High Performance Presence-Accelerated Ray Casting

Ming Wan*, Arie Kaufman*, and Steve Bryson†

*Center for Visual Computing (CVC) and Department of Computer Science
State University of New York at Stony Brook, Stony Brook, NY 11794-4400

†NASA Ames Research Center
Moffett Field, CA 94035-1000

Abstract

We present a novel presence acceleration for volumetric ray casting. A highly accurate estimation for object presence is obtained by projecting all grid cells associated with the object boundary on the image plane. Memory space and access time are reduced by run-length encoding of the boundary cells, while boundary cell projection time is reduced by exploiting projection templates and multiresolution volumes. Efforts have also been made towards a fast perspective projection as well as interactive classification. We further present task partitioning schemes for effective parallelization of both boundary cell projection and ray traversal procedures. Good load balancing has been reached by taking full advantage of both the optimizations in the serial rendering algorithm and shared-memory architecture. Our experimental results on a 16-processor SGI Power Challenge have shown interactive rendering rates for 256^3 volumetric data sets at $10 - 30$ Hz. This paper describes the theory and implementation of our algorithm, and shows its superiority over the shear-warp factorization approach.

Keywords: Volume rendering, presence acceleration, run-length encoding, projection template, multiresolution volumes, interactive classification, parallel processing.

1 Introduction

An effective approach to achieve high frame rates for volume rendering is to parallelize a fast rendering algorithm that relies on some algorithmic optimizations [1, 2, 3, 4]. Two requirements must be met for this approach to achieve interactive rendering. First, the serial volume rendering algorithm must be fast enough. Second, the parallel version of the serial algorithm must scale well as the number of processors increases.

Many parallel volume rendering algorithms have been developed by optimizing serial volume renderers. Among the most efficient ones is Lacroute's [3], a real-time parallel volume rendering algorithm on a multiprocessor SGI Challenge using the shear-warp factorization [5], which could render a 256^3 volume data set at over 10 Hz. A dynamic task stealing scheme was borrowed from [1] for load balancing. Parker et al. [4] proposed another interactive parallel ray casting algorithm on SGI workstations. Using 128 processors, their algorithm rendered a 1GByte full resolution Visible Woman data set at over 10 Hz. One of their optimizations for ray casting was using a multi-level spatial hierarchy for space leaping.

In this paper, we explore the effective parallelization of our boundary cell-based ray casting acceleration algorithm on multiprocessors. The serial algorithm is derived from the acceleration technique of *bounding-boxes*. This technique consists of three steps: First, the object is surrounded with tightly fit boxes or other easy-to-intersect geometric primitives such as spheres. Then, the intersection of the rays with the bounding object is calculated. Finally, the actual volume traversal along each ray commences from the first intersection point as opposed to starting from the volume boundary. Unlike other kinds of presence acceleration techniques which traverse a hierarchical data structure, such as octrees [4, 6] and K-d trees [7] to skip over empty regions, this approach directly and hence more quickly traverses the original regular grid.

Obviously, the effectiveness of a bounding-boxes approach depends on its ability to accurately calculate the intersection distance for each viable ray with minimal computational overhead. Therefore, in our previously proposed boundary-cell based ray casting method [8], we accurately detected the object boundary at each grid cell of the volumetric data set. Each cell was the volume contained within the rectangular box bounded by eight neighboring grid vertices (*voxels*). The distance information from the object boundary to the image plane was obtained by projecting all *boundary cells* (cells pierced by the object boundary) onto the image plane. This projection procedure was accelerated both by exploiting the coherence of adjacent cells and employing a generic projection template. The experimental results showed that the projection time was faster than that of the PARC (Polygon Assisted Ray Casting) algorithm [9] which was accelerated by graphics hardware.

However, our previously proposed method [8] had some limitations. First, it was more effective for small volume data of less than 128^3 voxels. Second, it only supported fast ray casting with parallel projection. In this paper, we present an improved version to solve these problems. We propose to run-length encode the detected boundary cells. This data compression reduces both memory space and access time. Multiresolution volumes are further exploited to reduce the number of boundary cells, so that our method is capable of rendering larger volumes at interactive rates. Efforts have also been made towards a fast perspective projection as well as interactive classification.

Based on such an improved serial rendering algorithm, we have developed our parallel rendering algorithm using effective task partitioning schemes for both boundary cell projection and subsequent ray traversal procedures. Good load balancing has been reached by taking full advantage of both the optimizations in the rendering algorithm and shared-memory architecture.

Our parallel algorithm has been implemented on a Silicon Graphics Power Challenge, a bus-based shared-memory MIMD (Multiple Instruction, Multiple Data) machine with 16 processors. Rendering rates for 256^3 volumetric data sets are as fast as $10 - 30$ Hz — among the fastest reported. A detailed comparison between our algorithm and shear-warp factorization approach [3] is given in Section 5. It is difficult to compare performances between the method in [4] and ours, since the former used a much larger data set and eight times more processors. Yet, these two methods do have some similarities — both are essentially ray casting algorithms running on multiprocessors, and both are interested in the

*Email:{mwan|ari}@cs.sunysb.edu
†Email:bryson@nas.nasa.gov

object boundary. One significant difference lies in that their method could only display the boundary surface of the object, while ours can also visualize the interior structures with translucency. The description of our serial algorithm and its parallel version are given in Sections, 2 and 3 respectively. Performance results are reported in Section 4.

2 The Serial Algorithm

Our serial rendering algorithm can be completed in three steps: (1) run-length encode the boundary cells at a preprocessing stage; (2) project the run-length encoded boundary cells onto the image plane to produce the intersection distance values for each pixel; and (3) for each viable ray that intersects an object in the volume, start sampling, shading and compositing from the intersection. A discussion on support of interactive volume classification between renderings is given at the end of this section.

2.1 Boundary Cell Encoding

Since boundary cell information is viewpoint-independent, we can obtain it by scanning the volume in an off-line preprocessing stage. Essentially, the scanline-based run-length encoding scheme exploits the 1D spatial coherence which exists along a selected axis direction [10]. It gives a kn^2 compressed representation of the data in a n^3 grid, where the factor k is the mean number of runs. A run is a maximal set of adjacent voxels having the same property, such as having the same scalar field value, or associated with the same classified material (see Section 2.4 for interactive classification). Obviously, only when k is low can such a scheme be efficient. Fortunately, this is true for a *classified* volume: a volume to which an opacity transfer function has been applied [5]. We use this scheme to encode boundary cells in the volume. In our algorithm, each run is a maximal set of adjacent boundary cells in the same grid cell scanline aligned with a selected axis. X axis is selected for run-length encoding in this paper.

The specific data structure we use for run-length encoding of boundary cells includes a linear run list L and a 2D table T (see Figure 1). List L contains all the runs of boundary cells. Each element $L[t]$ of L represents a run, including the location of the first boundary cell $C(i, j, k)$ of this run and the run length. The position of a cell $C(i, j, k)$ is determined by one of its eight voxels with the lowest X, Y, Z coordinate value i, j, k. Accordingly, cell $C(i, j, k)$ is the ith cell in scanline (j, k). Table T records the distribution information of the boundary cells among volume scanlines. Each element $T[j, k]$ holds the number of the boundary cells located in scanline (j, k). According to the information in table T and list L, we can quickly skip over empty scanlines and empty runs.

In order to reach a high data compression, we suggest that, first, for each run t in list L, only the X coordinate i of the starting cell $C(i, j, k)$ needs to be stored in $L[t]$, instead of all three coordinates. We can easily infer the other two coordinates. Second, all boundary cells which have 6 face-connected boundary cells should be ignored as non-boundary cells in our data structure, because they have no contribution for our object boundary estimation. Third, table T can also be run-length encoded. Each run is a maximal set of adjacent elements having the same number.

The space complexity S of our run-length encoding data structure is the sum of the space complexities of list L and table T:

$$S(Boundary) = kn^2 \times S(L[t]) + n^2 \times S(T[j, k]) \qquad (1)$$

where k is the mean number of runs per scanline. Two fields are needed for each element $L[t]$ and one for $T[j, k]$. These fields can

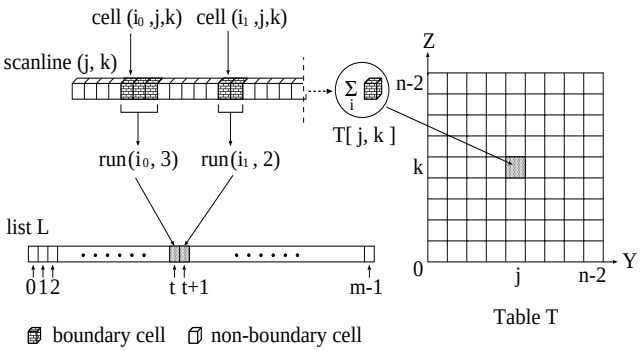


Figure 1: *Data structure for run-length encoding of boundary cells.*

be represented by integer numbers with 4 bytes each on SGI workstations, or 1 byte *character* each, if n is no more than 256. Therefore, Equation 1 can be written as:

$$S(Boundary) = (2k + 1)n^2 \cdot sizeof(int) \qquad (2)$$

2.2 Boundary Cell Projection

By skipping over the runs of non-boundary cells, our run-length encoding scheme not only provides high data compression, but also leads to fast 3D scan over the volume during the boundary cell projection procedure. Both parallel and perspective projections are supported in our algorithm.

2.2.1 Parallel Projection

In parallel projection, the projected area of every cell has the same shape and size in the image plane. Only the projected position and distance of the cell center to the image plane are different from cell to cell. Based on such a projection property, we employ a *generic* projection template M to speed up boundary cell projection.

Establish Projection Template

In our algorithm, the generic projection template M has the same size as the bounding box of the projected area of a cell on the image plane. Each element of the template (template-pixel) has two respective components recording near and far distances. To calculate the distance values of template M, we first choose an arbitrary cell from the volume, then place the center of the template over the center point of the cell. We ensure that the template is not only parallel to the image plane but also aligned with the primary axes of the image plane. The origin is defined at the center of the template. Three different levels-of-accuracy templates can be selected in our algorithm. In the low level-of-accuracy template, the far distance value of each template-pixel is distance d of the farthest voxel of the cell to the template, and the near distance value of each template-pixel is -d. In the middle level-of-accuracy template, distance values of those template-pixels which are not covered by the projected cell are set to infinity, and the remaining template-pixels have the same values as those in the low level template. In the high level-of-accuracy template, the near and far distance values of the template-pixels are accurately calculated by scan-converting both front facing and back facing surfaces of the cell.

Obviously, different level-of–accuracy templates provide different accuracy of distance information. Note that in a high resolution volume data set, the cells are very small and densely overlapped from any viewing direction. Therefore, a rough approximation

based on the low level-of-accuracy template is often good enough to support efficient skipping over empty space, as evidenced from our experimental results.

Determine Projection Position

In parallel projection, once the first cell $C(0,0,0)$ is projected onto the image plane, the position of remaining cells can be quickly calculated by incremental vectors with only addition operations. Specifically, assume that the center point of the first cell $C(0,0,0)$ is projected to position (x_0, y_0) on the image plane with depth z_0, and that $\Delta X, \Delta Y, \Delta Z$ are respectively the vector directions of volume axis X, Y, Z in image space, and volume size is $N_x \times N_y \times N_z$ with unit spacing between voxels. Then, position (x_1, y_1) and depth z_1 of the center point of the cells adjacent to cell $C(0,0,0)$ along volume axes X, Y, Z can be respectively calculated by the following equations:

$$[x_{i+1}, y_{i+1}, z_{i+1}] = [x_i, y_i, z_i] + \Delta X, \quad i = 0..(N_x - 3) \quad (3)$$

$$[x_{j+1}, y_{j+1}, z_{j+1}] = [x_j, y_j, z_j] + \Delta Y, \quad j = 0..(N_y - 3) \quad (4)$$

$$[x_{k+1}, y_{k+1}, z_{k+1}] = [x_k, y_k, z_k] + \Delta Z, \quad k = 0..(N_z - 3) \quad (5)$$

When run-length encoding is used in our algorithm, the relationship between two adjacent boundary cells in list L is more complicated. These two adjacent cells could be located in the same run, or in two adjacent runs at the same scanline, or in two runs at different scanlines. Assume that the projection information of the two adjacent boundary cells C_i and C_{i+1} are respectively $[x_i, y_i, z_i]$ and $[x_{i+1}, y_{i+1}, z_{i+1}]$. If these two cells are located in the same run, then $[x_{i+1}, y_{i+1}, z_{i+1}]$ can be found from $[x_i, y_i, z_i]$ with a single vector addition operation, by using Equation 3. Otherwise, if C_i and C_{i+1} are located in different runs at the same scanline, then $[x_{i+1}, y_{i+1}, z_{i+1}]$ can be calculated from $[x_i, y_i, z_i]$ with two vector addition operations and one vector multiplication operation:

$$[x_{i+1}, y_{i+1}, z_{i+1}] = [x_i, y_i, z_i] + (x_{i+1} - x_i) \cdot \Delta X \quad (6)$$

In the case that cell C_{i+1} is located on a new scanline in the same slice or in a new slice, its projection information can be similarly calculated from that of the first cell on the previous scanline or slice, by respectively using Equations 4 and 5.

By using this incremental method, the time-consuming 4×4 matrix multiplications for projection are applied solely to cell $C(0,0,0)$, rather than to all boundary cells. Thus, the projection procedure is greatly accelerated.

Fill Projection Buffer

We utilize two projection buffers, Z_n and Z_f, having the same size as the resultant image, to respectively record the nearest and farthest intersection distances to the object boundary along the rays cast from the image pixels.

Once the projected position of a boundary cell is determined, the specific projection template M_1 of this cell can be quickly generated by adding the distance between the cell center and image plane to both near and far distance values in each viable element of the generic template M. The manner is straightforward for using current template M_1 to update distance values in the two projection buffers. First, place template M_1 over the image plane with its center template-pixel over the image position where the cell center is projected. Then, for each image-pixel covered by the template, compare the nearest and farthest intersection distances (i.e., z_n and z_f) in the corresponding buffer-pixels with the non-infinity near and far distances (i.e., d_n and d_f) in the corresponding element of template M_1. Specifically, if z_n is greater than d_n, z_n is replaced by d_n in the near z-buffer; meanwhile, if z_f is less than d_f, z_f is replaced by d_f in the far z-buffer.

In the situation where the center point of the cell is not exactly projected on an image pixel, one image pixel covered by the template may be surrounded by two to four template-pixels. Thus, nearest distance d_n and farthest distance d_f of the surrounding template-pixels are taken to give a conservative distance estimation.

2.2.2 Perspective Projection

Perspective projection is of particular importance when the viewing point is getting close to the data or is located inside the volume, such as during the interactive navigation inside the human colon of our 3D virtual colonoscopy [11]. The implemented perspective projection procedure in our algorithm is similar to the parallel projection. We still adopt projection templates for fast projection. However, since different cells have different perspective projection shapes and sizes due to their different distances and directions to the view point, there is no generic projection template for all cells. Furthermore, the incremental method for finding the projection information of the adjacent boundary cell does not work for perspective projection. This is because cells aligned with a volume axis no longer have fixed spacing on the screen.

During our implementation, the low level-of-accuracy template has turned out to be the most competitive candidate among the three. This low template is essentially a degenerated template, including only the height, width, and near and far distances of the projected boundary cell. Whenever one cell does not cover too many screen pixels, the object boundary estimation based on these low templates is satisfactory. Under perspective projection, such a template for a specific boundary cell can be quickly generated. This can be done by projecting the eight vertices of the cell onto the image plane to find the bounding box of the projected area of the cell as well as its minimal and maximal distances to the screen. This template can be directly used to update the near and far projection buffers. Furthermore, since there are four vertices shared by two adjacent boundary cells in the same run, projection information of these four shared vertices from one boundary cell can be reused by the neighboring boundary cell for further speedup. As a result, although perspective projection involves more computation than parallel projection, it can still be done rapidly. Specifics of projection time from our experiments on various data sets are reported in Section 4.

2.3 Ray Traversal

Depending on the intersection distance information in projection buffers Z_n and Z_f, the ray casting procedure is accelerated by casting rays only from viable pixels on the image plane, and traversing each ray from the closest depth to the farthest depth. Other effective ray casting optimizations, such as adaptive image sampling [12] and early ray termination [6], can be conveniently incorporated to further speedup our ray traversal procedure. For example, by employing early ray termination, the traversal along each viable ray stops before the farthest intersection is reached if the accumulated opacity has reached unit or exceeded a user-selected threshold of opacity.

The ray traversal procedure of our algorithm is often rapidly completed, because the overall complexity of the ray casting algorithm is greatly reduced. Assume that the volume size is n^3 and image size is n^2. To generate such a ray casting image with parallel projection, rendering complexity of a brute-force ray caster would be $O(n^3)$. In our algorithm, rendering complexity can be reduced to $O(kn^2)$. Although the value of k is data dependent, it is often quite small compared with n, especially when early ray termination

is employed, unless a substantial fraction of the classified volume has low but non-transparent opacity. Note, however, that such classification functions are considered to be less useful [5].

In fact, our accelerated ray traversal speed sometimes becomes so fast that it may approach boundary cell projection speed, especially for larger data sets. When this happens, we are pleased to further reduce the projection time by decreasing the resolution of the boundary cells, since the accuracy of the current object boundary estimation is unnecessarily high. One solution is to reduce the volume resolution by merging m^3 neighboring cells into a *macrocell*. If all the cells in a macrocell are non-boundary cells, this macrocell is a non-boundary macrocell; otherwise, it is a boundary macrocell. From our experiment with a $256 \times 256 \times 124$ MRI data set of a human brain, even merging the eight neighboring cells ($m = 2$) in the original volume leads to a three-fold decrease in cell projection time and nearly the same ray traversal time.

Another approach is to use a lower levels-of-detail (LOD) volume for fast object boundary estimation, and then use the original high resolution volume for accurate ray traversal. This approach may produce more accurate estimation, especially when the selected value for m is large. Yet, the user should make sure that the object represented in the lower LOD volume is not "thinner" than its original size, which can be guaranteed either by the modeling algorithm for LOD or by adjusting our projection templates.

2.4 Interactive Classification

In a practical application, the user may want to change the opacity transfer function between renderings while exploring a new data set. Most existing algorithms that employ spatial data structures require an expensive preprocessing step when the transfer function changes, and therefore can not support interactive volume classification. Although our algorithm presented thus far works on a classified volume with a fixed transfer function, it can easily support interactive classification with some minor constraint on the modification of the transfer function.

In our algorithm, we define an *opacity threshold* in a transfer function as the minimal scalar field value in the volumetric data set associated with a non-zero opacity. Once this opacity threshold is given in the transfer function, all boundary cells can be determined, of which some but not all the eight vertices possess field values less than the opacity threshold. If the transfer function changes, the previous run-length encoding of the boundary cells based on the previous opacity threshold may not be an appropriate data structure for the new object. Yet, note that an increase in opacity threshold only shrinks the object volume coverage, and that an object with a higher opacity threshold is always enclosed by an object with a lower opacity threshold. Consequently, the run-length encoding of an object boundary with a low opacity threshold can be used as an overestimate of another object boundary with a higher opacity threshold. It follows that, if we start from an object with the lowest opacity threshold, and create run-length encoding of boundary cells according to that opacity threshold, then we can avoid repeating the preprocessing step for boundary cell detection and run-length encoding, when the opacity transfer function changes between renderings. We do this by always using the same run-length encoding data structure as an overestimate for the new object boundary specified by the modified transfer function with a higher opacity threshold.

Although we can now correctly render images of interactively classified volume, the rendering rates may slow down greatly under radical changes of transfer function, for two reasons. First, the number of boundary cells in our fixed run-length encoding data structure can be larger than that of the shrunken object specified by a higher opacity threshold. This may lead to a longer projection time. Second, such an overestimation for the shrunken object boundary may cause longer ray traversal time due to unnecessary samplings outside the shrunken object.

Fortunately, in a typical classified volume, $70 - 95\%$ of the voxels are transparent [7, 5]. From this we know that the total number of boundary cells from each object is very small compared with the volume size. The projection time of these boundary cells is further shortened by employing run-length encoding and template-assisted projection. Accordingly, the difference of projection time between different objects is often minor. Also, since the possible objects are crowded in a small part of the volume, the boundaries of these objects are often so close to each other that the overestimation does not cause much extra ray traversal time. In brief, our algorithm allows interactive classification with a moderate performance penalty. The experimental results from different data sets with both interactive classification and fixed pre-classification are given in Section 4.

3 The Parallel Algorithm

In general, there are two types of task partitionings for parallel volume rendering algorithms: object-based [2] and image-based partitionings [1, 3, 4], respectively working on the volume and image domains. In order to take full advantage of optimizations in the serial algorithm, we have designed an object-based task partitioning scheme for boundary cell projection, and an image-based partitioning scheme for the ray traversal procedure. The shared-memory architecture of the SGI Power Challenge fully supports the implementation of our parallel algorithm.

3.1 Object-Based Partitioning for Boundary Cell Projection

To achieve high processor utilization during the boundary cell projection procedure, the volume should be carefully divided and assigned to the processors so that each processor possesses a subset of the volume with an equal number of boundary cells. Based on our run-length encoding data structure, we are able to precisely divide the volume into subvolumes of contiguous grid cells, each containing a roughly equal number of boundary cells. For the convenience of implementation, we used a run instead of a cell as the fundamental unit of work. Compared with other options, such as static interleaved partitionings and dynamic partitionings, our static contiguous partitioning has several advantages. It maximizes spatial locality in the run-length encoding data structure, and therefore minimizes the memory stall time caused by cache misses. In addition, as a static scheme, less synchronization is required, and task redistribution overhead is also avoided.

Once the volume is distributed to all available processors, each processor works concurrently and independently on its subvolume by scanning and projecting all related boundary cells onto the image plane. Since the image plane is shared by all processors, each processor establishes a separate pair of near and far projection buffers with the same size as the resultant image, in order to avoid memory access conflict. Each processor finishes its work and supplies a pair of partial projection buffers within about the same span of time. The complete (unified) projection buffers of the whole volume are obtained by combining all of these partial projection buffers.

This combination procedure is also parallelized by dividing each partial projection buffer into a few sub-buffers of an equal number of contiguous buffer scanlines, with one sub-buffer per processor. Each processor respectively combines all near and far sub-buffers assigned to it, forming a pair of complete sub-buffers. By the end of this process, we obtain a pair of complete projection buffers Z_n and Z_f. Since the comparison and assignment operations performed during this process are very fast, computation overhead of the combination in our algorithm is very low.

Evidently, there is another solution to avoid memory access conflict without creating and combining each pair of partial projection buffers. All processors simultaneously access the shared projection buffers Z_n and Z_f during the parallelized projection procedure in an exclusive mode. Although the implementation is simpler, buffer access time may slightly increase due to the exclusive access mode.

3.2 Image-Based Partitioning for Ray Traversal

In our algorithm, the projection buffers not only provide closer bounds on the intervals where the ray integrals need to be calculated, but also view-dependent information of image complexity. A static image-based contiguous partitioning is therefore a natural choice.

Note that the amount of computation involved in a specific image section can be calculated by the following formula:

$$\sum_{i=1}^{m} g(d_i) \tag{7}$$

where m is the number of viable pixels in that image section, d_i is the length of a bounded ray interval associated with the ith viable pixel, and g is a function of length d_i. The value of $g(d_i)$ depends on both length d_i and the transparency property of the object to be rendered. Generally speaking, the more transparent the object and the greater the value d_i, the larger the value $g(d_i)$. The value of $g(d_i)$ can be adjusted during rendering to be more suitable for the object, according to the load balancing feedback.

Once function g in Equation 7 is determined, the image is divided into large image blocks of contiguous image scanlines. Each block contains roughly an equal amount of work (see Figure 2). The fundamental unit of work in our algorithm is an image pixel rather than an image scanline, which supports more accurate partitioning and hence better load balancing. Each processor then takes one image block, casts rays from the viable pixels in that block, and performs ray integrals within the bounded interval along each ray.

Our parallel ray traversal procedure is further accelerated by existing ray casting optimizations, which fall into two classes according to whether or not there are computational dependencies between rays. Those non ray-dependent optimizations, such as early ray termination, can be directly applied with an image-based partitioning. However, when incorporating the adaptive image sampling which belongs to the ray-dependent class, caution must be taken to avoid the cost of replicated ray casting from pixels shared by different processors.

In a serial ray casting algorithm, adaptive image sampling optimization [12] is performed by dividing the image plane into fixed size square image tiles of $\omega \times \omega$ pixels, and casting rays only from the four corner pixels of each tile. Additional rays are cast only in those image tiles with high image complexity, as measured by the color difference of corner pixels of the image tiles. All non ray-casting pixels are then bilinearly interpolated from ray-casting pixels. Nieh and Levoy [1] proposed a dynamic image-based partitioning scheme which reduces the cost associated with pixel sharing by delaying the evaluation of image tiles whose pixel values are being computed by other processors.

In this paper, we propose a more effective solution based on our static image-based contiguous partitioning. Compared to the previous dynamic image partitioning scheme [1], our method has no task redistribution overhead and thus fewer synchronization requirements. The method is described as follows:

1. A small fixed size square image tile of $\omega \times \omega$ pixels is defined as the fundamental unit of work.

2. For P processors, the image is split into P large image blocks of contiguous scanlines of tiles. Each block may not contain the same number of tiles, but contains a roughly equal amount of work.

3. Each processor takes one image block and performs adaptive image sampling on each tile in that block top down in scanline order. Note that for all shared pixels at the *bottom* of the image block, the processor directly gets their values from the shared memory, which have been computed by other processors.

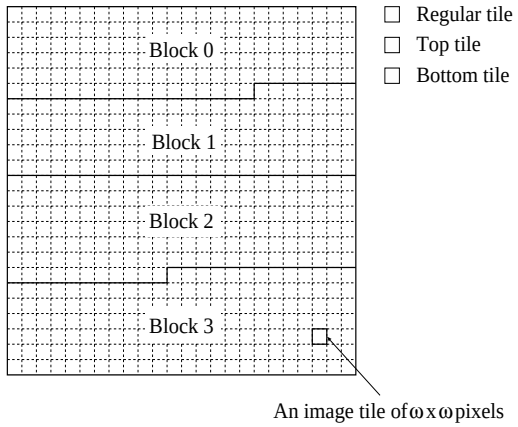


Figure 2: *A static image-based contiguous partitioning example.*

Figure 2 illustrates a four-processor example of how the image is partitioned in our algorithm, where the fundamental unit of work becomes a square image tile. All square image tiles which contain the shared pixels by different processors are marked with dark shading. They are gathered at the top and bottom of each image block. Therefore, tiles in each image block can be classified as: regular tiles (white tiles in the figure), top tiles (tiles with dark shading), and bottom tiles (with light shading). In our parallel algorithm, each processor P_i starts its work from the top tiles down in its image block B_i in tile scanline order. For all the top and regular tiles, normal adaptive image sampling are performed. However, for each bottom tile, we read their values directly from the shared memory. Therefore, replicated ray casting and interpolation operations at shared pixels are avoided.

Evidently, our algorithm works based on the premise that each processor has approximately an equal amount of work to do. To guarantee and test that no computation on the shared pixels is missed by any of the processors, we set an "alarm" signal s for each shared pixel at the top tiles, with initial values 1. Once a shared pixel has been evaluated, its signal s is set to 0. If a processor reaches a shared pixel with value 1 at its bottom tiles, the alarm sounds to notify the user and stop the rendering. Our experimental results have shown that our algorithm works well, and no alarm has sounded so far.

It is possible that when more processors are used, we will some day hear the alarm. When this happens, we would like to employ a dynamic task stealing based on the above contiguous image partitioning scheme. The dynamic scheme would produce better load balancing, but also increase the synchronization overhead and implementation complexity. We should realize, however, that when more processors are available, the trend will be to render much larger volume data sets with larger images. Then, the number of processors would be still significantly lower than the number of pixels, and thus our algorithm would remain competitive.

4 Experimental Results

Our algorithm has been implemented on an SGI Power Challenge with 16 R10000 processors. The performance results on classified volume data sets are given in Table 1 and 2. The brain data set in Table 1 is a $256 \times 256 \times 124$ MRI scan of a human brain (Figure 3). The head data set in Table 2 is a $256 \times 256 \times 225$ CT scan of a human head (Figure 4). Rendering times include both the boundary cell projection time and the subsequent ray traversal time, but not the off-line preprocessing time for boundary cell detection and run-length encoding. Preprocessing times are respectively 9.9 seconds and 18.3 seconds on a single processor for these two data sets. We would like to point out that when the projection procedure is parallelized on a multiprocessor, extra time is needed to combine all partial buffers generated by the different processors. However, our experiments have shown that combination times with buffer size 256^2 are negligible – less than our minimum measurable time (0.01 seconds).

In our preprocessing stage, we merged every eight neighboring grid cells into one macrocell to reduce the amount of boundary cells. Then, in the boundary cell projection procedure, we used low level-of-accuracy projection templates and run-length encoding data structure for both parallel and perspective projections. In the subsequent ray traversal procedure, we performed resampling (using trilinear interpolation), shading (using Phong model with one light source), and compositing within each bounded ray interval through the original volume data. The resultant images contain 256×256 pixels. We selected an early-ray-termination opacity cutoff of 95%. Ray traversal time with both adaptive and nonadaptive (normal) image sampling were measured. In adaptive image sampling, we used square image tiles of 3×3 pixels along with a minimum color difference of 25, measured as Euclidean distance in RGB ($256 \times 256 \times 256$) space. The fastest rendering rates for both data sets were above 20 Hz, among the fastest reported.

Table 1: *Volume rendering times (in sec) for a* $256 \times 256 \times 124$ *MRI brain data set.*

Processors	1	4	8	12	16
Parallel Projection	0.16	0.04	0.02	0.01	0.01
Perspective Projection	0.49	0.12	0.06	0.04	0.03
Nonadaptive Traversal	1.32	0.34	0.17	0.12	0.09
Adaptive Ray Traversal	0.53	0.14	0.07	0.05	0.03
Best Frame Rate (Hz)	1.4	5.5	11.1	16.6	25.0

Table 2: *Volume rendering times (in sec) for a* $256 \times 256 \times 225$ *CT head data set.*

Processors	1	4	8	12	16
Parallel Projection	0.20	0.05	0.02	0.01	0.01
Perspective Projection	0.86	0.22	0.11	0.07	0.05
Nonadaptive Traversal	0.51	0.14	0.07	0.05	0.04
Adaptive Ray Traversal	0.27	0.07	0.04	0.03	0.02
Best Frame Rate (Hz)	2.1	8.3	16.6	25.0	33.3

Our experimental results have shown that the perspective boundary cell projection times are about three to five times longer than parallel boundary cell projection times, depending on the number of boundary cells to be projected. However, subsequent ray traversal times for both perspective and parallel views of the same volumetric object are very close when the projected object has similar sizes on the projection plane. Therefore, the resultant perspective rendering times (including both projection and ray traversal times) are less than three times longer than corresponding parallel rendering times.

Figure 5 shows the speedup curves for both nonadaptive and adaptive renderings (including the boundary cell projection time) on the MRI brain data set with parallel projection. The speedup results on the CT head data set are similar. There are two observations from these speedup curves. First, our parallel program scales well on a multiprocessor. These near linear speedups are ascribed to our effective contiguous object- and image-based partitioning schemes, which lead to both spatial locality and good load balancing. In our boundary cell projection procedure, the computation work assigned to each processor is a subvolume of contiguous run-length encoded scanlines of boundary cells, and therefore provides good spatial locality. With such good spatial locality, we can effectively make use of the prefetching effect of long cache lines on the Challenge, which helps to mask the latency of main memory accesses. In fact, the two medical data sets in Tables 1 and 2 have significant coherence. With the opacity transfer functions we used, 3.1% and 3.2% of the grid cells in the MRI and CT data sets are boundary cells. Accordingly, the run-length encodings of the boundary cells are very small compared to the original volume. When such short run-length encodings are split and assigned to the multiprocessors, they can be easily fixed inside the local caches of these processors with minimal cache misses. Evidently, our ray traversal procedure also benefits from spatial locality provided by our contiguous image-based partitioning, since adjacent rays access data from the same cache line.

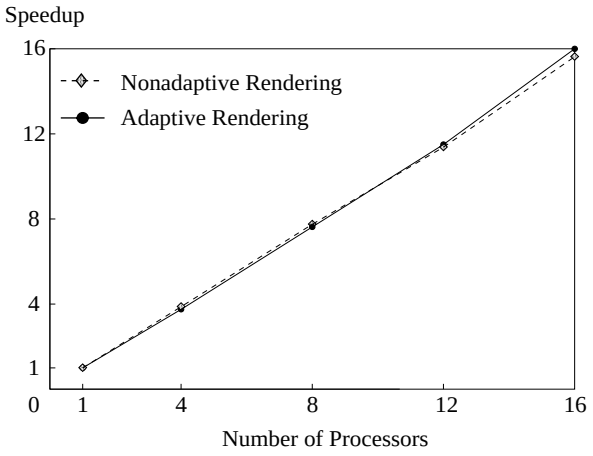


Figure 5: *Speedups of rendering the MRI brain data set on the Challenge.*

The second observation is that the speedups for adaptive rendering are nearly as good as those for nonadaptive rendering. Unlike the results reported by Nieh and Levoy [1] — where adaptive rendering always exhibits worse speedups than nonadaptive rendering, due to extra memory and synchronization overhead — our parallel algorithm shows more efficient adaptive rendering. In their algorithm, memory overhead is larger for adaptive rendering because access to additional shared writable data structures such as the local wait queue is not needed in nonadaptive rendering. Additional synchronization time is also required for the adaptive case, due to the waiting for all processors to complete ray casting before non ray-casting pixels are interpolated from ray-casting pixels. However, in our algorithm, neither the additional shared writable data structure nor additional synchronization time is needed for adaptive rendering. This is because the cost of replicated ray casting is avoided by our load balancing image partitioning scheme without dynamic task redistribution.

To show the performance of our load balancing schemes, we collected the times of both parallel projection and subsequent nonadaptive and adaptive ray traversal procedures on each processor during the rendering of the MRI brain data set using twelve processors. Ta-

Table 3: *Computation distribution (in sec) of the MRI brain data set on* 12 *processors.*

Procedures	Projection	Nonadaptive RT	Adaptive RT
Min-Max	$0.01 - 0.01$	$0.11 - 0.12$	$0.05 - 0.05$
Variation	0.00	0.01	0.00

ble 3 shows that the variations in rendering times among processors for adaptive and nonadaptive ray traversal are respectively zero and 0.01 seconds. A good load balancing was also reached during the boundary cell projection with no measurable variation in projection times among processors. Note that we present load balancing performance on 12 rather than 16 processors of our Challenge. This is because when more processors are used, the projection times are too short (often less than 0.01 seconds) for the purposes of comparison. Also, in Tables 4–6, we present the rendering rates for several data sets for up to 12 processors (Proc#).

Table 4: *Volume rendering times (in sec) for a positive potential of a high potential iron protein data set. (Proj: projection time; Ray: ray traversal time; L: overview with lower opacity threshold* 10*; H: interior with higher opacity threshold* 120*.)*

Proc#	Interactive Classification			Fixed Classification	
	Proj(L)	Ray(L)	Ray(H)	Proj(H)	Ray(H)
1	0.07	0.39	0.29	0.03	0.17
4	0.02	0.12	0.10	0.01	0.05
8	0.01	0.06	0.05	0.01	0.03
12	0.01	0.04	0.04	0.01	0.02

In Table 4, a commonly used 66^3 voxel positive potential of a high potential iron protein was rendered by using modified opacity transfer functions with different opacity thresholds between frames. The slow preprocessing stage for run-length encoding was avoided in our algorithm, provided that new opacity thresholds were never less than the initially specified opacity threshold. We set the initial opacity threshold to 10 (Figure 6a), and the modified threshold to 120 (Figure 6b). The ray traversal times did not increase with the modification. Projection times did not change since we did not change the view. Therefore, interactive rendering rates were maintained during rendering with interactive classification. Similar results are shown in Table 5, where a $320 \times 320 \times 34$ CT scan of a lobster was rendered with interactive classification. The initial opacity threshold was set to 30 to display the semi-transparent shell (Figure 7a). The new opacity threshold was set to 90 to display the meat without the shell (Figure 7b).

Table 5: *Volume rendering times (in sec) for a lobster data set. (Proj: projection time; Ray: ray traversal time; L: shell with low opacity threshold* 30*; H: meat with high opacity threshold* 90*.)*

Proc#	Interactive Classification			Fixed Classification	
	Proj(L)	Ray(L)	Ray(H)	Proj(H)	Ray(H)
1	0.11	0.47	0.41	0.04	0.24
4	0.03	0.13	0.12	0.01	0.07
8	0.02	0.07	0.06	0.01	0.04
12	0.02	0.04	0.04	0.01	0.02

For comparison purposes, we also rendered these two data sets with fixed classification. For each data set, we first recreated the run-length encoding data structure with the new boundary cells according to the modified opacity threshold. Then, we rendered the data set at the same view by using the modified transfer function and the new run-length encoding data structure. Tables 4 and 5 show that rendering times with interactive classification are almost twice as long as those with fixed classification. We also measured the different number of boundary cells with different opacity thresholds, and found that the number of boundary cells corresponding to the modified opacity thresholds was about two thirds of that corresponding to the initial opacity thresholds for these two data sets. It follows that the performance penalty in both rendering rate and memory space is moderate for interactive classification.

Table 6: *Volume rendering times (in sec) for a voxelized F15 aircraft data set using different kinds of multiresolution volumes for object boundary estimation. (Para; parallel projection time; Pers: perspective projection time; Ray: ray traversal time.)*

Proc#	Shrunken Volume			Low LOD Volume		
	Para	Pers	Ray	Para	Pers	Ray
1	0.13	0.35	0.22	0.07	0.17	0.14
4	0.04	0.11	0.06	0.02	0.04	0.04
8	0.02	0.06	0.03	0.01	0.02	0.02
12	0.01	0.04	0.02	0.00	0.01	0.01

In order to further speedup the boundary cell projection time for larger data sets, we used a lower resolution volume during the procedures of boundary cell detection, run-length encoding, and projection. We still used the original high resolution volume for accurate rendering. Such a lower resolution volume can be either a shrunken volume generated from the original volume by merging every m^3 neighboring cells into a macrocell, or a low LOD volume. We conducted some experiments on a rendering of a $186 \times 256 \times 76$ voxelized F15 aircraft data set (Figure 8a). We separately used two run-length encodings created from a shrunken volume (m=2) and a low LOD volume. The low LOD volume had $93 \times 128 \times 41$ voxels (see Figure 8b). The object modeling algorithm [14] we used guaranteed that the shape of the aircraft in the low LOD volume was not "thinner" than that in the original high resolution volume (as shown in Figure 8). Table 6 shows that both projection and ray traversal times from using run-length encoding of the low LOD volume are faster than those of the shrunken volume. We discovered that even though fewer boundary cells were contained in the low LOD volume, they led to a more accurate object boundary estimation, and, therefore, more time savings in both projection and ray traversal procedures. Also note that although we have employed a lighting model and 3D texture mappings (both implemented in software during the rendering time), the ray traversal speeds are very fast. This is because the binary classification of the aircraft decreased the rendering complexity to nearly $O(n^2)$, for an image size of n^2.

5 Comparison to Shear-Warp

The shear-warp factorization technique [5] is another fast volume rendering method, which has several similarities to our algorithm. The comparison between these two is helpful in evaluating ours.

First, both methods are high speed volume rendering algorithms without graphics hardware acceleration. Their high performances are reached by combining the advantages of image- and object-order approaches, and are therefore scalable. Rendering rates as fast as 10–30Hz are reported for both methods to render the same 256^3 volume data set on the 16-processor SGI Challenge. While our method inherits a high image quality from accurate ray casting, the shear-warp method suffers from some image quality problems due to its two-pass resampling and the 2D rather than 3D interpolation filter (as reported in [5]),

Second, the theoretical fundamentals of both methods are directly or indirectly based on the normal ray casting algorithm [13].

Our method directly speeds up the ray casting algorithm by efficiently skipping over empty space outside the classified object without affecting image quality. Thus, existing ray casting optimizations, such as early ray termination and adaptive image sampling, can be conveniently incorporated into our algorithm. The shear-warp method can also be viewed as a special form of ray casting, where sheared "rays" are cast from voxels in the principal face of the volume. Bilinear rather than trilinear interpolation operations are used on each voxel slice to resample volume data (which shortens rendering time, and also reduces image quality). The effect of early ray termination is also achieved.

Third, both methods employ the scanline-based run-length encoding data structure to encode spatial coherence in the volume for high data compression and low access time. In our algorithm, the small number of boundary cells compared to the volume size leads to minimal extra memory space for run-length encoding. Obviously, we still need the original volume during the ray traversal procedure. In the shear-warp method, although three encoded volumes are required along the three volume axes, the total memory occupation is reported to be much smaller than the original volume.

Fourth, both methods support interactive classification, with similar moderate performance penalties. In our method, interactive classification performs without extra programming efforts, providing the modified opacity threshold is never less than the initial opacity threshold. In the shear-warp method, a more sophisticated solution is presented with some other restrictions.

Fifth, both methods are parallelized on shared memory multiprocessors and show good load balancing. Dynamic interleaved partitioning scheme is employed in the parallel shear-warp algorithm [3], while static contiguous partitioning schemes are used in our method. Both methods exploit spatial locality in the run-length encoding data structure. In general, our contiguous partitioning of the volume provides higher spatial locality than interleaved partitioning. Also, compared to dynamic partitioning, our static scheme is more economical due to a simplified controlling mechanism and lower synchronization overhead.

We have compared the performance of parallelized shear-warp algorithm reported by Lacroute [3] with our experimental results, for the same $256 \times 256 \times 225$ voxel CT head data set, achieved on the Challenge with 16 processors. The fastest shear-warp rendering rate is 13 Hz for a 256×256 grey scale image with parallel projection. The rendering time doubles for a color image because of additional resampling for the two extra color channels. We reached a rendering rate of 20 Hz (or 33 Hz, when adopting adaptive image sampling) for color images of the same size, as shown in Figure 4.

6 Conclusions

We have proposed an interactive parallel volume rendering algorithm without using graphics hardware accelerators. It is capable of rendering $10 - 30$ frames per second on a 16-processor SGI Power Challenge for 256^3 volume data sets. We achieved these speeds by using an accelerated ray casting algorithm with effective space leaping and other available optimizations, and contiguous task partitioning schemes which take full advantage of optimizations in the serial algorithm with high load balancing and low synchronization overhead. When compared with the shear-warp approach, our method has shown both faster rendering speed and higher image quality.

Following the encouraging experimental results, we are currently investigating interactive ray casting for very large data sets with our algorithm. Run-length encoding of lower levels-of-detail volume data are being studied to create a near accurate object boundary estimation with much fewer boundary cells. The original full resolution data set will be utilized during ray traversal procedure for high-quality ray casting images.

Acknowledgments

This work has been partially supported by NASA grant NCC25231, NSF grant MIP9527694, ONR grant N000149710402, NRL grant N00014961G015, and NIH grant CA79180. Thanks to Huamin Qu, Kevin Kreeger, Lichan Hong, and Shigeru Muraki for their constructive suggestions and to Kathleen McConnell for comments. Special thanks to Milos Sramek for providing the multiresolution volumes of the F15 aircraft. The MRI data set is courtesy of the Electrotechnical Laboratory (ETL), Japan.

References

[1] J. Nieh and M. Levoy. "Volume Rendering on Scalable Shared-Memory MIMD Architectures". *Proc. 1992 Workshop on Volume Visualization*, 1992, 17-24.

[2] C. Silva and A. Kaufman. "Parallel Performance Measures for Volume Ray Casting". *Proc. IEEE Visualization '94*, 196-203.

[3] P. Lacroute. "Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization". *Proc. Parallel Rendering Symposium*, 1995, 15-22.

[4] S. Parker, P. Shirley, Y. Livnat, C. Hansen, P. Sloan. "Interactive Ray Tracing for Isosurface Rendering". *Proc. IEEE Visualization '98*, 1998, 233-238.

[5] P. Lacroute and M. Levoy. "Fast Volume Rendering using a Shear-warp Factorization of the Viewing Transformation". *Proc. SIGGRAPH '94*, 1994, 451-457.

[6] M. Levoy. "Efficient Ray Tracing of Volume Data". *ACM Transactions on Graphics*, 9(3), 1990, 245-261.

[7] K. Subramanian and D. Fussell. "Applying Space Subdivision Techniques to Volume Rendering". *Proc. IEEE Visualization '90*, 1992, 150-158.

[8] M. Wan, S. Bryson, and A. Kaufman . "Boundary Cell-Based Acceleration for Volume Ray Casting". *Computers & Graphics*, 22(6), 1998, 715-721.

[9] R. Avila, L. Sobierajski, and A. Kaufman. "Towards a Comprehensive Volume Visualization System". *Proc. IEEE Visualization '92*, 1992, 13-20.

[10] C. Montani and R. Scopigno. "Rendering Volumetric Data Using the STICK Representation Scheme". *Proc. Workshop on Volume Visualization*, 1990, 87-93.

[11] M. Wan, Q. Tang, A. Kaufman, Z. Liang, and M. Wax. "Volume Rendering Based Interactive Navigation within the Human Colon". *Proc. IEEE Visualization' 99*, 1999. (in these proceedings)

[12] M. Levoy. "Volume Rendering by Adaptive Refinement". *The Visual Computer*, 6(1), 1990, 2-7.

[13] M. Levoy. "Display of Surface from Volume Data". *IEEE Computer Graphics and Applications*, 8(5), 1988, 29-37.

[14] Milos Sramek and Arie Kaufman. "Object Voxelization by Filtering". *Proc. Symposium on Volume Visualization '98*, 1998, 111-118.

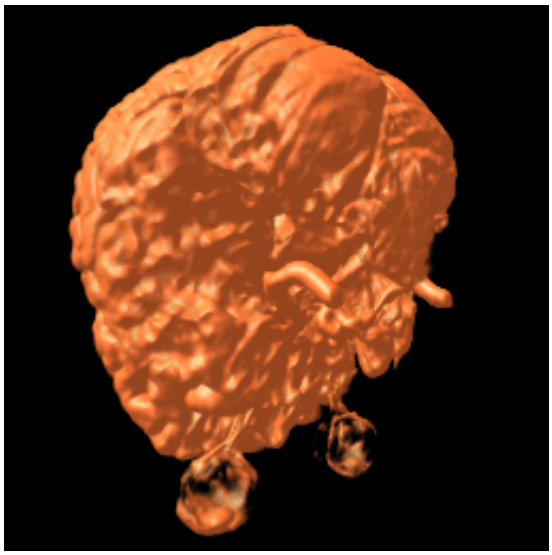


Figure 3: ***Volume rendering with perspective projection of a*** $256 \times 256 \times 124$ ***MRI brain.***

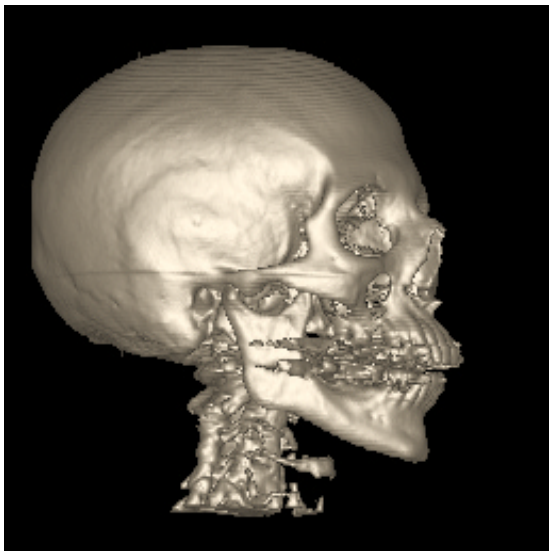
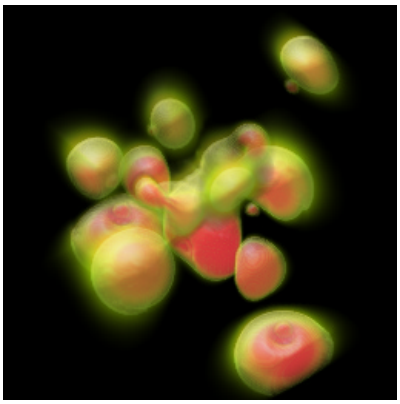


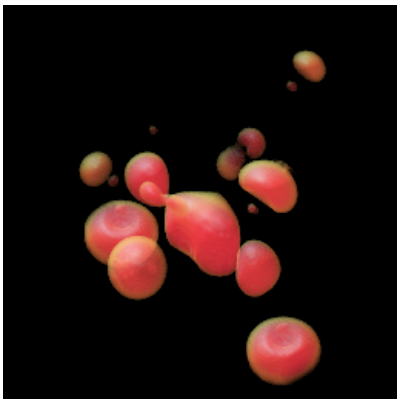
Figure 4: ***Volume rendering with parallel projection of a*** $256 \times 256 \times 225$ ***CT head.***



(a) Initial opacity threshold is 10

(a) Shell with opacity threshold 30

(a) High resolution volume



(b) Modified threshold is 100

Figure 6: ***Volume rendering with parallel projection of a positive high potential iron protein data set using interactive classification.***

(b) Meat with opacity threshold 90

Figure 7: ***Volume rendering with parallel projection of a lobster data set using interactive classification.***

(b) Lower LOD volume

Figure 8: ***Volume rendering with perspective projection of voxelized F15 aircraft data sets using multiresolution volumes.***