Accelerated Ray-Casting for Curvilinear Volumes

Lichan Hong
Software Production Research Department
Bell Laboratories, Lucent Technologies
Room 2F-321, 263 Shuman Blvd
Naperville, IL 60566-7050
lhong@research.bell-labs.com

Arie Kaufman
Center for Visual Computing (CVC)
and Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
ari@cs.sunysb.edu

Abstract

We present an efficient and robust ray-casting algorithm for directly rendering a curvilinear volume of arbitrarily-shaped cells. We designed the algorithm to alleviate the consumption of CPU power and memory space. By incorporating the essence of the projection paradigm into the ray-casting process, we have successfully accelerated the ray traversal through the grid and data interpolations at sample points. Our algorithm also overcomes the conventional limitation requiring the cells to be convex. Application of this algorithm to several commonly-used curvilinear data sets has produced a favorable performance when compared with recently reported algorithms.

Keywords: Volume Visualization, Volume Rendering, Irregular Grid, Curvilinear Grid, Ray-Casting, Parallel Rendering, Dynamic Simulation

1 Introduction

A major barrier preventing the widespread usage of volume rendering technology is its substantial requirements for computational power and memory space. In the past few years, numerous algorithms have been investigated to improve the rendering performance of rectilinear volumes. However, volume rendering of non-rectilinear (i.e., curvilinear and unstructured) grids [15] is still relatively under-explored.

A curvilinear grid can be considered as the result of a rectilinear grid of cubic voxels subjected to non-linear transformations, so as to fill or warp around an object of complex shape while preserving the grid topology. The curvilinear grid has the same implicit connectivity as the rectilinear grid, yet unlike the rectilinear grid, the 3D locations of its grid vertices must be explicitly defined. As a result, each quadrilateral cell-face is not necessarily planar and each six-sided cell is not necessarily convex. Curvilinear grids are commonly utilized in a variety of applications such as scientific computing and computer-based modeling. For example, in a Computational Fluid Dynamics (CFD) simulation, a curvilinear grid can be employed to efficiently model the surrounding area of an aerodynamic object [14]. While in the case of dynamic simulation [21, 1], a deformable object may initially be represented as a rectilinear volume, the volumetric object dynamically changes its underlying grid structure and soon becomes curvilinear, due to the influence of internal and external factors (e.g., when colliding with another object).

1.1 Related Work

The irregularity of the curvilinear grid, in comparison with the rectilinear grid, imposes a much higher complexity on the rendering process. The simplest way of visualizing a curvilinear volume is to resample it into a rectilinear grid and subsequently render the new volume with algorithms for rectilinear data [8]. Since the curvilinear grid could consist of cells of drastically different sizes, a significantly large rectilinear volume has to be employed in order to preserve the details of the smallest curvilinear cells (which usually include data of ultimate importance). In addition, the resampling abandons the original grid structure and may introduce sampling errors [3, 16]. Alternatively, one may decompose each curvilinear cell into five tetrahedra and then further process the tetrahedral representation [4, 7, 11, 12, 18, 20]. As a result, the implicit cell connectivity in the original grid structure is abandoned and a significant amount of memory space has to be allocated to explicitly define the connectivity of the tetrahedral cells. Commonly, the extra storage required is several times the amount of space needed to store the original curvilinear volume.

Yet another approach is to perform the rendering directly on the curvilinear volume, without converting it into an intermediate grid. Generally speaking, there are two such techniques: projection and ray-casting. In Van Gelder and Wilhelms [5], each curvilinear cell was projected onto the image screen in a certain visibility order, with its contributions to the image pixels being calculated with graphics hardware. To obtain the visibility order, the cells were approximated as convex hexahedra and sorted using a linear-time topological sorting algorithm. To overcome the potential problem that the cell visibility order may be impossible to find [13, 19], Wilhelms et al. [17] treated each cell-face as a projection primitive and independently scan-converted the cell-faces in software. All the polygons contributing to a particular image pixel were then sorted by depth obtained from the scan-conversion and correctly composited to produce the pixel color.

Using ray-casting to directly render a curvilinear volume has also been studied [9, 14]. Typically, a ray is cast from the virtual camera through each image pixel into the curvilinear volume. All the cell-faces intersecting with the ray are found and data are interpolated for the ray/cell-face intersection points. The pixel color is then determined by accumulating the contributions of the intersecting cells along the ray. Both Uselton [14] and Ramamoorthy and Wilhelms [9] approximated each cell-face to be planar and adapted Garrity's ray-casting algorithm of tetrahedral grids [4] to the curvilinear domain. Ramamoorthy and Wilhelms [9] also described their efforts in accelerating the rendering at the expense of extra memory space.

1.2 Our Contribution

Our work in curvilinear volume rendering was mainly driven by our project on dynamic simulation of deformable objects [21]. The goal of this project is to simulate how deformable objects dynamically change their shapes under certain circumstances. The FEM simulation model includes volumetric objects whose grid structures are initially rectilinear but quickly deform and become curvilinear. Subsequently, the curvilinear grid is likely to deform into another curvilinear grid at each time step of the simulation. Due to the extraordinary consumption of computational power and memory space by the simulation, in conjunction with the desire to perform the volume visualization on the same platform, our objective has been to find a fast rendering algorithm which does not rely on an extensive precomputation or a considerable amount of extra storage. In addition, the algorithm would have to be robust enough to process both convex and concave cells. Furthermore, we expected that the algorithm could be readily parallelized on a multi-processor computer, and extensible in the future to handle multiple objects as well as produce realistic effects such as shadows and reflections. After thoroughly evaluating the features of existing techniques, we found that none of them was well suited for our application and decided to develop a new rendering algorithm of our own, which is presented in this paper.

Like the previous work by Uselton [14] and Ramamoorthy and Wilhelms [9], our algorithm is a ray-casting technique directly generating images from the curvilinear volume. Our major contribution is in incorporating the essence of the projection approach into the ray-casting to speed up the critical steps of ray traversal and data interpolations [3, 9]. The acceleration is achieved without incurring any precomputation nor requiring a vast amount of extra storage. Specifically, to find the first-entry and re-entry cell-faces for a ray, we employ a pixel bucket to depth-sort those exterior cell-faces intersecting with the ray. When the ray enters a cell, we project the candidate cell-faces onto the image screen and efficiently perform the ray/cell-face intersection tests to identify the cell-face from which the ray exits the cell. This also enables us to overcome the limitation of Garrity's algorithm [4] which requires the cell to be convex. As a by-product of the tests, the data at the ray/cell-face intersections are reconstructed and subsequently used in the accumulation of color and opacity.

The remainder of this paper is structured as follows: In Section 2, we describe our ray-casting algorithm, and outline how we accelerate the ray traversal as well as the data interpolations. We also present the parallelization of our ray-casting algorithm on a shared-memory, multi-processor architecture. Then, in Section 3, we apply our algorithm on several well-known curvilinear data sets and compare its performance with two recently reported algorithms [12, 20].

2 Acceleration Techniques for Ray-Casting

In the ray-casting, a ray is cast from each image pixel into the curvilinear volume, as illustrated in Fig. 1. To accumulate the color c and opacity o along the ray for a pixel (x, y), a function in the form of *CastOneRay* is used (see Fig. 2). In general, there are three major challenges in the ray-casting process [3, 9] as follows:

(1) Identifying the exterior cell-faces through which a ray first enters and possibly re-enters the grid after the ray exits from an exterior cell-face;

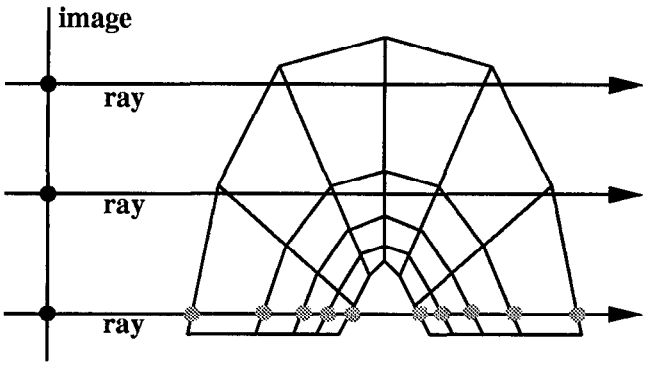

Figure 1: *Ray-casting a curvilinear volume.*

(2) Determining the cell-face through which a ray exits the current cell and enters the next cell;

(3) Reconstructing data values at the intersections between a ray and the cell-faces.

An *exterior* cell-face is a cell-face that belongs to the grid boundary and is exclusively owned by a cell, while an *interior* cell-face is the common wall shared by two adjacent cells. Since a great deal of rendering time is taken by these three operations, our goal is to accelerate them so as to improve the overall rendering speed.

CastOneRay(x, y, c, o)
{
  (1) cast a ray from pixel (x, y) and find the first cell-face intersected by the ray
  (2) while (the ray has not exited the grid *and* opacity o has not reached unity)
  {
    (2a) locate the exiting cell-face of the current cell
    (2b) reconstruct the scalar s and depth z at the intersection between the ray and the exiting cell-face
    (2c) accumulate c and o, based on the values of s and z
  }
  (3) if opacity o has not reached unity *and* the ray re-enters the grid, go to **(2a)**
}

Figure 2: *The function for casting a ray from pixel (x, y) to accumulate color c and opacity o.*

2.1 Determining Entry Cell-Faces

As shown in Fig. 1, to traverse along the ray, we need to find the exterior cell-face from which the ray first enters the volume. In addition, since the grid is likely to be concave, after the ray exits the volume from an exterior cell-face, it can potentially re-enter the grid multiple times (see Fig. 1).

To quickly locate the first-entry and re-entry exterior cell-faces, both Garrity [4] and Ramamoorthy and Wilhelms [9] suggested superimposing a rectilinear grid of certain resolution over the curvilinear grid. At a preprocessing step, each voxel of the embedded rectilinear grid is associated with a list of exterior cell-faces that are totally or partially within the voxel. During the rendering, when there is a need to find the first-entry cell-face or check whether the ray re-enters the volume, those rectilinear voxels intersecting with the ray are identified and their associated exterior cell-faces are tested against the ray.

Due to the fact that the grid structure changes over time in our application, it may be undesirable to use such an embedding scheme which was originally designed for a static grid. In addition to the extra storage, a new curvilinear grid at each time step of the simulation would require that the exterior cell-faces be re-distributed into the voxels of the rectilinear grid, which could probably have been reconfigured. Instead, by leveraging the projection paradigm, we have developed an alternative approach in our algorithm, as illustrated in the simplified 2D example of Fig. 3.

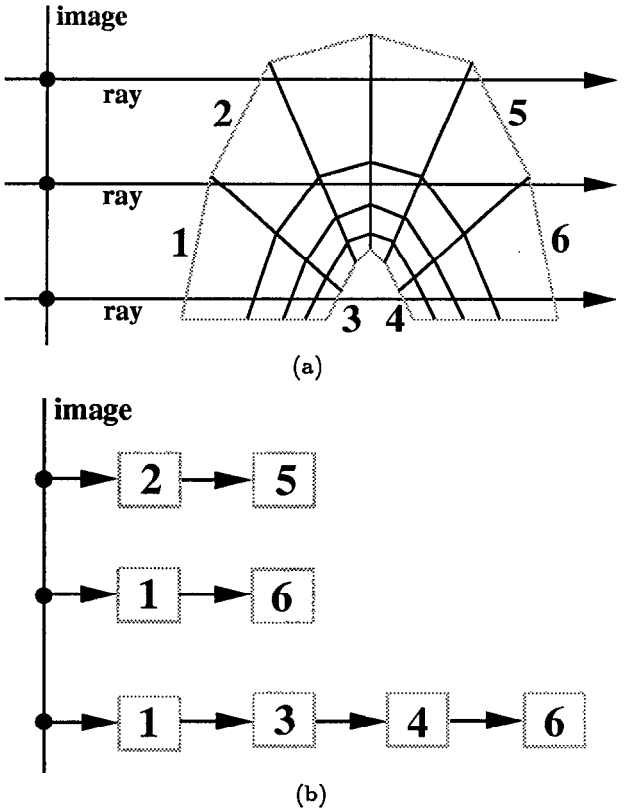

Figure 3: *(a) Assign a unique identification number to each exterior cell-face and project it onto the image screen. (b) For each image pixel, sort the intersecting exterior cell-faces by depth.*

Without incurring any extra storage, we first implicitly subdivide each quadrilateral cell-face (both exterior and interior) into two triangles. The triangle approximation is chosen over a bilinear patch because of our thrust to speed up the ray/cell-face intersections in the algorithm. Note that there are two possible ways of triangulating a cell-face, which may affect the data values reconstructed at the sample points (see Section 2.3). For simplicity, similar to the cell subdivision approach [4, 7, 11, 12, 18, 20], we arbitrarily choose one triangulation and consistently use it throughout the ray-casting process. Then, for each pixel on the image screen, we set up a bucket. Next, each exterior triangle is independently projected onto the image screen and scan-converted according to the image resolution. For each pixel (x, y) covered by the projection of an exterior triangle, a tuple (id, z) is generated from the 2D scan-conversion, where id is a unique identification number assigned to the triangle (see Fig. 3*a*) and z is the depth (screen-z) interpolated from the triangle vertices. This tuple (id, z) indicates that the ray cast from the pixel (x, y) intersects the exterior triangle id at the depth z. We place the tuple (id, z) into the bucket of pixel (x, y). As a result, after all the exterior triangles have been processed, within each pixel bucket there is a list of tuples which represent the exterior triangles intersecting with the ray cast from that pixel. For each bucket, we sort the tuples by the depth z, as shown in Fig. 3*b*. Subsequently, during the traversal of a ray, we retrieve the sorted tuples from the corresponding pixel bucket to find the first-entry and re-entry triangles.

This method is conceptually similar to Wilhelms et al. [17]. However, unlike their algorithm, where both the exterior and interior cell-faces are scan-converted and depth-sorted, at this phase we only process those exterior triangles that assist us in identifying the first-entry and re-entry triangles. For a typical curvilinear volume, since a ray usually re-enters the grid a limited number of times, the space consumed by our pixel buckets is mostly determined by the image resolution and should not be significant in comparison with the embedding scheme.

2.2 Identifying Exiting Cell-Faces

Once the ray enters a cell from one of its six cell-faces, it is necessary to find the cell-face from which the ray exits the current cell and enters the next cell. Garrity [4] showed that for a convex cell, one can intersect the ray with the 3D planes containing the candidate cell-faces. The cell-face whose intersection lies after the entry point and has the smallest distance from the entry point is the exiting cell-face. This technique was extended to the curvilinear domain by Uselton [14] and Ramamoorthy and Wilhelms [9]. Both groups employed a planar quadrilateral to approximate every cell-face. Furthermore, to speed up the ray/cell-face intersections, Ramamoorthy and Wilhelms [9] pre-computed the 3D planes containing the cell-faces and stored them at the expense of extra space.

Unfortunately, for a non-convex cell, Garrity's method [4] is no longer valid. As illustrated in Fig. 4, a ray enters the cell through cell-face AB. Following Garrity's technique, AD will be chosen as the exiting cell-face, although CD is the real exiting cell-face in this example. It can be seen that the additional requirement needed for a correct method is that the ray/cell-face intersection has to be inside the exiting cell-face. In other words, to handle both convex and concave cells, the algorithm would have to compute the 3D plane containing a candidate cell-face, find the 3D location of the ray/plane intersection, and check whether the intersection lies inside the cell-face. This could involve a great deal of computation. One may argue that for the curvilinear grids used in scientific computing, the cells are unlikely to be concave. However, even a small likelihood [9] makes the algorithm rather complicated to implement. It also prevents the algorithm from being applied on those curvilinear data

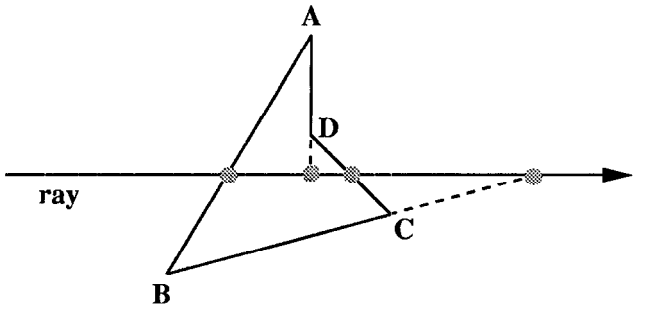

Figure 4: *Find the exiting cell-face of a non-convex cell.*

where cell convexity can not be readily guaranteed. In the following, we present an efficient alternative which can be used in both convex and concave cells.

Since we have subdivided each cell-face into two triangles, each cell consists of twelve triangles. As the ray enters a cell from one triangle, we check the ray against the other eleven triangles of the cell to find the exiting triangle. Instead of conducting the tests in 3D, we again explore the projection paradigm. Specifically, we project each candidate triangle T onto the image screen and test whether the 2D projected triangle contains the pixel (x, y) from which the ray is cast, as illustrated in Fig. 5. Note that this function is essentially equivalent to the operations of computing the ray/triangle intersection in 3D and checking whether the intersection lies within the triangle.

Boolean IntersectTriangle(x, y, T, z)
{

(1) project the three vertices of the triangle T onto the image screen

(2) check pixel (x, y) against the bounding rectangle of the 2D projected triangle. If the pixel is outside the bounding rectangle, return FALSE

(3) check pixel (x, y) against the three edges of the 2D projected triangle. If the pixel is outside the projected triangle, return FALSE

(4) interpolate the depth z at pixel (x, y) from the triangle vertices and return TRUE

}

Figure 5: *The function for testing whether a ray cast from pixel (x, y) intersects with triangle T.*

If the ray intersects with the triangle T, the function *IntersectTriangle*(x, y, T, z) will return TRUE coupled with the depth z of the intersection; otherwise, it will return FALSE. Steps 2-4 of the function are three operations of progressively increasing complexity. Their objective is to quickly terminate the test when the ray does not intersect with the triangle T, which usually happens to most of the candidate triangles. At Step 4, where pixel (x, y) is confirmed to be inside the 2D projected triangle, a linear interpolation scheme similar to 2D scan-conversion is used to reconstruct the depth of the intersection from the triangle vertices. Subsequently, if more than one triangle returns TRUE from the intersection test (i.e., the ray intersects with the cell multiple times), we designate the exiting triangle as the one whose intersection lies after the entry point and has the smallest depth. This allows the ray to re-enter the same cell at a later stage.

2.3 Data Interpolations

To accumulate the color c and opacity o along a ray, as well as the depth z, the scalar s at the ray/cell-face intersections has to be reconstructed from the grid vertices. That is, given the scalar values at the four grid vertices of a cell-face, one needs to find the scalar data for a sample point within the cell-face. The common solution is to transform the cell-face from its arbitrary 3D orientation to one of the coordinate planes (e.g., the XY-plane), and calculate the bilinear interpolation offsets in 2D [9]. This method unfortunately requires a considerable amount of computation.

Since we have used two triangles to approximate a quadrilateral cell-face in our algorithm, the task is reduced to interpolating the scalar data at the intersection from the triangle vertices. Taking advantage of the 3D-to-2D projection transformation already performed in the function *IntersectTriangle* (see Fig. 5), we implement the data interpolation as part of the ray/triangle intersection test. Specifically, at Step 4 of the function *IntersectTriangle*, where the ray is confirmed to have intersected with the triangle, we interpolate not only the depth z but also the scalar s from the triangle vertices. That is, the function becomes *IntersectTriangle*(x, y, T, z, s). Accommodating this extra interpolation may slightly increase the execution time of *IntersectTriangle*, but it completely eliminates a separate data interpolation phase. Now that we have the depth z and scalar s for the intersections, ray integral equations (the same as in [9]) which take into account the varying distances between consecutive samples are used to accumulate the color c and opacity o.

2.4 Algorithm Parallelization

Parallelization of the ray-casting algorithm was motivated by the fact that our FEM simulation was performed on a shared-memory, multi-processor architecture [21] and we expected to conduct the volume visualization of the simulation result on the same platform. Fortunately, as the rays are relatively independent of each other and only a single copy of the volumetric data is needed in the shared memory, it becomes fairly easy to parallelize the algorithm.

We have found from our initial experiments that in the sequential algorithm, less than 3% of the rendering time is spent on finding the first-entry and re-entry exterior triangles. Because of its interactions with the pixel buckets, a global resource shared by the exterior triangles, our parallel version of this step turned out to be not as fast as its sequential counterpart, considering the overheads caused by the parallelization. Therefore, we decided to run this step sequentially on a single processor. For the other steps of the algorithm, we have successfully parallelized them with a simple partitioning scheme and achieved satisfactory speedups (see Section 3.2). Specifically, given n^2 processors in the shared-memory architecture, we partition the image screen into sub-blocks of $n \times n$ pixels. For each pixel of a sub-block, we assign a processor to traverse the ray for computing the pixel color. When the ray traversal is terminated, the processor moves on to its corresponding pixel on the next sub-block. The overall computation is reasonably well distributed among the processors, due to the spatial coherence

among the rays within a sub-block. Fig. 6 provides a brief description of the parallel implementation.

```
ParallelRayCasting()
{
    (1) using processor 0, project every exterior triangle
        onto the image screen and depth-sort the tuples
        inside each pixel bucket
    (2) while (processor i is idle)
        {
            (2a) find its corresponding pixel on the next
                 partitioning sub-block and cast a ray
                 from that pixel
            (2b) with the assistance of the pixel bucket,
                 traverse the ray from cell to cell and ac-
                 cumulate the color c & opacity o based
                 on the interpolated data
        }
}
```

Figure 6: *Parallel version of our ray-casting algorithm.*

3 Experimental Results

In our project, the ray-casting algorithm and the FEM simulation model have been developed simultaneously. Before the simulation model had been well defined, we used several well-known curvilinear data sets to evaluate the consumption of computational power and memory space by our rendering algorithm. These included the *Blunt Fin* ($40\times32\times32$) [6], *Liquid Oxygen Post* ($38 \times 76 \times 38$) [10], and *Delta Wing* ($56 \times 54 \times 70$) [2], all freely available from NASA. In the following we present our testing results obtained from both the sequential and parallel implementations. See [21, 1] for our examples of the dynamic simulation of muscle volume deformation using FEM.

3.1 Sequential Implementation

The sequential algorithm was measured on an SGI Onyx (one 194MHz R10000 processor, 640MB memory, RE2 graphics). Using the ray-casting algorithm, we generated an animation which depicted a flight around the Blunt Fin. This animation consisted of more than 100 individual frames with the virtual camera looking at the Blunt Fin from different angles. All the images were rendered using the same transfer functions, with high density mapped to red and low density mapped to green. Fig. 7 (in the Color Section) shows three frames of the animation. Table 1 illustrates the average time spent on each step of the algorithm and the total rendering time.

Similarly, we generated animations for the Liquid Oxygen Post and the Delta Wing. Fig. 8 (in the Color Section) shows three images produced by rendering the energy scalar field of the Liquid Oxygen Post, where low energy was mapped to red and high energy was mapped to green. The rendering time distribution for this data set is illustrated in Table 2. Additionally, three images of the Delta Wing are shown in Fig. 9 (in the Color Section), with low density mapped to green and high density mapped to red. Table 3 illustrates the

Table 1: *Average time (seconds) spent on ray-casting the Blunt Fin with image size* 500×500.

Data Set	Blunt Fin
First-Entry & Re-Entry	0.43
Traversal & Interpolations	18.63
Color & Opacity Compositing	0.65
Total	19.71

Table 2: *Average time (seconds) spent on ray-casting the Liquid Oxygen Post with image size* 500×500.

Data Set	Liquid Oxygen Post
First-Entry & Re-Entry	0.69
Traversal & Interpolations	38.17
Color & Opacity Compositing	1.51
Total	40.37

Table 3: *Average time (seconds) spent on ray-casting the Delta Wing with image size* 500×500.

Data Set	Delta Wing
First-Entry & Re-Entry	0.75
Traversal & Interpolations	38.93
Color & Opacity Compositing	1.15
Total	40.83

rendering time distribution. Note that the images in Figs. 7-9 are relatively transparent. If a more opaque transfer function were employed to accumulate the opacity along the rays, many rays could have been terminated earlier, further reducing the rendering time.

In the following, we compare our algorithm with two techniques recently proposed by Silva and Mitchell [12] and Yagel et al. [20]. We have chosen these two algorithms because they represent state-of-the-art research on the rendering of non-rectilinear volumes and provide testing results on the same three data sets. Unlike our algorithm which performs the ray-casting directly on the curvilinear volume, both Silva and Mitchell [12] and Yagel et al. [20] decomposed the curvilinear cells into tetrahedra and then rendered the tetrahedral representation. Silva and Mitchell intersected the tetrahedral grid with sweep-planes going through the scanlines and being perpendicular to the image screen. Polygons formed on each sweep-plane were then rendered in a sweep-line manner to generate pixel colors for the scanline. Alternatively, Yagel et al. intersected the tetrahedral volume with a set of slicing planes parallel to the image screen and used graphics hardware to project the 2D polygons formed on the planes. We do not compare against previous ray-casting techniques for curvilinear volumes [9, 14], because they were developed several years ago on platforms considered today to be out-of-date.

Tables 4-6 show the performance comparison on the Blunt Fin, Liquid Oxygen Post, and Delta Wing, respectively. When interpreting these tables, note that different rendering parameters and computer workstations were used in these three algorithms. Silva and Mitchell performed the measurements on a single processor of an SGI Power Challenge (sixteen 194MHz R10000 processors, 3GB memory, IR graphics), while Yagel et al. obtained the timing from an SGI

Table 4: *Performance comparison among Silva and Mitchell [11], Yagel et al. [19], and our algorithm on the Blunt Fin data set.*

Algorithm	S & M	Yagel et al.	Ours
Image Size	530 × 230	*not reported*	300 × 300
Rendering (sec)	22	9.11	7.09
Memory (MB)	8	21.2	1.8

Table 5: *Performance comparison among Silva and Mitchell [11], Yagel et al. [19], and our algorithm on the Liquid Oxygen Post data set.*

Algorithm	S & M	Yagel et al.	Ours
Image Size	300 × 300	*not reported*	300 × 300
Rendering (sec)	37	20.45	14.60
Memory (MB)	22	56.7	3.7

Table 6: *Performance comparison among Silva and Mitchell [11], Yagel et al. [19], and our algorithm on the Delta Wing data set.*

Algorithm	S & M	Yagel et al.	Ours
Image Size	300 × 300	*not reported*	300 × 300
Rendering (sec)	64	42.97	14.79
Memory (MB)	44	111.7	5.7

Crimson (one 100MHz R4000 processor, 64MB memory, RE graphics), but using only 50 slicing planes. In terms of image quality, our algorithm is comparable to that of Silva and Mitchell, but Yagel et al.'s algorithm may not approach our image quality. This is because, instead of adaptively sampling along each individual ray to account for the change of grid density, even the adaptive slicing scheme proposed by Yagel et al. imposed the same screen-z sampling rate for all the image pixels. Consequently, some areas of the grid may be under-sampled, while other areas may be over-sampled. To capture the details of the small cells, a much higher number of slicing planes would have been required, leading to a significant increase in both rendering time and memory usage.

3.2 Parallel Implementation

Our parallel algorithm was implemented on an SGI Power Challenge (sixteen 194MHz R10000 processors, 3GB memory, IR graphics). The measurements were obtained while the machine was being routinely used by other users as well. For illustration purposes, in the following we divide the rendering process into two phases: *Entry* and *Others*. The former corresponds to Step 1 of function *ParallelRayCasting* (see Fig. 6), where only one process is employed to set up the pixel buckets. While the latter corresponds to Step 2 of function *ParallelRayCasting*, where multiple processors are concurrently used to traverse the rays and perform the accumulations.

We ran our parallel algorithm on nine processors and sixteen processors of the Challenge, respectively. Tables 7-9 show the performance comparison on the three data sets. The slight increase in the time taken by the *Entry* phase was due to the overheads of setting up the parallelization.

Table 7: *Comparison between the sequential and parallel algorithms on ray-casting the Blunt Fin with image size* 500 × 500.

	Entry	Others	Total
1 Processor (sec)	0.38	19.19	19.57
9 Processors (sec)	0.47	2.17	2.64
9 Processors Speedup	0.81	8.84	7.41
16 Processors (sec)	0.47	1.30	1.77
16 Processors Speedup	0.81	14.76	11.06

Table 8: *Comparison between the sequential and parallel algorithms on ray-casting the Liquid Oxygen Post with image size* 500 × 500.

	Entry	Others	Total
1 Processor (sec)	0.66	40.22	40.88
9 Processors (sec)	0.80	4.56	5.36
9 Processors Speedup	0.83	8.82	7.63
16 Processors (sec)	0.81	2.63	3.44
16 Processors Speedup	0.81	15.29	11.88

Table 9: *Comparison between the sequential and parallel algorithms on ray-casting the Delta Wing with image size* 500 × 500.

	Entry	Others	Total
1 Processor (sec)	0.70	40.22	40.92
9 Processors (sec)	0.85	4.62	5.47
9 Processors Speedup	0.82	8.71	7.48
16 Processors (sec)	0.84	2.72	3.56
16 Processors Speedup	0.83	14.79	11.49

The parallelized *Others* phase achieved an average speedup of 8.79 on nine processors, leading to an overall average speedup of 7.51 for the parallel algorithm. With sixteen processors, the *Others* phase achieved an average speedup of 14.95, resulting in an overall average speedup of 11.48 for the parallel algorithm. The algorithm non-linear scalability on all sixteen processors could be due to the cache and/or bus contentions, as well as the system resources taken by other users when the measurements were conducted.

4 Conclusions and Future Work

In this paper, we have presented an efficient yet robust ray-casting algorithm for high-quality rendering of a curvilinear volume. As demonstrated in the experimental studies, with this algorithm we have successfully achieved the goal of alleviating the consumption of both computational power and memory storage. To the best of our knowledge, our unoptimized rendering time and memory usages on the Blunt Fin, Liquid Oxygen Post, and Delta Wing, respectively, are the best reported to date in the literature. Although this work was originally driven by our application of dynamic simulation, the generality and favorable performance of our algorithm also makes it an appealing alternative to existing rendering techniques.

As indicated in Tables 1-3, for the sequential implementation about 95% of the rendering time is currently taken

by the steps of *Traversal* and *Interpolations* (i.e., function *IntersectTriangle*). We plan to further optimize this function to improve the overall rendering performance. For the parallel implementation the resulting speedup of the *Others* phase is rather satisfactory, but the sequential execution of the *Entry* phase negatively affects the overall speedup. We are working on techniques to address this issue.

As a long term goal, we plan to extend the proposed acceleration techniques to the rendering of unstructured grids. Since the projection method employed in our algorithm does not inherently depend on the structure of curvilinear cells, it should be applicable to unstructured cells as well. Another plan for future research is the anti-aliasing rendering for non-rectilinear grids. Although the ray-casting approach has taken into account the changing grid density along the screen-z direction, the potential under-sampling/over-sampling problem in the screen-x and screen-y directions needs to be investigated as well.

Acknowledgments

This work was supported by the National Science Foundation (grant MIP-9527694), the Office of Naval Research grant (N000149710402), the Naval Research Laboratory (grant N00014961G015), and by a grant from the Mitsubishi Electric Research Laboratory. The three curvilinear data sets of Blunt Fin, Liquid Oxygen Post, and Delta Wing are courtesy of NASA. Thanks to Chi-kun Lam for his implementation of the parallel algorithm. We also thank Kathleen McConnell, Edmond Prakash, Claudio Silva, and Ming Wan for valuable comments on drafts of the paper.

References

[1] Y. Chen, Q. Zhu, A. Kaufman, and S. Muraki. Physically-based animation of volumetric objects. In *Proc. Computer Animation '98*, pages 154–160. IEEE Computer Society Press, 1998.

[2] J. Ekaterinaris and L. Schiff. Vortical flows over delta wings and numerical prediction of vortex breakdown. In *AIAA Aerospace Sciences Conference*, 1990. Paper 90-0102.

[3] T. Frühauf. Raycasting of nonregularly structured volume data. In *Computer Graphics Forum (Proc. EUROGRAPHICS '94)*, pages 294–303, 1994.

[4] M. Garrity. Raytracing irregular volume. In *Computer Graphics (Proc. 1990 ACM Workshop on Volume Visualization)*, pages 35–40, 1990.

[5] A. Van Gelder and J. Wilhelms. Rapid exploration of curvilinear grids using direct volume rendering. In *Proc. IEEE Visualization '93*, pages 70–77. IEEE Computer Society Press, 1993.

[6] C. Hung and P. Buning. Simulation of blunt-fin induced shock wave and turbulent boundary layer separation. In *AIAA Aerospace Sciences Conference*, 1984. Paper 84-0457.

[7] N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. In *Computer Graphics (Proc. 1990 ACM Workshop on Volume Visualization)*, pages 27–33, 1990.

[8] C. Prakash and S. Manohar. Volume rendering of unstructured grids: A voxelization approach. *Computers and Graphics*, 19(5):711–726, 1995.

[9] S. Ramamoorthy and J. Wilhelms. An analysis of approaches to ray-tracing curvilinear grids. Technical Report UCSC-CRL-92-07, University of California at Santa Cruz, 1992.

[10] S. Rogers, D. Kwak, and U. Kau. A numerical study of three-dimensional incompressible flow around multiple posts. In *AIAA Aerospace Sciences Conference*, 1986. Paper 86-0353.

[11] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *Computer Graphics (Proc. 1990 ACM Workshop on Volume Visualization)*, pages 63–69, 1990.

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

[13] C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. In *Proc. 1994 ACM Symposium on Volume Visualization*, pages 83–89, 1994.

[14] S. Uselton. Volume rendering of computational fluid dynamics: Initial results. Technical Report RNR-91-026, NAS-NASA Ames Research Center, Moffett Field, California, 1990.

[15] J. Wilhelms. Pursuing interactive visualization of irregular grids. *The Visual Computer*, 9:450–458, 1993.

[16] J. Wilhelms, J. Challinger, N. Alper, S. Ramamoorthy, and A. Vaziri. Direct volume rendering of curvilinear volumes. In *Computer Graphics (Proc. 1990 ACM Workshop on Volume Visualization)*, pages 41–48, 1990.

[17] J. Wilhelms, A. Van Gelder, P. Tarantino, and J. Gibbs. Hierarchical and parallelizable direct volume rendering for irregular and multiple grids. In *Proc. IEEE Visualization '96*, pages 57–64. IEEE Computer Society Press, 1996.

[18] P. Williams. Interactive splatting of nonrectilinear volumes. In *Proc. IEEE Visualization '92*, pages 37–44. IEEE Computer Society Press, 1992.

[19] P. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.

[20] R. Yagel, D. Reed, A. Law, P. Shih, and N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *Proc. 1996 ACM/IEEE Symposium on Volume Visualization*, pages 55–62, 1996.

[21] Q. Zhu, Y. Chen, and A. Kaufman. Real-time biomechanically-based muscle volume deformation using FEM. To appear in *Proc. EUROGRAPHICS '98.*

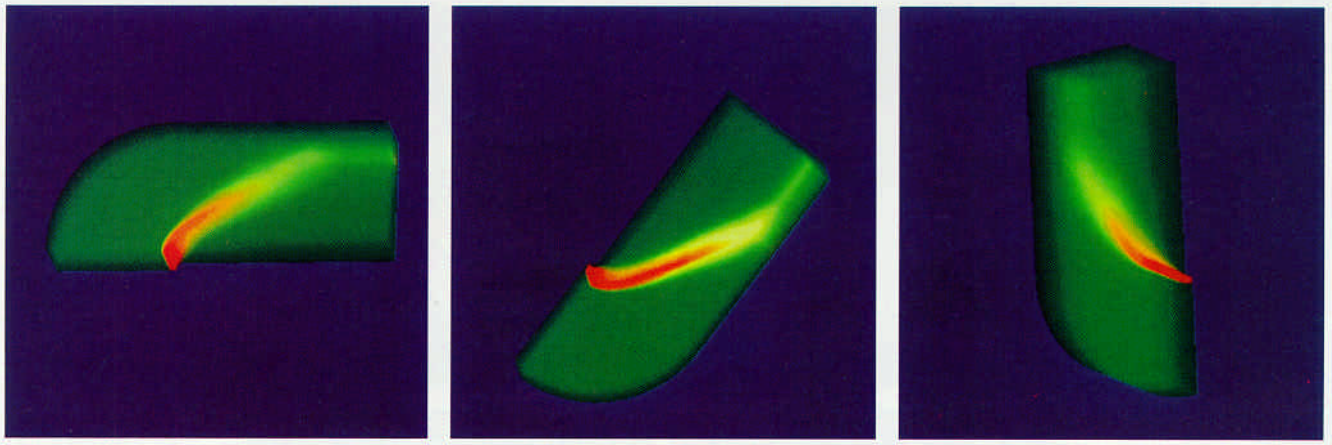

Figure 7: *Images of the Blunt Fin from three different viewing angles (high density is mapped to red and low density is mapped to green).*

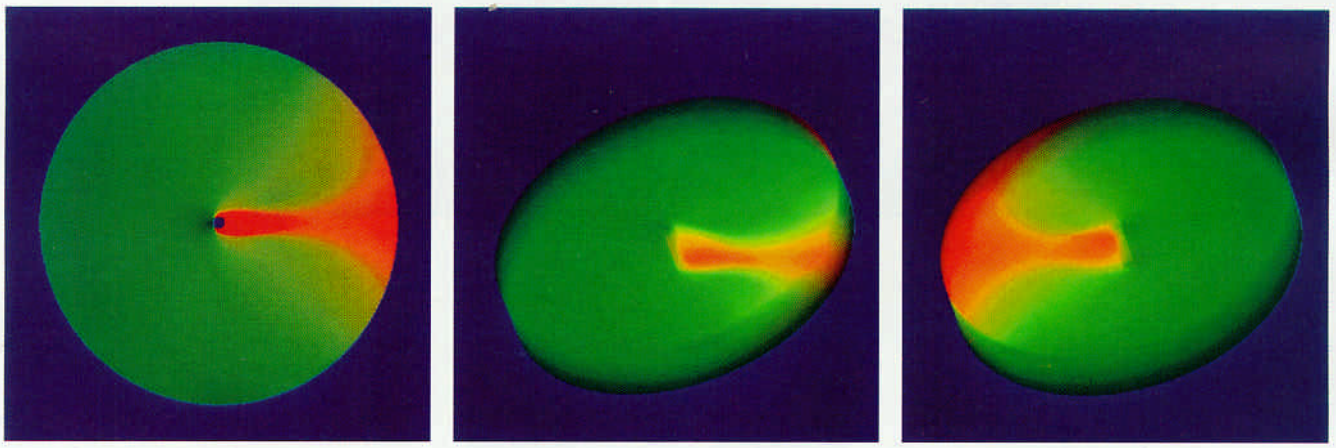

Figure 8: *Images of the Liquid Oxygen Post from three different viewing angles (low energy is mapped to red and high energy is mapped to green).*

Figure 9: *Images of the Delta Wing from three different viewing angles (low density is mapped to red and high density is mapped to green).*