EUROGRAPHICS '99 / P. Brunet and R. Scopigno
(Guest Editors)

Volume 18 (*1999*), *Number 3*

Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids

João Comba[†] James T. Klosowski[‡] Nelson Max[§] Joseph S. B. Mitchell[¶] Cláudio T. Silva[||] Peter L. Williams[**]

Abstract

Direct volume rendering based on projective methods works by projecting, in visibility order, the polyhedral cells of a mesh onto the image plane, and incrementally compositing the cell's color and opacity into the final image. Crucial to this method is the computation of a visibility ordering of the cells. If the mesh is "well-behaved" (acyclic and convex), then the MPVO method of Williams provides a very fast sorting algorithm; however, this method only computes an approximate ordering in general datasets, resulting in visual artifacts when rendered. A recent method of Silva et al. removed the assumption that the mesh is convex, by means of a sweep algorithm used in conjunction with the MPVO method; their algorithm is substantially faster than previous exact methods for general meshes.
In this paper we propose a new technique, which we call BSP-XMPVO, which is based on a fast and simple way of using binary space partitions on the boundary elements of the mesh to augment the ordering produced by MPVO. Our results are shown to be orders of magnitude better than previous exact methods of sorting cells.

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

1. Introduction

Direct volume rendering based on projective methods, such as Max *et al.* [13] and Williams [21], 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 aliasing artifacts.

Williams' MPVO method assumes that the mesh is "well-behaved" (acyclic and convex). For such meshes, it computes a visibility order at interactive rates; however, if this method is applied to general datasets, it only computes an *approximate* ordering, resulting in visual artifacts when rendered. Traditionally, there has been a big performance gap between approximate visibility sorting techniques (*e.g.*, based on Williams' MPVO algorithm [21]), and exact solutions (*e.g.*, based on Stein *et al.* [20]). While approximate solutions provide reasonable results for "well-behaved" datasets, the artifacts they induce increase with the presence of non-convex boundaries, and "badly-shaped" cells.

Recently, Silva *et al.* [18] described XMPVO, a fast sorting algorithm based on an extension of the MPVO algorithm. They showed it is possible to generalize the MPVO algorithm, which exploits the ordering implied by adjacencies within the mesh, by simply augmenting the DAG created in Phase II of the MPVO algorithm. Thus, their technique removes the requirement of the MPVO algorithm that the mesh be convex. The augmentation involves the use of a sweep plane method to generate dependencies between external facets of the mesh. The XMPVO algorithm is orders of magnitude faster than the algorithm originally proposed by Stein *et al.* [20]. For n cells, with b boundary facets, XMPVO improves on Stein *et al.* by dropping the sorting complexity from $O(n^2)$ to $O(b^2+n)$. However, the speeds reported in Silva *et al.* [18] are still far from those required to drive current high-performance 3D graphics hardware.

In this paper, we propose a new technique, "BSP-XMPVO", which is an order of magnitude faster than that of Silva *et al.* [18]. We get this speed-up by moving the XMPVO view-dependent DAG augmentation, into a view-*independent* preprocessing phase, based on constructing an appropriate binary

† Stanford University; comba@cs.stanford.edu
‡ IBM T. J. Watson Research Center; jklosow@watson.ibm.com
§ Lawrence Livermore National Laboratory; max2@llnl.gov
¶ University at Stony Brook; jsbm@ams.sunysb.edu
|| IBM T. J. Watson Research Center; csilva@watson.ibm.com
** Lawrence Livermore National Laboratory; plw@llnl.gov

space partition (BSP) tree on the set of boundary facets of the mesh. By carefully utilizing the partial ordering information implied by the BSP-tree, together with the MPVO ordering, we are able to achieve an order of magnitude speed-up over XMPVO.

2. Previous Work

An algorithm, called the "Meshed Polyhedra Visibility Ordering" (MPVO) algorithm, for visibility ordering the cells of an acyclic convex mesh is described by Williams [21]. A similar algorithm to the MPVO Algorithm was developed independently by Max, Hanrahan and Crawfis [13]. Both algorithms were based on the work of Edelsbrunner described in his paper on the acyclicity of certain cell complexes [10]. The MPVO algorithm runs in linear time and uses linear storage. Williams [21] also described a heuristic, called the MPVONC algorithm, which sorts the cells of acyclic *non-convex* 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* are present. The MPVONC algorithm, in practice, is linear in time for most meshes. For some important classes of meshes (*e.g.*, rectilinear meshes and Delaunay meshes [10]), 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, then 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 [2, 7]. The binary space partition (BSP) tree algorithm [11], which is typically used to depth-sort polygons, is not suitable for visibility ordering large polyhedral meshes, since the splitting planes can readily cause an unacceptable increase in the number of polyhedra. (Paterson and Yao [16] have shown the a BSP of objects in space can have quadratic worst-case complexity; while this growth is typically not experienced in practice, even a constant-factor increase in the number of cells of the mesh is unacceptable for large volumetric datasets.) An A-buffer [6] 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.

Stein *et al.* [20] describe an algorithm for visibility ordering an arbitrary collection of acyclic non-intersecting convex polyhedra. This algorithm runs in time $O(n^2)$ (worst case) for n arbitrarily shaped, non-intersecting 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 [14, 15]. Williams *et al.* [22] 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 5.) Another related visibility-ordering technique based on Newell, Newell and Sancha is described by Snyder and Lengyel [19].

Theoretical results on exact visibility ordering are described by de Berg, Overmars, and Schwarzkopf [4], who give an algorithm requiring worst-case time $O(n^{4/3+\varepsilon})$ (for any fixed $\varepsilon > 0$) for determining an ordering or reporting that none exists (because of a cycle in the "behind" relation). Their algorithm utilizes a general framework for computing and verifying linear orders extending implicitly defined binary relations and it relies on the rather complicated dynamic data structure of Agarwal and Matoušek [1], which detects intersections between line segments in space and "curtains" (shadow surfaces cast by segments). Although not readily implemented, the theoretical significance of this work is that it shows that it is possible to determine, in *subquadratic worst-case time*, if a linear ordering exists, while avoiding the computation of the full behind relation (which is worst-case quadratic in the number of objects being ordered).

Karasick *et al.* [12], 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. A related technique for sorting cells using "power diagrams" is described in Cignoni and De Floriani [8].

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. 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*. We let B denote the set of boundary facets of S. A *boundary cell* is any cell having a boundary facet. The union of all boundary facets in B 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 *non-convex* 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, having convex cells, but arbitrary boundary. We let c denote the number of connected components of S. If $c = 1$, the mesh is *connected*; if $c > 1$, 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 [9]), 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 let v denote the viewpoint and let ρ_u denote the ray from 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. We assume that our input mesh S has a visibility ordering.

A Binary Space Partitioning tree (*BSP-tree*) is a data structure that represents a hierarchical convex decomposition of a given space (in our case, $\Re^3$). See [5, 11, 17]. Each node ν of a BSP-tree $\mathcal{T}$ corresponds to a convex polyhedral region, $P(\nu)$, of $\Re^3$; the root node corresponds to all of $\Re^3$. Each nonleaf node ν also corresponds to a plane, $h(\nu)$, which partitions $P(\nu)$ into two subregions, $P(\nu^+) = h^+(\nu) \cap P(\nu)$ and $P(\nu^-) = h^-(\nu) \cap P(\nu)$, corresponding to the two children, ν^+ and ν^-, of ν. Here, $h^+(\nu)$ (resp., $h^-(\nu)$) is the halfspace of points above (resp., below) plane $h(\nu)$.

Typically, a BSP-tree is built with respect to a given set of objects (*e.g.*, polygons or polyhedra), with the construction proceeding recursively until some stopping criterion is met (*e.g.*, that the region $P(\nu)$ contains portions of at most k objects, for some integer $k \geq 1$). Often, then, the partitioning planes are restricted to be from among those planes that support (contain) facets of the polyhedral objects; such BSP-trees are called *auto-partition* BSP-trees. Fuchs *et al.* [11] demonstrated that BSP-trees can be used for visibility ordering a set of objects (or, more precisely, an ordering of the fragments into which the objects are cut by the partitioning planes). The key observation is that the structure of the BSP-tree permits a simple recursive algorithm for "painting" the object fragments from back to front: If the viewpoint lies in, say, the positive halfspace $h^+(\nu)$, then we (recursively) paint first the fragments stored in the leaves of the subtree rooted at ν^-, then the object fragments $S(\nu) \subset h(\nu)$, and then (recursively) the fragments stored in the leaves of the subtree rooted at ν^+.

4. The BSP-XMPVO Algorithm

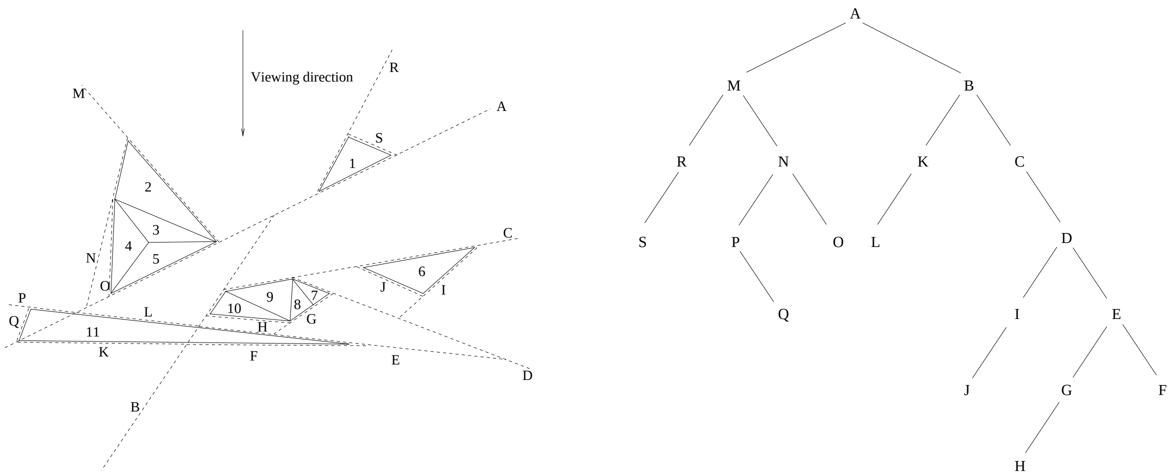
The goal of our BSP-XMPVO algorithm is to obtain a valid visibility ordering of the cells of the mesh S, assuming such an ordering exists. In order to do this efficiently, we build on the idea of the MPVO method, utilizing the simple-to-compute partial order induced by the adjacency graph of the mesh. We augment this partial order with additional dependencies, induced by the boundary facets of the mesh, in order to be able to complete it into a total order. The main idea of the BSP-XMPVO algorithm is to utilize the BSP-tree of the set B of boundary facets in order to determine these extra dependencies efficiently.

The MPVO algorithm of Williams [21] works by constructing (in preprocessing) the (undirected) adjacency graph G of the mesh S, and then, for any given viewpoint v, determining the corresponding orientation of each undirected edge of G, so that the directed edge points from C' towards a neighboring C if C' lies behind C (a test that is simply an evaluation of the viewpoint with respect to the plane equation for the shared facet between C and C'). In this case, we write $C' <_{ADJ} C$, in order to indicate the dependency, implied by adjacency, that C' must precede C in a visibility ordering of cells. The resulting directed graph defines a partial ordering ($<_{ADJ}$) of cells; topological sorting (in linear time) then produces a total ordering that yields the desired visibility ordering.

The correctness of the MPVO algorithm depends, however, on the mesh being convex. In the absence of these assumptions, there are additional dependencies that exist among cells of S that are not captured by the directed adjacency graph. For example, in Figure 1, a two-dimensional example is given to illustrate some basic principles. There, it is seen that cell 10 lies behind cell 5, and cell 11 lies behind cell 10, but neither of these dependencies is implied by the adjacency relation $<_{ADJ}$. (Indeed, the cells lie in distinct connected components of the mesh.)

In order to augment the ordering information given by the directed adjacency graph, we build a BSP-tree, $\mathcal{T}$, using an auto-partitioning of the set B of $b = |B|$ boundary facets of S. Specifically, our construction algorithm uses the common heuristic of selecting a partitioning plane, passing through a facet of B, that minimizes the number of elements of B within the region $P(\nu)$ that are split. (Our actual implementation does not examine all possible cuts, but rather selects a small number (*e.g.*, 10) of candidate cuts at random, and picks the best among these.) We let B' denote the set of facet fragments induced by $\mathcal{T}$.

It is known (*e.g.*, see [16, 5]) that the size of the BSP-tree (or, equivalently, the number $b' = |B'|$ of facet fragments) is quadratic ($\Theta(b^2)$) in the worst-case; however, in most realistic situations (*e.g.*, under assumptions of "fatness" of a set of



(a) (b)

Figure 1: *Example of a two-dimensional mesh, with 5 connected components. Dashed lines show the cuts in a BSP-tree, shown on the right. The viewpoint is assumed to be above and far away, so that the view direction is downward. The BSP-tree has been drawn so that the right child is explored before the left child, for this particular view direction. Thus, the BSP-tree traversal proceeds in the order* $A, B, C, \ldots, S$.

objects [3]), BSP-trees tend to exhibit near-linear complexity. Thus, we expect that $b' = O(b)$, in practice, and that the construction time for $\mathcal{T}$ is also near-linear in b. Further, we expect b to be much less than n (the total number of cells in S), in practice. (For a regular grid, one expects $b = O(n^{2/3})$.) Thus, we expect a very low overhead for the computation of $\mathcal{T}$, both in terms of memory and in terms of time.

Note that the BSP-tree $\mathcal{T}$ is cutting boundary facets into fragments, but we are specifically *not* partitioning any of the 3-cells of the mesh, as this would cost considerably more both in terms of time and memory.

We now describe how the BSP-tree $\mathcal{T}$ allows us to define two other types of dependencies among cells. Let C be a boundary cell of S, having boundary facet $c \in B$ that lies immediately behind C with respect to the viewpoint v. (In other words, any ray from v through c passes through the interior of C before exiting through facet c.) Let h denote the plane containing facet c and let ν be a node of $\mathcal{T}$ that corresponds to h. (There may be more than one such node, if c is split into fragments.) Then, h cuts the region $P(\nu)$ into two regions, $P(\nu^+) = h^+ \cap P(\nu)$ and $P(\nu^-) = h^- \cap P(\nu)$; without loss of generality, assume that $v \in h^+$, which implies that also $C \in h^+$ (since c lies immediately behind C with respect to v). Then, we define the following types of dependencies:

(a) We say that each boundary fragment c' on the boundary of C defines a *BSP-dependency* for cell C, written $c' <_{BSP} C$. The meaning of this dependency is that before C can be projected, each of its facet fragments (whether in front of C or behind C) must first be "projected." Facet fragments are also ordered according to the standard BSP-tree traversal for the boundary set B; we say that $c' <_{BSP} c''$, for facet fragments c' and c'', if c' precedes c'' in the BSP-tree traversal (as in the painter's algorithm of [11]). In reality, we are not "projecting" these facet fragments; rather, we are defining these dependencies so that we obtain implied dependencies among 3-cells.
In our traversal algorithm, at the instant that the last facet fragment of a boundary cell is projected, we simultaneously project that 3-cell.
For example, in Figure 1, cell 5 cannot be projected until its (unique) facet fragment is projected, and, from the BSP-tree traversal, this will not happen until all facet fragments in the halfspace below plane "A" have been projected; in particular, the two facet fragments of cell 10 must both be projected before the facet fragment of cell 5. This guarantees that cell 10 precedes cell 5 in our ordering, since cell 10 will be projected at the instant that its second (*i.e.*, last) facet fragment is projected.

(b) We say that C' has been *partially projected* if *at least one* (but *not* all) of the facet fragments on the boundary of C' has been projected; thus, by the BSP-dependencies, we know that C' itself has not yet been projected, if it is partially projected, since it cannot be projected before *all* of its facet fragments have been projected. We say that there is a *PPC-dependency* between a 3-cell C' and the 3-cell C, written $C' <_{PPC} C$, if C' has been "partially projected" at the time that the BSP traversal algorithm examines node ν, and cell C' lies behind cell C with respect to viewpoint v. Our algorithm maintains a list, the "PPC-list", of the set of cells that are currently partially projected.
For example, in Figure 1, the boundary fragments of cell 11 on planes "F" and "E" are the first two to be projected. At

```
Algorithm BSP-XMPVO_traversal(node, vp)
     /* The algorithm projects in back-to-front
        order the part of the mesh S
        corresponding to BSP tree node node
        with respect to the viewpoint vp. */
1.   if (node == NULL) then
2.       return;
3.   if (vp is in front plane)
4.       BSP-XMPVO_traversal(back(node));
5.       BSP-XMPVO_update_dep(node);
6.       BSP-XMPVO_traversal(front(node));
7.   else
8.       BSP-XMPVO_traversal(front(node));
9.       BSP-XMPVO_update_dep(node);
10.      BSP-XMPVO_traversal(back(node));
```

(a)

```
Algorithm BSP-XMPVO_update_dep(node)
     /* Updates the dependency counters
        for the cells whose faces
        lie on node's base plane. */
1.   for (i = 0; i < numPPC; i++)
2.       for (j = 0; j < numCutCells(node); j++)
3.           Check_update_ppc_dep_count (C_i, C_j);
4.   for (i = 0; i < numCutCells(node); i++)
5.       Update_PPC (C_i);
6.   for (i = 0; i < numCutCells(node); i++)
7.       Decrem_bsp_dep_count(C_i);
8.       if (num_inbound(C_i) == 0) and
9.         (bsp_dep_count(C_i) == 0) and
10.        (ppc_dep_count(C_i) == 0)
11.          enqueue(C_i);
12.  MPVO_traverse();
```

(b)

```
Algorithm MPVO_traverse()
     /* Modified MPVO traverse. */
1. while (deque(c) != false)
2.   output(C);
3.   for (i = 0; i < numFaces(C); i++)
4.       if arrow(i, C) == INBOUND
5.           continue;
6.       C_i = neighbor(C, i);
7.       Decrem_num_inbound(C_i);
8.       if ((num_inbound(C_i) == 0) and
9.         (bsp_dep_count(C_i) == 0) and
10.        (ppc_dep_count(C_i) == 0))
11.            enqueue(C_i);
```

(c)

Figure 2: *The complete BSP-XMPVO traversal algorithm. The node node of the BSP-tree is being projected. numCutCells(node) is the number of cells with facets that are on the cutting plane associated with node. C_i is one of these cells. Its dependency counts are given by: (i). num_inbound(C_i), the number of INBOUND arrows remaining (i.e., the number of $<_{ADJ}$-predecessors); (ii). bsp_dep_count(C_i), the number of BSP dependencies (i.e., the number of $<_{BSP}$-predecessors); and (iii). ppc_dep_count(C_i), the number of PPC dependencies (i.e., the number of $<_{PPC}$-predecessors). Also, numFaces(C) gives the number of facets of cell C (e.g., 4, in the case of a tetrahedron), and arrow(i, C) gives the type of the ith "arrow" for cell C (i.e., INBOUND if the neighbor is behind C, OUTBOUND otherwise). Update_PPC(C_i) inserts or deletes C_i on the PPC list; C_i is inserted when it is first visited, and bsp_dep_count(C_i) > 1, and it is deleted when it is one (since it will be decremented to zero). At the time cell C_i is deleted from the PPC, cells that have a dependency on it, are checked for potential projection with code similar to lines 8–11 in (b).*

this point, cell 11 is partially projected and is the sole element in the PPC-list. Then, as cells 7-10 are considered, cell 11 must be considered, as it generates a PPC-dependency; this prevents any of cells 7-10 from being projected before cell 11 is projected. The possibility that cell 11 generates a PPC dependency for cell 6 is also considered when we project the facet on plane C; however, it generates no PPC-dependency, since cell 11 does not lie behind cell 6, with respect to the viewpoint.

Note that, while we do not explicitly write the dependence on v, each of the relationships $<_{ADJ}$, $<_{BSP}$, and $<_{PPC}$ is dependent on the viewpoint.

Our BSP-XMPVO algorithm can be thought of as a means of running *in lock step* a BSP-tree traversal algorithm (on boundary facets), together with the MPVO traversal algorithm of Williams. Another interpretation is that we perform a topological sort, based on the partial order induced by the three types of dependencies $<_{ADJ}$, $<_{BSP}$, and $<_{PPC}$, which induce a partial ordering on the set $S \cup B'$ of 3-cells and facet fragments. As with standard topological sorting, we start by identifying those elements that have "in-degree" zero – these have no dependencies and can be projected immediately. With each projection of an element, we remove the dependencies that the element had on other elements, as given by the relations $<_{ADJ}$, $<_{BSP}$, and $<_{PPC}$, each of which can be thought of as a directed edge in a graph on the set $S \cup B'$ of cells and boundary facet fragments. Our implementation is based on keeping three separate dependency counters (*num_inbound*, *bsp_dep_count*, and *ppc_dep_count*), which give the number of dependencies of each of the three types. Once all of the dependency counters hit zero, an element is projectable, and then updates are made. Pseudo-code for the BSP-XMPVO traversal algorithm is contained in Figure 2.

Note that we compute the PPC dependencies on an as-needed basis. In order to speed up the test for PPC dependencies, we use a simple bounding sphere test on a candidate pair of cells, (C_i, C_j), in order to prune from consideration those pairs whose corresponding cones are disjoint.

The technical justification for the BSP-XMPVO method comes from two lemmas:

Lemma 1 The dependencies $<_{ADJ}$, $<_{BSP}$, and $<_{PPC}$ induce a partial ordering on the set $S \cup B'$.

Proof. By definition, if $C <_{ADJ} C'$ or $C <_{PPC} C'$, then C is behind C'; thus, a directed cycle could not consist purely of directed edges corresponding to $<_{ADJ}$ and $<_{PPC}$ (by the acyclicity assumed in the behind relation). Thus, a directed cycle, if it exists, must contain edges of type $<_{BSP}$. Assume that there is such a cycle and let c' be a facet fragment that corresponds to a node in the cycle; in fact, assume that c' is the *last* such facet fragment in the BSP-tree ordering given by the traversal. (Such a "last" element exists, since the BSP traversal induces a partial ordering on facet fragments.) Then, there exists a directed path from c' to some other facet fragment c'' (possibly, $c'' = c'$) in the cycle, with this path containing a node corresponding to a 3-cell; let C be the last such 3-cell. But this is a contradiction, since the only directed edges defined by our dependencies that are directed *out* of a 3-cell are those that link the 3-cell to another 3-cell ($<_{ADJ}$ or $<_{PPC}$). We conclude that there can be no directed cycle. □

The second lemma asserts that the three dependencies that our algorithm respects are sufficient for determining a visibility ordering:

Lemma 2 Any linear ordering that conforms with the dependencies $<_{ADJ}$, $<_{BSP}$, and $<_{PPC}$ gives a valid visibility ordering of S.

Proof. Suppose that cell C' lies behind cell C; i.e., $C' <_v C$. We must exhibit a directed path within the directed graph induced by $<_{ADJ}$, $<_{BSP}$, and $<_{PPC}$, from C' to C. Since $C' <_v C$, there is a ray ρ from the viewpoint v that intersects C before C'. If the portion, $\overline{rr'}$, of the ray ρ between $\rho \cap C$ and $\rho \cap C'$ lies within the union of the cells S, then no boundary effects are present, and there exists a directed path within the directed adjacency graph, from C' to C, so we are done. Otherwise, the segment $\overline{rr'}$ exits the mesh and then reenters, at least once. Let $\overline{ab}$ denote one such segment of ρ that lies outside the mesh, with a the closer endpoint to v. Then, a lies on a boundary facet fragment, c_1, of a cell C_1 such that $c_1 <_{BSP} C_1$, and b lies on a boundary facet fragment c_2, of a cell C_2 such that $C_2 <_{BSP} c_2$. If, at the time in the traversal that we visit the node corresponding to plane h_1 that contains c_1, the cell C_2 is in the list of partially projected cells (the PPC-list), then we know that $C_2 <_{PPC} C_1$, establishing the necessary link in the partial ordering. Otherwise, at the time of visiting the node for h_1 the cell C_2 has already been projected, and therefore also c_2 (which precedes c_1 in the ordering $<_{BSP}$.) □

Computational Complexity. In comparing the performance of our algorithm to XMPVO, which takes $O(b^2+n)$ time, where b is the number of cells in the boundary; our technique takes time $O(bp+n)$, where $p = |\mathrm{PPC}|$ (since we need to examine all elements of the PPC-list each time we update dependencies). The PPC actually changes as the algorithm progresses, but an upper bound on its size can be obtained by counting the boundary cells which are cut by more than one face of the BSP. These are exactly the cells that will be included in the PPC. Fortunately, in practice the PPC-list does not grow to be big (*e.g.*, in our experiments reported below, the PPC-list never grew above 0.3%, and averaged about 0.1% of the elements), since most mesh elements tend to be well-shaped and do not "spike in" behind other elements (as does cell 11 in Figure 1). The fact that p is most often less than 0.3% of the total numbers of cells, and in general much smaller than b (which can be 5%, or more of the total cells), makes our algorithm essentially linear in the total number of cells. Further, our bounding sphere-based test for possible dependencies allows for a quick filtering of those PPC-list elements that clearly are not behind the cell in question.

5. Results

The implementation of the BSP-XMPVO method is relatively straightforward, due to the simplicity of the algorithm. In fact, exclusive of the MPVO portion of the code, and the BSP-tree generation, BSP-XMPVO consists of just 600 lines of C++ code.

To evaluate the performance of BSP-XMPVO, we ran a battery of experiments. We measured basic statistics of the BSP-tree construction (Table 2) and, of course, the time required to obtain a visibility sorting of the cells (Table 1). We have experimentally validated the correctness of our code by concurrently running the HIAC depth-witness-check code of Williams *et al.* [22] during our cell projections. This check projects the cells in the visibility order determined by our algorithm and determines (by looking at the depth buffer) whether a cell has been projected out of order.

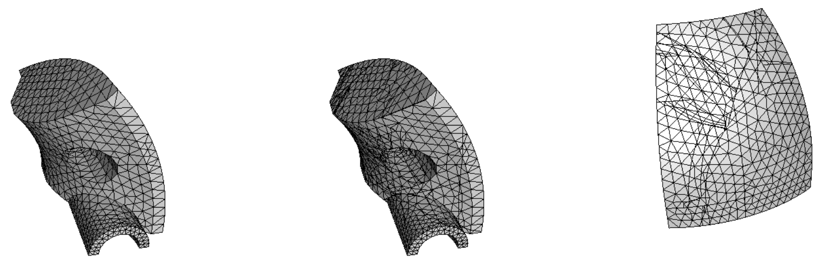
Because of constraints in machine availability at this time, we are forced to report timings on two separate machines: the BSP-XMPVO and MPVONC times are reported on an IBM RS/6000 43P, with a 333MHz PowerPC 604 processor (this is a slower processor than the ones available on the high-end PowerMacs), while all other times are on a single 194 MHz R10000 CPU of an SGI Power Onyx, as in Silva *et al.* [18]. We estimate the 43P to be slightly faster than the R10000 (between 10%-30%). We report results on three irregular grid datasets, ranging in size from roughly 13,000 cells to a mesh of over 240,000 cells.

5.1. BSP-Tree Performance

Our BSP-tree construction method uses a simple heuristic in an attempt to get reasonably small trees that, in practice, avoid the known worst-case quadratic behavior. At every node, we evaluate a small set of randomly chosen candidates, and choose the cutting plane that minimizes the number of cuts of the input data. As the number of candidates increases, so does the BSP tree generation time. We have chosen to use 10 candidates as our default, as this provides us with enough flexibility to avoid cutting many cells, while at the same time is not overly costly in construction time. Figure 3 shows the effect of the BSP-tree generation on the boundary facets of the 13,000 cell dataset.

Table 2 summarizes our construction results for all of the datasets. We feel it contains some interesting data. For instance, there is no direct correlation between the number of boundary faces and the depth of the BSP tree. The depth of the BSP tree is more related to the complexity of the mesh boundary (*e.g.*, non-convexities). Our BSP tree construction algorithm is working very well, as can be seen in the last column of the table. In the worst case, the number of BSP faces is only two times the original number of boundary faces. This is further justification of our choice of 10 candidates when constructing the trees.

Since BSP traversal time is dominated by the number of nodes and not by the depth of the tree (as every node in the tree is visited during each traversal), we decided to optimize the construction for minimizing the number of unnecessary splits, which has the side effect of increasing the depth of the tree.



(a) (b) (c)

Figure 3: *The boundary of the 13,000 cell complex: (a) shows the original boundary facets; (b) and (c) show two views of the BSP-facets. The BSP cuts are apparent. In the center of (a) and (b), a hole which runs through the center of the mesh can be seen.*

No. Cells	Stein Sort	Multi-Tiled Sort	XMPVO	MPVONC	BSP-XMPVO
13,000	14 sec.	7.2 sec.	3.5 sec.	0.07 sec.	0.37 sec.
190,000	2,880 sec.	162 sec.	25 sec.	0.70 sec.	2.5 sec.
240,000	N/A	475 sec.	48 sec.	0.90 sec.	2.9 sec.

Table 1: *Comparative timings, in seconds, for visibility ordering using five methods: (1) the sort reported in Stein et al.* [20]*, (2) the multi-tiled sort of Williams et al.* [22]*, (3) the XMPVO algorithm of Silva et al.* [18]*, (4) the MPVONC algorithm of Williams* [21]*, and (5) our BSP-XMPVO algorithm. The first three timings were performed on an R10000 CPU of an SGI Power Onyx; MPVONC and BSP-XMPVO were timed on a 333MHz PowerPC 604. Note that BSP-XMPVO is an order of magnitude faster than XMPVO.*

5.2. Visibility Sorting Times

We compare our results with the algorithm of Stein *et al.* [20], the multi-tiled sort of Williams *et al.* [22], the XMPVO algorithm of Silva *et al.* [18], and MPVONC of Williams [21]. Table 1 summarizes our sorting times. We see that for all three (irregular) datasets, our BSP-XMPVO algorithm is over an order of magnitude faster than the XMPVO algorithm, and almost as fast as MPVONC. Compared to the other two approaches, our method looks even more promising. We can sort about 80,000 cells per second.

6. Conclusion

In this paper, we have proposed a fast new method for visibility ordering unstructured grids. We have achieved an order of magnitude improvement over the most recent improvements of Silva *et al.* [18]. The main innovation was the use of a coordinated traversal algorithm, based on the MPVO ordering of Williams [21], together with a carefully augmented traversal of a BSP-tree based purely on the boundary facets of the mesh, which led to an improvement in running time from $O(b^2+n)$ to $O(b|\mathrm{PPC}|+n)$, where b is substantially larger than $|\mathrm{PPC}|$.

Our BSP-XMPVO method makes approximate visibility-ordering techniques substantially less attractive as an option for rendering irregular grids by projective methods. This helps to close the gap between rendering regular and irregular grids, which historically has shown a disparity of orders of magnitude in speed of rendering. It also opens up the possibility of using irregular grids to approximate volumetric datasets defined on regular grids, in much the same way that triangulated irregular networks (TINs) have been used to approximate and compress regular digital elevation map (DEM) datasets. We are currently exploring this direction, as well as the parallelization of our algorithm.

Acknowledgements

J. Mitchell is partially supported by Bridgeport Machines, Sandia National Labs, Seagull Technologies, Sun Microsystems, and the National Science Foundation (CCR-9732220).

References

1. P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.*, 22(4):794–806, 1993.
2. M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal.* volume 703 of *Lecture Notes Comput. Sci.* Springer-Verlag, Berlin, Germany, 1993.
3. M. de Berg. Linear size binary space partitions for fat objects. *Proc. 3rd Annu. European Sympos. Algorithms*, LNCS Vol. 979, Springer-Verlag, pp. 252–263, 1995.

No. Cells	Const. Time	BSP Depth	(b) No. Bndy Facets	(b') No. Facet Fragments	No. BSP Nodes	b' / b
13,000	3.0 sec.	233	2,760	5,584	4,925	2.02
190,000	6.4 sec.	44	13,516	16,263	283	1.20
240,000	5.8 sec.	43	9,884	14,482	2,912	1.46

Table 2: *Statistics of BSP-tree construction. Construction time is based on building the BSP-tree of the boundary facets B of the input data, using 10 random candidate cutting planes at each node. During the construction, some of the b boundary faces are cut, resulting in b' facet fragments. Times computed on an IBM RS/6000 43P with a 333MHz PowerPC 604.*

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

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

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

7. 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.

8. P. Cignoni and L. De Floriani. Power Diagram Depth Sorting. *Proc. 10th Canad. Conf. Comput. Geom.*, August 1998.

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

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

11. 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.

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

13. 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.

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

15. 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).

16. 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.

17. H. Samet. *Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods*. Addison-Wesley, Reading, MA, 1989.

18. C. T. Silva, J. S. B. Mitchell, and P. Williams. An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes. *ACM Symposium on Volume Visualization*, pp. 87–94. October 1998.

19. J. Snyder and J. Lengyel. Visibility Sorting and Compositing Without Splitting for Image Layer Decomposition. Proc. SIGGRAPH '98, pp. 219–230, 1998.

20. C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. *1994 Symposium on Volume Visualization*, pp. 83–90. ACM SIGGRAPH, October 1994.

21. P. L. Williams. Visibility Ordering Meshed Polyhedra. *ACM Transactions on Graphics*, vol. 11, no. 2, 1992.

22. 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. 37–54, March 1998.

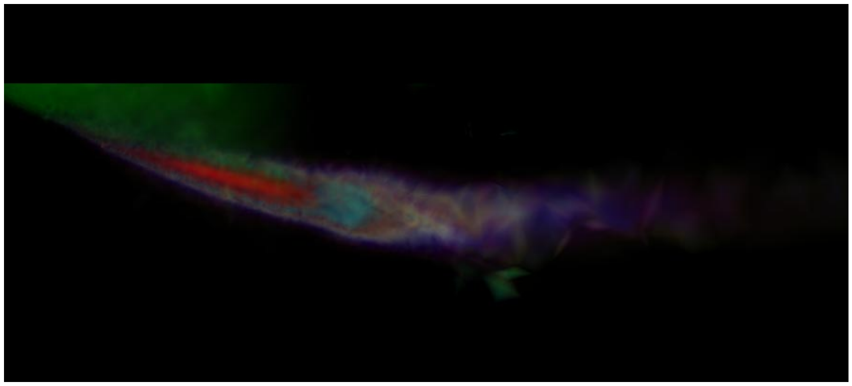


Figure 4: *Image computed with BSP-XMPVO of a 240,000-cell complex.*

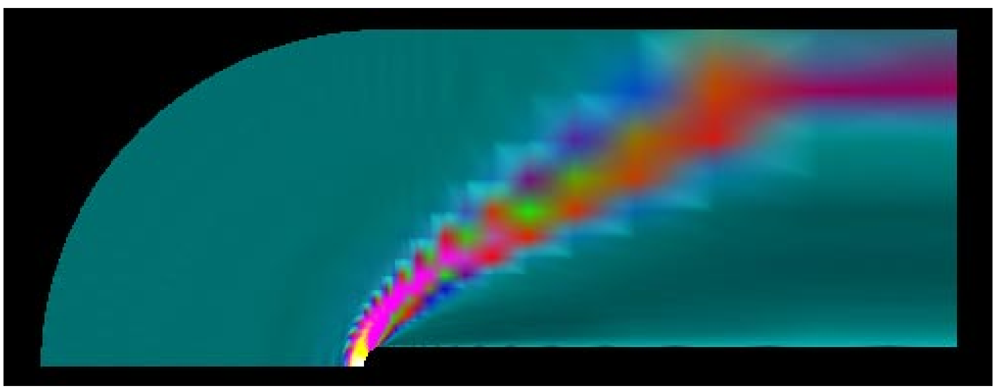


Figure 5: *Image computed with BSP-XMPVO of a 190,000-cell complex.*

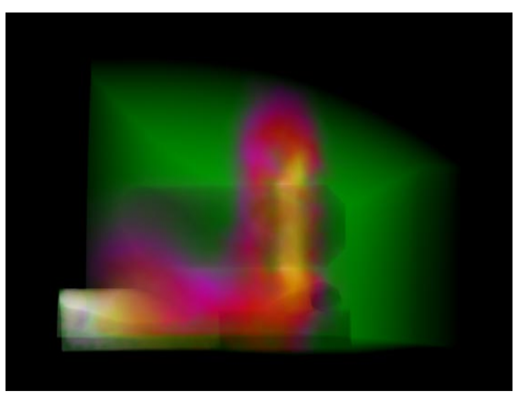


Figure 6: *Image computed with BSP-XMPVO of a 13,000-cell complex.*