IEEE/ EG Symposium on Volume and Point-Based Graphics (2008)
H.- C. Hege, D. Laidlaw, R. Pajarola, O. Staadt (Editors)

Memory Efficient GPU-Based Ray Casting for Unstructured Volume Rendering

A. Maximo[1] and S. Ribeiro[1] and C. Bentes[2] and A. Oliveira[1] and R. Farias[1]

[1]COPPE - Federal University of Rio de Janeiro, Brazil
[2]DESC - State University of Rio de Janeiro, Brazil

Abstract

Volume ray casting algorithms benefit greatly with recent increase of GPU capabilities and power. In this paper, we present a novel memory efficient ray casting algorithm for unstructured grids completely implemented on GPU using a recent off-the-shelf nVidia graphics card. Our approach is built upon a recent CPU ray casting algorithm, called VF-Ray, that considerably reduces the memory footprint while keeping good performance. In addition to the implementation of VF-Ray in the graphics hardware, we also propose a restructuring in its data structures. As a result, our algorithm is much faster than the original software version, while using significantly less memory, it needed only one-half of its previous memory usage. Comparing our GPU implementation to other hardware-based ray casting algorithms, our approach used between three to ten times less memory. These results made it possible for our GPU implementation to handle larger datasets on GPU than previous approaches.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Display algorithms I.3.7 [Three-Dimensional Graphics and Realism]: Raytracing

1. Introduction

Direct volume rendering is an important technique for visualizing 3-D data volumes. It provides high quality images of the interior of the volume data that have valuable contributions in many different areas, such as medical images analysis, geological data visualization, weather simulations, or fluid dynamics interactions.

Recently, the increasing capability and performance of commercial graphics hardware, such as ATI Radeon and nVidia GeForce, brought hardware implementations of volume rendering to gain the attention of researchers, as they allow renderers to achieve interactive frame rates. Several hardware-based algorithms have been proposed in the literature [WKME03b, WKME03a, EC05, MMFE06]. Some approaches are based on the cell projection algorithm [WKME03a, MMFE06] and others on the ray casting algorithm [WKME03b, EC05].

In the ray casting approach for unstructured grids, rays are casted from the viewpoint through every pixel of the image, and by line/plane intersection, it is possible to determine all cells intersected throughout the ray traversal. Every pair of intersections is used to compute a contribution for the pixel color and opacity. In the cell projection approach, however, each polyhedral cell of the volumetric data is projected onto the screen, requiring the cells to be first sorted in visibility order. It is done in order to correctly determine the contribution of each cell for the color and opacity values of the final image pixels.

The great advantages of ray casting methods are that the computation for each pixel is independent of the others, and the traveling of a ray throughout the mesh is guided by the connectivity of the cells, avoiding the need of sorting the cells. The disadvantage is that cells connectivity has to be explicitly computed and kept in memory.

When these two approaches are compared in terms of performance, for smaller datasets, the cell projection approach performs better [EC05, MMFE06]. For larger datasets, however, ray casting usually performs better than cell projection because it does not suffer from the high cost of sorting the data before the rendering process.

Although ray casting has been more efficient for large datasets, their recent implementations on the GPU present high memory requirements [WKME03b, EC05]. These high requirements, make the GPU implementations unsuitable for handling large datasets, i.e. made of several millions of tetrahedra. The graphics hardware memory is usually smaller than the CPU memory, imposing strong limitations in the store of cell connectivity.

In this paper, we propose a novel hardware-based ray casting algorithm for tetrahedral meshes that tackles this problem. Our idea is to take advantage of a recent memory efficient software implementation of ray casting, called Visible Faces Driven Ray Casting [RMB*07] – *VF-Ray*. This algorithm uses a reduced and non-redundant data structure that provides consistent and significant gains in memory usage, and explores ray coherence in order to maintain in memory only information of the most recent traversals. As VF-Ray used only from $1/3$ to $1/6$ of the memory used by previous approaches, it can deal with larger datasets. However, as VF-Ray was implemented in the CPU, interactive performance has never been achieved.

In order to achieve interactive performance with VF-Ray, our hardware implementation uses General Purpose computation on GPU (GPGPU) [Har04] and the Compute Unified Device Architecture – CUDA [NVI07] from nVidia. In addition to the use of graphics hardware, we propose a restructure in the original data structures of VF-Ray. As a result, our algorithm is much faster than the original software version, while using significantly less memory.

Our results show that, when our algorithm is compared to other recent ray casting hardware-based algorithms, it uses from 77% to 90% less memory. In this way, our algorithm could render datasets for which the others failed due to the lack of memory in the graphics card.

The remainder of this paper is organized as follows. In the next section we discuss the related work. Section 3 briefly describes the VF-Ray algorithm. Section 4 describes our hardware ray casting algorithm and the improvements in its data structures. In section 5, we present the results of our most important experiments. Finally, in section 6, we present our conclusions and proposals for future work.

2. Related Work

Many volume rendering algorithms have been proposed throughout the years. Volume ray casting is the most popular one, and several implementations have been developed. Garrity [Gar90] proposes the use of the cell connectivity to compute the ray traversal inside the volume. This approach was further improved by Bunyk *et al.* [BKS97], where all cells are broken into faces, in a preprocessing step. The Bunyk's algorithm determine the visible faces, that when projected on the screen, define the entry point for each pixel, for the ray casting through the volumetric data. This approach, however, has to keep some large auxiliary data structures in memory. In the work by Pina *et al.* [APCBRF07], two new data to the ray casting algorithm were proposed, they achieved significant gains in memory usage. Unfortunately, their algorithms are slower than Bunyk. The efforts in CPU-based approaches continue with the recent work of Ribeiro *et al.* [RMB*07] introducing the VF-Ray algorithm. In their work, instead of keeping the information of all faces in memory like Bunyk, VF-Ray keeps only a limited set of faces, in order to bring down the memory footprint. Nonetheless, VF-Ray stores, in cell and face data structures, some indices for faces, which we avoid by indexing the faces in a different way, as described in section 4.1. Also, they keep a list of cells for each vertex to properly determine the next cell when a ray hits a vertex or an edge. This data structure was replaced in our implementation by a simpler method, in order to avoid this problem. All these approaches, however, are time-consuming and could not achieve real-time performance.

In order to achieve interactive frame rates, graphics processing units (GPUs) have been used [WMFC02, WKME03a, EC05, BPaLDCS06, MMFE06]. In the last years, the major increase in the performance and programmability of GPUs has led to significant increase in the ray casting performance. Weiler *et al.* [WKME03a] implemented a hardware-based ray casting algorithm, that is based on the work of Garrity *et al.*. This algorithm was further extended by Espinha and Celes [EC05]. The original algorithm uses the pre-integration technique to evaluate the final color, while Espinha and Celes approach uses the partial pre-integration technique [MA04]. Bernardon *et al.* [BPaLDCS06] proposes a hardware-based algorithm based on Bunyk's implementation and depth peeling to render non-convex unstructured grids. On the other hand, Weiler *et al.* [WMKE04] propose the use of tetrahedral strips, in order to deal with the problem of storing the whole dataset in GPU memory.

In the scope of projection algorithms, there are some software implementations [MHC90, FMS00] that provide flexibility and easy parallelization. The GPU implementations of projection algorithms, however, are more notorious. One of the first hardware implementation of cell projection was proposed by Wylie *et al.* [WMFC02], and was based on the Projected Tetrahedra (PT) algorithm by Shirley and Tuchman [ST90]. The PT algorithm decomposes the tetrahedral cells into triangles, according to a projection class, and transmit them to the triangle rendering hardware. Wylie *et al.* extended PT algorithm to implement all tetrahedral primitives directly into the GPU. Further, Weiler *et al.* [WKME03b] developed a combined ray casting and cell projection approach, where the projection is done in a view-independent way. More recently, Marroquim *et al.* [MMFE06] proposed another improvement in the PT algorithm that was implemented almost entirely on GPU and could keep the whole model in the texture memory, avoiding the bus transfer over-

head. Their proposal also used partial pre-integration in order to improve the quality of the image. The work of Callahan et al. [CICS05] introduced the Hardware-Assisted Visibility Sorting (HAVS) algorithm which simplifies the CPU-based processing and shifts much of the sorting burden to the GPU. More recently, Callahan et al. [CBPS06] used the HAVS algorithm to provide a client-server architecture that incrementally stream portions of the mesh from a server to a client.

3. Visible Faces Driven Ray Casting

The Visible Face Driven Ray Casting algorithm, called *VF-Ray* [RMB*07], deals with unstructured grids composed of tetrahedral or hexahedral cells. Each tetrahedral cell is composed of four faces and each hexahedral cell is composed of six faces. The algorithm is based on the following premise: the storing of the information about the faces of each cell is the key for memory consumption and execution time. This information is stored in a face data structure, that includes the geometry and the face parameters, constants for the plane equation defined by the face, which is the most consuming data structure in ray casting.

The basic idea behind VF-Ray is to explore ray coherence, improving caching performance by keeping in memory only the face data of the traversals of a set of nearby rays. The nearby rays, which will be casted through the neighboring pixels, are under the projection of a given visible face, as shown in Figure 1. This set of pixels is called *visible set*.

The algorithm first creates a list of vertices, a list of cells and, for each vertex, a list of all cells incident to the vertex, called the *Use_set* of the vertex. The rendering process starts by determining the visible faces from the external faces of the volume. An external face is a face that belongs to only one cell and is not shared by any other cell [BKS97] and a visible face is an external face whose normal makes an angle greater than 90^o with the viewing direction. The VF-Ray algorithm processes each visible face at a time, projecting it onto the screen, determining the visible set. For each pixel in this set, the ray traversal is done and each pair of intersections is computed. The internal faces are determined by the intersections and the faces data are stored in a temporary buffer, called *computedFaces*. Whenever a ray casted from the current visible set hits an internal face, already stored in the buffer, the VF-Ray algorithm just reads the face data from the computedFaces buffer, without having to recompute the faces parameters. The face data are used to evaluate the traversed distance d and the entry and exit scalar values, s_0 and s_1 respectively. Finally, the cell contribution for the pixel color and opacity is computed using d, s_0 and s_1 and an illumination integral.

VF-Ray algorithm explores ray coherence, using the visible face information to guide the creation and destruction of face data in memory. The faces intersected by neighbor rays

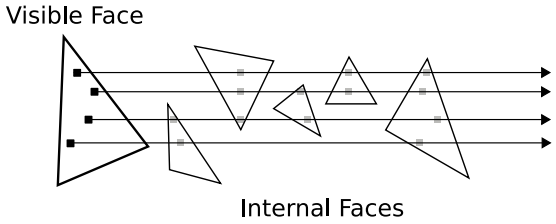

Figure 1: *Ray coherence for one visible face.*

tend to be the same (as shown in Figure 1) and their data are kept in the computedFaces buffer while the visible set is being computed. When all pixels in a visible set are processed, the VF-Ray algorithm clears the computedFaces buffer.

3.1. Handling Degeneracies

Degenerated situations may occur in the ray casting methods, when a ray hits an edge or a vertex. In Bunyk approach, the algorithm would check for the next intersection only on the cells neighboring the current cell faces, and, in this way, would not be able to continue the ray traversal, generating incorrect pixels colors. VF-Ray can handle these two situations by using the Use_set data structure. When a ray hits a vertex v, VF-Ray can find the next intersection by scanning the cells belonging to the Use_set of v. When the ray hits an edge v_0v_1, Vf-Ray can find the next intersection by scanning the Use_set of v_0 and v_1. In this way, VF-ray guarantees that the image will be correctly generated.

4. Memory Efficient GPU-Based Ray Casting

Our hardware-based ray casting algorithm relies on the GPGPU implementation concept to parallelize the original VF-Ray algorithm. The idea behind this concept is to assume the graphics hardware as a coprocessor capable of performing high arithmetic intensity tasks, regardless the graphics board specific computations. The architecture used to implement our GPGPU ray casting algorithm was CUDA [NVI07] due to its simplicity and the possibility of exploitation of all GPU resources. In this architecture, the implementation is done through kernels, that run in multiple threads. The threads are grouped in blocks, meaning that threads inside the same block share the resources of one GPU multiprocessor. Each multiprocessor has a Single Instruction Multiple Data (SIMD) architecture which runs the same kernel instructions but operating on different data.

We divide our algorithm in three steps and, as a result, in three different kernels (see Figure 2). In the first kernel, the external faces are read and the visible faces are determined. The second kernel computes the projection of each visible face, splitting them in pixels on the screen space. Finally, the third kernel evaluates the ray casting algorithm over each visible face pixel previously computed.

The first kernel is important to reduce the computation done by the next two kernels. In the first kernel, the computation is bounded to the number of external faces, while in the second and third kernels, the computation is associated with the number of visible faces. The latter number is generally half of the former number.

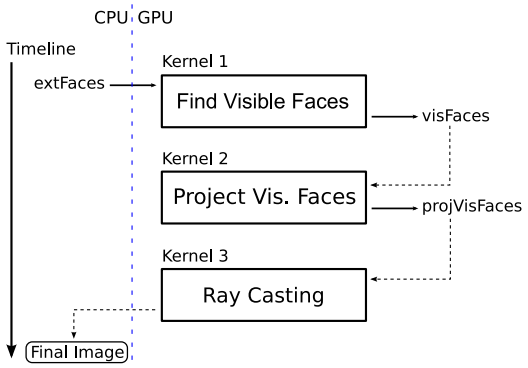

Figure 2: *The three kernels of our algorithm.*

4.1. Data structures

Before the three GPU kernels can be processed, we precompute the following data structures. From the list of vertices (*vertList*) and list of tetrahedra (*tetList*) of the volumetric dataset, we compute the tetrahedra connectivity (*conTet*) which stores for each tetrahedron face, the tetrahedron id t_i which shares the face. Figure 3 shows the data structure used by our algorithm, illustrating the first tetrahedron ($tetrahedron_0$). The vertList contains the x, y and z coordinates and the scalar value s of each vertex. The tetList contains the vertices id v_i which composes each tetrahedron.

To avoid building other data structures, we use the vertices order inside the tetList to determine each tetrahedron face. The vertices of the face f_i are v_i, $v_{(i+1)mod4}$ and $v_{(i+2)mod4}$. For example, the face f_2 of some tetrahedron t_i is composed by the vertices v_2, v_3 and v_0 of t_i, as can be seen in Figure 3. In addition to these data structures, our algorithm uses the external faces list (*extFaces*) pre-computed in the same way as Bunyk.

The original VF-Ray algorithm stores three vertices indices inside its face data structure. In our algorithm, since we target to run in GPU limited memory, we store only two indices for each face created. These indices are the tetrahedron t_i and the face f_i, which are sufficient to identify the vertices of each face, as explained above.

4.2. Find Visible Faces kernel

The first kernel reads the external faces from texture memory and computes the parameters, i.e. plane equation coefficients, for each one. This computation is called face creation

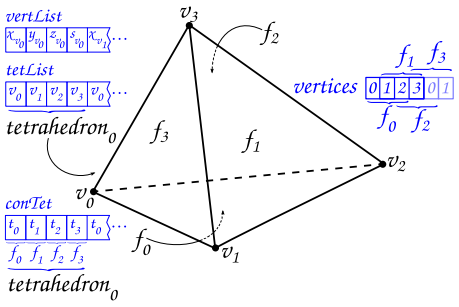

Figure 3: *The basic data structures for our GPU-Based ray casting algorithm.*

and targets to solve two $3x3$ linear systems: one to interpolate the z coordinate; and another to interpolate the scalar value s. In this kernel, only the z coordinate interpolation is done for each external face, once this face is checked as visible the s value interpolation is made and both interpolation parameters are stored in the list of visible faces (*visFaces*).

The visibility for an external face is done by comparing the z coordinate of the fourth vertex of the tetrahedron, i.e. the vertex that does not belong to the face, with the z coordinate of its projection on the external face. The fourth vertex projection is done using the z coordinate interpolation parameters computed for the external face. The face is visible if the projected z is smaller than the z coordinate of the fourth vertex.

Only the visible face parameters are written in CUDA global memory, that is, the z and s interpolation parameters are stored in the visFaces list.

The first kernel employs the visible face test, i.e. back face culling, and face creation for each thread, using the maximum number of threads available per block. The number of external faces is fixed for each volume data and thus is the number of threads across all blocks in this kernel. The computation time and memory footprint spent in the first kernel correspond to less than 5% of the total (time and memory). Nevertheless, the main role of this kernel is to reduce the number of threads that will be used by the next two kernels. From now on, they will run over the visFaces list instead of the extFaces list, which is about half the size of the extFaces list.

4.3. Project Visible Faces kernel

The second kernel reads the coordinates of the vertices of each visible face from the vertList in texture memory. The vertices ids to access the vertList are read from the tetList in global memory. Note that texture components can not be indexed directly, forcing us to store the tetList in global memory, instead of texture memory, in order to avoid unnecessary branches.

The vertices coordinates are used to project the visible faces onto the screen space. The bounding box of the projection is computed and stored in the projected visible faces list (*projVisFaces*), which is written in global memory to be used by the next kernel.

The second kernel uses one thread for each visible face computation. As the first kernel, this one is performed by each thread using few resources of a block. Therefore, the number of threads per block can be maximized and the computation time and memory footprint are unnoticeable.

4.4. Ray Casting kernel

The third kernel reads the visFaces list, computed in the first kernel, and the projVisFaces list, computed in the second kernel. The visible face pixels are determined using the bounding box of the projection and computing a point inside triangle test. This test employs simple cross product operations. For each visible face pixel determined, the ray casting is triggered using the visible face as the first entry face. All the pixels will use the face parameters stored in the visFaces list. From this point, the algorithm travels finding each next face and computing the ray integration using the same integration method proposed by Bunyk.

Just like VF-Ray, we store the next faces in the 1D computedFaces buffer. However, our ray casting algorithm is designed to run in parallel taking advantage of the ray coherence inside each thread computation. In order to avoid dynamic allocation, the computedFaces buffer is allocated in CUDA local memory for each thread with a fixed size and it is indexed by a hash table. The hash index is the z coordinate centroid of the face, refer to Figure 4 for more details. The first buffer position is associated with the minimum z coordinate (*minZ*) of the volume dataset, while the last position with the maximum z coordinate (*maxZ*). Figure 4 presents two near rays processed by the same thread one after the other. The ray 1 creates four internal faces and the ray 2 reads them from the computedFaces buffer.

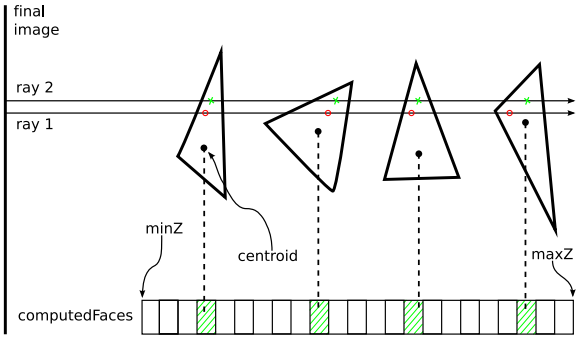

Figure 4: *The **computedFaces** buffer stores previous hits (red circles) to be read in future hits (green crosses). The **centroid** is used to hash the buffer.*

Furthermore, the computedFaces buffer stores the tetrahedron id t_i and the face id f_i apart from the face parameters. These two indices secure that the face to be read is the face that was hit, in case of collision in the hash slot. On the other hand, the original VF-Ray algorithm spends more memory storing four face indices for each tetrahedron to point to the computedFaces buffer.

The third kernel divides the CUDA grid in blocks (see Figure 5) associating one block to one visible face. Each block is, in turn, divided into threads where each one computes a small set of pixels. The threads with pixels outside the visible face, or fewer pixels inside, finish their computation faster than the other threads, but are not wasted. These threads are warped to other blocks, by the graphics hardware driver, in order to continue the kernel computation. Unlike the first two kernels, the amount of computation performed in the ray traversal consumes much more resources of a block, making the computation of all visible face pixels by one thread impractical.

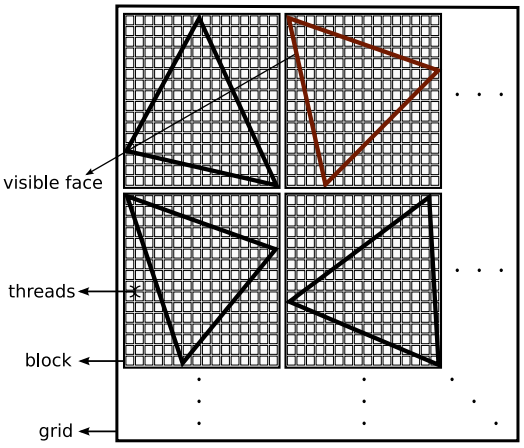

Figure 5: *The CUDA grid scheme used in our implementation. Each block inside the grid computes one visible face, while each thread inside the block computes a set of pixels.*

4.5. Handling Degeneracies

We treat the degenerate cases differently than the original VF-Ray. While they use the Use_set list to correctly find the next tetrahedron for the traversal, we perturb the ray to get rid of the case, then the next iteration continues with the same ray direction without this displacement. Our approach obtains approximate results, but saving the memory spent to keep the Use_set list.

5. Results

In this section we present the results of the performance and memory usage of our hardware-based ray casting algorithm. The algorithm was written in CUDA language, which is an

extension of a subset of C programming language. Our experiments were conducted on an Intel Pentium Core 2 Duo 6400 with 2.13 GHz per processor and 2 GB RAM. We used a nVidia GeForce 8800 Ultra graphics hardware with 768 MB of memory.

The evaluation of our algorithm, that we call VF-Ray-GPU, was done against recent hardware-based algorithms. We also compare VF-Ray-GPU with the original version of VR-Ray that runs solely on the CPU. The idea behind this last comparison is to: first, how much the use of graphics hardware can improve the performance over a CPU implementation; and second, the gain we obtained by using different and more efficient structures. Before evaluating our results, we give a brief description of the hardware-based algorithms used as baselines for our comparisons.

5.1. Baselines

The following algorithms was used for comparison:

VICP: The View-Independent Cell Projection algorithm (VICP) was proposed by Weiler *et al.* [WKME03b]. This is a hybrid version of a cell projection algorithm. The algorithm projects the each cell of the dataset, but the integrations inside the cells are performed exactly in the same way as the ray casting does.

HARC: The Hardware-Based Ray Casting algorithm (HARC) was proposed by Weiler *et al.* [WKME03a] and is based on Bunyk *et al.* [BKS97]. The idea is to store the adjacency information in texture and compute the contribution of each cell by accessing a 3D texture indexed by the pre-integration results. The intersections are computed using faces normals and the integration uses the pre-computed gradient for each cell.

HARC-Partial: This algorithm [EC05] is an extension of the HARC algorithm with partial pre-integration. The algorithm also employs an alternative data structure, in order to reduce the memory usage.

5.2. Workload

We used four different datasets in our experiments: Blunt Fin, Oxygen Post and Fighter from NASA's NAS website, and F16 from EADS. Screenshots of the datasets are shown in Figure 6. Table 1 shows the number of vertices, faces, boundary faces, i.e. external faces, and cells for each dataset. As we can observe in this table, we used two small size datasets, Blunt and Oxygen, and two large size datasets, Fighter and F16. The Fighter dataset used in our experiments is, in fact, a larger and more precise version of the original NASA dataset, that we call Fighter+. All the results considered that the volume dataset is constantly rotating. For small datasets, we used a viewport of 512^2 pixels, and for large datasets, we used a higher resolution, 1024^2 pixels, in order to detail the large number of cells in these volumes.

Dataset	# Verts	# Faces	# Boundary	# Cells
Blunt	41 K	381 K	13 K	187 K
Oxygen	109 K	1 M	27 K	513 K
F16	1.1 M	12.9 M	309 K	6.3 M
Fighter+	1.9 M	22.1 M	334 K	11.2 M

Table 1: *Datasets sizes.*

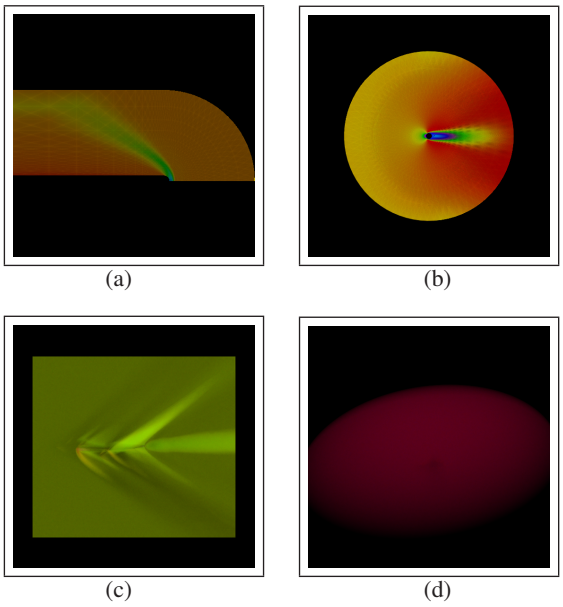

Figure 6: *Datasets: Blunt Fin (a), Oxygen Post (b), Fighter (c) and F16 (d).*

5.3. Small Datasets

In this section we evaluate the performance and memory usage of VF-Ray-GPU against the other hardware-based implementations results. We used in this evaluation only the small datasets, since the other algorithms could not handle the large datasets.

Table 2 shows the time and memory usage results for the baseline rendering algorithms and VF-Ray-GPU, for Blunt and Oxygen. Each two columns for each dataset summarizes the total memory footprint (in kilobytes) and timing (in milliseconds).

Table 3 shows some memory usage aspects of the algorithms. The number of bytes per tetrahedron needed to store the volume data (Bytes/Tet); the number of bytes per pixel, that depends on the final image (Bytes/Pixel); and the total number of megabytes spent in pre-integration or partial pre-integration technique (Pre-Int.). These numbers indicate how the memory usage grows with respect to the dataset size, the final image precision and the use of pre-integration table lookup.

Algorithm	Blunt Fin		Oxygen Post	
	Memory (KB)	*Time (ms)*	*Memory (KB)*	*Time (ms)*
VICP	118,524	190	249,928	546
HARC	72,267	18	123,245	33
HARC-Partial	22,636	32	50,248	51
VF-Ray-GPU	7,029	186	19,494	370

Table 2: *Memory and timing results for VF-Ray-GPU and other hardware-based algorithms.*

Algorithm	*Bytes/Tet*	*Bytes/Pixel*	*Pre-Int*
VICP	456	–	16
HARC	160	96	16
HARC-Partial	96	96	1
VF-Ray-GPU	38	–	–

Table 3: *Hardware-based algorithms memory usage.*

When we compare VF-Ray-GPU with the other ray casting algorithms, HARC and HARC with partial pre-integration (HARC-Partial), we can observe that, our algorithm uses only about 33% of the memory used by HARC with partial pre-integration and about 12% of the memory used by the original HARC algorithm. However, VF-Ray-GPU is slower than both hardware ray casting implementations. This is due mainly to the ray traversal technique.

The main difference between the two hardware ray casting algorithms and VF-Ray-GPU resides on the ray traversal and integration techniques. The ray traversal in HARC is done using pre-computed normals per face and gradients per tetrahedron. In VF-Ray-GPU approach, computations over the face's parameters replace the use of normals and gradients for the ray traversal. The HARC's ray integration technique uses the pre-integration (or partial pre-integration) method to evaluate each cell contribution. VF-Ray-GPU algorithm, on the other hand, uses a volume rendering integral computed on-the-fly. The use of faces slows down the VF-Ray-GPU, nevertheless, it reduces the memory used per tetrahedron by more than 50%, compared with the two HARC algorithms. This type of memory footprint is fundamental when dealing with large volume datasets.

When we compare VF-Ray-GPU with the hybrid algorithm, we observe that VICP algorithm uses 16 times more memory and is slower than VF-Ray-GPU for Oxygen. VICP being a hybrid algorithm, uses all necessary data for ray casting and it also employs part of the cell projection calculation, which makes it the most memory consuming algorithm.

5.4. Large Datasets

In this section we evaluate the performance and memory usage of VF-Ray-GPU for large datasets. One of the most important results of our work is that from all hardware-based ray casting algorithms, VF-Ray-GPU was the only one that could render F16 and Fighter+. The GeForce 8800 Ultra does not have enough memory, required by the other ray casting algorithms.

In Table 4 we compare VF-Ray-GPU with VF-Ray. The table shows the memory footprint (in kilobytes) and the total execution time (in millisecond) for the two large datasets rendered by VF-Ray-GPU (GPU) and the original VF-Ray algorithm (CPU). As we can observe in this table, in comparison with the original VF-Ray, our algorithm runs about 4 times faster and uses about 50% less memory. The performance gain is due to the parallel nature of our ray casting algorithm, while the data restructuring improves the memory usage.

Datasets	Memory (MB)		Time (ms)	
	CPU	GPU	CPU	GPU
Fighter+	876	426	16263	3081
F16	499	239	2515	804

Table 4: *Memory footprint and timing comparison between VF-Ray-CPU and VF-Ray-GPU algorithms.*

6. Conclusions

In this work, we have proposed a novel hardware-based ray casting algorithm, based on a memory-aware software implementation of ray casting, VF-Ray. The main contribution of our proposal was to provide ray casting in hardware graphics with low memory usage in order to handle large datasets. Besides the low memory consumption, our algorithm also allows the handling of some degenerate cases, produced by the hardware implementations that are based on previous ray casting approaches, like Bunyk approach.

We compared our algorithm with other recent hardware-based volume rendering algorithms, based on the ray casting paradigm, and on a hybrid version mixing ray casting and cell projection. For smaller datasets, our algorithm is slower than the ray casting approaches, but uses from 77% to 90% less memory. When compared to the hybrid approach, our algorithm is faster and uses about 95% less memory.

For larger datasets, our algorithm could render datasets for which the others failed, due to the lack of memory in the graphics card. Our results show a linear increase in the execution time with the increase in the dataset size, indicating

that the algorithm is scalable to larger datasets. In addition, our algorithm was faster than the original VF-Ray, while using significantly less memory.

As future work we intend to use the thread synchronization scheme of CUDA to handle non-convex meshes. Another future work is to use the partial pre-integration technique, in order to improve rendering quality.

7. Acknowledgments

We would like to thank Claudio Silva for providing us the Fighter and F16 datasets (Fighter from Neely and Batina – NASA, and F16 from Udo Tremel – EADS-Military). We also acknowledge the grant of the first and second authors provided by Brazilian agencies CNPq and CAPES, respectively.

References

[APCBRF07] ALINE PINA, CRISTIANA BENTES, RICARDO FARIAS: Memory efficient and robust software implementation of the raycast algorithm. In *WSCG'07: The 15th Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision* (2007).

[BKS97] BUNYK P., KAUFMAN A. E., SILVA C. T.: Simple, fast, and robust ray casting of irregular grids. In *Dagstuhl '97, Scientific Visualization* (Washington, DC, USA, 1997), IEEE Computer Society, pp. 30–36.

[BPaLDCS06] BERNARDON F. F., PAGOT C. A., AO LUIZ DIHL COMBA J., SILVA C. T.: GPU-based Tiled Ray Casting using Depth Peeling. *Journal of Graphics Tools 11.3* (2006), 23–29.

[CBPS06] CALLAHAN S. P., BAVOIL L., PASCUCCI V., SILVA C. T.: Progressive volume rendering of large unstructured grids. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1307–1314.

[CICS05] CALLAHAN S. P., IKITS M., COMBA J. L., SILVA C. T.: Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (2005), 285–âĂŞ295.

[EC05] ESPINHA R., CELES W.: High-quality hardware-based ray-casting volume rendering using partial pre-integration. In *SIBGRAPI '05: Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing* (2005), IEEE Computer Society, p. 273.

[FMS00] FARIAS R., MITCHELL J. S. B., SILVA C. T.: ZSWEEP: an efficient and exact projection algorithm for unstructured volume rendering. In *VVS'00: Proceedings of the 2000 IEEE Symposium on Volume visualization* (New York, NY, USA, 2000), ACM Press, pp. 91–99.

[Gar90] GARRITY M. P.: Raytracing irregular volume data. In *VVS'90: Proceedings of the 1990 workshop on Volume visualization* (New York, NY, USA, 1990), ACM Press, pp. 35–40.

[Har04] HARRIS M.: GPGPU – General-Purpose computation using Graphics Hardware, 2004. http://www.gpgpu.org/.

[MA04] MORELAND K., ANGEL E.: A fast high accuracy volume renderer for unstructured data. In *VVS '04: Proceedings of the 2004 IEEE Symposium on Volume visualization and graphics* (Piscataway, NJ, USA, 2004), IEEE Press, pp. 13–22.

[MHC90] MAX N., HANRAHAN P., CRAWFIS R.: Area and volume coherence for efficient visualization of 3d scalar functions. In *VVS'90: Proceedings of the 1990 workshop on Volume visualization* (New York, NY, USA, 1990), ACM Press, pp. 27–33.

[MMFE06] MARROQUIM R., MAXIMO A., FARIAS R., ESPERANCA C.: GPU-Based Cell Projection for Interactive Volume Rendering. In *SIBGRAPI '06: Proceedings of the XIX Brazilian Symposium on Computer Graphics and Image Processing* (Los Alamitos, CA, USA, 2006), IEEE Computer Society, pp. 147–154.

[NVI07] NVIDIA™: CUDA Environment – Compute Unified Device Architecture, 2007. http://www.nvidia.com/object/cuda_home.html.

[RMB*07] RIBEIRO S., MAXIMO A., BENTES C., OLIVEIRA A., FARIAS R.: Memory-Aware and Efficient Ray-Casting Algorithm. In *SIBGRAPI '07: Proceedings of the XX Brazilian Symposium on Computer Graphics and Image Processing* (Los Alamitos, CA, USA, 2007), IEEE Computer Society, pp. 147–154.

[ST90] SHIRLEY P., TUCHMAN A. A.: Polygonal approximation to direct scalar volume rendering. In *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics* (1990), vol. 24(5), pp. 63–70.

[WKME03a] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of the 14th IEEE conference on Visualization '03* (2003), pp. 333–340.

[WKME03b] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics 9*, 2 (2003), 163–175.

[WMFC02] WYLIE B., MORELAND K., FISK L. A., CROSSNO P.: Tetrahedral projection using vertex shaders. In *VVS'02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics* (Piscataway, NJ, USA, 2002), IEEE Press, pp. 7–12.

[WMKE04] WEILER M., MALLON P. N., KRAUS M., ERTL T.: Texture-encoded tetrahedral strips. In *VV '04: Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 71–78.