An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes

Cláudio T. Silva* Joseph S. B. Mitchell† Peter L. Williams‡

IBM T. J. Watson Research Center
State University of New York at Stony Brook

Abstract

A visibility ordering of a set of objects, from a given viewpoint, is a total order on the objects such that if object *a* obstructs object *b*, then *b* precedes *a* in the ordering. Such orderings are extremely useful for rendering volumetric data. We present an algorithm that generates a visibility ordering of the cells of an unstructured mesh, provided that the cells are convex polyhedra and nonintersecting, and that the visibility ordering graph does not contain cycles. The overall mesh may be nonconvex and it may have disconnected components. Our technique employs the sweep paradigm to determine an ordering between pairs of exterior (mesh boundary) cells which can obstruct one another. It then builds on Williams' MPVO algorithm [33] which exploits the ordering implied by adjacencies within the mesh. The partial ordering of the exterior cells found by sweeping is used to augment the DAG created in Phase II of the MPVO algorithm. Our method thus removes the assumption of the MPVO algorithm that the mesh be convex and connected, and thereby allows us to extend MPVO algorithm, without using the heuristics that were originally suggested by Williams (and are sometimes problematic). The resulting XMPVO algorithm has been analyzed, and a variation of it has been implemented for unstructured tetrahedral meshes; we provide experimental evidence that it performs very well in practice.

Key Words and Phrases: Volume rendering, scientific visualization, finite element methods, depth ordering, volume visualization, visibility ordering.

1 Introduction

A *visibility ordering* (or *depth ordering*), $<_v$, of a set S of objects from a given viewpoint, $v \in \Re^3$, is a total (linear) order on S such that if object $a \in S$ visually obstructs object $b \in S$, partially or completely, then b precedes a in the ordering: $b <_v a$.

Direct volume rendering based on projective methods [20, 33, 30, 32] works by projecting the polyhedral cells of a mesh onto the image plane, in visibility order, and incrementally compositing the cell's color and opacity into the final image. Projective methods, as opposed to those using ray tracing, have the advantage of being able to make extensive use of graphics hardware, and have the potential of avoiding anti-aliasing artifacts. Existing visibility ordering algorithms for arbitrary polyhedral cell complexes (*e.g.* curvilinear or non-convex unstructured meshes) are quite slow. Some visibility ordering methods for arbitrary polyhedral cell complexes utilize heuristics which generate a fast but inexact sorting.

This paper describes an extension of the meshed polyhedra visibility ordering (MPVO) algorithm of Williams [33], which we call the XMPVO algorithm, that efficiently generates an exact visibility ordering of the cells of arbitrary polyhedral cell complexes in interactive time. The cells are expected to be convex and nonintersecting, and the visibility ordering graph of the cells should not contain cycles. As our results show, the algorithm presented herein runs substantially faster (by orders of magnitude) than any previously published algorithm.

The next section discusses related previous work. Section 3 defines basic terminology. Section 4 gives an overview of the MPVO algorithm. Section 5 describes the XMPVO algorithm, and then in Section 6 we present implementation details for a variation of this algorithm. Finally in Section 7 we present timing results and images from diverse data sets, and in Section 8 we give our conclusions and discuss future work.

2 Previous Work

An algorithm, called the "Meshed Polyhedra Visibility Ordering" (MPVO) algorithm, for visibility ordering the cells of an acyclic convex mesh of convex cells is described by Williams [33]. A similar algorithm to the MPVO Algorithm was developed independently by Max, Hanrahan and Crawfis [20]. Both algorithms were based on the work of Edelsbrunner described in his paper on the acyclicity of cell complexes [13]. The MPVO algorithm runs in linear time and uses linear storage.

For some important classes of meshes (*e.g.*, rectilinear meshes and Delaunay meshes [13]), it is known that a visibility ordering always exists, with respect to any viewpoint. If the visibility ordering graph has cycles for a given viewpoint, no visibility ordering exists. It is an important problem to find a small number of "cuts" that partition the cells so as to eliminate such cycles; see [6, 5]. The binary space partition (BSP) tree algorithm [15], which is typically used to depth-sort polygons, is not suitable for visibility ordering large polyhedral meshes since the BSP algorithm uses splitting planes. Since the cells are meshed, a large number of cells can be split, re-

*Visual and Geometric Computing Group, P.O. Box 704, IBM T. J. Watson Research Center, Yorktown Heights, NY, 10598; csilva@watson.ibm.com.

†Department of Applied Mathematics and Statistics, State University of New York at Stony Brook, Stony Brook, NY 11794–3600; jsbm@ams.sunysb.edu.

‡Data Explorer (DX) Research Group, P.O. Box 704, IBM T. J. Watson Research Center, Yorktown Heights, NY, 10598; p.williams@computer.org.

sulting in a potential explosion in the total number of cells. As shown by Paterson and Yao [25], the BSP tree algorithm can have performance that is quadratic in f, the number of faces in the original mesh. An A-buffer [11] is also not suitable for visibility ordering large meshes for volume rendering because there are too many transparent cells at each pixel, making memory requirements prohibitive with current hardware.

Williams [33] also described a heuristic, called the MPVONC algorithm, which sorts the cells of acyclic *nonconvex* meshes of convex cells, *i.e.* meshes with cavities and/or voids. This heuristic generates an exact sorting of the cells only if no *boundary anomalies* [33] are present. The MPVONC algorithm, in practice, is linear in time for most meshes; see [33].

Stein *et al* [32] describe an algorithm for visibility ordering an arbitrary collection of acyclic nonintersecting convex polyhedra. This algorithm runs in time $O(n^2)$ (worst case) for n arbitrarily shaped, nonintersecting convex polyhedra with planar faces, whose visibility ordering does not contain cycles. The faces of adjacent cells need not be aligned, and the meshes may have disconnected portions. The algorithm is effectively a 3D generalization of the Newell, Newell and Sancha sort for polygons [22, 23]. Williams *et al* [34] describe a correction and an optimization to the original Stein algorithm. Even with the optimization, this algorithm does not run in interactive time, *e.g.* it requires on the order of 3 minutes to sort 200,000 cells and 15 minutes to sort 1,000,000 cells, on an SGI Power Onyx using an R10000 194 MHZ CPU. (See the results in Section 7.)

An exact visibility ordering algorithm is described by de Berg, Overmars, and Schwarzkopf [9] which, in worst-case time $O(n^{4/3+\varepsilon})$ for any fixed $\varepsilon > 0$, determines an ordering or reports that none exists (due to a cycle in the "behind" relation). However, this algorithm, which is based on a general framework for computing and verifying linear orders extending implicitly defined binary relations, is mostly of theoretical interest and is not readily implemented. In particular, it relies on the fairly complex dynamic data structure of Agarwal and Matoušek [1] for searching for intersections between line segments in space and "curtains" (the shadow surface cast by a segment) induced by line segments. Its theoretical importance stems from the fact that it determines, in *sub*quadratic worst-case time, if a linear ordering exists *without* necessarily computing the full behind relation (which may have size quadratic in the number of objects).

Karasick *et al* [19], building on the earlier work of Edelsbrunner, describe a linear expected time algorithm for sorting the cells of 3D *Delaunay meshes* (the Delaunay tetrahedralization of some set of discrete points). Their algorithm is based on sorting the cells by their "powers". While this approach is elegant and efficient, many unstructured and curvilinear meshes encountered in scientific visualization are not Delaunay meshes.

Finally, we mention some related work in ray casting of unstructured grids, in which the goal is to compute the depth ordering along each of the rays that passes through one of the screen pixels. Garrity [16] employed the idea (as do we) of focusing attention on boundary facets in a nonconvex mesh; he needs to perform "ray shooting queries"* only to jump from one boundary facet to another. Giertsen [17], Yagel *et al* [35], and Silva and Mitchell [31] have shown methods of speeding up ray casting by exploiting coherence through the use of a sweep algorithm (as we do here – our approach is related to that of [31]).

In this paper, we describe an algorithm which extends the MPVO algorithm to generate an exact visibility ordering of the cells of arbitary acyclic polyhedral cell complexes, having convex nonintersecting cells, in interactive time. We review the MPVO algorithm in Section 4, and then describe the required extension in Section 5.

3 Preliminaries

We begin with some basic definitions. A *polyhedron* is a closed subset of $\Re^3$ whose boundary consists of a finite collection of convex polygons (*2-faces*, or *facets*) whose union is a connected 2-manifold. The *edges* (*1-faces*) and *vertices* (*0-faces*) of a polyhedron are simply the edges and vertices of the polygonal facets. A convex polyhedron is called a *polytope*. A polytope having exactly four vertices (and four triangular facets) is called a *simplex* (*tetrahedron*). A finite set S of polyhedra forms a *mesh* (or an *unstructured grid*) if the intersection of any two polyhedra from S is either empty, a single common edge, a single common vertex, or a single common facet of the two polyhedra; such a set S is said to form a *cell complex*. The polyhedra of a mesh are referred to as the *cells* (or *3-faces*). We say that cell C is *adjacent* to cell C' if C and C' share a common facet. The adjacency relation is a binary relation on elements of S that defines an *adjacency graph*.

A facet that is incident on only one cell is called a *boundary facet*. A *boundary cell* is any cell having a boundary facet. The union of all boundary facets is the *boundary* of the mesh. If the boundary of a mesh 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. If the cells are all simplicies, then we say that the mesh is *simplicial*.

The input to our problem will be a given mesh S. We let c denote the number of connected components of S. If $c = 1$, the mesh is *connected*; otherwise, the mesh is *disconnected*. We let n denote the total number of edges of all polyhedral cells in the mesh. Then, there are $O(n)$ vertices, edges, facets, and cells.

For some of our discussions, we will be assuming that the input mesh is given in a standard data structure for cell complexes (*e.g.*, a facet-edge data structure [12]), so that each cell has pointers to its neighboring (incident) cells, and basic traversals of the boundary edges of facets are also possible by following pointers. If the raw data does not have this topological information already encoded in it, then it can be obtained by a preprocessing step, using basic hashing methods, in worst-case time $O(n \log n)$.

We use a coordinate system in which the viewing direction is in the $-z$ direction, and the image plane is the (x,y) plane.

*A "ray shooting query" requires that we find the first object hit by a ray that is fired in a given direction from a given query point. The field of computational geometry provides some data structures for handling them: One can either answer queries in time $O(\log n)$, at a cost of $O(n^{4+\varepsilon})$ preprocessing and storage [1, 8, 26], or answer queries in worst-case time $O(n^{3/4})$, using a data structure that is essentially linear in n [2, 29]. In terms of worst-case complexity, there are reasons to believe that these tradeoffs between query time and storage space are essentially the best possible. Unfortunately, these algorithms are rather complicated, with high constants, and have not yet been implemented or shown to be practical. See also the work of Mitchell *et al* [21], who devise methods of ray shooting that are "query sensitive."

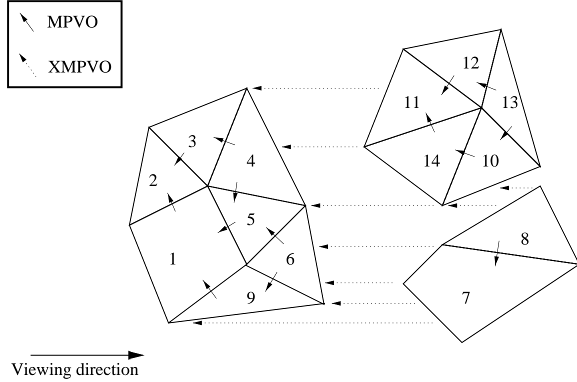

Figure 1: Visibility ordering of the cells of a mesh relative to viewpoint v: MPVO relations (solid arrows) capture local ordering, while XMPVO relations (dotted) capture global ordering among cells, allowing even disconnected meshes to be handled.

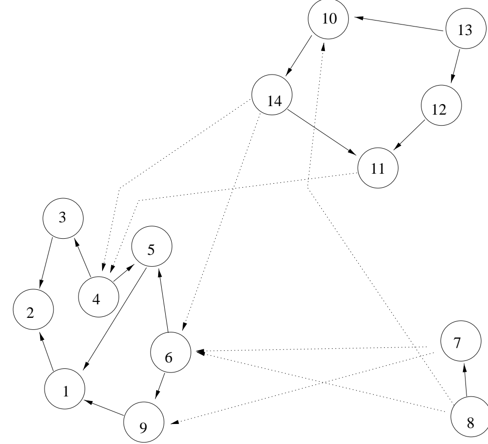

Figure 2: The DAG for the mesh in Figure 1. Note that XMPVO generates redundant visibility ordering relations.

We let ρ_u denote the ray from the viewpoint v through the point u.

We say that cells C and C' are *immediate neighbors* with respect to viewpoint v if there exists a ray ρ from v that intersects C and C', and no other cell $C'' \in S$ has a nonempty intersection $C'' \cap \rho$ that appears in between the segments $C \cap \rho$ and $C' \cap \rho$ along ρ. Note that if C and C' are adjacent, then they are necessarily immediate neighbors. Further, in a convex mesh, the *only* pairs of cells that are immediate neighbors are those that are adjacent.

A *visibility ordering* (or *depth ordering*), $<_v$, of a mesh S from a given viewpoint, $v \in \Re^3$ is a total (linear) order on S such that if cell $C \in S$ visually obstructs cell $C' \in S$, partially or completely, then C' precedes C in the ordering: $C' <_v C$. A visibility ordering is a linear extension of the binary *behind* relation, "$<$", in which cell C is *behind* cell C' (written $C < C'$) if and only if C and C' are immediate neighbors and C' at least partially obstructs C; *i.e.*, if and only if there exists a ray ρ from the viewpoint v such that $\rho \cap C \neq \emptyset$, $\rho \cap C' \neq \emptyset$, $\rho \cap C'$ appears in between v and $\rho \cap C$ along ρ, and no other cell C'' intersects ρ at a point between $\rho \cap C$ and $\rho \cap C'$. A visibility ordering can be obtained in linear time (by topological sorting) from the behind relation, $(S, <)$, provided that the directed graph on the set of nodes S defined by $(S, <)$ is acyclic. If the behind relation induces a directed cycle, then no visibility ordering exists.

4 An Overview of the MPVO Algorithm

An intuitive overview of the MPVO Algorithm is as follows. First, the adjacency graph for the cells of a given convex mesh is constructed. Then, for any specified viewpoint, a visibility ordering can be computed simply by assigning a direction to each edge in the adjacency graph and then performing a topological sort of the graph. The adjacency graph can be reused for each new viewpoint and for each new data set defined on the same static mesh.

The direction assigned to each edge of the adjacency graph is determined by calculating the behind relation for the two cells connected by the edge. Informally, the behind relation is calculated as follows. Each edge corresponds to a facet shared by two cells. That facet defines a plane which in turn defines two half-spaces, each containing one of the two cells. If we represent the behind relation by a directed arc (arrow) through the shared face, then the direction of the arrow is towards the cell whose containing half-space contains the viewpoint; see Figures 1 and 2. To implement this, the plane equation for the shared face can be evaluated at the viewpoint v. The adjacency graph and the plane equation coefficients can be computed and stored in a preprocessing step.

5 The XMPVO Algorithm

The XMPVO algorithm extends the MPVO algorithm to handle nonconvex meshes efficiently. When a mesh is nonconvex or has disconnected components, it does not suffice to examine only edges of the adjacency graph. The key idea in XMPVO is to extend the DAG created by Phase II of MPVO with additional relations — relations between pairs of exterior (boundary) cells which can obstruct one another. These new relations are determined by ray-shooting queries in the context of the sweep paradigm described below. The addition of these relations make the DAG sufficient for visibility ordering any mesh exactly, thus removing the assumption of the MPVO algorithm that the mesh be convex and connected (see Figures 1 and 2). In the rest of this section, we discuss the theory behind this extension (with some practical remarks in the end). In Section 6, an actual implementation based on these ideas is presented, and in Section 7, XMPVO is shown (experimentally) to be orders of magnitude faster than previous techniques.

The algorithm is based on a standard algorithmic paradigm in computational geometry — the "sweep" paradigm [10, 27], in which a *sweep-line* is swept across the plane, or a *sweep-plane* is swept across 3-space. A data structure, called the *sweep structure* (or *sweep status*), is maintained during the simulation of the continuous sweep, and at certain discrete *events* (*e.g.*, when the sweeping object hits one of a discrete set of points), the sweep structure is updated to reflect the change. This allows the algorithm to localize the problem to be solved, solving it within the lower dimensional space of the sweep-line or sweep-plane, while exploiting spatial coherence in the data.

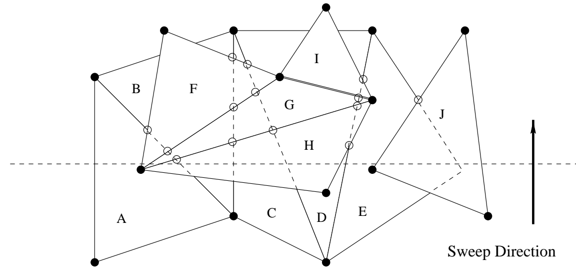

Figure 3: Three distinct connected sets of boundary facets are depicted, projected onto the (x,y)-plane. Edge portions that are not visible are shown dashed. The sweep continues upward (in positive y direction). The vertex events are shown with solid circles; the crossing point events in the event queue are shown with hollow circles at points where two edges have a crossing in the projection. At such event points, we compute the z-ordering of the respective facets. This allows us to infer the visibility order relations.

The goal of the algorithm is to identify, for each boundary cell C_i, the set of boundary cells C_j that are C_i's immediate neighbors. This permits us to compute the entire behind relation for all cells of the mesh, since all other pairs of immediate neighbors are determined by the adjacency graph of S.

Our sweep algorithm executes the following steps:

(1) We identify, in a single pass over the cells of S, the set of boundary facets, B. We also identify the adjacency relationship, and, as in MPVO, establish the behind relationship for every pair of adjacent cells, establishing a directed arc corresponding to each facet shared by an adjacent pair of cells.

(2) We perform a sweep of the set B in 3-space, using a sweep-plane that is parallel to the x-z plane, sweeping in the direction of increasing y.

This sweep is equivalent to sweeping, with a horizontal line, the arrangement $\mathcal{A}$ of line segments in the x-y plane that are obtained by projecting the edges of facets in B; it is essentially a Bentley-Ottmann sweep of $\mathcal{A}$, lifted to 3-space. (Note, however, that we do not need explicitly to project the edges of B, nor explicitly to construct $\mathcal{A}$.) In particular, events in the sweep occur at vertices V_B of B, and at crossing points in $\mathcal{A}$ (where the projections of two edges of B cross).

The sweep status encodes the set of elements of B intersected by the sweep plane (see Figure 3); specifically, we maintain an x-ordering (X-LIST, in a balanced binary tree) of the edges of $\mathcal{A}$ that cross the sweep line (corresponding to the sweep plane). Additionally, the sweep status maintains the elements of B that intersect the sweep plane in a data structure that allows us to keep track of, for each interval in the X-LIST, the set of elements of B whose x-projection (within the sweep plane), onto the sweep line, contains the interval. This can be done using segments trees [10], in time $O(\log b')$ per operation (insertion or deletion) with $O(b' \log b')$ storage, where b' is the number of elements of B intersected by the sweep plane.

The event queue (maintained as a heap) is initialized by sorting the y-coordinates of V_B; then, as the sweep discovers crossing point vertices of $\mathcal{A}$ (*i.e.*, whenever there is a change in the x-ordering of the projected edges), the y-coordinates of crossing points that correspond to them are enqueued.

At each event point, there is some bookkeeping to do to maintain the sweep status – a constant number of insertions or deletions. At each event point, we also compute the new neighbor relations that are discovered at the event point: this corresponds to the adjacencies in the z ordering of the cells that participate in the event. In other words, we examine each of the cells containing the edges that are responsible for the event point, and determine their relative ordering along ρ, among the other cells that are also stabbed by ρ. (This information is maintained in the sweep status.) The segment tree of the sweep status allows us to do this efficiently (see analysis below).

Correctness of the algorithm is implied by the following simple lemma:

Lemma 1 *If boundary cell C_i is behind boundary cell C_j, then there exists a ray, ρ, through some vertex of the arrangement $\mathcal{A}$ that serves as a witness for the fact that $C_i < C_j$; i.e., such that $\rho \cap C_i \neq \emptyset$, $\rho \cap C_j \neq \emptyset$, and no other cell of S intersects ρ at a point in between the segment $\rho \cap C_i$ and the segment $\rho \cap C_j$.*

Proof. Let ρ be a witness ray to the fact that $C_i < C_j$. Then, ρ corresponds to a point, p_ρ, in the viewing plane. If p_ρ lies interior to a 2-face of $\mathcal{A}$, then we can move p_ρ rigthwards, say, until it comes in contact with an edge (1-face) of $\mathcal{A}$; this corresponds to rotating ρ to the right until it comes into contact with (at least) one edge of a cell of S. Now p_ρ lies on the edge set of $\mathcal{A}$; if it is not yet at a vertex of $\mathcal{A}$, then we can slide it along its containing edge (in either direction) until it encounters a vertex of the arrangement $\mathcal{A}$, at which point ρ passes either through a vertex of some cell of S or through a pair of edges of cells of S. In any case, the motion of p_ρ (and ρ) is such that it has stayed within the closure of a 2-face of $\mathcal{A}$, and therefore, except for possible degeneracies on the boundary of the 2-face, intersects the same set of cells as it did originally. Thus, ρ has remained a witness ray. $\square$

After the sweep algorithm establishes the behind relations among pairs of boundary cells, the visibility ordering algorithm completes its task simply by doing topological sort of $(S,<)$, which takes time linear in the size of the directed graph corresponding to the relation.

Analysis

To analyze the above algorithm, we let $b = |B|$ be the number of boundary facets and let I denote the number of crossing points in $\mathcal{A}$. In the worst case, $I = \Omega(b^2)$, as the projected edges could form a grid pattern; this is unlikely in practice. The maintenance of the sweep structures requires time $O(\log b)$ per event. At event points, we must also compute the new information gained about the behind relation at the event point. Assuming nondegeneracy (which can be ensured by general perturbation techniques [14]), only a constant number of cells are involved in the change of the behind relation. The change can be computed naively in time $O(m)$, where m is the number of cells intersected by the ray ρ through the event point. (These cells are identifiable efficiently, in time $O(\log b)$, using the segment tree data structure, since each is represented within the segment tree as a union of $O(\log b)$ canonical intervals.) If, however, we also augment the segment tree with an auxilliary data structure, storing the canonical intervals stored

at each node in sorted order by z, then the search for a neighbor along ρ is done in time $O(\log b)$ per node of the segment tree visited; overall, this results in time $O(\log^2 b)$ per event point in the worst case. The final time complexity is then $O((b+I)\log^2 b)$. The space complexity is simply $O(b)$, if we use the standard trick for reducing the space complexity of the Bentley-Ottmann sweep from $O(I)$ to $O(b)$ (by removing non-adjacent segment pairs from the event queue; see, *e.g.*, [10]), plus the $O(b \log b)$ space required for storing up to b intervals in the segment tree.

Theorem 2 *The XMPVO sweep algorithm requires* $O((b+I)\log^2 b)$ *time, worst case, and working storage* $O(b \log b)$.

Remarks:

(a) In practice, the number b of boundary cells is much less than the number n of cells of S; we may expect it to be roughly $n^{2/3}$ (*e.g.*, if the grid is a regular k-by-k-by-k grid, then $b = 6k^2$ while $n = k^3$). Further, it suffices to consider only those boundary facets of S that are not on the convex hull of S. In practice, we also expect I to be much smaller than its worst-case upper bound of $O(b^2)$.

(b) Using much more sophisticated methods for computing intersections among line segments, the arrangement $\mathcal{A}$ can in fact be constructed in optimal worst-case time $O(I + b \log b)$, using only $O(b)$ space (see [4, 3]). However, an auxilliary data structure would still be required to obtain the ray-shooting information along each ray ρ at event points.

(c) This sweep algorithm is similar to that of [5] and [24], who considered the problem of depth-ordering a set of line segments in $\Re^3$. In contrast, we are visibility sorting only the boundary facets of S, and utilizing the adjacency information to sort interior cells of S.

(d) Our sweep algorithm can also be stated in terms of a space sweep, in which we maintain in the sweep status the *slice* of B (the intersection of the sweep plane with the set B), which, in general, consists of a set of noncrossing, possibly nested, closed polygonal chains. If we maintain the 2-dimensional *trapezoidal decomposition* (with "vertical" direction in the z-direction) of the sweep slice, then the events correspond to (1) the sweep plane hitting a vertex of B, or (2) two z-parallel edges of the trapezoidal decomposition coming together (indicating a crossing point in $\mathcal{A}$). Then, the sweep status can be maintained using dynamic data structures for polygonal subdivisions, allowing point location and updates in time $O(\log^2 b)$ (see [18]).

(e) Our algorithm can be seen as constructing a "vertical decomposition" of the voids (the portion of $\Re^3$ that lies within the convex hull of S, but outside the union of the cells of S), where each trapezoid within the vertical decomposition corresponds to a pair of cells for which we discover the behind relationship. Computational geometry methods lead to improved theoretical worst-case bounds for vertical decompositions of arrangements; see [7, 28].

(f) As discussed in the next section, the implementation that we use in our experiments uses a simple grid-based hashing scheme to compute the ordering among boundary facets. The full implementation of the above sweep-based algorithm is under current development.

6 Implementation Details

Our implementation of XMPVO is based on the original MPVO code. XMPVO extends the existing code (for the implementation of MPVO based on a depth-first search topological sort) by computing extra depth-ordering dependencies, which are then used in the topological-sort (or splatting) phase.

We have a preliminary implementation of the sweep paradigm described in the previous section, as well as a simplified version of it based on a simple (regular) grid-based hashing scheme (explained below). Our experiments are conducted using the grid-based method, which is currently more robust, and can be applied to a wider variety of data (Sweep algorithms are delicate to implement robustly in floating point arithmetic, particularly in the presence of degenerate or possibly corrupted data.)

Since we decided to write the extension code in C++ using STL (the Standard Template Library), while the original code was written in C, we had to be careful not to replicate major data structures, and to maintain a clean interface between the two portions of code. This was not difficult, since the MPVO code has a nicely modular design. In the end, our current implementation of XMPVO adds less than 1,000 lines of code to MPVO.

XMPVO includes a modification of the traversal algorithm of MPVO in order to handle general meshes. We recall that MPVO does not compute and store an explicit DAG (directed acyclic graph) for its traversal; instead, the DAG information is stored implicitly by labeling each facet shared by two cells of the mesh as "inbounding" or "outbounding". Then, during a traversal, a cell is said to be *traversable* if it has no inbounding facets, *i.e.*, if its in-degree is zero, indicating that all cells that must precede it in the ordering have already been traversed (output).

The enhanced traversal phase for XMPVO requires three new data structures. These are designed not to require global searches of the data and to contain auxilliary data only for the boundary facets. The three data structures are:

(1) an explicit dependency graph, kept as an STL `map`, which keeps for each boundary cell C_i, a `vector` of the cells that depend on C_i being projected first (*i.e.*, that precede C_i in the ordering);

(2) for each boundary cell C_i, a semaphore that indicates how many boundary cells need to be projected before C_i can be projected;

(3) a list of the (boundary) cells that can be projected immediately (*i.e.*, the semaphore is zero).

The data structure definitions are:

```
map<int,vector<int>,less<int>> depend_on_MAP;
map<int,int,less<int>>         semaphore_MAP;
list<int>                      cells_to_render_LIST;
```

Figure 4 has the traversal code for XMPVO. The function `dfs` is basically the MPVO traversal code with one change: it only outputs boundary cells whose semaphore is zero, in addition to the usual restriction that the cell must have no inbounding facets.

Most of the new computational time of XMPVO is spent in computing `depend_on_MAP`. This computation is at the core of the sweep method described in the previous section. Here, we describe a simple, yet practical variant of the sweep approach, which is the method on which the reported experimental results are based.

The purpose of the sweep is to identify efficiently the dependencies implied by the behind relation. An alternative method is to perform a basic geometric hashing of the boundary facets, based on a regular grid decomposition of the viewing plane into fixed-size boxes. (Here, we will use "grid" and "box" when referring to this regular hashing grid and will use "mesh" and "cell" when referring to the original irregular grid S.)

We make two passes over the set of boundary facets. In the first pass, we look for ordering relations that are inferred by rays through crossing points, where the projected edges of two facets intersect; in the second pass, we examine relations that are inferred by rays through vertices of the projected facets. In each pass, we consider the boundary facets one by one; as we "insert" each facet f into the grid, we add f to a facet-list associated with each box of the grid that is intersected by the projection of f onto the viewing plane. (Actually, we have found it advantageous to do something even simpler — we add f to all those face-lists corresponding to boxes that are intersected by the (axis-aligned) bounding rectangle of the projection of f.) As facet f is inserted into the facet-list of some box, we test its projection for overlap with the projections of each facet already in the facet-list; for those that overlap, we compute the ordering of the respective containing cells and add this information to the `depend_on_MAP`. While this procedure is certainly worst-case quadratic in the size of each facet-list, we do some optimizations to help reduce the overall effort:

(a) the overlap comparison between projected facets includes a filter that tests for overlap between bounding rectangles first;

(b) if the pair of containing cells are already ordered (the relationship is known), no comparison is made;

(c) if the pair of facets having overlapping projections correspond to the same cell, then no ordering comparison is made (since none is needed); and,

(d) if the pair of facets do not have oppositely directed normals (one toward the viewpoint, one away from the viewpoint, as determined by the signs of simple dot products), then the corresponding cells need not be compared (some other pair of boundary facets obeying this condition will imply the necessary order relationship).

7 Results

In order to evaluate the performance of our current implementation, described in the previous section, we performed experiments using a single 194 Mhz R10000 CPU of an SGI Power Onyx. Table 1 summarizes our experiments. (See Figure 5 for an image computed with XMPVO.) We compare our results with those reported previously in Stein *et al* [32], and the (recently published) multi-tiled sort of Williams *et al* [34]. Our results show that XMPVO is orders of magnitude faster than these previous works, and in fact, scales much better with the dataset size.

No. Cells	Stein Sort	Multi-Tiled Sort	XMPVO
13,000	14 sec.	7.2 sec.	3.5 sec.
190,000	2880 sec.	162 sec.	25 sec
223,000	N/A	475 sec	48 sec.
600,000	29,370 sec.	570 sec.	11 sec.
1,000,000	54,516 sec.	900 sec.	12 sec.

Table 1: Comparative timings, in seconds, for visibility ordering using three methods: the sort reported by Stein *et al* [32], the multi-tiled sort of Williams *et al* [34], and the current implementation of the XMPVO algorithm. All timings were done using a single R10000 CPU of an SGI Power Onyx. Note that XMPVO is almost two orders of magnitude faster than the multi-tiled sort, and almost four orders of magnitude faster than the Stein *et al* sort.

The 13,000 cell result in Table 1 is from the use of XMPVO on a finite element mesh with tetrahedral cells. A volume rendering of data on this mesh is shown in Figure 5. The 190,000 cell result is from a tetrahedralization of the standard bluntfin data set. The 223,000 result is from the mesh for a finite element method simulation of air flow past an *F117a* jet aircraft, and uses tetrahedral cells. Due to the scarceness of large unstructured data sets, the remaining two results are from tetrahedralized versions of rectilinear meshes.

The reason the results for the 223,000 cell mesh are much slower than might be expected (when compared to timings for the larger datasets) are due to the fact that this mesh is adaptively refined, i.e., the mesh is created so that in areas in which the data field is changing most rapidly, the cell size is much smaller compared to the average cell size. Therefore, there are regions of the mesh which contain orders of magnitude more cells than other areas. Consequently, our preliminary (non-sweep) implementation of XMPVO which uses a regular grid decomposition into boxes as described in the previous section, gives a very unbalanced load to each box, basically defeating the speedup normally obtained by doing the decomposition. For example, in a 64 box decomposition, the mean number of cells/box is 54, while the max number of cells in a box is 73,450. Load balancing the boxes should result in a significant reduction in sorting time (in the same way that the tiling of the Stein *et al* [32] sort dramatically reduced the timings as shown in Table 1).

Not surprisingly, the performance of the current, grid-based code is dependent on the resolution of the grid that we use for hashing. The reported experiments were run with a 20-by-20 grid resolution (400 bins); by using a 10-by-10 grid, our code is slowed down by a factor of two. We anticipate that the full implementation of our proposed sweep algorithm will permit even faster execution times, and will remove the dependence on this resolution parameter. Another approach we are planning to explore is that of using a variable resolution grid (*e.g.*, quadtree) in place of our simple-minded bucketing scheme.

```
while(cells_to_render_LIST.empty() != true) {
  int cell_id = cells_to_render_LIST.back();
  cells_to_render_LIST.pop_back();
  // Need to decrement all the cells that depend on cell_id.
  di = depend_on_MAP.find(cell_id);
  for(vector<int>::iterator vi = (*di).second.begin();
      vi != (*di).second.end(); vi++)
    {
      map<int, int, less<int> >::iterator mi;
      mi = semaphore_MAP.find(*vi);
      (*mi).second--;
      if((*mi).second == 0) {
        cells_to_render_LIST.push_back( (*vi) );
      }
    }
  if (T[cell_id].notVisited)
    dfs(cell_id);
}
```

Figure 4: XMPVO traversal code.

8 Conclusion and Future Work

XMPVO fills a gap in unstructured grid depth-sorting. Previously, on one side, there were accurate techniques, such as [32], which were too costly to be used in practice; on the other side, there were techniques such as MPVO which were fast and suitable for interactive applications, but only provided either accurate support for limited types of meshes, or inaccurate support for general meshes.

A considerable amount of work remains. There is still an order of magnitude difference in speed between XMPVO and MPVO. For example, in the 11 seconds it takes to sort the 600,000-cell complex, about 90% of that time is spend in our depth-sorting of boundary facets. We emphasize, however, that our current implementation of XMPVO does not exploit all of the ideas outlined in Section 5; we made substantial simplifications in order to run the experiments reported here. We do have a prototype implementation of the sweep method, and we expect that, with it, XMPVO will take approximately the same time as MPVO, and we will be able, with the help of advanced graphics hardware, to render irregular grids in real-time.

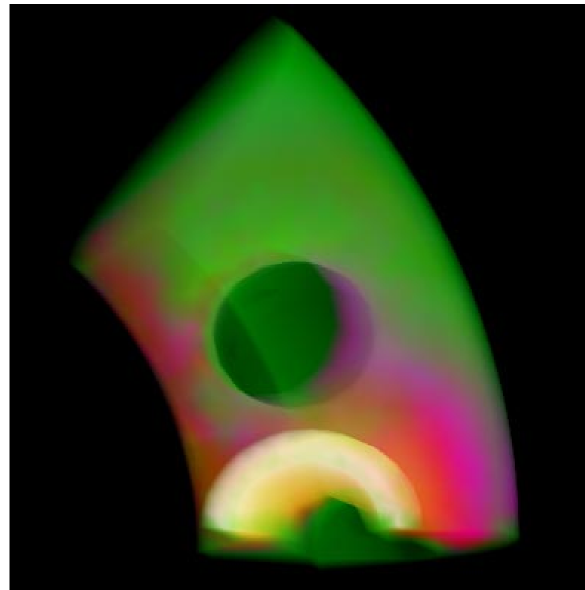

Figure 5: Image computed with XMPVO of a 13,000-cell complex.

Acknowledgements

We thank the referees for insightful comments and suggestions that help improve our paper. We regret that in the short time allowed for revisions we were unable to report results on the new (more efficient) sweep implementation of XMPVO described in Section 5. We would like to thank Nelson Max for discussions that lead to XMPVO. J. Mitchell largely conducted this research while a Fulbright Research Scholar at Tel Aviv University. He is also partially supported by NSF grant CCR-9504192, and by grants from Boeing Computer Services, Bridgeport Machines, Hughes Aircraft, Sandia National Labs, and Sun Microsystems.

References

[1] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.*, 22(4):794–806, 1993.

[2] P. K. Agarwal and J. Matoušek. On range searching with semi-algebraic sets. *Discrete Comput. Geom.*, 11:393–418, 1994.

[3] I. J. Balaban. An optimal algorithm for finding segment intersections. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 211–219, 1995.

[4] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39:1–54, 1992.

[5] B. Chazelle, H. Edelsbrunner, L. J. Guibas, R. Pollack, R. Seidel, M. Sharir, and J. Snoeyink. Counting and cutting cycles of lines and rods in space. *Comput. Geom. Theory Appl.*, 1:305–323, 1992.

[6] M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*, volume 703 of *Lecture Notes Comput. Sci.* Springer-Verlag, Berlin, Germany, 1993.

[7] M. de Berg, L. J. Guibas, and D. Halperin. Vertical decompositions for triangles in 3-space. *Discrete Comput. Geom.*, 15:35–61, 1996.

[8] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. *Algorithmica*, 12:30–53, 1994.

[9] M. de Berg, M. Overmars, and O. Schwarzkopf. Computing and verifying depth orders. *SIAM Journal on Computing*, 23:437–446, 1994.

[10] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[11] L. Carpenter, "The A-buffer, an antialiased hidden surface method," *Computer Graphics*, vol. 18, no. 3, pp. 103–108, 1984.

[12] D. P. Dobkin and M. J. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 4:3–32, 1989.

[13] H. Edelsbrunner. An acyclicity theorem for cell complexes in d dimensions. *Combinatorica*, 10:251–260, 1990.

[14] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9:66–104, 1990.

[15] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.*, 14(3):124–133, 1980. Proc. SIGGRAPH '80.

[16] M. P. Garrity, "Raytracing Irregular Volume Data," *Computer Graphics (San Diego Workshop on Volume Visualization)*, vol. 24, pp. 35–40, Nov. 1990.

[17] C. Giertsen, "Volume Visualization of Sparse Irregular Meshes," *IEEE Computer Graphics and Applications*, vol. 12, no. 2, pp. 40–48, March 1992.

[18] M. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 523–533, 1991.

[19] M. S. Karasick, D. Lieber, L. R. Nackman, and V. T. Rajan. Visualization of three-dimensional Delaunay meshes. *Algorithmica*, 19(1–2):114–128, Sept. 1997.

[20] N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *Comput. Graph.*, 24(5):27–33, 1990.

[21] J. S. B. Mitchell, D. M. Mount, and S. Suri. Query-sensitive ray shooting. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 359–368, 1994.

[22] M. Newell, R. Newell and T. Sancha, "Solution to the hidden surface problem," *Proc ACM National Conference*, 1972, pp. 443–450.

[23] M. Newell, "The utilization of procedure models in digital image synthesis," Ph.D. Thesis, University of Utah, 1974 (UTEC-CSc-76-218 and NTIS AD/A 039 008/LL).

[24] O. Nurmi. On translating a set of objects in two- and three-dimensional spaces. *Comput. Vision Graph. Image Process.*, 36:42–52, 1986.

[25] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

[26] M. Pellegrini. Ray shooting on triangles in 3-space. *Algorithmica*, 9:471–494, 1993.

[27] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[28] O. Schwarzkopf and M. Sharir. Vertical decomposition of a single cell in a three-dimensional arrangement of surfaces and its applications. *Discrete Comput. Geom.*, 18:269–288, 1997.

[29] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, New York, 1995.

[30] P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics (San Diego Workshop on Volume Visualization)*, vol. 24, pp. 63–70, Nov. 1990.

[31] C. T. Silva and J. S. B. Mitchell. The lazy sweep ray casting algorithm for rendering irregular grids. *IEEE Trans. Visualizat. Comput. Graph.*, 3(2):142–157, Apr. 1997.

[32] C. Stein, B. Becker, and N. Max, "Sorting and Hardware Assisted Rendering for Volume Visualization," *Symposium on Volume Visualization*, pp. 83–90, October 1994.

[33] P. Williams, "Visibility Ordering Meshed Polyhedra," *ACM Transactions on Graphics*, vol. 11, no. 2, 1992.

[34] P. L. Williams, N. Max, and C. Stein, "A high accuracy volume renderer for unstructured data," *IEEE Trans. on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 1–18, March 1998.

[35] R. Yagel, D. Reed, A. Law, P-W. Shih, and N. Shareef, "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing," *IEEE-ACM Volume Visualization Symposium*, pp. 55–62, Nov. 1996.