting calculation must end when all rays leave the mesh, but this is not directly returned by the intersection calculation. This can be accomplished by two special fragment shaders. The first shader (Z-update), renders a screen-(or tile-)aligned rectangle, and writes into the depth buffer with Z_{near} for all fragments that have left the mesh (the traversal structure contains an invalid tetra index). In addition, writing values to the depth-buffer allows early z-test to reject fragment processing for rays that already left the mesh.

The second fragment shader, occlusion test, issues an occlusion query before rendering a screen-(or tile)-aligned rectangle. The result of the occlusion query will represent how many fragments are still active (rays are inside the mesh). Since this additional pass incurs in additional costs, this can be amortized if run after k ray casting passes.

Phase 4 - Display Resulting Color The final shader is responsible for displaying the color computed during the ray casting calculation into the screen. It renders a screen-aligned rectangle that samples the color texture generated by the ray casting pass.

5 Results

Experiments were performed in a PC with a Pentium IV 2.0GHz, 640MB memory, a Radeon 9700 Pro 128MB graphics card (driver Catalyst 4.4). Code was implemented using C++, DirectX 9.0b, and Microsoft High Level Shading Language (HLSL). This choice was influenced by the following aspects: only current ATI boards support Multiple Render Targets (required by the ray casting shader), and only DirectX in ATI boards with the current driver allow simultaneous writes to floating-point textures and the depth-buffer (required in the depth-peeling shaders).

Three different non-convex datasets, with sizes ranging from 103K to 240K tetra were used. For each dataset, we tested different camera positions, and different tile counts (from 1 to 36 tiles). Minimum and maximum times were recorded for each dataset and tile count, and they are reported in tables 1-4.

Image quality is very good, see figure 10. We have, however, experienced a small number of artifacts, as can be observed in some of the images. This might be caused by the reduced mantissa in the ATI 9700 floating point representation. Bias factors used in depth peeling might also be the source of numerical problems.

Mesh	Tets	Min FPS	Max FPS	Min Tets/s	Max Tets/s
spx1	103K	1.18	1.68	121K	173K
blunt	187K	2.13	7.09	398K	1.32M
f117	240K	1.36	2.46	326K	590K

Table 1: Summary of Datasets Statistics

Mesh	Min Pass 2D	Max Pass 2D	Min Pass 3D	Max Pass 3D
spx1	571	601	561	611
blunt	341	439	392	542
f117	571	601	561	611

Table 2: Ray Casting Passes - 2D-3D Pre-Int.

Mesh	1x1	2x2	3x3	4x4	5x5	6x6
spx1	625-812	625-797	625-797	609-766	609-750	594-844
blunt	203-422	187-391	172-407	188-406	141-391	141-469
f117	453-734	422-672	406-641	406-656	359-640	406-625

Table 3: Running Time in ms - 2D Pre-Integration

6 Conclusions

Implementing algorithms on a GPU is an involved process. While working on this project, we were surprised with how hard it was to reproduce the results in [14]. One central difficulty we faced was to implement the ray casting shader in a single pass (which limits our shaders to up to 64 ALU and 32 texture instructions). Part of the problem was the lack of comprehensive documentation for the GPU features, driver releases and APIs, not to mention inadequate debugging tools. Our first success was a limited version of Weiler's algorithm working without the pre-integration calculation, but it was slow, possibly due to the large number of 3D texture look-ups. We were unable to incorporate the pre-integration without violating the instruction limit.

This lead us to consider a complete redesign of the mesh representation, leading to a more compact data structure based on 2D textures. This worked quite well. We are able to have the full ray casting code in a single pass using 60 instructions (The HLSL code is included as an appendix). A video showing the system in action is also included. The use of tiles is crucial to speed up the rendering, because datasets tend to have uneven screen coverage. Finally, depth peeling turned out to be an elegant and general solution, handling non-convex meshes, and requiring no manual intervention.

Mesh	1x1	2x2	3x3	4x4	5x5	6x6
spx1	750-921	750-907	735-906	734-906	688-907	718-969
blunt	390-547	375-547	375-532	375-515	344-500	344-562
f117	500-812	484-766	484-734	469-750	453-719	500-687

Table 4: Running Time in ms - 3D Pre-Integration

Acknowledgments

The authors thank Manfred Weiler for help with the multiple render target functionality in DirectX. And Steven Callahan and Milan Ikits for help with the implementation of the 3D pre-integrated volume rendering. We thank Carlos Scheidegger for useful criticisms on this work that helped to improve its content and presentation. The authors also acknowledge Bruno Notrosso (Electricite de France) for the spx dataset, Hung and Buning for the blunt fin dataset. Cláudio T. Silva is partially supported by the DOE under the VIEWS program and the MICS office, and the National Science Foundation under grants CCF-0401498 and EIA-0323604. The work of João L. D. Comba is supported by a CNPq grant 540414/01-8 and FAPERGS grant 01/0547.3.

References

[1] P. Bunyk, A. Kaufman, and C. Silva. Simple, Fast, and Robust Ray Casting of Irregular Grids. In *Proceedings of Dagstuhl '97*, pages 30–36, 2000.

[2] J. L. Comba, J. S. Mitchell, and C. T. Silva. On the Convexification of Unstructured Grids From A Scientific Visualization Perspective. Technical report, University of Utah, 2003.

[3] C. Everitt. Interactive Order-Independent Transparency. White paper, NVIDIA Corporation, 1999.

[4] R. Farias, J. S. B. Mitchell, and C. T. Silva. ZSWEEP: an efficient and exact projection algorithm for unstructured volume rendering. In *Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 91–99. ACM Press, 2000.

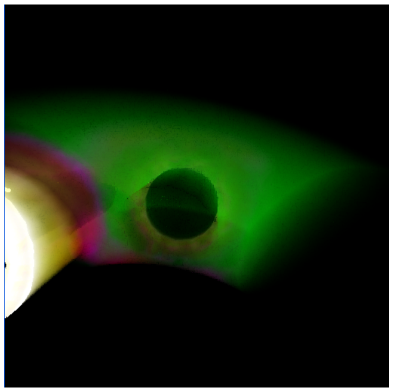
[5] R. Farias and C. T. Silva. Parallelizing the ZSWEEP Algorithm for Distributed-Shared Memory Architectures. In K. Mueller and A. Kaufmann, editors, *Proceedings of the Joint IEEE TCVG and Eurographics Workshop (VolumeGraphics-01)*, pages 181–194, Wien, June 21–22 2001. Springer-Verlag.

[6] M. P. Garrity. Raytracing Irregular Volume Data. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24(5):35–40, Nov. 1990.

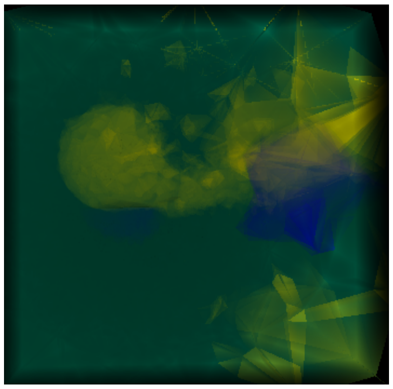
[7] S. Krishnan, C. Silva, and B. Wei. A Hardware-Assisted Visibility-Ordering Algorithm With Applications to Volume Rendering. In *Data Visualization 2001*, pages 233–242, 2001.

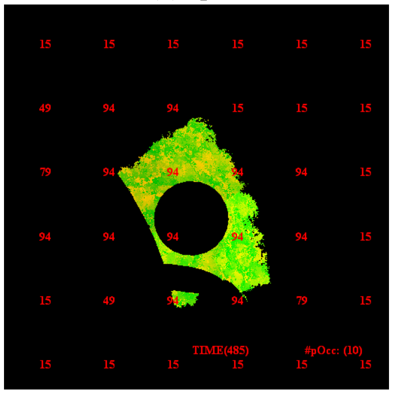
[8] J. Krueger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[9] A. Mammen. Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. *IEEE Comput. Graph. Appl.*, 9(4):43–55, 1989.

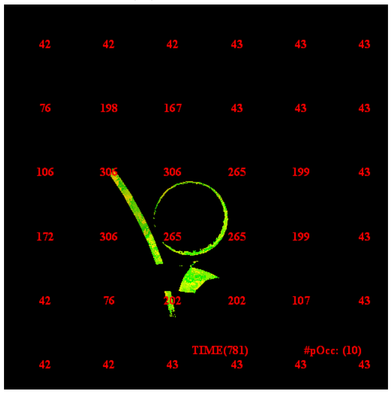

(a) spx1

(b) blunt fin

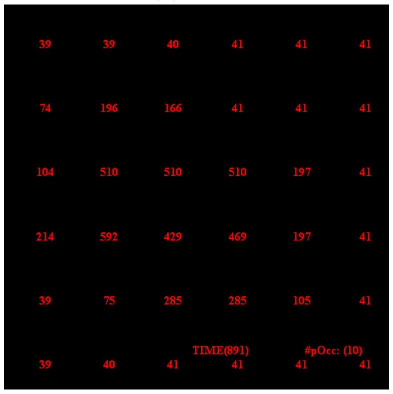

(c) f117



(d)



(e)



(f)

Figure 10: Volume Rendering Results (a)-(c) Tiled Ray Casting in action (d)-(f). We show the number of passes necessary for each tile.

[10] Z. Nagy and R. Klein. Depth-Peeling for Texture-Based Volume Rendering. In *Pacific Conference on Computer Graphics and Applications*, 2003.

[11] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[12] S. Roettger, S. Guthe, D. Weiskopf, and T. Ertl. Smart Hardware-Accelerated Volume Rendering. In *Procceedings of EG/IEEE TCVG Symposium on Visualization VisSym '03*, pages 231–238, 2003.

[13] P. Schneider and D. Eberly. *Geometric Tools for Computer Graphics*. Morgan Kaufmann, 2003.

[14] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of IEEE Visualization 2003*, pages 333–340, 2003.

[15] M. Weiler, P. N. Mallon, M. Kraus, and T. Ertl. Texture-Encoded Tetrahedral Strips. In *Proceedings IEEE Volume Visualization and Graphics Symposium 2004*, to appear.

[16] P. L. Williams. Visibility-Ordering Meshed Polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, Apr. 1992.

Appendix

Fragment Programs for TRC

```
//---------------------------------------------------------------------------
// Fragment programs for our manuscript:
// "A GPU-Based Tiled Ray Casting using Depth Peeling"
// (C) 2004 Fabio Bernardon, Christian Pagot, Joao Comba, Claudio Silva
// ---------------------------------------------------------------------------
//---------------------------------------------------------------------------
// First Intersection and Depth Peeling Vertex shader
//---------------------------------------------------------------------------
FirstPass_VS_OUTPUT FirstHit_VS(FirstPass_VS_INPUT IN)
{
FirstPass_VS_OUTPUT OUT = (FirstPass_VS_OUTPUT) 0;

float4 pos = mul( matModel, IN.position );
float4 ray = mul (matModelInverse, pos - eye);
pos = mul (matView, pos);
pos = mul( matProjection, pos );

OUT.position  = pos;
OUT.texcoord0 = IN.texcoord0;
OUT.texcoord1 = IN.position;
OUT.texcoord2 = OUT.position;
OUT.ray = ray;
return OUT;
}

//---------------------------------------------------------------------------
// Ray Casting Vertex shader
//---------------------------------------------------------------------------
RayCasting_VS_OUTPUT RayCasting_VS(RayCasting_VS_INPUT IN)
{
RayCasting_VS_OUTPUT OUT = (RayCasting_VS_OUTPUT) 0;

// transform position and eye to camera space.
float4 pos = mul( matModelInverse, IN.position );
float4 ray = pos - mul( matModelInverse, eyeLocal);

OUT.position  = IN.position;
OUT.position.z = 0.8 * OUT.position.w;
OUT.texCoord0 = IN.texCoord0;
OUT.texCoord1 = ray;
OUT.tEye      = eyeLocal;
OUT.diffuse   = IN.diffuse;
return OUT;
}

//---------------------------------------------------------------------------
// First Intersection Fragment Shader
//---------------------------------------------------------------------------
FirstPass_PS_OUTPUT FirstHit_PS(FirstPass_PS_INPUT IN)
{
FirstPass_PS_OUTPUT OUT = (FirstPass_PS_OUTPUT) 0;

float4 result = float4(IN.texcoord0.rg, 0.0, 1.0);

// Retrieve data from mesh textures
float4 vertex0, vertex1, vertex2, vertex3;
float4 normal0, normal1, normal2, normal3;
vertex0 = tex2D(newVertexSampler, result.xy);
normal0 = tex2D(newNormalsSampler, result.xy);
result.x += passColor.x;
vertex1 = tex2D(newVertexSampler, result.xy);
normal1 = tex2D(newNormalsSampler, result.xy);
result.x += passColor.x;
vertex2 = tex2D(newVertexSampler, result.xy);
normal2 = tex2D(newNormalsSampler, result.xy);
vertex3 = float4(vertex0.w, vertex1.w, vertex2.w, 0.0) - eyeRC;
normal3 = float4(normal0.w, normal1.w, normal2.w, 0.0);
vertex0 -= eyeRC;
vertex1 -= eyeRC;
vertex2 -= eyeRC;

// Normalize the interpolated ray direction
float3 ray = normalize(IN.ray.xyz);

// Compute intersections of the faces of the tetrahedron against the
// ray, discarding the one that the ray enters the tetrahedron
float4 num = float4(dot(vertex3.xyz, normal0.xyz), dot(vertex2.xyz, normal1.xyz),
    dot(vertex1.xyz, normal2.xyz), dot(vertex0.xyz, normal3.xyz));
float4 den = float4(dot(ray.xyz, normal0.xyz), dot(ray.xyz, normal1.xyz),
    dot(ray.xyz, normal2.xyz), dot(ray.xyz, normal3.xyz));

float4 lambda = num / den;
lambda = (den < 0 && lambda > 0) ? lambda : 0;
result.z = (lambda.x > lambda.y) ? lambda.x : lambda.y;
result.z = (lambda.z > result.z) ? lambda.z : result.z;
result.z = (lambda.w > result.z) ? lambda.w : result.z;

// scalar value computation
float4 grad = tex2D(newNeighborSampler, result.xy);
vertex2 = tex2D(newVertexSampler, result.xy);
float3 x = result.z * ray + eyeRC;
float gtx = dot(grad.xyz, x);
result.w = gtx + grad.w;
result.xy = IN.texcoord0.xy;

OUT.currentCell = result;

return OUT;
}

//---------------------------------------------------------------------------
// Depth peeling pixel shader
//---------------------------------------------------------------------------
DepthPeeling_PS_OUTPUT DepthPeeling_PS(FirstPass_PS_INPUT IN)
{
DepthPeeling_PS_OUTPUT OUT = (DepthPeeling_PS_OUTPUT) 0;

float2 index = (IN.texcoord2.xy/IN.texcoord2.w+1)/2 + biasWindow.xy;
index.y = 1.0 - index.y;
float4 currentValue = tex2D(currentCellSampler, index);

float4 result = float4(IN.texcoord0.rg, 0.0, 1.0);

// Retrieve data from mesh textures
float4 vertex0, vertex1, vertex2, vertex3;
float4 normal0, normal1, normal2, normal3;
vertex0 = tex2D(newVertexSampler, result.xy);
normal0 = tex2D(newNormalsSampler, result.xy);
result.x += passColor.x;
vertex1 = tex2D(newVertexSampler, result.xy);
normal1 = tex2D(newNormalsSampler, result.xy);
result.x += passColor.x;
vertex2 = tex2D(newVertexSampler, result.xy);
normal2 = tex2D(newNormalsSampler, result.xy);
vertex3 = float4(vertex0.w, vertex1.w, vertex2.w, 0.0) - eyeRC;
normal3 = float4(normal0.w, normal1.w, normal2.w, 0.0);
vertex0 -= eyeRC;
vertex1 -= eyeRC;
vertex2 -= eyeRC;

// Normalize the interpolated ray direction
float3 ray = normalize(IN.ray.xyz);

// Compute intersections of the faces of the tetrahedron against the
// ray, discarding the one that the ray enters the tetrahedron
float4 num = float4(dot(vertex3.xyz, normal0.xyz), dot(vertex2.xyz, normal1.xyz),
dot(vertex1.xyz, normal2.xyz), dot(vertex0.xyz, normal3.xyz));
float4 den = float4(dot(ray.xyz, normal0.xyz), dot(ray.xyz, normal1.xyz),
dot(ray.xyz, normal2.xyz), dot(ray.xyz, normal3.xyz));

float4 lambda = num / den;
lambda = (den < 0 && lambda > 0) ? lambda : 0;

result.z = (lambda.x > lambda.y) ? lambda.x : lambda.y;
result.z = (lambda.z > result.z) ? lambda.z : result.z;
result.z = (lambda.w > result.z) ? lambda.w : result.z;

// scalar value computation
float4 grad = tex2D(newNeighborSampler, result.xy);
vertex2 = tex2D(newVertexSampler, result.xy);
float3 x = result.z * ray + eyeRC;
float gtx = dot(grad.xyz, x);
result.w = gtx + grad.w;

result.xy = IN.texcoord0.xy;

float thisZ = IN.texcoord2.z / IN.texcoord2.w;

if (result.z > (currentValue.z + 0.1)){
OUT.currentCell = result;
OUT.depth = thisZ;
}else{
// this value must be written behind all the others
// set depth value to large value and scalar to -1.0
OUT.currentCell = float4(0.0, 0.0, 999999.0, 999999.0 );
OUT.depth = 1.0f;
}

return OUT;
}
```

```
//----------------------------------------------------------------------------
// Ray Casting Fragment Shader
//----------------------------------------------------------------------------
RayCasting_PS_OUTPUT RayCasting_PS(RayCasting_PS_INPUT IN) {
RayCasting_PS_OUTPUT OUT = (RayCasting_PS_OUTPUT) 0;

float4 currentValue = tex2D(currentCellSampler, IN.rasterPos.xy);
float4 result = float4(currentValue.xy, 0.0, 1.0);

// Retrieve data from mesh textures
float4 vertex0, vertex1, vertex2, vertex3;
float4 normal0, normal1, normal2, normal3;
float4 result1, result2, result3, result4;
vertex0 = tex2D(newVertexSampler, result.xy);
normal0 = tex2D(newNormalsSampler, result.xy);
result1 = tex2D(newNeighborSampler, result.xy);
result.x += passColor.x;
vertex1 = tex2D(newVertexSampler, result.xy);
normal1 = tex2D(newNormalsSampler, result.xy);
result3 = tex2D(newNeighborSampler, result.xy);
result.x += passColor.x;
vertex2 = tex2D(newVertexSampler, result.xy);
normal2 = tex2D(newNormalsSampler, result.xy);
vertex3 = float4(vertex0.w, vertex1.w, vertex2.w, 0.0) - eyeRC;
normal3 = float4(normal0.w, normal1.w, normal2.w, 0.0);
vertex0 -= eyeRC;
vertex1 -= eyeRC;
vertex2 -= eyeRC;

// Normalize the interpolated ray direction
float3 ray = normalize(IN.ray.xyz);

// Compute ray-tetrahedron faces intersections
float4 num = float4(dot(vertex3.xyz, normal0.xyz), dot(vertex2.xyz, normal1.xyz),
dot(vertex1.xyz, normal2.xyz), dot(vertex0.xyz, normal3.xyz));
float4 den = float4(dot(ray.xyz, normal0.xyz), dot(ray.xyz, normal1.xyz),
dot(ray.xyz, normal2.xyz), dot(ray.xyz, normal3.xyz));

float4 lambda = num / den;
lambda = (den > 0 && lambda > 0) ? lambda : 999999;

result2.xy = result1.zw;
result4.xy = result3.zw;

result1.zw = float2(lambda.x, 0.0);
result2.zw = float2(lambda.y, 0.0);
result3.zw = float2(lambda.z, 0.0);
result4.zw = float2(lambda.w, 0.0);

// scalar value computation
float4 grad = tex2D(newNeighborSampler, result.xy);
vertex2 = tex2D(newVertexSampler, result.xy);

result = (lambda.x < lambda.y) ? result1 : result2;
result = (result.z < lambda.z) ? result : result3;
result = (result.z < lambda.w) ? result : result4;

float3 x = result.z * ray + eyeRC;
float gtx = dot(grad.xyz, x);
result.w = gtx + grad.w;

// color integration
float3 lutIndex = float3(currentValue.w, result.w,
 1.0-(result.z - currentValue.z)/eyeRC.w);
float4 contribution = tex3D(lutSampler, lutIndex);

float4 color = tex2D(colorSampler, IN.rasterPos.xy);
contribution = (currentValue.x == 0.0) ? float4(0.0, 0.0, 0.0, 0.0) : contribution;
color += contribution * (1.0 - color.w);

// if the color contribution exceeds 0.95, eliminate the tetrahedra
if(currentValue.x > 0.0 && color.w < 0.95) OUT.currentCell = result;
else OUT.currentCell = float4(0.0, 0.0, 999999.0, 0.0);

OUT.color = color;

return OUT;
}
```