Memory Efficient and Robust Software Implementation of the Raycast Algorithm

Aline A. de Pina
COPPE Sistemas/UFRJ,
aline@lcg.ufrj.br

Cristiana Bentes
DESC/UERJ,
cris@eng.uerj.br

Ricardo Farias
COPPE Sistemas/UFRJ,
rfarias@cos.ufrj.br

ABSTRACT

In this paper we propose two novel software implementations of the ray-casting volume rendering algorithm for irregular grids, called ME-Raycast (Memory Efficient Ray-casting) and EME-Raycast (Enhanced Memory Efficient Ray-Casting). Our algorithms improve previous work by Bunyk *et al* [1] in terms of complete handling of degenerate cases, memory consumption, and type of cell allowed in the grid (tetrahedral and/or hexahedral). The use of a more compact and non-redundant data structure, allowed us to achieve higher memory efficiency. Our results show consistent and significant gains in the memory usage of ME-Raycast and EME-Raycast when compared to Bunyk *et al* implementation. Furthermore, our results also show that handling of degenerate cases generates accurate images, correctly rendering all the pixels in the image, while Bunyk *et al* implementation fails in rendering up to 38 pixels in the final image. When we compare our algorithms to other robust rendering algorithm, like ZSweep [2], we have considerable performance gains and competitive memory consumption. We conclude that ME-Raycast and EME-Raycast are efficient methods for ray-casting that allow in-core rendering of large datasets with no image errors.

Keywords: Volume rendering, Ray-casting.

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, and ultrasound. A big advantage of direct volume rendering is to allow the investigation of the interior of the data volume, because the objects are considered as composed of a semi-transparent material.

Volumetric data used in volume rendering is usually represented in the form of a regular or irregular grid. Regular grids are built with a rigid topological framework, and can be represented in an implicit form. Irregular grids, on the other hand, have the advantage of generality since they can conform to nearly any desired geometry, and thus, they are useful to represent complex geometries in a compact way.

Although several algorithms and methods have been proposed to efficiently render irregular grids, the most popular one is the **ray-casting** method. In this method, rays are casted from the viewpoint through every pixel of the image what determines which cells of the volume each ray intersects. Every pair of intersections is used to compute a contribution for the pixel color and opacity. The ray stops when it reaches full opacity or when it leaves the volume.

There are many different implementations of the ray-casting algorithm, [6, 5, 7, 8]. Only a few software solutions, however, deal with irregular grids. Garrity [3] proposed an efficient method for ray-casting irregular grids using the connectivity of cells. In his method, as the ray intersects one cell, it must exit through one of its faces. At this point it is only necessary to check intersections of the ray with the cell's faces. Therefore, Garrity used the connectivity of the data to move from cell to cell of the grid, in order to reduce the cost of identifying the cells which the ray intersects. This scheme leads to a quadratic cost on the number of cells. Later, Bunyk *et al* [1] improved Garrity's work by determining for each pixel an ordered list of intersections on external visible faces. This allows them to efficiently enumerate which boundary face intersects a given ray, and the correct order of the entry points for the ray. The rendering process follows Garrity's method, but when a ray exits the grid, the algorithm can easily determine in which cell the ray will re-enter the grid. This approach becomes simpler and more efficient than Garrity's propose, however it keeps some large auxiliary data structures.

The memory consumption of Bunyk *et al* approach is very high. This can have some implications in the algorithm efficiency when the computer does not have enough main memory. In addition, the amount of memory used by the ray-casting algorithm could complicate its implementation in the graphics hardware. Nowadays, this be-

 ISBN 978-80-86943-98-5

comes a huge obstacle for achieving real-time performance in rendering.

Besides the memory consumption problem, Bunyk *et al* approach has other shortcomings. First, there are some degenerate cases that cannot be handled by their algorithm. And second, it deals only with tetrahedral grids. In this work, we propose two novel ray-casting algorithms based on Bunyk *et al* approach, but improving it in different ways. Our goal is to develop memory efficient ray-casting algorithms that provide accurate results. Our approaches: (i) completely handle degenerate cases; (ii) use different data structures that are much smaller than the ones used in Bunyk *et al* approach; and (iii) deal with both tetrahedral and/or hexahedral grids.

Our algorithms, called ME-Raycast (Memory Efficient Ray-casting) and EME-Raycast (Enhanced Memory Efficient Ray-Casting), presented consistent and significant gains in memory usage over Bunyk *et al* approach. Our gains were not only in memory usage, but also in the correctness of the final image. Bunyk *et al* approach did not handle all possible degenerated cases, so it generated some incorrect pixels in the image.

We also compared our algorithms to other robust direct volume rendering algorithms based on cell projection paradigm, ZSweep. Our algorithms outperform ZSweep for all datasets. In terms of memory usage, for smaller images resolutions, ME-Ray spends more memory than ZSweep. EME-Ray, otherwise, spends less memory than ZSweep for most of the cases.

The remainder of this paper is organized as follows. In the next section we relate our work to others in the field of volume rendering of irregular grids. Section 3 describes our ray-casting algorithms and the improvements we made on Bunyk's approach, and shows how our algorithms handle the degenerate cases. In section 4 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

There are mainly two categories of algorithms for direct volume rendering on irregular grids: ray-casting and projection.

Ray-casting algorithms are usually called image-space methods, since in its outer loop, it iterates over all the pixels of the output image. In the work by Garrity [3], as mentioned before, for each ray, exterior faces are tested to find the first intersection point. After that, the cells are traversed using the connectivity relation between them. This work was further improved by Bunyk *et al* [1], by computing for each pixel a list of intersections on external visible faces, and easily determining the correct order of the entry points for the ray. These two are all-software approaches, which means that they do not require any graphics hardware. Our work is also an all-software implementation, but provides improvements over Bunyk *et al* work. Weiler *et al* [12], on the other hand, implemented ray-casting using the graphics hardware. 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, and is based on GPU programming.

Another class of rendering algorithms is the one that performs the render based on the sweeping paradigm to lower the cost of the ray-casting. The first work in this class was developed by Giertsen [4]. In his work, a plane sweeps the dataset in the up direction, or in the direction of Y axis, intersecting with cells. For every line of pixels of the image, all intersections of the sweeping plane with the cells of the grid is approximated by a regular 2D grid, and a bidimensional raycast is performed. One weakness of this method is the approximation imposed in the accommodation of the 2D grid, result of the intersection of the plane sweep with the data cells, onto the regular grid. Later, the work by Silva *et al* [10] improved Giertsen work. The Lazy Sweep algorithm avoids the approximation mentioned above.

Projection algorithms, on the other hand, reconstruct the image from the object space to the image space. The projection requires that the cells are first sorted in visibility ordering and then composed to generate their color and opacity in the final image. The first algorithm to be fully implemented to use projection was the ZSweep by Farias *et al.* [2]. The algorithm was implemented using only the CPU, what provided flexibility and easy parallelization. The ZSweep is a simple and efficient face projection rendering algorithm. ZSweep sweeps the dataset vertices, in depth order, with a plane perpendicular to the viewing direction. When the sweep plane hits a vertex, ZSweep project the faces incident on that vertex. To achieve memory efficiency, they used a mechanism called early ray composition. We used ZSweep algorithm as a baseline for our performance evaluation, in order to compare the speed and memory usage of our ray-casting algorithms over a projective one. The great advantage of projective methods is that they are efficiently implemented in programmable graphics hardware. Several cell projection algorithms were implemented using hardware graphics (e.g., [13], [9], [11]).

3 OUR APPROACHES

The main goal of ME-Raycast and EME-Raycast algorithms is to combine correctness of the results with effi-

ciency in memory usage, without degrading the execution time.

For both algorithms, the traversal for each pixel starts in the same way proposed in Bunyk *et al* implementation. We project the visible faces on the screen and keep for each pixel the list of intersection points which enters the volume. Nevertheless, for the internal grid adjacency representation, ME-Raycast and EME-Raycast use completely different data structures. In fact, EME-Raycast was developed as an optimization of ME-Raycast in terms of memory usage. EME-Raycast uses simpler data structures than ME-Raycast. Our algorithms also include an identical and efficient method to deal with the degenerate cases that can occur during the ray traversal process, described in section 3.3.

3.1 ME-Raycast Algorithm

Before explaining the ME-Raycast algorithm itself, we describe its basic data structures. These data structures are also used by EME-Raycast, except for the most memory expensive array and some auxiliary structures, which are eliminated in EME-Raycast to lower memory consumption.

Basic Structures ME-Raycast keeps three basic structures: the vertex array (`Points_VEC`), the cell array (`Cells_VEC`) and the face array (`Faces_VEC`). There are also some auxiliary structures: the `Use_Set` of a vertex v is a list of all cells incident on v (see Farias *et al* [2]); the `Neighbor_Array` of a cell c is an array of indices of all neighboring cells of c; and the `Triangular_Faces` of a cell c is an array of indices of the triangular faces that bound c (in hexahedral cells, the faces need to be broken in two triangular faces).

The `Use_Set` array substitutes the `referredBy` list used in Bunyk's implementation. The `Use_Set` for each vertex, is a list of all cells incident on the vertex, in contrast with the `referredBy` list, which is a list of faces incident on the vertex. The `Use_Set` can be created in the preprocessing phase, in $O(c)$, where c is the number of cells. We allocate an array of integers (`int_array`), of the size of the number of vertices. For each cell, we loop through each of its vertices, and increment the element of `int_array` indexed by the number of the vertex, and a global counter. At the end, we know how many cells are incident on each vertex and the global total of incident cells on every vertex. Then, we allocate another array (`Use_Set`) using the global counter. We repeat the loop on the array of cells and fill in the `Use_Set` of each vertex.

Another step in the preprocessing phase is to find for each cell C_i its *face-neighbor* cells, which are the cells that share a face with C_i, and create the `Neighbor_Array` for C_i. We determine all the *face-neighbor* cells by scanning the `Use_Set` of the vertices of the cell. During this scanning, we create for each cell a list with the indices for its *face-neighbor* cells. We save a great amount of memory by keeping such lists on the cell structure instead of on the face structure (as done in Bunyk's method), since the number of faces is always greater than the number of cells. This information speeds up the process of stepping through the grid during ray-casting.

The `Faces_VEC` array is created on demand during the raycast process, as the faces are intersected by the rays. Only intersected faces are inserted. As a face is inserted, all its related parameters are computed. The number of faces in the array will depend on the image resolution and on the size of the dataset. For example, for a small resolution image and a large scale dataset, lots of faces will never be intersected by any ray and consequently will not be created by the process. Bunyk's method, on the other hand, inserts all faces in the preprocessing phase and compute their parameters at the beginning of the rendering. Processing time is saved, but with the cost of great memory overhead.

Algorithm The ME-Raycast algorithm can be divided into two phases: the preprocessing phase, and the core engine. Just like Bunyk's implementation, the preprocessing is performed while the dataset is read. In the preprocessing phase, the following steps are performed:

1. Read and store the vertices and cells of the dataset, creating `Points_VEC` and `Cells_VEC`.
2. Generate the `Use_Set` list for each vertex.
3. Determine for each cell its *face-neighbor* list.

To identify external faces of a cell, we store its own index, indicating that there is no *face-neighbor* cell sharing this face. We also create a list with all external faces. This list keeps, for each face, the index to the cell and to the relative face in the cell. The core engine of ME-Raycast algorithm performs the rendering process. For each point of view, ME-Raycast execute the following code.

```
Project external faces
   creating Ext_Faces;
For each pixel
  While( Ext_Faces not empty){
    Repeat {
      Find next intersection by
      checking other cell's faces;
      If (no intersection)
         check degenerate case;
      Accumulate colors/opacity;
```

 ISBN 978-80-86943-98-5

```
    } While (next intersected
             face is internal)
}
```

The visible external faces (the ones whose normals have angles greater than 90^o with the viewing direction) are projected on the screen, generating for each pixel a list of intersections with the external faces. These intersections will be used to start the raycast for each pixel.

To visualize the dataset from a different point of view, the list of faces is reused, saving processing time. Only the faces parameters must be recomputed for all faces. Also, our implementation allows both parallel and perspective projections.

3.2 EME-Raycast Algorithm

EME-Raycast and ME-Raycast have similar algorithms, but in EME-Raycast we have removed some data structures used in ME-Raycast.

The basic structures used in the EME-Ray are only two: the array of vertices (`Points_VEC`), and the array of cells (`Cells_VEC`). We have removed the array of faces (`Faces_VEC`), since it was one of the most memory expensive structures in ME-Raycast. Without the array of faces, we can save about $5*2*f$ bytes of memory, where f is the total number of triangular faces in the data set. Therefore, in a dataset with about 1 million faces, we are save about 52 Mb of memory.

From the cell structure, we removed the array `Triangular_Faces`. As tetrahedral cell uses 36 bytes, saving 16 bytes, and a hexahedral cell uses 60 bytes, saving 48 bytes of memory. In all we save about $(16*t+48*h)$ bytes of memory, where t is the total number of tetrahedral cells and h is the total number of hexahedral cells.

The vertex structure is identical to the one used in ME-Ray. However, the data structures removals are responsible for increasing the execution time. As we do not store the faces anymore, we need to recalculate the parameters for verification of ray intersection every time that a new face is checked.

3.3 Handling Degeneracies

The intersection between a ray and a face of a cell is the result of the algebraic calculation of the intersection between a line and a plane, see [1]. For tetrahedral grids, it is the intersection between the line defined by the ray path and the plane defined by the three vertices of a triangular face. For hexahedral grids, where each quadrangular face is defined by four vertices, the intersection is found by splitting the quadrangular face into two triangular faces, and performing the same calculation mentioned above, for each face. This way, we do not need to worry about the four points being coplanar to define a plane.

Bunyk *et al* use the Point-Within-Triangle algorithm to determine the ray intersections and to look for the next cell the ray will intersect. In their algorithm, however, degenerated situations may arise when the ray hits a vertex or an edge. In Figure 1, we exemplify in 2D the case where the ray hits a vertex. The blue cell corresponds to the first cell that the ray intersects, called the current cell. Bunyk *et al* approach would check only the cells neighboring the current cell faces, i.e., faces of the cells A and E. However, the ray does not intersect with neither A nor E faces. In this case, the final color of the pixel will be wrong, since the composition process will be interrupted.

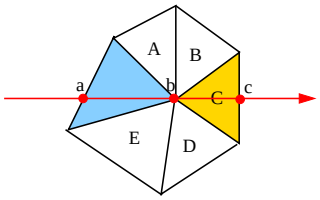

Figure 1: 2D example where the ray hits a vertex and Bunyk approach does not find c

To avoid this type of error, we propose a different kind of verification to look for the next cell intersected by the ray. The idea is to allow the continuation of the ray traversal, by looking for the next cell scanning the `Use_Set` of each vertex which determine the current cell. In the example in Figure 1, this scanning will return cells A, B, C, D and E. Therefore, this scheme asserts that another intersection will be found in a face of cell C, guaranteeing that the ray traversal will continue and the image will be correctly generated.

When the ray hits an edge, the problem can be solved by the same procedure explained above. In Figure 2, we show an 3D example for this case. In this example, the blue cell of (a) corresponds to the first cell that the ray intersects. The next intersection is in the edge V_0V_1 in the point b. Bunyk *et al* approach would look for the next intersection in all the faces adjacent to V_0V_1 edge. These faces are shown in (b). As we can observe in the figure, it is not possible to find the next intersection in the adjacent faces of V_0V_1. Our approach, on the other hand, uses the `Use_Set` of the vertices of the blue cell to find the next intersection. Using the `Use_Set` of V_0 and V_1, we find the yellow cell of (a). In the yellow cell, we find the next face the ray intersects, determining c.

4 EXPERIMENTAL RESULTS

In this section we evaluate the performance and memory usage of ME-Raycast (**ME-Ray**) and EME-Raycast

 ISBN 978-80-86943-98-5

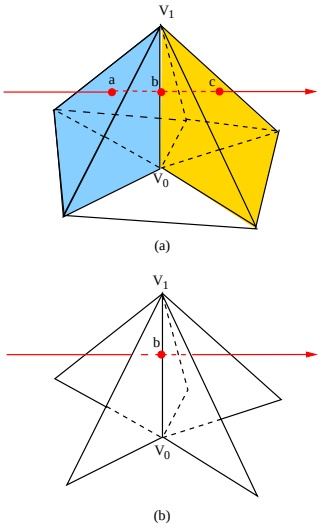

Figure 2: 3D example where the ray hits an edge and Bunyk approach does not find *c*

(**EME-Ray**). Our evaluation uses two different baselines for the comparisons. The baselines are two direct volume rendering algorithms for irregular grids: the Bunyk *et al* implementation of ray-casting (**BUNYK**); and the ZSweep cell projection algorithm (**ZSweep**). The ideas behind these comparisons are to: (1) Measure the improvements over Bunyk work in terms of memory usage and correctness of the final image. (2) Put our results in perspective, with respect to other robust rendering algorithm (that generates correct final images). Following, we briefly describe the two baselines, show the datasets used in our experiments, and, then, describe our performance analysis.

4.1 Baselines

BUNYK: Bunyk *et al.* implementation initially projects the external visible faces on the screen, creating for each pixel, a list of the intersections generated by these projections. The process starts by projecting all faces whose normal make an angle greater then 90^o with the viewing direction. After projecting every visible face, the algorithm knows, for each pixel, which face through which the ray enters the volume. Since the algorithm computes all cell's neighbors in preprocessing, it is computed, in constant time, the next face the ray is going to intersect. For every two consective intersections, opacity and color integrations are computed. Once there is no more entry point for the pixel, the ray has left the volume, and the process is finshed for the current pixel. It is important to notice that, the list of faces created to carry on this method is responsible for about half the memory usage of this implementation.

Table 1: Datasets used in our experiments.

Dataset Information				
Datasets	Vertices	Faces	Boundary	Cells
Blunt Fin	40.960	381.548	13.516	187.395
Comb. Chamber	47.025	437.888	15.616	215.040
Oxygen Post	109.744	1.040.588	27.676	513.375
SPX	149.224	1.677.888	44.160	827.904
Delta Wing	211.680	2.032.084	41.468	1.005.675
Hexa	2.684	6.432	1.344	1.920

ZSweep: The ZSweep algorithm is a direct volume rendering algorithm based on the sweeping paradigm, and built over the success of prior sweep approaches [10]. 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, using a heap sort, 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. When a face is projected onto a given pixel, the result is equivalent to the intersection of the ray emanating from this same pixel and the face being projected. ZSweep stores its z-value, and other auxiliary information, in sorted order in a list of intersections for the given pixel, called pixel list. To achieve memory efficiency, ZSweep uses a mechanism called early composition. The composition of the intersections in a pixel list is performed as the *target-Z* is reached. The *target-Z* represents the maximum z coordinate among the vertices adjacent to the first vertex encountered by the sweeping plane. When the plane reaches a target z, the next target z will be again the maximum z coordinate among the vertices adjacent to the current reached target, and the process continues.

4.2 Workload

Our experiments were conducted in a Pentium 4, 2.80GHz with 1GB of memory, running Linux Fedora Core 2. We have used five different tetrahedral datasets: Blunt Fin, Combustion Chamber, Oxygen Post, SPX and Delta Wing, and also one small hexahedral dataset, called Hexa, used only to show our handling of hexahedral grids. The number of vertices, faces, boundary faces and cells for each dataset are listed in Table 1. We also varied the image sizes, from 128×128 to 1024×1024 pixels.

4.3 ME-Raycast and EME-Raycast Performance

In this section we evaluate ME-Raycast and EME-Raycast algorithms, compared to ZSweep and BUNYK results. In terms of the number of pixels rendered, ME-Ray, EME-Ray and ZSweep rendered all the pixels,

generating a correct image. However, since BUNYK does not handle all possible degenerate cases, it fails in rendering the amount of pixels shown in Table 2. As we can observe in this table, for Blunt Fin, Oxygen, Delta and SPX, BUNYK generates a great amount of flaws in the 512×512 and 1024×1024 images. Up to 38 pixels were not rendered correctly.

Table 2: Pixels not rendered by BUNYK

Bad pixels - BUNYK					
Image Size	Blunt Fin	Combustion	Oxygen	Delta	SPX
128^2	2	-	-	1	-
256^2	3	1	4	2	3
512^2	10	-	11	7	5
1024^2	31	2	38	17	18

Tables 3, 4, 5, 6 and 7 show the results of time and the amount of memory consumed for Blunt Fin, Combustion Chamber, Oxygen Post and SPX datasets, respectively, rendered by ME-Ray and EME-Ray when compared to BUNYK and ZSweep execution, for four different image resolutions, 128×128, 256×256, 512×512, and 1024×1024. The results for the Hexa dataset are not considered because it is a very small dataset, and BUNYK algorithm cannot handle heaxahedral datasets. The percentages presented in these tables correspond to the ratio of our algorithm (ME-Ray or EME-Ray) result over the baseline (BUNYK or ZSweep). In other words, we consider BUNYK or ZSweep results as 100% and are presenting how much we increase or decrease this baseline.

Table 3 presents the results for the Blunt Fin dataset. Comparing ME-Ray and EME-Ray with BUNYK, we observe that they use considerably less memory than BUNYK. ME-Ray uses, for a 1024×1024 image, almost the same memory BUNYK uses for a 128×128 image. EME-Ray uses 3.5 times less memory than BUNYK for a 512×512 image and 2.5 times less memory for a 1024×1024 image. These significant reductions in memory usage comes with an increase in the execution time. The increase, however, is only about 26% for ME-Ray for a 1024×1024 image. When compared to ZSweep, we observe that EME-Ray outperforms ZSweep in terms of render time and memory usage for the three larger image precisions.

Table 4 shows the results for Combustion Chamber dataset. ME-Ray and EME-Ray also consume less memory than BUNYK. For a 512×512 image, EME-Ray spends 3.6 times less memory than BUNYK and, for a 1024×1024 image EME-Ray spends 2.7 times less memory. In terms of execution time, BUNYK outperforms ME-Ray, but for a 1024×1024, ME-Ray is only 6% slower. When compared to ZSweep, ME-Ray spends more memory, but is faster, and EME-Ray is faster and uses less memory for larger images.

Table 3: Blunt Fin Data Results

Image		Time		Memory	
		BUNYK	ZSweep	BUNYK	ZSweep
128^2	ME-Ray	145%	32%	52%	229%
	EME-Ray	225%	50%	24%	106%
256^2	ME-Ray	133%	30%	61%	184%
	EME-Ray	281%	64%	25%	75%
512^2	ME-Ray	138%	32%	69%	96%
	EME-Ray	335%	78%	28%	39%
1024^2	ME-Ray	126%	29%	75%	40%
	EME-Ray	304%	70%	39%	20%

Table 4: Combustion Chamber Results

Image		Time		Memory	
		BUNYK	ZSweep	BUNYK	ZSweep
128^2	ME-Ray	184%	57%	73%	351%
	EME-Ray	166%	51%	24%	115%
256^2	ME-Ray	141%	43%	74%	287%
	EME-Ray	215%	65%	25%	96%
512^2	ME-Ray	153%	57%	75%	169%
	EME-Ray	232%	87%	28%	63%
1024^2	ME-Ray	106%	38%	76%	76%
	EME-Ray	224%	80%	37%	37%

Table 5 shows the results for Liquid Oxygen Post dataset. ME-Ray and EME-Ray use considerably less memory than BUNYK. For larger images, BUNYK uses about 3 times more memory than our algorithms. As the image size grows, however, BUNYK becomes much more faster than EME-Ray. Compared to ZSweep, ME-Ray is about 2.6 times faster and uses 1.5 times less memory for the largest image precision. For a 512×512 image, EME-Ray is about 1.5 times faster and uses 1.6 times less memory than ZSweep.

Table 5: Liquid Oxygen Post Results

Image		Time		Memory	
		BUNYK	ZSweep	BUNYK	ZSweep
128^2	ME-Ray	162%	24%	44%	216%
	EME-Ray	200%	29%	24%	118%
256^2	ME-Ray	137%	39%	50%	205%
	EME-Ray	268%	76%	24%	100%
512^2	ME-Ray	142%	41%	57%	142%
	EME-Ray	276%	79%	26%	64%
1024^2	ME-Ray	136%	38%	66%	70%
	EME-Ray	289%	82%	30%	32%

Table 6 shows the results for the largest dataset, Delta Wing, that has more than 1 million cells. ME-Ray method is about 1.5 times slower than BUNYK, but uses 1.6 times less memory and EME-Ray uses less than 30% of the memory used by BUNYK. Compared to ZSweep, ME-Ray is faster for all images sizes, but consumes more memory. For small images, even EME-Ray consumes more memory than ZSweep. This is due to the indices we keep for the neighboring cells for each cell. Nevertheless, EME-Ray performs significantly better than ZSweep.

Table 6: Delta Wing Results

Image		Time		Memory	
		BUNYK	ZSweep	BUNYK	ZSweep
128^2	ME-Ray	141%	15%	38%	200%
	EME-Ray	150%	16%	23%	124%
256^2	ME-Ray	162%	26%	44%	216%
	EME-Ray	203%	32%	24%	118%
512^2	ME-Ray	139%	32%	51%	209%
	EME-Ray	244%	56%	24%	100%
1024^2	ME-Ray	148%	33%	61%	152%
	EME-Ray	271%	61%	27%	68%

Table 7 shows the results for the SPX dataset. Although ME-Ray is slower than BUNYK, it uses less memory. EME-Ray uses even less memory. For example, to create a 512×512 image, EME-Ray uses about 1/4 of the memory necessary for BUNYK to create a 256×256 image. Compared to ZSweep, ME-Ray and EME-Ray are faster than ZSweep. In terms of memory usage, ME-Ray uses more memory than ZSweep and EME-Ray uses 1.2 times less memory than ZSweep for a 1024×1024 image.

Table 7: SPX Results

Image		Time		Memory	
		BUNYK	ZSweep	BUNYK	ZSweep
128^2	ME-Ray	120%	20%	51%	273%
	EME-Ray	197%	33%	23%	125%
256^2	ME-Ray	154%	41%	68%	349%
	EME-Ray	178%	48%	24%	121%
512^2	ME-Ray	136%	46%	72%	320%
	EME-Ray	192%	66%	24%	109%
1024^2	ME-Ray	116%	41%	74%	222%
	EME-Ray	222%	79%	27%	83%

5 DISCUSSION

ME-Ray and EME-Ray had obtained consistent and significant gains in memory usage over BUNYK. In terms of the image resolution, we can observe that the gains of ME-Ray and EME-Ray over BUNYK are bigger for smaller image sizes. This occurs because BUNYK creates at once an array with all the faces in the dataset, and this does not depend on the image size. While ME-Ray creates the faces as they are intersected by the rays. Otherwise, in terms of the dataset size, as we expected, when the dataset increases, the reductions in memory usage of ME-Ray also increases. This result confirms that our data structures are set to handle big datasets.

Furthermore, the reductions in memory requirements we obtained with our data structures, allowed us to use double precision in the parameters to calculate the intersection between a ray and a face. BUNYK uses float for these parameters, consequently causing some precision errors. We have made some experiments with BUNYK algorithm increasing the parameters precision to double and obtained better images. On the other hand, the memory requirements increased about 12F bytes (where F is the number of faces in the dataset).

The increase in the execution time, when compared to BUNYK, comes from the fact that we have to scan through out the `Use_Set` of the vertices to perform the ray traversal. Since the `Use_Set` keeps the indices for the cells incident on each vertex, its likely to occur double intersection computation for internal faces. On the other hand, BUNYK keeps all faces incident on the vertices which makes it faster to compute such intersections, while spending more memory. In our experiments, however, we are only comparing executions where the whole dataset fits in main memory for both methods. As the memory usage increases, the rendering will need to use of the virtual memory mechanisms of the operating system, which would have great influence on the overall 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. BUNYK does not handle all possible degenerated cases. For 1024×1024 images, in all the datasets, BUNYK algorithm generates some flaws in the image. For Delta Wing, for example, BUNYK fails in rendering 17 pixels. This causes some black spots in the image as we can observe in the 512×512 image of Figure 3.

When compared to ZSweep, ME-Ray outperforms ZSweep, in execution time, significantly for all the datasets and all the image resolutions. Although it is not an intuitive result, it is explained by the fact that, in ZSweep, while the *target-Z* is not reached, the pixel list increases. The bigger the list is, the more expensive is the insertion, since it is ordered. Depending on the dataset, the *target-Z* could be a bad parameter to start the composition. ME-Ray, on the other hand, composes the pixels on-the-fly as each intersection is found.

Another important difference in the performance of ME-Ray and EME-Ray, when compared to ZSweep, is the dataset structure. More "irregular" datasets with holes and much more external visible faces would benefit ZSweep, since ME-Ray and EME-Ray would have to compute more external faces intersection. This, however, is not the case for our workload, except for SPX, that provides the smaller performance difference between ME-Ray and EME-Ray compared to ZSweep.

In terms of memory requirements, ME-Ray spends more memory than ZSweep, except for 1024×1024 image resolutions. EME-Ray, on the other side, spends less memory than ZSweep for most of the datasets and image resolutions. ZSweep increases linearly the memory requirement with the increase in the image size, since as the

image size increases, each face projected will insert intersection units into more pixel lists. ME-Ray also increases linearly the memory requirements with the increase in the image size. This increase is due to the increase in the size of `Faces_VEC`, since more faces are intersected. EME-Ray, otherwise, does not have `Faces_VEC` data structure, so the memory usage maintains almost constant, even when the image size increases.

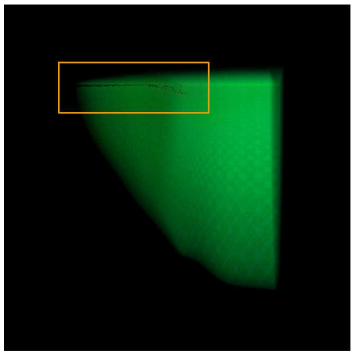

Figure 3: Delta (512x512) generated by BUNYK with bad rendered pixels highlighted by the orange box.

6 CONCLUSIONS

We proposed two novel ray-casting algorithms, ME-Raycast (Memory Efficient Ray-casting) and EME-Raycast (Enhanced Memory Efficient Ray-Casting). Our algorithms improve previous work by Bunyk *et al* in terms of memory consumption, type of cell allowed in the grid (tetrahedral and hexahedral), and complete handling of degenerate cases. Our goal in improving Bunyk *et al* work was to provide a software implementation of ray-casting that is memory efficient without performance degradation, and robust, i.e., generates correct images

Our experimental results showed that ME-Raycast and EME-Raycast are comparable in performance to Bunyk *et al* in most of the cases, but had obtained consistent and significant gains in memory usage over their approach. These results confirm that our data structures store only essential information. When compared to other accurate rendering algorithm, ZSweep, ME-Raycast and EME-Raycast obtained considerable performance gains, and competitive memory consumption. EME-Raycast by itself spends less memory than ZSweep for most of the datasets and image resolutions.

Our results also showed that ME-Raycast and EME-Raycast complete handling of degenerate cases generates accurate images, rendering correctly all the pixels of the image. Nevertheless, Bunyk *et al* work failed on rendering some pixels in the final image, generating incomplete results. Besides the memory and performance results, we showed that we can deal with grids represented by tetrahedra, hexahedra or both. As far as we know, they are the first ray-casting implementations which handle, at the same time, both types of irregular grids.

We conclude that ME-Raycast and EME-Raycast are efficient algorithms for ray-casting that allows the in-core rendering of big datasets, avoiding paging operations on disk. The low memory usage of our algorithms also makes them suitable for hardware-based implementations, in order to achieve real-time rendering. As future work, we consider the study of out-of-core versions of the codes that run on clusters of PCs.

REFERENCES

[1] 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.

[2] 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.

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

[4] C. Giertsen. Volume visualization of sparse irregular meshes. *IEEE Comput. Graph. Appl.*, 12(2):40–48, 1992.

[5] L. Hong and A. Kaufman. Accelerated ray-casting for curvilinear volumes. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 247–253, 1998.

[6] W. M. Hsu. Segmented ray casting for data parallel volume rendering. In *PRS '93: Proceedings of the 1993 symposium on Parallel rendering*, pages 7–14, 1993.

[7] K. Koyamada. Fast ray-casting for irregular volumes. In *ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing*, pages 557–572, 2000.

[8] 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.

[9] S. Rottger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 109–116, 2000.

[10] C. T. Silva and J. S. B. Mitchell. The lazy sweep ray casting algorithm for rendering irregular grids. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):142–157, 1997.

[11] C. T. Silva, J. S. B. Mitchell, and P. L. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 87–94, 1998.

[12] M. Weiler, M. Kraus, and T. Ertl. Hardware-based view-independent cell projection. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 13–22, 2002.

[13] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral projection using vertex shaders. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 7–12, 2002.