Memory-Aware and Efficient Ray-Casting Algorithm

Saulo Ribeiro[1], André Maximo[1], Cristiana Bentes[2], Antônio Oliveira[1], and Ricardo Farias[1]

[1] COPPE - Systems Engineering Program
Federal University of Rio de Janeiro - Brazil
Cidade Universitária - CT - Bloco H - 21941-972
{saulo, andre, antonio, rfarias}@lcg.ufrj.br

[2] Department of Systems Engineering
State University of Rio de Janeiro - Brazil
Rua São Francisco Xavier, 524 - 20550-900
cris@eng.uerj.br

Abstract

Ray-casting implementations require that the connectivity between the cells of the dataset to be explicitly computed and kept in memory. This constitutes a huge obstacle for obtaining real-time rendering for very large models. In this paper, we address this problem by introducing a new implementation of the ray-casting algorithm for irregular datasets. Our implementation optimizes the memory usage of past implementations by exploring ray coherence. The idea is to keep in main memory the information of the faces traversed by the ray cast through each pixel under the projection of a visible face. Our results show that exploring pixel coherence reduces considerably the memory usage, while keeping the performance of our algorithm competitive with the fastest previous ones.

1. Introduction

Direct volume rendering has become a popular technique for visualizing volumetric data from sources such as scientific simulations, analytic functions, and medical scanners, such as MRI, CT, among others. The main advantage of direct volume rendering is to allow the investigation of the interior of the data volume, exhibiting its inner structures.

Several algorithms and methods have been proposed for direct volume rendering, the most popular one is the *ray-casting* algorithm. In this algorithm, a ray is cast through each pixel of the image from the viewpoint. The trace of the ray determines which cells of the volume each ray intersects. Each pair of intersections is used to compute the cell contribution for the pixel color and opacity. The ray stops when it reaches full opacity or when it leaves the volume.

Compared to other direct volume rendering methods, the great advantages of ray-casting methods are: the computation for each pixel is independent of all other pixels; and the traveling of a ray through out the mesh is guided by the connectivity of the cells of the mesh, avoiding the need of sorting the cells. The disadvantage is that cell connectivity has to be explicitly computed and kept in memory. In other words, the amount of memory used by ray-casting methods is a huge obstacle for handling very large models.

Recent previous efforts towards improving ray-casting performance have focused on reducing the execution time by exploiting hardware graphics, or GPUs (Graphic Processing Unit). This approach, however, only provides interactive rendering time when the whole dataset fits into the GPU memory. Once the data size exceeds the onboard memory, expensive transfers from main memory to the GPU have great impact on the rendering speed.

Therefore, the problem of rendering large datasets (that does not fit in GPU memory) must be addressed by software solutions, due to the inherent larger memory capacity [9, 10]. Only a few software solutions, however, deal with irregular meshes. Garrity [5] proposed the first method for ray-casting irregular meshes using the connectivity of cells. Bunyk *et al.* [2] later improved Garrity's work providing a faster algorithm. Pina *et al.* [11] improved Bunyk approach in: memory consumption; completely handling degenerate cases; and dealing with both tetrahedral and/or hexahedral meshes. By using new data structures, Pina *et al.* reduced significantly the memory requirements of the Bunyk approach, but the algorithms proposed were still slower than Bunyk.

In this work, we propose a novel ray-casting algorithm based on Pina *et al.* approaches, reducing even more the memory consumption and improving the execution time. Our approach is based on the fact that the decision of how to store the face information is the key for the ray-casting method. The more information is kept in memory, the faster the algorithm computes the next intersection of the ray. In this work, we tackle this tradeoff by exploring ray coherence in order to maintain in memory only the face information of the most recent traversals. Our idea is to use each

visible face computed in the preprocessing step to guide the creation and destruction of internal faces data in memory. Our algorithm, called Visible Faces Ray-casting (VF-Ray), obtained consistent and significant gains in memory usage over previous approaches. We also reduced the execution time of Pina *et al.* approaches by making a better use of the cache. When compared to the fast and memory expensive approach by Bunyk *et al.*, we obtained comparable performance, spending only from $1/3$ to $1/6$ of the memory.

The remainder of this paper is organized as follows. In the next section we discuss direct volume rendering of irregular meshes. Section 3 briefly describes the Bunyk's and Pina's approaches. Section 4 describes our ray-casting algorithm and the improvements we made over previous approaches. In section 5, we present the results of our most important experiments. Finally, in section 7, we present our conclusions and proposals for future work.

2. Related Work

Within the research area of accelerating ray-casting methods, two main research streams have emerged: software approaches and graphics hardware approaches. The very first implementation of the ray-casting algorithm for rendering irregular meshes was proposed in software by Garrity in [5]. His approach used the connectivity of cells to compute the entry and exit of each ray. This work was further improved by Bunyk *et al.* [2], by computing for each pixel a list of intersections on visible faces, and easily determining the correct order of the entry points for the ray. The rendering process follows Garrity's method, but when a ray exits the mesh, the algorithm can easily determine in which cell the ray will re-enter the mesh. This approach becomes simpler and more efficient than Garrity has proposed, however it keeps some large auxiliary data structures. The recent work by Pina *et al.* [11] proposes two new ray-casting algorithms, ME-Ray and EME-Ray, that presented consistent and significant gains in memory usage and in the correctness of the final image over Bunyk's approach. Their algorithms, however, are slower than Bunyk.

In terms of the acceleration techniques used in this work, ray coherence has been used earlier for accelerating ray tracing of traditional surface models [12, 14], and later used in ray-casting for skiping over empty spaces [6].

With the advent of programmable graphics hardware, several volume rendering algorithms have been developed taking advantage of this feature to achieve high performance. Graphics hardware based solutions usually provide real-time performance and high quality images. Weiler *et al.* [15] present a hardware-based ray-casting algorithm based on the work of Garrity. They find the initial ray entry point by rendering front faces, and then traverse through cells using the fragment program by storing the cells and connectivity graph in textures. Their method, however, work only on convex unstructured data. Bernardon *et al.* [1] improve Weiler's approach, by using Bunyk's algorithm as the basis for the hardware implementation and depth peeling to correctly render non-convex irregular meshes. Espinha and Celes [3] propose an improvement in Weiler's work. They use partial pre-integration and provide interactive modifications of the transfer function. They also use an alternative data structure, but as their work is based on Weiler's work, the memory consumption is still high. Recently, Weiler *el al.* [16] proposed an improvement over their previous work, where they deal with the problem of storing the whole dataset in GPU memory by using a compressed form, tetrahedral strips.

In terms of other direct volume rendering algorithms different from ray-casting, it is important to mention the cell projection approach. In this approach each polyhedral cell of the volumetric data is projected onto the screen, avoiding the need of maintaining the data connectivity into memory, but requiring the cells to be first sorted in visibility ordering and then composed to generate their color and opacity in the final image. In this scope, there are some software implementations [13, 4], that provide flexibility and easy parallelization. The GPU implementations of projection algorithms, however, are more popular [15, 7].

3. Previous Approaches

In this section we briefly describe the ray-casting algorithm proposed by Bunyk *et al.* in [2], that we simply call *Bunyk*, and the ones proposed by Pina *et al.* in [11], called *ME-Ray* (*Memory Efficient Ray-cast*) and *EME-Ray* (*Enhanced Memory Efficient Ray-cast*).

The data structures used by these algorithms to store the face data guided their memory usage and performance. The face data correspond to the information about the geometry and the parameters of the face. The geometry of the face represents the coordinates of the points that define the face. The face parameters are the constants for the equation of the plane defined by the face, that will be used to compute the intersection between the ray and the face. Additional face parameters are used to interpolate the scalar value inside the face. The face data is stored in a structure that we call *face*. The face parameters are costly to compute and the whole face data is the most consuming data structure in ray-casting.

3.1. Bunyk

In a preprocessing step, all points and tetrahedra of the input dataset are read, a list of all vertices and a list of cells are created. In addition, for each input tetrahedron, all four

of its faces are kept in a list. Bunyk also maintains, for each vertex, a list of all faces that use that vertex, called `referredBy` list. After reading the dataset, Bunyk determines which faces belong to the visible side of the scene boundary. These faces are the visible faces. The algorithm projects these visible faces on the screen, creating for each pixel, a list of the intersections, that will be used as the entry point of its ray. After the entry points are stored, the actual ray-casting begins. For every pixel, the first intersection is its entry point into the data, and each next intersection is obtained by inspecting the intersection between the ray and all other faces of the current cell. Each pair of intersections are used to compute the contribution of the cell for the color and opacity of the pixel. The searching for the next intersection in Bunyk is quite fast, since this search is performed among the other three faces of the tetrahedron. However, the list of all faces and the `referredBy` lists result in a very high memory consumption.

3.2. ME-Ray

ME-Ray also starts in a preprocessing step, reading the input set and creating some data structures. It creates a list of all vertices, a list of all cells and the `Use_Set` list for each vertex. The `Use_Set` lists substitute the `referredBy` lists used in Bunyk, each `Use_Set` contains a list of all cells incident on the vertex. Another step in the preprocessing phase is to create for each cell the list of cells that share a face with it. In the rendering algorithm, ME-Ray also creates a list of visible faces, and determine the entry point of each ray, in the same way as Bunyk. The algorithm proceeds looking for the next intersection, by scanning through out the neighbor cells. ME-Ray creates on demand a list of faces, as the face is intersected by the ray. Faces that are never intersected by any ray will not be created. After each pair of intersection is found, ME-Ray computes their contribution to the final color and opacity. ME-Ray can save about 40% of the memory used by Bunyk, but is slower, since it has to create the faces for the first time during the rendering.

3.3. EME-Ray

EME-Ray was developed as an optimization of ME-Ray in terms of memory usage. They have similar behavior. However, EME-Ray does not store the faces in memory, since it was one of the most memory expensive structures in ME-Ray. Therefore, EME-Ray has to recalculate the faces parameters for the verification of ray intersection every time that a new face is found. With this optimization, EME-Ray use only about $1/4$ of the memory used by Bunyk. The significant reductions in memory usage comes with an increase in the execution time, EME-Ray usually doubles Bunyk execution time. It is also important to notice that ME-Ray and EME-Ray gains over Bunyk are not only in memory usage, but also in the correctness of the final image. Their data structures allow them to deal with all degenerate situations that Bunyk is not able to handle.

4. Visible Face Driven Ray-Casting

Comparing the three ray-casting approaches described in the previous section, we can observe that the more faces data are kept in memory, the faster the algorithm computes the next intersection of the ray. For this reason, Bunyk is still the fastest software implementation of the ray-casting algorithm.

Our algorithm, called Visible Face Driven Ray-Casting – **VF-Ray**, addresses the problem of maintaining information about the faces in memory, minimally degrading the rendering performance. The basic idea behind VF-Ray is to keep in main memory only the faces of the most recent traversals. To do so, we explore ray coherence, using the visible face information to guide the creation and destruction of internal faces in memory.

4.1. Exploring Ray Coherence

VF-Ray is based on ray coherence, which comes from the fact that the set of faces intersected by two rays, cast from neighboring pixels, will contain almost the same faces. Our algorithm takes advantage of ray coherence by keeping in memory only the faces of nearby rays.

One important issue in exploring ray coherence is the determination of the set of rays that are likely to have the same set of intersected faces. We use the visible faces information to do so. The set of rays that may reuse the same list of faces are the ones that correspond to the set of pixels under the projection of a certain visible face. This set of pixels will be called here *visible set*. Figure 1 shows an example of this idea. The rays that are cast through the visible face, presented in the figure, tend to intersect the same internal faces. Therefore, the internal faces are created and stored once, for the first ray intersection. After that, they are reused until all the pixels of the visible set are computed. This process is repeated for all visible faces. To guarantee correctness, all visible faces are ordered in depth order.

Our results show that exploiting pixel coherence reduces considerably the memory usage, while keeping the VF-Ray performance competitive with the fastest previous algorithms because the reuse of face data improves cache performance. In fact, for all datasets we tested, approximately more than half of the faces are created at most two times for medium size images.

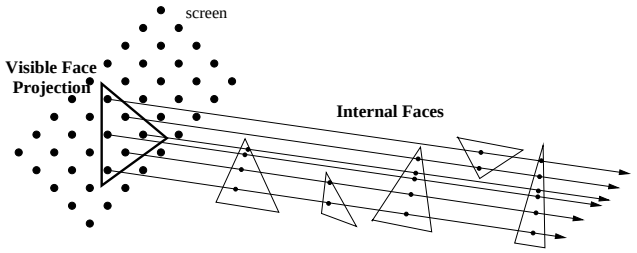

Figure 1. Visible face rays coherence.

4.2. Data structures

The data structures used by VF-Ray are similar to that of EME-Ray: an array of vertices, an array of cells, and the `Use_Set` of each vertex. Besides these structures, VF-Ray also has a list, called `computedFaces` that holds the internal faces throughout the model. Note that this list is cleaned up after the ray-casting computation of all pixels of a visible set. In this way, we don't need to store all the internal faces, as done by Bunyk and ME-Ray algorithms.

Furthermore, just like ME-Ray and EME-Ray, our algorithm can handle any type of convex cell. For tetrahedral meshes, the intersections are computed between the line defined by the ray path and the plane defined by the three vertices of a triangular face. For hexahedral meshes, where each face is quadrangular, the intersection is found by splitting these quadrangular faces into two triangular faces, and the same calculation is performed for each one.

4.3. Handling Degeneracies

The degenerate situations, which can be found during the ray traversal, are treated in the same way that was performed in Pina *et al.* [11], by investigating the `Use_set` of each vertex. So, when a ray hits a vertex or an edge, the algorithm can determine the next intersection correctly.

4.4. Algorithm

VF-Ray algorithm starts by reading the input dataset and creating three lists: a list of vertices, a list of cells and the `Use_Set` list for each vertex. In the rendering process, VF-Ray creates a list of all visible faces, that are the faces whose normal makes an angle greater than 90^o with the viewing direction. For each visible face, the algorithm follows the steps presented in the pseudo-code of Figure 2.

Initially, the visible face, $vface_i$, is projected onto the screen, generating the visible set of $vface_i$. For each pixel p in the visible set, the algorithm first computes the intersection between the ray cast from p and $vface_i$. After this first intersection is found, the ray traversal begins, when the algorithm computes the intersections of the ray with the internal faces. The next face intersected, $face_j$, is found on another face of the current cell. If $face_j$ was not created before (this is the first intersection in $face_j$), the face is created and inserted into `computedFaces` list. Otherwise, the $face_j$ data are loaded from the list. The insertion and loading operations in the list are done in constant time, since in the cell we store the index for the position of each face in the list. The computation of the intersections is done using the face parameters computed when the face is created. For each intersection, the scalar value is interpolated between the three vertices of each face, which are used in the illumination model proposed by Max [8].

```
For each visible face vface_i do
  Project vface_i; // Determine visible set
  For each pixel p of visible set do
    Intersect vface_i; // Interpolate scalar value
    Do
      Find next internal face face_j; // Intersection
      If face_j not in computedFaces then
        Create face_j and Insert in computedFaces;
      else
        Load face_j from computedFaces;
      Ray integrate; // Optical model
    While exist next face face_j;
  End For
  Clear computedFaces; // Clear list
End For
```

Figure 2. The pseudo-code of VF-Ray.

5. Results

In this section we evaluate the performance and memory usage of VF-Ray when compared to ME-Ray, EME-Ray and Bunyk. We also show a comparison between VF-Ray and a cell projection algorithm, called ZSweep [4]. This last experiment was done in order to put our results in perspective, with respect to other rendering paradigm. The main idea of ZSweep algorithm is the sweeping of the data with a plane parallel to the viewing plane XY, towards the positive z direction. The sweeping process is performed by ordering the vertices by their increasing z coordinate values, and then retrieving one by one from this data structure. For each vertex swept by the plane sweep, the algorithm projects, onto the screen, all faces that are incident to it. During face projection, ZSweep stores a list of intersections for each pixel and uses delayed compositing. The pixels lists intersections are composed and the lists are flushed each time the plane reaches the *target-z*. The *target-z* is the

maximum z-coordinate among the vertices adjacent to the vertex intersected by the sweeping plane.

5.1. Testing Configuration

The VF-Ray algorithm was written in C++ ANSI without using any particular graphics libraries. Our experiments were conducted on a Intel Pentium IV 3.6 GHz with 2 GB RAM and 1 MB of L2 cache, running Linux Fedora Core 5.

We have used four different tetrahedral datasets: Blunt Fin, SPX, Liquid Oxygen Post and Delta Wing. Table 1 shows the number of vertices, faces, boundary faces and cells for each dataset. We varied the image sizes, from 512×512 to 8192×8192 pixels. The models rendered with VF-Ray are shown in Figures 3, 4, 5 and 6. The images generated by VF-Ray, ME-Ray, and EME-Ray are identical, and better than the images generated by Bunyk. There are two reasons for this difference: VF-Ray, ME-Ray, and EME-Ray can use double precision to calculate intersections and Bunyks does not handle degenerate situations, generating some incorret pixels.

Dataset	*# Verts*	*# Faces*	*# Tets*	*# Boundary*
*B*lunt	41 K	381 K	187 K	13 K
*S*PX	149 K	1.6 M	827 K	44 K
*P*ost	109 K	1 M	513 K	27 K
*D*elta	211 K	2 M	1 M	41 K

Table 1. Dataset sizes.

5.2. Memory Consumption

Table 2 shows the memory used by VF-Ray to render one frame (in MBytes) for Blunt, SPX, Post and Delta, and the relative memory usage results of ME-Ray, EME-Ray, Bunyk and ZSweep when compared to VF-Ray usage. The percentages presented in this table correspond to the ratio of the other algorithms memory usage over VF-Ray memory consumption. In other words, we consider VF-Ray results as 100% and are presenting how much the other algorithms increase or decrease this result.

As we can observe in Table 2, in most of the cases, VF-Ray uses less memory than the three other algorithms, and its memory usage increases linearly with the resolution. When compared to Bunyk, VF-Ray uses from $1/3$ to $1/6$ of the memory spent by Bunyk. These gains are impressive. For Blunt, for example, VF-Ray renders a 4096×4096 image using the same amount of memory that Bunyk uses to render a 512×512 image. For the largest dataset, Delta, Bunyk uses more memory to render a 512×512 image than VF-Ray uses to render a 8192×8192 image.

When compared with the two memory-aware ray-casting algorithms, ME-Ray and EME-Ray, we can observe that ME-Ray uses from 1.5 to 6 times more memory than VF-Ray. For EME-Ray, we notice an interesting result. Although EME-Ray does not store the faces in memory, for high resolution images, larger than 1024×1024, EME-Ray uses more memory than VF-Ray. This result can be explained by the fact that EME-Ray, just like Bunyk and ME-Ray, maintains for each pixel a list containing the entry faces for the pixel. The number of lists increases with the increase in image resolution. VF-Ray, on the other hand, does not require this kind of list, since it computes the entry point of each ray on demand, for each visible face.

When compared to ZSweep, we can observe that except for SPX with a 512×512 image, VF-Ray uses much less memory than ZSweep. For the higher resolution images, ZSweep can use 8 or 9 times more memory than VF-Ray. To render SPX with a 8192×8192 image, for example, ZSweep spends about 2 GB of memory. The ZSweep memory consumption is directly proportional to the final image resolution, since it creates pixels lists to store the intersections found for each pixel.

5.3. Rendering Performance

Table 3 shows the rendering time (in seconds) of VF-Ray for Blunt, SPX, Post and Delta, and the relative timing results of ME-Ray, EME-Ray, Bunyk and ZSweep when compared to VF-Ray execution. Just like the previous table, the percentages presented in this table correspond to the ratio of the other algorithms execution time over VF-Ray execution time. The time results correspond to the rendering of one frame and do not include preprocessing time.

As we can observe in Table 3, VF-Ray outperforms ME-Ray, EME-Ray and ZSweep for all datasets and all image resolutions. For Blunt, Post and Delta, VF-Ray is about 3 times faster than EME-Ray and ZSweep, and uses about $3/4$ of the time used by ME-Ray to render the image. For SPX, the gains of VF-Ray against EME-Ray and ZSweep are smaller, but still high. When compared to Bunyk, for Blunt and Delta, VF-Ray presents almost the same execution time, since the difference is only about 10%. For SPX and Post, VF-Ray is only about 22% slower than Bunyk.

Observing the increase in the rendering time of VF-Ray for different image resolutions, we notice that as the image resolution increases, the rendering time increases in the same proportion. The gains over the other algorithms are almost the same for different resolutions.

6. Discussion

VF-Ray had obtained consistent and significant gains in memory usage over Bunyk, providing almost the same per-

Dataset	Resolution	Memory (MB)	ME-Ray	EME-Ray	Bunyk	ZSweep
Blunt	512×512	14	404%	166%	528%	208%
	1024×1024	30	240%	129%	297%	177%
	2048×2048	39	333%	251%	377%	369%
	4096×4096	75	495%	451%	517%	689%
	8192×8192	219	610%	655%	617%	917%
SPX	512×512	96	215%	74%	299%	87%
	1024×1024	99	234%	88%	305%	107%
	2048×2148	108	271%	141%	337%	188%
	4096×4096	144	383%	284%	428%	407%
	8192×8192	288	533%	498%	569%	711%
Post	512×512	63	165%	74%	290%	97%
	1024×1024	66	203%	95%	301%	129%
	2048×2048	74	281%	172%	353%	238%
	4096×4096	110	424%	349%	470%	499%
	8192×8192	256	601%	560%	603%	800%
Delta	512×512	116	167%	74%	303%	97%
	1024×1024	119	200%	84%	308%	116%
	2048×2048	129	246%	124%	330%	177%
	4096×4096	165	346%	247%	403%	375%
	8192×8192	307	500%	467%	534%	704%

Table 2. Memory usage results for VF-Ray compared to ME-Ray, EME-Ray, Bunyk and ZSweep.

formance (the difference on their execution time is only about 16%). As the image resolution increases, our gains in memory usage over Bunyk are even more pronounced, but the difference in the rendering time remains the same. For 8192×8192 images, Bunyk uses about 6 times more memory than VF-Ray with a performance difference of only about 18%. In our experiments, however, we are only comparing executions where the whole dataset fits in main memory for all algorithms. Our experimental platform has 2 GB of main memory that can store all the data structures used by our workload. As the memory usage increases, the rendering will need to use the virtual memory mechanisms of the operating system, which would have great influence on the overall execution time.

The small increase in the execution time, when compared to Bunyk, comes from the fact that Bunyk computes all internal faces in the preprocessing step, before the ray-casting loop, while VF-Ray does it on demand, as the face is intersected for the first time. In addition, in VF-Ray, if a face is intersected by rays of two different visible faces, it has its data computed twice.

The comparison of VF-Ray with ME-Ray and EME-Ray provided interesting results. ME-Ray and EME-Ray are memory-aware algorithms designed to reduce the great memory usage of Bunyk. VF-Ray was designed to reduce ME-Ray memory usage, close to the usage of EME-Ray, but running much faster than EME-Ray. Our results showed, however, that for high resolution images, our algorithm is not only faster than ME-Ray but uses less memory than EME-Ray. The gains of VF-Ray over EME-Ray in memory usage come from the elimination of the lists of entry faces for each pixel. The gains of VF-Ray over ME-Ray in execution time come from the good cache performance of VF-Ray. We have made some experiments to evaluate the cache misses of VF-Ray and ME-Ray and obtained for VF-Ray a cache miss ratio very low, near 0%, while the cache miss ratio of ME-Ray is about 1.9% for modest size images. This result was obtained because our algorithm reuse the faces for the rays in the same visible face. Given the fact that rays are shot one after the other, the closer the neighboring rays are to each other, the higher the probability is that they reuse the cached data.

The comparison of VF-Ray with the cell projection approach, implemented by ZSweep, showed that VF-Ray is faster and spends less memory than ZSweep. The gains are huge. For the Delta dataset, ZSweep uses 7 times more memory, running about 3.5 times slower than VF-Ray. The pixel list maintained by the ZSweep approach increases while a *target-Z* was not reached. As this list grows, the insertion is more expensive, since the pixel list must be ordered. On the other hand, the data structures on the VF-Ray algorithm does not depend on ordering. The difference in memory usage of ZSweep is due to the increase in these pixels lists. For ZSweep, as the image size increases, each face projected will insert intersection units into more pixel lists. If the *target-Z* is not appropriate, the pixels lists will increase considerably.

In terms of the dataset characteristics, the performance of VF-Ray increases as the number of pixels in each visible face increases. Datasets with a small number of faces

Dataset	Resolution	Time (sec)	ME-Ray	EME-Ray	Bunyk	ZSweep
Blunt	512×512	1.9	139%	296%	105%	253%
	1024×1024	7.0	143%	317%	94%	431%
	2048×2048	27.2	146%	337%	88%	481%
	4096×4096	107.4	147%	328%	88%	516%
	8192×8192	426.7	154%	341%	91%	382%
SPX	512×512	4.1	111%	161%	76%	287%
	1024×1024	13.0	103%	203%	81%	202%
	2048×2048	46.6	103%	225%	83%	219%
	4096×4096	177.1	105%	234%	82%	226%
	8192×8192	696.3	105%	238%	81%	260%
Post	512×512	5.0	138%	251%	80%	201%
	1024×1024	19.5	136%	254%	76%	167%
	2048×2048	78.6	133%	258%	74%	194%
	4096×4096	309.9	135%	257%	74%	227%
	8192×8192	1246.3	135%	263%	72%	246%
Delta	512×512	3.1	130%	242%	91%	465%
	1024×1024	11.2	122%	270%	88%	271%
	2048×2048	41.9	121%	280%	87%	312%
	4096×4096	162.7	120%	290%	84%	341%
	8192×8192	640.3	123%	290%	83%	369%

Table 3. Time results for VF-Ray compared to ME-Ray, EME-Ray, Bunyk and ZSweep.

have a greater amount of pixels per visible face. Therefore, datasets, like Blunt, that has a small number of faces, tend to present better performance results for VF-Ray.

Finally, it is important to keep in mind that the reductions in memory usage will allow the implementation of the algorithm in the graphics hardware. Furthermore, these reductions will also allow the use of double precision in the parameters to calculate the intersection between a ray and a face, raising the chance to obtain more precise images.

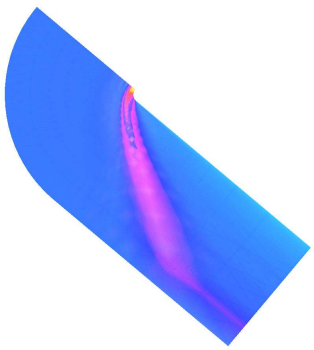

Figure 3. Blunt image generated by VF-Ray.

Figure 4. SPX image generated by VF-Ray.

7. Conclusions

In this work, we have proposed a novel memory-aware and efficient software ray-casting algorithm for volume rendering, called VF-Ray. The algorithm is suitable for tetrahedral or hexahedral datasets, including non-convex and disconnected meshes even with holes.

Our idea was to explore ray coherence in order to reduce the memory requirements for the storing of faces data while achieving a good cache performance. We compared VF-Ray memory usage and performance with previous ray-casting approaches by Bunyk *et al.* and Pina *et al.*, and also with a cell projection algorithm, ZSweep. When compared to the fastest and memory expensive approach by Bunyk

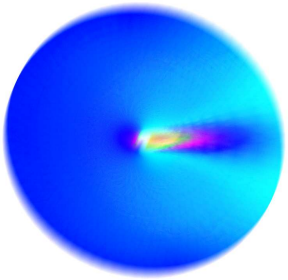

Figure 5. Post image generated by VF-Ray.

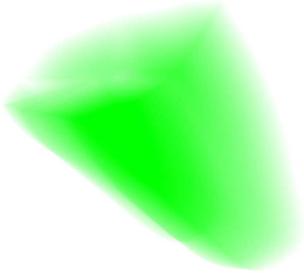

Figure 6. Delta image generated by VF-Ray.

et al., VF-Ray obtained comparable performance, spending only from $1/3$ to $1/6$ of the memory. When compared to the memory-aware Pina *et al* algorithms, VF-Ray obtained consistent and significant gains in memory usage and in execution time. When compared to another volume rendering paradigm, cell projection, VF-Ray also reduced significantly the memory consumption and the execution time.

The reductions in memory usage makes VF-Ray suitable for using double precision in the parameters to calculate the intersection between a ray and a face. Using double precision avoids the artifacts that appear when float precision is used. We conclude that VF-Ray is an efficient ray-casting algorithm that allows the in-core rendering of big datasets. As future work, we intend to optimize VF-Ray by keeping face data for neighboring visible faces, and to parallelize VF-Ray in order to run on a cluster of PCs

8. Acknowledgments

We acknowledge the grant of the first and second authors provided by Brazilian agencies CAPES and CNPq.

References

[1] F. Bernardon, C. Pagot, J. Comba, and C. Silva. Gpu-based tiled ray casting using depth peeling. *Journal of Graphics Tools*, To appear in 2007.

[2] P. Bunyk, A. Kaufman, and C. Silva. Simple, fast, and robust ray casting of irregular grids. *Advances in Volume Visualization, ACM SIGGRAPH*, 1(24), July 1998.

[3] R. Espinha and W. Celes. High-quality hardware-based ray-casting volume rendering using partial pre-integration. In *SIBGRAPI '05: Proc. of the XVIII Brazilian Symposium on Computer Graphics and Image Processing*, October 2005.

[4] R. Farias, J. Mitchell, and C. Silva. Zsweep: An efficient and exact projection algorithm for unstructured volume rendering. In *2000 Volume Visualization Symposium*, pages 91 – 99, October 2000.

[5] M. P. Garrity. Raytracing irregular volume data. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization*, pages 35–40. ACM Press, 1990.

[6] S. Lakare and A. Kaufman. Light weight space leaping using ray coherence. In *Proc. IEEE Visualization 2004*, pages 19–26, 2004.

[7] R. Marroquim, A. Maximo, R. Farias, and C. Esperanca. Gpu-based cell projection for interactive volume rendering. In *SIBGRAPI '06: Proc. of the Brazilian Symposium on Computer Graphics and Image Processing*, October 2006.

[8] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[9] M. Meibner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of popular volume rendering algorithms. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 81–90, 2000.

[10] A. Neubauer, L. Mroz, H. Hauser, and R. Wegenkittl. Cell-based first-hit ray casting. In *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002*, pages 77–ff, 2002.

[11] A. Pina, C. Bentes, and R. 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.

[12] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1176–1185, 2005.

[13] P. Shirley and A. A. Tuchman. Polygonal approximation to direct scalar volume rendering. In *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics*, volume 24(5), pages 63–70, 1990.

[14] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. In *Eurographics '01 Proceedings*, pages 153–164, 2001.

[15] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings of the 14th IEEE conference on Visualization '03*, pages 333–340, 2003.

[16] M. Weiler, P. Mallón, M. Kraus, and T. Ertl. Texture-encoded tetrahedral strips. In *2004 IEEE Symposium on Volume Visualization and Graphics (VolVis 2004)*, pages 71–78, October 2004.