Image-Space Visibility Ordering for Cell Projection Volume Rendering of Unstructured Data

Richard Cook, Nelson Max, *Member*, *IEEE Computer Society*,
Cláudio T. Silva, *Member*, *IEEE*, and Peter L. Williams, *Member*, *IEEE Computer Society*

Abstract—Projection methods for volume rendering unstructured data work by projecting, in visibility order, the polyhedral cells of the mesh onto the image plane, and incrementally compositing each cell's color and opacity into the final image. Normally, such methods require an algorithm to determine a visibility order of the cells. The Meshed Polyhedra Visibility Order (MPVO) algorithm can provide such an order for convex meshes by considering the implications of local ordering relations between cells sharing a common face. However, in nonconvex meshes, one must also consider ordering relations along viewing rays which cross empty space between cells. In order to include these relations, the algorithm described in this paper, the scanning exact meshed polyhedra visibility ordering (SXMPVO) algorithm, scan-converts the exterior faces of the mesh and saves the ray-face intersections in an A-Buffer data structure which is then used for retrieving the extra ordering relations. The image which SXMPVO produces is the same as would be produced by ordering the cells exactly, even though SXMPVO does not compute an exact visibility ordering. This is because the image resolution used for computing the visibility ordering relations is the same as that which is used for the actual volume rendering and we choose our A-Buffer rays at the same sample points that are used to establish a polygon's pixel coverage during hardware scan conversion. Thus, the algorithm is image-space correct. The SXMPVO algorithm has several desirable features; among them are speed, simplicity of implementation, and no extra (i.e., with respect to MPVO) preprocessing.

Index Terms—Volume rendering, visibility ordering, unstructured mesh.

1 Introduction

In this paper, we study the problem of cell sorting for volume rendering unstructured volumetric data. In volume rendering, the 3D scalar field to be visualized is modeled as a cloud-like material which both attenuates light along the viewing ray and adds light into it [14]. To create an image, the effects of the material must be integrated along the viewing ray through each pixel. This requires a separate integral for the contribution along the ray segment inside each cell. See Fig. 1. If the order of these segments is known, these contributions can be accumulated using front-to-back or back-to-front compositing.

Instead of separately computing and sorting the segments along each ray, it is potentially more efficient to compute a global visibility sort of the cells and use cell projection to find the contribution of all ray segments in a cell. Then, each cell is considered only once and we can use efficient software or hardware scan conversion and compositing methods.

Thus, volume rendering a set of cells has two phases. One, which we refer to as the *rendering phase*, has to do with the sampling and integration of the cells, i.e., computing, for each ray, the intersection segments generated and their potential contribution to the screen. The other, which we call the *sorting phase*, is the sorting of the different cell contributions so that they can be composited in the correct order (see Fig. 2). Our paper deals with the sorting phase of unstructured volume rendering. (Exceptional situations where sorting is not needed include maximum intensity projection and X-ray-like projections that accumulate only total opacity, the color of the entire volume being the same.) In the parallel volume rendering system described in [1], the rendering phase is done in parallel using graphics hardware; however, the sorting phase needs to be done in serial, in software, and, so, sorting can be a bottleneck.

When a mesh is rectilinear, generating a visibility order of its cells is relatively straightforward. However, most of the data sets we work with are generated by finite element method simulations and these solution sets are usually defined on curvilinear or unstructured meshes, often with different element (cell) types in the same mesh. When the mesh is unstructured, computing a visibility ordering of the cells is a nontrivial problem. The problem is further compounded because finite element meshes may be disconnected, be nonconvex, or have cells with nonplanar faces.

One way to deal with general meshes is to resample them into rectilinear meshes, for which more straightforward sorting and volume rendering methods exist, several of which, e.g., 3D texture mapping, are currently done in hardware. But, resampling can blur the data, miss certain features, and/or greatly increase the size of the data set.

- *R. Cook, N. Max, and P.L. Williams are with the Lawtence Livermore National Laboratory, 7000 East Ave., Livermore, CA 94550. E-mail: rcook@llnl.gov.*
- *C.T. Silva is with the School of Computing, Scientific Computing and Imaging (SCI) Institute, University of Utah, Salt Lake City, UT 84112. E-mail: csilva@cs.utah.edu.*

Manuscript received 9 Sept. 2003; accepted 29 Jan. 2004.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-0078-0903.

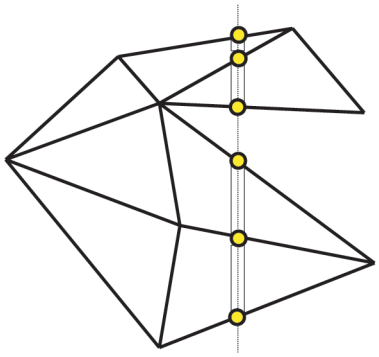

Fig. 1. Computing the volume rendering integral for a pixel requires the ray-tetrahedron intersection segments in sorted order.

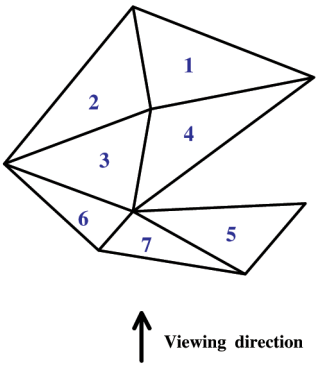

Fig. 2. A back-to-front ordering of the cells of a mesh. Note that a visibility ordering can be somewhat unintuitive. For instance, note that cell 3 comes before cell 4 in a correct visibility ordering, although the centroid of cell 4 is farther from the viewer than the centroid of cell 3. This same error could occur if the sort was based on the power distance. This is the reason that simple schemes, such as power distance sorts or centroid-based sorts, may fail from certain viewpoints or viewing directions.

Here, we only discuss methods that keep the original unstructured mesh.

The remainder of this paper is organized as follows: In Section 2, we provide some definitions and then, in Section 3, we discuss related work. In Section 4, we describe our scanning exact meshed polyhedra visibility ordering (SXMPVO) algorithm. In Section 5, we present a detailed analysis of the performance of the SXMPVO algorithm, including extensive timing results. Finally, in Section 6, we discuss future work and give our conclusions.

2 Preliminary Definitions

To provide a formal basis, we start with a few definitions, some slightly modified from [27]. The *viewpoint vp* is some point in three-dimensional space representing the viewer or camera position. The *occludes* relation is defined as follows: Let c_1 and c_2 be two distinct cells of a mesh S and $\mathrm{int}(c_1)$ and $\mathrm{int}(c_2)$ be the interiors of c_1 and c_2. Relative to viewpoint vp, c_2 occludes c_1 if there is a half-line hl starting at vp and points p_1 in $hl \cap \mathrm{int}(c_1)$ and p_2 in $hl \cap \mathrm{int}(c_2)$ so that p_2 lies between vp and p_1 on hl. A visibility ordering can be defined in the following way: For a given viewpoint, if cell a occludes cell b, then cell a must come after b in the visibility ordering. Note that, if the viewpoint remains fixed and a subset of the cells of interest is selected, the visibility ordering restricted to this subset is still valid and does not need to be recomputed. Projecting and compositing the cells of a mesh in visibility order (back-to-front order) results in a correct volume rendering of the image.

We also define the *behind* relation $<_{vp}$ such that $c_1 <_{vp} c_2$ if and only if c_2 occludes c_1. Note that the set of *behind* relationships for pairs of cells in the mesh represents only a partial ordering of the cells, so many correct visibility orderings may exist consistent with this partial order. A *visibility cycle* is a sequence $a <_{vp} b <_{vp} \cdots <_{vp} c <_{vp} a$ of cells of S. We say S is *acyclic* if, for every viewpoint, no visibility cycles exist.

If a face f of some cell in S is not shared by any other cell in S, then f is an *exterior face*. An *exterior cell* has at least one exterior face. The union of all exterior faces of S constitutes the *boundary* of S. A face that is not an exterior face is an *interior face*, also referred to as a *shared face*. If the boundary of S is also the boundary of the convex hull of S, then S is called a *convex* mesh; otherwise, it is called a *nonconvex* mesh. Two meshes are *disconnected* if they do not share any faces. (Two or more meshes that are disconnected are often referred to as a single mesh with *disconnected components*.) A segment of a ray between a point on an exterior face of a mesh where the ray leaves the mesh and another such point where the ray reenters a mesh we call a *ray-gap*. We refer to a mesh that has cells of different types, e.g., tetrahedra, hexahedra, prisms, and pyramids, as a *zoo mesh*.

For sorting nonconvex meshes, we need to consider visibility relations among exterior faces and this involves their orientation, defined as follows: We use the term *front faces* or *front-facing* to refer to cell faces whose outward normals have a positive component in the direction of the viewpoint, for perspective projections, or a negative dot product with the view direction, for orthogonal projection. Similarly, *back-facing* faces are those whose normals point away from the viewpoint or agree with the view direction.

3 Related Work

In this section, we start with a discussion of several different techniques for implementing the rendering phase of cell projection volume rendering, including the one used for this paper. Next, in Section 3.2, we cover the MPVO algorithm since our algorithm and some other algorithms discussed later are based on it. Then, in Section 3.3, we cover other sorting methods, some of which are integrated with rendering.

3.1 Rendering Phase Techniques

There are a number of different techniques for implementing the rendering phase of unstructured volume rendering. We now mention the techniques which rely on a visibility ordering of the cells of the mesh. Tetrahedra [21], hexahedra [19], [25], and more general cells [1] can all be converted into polygons, triangle strips, or triangle fans for hardware rasterization and fragment processing. Various hardware techniques can then speed up or enhance the integration, along the ray segment for each pixel in the cell's projection. As described in Williams et al. [28], our current rendering system integrates piecewise linear transfer functions, using a 2D texture map, parameterized by opacity and thickness to get the correct exponential per pixel for the opacity. It

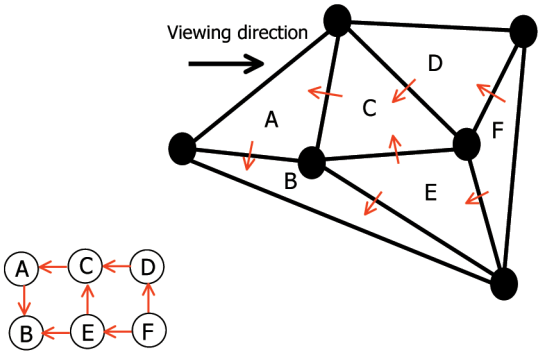

Fig. 3. In the top right, we show a set of cells in a convex mesh and their MPVO arrows marked in red. In the bottom left, we show the resulting directed graph. Cell B is a sink cell.

computes the integral for the color in software at the vertices and at the intersections of projected edges. It interpolates the color in hardware across the polygons bounded by the projected edges and computes these polygons in software for arbitrary polyhedral cells.

Röttger et al. [18] use a 3D texture map, parameterized by front scalar value, back scalar value, and thickness, to look up the volume rendering integral for an arbitrary transfer function, which may include isosurface effects. Wylie et al. [31] show how vertex programs on the nVidia GeForce4 can be used to project an untransformed tetrahedron into a triangle fan by sending the four tetrahedron vertices, plus a fifth phantom vertex for the "thick" vertex at the center of the triangle fan, through the graphics pipeline. Weiler et al. [24] scan-convert, in hardware, just the front faces of a tetrahedron and find the distance to the ray exit point by linearly interpolating the distances to the back faces and take the minimum using a texture map. Since all these hardware techniques depend on a previously determined visibility sort, our sorting method can make them more efficient.

3.2 The Meshed Polyhedra Visibility Ordering (MPVO) Algorithm

One way to compute a visibility ordering of the cells of a convex mesh is to partially order the cells based on adjacency information and the orientation of faces and do a search through the resulting directed graph to determine a correct visibility ordering (see Fig. 3). The MPVO sorting algorithm described by Williams [27] does this. A similar algorithm was also published by Max et al. [15]. Since MPVO is the basis of our new algorithm, we describe its workings here in some detail.

The MPVO algorithm consists of two phases. Phase I is the creation of a partial ordering of the cells by marking each interior (shared) face f with an arrow a such that, if cell c_1 and cell c_2 share f and $c_1 <_{vp} c_2$, then a points from c_1 to c_2. We say a is an *outbound* arrow for c_1. The arrow direction is calculated using the plane equation for the face and the coordinates of the viewpoint, for perspective views, or a vector along the direction of projection, for orthogonal views. The set of arrows across faces can be thought of as the directed edges of a directed adjacency graph, where the nodes of the graph correspond to the cells of the mesh. This graph defines a partial ordering of the cells based on the relationships of cells to their neighbors across shared faces. When a depth-first search (DFS) is used for Phase II, as described below, *sink cells*, cells with no outbound arrows across interior faces, are identified during the traversal of cells which sets the arrows. The sink cells so identified are put on a sink cell list. In addition, each cell is marked with a *visited* flag set to *false*.

Once a partial ordering is determined, Phase II of the MPVO algorithm consists of a topological sort of the directed graph to yield a total ordering, that is, an ordering sequence for the cells so that c_1 comes before c_2 in the sequence whenever there is an edge in the graph for the relation $c_1 <_{vp} c_2$. The total ordering will be generated as long as there are no visibility cycles; otherwise, an error is detected (see below). That total ordering is a correct visibility ordering, provided the mesh and cells are convex, by the following argument: If cell a occludes cell b in a convex mesh, the ray segment between a and b lies inside the mesh and passes without ray-gaps through a sequence of intervening cells $c_1, c_2, \cdots, c_n$. The arrows across the shared faces between a and c_1, and c_1 and c_2, $\cdots$, and c_n and b will add the relations $b <_{vp} c_n, c_n <_{vp} c_{n-1}, \cdots, c_1 <_{vp} a$ to the directed graph. Since the total ordering produced by the topological sort respects each of these relationships, cell b must come before cell a.

Phase II, the topological sort, can be accomplished by a DFS of the directed graph as follows: A cell is removed from the sink cell list and the algorithm RECURSIVE-DFS is performed on that cell. This is done until there are no more cells on the sink cell list. For the purpose of the DFS algorithm, a cell c is defined to be an *eligible neighbor* of cell a if there is an arrow from c to a. RECURSIVE-DFS is defined as follows:

RECURSIVE-DFS(currentCell). *First, set currentCell's* visited *and* cycle detection *flags to* true. *Then, call RECURSIVE-DFS with each eligible neighbor of currentCell that is not* visited. *(If an eligible neighbor is found whose* visited *flag and* cycle detection *flag are both set to* true, *output a visibility cycle warning and exit.) When there are no remaining eligible neighbors, output currentCell for rendering, and set its* cycle detection *flag to* false.

The MPVO sorting algorithm is both fast and accurate: It runs in linear time with low computational overhead and uses linear space for its data structures. It also can detect visibility cycles; if there is a visibility cycle, then no visibility ordering is possible unless one or more of the offending cells is subdivided [16]. However, many data sets are defined on nonconvex and/or disconnected meshes. In such cases, there may be cells that cannot be related by any transitive chain of behind relationships across shared faces and yet which may occlude each other, so an ordering based purely on such relationships is not possible. See Fig. 4.

3.3 Other Sorting Methods

In [27], Williams described the MPVONC heuristic which he developed to extend the MPVO algorithm to nonconvex meshes. The MPVONC heuristic sorts exterior cells that have exterior front-facing faces by decreasing the distance of the cell's centroid to the viewpoint. A depth first search is then conducted from each such exterior cell, taken in sorted order, to construct the total visibility ordering from the

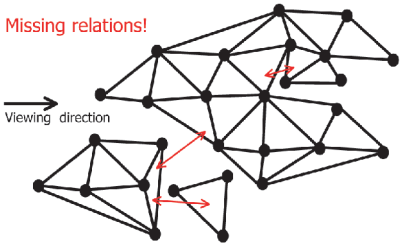

Fig. 4. This figure depicts some missing relations when the MPVO algorithm is applied to a nonconvex mesh.

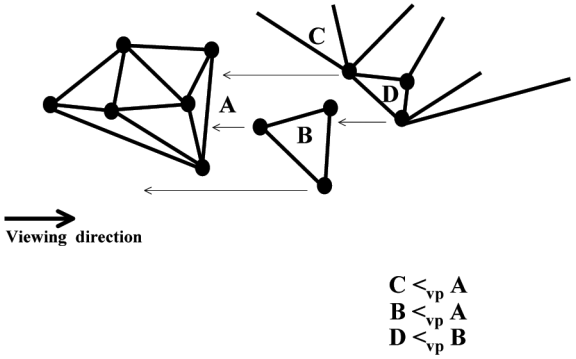

Fig. 5. The XMPVO algorithm adds new relations to the MPVO relations based on ray shooting queries at vertices (and, in 3D, at the intersections of projected edges).

$<_{vp}$ relations. The sorting step effectively and easily augments the MPVO partial ordering described above and, when combined with a topological sort, usually generates a correct visibility ordering and has the advantage that it is very fast. However, MPVONC may fail, for example, in certain cases where one cell occludes another across a region where ray-gaps are possible (see [22], [27] for examples) since it is possible for the occluding cell to have a centroid which is farther from the viewpoint than the centroid of the occluded cell. Fig. 8b shows a volume rendered image with artifacts from sorting errors due to this problem.

Our goal in the research for this paper was to find a sorting algorithm for nonconvex meshes for use in volume rendering that would result in a correct image without any artifacts due to sorting errors and, further, to be able to perform the sort in time comparable to that required for the efficient MPVONC heuristic. As shown in Section 5.2, we were successful in achieving our goal.

An early, exact, and more general method than MPVONC was developed by Stein et al. [23], [28], but runs in $O(n^2)$ time for n cells, which is unacceptable for large data sets. It is possible to reduce this algorithmic complexity by noting that the extra relationships only occur when an exterior face in the mesh occludes another exterior face. Using this fact, Silva et al. [22] developed XMPVO, which used ray shooting queries (see Fig. 5), at each edge intersection and vertex of the projection complex, to discover the exterior face relations and then performed a topological sort, as in MPVO, on the visibility relations found. They thus improved the runtime to $O(n + b^2)$, where b is the number of exterior faces. The XMPVO algorithm relies on being able to compute a visibility ordering of the exterior cells by first performing a sufficient set of ray shooting queries. In XMPVO, the actual ray shooting queries are performed in object-precision, where it is quite hard to avoid performing redundant queries and it is also quite expensive to compute queries. Part of the problem of implementing XMPVO is the fact that it performs ray shooting queries along the intersections of the projections of the edges of cells. Robustly computing such intersections is quite hard and possibly requires the use of exact arithmetic packages (which slows things down considerably). In our new algorithm, the scanning exact meshed polyhedra visibility ordering (SXMPVO) algorithm, described in this paper, we effectively shoot the query rays through the pixel centers, using polygon scan conversion of the exterior faces, and thereby add the exterior face relations only at the precision necessary for the current image.

The BSP-XMPVO algorithm of Comba et al. [6] further improved the XMPVO results to $O(n + bp)$, where p is the size of a small subset of the exterior cells, leading to an order of magnitude improvement in sorting times over XMPVO. This technique requires a view-independent preprocessing which amounts to building a BSP tree of the exterior faces. Building the BSP structure is a very slow step for this algorithm, however. Also, the BSP tree appears to add considerable processing overhead and seems sluggish in some practical cases when the BSP tree becomes deep and unbalanced. This sometimes leads to an $O(b)$ time cost for each of $O(b)$ searches into the tree, thus giving the algorithm an $O(b^2)$ flavor.

Sort algorithms based on the "power distance" have been described by Cignoni et al. [3], [4], [5] and Wittenbrink [29]. However, these are not guaranteed to produce an accurate visibility ordering of the cells of the mesh unless the mesh is a Delaunay triangulation and there is no way to tell in advance whether any given non-Delaunay mesh will be correctly sorted by this method. We would like to note that, for certain meshes and viewpoints, these algorithms do generate correct ordering, and are quite fast since the computations involved have a very small constant and they are straightforward to implement.

In our SXMPVO algorithm, we use an A-Buffer [2] to sort, at each pixel, the exterior faces projecting onto it. As described below, several other variants of the sort per pixel idea have been presented, which instead sort all the faces projecting onto a pixel, not just the exterior faces, as does SXMPVO. These variants do the sort at render time, just prior to the compositing, and they usually spend far greater effort on the per-pixel sorting since, in most meshes, the exterior faces are a small fraction of the total. However, these variants have the advantage of handling visibility cycles gracefully since, for convex nonoverlapping cells, there is always a visibility sort for each pixel center even if there is no global visibility sort of the cells. Kraus and Ertl [12] also deal with visibility cycles in convex meshes, using special hardware compositing methods to scan-convert each cycle. Since they use the basic MPVO sorting algorithm to detect and isolate the cycles, their method could be extended to nonconvex meshes using our SXMPVO algorithm. However, their method has rendering costs per cycle which increase quadratically in the number of cells involved in the cycle.

Wilhelms et al. [26] sort all the cell faces per pixel using methods similar to those in scan line visible surface algorithms, with y-buckets for the first scan line a face crosses, an active list for the faces overlapping the current scan line, x-buckets for the first pixel on the scan line a face covers, and a z-sorted list of faces at the current pixel. Because multiple overlapping meshes are handled, z values must be updated and resorted (or at least the sort must be rechecked for correctness) at each new pixel, taking partial advantage of the coherence in x, but not in y. Since we scan convert each exterior face as a whole, the SXMPVO algorithm can take advantage of coherence in x and y.

King et al. [11] describe tetrahedral primitives, as well as tetrahedral strips and fans, to be rendered by the proposed R-Buffer hardware of Wittenbrink [30]. The R-Buffer implements a method similar to the software algorithm of Mammen [13] to sort per pixel the transparent fragments in front of the front-most opaque one. Current hardware implementations of Mammen's technique require multiple passes through the polygons in the scene [7]. Instead, Wittenbrink [30] proposes to scan-convert all polygons only once and save the not yet composited or rejected fragments in a large unordered recirculating fragment buffer on the graphics card from which the multiple depth comparison passes would be made. The R-Buffer hardware does not yet exist and a software version does even more work than in Wilhelms et al. [26] since the sorts are repeated from scratch independently at each pixel, instead of incrementally updating a sort from the previous pixel to the left. Furthermore, R-Buffer and A-Buffer techniques require substantial amounts of memory over what we need. In our algorithm, the number of fragments we handle is roughly proportional to the projected area, in pixels, of the boundary of the mesh. This contrasts with these other techniques that require memory proportional to the projected area, in pixels, of all the faces in the mesh. We believe that, if built, the R-Buffer will be useful in situations where the Mammen algorithm is currently used, e.g., for rendering glass helmets or windshields in games, but not for volume rendering, for reasons explained in Appendix A.

The ZSWEEP algorithm of Farias et al. [8] is an example of an A-Buffer technique that optimizes for sorting and memory usage. Unfortunately, there appears to be no simple way to map this algorithm into currently available graphics hardware. We point the interested reader to [6], [22], [28] for more complete surveys of previous work on visibility sorting.

4 The SXMPVO Algorithm

For a given viewpoint, the SXMPVO algorithm finds a global ordering of the cells of an unstructured mesh which produces the correct sort of the segments along the viewing ray through each pixel. The boundary of the mesh may be nonconvex and the mesh may have disconnected components. One restriction of our algorithm is that the mesh must be acyclic for a correct ordering to be found. If a cycle exists, our algorithm can be used for reporting it. Furthermore, the mesh may be a zoo mesh. The cells of the mesh may even have nonplanar surfaces, provided the edges do not have projections that cross each other (so-called "bow tie" quadrilateral projections). In [16], the authors show how to selectively subdivide cells into tetrahedra to eliminate the problems caused by nonplanar quadrilateral faces with "bow tie" projections.

Like XMPVO, our new method augments the ordering relationships of MPVO by performing ray-shooting queries among the exterior faces of the mesh. However, instead of computing the extra relations with object-precision (i.e., correct for any ray), our new method performs the necessary ray shooting queries with image-precision (i.e., along rays that pass through the centers of the pixels in a specific view). Furthermore, instead of explicitly computing ray shooting queries, SXMPVO scan-converts the exterior faces, saving the ray-face intersections in an A-Buffer [2] type of data structure that stores all the ordering relations along each ray. The visibility ordering which SXMPVO produces is correct if the image resolution used for the volume rendering is the same as the one used for computing the visibility ordering relations. The following two sections describe the details of the algorithm.

4.1 Creating a Partial Ordering

The SXMPVO algorithm begins by marking all shared faces with arrows exactly as is done in Phase I of the MPVO algorithm, thus determining all *behind* relations, henceforth just referred to as relations, arising from adjacent cells. During this traversal of the cells when the arrows are marked, rather than putting the sink cells on a list, a sink cell flag in each cell is set to either *true* or *false*, as appropriate. In addition to the MPVO data structures, an exterior face list is created to hold a record for each exterior face. This record contains a pointer to the face and to the cell to which the face belongs, the face's centroid, and a next pointer.

Then, the exterior faces are sorted by increasing distance from the screen to the face's centroid. An A-Buffer is created, implemented as an array of pixel (PIX) lists, one per pixel in the screen image. A PIX list consists of a series of PIX list entry records (PIX entries), as described below. The exterior faces are then removed from their sorted queue one at a time and their interiors are scan converted in 3D. (When using OpenGL, this means sampling at pixel centers; see [20].) As each pixel p of an exterior face f is enumerated by the scan conversion, a new PIX entry is created containing the distance z from the screen to the point where the ray through p intersects f, a pointer to f, and a next pointer for the PIX list. The PIX entry is then inserted into the appropriate PIX list in the A-Buffer in order of decreasing z. Since the exterior faces are sorted by their centroids before starting the scan conversion process, the vast majority of PIX entry insertions are at or very near the beginning of the PIX list. Thus, an efficient insertion sort can be used.

When all exterior faces have been rasterized, the PIX list for every screen pixel contains, in order of decreasing z, all the exterior faces that are intersected by a viewing ray through that pixel. Except for the first and last entries on a PIX list, consecutive entries alternate between referring to a front-facing face of a cell and a back-facing face of another cell. The first face in a PIX list is a back-facing face where the ray from the viewpoint exits the mesh for the last time; hence, there is no matching front-facing face behind it and it is discarded. (If the viewpoint is outside the mesh, the PIX lists have an even number of entries, so the last entry on the PIX list, the face through which the ray enters the mesh for the first time, is also not used.) Now, between each front-facing/back-facing face pair on the PIX list, there is a ray-

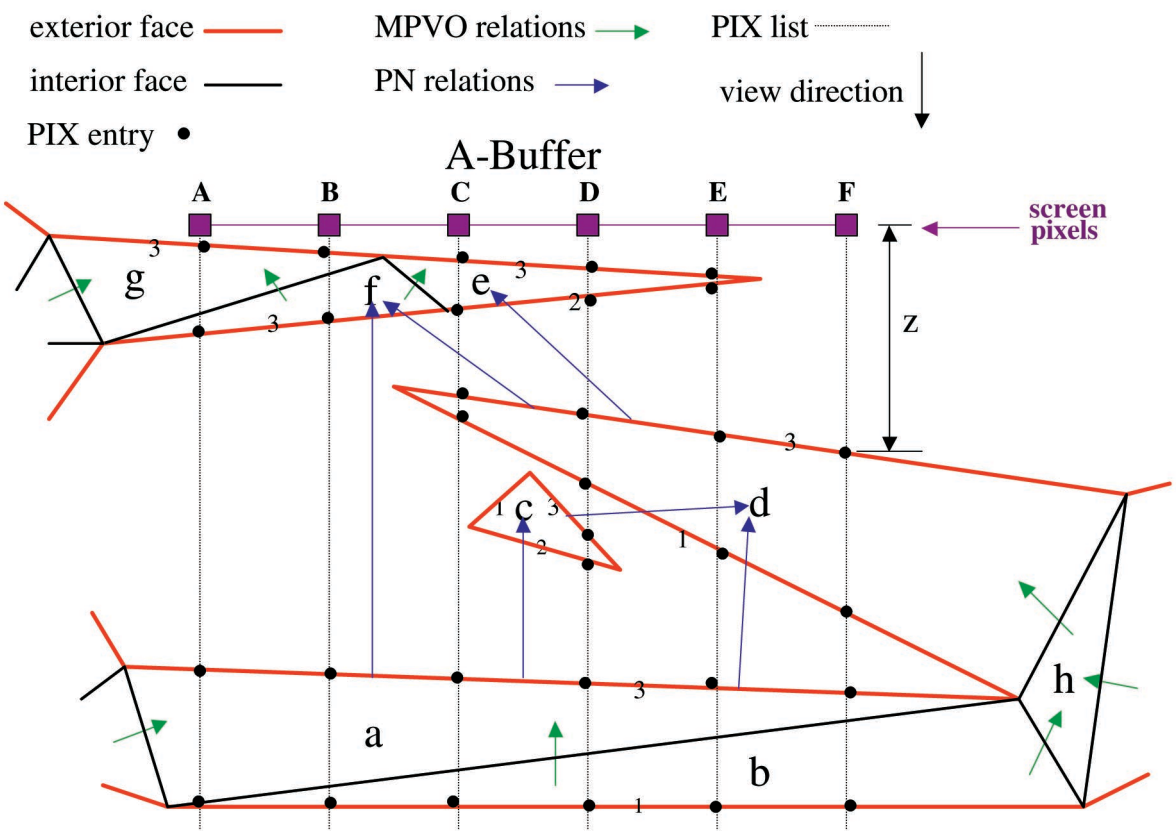

Fig. 6. Diagram of an A-Buffer on a 2D slice across a scan line through an unstructured mesh with disconnected components. The z distance for the PIX entry for face d_3 on the PIX list for pixel F is shown. (The numbers on the exterior faces are just to identify the faces for the purpose of this diagram.) Note here that all insertions of PIX entries into PIX lists occur at the heads of the lists except for the entry into the PIX list for pixel D for face d_1, whose centroid happens to be farther from the screen than the centroids of faces c_2 and c_3. The completed PIX list for pixel D is $\{b_1, a_3, c_2, c_3, d_1, d_3, e_2, e_3\}$, before discarding the first and last entries. The first pair in this list after discarding b_1 is (a_3, c_2), so a and c are PNs. Cell c is the occluding PN and, so, its PN list contains a. The PN and MPVO relations for the relevant portion of this mesh are shown in the diagram. The sink cells on which RECURSIVE-DFS is called are g and e. The DFS algorithm needs pointers which descend against the arrows, e.g., from e to d to c to a to b, and this is why the occluding PN stores the dependency.

gap. We say that the two cells whose faces make up this pair are *pseudoneighbors* (PNs). The PN cell that contributes the back-facing face is called the *occluding PN*, the other cell of the pair is called the *occluded PN*. The desired relations between exterior cells are given by these PN pairs. We refer to these new relations as the *PN relations*. See the diagram in Fig. 6 where PN relations are shown by arrows similar to those used in the MPVO algorithm. The arrows point from the occluded PN to the occluding PN. The PN relations together with the regular MPVO relations are sufficient to generate an exact image space visibility ordering for volume rendering.

Every cell has a PN list pointer, initially set to NULL. When a DFS is used to enumerate the ordering (see Section 4.2), each PN relation is stored as an entry in a PN list maintained by the occluding PN. A PN list entry (PN entry) contains the cell number of the occluded cell and a next pointer. Thus, the PN list for a cell contains all the exterior cells that it occludes, its *dependencies*. So, in Fig. 6, cell d is an occluding PN and its PN list contains cells c and a.

We now return to the details of the SXMPVO algorithm. After completing the PIX lists, the algorithm continues by traversing each PIX list in the A-Buffer. For each pair of entries on a PIX list, a PN entry is added to the PN list of the occluding PN and the sink cell flag of the occluded PN is set to false (the definition of a sink cell is now expanded, from that defined for MPVO, to be a cell that has no outbound arrows and that is not an occluded PN). Duplicate entries are not allowed in PN lists as such entries would cause unnecessary searching later (see Section 5.1). If the PN relation is not already recorded in the appropriate PN list, it is added to the head of the list. Checking whether the PN relation is already recorded in a PN list may require scanning the entire list. However, it is highly likely that, if the entry is already in the PN list, it will be located at or very near the head of the PN list due to the coherence of the scanning process. To further encourage this "found at the head" behavior, whenever an entry is found in the list, that entry is moved to the head of the list if it is not already there. In this way, if this entry is encountered again in another PIX list, which is highly likely due to coherence, it will be found immediately. The combination of coherence and moving entries to the head of the list leads to a nearly constant-time search for duplicates. At the end of this process, all the relations necessary to generate an exact image space visibility ordering for volume rendering are recorded.

4.2 Creating a Total Ordering

The SXMPVO algorithm now completes by executing Phase II of the MPVO algorithm as described in Section 3.2, with two modifications. First, all cells are traversed to find the sink cells, those cells whose sink cell flag is still set. Second, occluded neighbors of a cell c (i.e., the cells on the PN list of

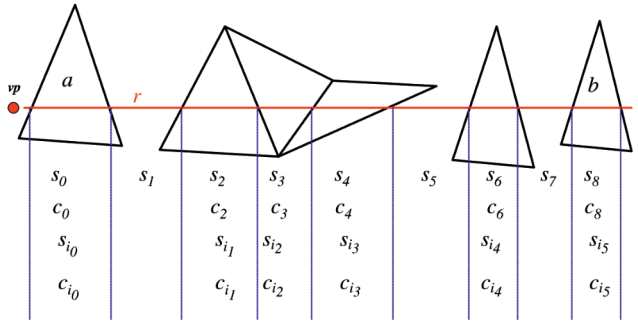

Fig. 7. This figure shows proof of correctness of the SXMPVO algorithm showing an example with four cells and three ray-gaps between cells a and b. In this example, $n = 7$ and $m = 5$. So, for example, for $j = 5$, $c_{i_j} = c_{i_5} = c_8$ and $c_{i_{j-1}} = c_{i_4} = c_6$. The ordering $c_8 < c_6$ is enforced by a PN relation. See Section 4.2 for details.

cell c) are now eligible neighbors of c in RECURSIVE-DFS. The sort generated may not be a strict visibility ordering if faces which overlap are not sampled at points in the region of overlap. However, the visibility ordering will be correct in image space and the volume rendered image will be correct because we took care to sample all faces at actual pixel locations.

To prove correctness, consider a cell a which occludes a cell b along a ray r from the viewpoint vp through the center of a pixel p. We need to show that the SXMPVO algorithm generates a sequence of relations, either PN or MPVO, between a and b which will be respected by the total ordering sequence produced by the DFS algorithm in order to conclude that b comes before a in the sequence. The mesh may now be nonconvex, so we cannot conclude, as in Section 3.2, that there is a sequence of cells that the ray crosses without ray-gaps; ray-gaps may occur where the ray leaves and reenters the mesh. Instead, we can divide the line segment, between the point where the ray exits cell a and the point where it enters cell b, into a sequence of segments $s_1, s_2, \cdots, s_n$, where s_i is either the intersection of the ray with a cell c_i or a ray-gap between cells c_{i-1} and c_{i+1}. Let $c_0 = a$ and $c_{n+1} = b$ and let s_0 and s_{n+1} be the corresponding segments in those cells. See Fig. 7.

Consider the subsequence s_{i_j}, $j = 0, \cdots, m$, of s_i consisting of those ray segments corresponding to a cell instead of a ray-gap and let c_{i_j} be the corresponding sequence of cells, with $c_{i_0} = a$ and $c_{i_m} = b$. Then, for each $j = 1, 2, \cdots, m$, we have a relation $c_{i_j} <_{vp} c_{i_{j-1}}$ in the directed graph. If the two indicies i_{j-1} and i_j are consecutive, with $i_j = i_{j-1} + 1$, then the ray crosses from one cell to the other across a common face and this relation comes from an MPVO arrow. If not, since there are never two consecutive ray-gap segments, $i_j = i_{j-1} + 2$ and there is a ray-gap segment $s_{i_{j-1}+1}$ between cells $c_{i_{j-1}}$ and c_{i_j}. The ray exits a back-facing face of $c_{i_{j-1}}$ and next reenters the mesh through a front-facing face of c_{i_j}. Therefore, $c_{i_{j-1}}$ and c_{i_j} are a front-facing/back-facing pair on the PIX list for pixel p and a PN relation $c_{i_j} <_{vp} c_{i_{j-1}}$ will be added to the directed graph. Thus, in the total ordering sequence, $b = c_{i_m} < c_{i_{m-1}} < \cdots < c_{i_1} < c_{i_0} = a$, so b comes before a in the sorted sequence.

As stated in Section 1, our sorting algorithm is designed to provide a visibility ordering for use with volume rendering systems based on cell projection. For this purpose, our algorithm is guaranteed to provide a correct sort. However, some systems, e.g., [28], combine cell projection with splatting techniques, whereby tiny cells which do not project onto any pixels are splatted, to provide some antialiasing. We discuss this topic further in Appendix B.

TABLE 1
Values of the Parameters Used in the Analysis of the SXMPVO Algorithm for Several Data Sets

Data Set	n	$n/\log_2 n$	b	a	$w * h$
spx	12,936	947	2,760	580,748	420,000
bluntfin	187,395	10,699	13,516	321,791	207,000
f117	240,122	13,435	9,884	770,902	351,000
helix450k	450,000	23,962	29,000	335,731	168,750
helix1.0	1,003,520	50,335	41,472	417,984	210,000
helix1.5	1,485,000	72,432	59,500	232,729	168,750
helix2.4	2,370,000	111,917	67,400	212,567	168,750
helix3.5	3,570,000	164,006	91,400	252,825	168,750
helix4.5	4,500,000	203,606	160,000	224,871	175,175

5 Analysis and Timing Results

The execution of the SXMPVO algorithm described in Sections 4.1 and 4.2 is timed and analyzed in four distinct phases:

I. creating the partial ordering of the interior cells (i.e., computing arrows for each interior face),
II. sorting the exterior faces by centroid,
III. scan converting the exterior faces and recording the PN relations,
IV. finding the sink cells and performing the DFS to complete the visibility ordering.

5.1 Complexity

For this analysis, n is the number of cells in the mesh, b is the number of exterior cells, a is the total area of all the exterior faces as measured by the total number of pixels generated in their scan conversion, and w and h are the image width and height in pixels. The asymptotic times discussed below are, as argued, expected times, not worst-case times.

Phase I is $O(n)$ because every cell in the data set must be examined to mark arrows on its $O(1)$ faces. Note that the plane equations for the faces, needed to determine the arrow directions, are view independent and can be precomputed. Phase II is a simple sort of the exterior faces and is thus $O(b \log b)$. Unless b is asymptotically larger than $n/\log_2 n$, the cost $O(b \log b)$ is only $O(n)$. As can be seen in Table 1, typically, b is less than this threshold.

Phase III is the heart of the algorithm. Rasterizing the exterior faces is $O(a)$. The per-polygon and per-edge set up cost of the scan conversion algorithm is $O(b)$. The scan conversion may also have a cost per scan line. But, the total number of scan lines involved depends on the projected shapes and orientations of the exterior faces, which are difficult to parameterize, so we have neglected this cost. There is an additional potential $O(b^2)$ term necessary due to the insertion of new PIX entries into the PIX lists for each cell during the scanning. Every test we ran showed that insertions into the PIX list occurred at the head of the list more than 99 percent of the time due to presorting the cells by centroid. Thus, these insertions are each approximately

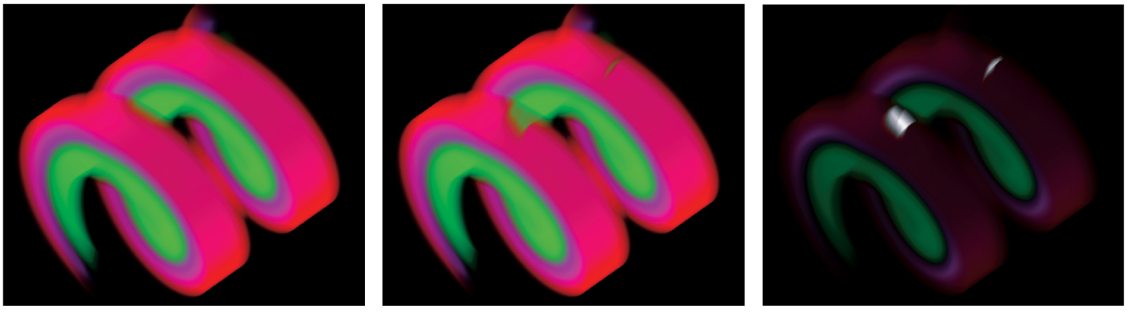

Fig. 8. (a) A volume rendered image of the *helix3.5* data set, with 3.57 million cells, created using the SXMPVO algorithm for sorting. (b) A volume rendered image of the *helix3.5* data set, with 3.57 million cells, created using the MPVONC heuristic for sorting. Note the errors. (c) An image of the difference between the images shown in (a) and (b). The white pixels occur where there are differences in the images. The entire image has been included dimly as a frame of reference.

constant work and the total insertion cost should thus be $O(a)$. There is also a $O(w * h)$ contribution in Phase III to initialize and check each PIX list.

When creating the PN lists from the PIX lists, traversing the PIX lists to discover dependencies is simply $O(a)$. There is also the cost of scanning the PN lists to prevent duplicate entries, which could potentially cost $O(b^2)$ per list. It is possible to reduce the searching cost in the list to $O(\log b)$ by keeping the PN lists sorted. This would lower the insertion cost to $O(b \log b)$ per list. In most cases, coherence and our practice of moving duplicates, when found, to the head of the list reduce this cost to $O(1)$. In the tests shown in Figs. 11, 12, and 13, for a pathological case of many parallel slabs, the cost was proportional to a, so the cost per PIX entry was, in fact, $O(1)$. Thus, Phase III requires $O(w * h + a)$ time for most data sets.

If we were to allow duplicates in the PN list, each cell would have, on average, $O(a/b)$ occluded PNs, and scanning these lists for new dependencies could take $O(a/b)$ time per dependency. Since there are $O(a/b)$ dependencies per list, we thus avoid adding $O((a^2/b^2) * b) = O(a^2/b)$ work by eliminating duplications in the lists.

In Phase IV, we examine each cell to see which are still marked as sink cells, which has a cost of $O(n)$. The DFS algorithm traverses each edge in the directed graph twice, so its cost is proportional to the number of edges. The number of graph edges from shared faces is $O(n)$ and the total number of all PNs, as constructed from scan conversion, is $O(a)$ (and usually much less), so Phase IV is $O(n + a)$. Therefore the overall algorithm requires $O(w * h + a + n)$ time for most data sets. Table 1 shows the values of the parameters n, $(n/\log_2 n)$, b, a, and $w * h$ for several typical unstructured data sets. In the next section, timings are given for these data sets which are consistent with the cost estimate above.

We now discuss storage requirements for SXMPVO. For a tetrahedral mesh, the MPVO data structures require $4f + 14n$ words of storage, where f is the number of interior faces in the mesh and, thus, is $O(n)$. This includes space for the adjacency graph, cell vertices, the plane equation coefficients, the arrows, and one word per cell for flags. For SXMPVO, $O(w * h + a)$ space is required for the A-Buffer, including the PIX lists. The PN lists require an additional $O(a)$ words, plus an extra word per cell (another $O(n)$ words) to store the PN list head pointer, even if it is null. The exterior face list requires $O(b)$ words.

5.2 Test Results

Fig. 8a and Fig. 8b show volume rendered images of the same tetrahedral data set, using the SXMPVO and MPVONC algorithms, respectively. This data set has 3.57 million cells arranged in a helix, with a green inner helix surrounded by a pink outer helix. Note that the SXMPVO algorithm correctly orders data sets upon which MPVONC fails. In Fig. 8b, it can be seen that some cells from the green helix incorrectly occlude some cells from the pink helix. Fig. 8c shows the pixels which differ between these two images. We verified the correctness of our implementation of the SXMPVO algorithm by checking the ordering of the cells by rasterizing them into a software Z-Buffer [28].

We ran detailed performance comparison tests of SXMPVO and MPVONC on an SGI Power Onyx, using a single R10000 250 MHz IP27 processor. For the *helix3.5* data set, shown in Fig. 8a, which has 3.57 million cells, MPVONC requires 7.4 MB to store the vertices, 100.0 MB to store the cells, and 240.0 MB to store the face structures, for a total of 347.4 MB. The additional overhead for SXMPVO is very modest in comparison. For a 375×450 pixel image, the A-Buffer, an array of PIX lists, requires 3.7 MB for the $a = 252,825$ pixels scanned over the 91,400 exterior faces. For an image of this same data set with 1,000,000 pixels, the A-Buffer consumes 22.0 MB, ($a = 1,496,468$.) The PN data structures use 14.5 MB (for the PN list pointers for each cell and 31,084 dependencies) and the exterior face list uses 1.5 MB, for a total of 367 MB.

Table 2 compares times for the SXMPVO and BSP-XMPVO algorithms. Prior to the SXMPVO algorithm, the BSP-XMPVO algorithm was the fastest known algorithm for accurately visibility ordering acyclic unstructured meshes. The times for BSP-XMPVO are from a 333 MHz Power PC as given in [6], whereas the timings for SXMPVO (and MPVONC) are from a slower 250 MHz machine described above. In addition to being faster, the SXMPVO algorithm has the following advantages over the BSP-XMPVO algorithm:

1. A long preprocessing phase is needed to construct the BSP tree,

TABLE 2
Comparative Timings for the BSP-XMPVO and SXMPVO Algorithms

Image Size in Pixels ⟶			170,000	350,000	700,000	1,050,000
Data Set	n	BSP-XMPVO	SXMPVO	SXMPVO	SXMPVO	SXMPVO
bluntfin	187,395	2.5	0.60	0.79	1.12	1.45
f117	240,122	2.9	0.73	0.90	1.23	1.56
helix1.5	1,485,000	n/a	3.92	4.24	4.82	5.27

Prior to the SXMPVO algorithm, the BSP-XMPVO algorithm was the fastest known algorithm for accurate visibility ordering acyclic unstructured meshes. Note that the times for BSP-XMPVO are from a 333 MHz PowerPC, as given in [6], whereas the timings for XMPVCO are from a slower 250 MHz machine, the one described in Section 5.2. As discussed in the text, the BSP-XMPVO algorithm does not appear to be of practical use due to implementation complexity. The change in timing with image size is due almost entirely to Phase III.

TABLE 3
Breakdown of Execution Time, in Seconds, for the SXMPVO Algorithm for Various Data Sets for the Image Sizes Given in Table 1

	Data Set ⟶	*bluntfin*	*helix450k*	*helix1.5*	*helix4.5*
Phase	Number of Cells ⟶	187,395	450,000	1,485,000	4,500,000
I	Create Partial Order	0.23	0.58	1.80	5.28
II	Sort Exterior Faces	0.02	0.05	0.20	0.61
III	Scan Faces and Add Dependencies	0.20	0.31	0.63	0.76
IV	DFS	0.14	0.34	1.29	4.33
	Total Time (seconds)	0.60	1.28	3.92	10.98

2. The SXMPVO algorithm is easier to implement since the implementation of BSP-XMPVO requires the implementation of the BSP tree and the "glue" code described in Fig. 2 of [6],
3. Handling degeneracies in the BSP tree often requires complex code and the use of exact arithmetic,
4. BSP trees for complex boundaries can get large and consume substantial memory, and
5. The performance of BSP-XMPVO has a quasi-quadratic term, which seems to show up in practice and limits performance when compared to the SXMPVO algorithm.

To determine the effect of image size on the performance of the SXMPVO algorithm, we ran it on several data sets, as shown in Table 2. This behavior is in agreement with the complexity found in the previous section—that the running time is linear in terms of the image size in pixels. In each case, the window size was set to match the bounding box of the data set.

Table 3 breaks down the execution time for the four phases of the SXMPVO algorithm for some of the data sets used in our tests. Fig. 9 shows how each of these phases perform for different sized data sets. Fig. 10 shows the performance of MPVONC broken down into its two phases: creating the partial order by setting arrows for the interior arrows (CPO time) and executing the depth first search (DFS time).

A comparison of the SXMPVO algorithm's performance with that of MPVONC is shown in Table 4. The time for both MPVONC and SXMPVO grows approximately linearly in the number of cells. Since SXMPVO must always determine and respect every relationship that MPVONC

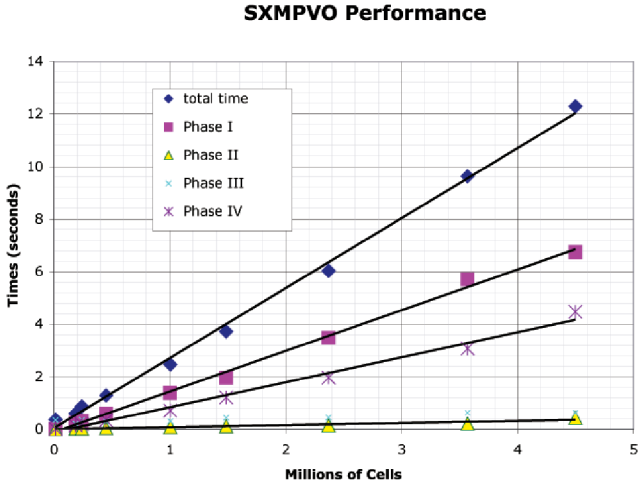

Fig. 9. Timings of the various phases of the SXMPVO algorithm for the data sets and image sizes given in Table 1.

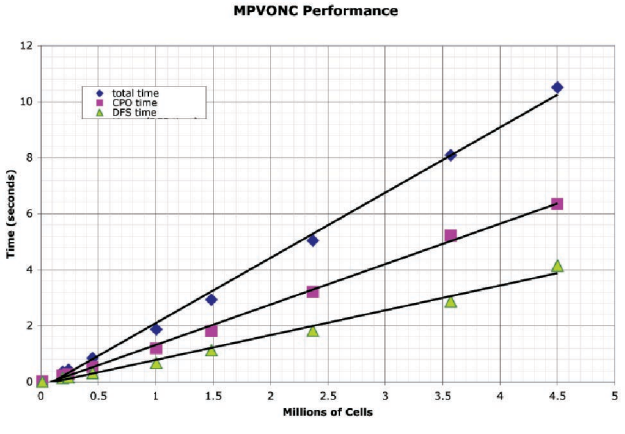

Fig. 10. Timings for the different phases of the MPVONC heuristic for the same data sets as shown in Fig. 9.

TABLE 4
Comparison of Execution Times for the SXMPVO Algorithm and the MPVONC Algorithm for Certain of the Data Sets Given in Table 1

Data Set Size (Number of Cells)	SXMPVO Time (secs.)	MPVONC Time (secs.)
54,000	0.32	0.1
187,395	0.60	0.35
450,000	1.28	0.83
1,485,000	3.60	3.00
4,500,000	10.98	10.85

does, as well as creating and following extra relations between exterior faces, it is always slower than MPVONC for every data set and view angle. For data sets with more than 100,000 cells and for the image sizes shown in Table 1, our SXMPVO algorithm is able to generate an exact image-space visibility ordering of the cells of an nonconvex mesh at about the same speed as the MPVONC heuristic, a nonexact sorting heuristic.

To determine the effect on the execution time, of the number of exterior faces relative to the total number of cells, we created 20 data sets whose meshes had a varying number of disconnected components. Each data set had the same number of cells, but a varying number of exterior faces. The first data set had a single rectilinear $50 \times 50 \times 100$ mesh of hexahedra (250,000 in all) which was then subdivided into tetrahedra, creating six tetrahedra per hexahedron, for a total of 1.5 million tetrahedra. The subdivision created two triangular faces for each rectangular face in the original mesh, so the subdivided mesh had 50,000 exterior faces. There were 5,000 exterior faces on each end and 10,000 exterior faces on each of the remaining four sides.

The second data set was identical to the first, but, instead of creating a single mesh, we sliced the original mesh into six equivalent parts (like slicing bread), creating six equivalent disconnected meshes. We refer to the space between two slices as a *gap*. The planes of exterior faces created by the gaps were parallel to the ends of the mesh, so each gap added 10,000 exterior faces, without changing the total number of cells. So, these five gaps added 50,000 exterior faces. For each additional data set, we added another five gaps to the mesh so that, for the 20th data set, there were 95 gaps, forming a mesh with 96 disconnected components. That mesh had 1,000,000 exterior faces and 1,500,000 cells—a 67 percent ratio of exterior faces to total cells. Fig. 11 shows that the time required to sort the cells was a linear function of the projected area, in pixels, of the exterior faces. Figs. 12 and 13 show volume rendered images of one of the meshes with 50 gaps, from two different views.

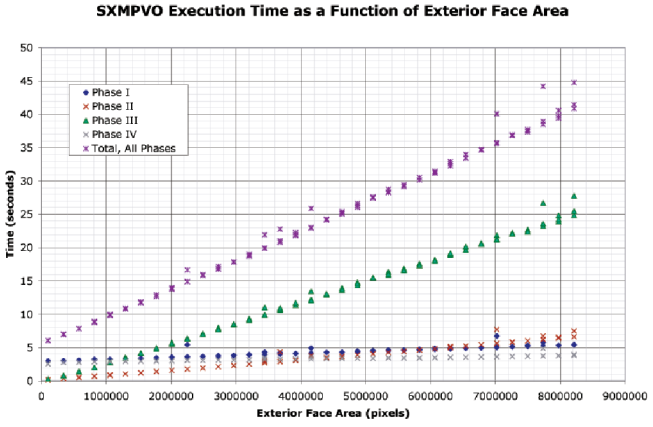

Fig. 11. Times in seconds for execution of the SXMPVO algorithm as a function of the projected area, in pixels, of exterior faces. The data set was rotated 45 degrees around the x and y axes, as shown in Fig. 13, and is for an image size of 375×450 pixels.

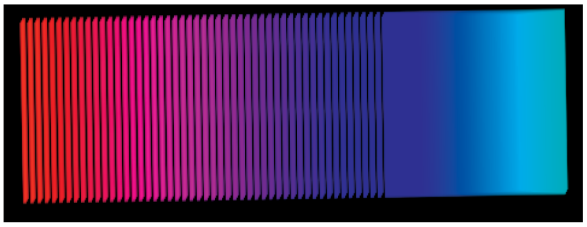

Fig. 12. Volume Rendering of a Tetrahedral Mesh with 1.5 million cells, partitioned with 50 gaps so as to have 51 disconnected components, resulting in 550,000 exterior faces, as described in Section 5.2.

The preceding discussion has dealt with tetrahedral meshes, however, the SXMPVO algorithm can also sort the cells of zoo meshes. Fig. 14 shows an image generated using our algorithm on six disconnected zoo meshes. Each disconnected mesh approximates a sphere and is built in layers using zoo elements. The layer of cells touching the center of each sphere consists of tetrahedra around the north and south poles, and pyramids elsewhere. Subsequent layers consist of prisms around the poles and hexahedra elsewhere. There are a total of 240 tetrahedra, 960 pyramids, 1,200 prisms, and 4,800 hexahedra. The time to compute the visibility ordering for this image was 1.47 seconds. The reason for the slow time is that we utilized complex data structures for zoo meshes, which were created to allow the selective subdivision of nonplanar faces described in [1] and which require following multiple pointers to establish the MPVO face adjacency. The modifications to add the PN relations are basically the same as for tetrahedral data sets.

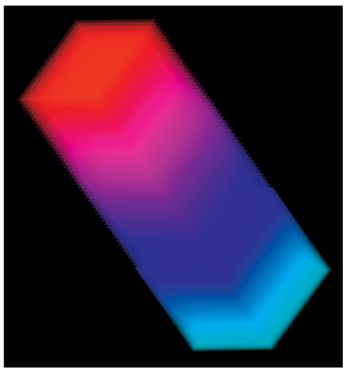

Fig. 13. The same data set as in Fig. 12, but shown rotated 45 degrees around the x and y axes.

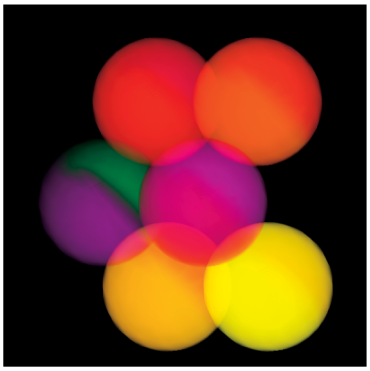

Fig. 14. A volume rendered image of data defined on a zoo mesh, using the SXMPVO algorithm. The mesh has six disconnected components, with 240 tetrahedra, 960 pyramids, 1,200 prisms, and 4,800 hexahedra.

6 Conclusion and Future Work

In this paper, we have presented the SXMPVO algorithm that efficiently finds a global ordering of the cells of an unstructured mesh. This ordering produces the correct sort of the segments along a viewing ray through each pixel. Thus, the algorithm is image-space correct and can be used for the sorting phase of cell projection volume rendering to generate images free from any sorting errors, provided the mesh is acyclic. The boundary of the mesh may be nonconvex and the mesh may be disconnected. Furthermore, the mesh may have cells of different types. The cells of the mesh may even have nonplanar surfaces.

As a result of careful analysis and optimizations, this algorithm has several desirable features; among them are speed, simplicity of implementation, and no extra (i.e., with respect to the MPVO algorithm) preprocessing. This algorithm is faster than all published algorithms that produce an exact visibility ordering, either in object or in image space. It has the potential to be the algorithm of choice for exact visibility sorting for volume rendering. Note that our timing tests were performed using a 250 MHz MIPS R10000 processor. Current CPUs are an order of magnitude faster than this, so our algorithm can be expected to give a faster sorting performance.

The most recently published, and the fastest, times for the rendering phase of volume rendering based on cell projection are reported by Wylie et al. [31] and Weiler et al. [24]. For data sets varying from 187,000 to 1,000,000 tetrahedra, they report rendering times, exclusive of visibility sorting, of between 0.4 and 2.0 seconds (250K-500K tetrahedra per second), using the latest graphics cards. Both of these systems require a visibility sorting algorithm that can keep pace with them. We expect that if the SXMPVO algorithm was implemented and run on the same machines as used by Wylie and Weiler (1200 MHz Athlon PC and 2 GHz Pentium 4) as opposed to our 250 MHz MIPS R10000, it would give very satisfactory performance in the context of these two volume renderers.

Our latest results from the execution of the SXMPVO algorithm on a 2.53 GHz Pentium 4 machine, which is comparable to the machine use by Wylie, are shown in Table 5. These times yield a throughput of 500K-600K tetrahedra per second (for a 1,050,000 pixel image), thus making the SXMPVO algorithm a very suitable candidate for use with these latest rendering techniques. Phases I and III of the algorithm, which account for a substantial portion of the sorting time, are ideal candidates for optimization using Intel's Streaming SIMD Extensions 2 (SSE2).

We see a few directions for future work. It would be interesting to integrate the ideas of Kraus and Ertl [12] into SXMPVO, so our algorithm would be able to handle meshes with visibility cycles. Also, in terms of optimization, we could potentially further optimize the code by using the ideas in [10] for optimizing the cache performance of the scanning phase of our algorithm. Less memory would be required for the PIX lists if the A-Buffer was tiled and created in stages, thus memory for each tile's PIX lists could be reused [9]. Another optimization would be to tile the image and perform parallel sorting [10].

Appendix A

Here, we analyze the complexity of the R-Buffer algorithm of Wittenbrink [30] in the case that all volume cells are semitransparent and none are completely opaque. Let $d(p)$, the depth complexity at pixel p, be the number of cells overlapping the center of p, i.e., the number of fragments that must be sorted and composited at p. (If, instead, there are opaque cells, $d(p)$ will be the number of fragments in front of the first opaque one.) Then, since only one of these fragments is composited per pass through the R-Buffer and the others remain and each pass processes all the remaining fragments, the total number of fragments processed, in either the first pass where all fragments are processed or in subsequent passes using the R-Buffer is

$$\sum_{p} \sum_{i=0}^{d(p)-1} (d(p) - i) = \sum_{p} \frac{d(p)(d(p)+1)}{2} \geq \frac{1}{2} w * h * \mathcal{A}(d(p)^2),$$

where w and h are the width and height of the screen and

TABLE 5
Timings for the Execution of the SXMPVO Algorithm on an Intel Pentium 4 2.53 MHz Machine for Five Different Image Sizes

		Image Size in Pixels				
Data Set	n	170,000	350,000	700,000	1,050,000	2,000,000
spx	12,936	0.06	0.11	0.2	0.29	0.49
bluntfin	187,395	0.17	0.21	0.28	0.34	0.53
kew	416,926	0.54	0.59	0.67	0.76	0.96
fighter	1,403,504	1.92	1.99	2.15	2.23	2.49

The times for a 1,050,000 pixel image correspond to a throughput of 500K-600K tetrahedra per second. In each case, the window size was set to match the bounding box of the data set. The data sets, spx, kew, and fighter, are from finite element method simulations on unstructured meshes.

TABLE 6
The Number of Tiny Cells, Cells that Do Not Project onto Any Screen Pixels, for Several Different Image Sizes and Several Different Unstructured Data Sets

		Image Size in Pixels				
Data Set	n	170,000	350,000	700,000	1,050,000	2,000,000
bluntfin	187,395	13,356	9,427	5,602	3,831	1,814
f117	240,122	67,340	34,133	11,126	4,601	761
helix1.5	1,485,000	0	0	0	0	0

In each case, the image size matches the bounding box of the data set.

$$\mathcal{A}(d(p)^2) = \frac{1}{w * h} \sum_p d(p)^2$$

is the average squared value of the depth complexity. Since $d(p)$ is nonnegative and the square function has a positive second derivative, $\mathcal{A}(d(p)^2) \geq \mathcal{A}(d(p))^2 = \hat{d}^2$, where $\hat{d}$ is the average depth complexity. Therefore, the cost of the R-Buffer sorting algorithm is $\Omega(w * h * \hat{d}^2)$. For volume data of moderate to large complexity, this makes the R-Buffer impractical, even in hardware. The A-Buffer algorithm of Wilhelms et al. [26] can take advantage of x-coherence of the z-sort, where such coherence exists, so it may cost somewhat less. However, if the x range of most cells is only one or two pixels, there is no such coherence and, if an insertion sort is used for the z-sort, its costs will also be quadratic in $\hat{d}$, as above.

Now, let $\hat{d}_e$ be the average depth complexity of the exterior faces of the mesh. If the mesh consists of disjoint cells with no common faces, then $\hat{d}_e = 2\hat{d}$, but, in the meshes that are visualized in practice, $\hat{d}_e$ is much less than $\hat{d}$. The total area a of the exterior faces, measured in pixels, is $w * h * \hat{d}_e$. Therefore, according to the analysis in Section 5.1, our algorithm's expected cost for data sets seen in practice is $O(w * h * (\hat{d}_e + 1) + n)$. Thus, our sorting algorithm is much faster than the R-Buffer or A-Buffer algorithms that sort all the fragments, which also have an $\Omega(n)$ term since they must at least read all input cells.

APPENDIX B

In this appendix, we discuss our algorithm in the context of aliasing that is inherent in volume rendering based on cell projection. If a cell is so small that it does not project onto any screen pixels, we call it a *tiny* cell. Of course whether or not a cell is tiny is view-dependent. A cell may be tiny for one view direction and not another, for one image size and not another, etc. In Table 6, we show the results of tests of several data sets to determine the number of tiny cells for different image sizes. It is an open question, which we are investigating, just how much of an impact splatting tiny cells will have on the performance of the rendering phase when using current graphics cards. Certainly, one way to include tiny cells in the image would be to use current methods for the rendering phase and zoom in on the data.

The system reported in [28] is capable of some antialiasing. It can render tiny cells too small to be accurately sampled at pixel centers by splatting them using a piecewise quadratic splat kernel with a 3×3 pixel footprint. If one of the system's object space sorting options, from [6], [22], or [23] is used, these tiny cells will be sorted correctly, together with all the other cells. If the image space methods of this paper are used, however, the relationship $a <_{vp} b$ will be only guaranteed to be respected if there is a continuous sequence of adjacent intervening cells between cells a and b, as in Section 3.2, which will be enforced by the behind relations from adjacent cells, or if the necessary relations across regions where ray-gaps may occur happen to be sampled at some pixel center. This is not always guaranteed to be the case, so there may be some sorting errors if splatting is used to antialias tiny cells. Tiny cells that are interior cells will be output by SXMPVO in correct order. However, if a tiny cell is an exterior cell, then it may occur out of order. Sorting the sink cells by increasing distance of their centroid to the viewpoint should reduce the number of these sorting errors. In any case, all cells, regardless of size, will be output by SXMPVO.

We describe a completely different method of dealing with tiny cells in [17] in connection with the cell slicing algorithm of Yagel et al. [32]. It splats the tiny cells onto a texture in the closest slice plane.

ACKNOWLEDGMENTS

This work was performed under the auspices of the US Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48. Cláudio T. Silva is partially supported by the US Department of Energy under the VIEWS program and the MICS office and the US National Science Foundation under grants CCR-0306530 and EIA-0323604. The authors are grateful to the referees who reviewed this paper and offered very helpful suggestions and criticisms.

REFERENCES

[1] J. Bennett, R. Cook, N. Max, and P. Williams, "Parallelizing a High Accuracy Hardware-Assisted Volume Renderer for Meshes with Arbitrary Polyhedra," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics*, pp. 101-106, Oct. 2001.

[2] L. Carpenter, "The A-Buffer, an Antialiased Hidden Surface Method," *Computer Graphics, Proc. SIGGRAPH '84*, pp. 103-108, July 1984.

[3] P. Cignoni and L. De Floriani, "Power Diagram Depth Sorting," *Proc. 10th Canadian Conf. Computational Geometry*, 1998.

[4] P. Cignoni, C. Montani, D. Sarti, and R. Scopigno, "On the Optimization of Projective Volume Rendering," *Proc. Visualization in Scientific Computing '95*, pp. 58-71, 1995.

[5] P. Cignoni, C. Montani, and R. Scopigno, "Tetrahedra Based Volume Visualization," *Math. Visualization—Algorithms, Applications, and Numerics*, H.-C. Hege and K. Polthier, eds., pp. 3-18, Springer Verlag, 1998.

[6] J. Comba, J. Klosowski, N. Max, J.S.B. Mitchell, C. Silva, and P. Williams, "Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids," *Computer Graphcs Forum*, vol. 18, pp. 367-376, 1999.

[7] C. Everett, "Interactive Order-Independent Transparency," technical report, NVIDIA Corp., 2001, http://developer.nvidia.com/view.asp?IO=Interactive_Order_Transparency.
[8] R. Farias, J. Mitchell, and C. Silva, "ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering," *Proc. 2000 Volume Visualization Symp.,* pp. 91-99, Oct. 2000.
[9] R. Farias and C. Silva, "Out-of-Core Rendering of Large, Unstructured Grids," *IEEE Computer Graphics & Applications,* vol. 21, no. 4, pp. 42-51, July/Aug. 2001.
[10] R. Farias and C. Silva, "Parallelizing the ZSWEEP Algorithm For Distributed-Shared Memory Architectures," *Proc. Int'l Volume Graphics Workshop 2001,* pp. 181-192, 2001.
[11] D. King, C. Wittenbrink, and H. Wolters, "An Architecture for Interactive Tetrahedral Volume Rendering," *Proc. Int'l Volume Graphics Workshop 2001,* pp. 101-110, 2001.
[12] M. Kraus and T. Ertl, "Cell Projection of Cyclic Meshes," *Proc. IEEE Visualization 2001,* pp. 215-222, 2001.
[13] A. Mammen, "Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique," *IEEE Computer Graphics and Applications,* vol. 9, pp. 43-55, July 1984.
[14] N. Max, "Optical Models for Direct Volume Rendering," *IEEE Trans. Visualization and Computer Graphics,* vol. 1, pp. 99-108, June 1995.
[15] N. Max, P. Hanrahan, and R. Crawfis, "Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions," *Computer Graphics,* vol. 24, pp. 27-33, Nov. 1990.
[16] N. Max, P. Williams, and C. Silva, "Approximate Volume Rendering for Curvilinear and Unstructured Grids by Hardware-Assisted Polyhedron Projection," *Int'l J. Imaging Systems and Technology,* vol. 11, pp. 53-61, 2000.
[17] N. Max, P. Williams, C. Silva, and R. Cook, "Volume Rendering for Curvilinear and Unstructured Grids," *Proc. Computer Graphics Int'l 2003,* pp. 210-215, 2003.
[18] S. Röttger, M. Kraus, and T. Ertl, "Hardware Accelerated Volume and Isosurface Rendering Based on Cell Projection," *Proc. IEEE Visualization 2000,* pp. 109-116, 2000.
[19] G. Schussman and N. Max, "Hierarchical Perspective Volume Rendering Using Triangle Fans," *Proc. Int'l Volume Graphics Workshop 2001,* pp. 195-200, 2001.
[20] M. Segal and K. Akeley, *The OpenGL Graphics System: A Specification,* version 1.2.1. Silicon Graphics, Inc., Apr. 1999.
[21] P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics (Proc. San Diego Workshop Volume Visualization),* vol. 24, pp. 63-70, Nov. 1990.
[22] C. Silva, J.S.B. Mitchell, and P. Williams, "An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes," *Proc. ACM Symp. Volume Visualization,* pp. 87-94, Oct. 1998.
[23] C. Stein, B. Becker, and N. Max, "Sorting and Hardware Assisted Rendering for Volume Visualization," *Proc. SIGGRAPH Symp. Volume Visualization,* pp. 83-90, Oct. 1994.
[24] M. Weiler, M. Kraus, and T. Ertl, "Hardware-Based View-Independent Cell Projection," *Proc. 2002 Volume Visualization Symp.,* pp. 13-22, 2002.
[25] J. Wilhelms and A. Van Gelder, "A Coherent Projection Approach for Direct Volume Rendering," *Computer Graphics,* vol. 25, pp. 275-283, July 1991.
[26] J. Wilhelms, A. Van Gelder, P. Tarantino, and J. Gibbs, "Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids," *Proc. IEEE Visualization 1996,* pp. 57-64, 1996.
[27] P. Williams, "Visibility Ordering Meshed Polyhedra," *ACM Trans. Graphics,* vol. 11, no. 2, pp. 103-126, Apr. 1992.
[28] P. Williams, N. Max, and C. Stein, "A High Accuracy Volume Renderer for Unstructured Data," *IEEE Trans. Visualization and Computer Graphics,* vol. 4, no. 1, pp. 37-54, Jan.-Mar. 1998.
[29] C. Wittenbrink, "Cellfast: Interactive Unstructured Volume Rendering," *Proc. IEEE Visualization '99, Late Breaking Hot Topics,* pp. 21-24, 1999.
[30] C. Wittenbrink, "R-Buffer: A Pointerless A-Buffer Hardware Architecture," *Proc. ACM-Eurographics Workshop Graphics Hardware,* pp. 73-80, 2001.
[31] B. Wylie, K. Moreland, L.A. Fisk, and P. Crossno, "Tetrahedral Projection Using Vertex Shaders," *Proc. 2002 Volume Visualization Symp.,* pp. 7-12, 2002.
[32] R. Yagel, D. Reed, A. Law, P.-W. Shih, and N. Shareef, "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing," *Proc. 1996 Volume Visualization Symp.,* pp. 55-62, Oct. 1996.

Richard Cook received the MS degree in computer science from the University of California at Davis in 2001. He is a computer scientist working at Lawrence Livermore National Laboratory in the Services and Development Department. His current projects include work on the Tera-Scale Browser Project (TSB) for the US Department of Energy's Accelerated Strategic Computing Initiative (ASCI) and scientific visualization support duties for the Information Management and Graphics Group (IMG).

Nelson Max received the PhD degree in mathematics from Harvard University in March 1967. His research interests are in the areas of scientific visualization, volume and flow rendering, computer animation, molecular graphics, predicting protein folding from sequence, and realistic computer rendering, including shadow and radiosity effects. Since 1977, he has been a computer scientist at Lawrence Livermore National Laboratory and has been teaching part time at the University of California, Davis, currently as a 50 percent professor of applied science. He has taught mathematics and computer science at the University of California, Berkeley, the University of Georgia, Carnegie Mellon University, and Case Western Reserve University. He was director of the US National Science Foundation supported Topology Films Project in the early 1970s, which produced computer animated educational films on mathematics. He worked in Japan for 3 1/2 years as codirector of two Omnimax (hemisphere screen) stereo films for international expositions, showing the molecular basis of life. He is a member of the IEEE Computer Society.

Cláudio T. Silva received the Bachelor's degree in mathematics from the Federal University of Ceara, Brazil, and the MS and PhD degrees in computer science from the State University of New York at Stony Brook. He is an associate professor of computer science at the University of Utah and a member of the Scientific Computing and Imaging (SCI) Institute. His main research interests are in graphics, visualization, applied computational geometry, bioinformatics, and high-performance computing. His current projects include the development of out-of-core algorithms for large-scale scientific visualization, techniques for point-based modeling and rendering, and efficient algorithms for modern graphics hardware. He has published more than 60 publications in international conferences and journals, holds four US patents, and presented tutorials at ACM SIGGRAPH, Eurographics, and IEEE Visualization conferences. He serves on numerous program committees and is cochair of the IEEE Symposium on Volume Visualization and Graphics (VolVis 2004). He is a member of the ACM, Eurographics, and IEEE.

Peter L. Williams received the PhD degree in computer science from the University of Illinois at Urbana-Champaign and the BS degree in engineering-physics from the University of California at Berkeley. He has taught computer science at Vassar College, the University of Connecticut at Storrs, and Harvey Mudd College. He is currently a computer scientist at the Lawrence Livermore National Laboratory, where he is a member of the Center for Applied Scientific Computing. His research interests include graphics, scientific visualization, volume rendering (especially unstructured data), and high performance parallel and distributed computing. He is a member of the IEEE Computer Society.