GPU-based Tiled Ray Casting using Depth Peeling

Fábio F. Bernardon[2] Christian A. Pagot[2] João L. D. Comba[2] Cláudio T. Silva[1]

[1] Scientific Computing and Imaging Institute, School of Computing, University of Utah
[2] Instituto de Informática, Universidade Federal do Rio Grande do Sul

Abstract

We propose a new hardware ray casting algorithm for unstructured meshes composed of tetrahedral cells. Our work builds on the technique published at IEEE Visualization 2003 by Weiler *et al.*. 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. 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. Our experiments show that our technique is substantially faster than the technique of Weiler *et al.* on the same hardware.

1 Introduction

In this paper, we propose a new hardware-based ray casting algorithm for volume rendering of unstructured meshes. Unstructured grids are extensively used in computational science and engineering 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].

Garrity [6] pioneered the use of ray casting for rendering unstructured meshes. His 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. Because meshes are not necessarily convex, any given ray may get in and out of the mesh multiple times. To avoid expensive search operations during re-entry, Garrity used a hierarchical spatial data structure to store boundary cells.

Bunyk *et al.* [1] proposed a different technique for handling non-convexities in ray casting. Instead of building a hierarchical spatial data structure of the boundary cells, they exploit the discrete nature of the image. And simply determine for each pixel in the screen, the fragments of the front boundary faces that project there (and are stored in front-to-back order). During ray traversal, each time a ray exits the mesh, Bunyk *et al.* use the boundary fragment intersection to determine whether the ray will re-enter the mesh, and where (i.e., the particular cell). This approach turns out to be simpler, faster, and more robust to floating-point calculations.

Weiler *et al.* [13] propose a hardware-based rendering algorithm based on the work of Garrity. They store the mesh and traversal data structures on the GPU using 3D and 2D textures respectively. On their technique, there is no need for information to be send back and forth between the CPU and GPU, and all computations are performed directly on the GPU. GPU performance currently outpaces the CPU, thus

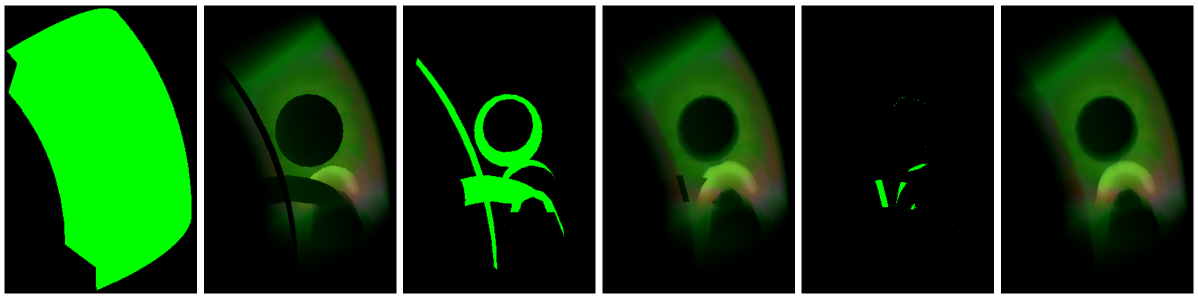

Figure 1: Volume rendering using depth peeling. Three pairs of images are shown. On the right of each pair, we show the pixels that have active intersections to be performed on the current pass. On the left, we show a partial image after the rays corresponding to those intersections have been terminated (i.e., they have left the current component).

naturally leading to a technique that scales well over time as GPUs get faster. Strictly speaking, their algorithm only handles convex meshes. (i.e., it is not able to find the re-entry of rays that leave the mesh temporarily), and they use a technique originally proposed by Williams [15] 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 unstructured meshes has many unresolved issues that make it a hard (and currently unsolved) problem. We point the reader to Comba *et al.* [2] for details. For their experimental results, Weiler *et al. manually* convexified the meshes.

In this paper, we propose a novel hardware-based ray casting algorithm for volume rendering unstructured meshes. Our work builds on the previous work of Weiler *et al.* [13], but improves it in several significant ways. Our rendering algorithm is based on the work of Bunyk *et al.* [1], which has a better match for a hardware implementation, since, as shown later, 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 substantially improved rendering rates.

2 Tiled Ray Casting Algorithm

Our algorithm is based on the hardware ray casting algorithm of Weiler *et al.* [13]. The tetrahedral mesh data and pre-integrated tables are packed inside textures to avoid the bottleneck caused by data transfer between GPU and CPU. Ray casting is a trivially parallelizable algorithm that fits the SIMD programming model supported by GPUs quite effectively. Ray traversal is done by several ping-pong rendering steps, where the GPU advances over the mesh, one cell at a time, computing the volume rendering integral over all cells that are inside the viewing frustum. The algorithm takes advantage of the early-z test to avoid the cost of computations related to rays that miss the mesh, or that have been traversed already. (We note that 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. Roettger *et al.* [11] presents another approach for ray casting regular grids.)

The mesh data (vertices, normals, faces, tetrahedra) is kept inside textures, which leads to the maximum

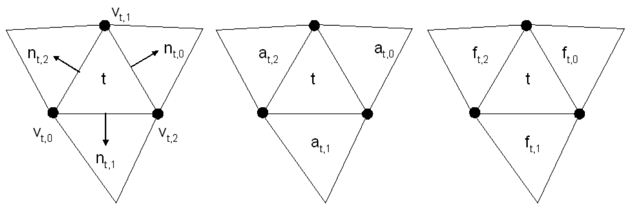

Figure 2: Mesh notation showing the relationship between vertices, faces, normals, and tetrahedra.

size of a mesh to be constrained by the amount of GPU memory available. This also means that special care must be taken when choosing how to layout the data inside textures. We present 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.

In our algorithm we use image tiling, which has been shown to improve performance by taking advantage of memory (cache) coherence. In particular, Farias and Silva proposed an image-based task partitioning scheme for their ZSWEEP algorithm for rendering unstructured meshes [4, 5] that works well in distributed shared-memory machines. In their approach, 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] in their hardware-based visibility ordering algorithm.

The final component of our algorithm is depth peeling, which is used to compute for each pixel in the screen, the fragments of the front boundary faces that project there. We note that Nagy and Klein [10] also developed an approach for the visualization of the *n*th iso-layer of a volumetric data set based on depth peeling. Their technique only requires one pass for the visualization of any number of layers, and can render interior and exterior iso-surfaces, but it is not capable to integrate through the volumetric data set.

Before describing our complete algorithm in detail we first discuss how to perform the individual steps necessary for the implementation of the technique.

2.1 Notation

The mesh is referenced using the following notation. A mesh M is composed of n_t tetrahedral cells (tetra) and n_v vertices. Each tetrahedron can be referenced by an index t_i. For each tetrahedron, f_i represents the $i^{th}(i = 0..3)$ face of a tetrahedron, $n_{t,i}$ represents the normal of the i^{th} face (always pointing outside), the vertex opposed to the i^{th} face is referred to as $v_{t,i}$. Each cell is adjacent to up to four cells by one of its faces. We call $a_{t,i}$ the cell that is adjacent to cell t by its i^{th} face. Figure 2 illustrates the mesh notation.

The parametric equation of a line is 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 are used for intersection calculations. The *object coordinate system* (OCS) represents the coordinate system where the mesh is defined, and the *world coordinate system* (WCS) the coordinate system 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.

2.2 Computing Ray-Mesh Intersections

The ray casting algorithm is designed to exploit 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 tetrahedron, and advances one intersection at a time. The process ends when all rays leave the mesh.

A ray-tetrahedron intersection can be done by computing four intersections of a given ray against the supporting planes of tetrahedron faces [12]. Each parametric intersection λ_i can be computed using the following equation:

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

Only positive λ_i values need to be considered. The entry point λ_{in} corresponds to the maximum lambda among values with negative denominator (using our normal convention). Similarly, the exit point $\lambda_o ut$ corresponds to the minimum lambda among values with positive denominator. If $\lambda_{in} > \lambda_{out}$ then no intersection exists.

Since the algorithm proceeds incrementally, computing the exit point of a tetrahedron also requires computing the intersection against all faces, including the entering face. Distinguishing between them is done using the denominator rule described above. Weiler [13] 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 field 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 tetrahedron t is computed as:

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

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 indices into a pre-integrated 3D lookup table (LUT) 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:

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

2.3 Tile Partitioning

Spatial coherence is used in many different ways to accelerate ray casting queries. Several calculations performed by the algorithm proposed by Weiler [13] 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, subdividing 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

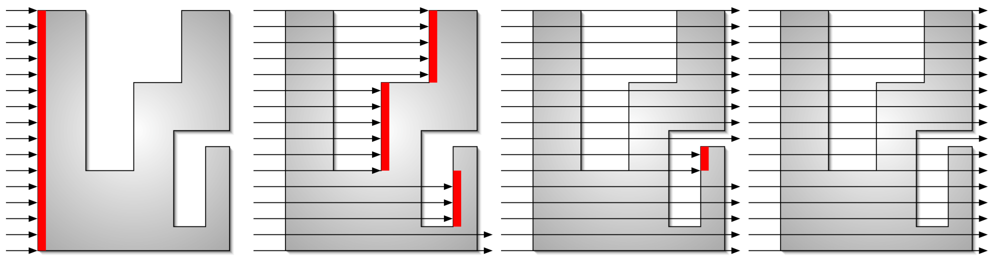

Figure 3: Depth peeling is used to compute for each pixel in the screen, the fragments of the front boundary faces that project there. This allows for a simple and effective way to handle non-convexities without the need for convexification.

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 our algorithm.

2.4 Depth Peeling

One of the problems of the basic ray casting algorithm that simply traverses cells one by one is that it only works for convex meshes. For non-convex meshes, it requires computing the point that each ray re-enters the mesh.

A general solution to this problem is to use a depth peeling approach, as shown in Figure 3. This is equivalent to the idea used in Bunyk *et al.* [1]. 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 initial set of visible faces. Subsequent layers contain the next visible faces discarding faces in previous layers.

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.

3 GPU Implementation

The implementation of the tiled ray-casting algorithm described in the previous section is not trivial, and it requires several fragment shaders. An overview is given in Figure 4. The implementation 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.

3.1 Details of Data Structure Storage

Two different types of data structures need to be stored into the GPU memory: mesh and traversal data structures. The first type corresponds to the mesh information for all tetrahedron: vertices, normals, face

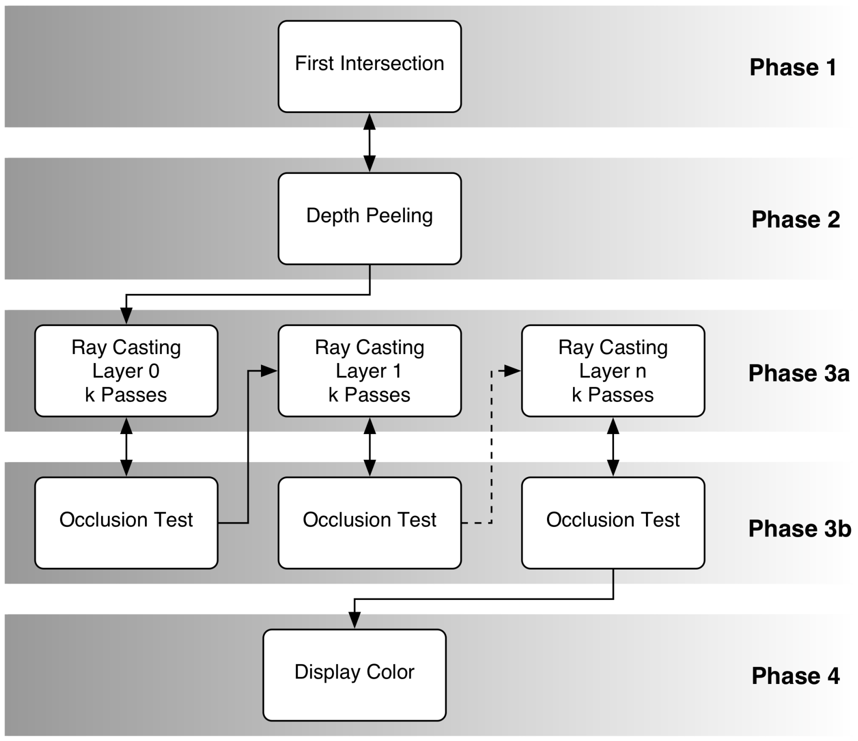

Figure 4: Overview of the implementation of our tiled ray-casting algorithm. The complete source code for the system can be obtained from the publisher's or the authors' website. Our implementation uses DirectX 9 and only works under Microsoft Windows.

adjacencies and scalar values. The second type corresponds to the information needed to recover the cell that the ray is currently located, as well as the intersection point and scalar field at this point. This information is needed for each pixel.

The data structures described in Weiler [13] requires 160 bytes 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 and a third index that specifies which vertex (or normal, or face adjacency) is to be retrieved. We note that the need to have the tetrahedral index separated into two indices comes from the fact that current GPUs limit the index of 2D textures to small values (currently up to 4096 per dimension).

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). We need to have a buffer that accumulates the color information that would normally go directly into the frame-buffer because current GPUs do not allow writing to the frame-buffer while using MRTs.

One of the issues with Weiler [13] is that their approach leaves several texture components unused, wasting valuable GPU memory. We propose a different way to store the information. First, we store mesh data into 2D textures instead of 3D textures. This is very important, since in most current graphics processors 3D textures are slower than 2D textures. Also, because most GPUs either have no true branching or branching is quite expensive, an intersection calculation has to fetch all vertices and normals of a given tetrahedron. By storing them consecutively in texture memory, there is no need to have a third index (a

Mesh data	Texture format	Texture coordinates		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}	$v_{t,1.x}$	$v_{t,1.y}$	$v_{t,1.z}$	$v_{t,3.y}$
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}	$n_{t,1.x}$	$n_{t,1.y}$	$n_{t,1.z}$	$n_{t,3.y}$
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	$a_{t,0}$		$a_{t,1}$	
scalar data	F32x4	t_u	t_{v+2dv}	$g_{t,x}$	$g_{t,y}$	$g_{t,z}$	s_t

Table 1: Data structures that are stored in GPU memory to maintain mesh information. Overall, we use 144 bytes per tetrahedron and only 2D textures.

.

simple displacement in texture indices is used to get the next value). As a result, our approach only requires 3 two-dimensional structures, and each tetrahedron requires 144 bytes leading to a 10% economy in space. Detailed information is shown in Table 1.

Traversal Structures	Texture format	Texture coordinates		Texture data			
		u	v	r	g	b	a
current cell	F32x4	raster position		t_u	t_v	λ	$s(e+\lambda r)$
color, opacity	F32x4	raster position		r	g	b	a

Table 2: Data structures used for ray traversal. As the mesh information, this is also stored directly in GPU memory as 2D textures.

The traversal data is also more compact than that of Weiler *et al.* since removing the storage of the third index leads to the need for only 2 traversal structures: (1) the current cell/intersection and (2) color. Details are shown in Table 2).

3.2 Details of the Vertex and Fragment Shaders

In this section we describe the different vertex and fragment shaders used to implement our tiled ray-casting algorithm. We refer the reader to the available complete source code for complete details of our implementation. Figure 4 provides an overview of the complete process.

Phase 1 - Computing First Intersection The first shader is responsible for initializing the traversal data 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 tetrahedron 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 intersection results must match. This is accomplished by changing 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 texture (vertices, normals, face adjacencies and scalar data). The directions of the ray, 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 data structure with the tetrahedron indices.

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, we 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 visible 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 no read-write textures are available in current GPUs, a pair of traversal textures are used and switched at each pass. This is often 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 tetrahedron 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 accumulated 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 tetrahedron 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 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 is accomplished by first rendering a screen-(or tile)-aligned rectangle that writes z_{near} into the depth buffer for every fragment that left the mesh

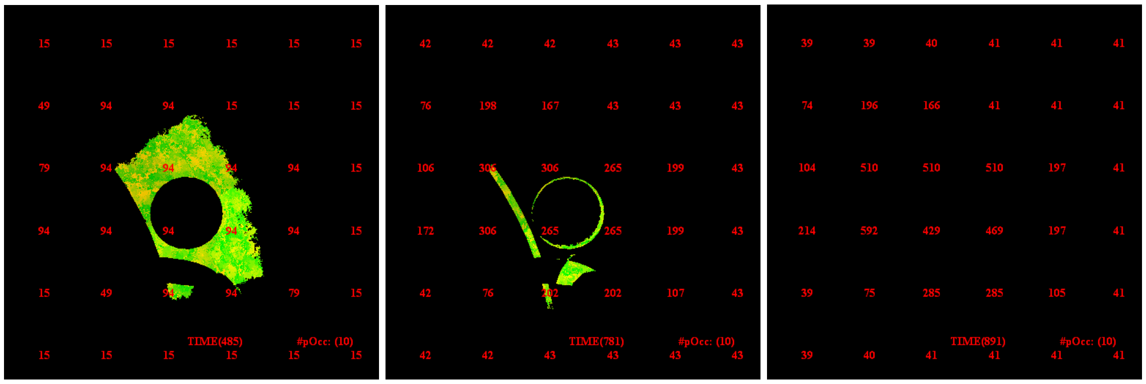

Figure 5: We show our Tiled Ray Casting in action. In particular, each image shows the number of passes necessary for each tile.

(the traversal structure contains an invalid tetrahedron index). Once the depth values were updated for all rays, a second rectangle is rendered with occlusion query turned on. The result of the occlusion query will represent how many fragments are still active (i.e., rays that are still inside the mesh). This approach has the additional advantage that writing values to the depth-buffer allows the early z-test to reject fragment processing for rays that already left the mesh. However, the additional z-update and occlusion test incur in additional costs, and this needs to be be amortized by only running this step 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.

4 Results

We report on experiment results performed in a machine equipped with an ATI Radeon 9700 Pro 128MB graphics card and a NVIDIA GeForce 6800 GT with 256 MB. Our code is implemented using C++, DirectX 9.0, and Microsoft High Level Shading Language (HLSL).

We have tried our system with a large number of non-convex tetrahedral datasets. We report numbers using 512×512 viewports on three non-convex datasets composed of tetrahedra: spx1 (103K), blunt (187K) and F117 (240K). For each dataset, we report the performance on different camera positions, and different tile counts (from 1 to 36 tiles). On Figure 5, we show our system in action, in particular, we plot the number of passes required to finish rendering the SPX dataset with 36 tiles. We implemented both 2D and 3D pre-integration approaches to accumulate the final color contribution. Since the artifacts from using a 2D pre-integration were minimal if compared to the 3D approach, all results reported here used the 2D pre-integration (current graphics hardware has faster 2D texturing capabilities than 3D).

In Tables 3 to 6 we report minimum and maximum rendering times for were recorded for each dataset under all the different tile configurations. It is interesting to note that tiling shows better results on the ATI card, while on the NVIDIA there is no clear advantage. In addition, increasing the number of tiles improves performance until a ”sweet” spot is found (an optimal tiling subdvision), after this performance decreases. For instance, the results for spx1 on the ATI 9700 in table 3 show a sweet spot at the tiling resolution 5×5, where we obtain the best minimum and maximum values.

Image quality is very good. We have, however, experienced a small number of artifacts on the ATI 9700 (regardless if 2D or 3D pre-integration was used). The exact cause of the artifacts is hard to determine. But we hypothesize that this is probably caused by the 24-bit floating point representation. We note that the artifacts on the NVIDIA 6800 are much less severe. Since this card has 32-bit precision, this seems to substantiate the claim. Another potential cause of the problem might be the bias factor that is used in the depth peeling though.

5 Discussion

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 Weiler *et al.* [13]. 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 their 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 (See source code.) 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.

Web Information

Complete source code containing all C++ files, HLSL shaders and a sample dataset is available at `http://www.acm.org/jgt/papers/BernardonEtAl05/`.

Acknowledgments

We would like to note that concurrently with our work, Weiler and colleagues developed a similar depth-peeling approach for handling non convex meshes. Their work is reported in [14].

The authors thank Manfred Weiler for help with the multiple render target functionality in DirectX. 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, and Steve Callahan for help with figures. The authors also acknowledge Bruno Notrosso (Electricite de France) for the spx dataset, Hung and Buning for the blunt fin dataset. We also thank NVIDIA for the donation of the graphics hardware. Cláudio T. Silva is partially supported by the Department of Energy under the VIEWS program and the MICS office, the National Science Foundation under grants CCF-0401498, EIA-0323604, and OISE-0405402, and a University of Utah Seed Grant. The work of João L. D. Comba, Fábio Bernardon and Christian Pagot is supported by CNPq grants 478445/2004-0, 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. D. Comba, J. S. B. Mitchell, and C. T. Silva. On the Convexification of Unstructured Grids From A Scientific Visualization Perspective. In G.-P. Bonneau, T. Ertl, and G. M. Nielson, editors, *Scientific Visualization: Extracting Information and Knowledge from Scientific Datasets*. Springer-Verlag, 2005.

[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 IEEE Symposium on Volume Visualization 2000*, pages 91–99, 2000.

[5] R. Farias and C. T. Silva. Parallelizing the ZSWEEP Algorithm for Distributed-Shared Memory Architectures. In *Proceedings of the Joint IEEE and Eurographics Workshop on Volume Graphics 2001*, pages 181–194, 2001.

[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. T. Silva, and B. Wei. A Hardware-Assisted Visibility-Ordering Algorithm With Applications to Volume Rendering. In *Proceedings of the EG/IEEE TCVG Symposium on Data Visualization 2001*, pages 233–242, 2001.

[8] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proc. IEEE Visualization*, pages 287–292, 2003.

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

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

[11] S. Roettger, S. Guthe, D. Weiskopf, and T. Ertl. Smart Hardware-Accelerated Volume Rendering. In *Procceedings of EG/IEEE TCVG Symposium on Data Visualization 2003*, pages 231–238, 2003.

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

[13] 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.

[14] M. Weiler, P. N. Mallón, M. Kraus, and T. Ertl. Texture-Encoded Tetrahedral Strips. In *Proceedings of Symposium on Volume Visualization 2004*, pages 71–78, 2004.

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

Table 3: Summary of performance statistics for each dataset tested. We report the minimum and maximum frame rate (FPS), and also the minimum and maximum rendering rates in thousands of tetrahedra per second.

ATI 9700 Pro	Min FPS	Max FPS	Min Tets/s	Max Tets/s
spx1	1.18	1.68	121K	173K
blunt	2.18	7.09	398K	1.32M
f117	1.36	2.46	326K	590K
GeForce 6800 GT	**Min FPS**	**Max FPS**	**Min Tets/s**	**Max Tets/s**
spx1	1.88	3.38	195K	350K
blunt	0.93	10.64	173K	1.99M
f117	1.83	3.77	438K	906K

Table 4: Number of ray casting passes necessary in the 2D pre-integration.

Mesh	Min Pass	Max Pass
spx1	571	601
blunt	341	439
f117	571	601

Table 5: Rendering time in milliseconds when using a 2D pre-integration lookup table.

ATI 9700 Pro	1×1	2×2	3×3	4×4	5×5	6×6
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
GeForce 6800 GT	**1×1**	**2×2**	**3×3**	**4×4**	**5×5**	**6×6**
spx1	296-516	312-500	313-547	296-531	312-531	312-610
blunt	94-1250	109-1125	109-1094	94-1078	109-1094	125-1094
f117	281-610	265-547	266-562	265-547	266-563	266-828

Table 6: Rendering time in milliseconds when using a 3D pre-integration lookup table.

ATI 9700 Pro	1×1	2×2	3×3	4×4	5×5	6×6
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
GeForce 6800 GT	**1×1**	**2×2**	**3×3**	**4×4**	**5×5**	**6×6**
spx1	297-516	312-516	312-515	297-500	313-547	329-609
blunt	94-1203	94-1047	94-1062	94-1031	109-1032	109-1047
f117	156-484	157-453	157-453	156-437	172-438	172-469