Preprint
UCRL-JC-143126

Efficient and Exact Visibility Sorting of Zoo-Mesh Data Sets

R. Cook, N. Max, C. Silva, P. Williams

This article was submitted to
The Institute of Electrical and Electronics Engineers 2001
Symposium on Parallel and Large-Data Visualization and Graphics,
San Diego, CA, October 21-26, 2001

April 1, 2001

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

This work was performed under the auspices of the United States Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Efficient and Exact Visibility Sorting of Zoo-Mesh Data Sets

Richard Cook[1,2], Nelson Max[1,2], Claudio Silva[3] and Peter Williams[2]

April 1, 2001

Abstract

We describe the SXMPVO algorithm for performing a visibility ordering zoo-meshed polyhedra. The algorithm runs in practice in linear time and the visibility ordering which it produces is exact.

1 Introduction

Since the data sets our group at the Lawrence Livermore National Laboratory deals with are on the order of several terabytes, most of our visualization tools incorporate many approximations in order to achieve some degree of interactivity. To understand the effect of these approximations on image quality, we are developing a highly accurate (HIAC) volume rendering system [1], that uses as few approximations as possible, to use as a gold standard for comparing images from the approximate volume rendering algorithms.

Generally speaking, volume rendering techniques based on projection methods are more accurate than techniques that use ray casting since projection methods allow subpixel acummulation and can avoid sampling artifacts. They can also take advantage of coherence in large cells, and of hardware rendering and compositing. However, projection methods require a visibility ordering of the cells of the mesh being rendered, since the cells are composited in back-to-front order.

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 defined on curvilinear or unstructured meshes, often with different element (cell) types in the same mesh. These meshes are sometimes referred to as *zoo* meshes because there is a zoo of element types, i.e. hexahedra, tetrahedra, pyramids, prisms, etc. as shown in Figure 1. When the mesh is unstructured, computing a visibility ordering of the cells is a nontrivial problem. The problem is further compounded because finite element method meshes may have (a) cells with nonplanar faces, e.g. twisted hexahedra, (b) meshes with disconnected regions, or (c) nonconvex boundaries.

Williams [2] describes an sorting algorithm for convex meshes called the meshed polyhedra visibility ordering (MPVO) algorithm, and a heuristic extending the MPVO algorithm to nonconvex meshes called the MPVONC algorithm. However, MPVONC relies on explicit connectivity information between cells, and thus is not guaranteed to generate an accurate ordering in the case where one cell occludes another across a gap without any cells between them or in the case of cells with nonplanar faces (see Figure 3). Sort algorithms have been described by Cignoni, et al [3, 4, 5] and Wittenbrink [6]. However, these aren't guaranteed to produce an accurate visibility ordering of the cells. Since we are developing a gold standard volume renderer, we want a guaranteed accuracy visibility ordering.

Silva, et al [7], improve the results of Williams [2] using ray casting or a sweep to augment the dependencies. Comba et al [8] further improve the results of Williams with the BSP-XMPVO alogrithm, which uses a BSP tree of the boundary faces to exactly (thus the X for *eXact*) extend the basic MPVO algorithm. Building the BSP structure is a very slow step for this algorithm, however. In [9], Krishnan, Silva and Wei describe a hardware-assisted visibility ordering algorithm, but it is not clear that this algorithm, which accurately sorts polygons, can be extended to perform an accurate sort of volume cells.

In this paper, we describe a visibility ordering algorithm which we call the Scanning Exact MPVO visibility ordering (SXMPVO) algorithm that accurately sorts the cells of unstructured meshes, where the cells of the meshes may have non planar surfaces (provided the faces project one-to-one onto the image plane, rather than into "bowtie" quadrilaterals), the boundary of the mesh may be nonconvex, the mesh may have cells of different types (i.e., may be a zoo mesh) and the mesh may be disconnected.

In Section 2 we describe the SXMPVO visibility sorting algorithm, in Section 3 we present timing results for zoo meshes of different sizes, and finally in Section 4 we give our conclusions and describe future work.

1.1 Cell Sorting

Since cells are projected from back to front, an important issue in cell projection is determining which cells are in front of other cells. Note that the relationships thus discovered are not a total ordering of the cells, so a correct total ordering for a set of cells based on the *in-front-of* and *behind* relationships for pairs of cells is usually not unique. 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. This results in a correct rendering of the image from back to front. Such an ordering can be computed in a number of ways. The goal of this computation is always to correctly and efficiently sort the cells into a visibility ordering. The determination of order may, for example, be based on the construction

[1] University of California, Davis
[2] Lawrence Livermore National Laboratories
[3] AT&T Laboratory–Research

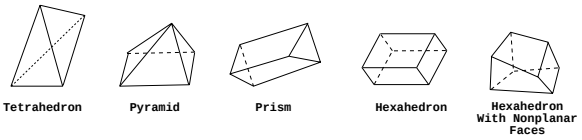

Figure 1: The basic types of cells which SXMPVO can sort.

of a priority graph derived from the overlap of polygons from a certain point of view [10]. Another method is to create a binary space partition (BSP) tree and traverse this tree to find the correct rendering order [11]. Alternatively, one can partially order the cells based on adjacency information and the orientation of faces and do a search through the resulting graph to determine a correct depth ordering, as for example in the MPVO sorting algorithm (see [12] and [2]).

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. The MPVO algorithm works by creating a partial ordering of the cells based on their relationships to their neighbors across a shared face. Once such a partial ordering is given, a topological sort of the graph representing the cells yields a correct sort, provided that the cells are a convex acyclic set of meshed polyhedra. Unfortunately, many data sets violate the convex mesh constraints of MPVO. For example, cells may be in a nonconvex mesh, or there may be multiple disconnected components to the mesh. In this case, a cell may occlude another cell with which it does not share a common face. We would like an exact sorting algorithm for such cases, because errors of this nature can cause noticeable differences in image quality and make the image impossible to reliably interpret.

An early exact method by Stein, et al. [13] ran in $O(n^2)$ time for n cells, which is unacceptable for large data sets. We can improve the required time by noting that the extra relationships only occur when an external cell face in the mesh occludes another external cell face. Using this fact, Silva et al. [7] used ray shooting queries at each intersection and vertex of the projection complex to discover the overlapping exterior face relations, improving the run time to $O(n + b^2)$, where b is the number of exterior faces. However, this algorithm in practice was still too slow to drive even the older graphics hardware available at the time at interactive frame rates, because the relationships had to be rediscovered each time the viewpoint changed. Another solution detailed in [8] is to identify all the exterior faces and perform a binary space partition of the entire data space into subspaces, using the planes upon which the exterior faces each lie as the partition basis. The graph has the advantage of being view-independent. The BSP appears to add considerable processing overhead, however, and seems sluggish in some practical cases, when the BSP tree becomes deep and unbalanced, leading to an $O(n)$ time cost of many searches into it, thus giving it an $O(n^2)$ flavor. (See [14] for a more detailed description of these sorting methods.)

```
int *tt; /*array of cell indices*/

typedef struct Subface {
  Cell *shared[2];
  float A, B, C, D;
  char arrow;
  pseudoNeighbor *mNeighbors;
} Subface;

struct NewQuadFace {
   short concave;
   short nsubfaces;
   struct Subface subface[2];
   };

typedef struct NewCell {
   char subdivided, oldsubdivided;
   char numbInbound;
   char type;
   char cycleTestBit;
   char notVisited;
   char nverts;
   int  UsedTIndex;
   short  projected;
   struct NewCell *Parent;
   struct NewFace **face;
   int vert[4];
} Cell;
```

Figure 2: Data structures to support HIAC cell rendering.

2 Algorithm

In the preprocessing phase, the entire data set is read from disk into core memory. The input data for HIAC are zoo-mesh elements or cells [1]. Such cells (see Figure 1) may be quadrilateral-faced hexahedra or "bricks," pentahedra (either "triangular prisms" or "quadrilateral-based pyramids"), or tetrahedra. Their faces may be planar or non-planar, so they may also be nonconvex cells.

The data are stored on disk in the Silo unstructured mesh data format [15]. This format provides information about the cells and the associated vertices in the mesh. Since this only supplies the connectivity between the vertices, HIAC creates other adjacency information as needed. For example, it is necessary to know, for a given cell face, which cells share the face.

In core memory, the zoo elements are stored as an internal representation, as `Cells` (see Figure 2). Each `Cell` has pointers to its vertices and faces. Each face has a pointer to two `Subface` structures, to allow approximation of nonplanar quadrilateral faces as two triangular subfaces and to allow for tetrahedral subdivision of non-simplices.

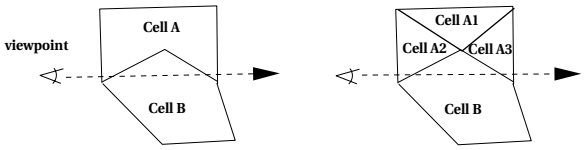

Figure 3: An example of how a twisted face between cells A and B cause their front/back relationship to be indeterminate. Note that there would be no problem with the twisted face if the viewpoint were looking up in the picture from the bottom of cell B. One way to fix the problem is by subdividing cell A as shown.

2.1 Sorting Phase

Prior to actually sorting the cells, a partial ordering on the cells is created by marking each cell face with an arrow. The arrow indicates which of the two cells sharing the face is in front of the other cell with respect to the viewer position. This is calculated using the plane equation for the face and the position of the viewpoint. Then an initial pass is made through all the faces incrementing the `numbInbound` counter for the cell to which each face's arrow points.

Cells with twisted, non-planar faces may create the possibility that a ray from the viewpoint exits a cell, passes through another cell and reenters the first cell (see Figure 3). Such a situation causes an undefined visibility ordering (which of these two cells would be in front of the other?) and thus is problematic for visibility sorting. Such cells are tested to see if they do pose a problem and are subdivided into tetrahedra if necessary to avoid any visibility-ordering cycles. (See [14] for details on this subdivision process.)

The cells are topologically sorted from back to front using the partial ordering. (Thus, by the classification in [16], HIAC is a *sort-middle* algorithm.) HIAC has used a number of sorting algorithms as it has evolved. Currently, the sorting is done by one of three methods depending on user preference for speed or accuracy. It may be done in a correct but relatively slow manner by using the *Binary Search Partition–Exact Meshed Polyhedron Visibility Ordering* (BSP-XMPVO) algorithm [8] or it may be done using the simpler and faster *Meshed Polyhedron Visibility Ordering – Non-Convex* (MPVONC) algorithm [2], which performs efficiently but may not find a correct visibility ordering (see section 1.1).

Due to the slowness of the exact BSP-XMPVO, we have developed a new sorting algorithm, the *Scanning Exact MPVO* (SXMPVO) algorithm, which computes dependencies between exterior faces by scan converting them and directly comparing the depth of each face for every pixel to compute the ordering. While this sounds onerous and inefficient, in fact it can be sped up using a few tricks, which are described later in this section. Once the exterior face dependencies are computed, these new adjacencies are then used just as regular adjacencies across shared faces in the regular MPVO algorithm to traverse the cells and produce a final ordering.

The SXMPVO algorithm works as follows. In HIAC, all faces are either quadrilateral or triangular, and all exterior quadrilateral faces are divided into two triangular subfaces, so all polygons to be scanned will be triangular subfaces (see figure 2 for a list of the data structures involved). The b exterior subfaces among the n `Cell`s with a total of v vertices in the data set are identified in $O(n+v)$ time and sorted in a preliminary manner in $O(b \log b)$ time (this is similar to the ZSWEEP algorithm [17]), using quicksort on the distance of their centroids from the viewpoint. An array of pointers to `pixelListEntry` structures (see Figure 4) is created, one per pixel in the final image.

The exterior subfaces are then removed from their sorted queue one at a time and scan converted, sampled at the same pixel resolution and location (i.e., at pixel centers as per OpenGL [18]) as will appear in the final image. As each pixel of an exterior subface is encountered, a new `pixelListEntry` is created with a pointer to the subface and its distance from the viewpoint at the pixel center. The `pixelListEntry` is then placed in order of distance from viewpoint in the linked list of entries for the pixel.

At this point, when all subfaces have been scanned into pixel list entries, the linked list for every pixel contains a list of exterior faces in reverse depth order, which represent the subfaces which are encountered while traversing a viewing ray through that pixel center, from far away towards the viewpoint. The subfaces referred to by adjacent `pixelListEntry`s in this list therefore alternate between referring to a top-facing[3] subface of one cell and a bottom-facing subface of another cell, because a view ray must first have entered one cell to exit another, and, with the exception of the topmost subface, cannot enter one cell without having exited one previously.

Our aim is to discover dependencies between external subfaces by traversing the pixel lists. Such dependencies are important only across "gaps" in the data set where no cells intervene, because the normal MPVO algorithm already handles ordering between adjacent cells sharing a face. Since each pixel list is ordered from back to front, these gaps are places in the pixel list where two adjacent `pixelListEntry`s are a top-facing subface followed by a bottom-facing subface. The first subface in a given pixel list is a back-facing subface and is thus discarded, and then pairs of `pixelListEntry`s are used, if they exist, to infer information about occluding external subfaces as described shortly.

In each discovered pair of `pixelListEntry`s for a pixel, the first refers to a top-facing, farther `Subface` and the second refers to a bottom-facing, nearer `Subface`. The second `Subface` by definition occludes the first and there are no cells between them for this pixel. Therefore, the second `Subface` is added to the *pseudo-neighbors* list of the first `Subface` if it does not already exist there (that is, if it was not previously discovered and inserted in another nearby pixel – no duplicate entries are allowed). The `numbInbound` counter for the nearer cell (associated with the second `Subface`) is also incremented, if the dependency is new, to reflect the new discovery of the dependency of that cell on the farther. In this way, when examining any cell, it is known how many cells it depends on and which cells depend on it. This is necessary for the BFS described below.

At this point, each `Subface` of every `Cell` has an arrow as described in [2] which tells whether the `Cell` is in front of the sub-

[3] We use the term "top-facing" to refer to subfaces with outward normals which have a positive component in the direction of the viewpoint, and "bottom-facing" to be those with a negative component.

face (a so-called INBOUND arrow) or behind it (OUTBOUND arrow) with respect to the viewpoint. Each `Subface` also has a linked list of dependencies on other subfaces discovered by scanning. Each `Cell`'s `numbInbound` counter properly indicates the sum of the number of `Cells` adjacent to it which it occludes (these will share a subface with an arrow marked INBOUND relative to the occluding `Cell`) and the number of `Cells` which it occludes due to pseudo-neighbor relationships discovered by scanning.

We now can do a breadth-first search of the `Cells` to discover a valid painting order as follows. First, a search is done through the `Cells` to find *source cells* with `numbInbound` $= 0$. These are `Cells` which have no incoming dependencies on other `Cells`. We put all source `Cells` into a global queue, `gCellQueue`, and pop the first `Cell` from the head of the queue. As explained in [2], `Cells` whose `numbInbound` is zero do not occlude any `Cells` and may be immediately rendered, so we put this `Cell`'s unique identifying integer index into the `tt[]` array (see Figure 2), indicating it may be rendered first. It may be that other `Cells` depend on the one we have just discovered, so we now decrement their `numbInbound` counters by one to indicate that we have satisfied one dependency condition for the dependant `Cell` by placing the current `Cell` into the `tt[]` array before the dependent `Cell`. Any `Cell` which thus has its `numbInbound` counter decremented to zero is now placed at the tail of `gCellQueue`. The next `Cell` is popped from `gCellQueue` and the process is repeated until no more cells remain.

No `Cell` is placed on `gCellQueue` until its `numbInbound` count is zero and all cells that it could occlude are already in the rendering queue `tt[]`, so no cell is placed on the rendering queue out of order. If `gCellQueue` becomes empty and cells remain which have not been listed in `tt[]`, there must be a visibility cycle in the data, i.e., a circular set of cells, each of which obscures the next, and no visibility sort is possible unless one or more of the offending cells is subdivided.

This is guaranteed to be correct to the resolution of the final image because of the transitivity of the front-back relationship and the correctness of the individual front-back relations discovered. However, the sort may not be absolutely correct if faces which overlap are not sampled at the points where they do overlap. We took care to sample all faces at actual pixel locations so that the final image was correct even if the sorting contained invisible errors.

This algorithm is nominally still $O(b^2)$, because that is the worst case for the number of new dependencies which exist among exterior faces, and each dependency must certainly be considered at least once. However, the algorithm runs in $O(W \cdot H + A + b + n)$ time in practice for small and medium data sets, where W and H are the window width and height in pixels and A is the total area of the exterior faces in square pixels. For applications where the area of each exterior subface polygons tends to be fairly homogeneous and where the number of exterior polygons is on the order of the screen resolution or higher, which may be quite common in practice for reasonably large scientific data sets, this is roughly a linear algorithm in b, which one expects to be $O(n^{2/3})$. Insertions into the linked lists for the pixels are almost always at the head of the queue due to the presorting by centroid. Our tests showed that insertions into the pixel queue occur at the head of the queue at least 99% of the time.

```
/* used in scan conversion: */
typedef struct richEdge {
  double dxdy, dd_dy;
} richEdge;

typedef struct externalSubface {
  Cell *mParentCell;
  char mFaceID, mSubfaceID;
  int verts[3];
  Subface *mSubface;
  float dist;
} externalSubface;

GLOBAL externalSubface
*gExternalSubfaces;
GLOBAL int *gCellQueue;

typedef struct pixelListEntry {
  externalSubface *mExternalSubface;
  double d;
  struct pixelListEntry *next;
} pixelListEntry;
```

Figure 4: Data structures to support the SXMPVO sort.

Inserting new neighbors into the `Subfaces` could potentially be an expensive operation, as the length of each list of neighbors for each face could be on the order of b, the number of exterior faces. Since the entire list must be checked to avoid duplication, this implies a worst-case $O(b^2)$ complexity for the overall algorithm. However, it is highly likely that a dependency of a cell `Subface` upon another cell `Subface`, if it already exists in the dependency list for the `Subface`, is at or very near the top of the list. This is due to the natural locality of memory references generated by the pixel scanning process, i.e., the next pixel over is very likely to express the same dependencies for many of the same sets of faces, unless there are unusual cell geometries involved (for example, long thin faces which overlap a fat one). To further encourage this behavior, however, and to avoid repeatedly finding duplicate dependencies deep in a `Subface`, when a dependency is found to exist already, it is relocated at the top of the list if not there already. In this way, insertions will be done in nearly constant amortized time for almost all data sets. Having avoided this potential slowdown, creating the actual dependencies from the pixel lists is $O(W \cdot H \cdot d)$, where d is the average depth complexity of the exterior faces in the scene.

3 Results

3.1 SXMPVO Results

The SXMPVO algorithm is at its heart both fast and accurate. It was desirable to get an exact sorting algorithm working and this was the intent of SXMPVO; it was a pleasant surprise just how fast it operates, although there are issues with the

Algorithm Step	Time (seconds)
Build External Face Array	1.29
Sort External Subfaces	0.08
Scan Subfaces	
and Add Dependencies	0.56
Discover Source Cells	0.17
Do Cell BFS	1.53

Table 1: Breakdown of execution time for the SXMPVO sort for a data set with 372,581 cells and 128,466 vertices.

underlying data structures in HIAC that must still be resolved. For example, using BSP-XMPVO on a data set of 5,661 cells with 3,456 exterior faces took 13.2 seconds to sort while the same data set required just 0.25 seconds using SXMPVO. The time to execute BSP-XMPVO is about 60% creating the BSP data structures and 40% traversing the BSP once complete. An advantage of the BSP approach, however, is that the BSP data structure is view-independent and so can be reused for different views once created. (In the SXMPVO algorithm, the plane equations of the faces can be reused, but the arrow directions must be recomputed for a new viewpoint.)

The time spent in SXMPVO for smaller data sets is in scanning the subfaces and adding new dependencies as expected. However, for medium sized data sets, the actual BFS and creation of the array of external faces starts to dominate the sorting process due to the necessity of searching the cells for the appropriate subfaces. This would very likely be sped up by integrating the creation of the external subface array into the program initialization phase and adding more information to each cell to aid in the BFS traversal.

The great speed advantage from SXMPVO lies in the relatively small amount of computational geometry involved, which is little more difficult than calculating slopes during scan conversion of the faces, while BSP-XMPVO must do much more to create the BSP. Furthermore, the BSP algorithm has much functional recursion, also a slowdown. It is possible that improvements in the BSP method might be gained if different data structures were employed such as self-balancing trees, but the creation of such a tree will not be faster and so it seems likely that SXMPVO is preferable.

SXMPVO, as discussed in Section 2.1, proceeds in several distinct phases. In the first, it creates a partial ordering on the cells by marking shared subfaces with arrows, then it subdivides cells as needed to accomodate twisted faces and other sorting hazards, then it enumerates all external faces and builds an array for them. The heart of the algorithm is when it extends the partial ordering of the cells to include the extra dependencies between external faces, and finally, it must discover the source cells and do a BFS to compute a final total order. Table 1 breaks down the execution time typical for these SXMPVO phases. Figure 5 shows how each of these phases perform under various workloads ranging from about 100,000 cells to nearly 2,000,000 cells. The time required varies nearly linearly to the workload for these examples.

It bears repeating here that, as mentioned in Section 2.1, not only the number of cells impacts the sorting time. A careful look at Figure 5 shows an superlinear trend; this is most

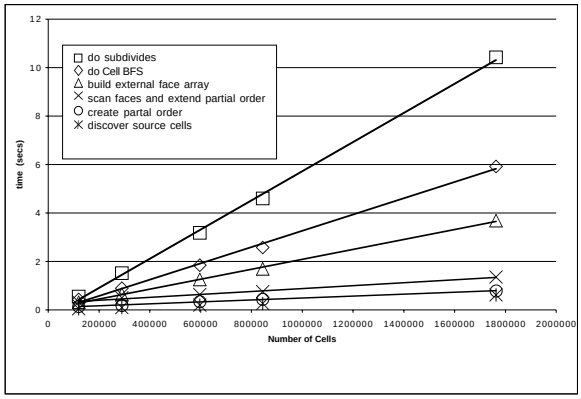

Figure 5: Performance of the different SXMPVO phases vs. workload.

Data Set Size (Number of Cells)	MPVONC Time (secs.)	SXMPVO Time (secs.)
31,946	0.17	1.11
308,600	2.22	4.87
1,794,000	6.65	15.01
2,246,250	8.15	16.44

Table 2: Comparison of execution times for the SXMPVO sort algorithm and the MPVONC algorithms.

likely due to the increasing percentage of overlapping cells in the larger data sets (and thus an increase in the average depth complexity), due to the way the data sets were generated. Thus, more dependencies per cell must be created and more external subfaces must be considered per cell. Similarly, data sets with identical numbers of cells but with differing projection areas of external cell subfaces will take different times to scan the faces, even though the number of external dependencies found may be identical. Therefore, it is difficult to exactly depict in a two-dimensional graph the various effects which cell topology exerts on SXMPVO's performance. Given these effects, the near-linearity which Figure 5 shows is remarkable.

A comparison of the SXMPVO algorithm's performance with that of MPVONC is shown in 2. The time for both MPVONC and SXMPVO grow approximately linearly in the number of cells. SXMPVO times exhibit a greater deviation (making it appear even sublinear for this example), due to the algorithm's dependence on other parameters such as depth complexity and image resolution as discussed in Section 2.1. Since SXMPVO must always determine and respect every relationship that MPVONC does as well as creating and following extra relations between external faces, it is always slower than MPVONC for every data set and view angle.

Acknowledgement

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

References

[1] Peter Williams, Nelson Max, and Clifford Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):37–54, January-March 1998.

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

[3] P. Cignoni and L. De Floriani. Power diagram depth sorting. In *10th Canadian Conference on Computational Geometry*, 1998.

[4] P. Cignoni, C. Montani, and R. Scopigno. Tetrahedra based volume visualization. In H.-C. Hege and K. Polthier, editors, *Mathematical Visualization – Algorithms, Applications, and Numerics*, pages 3–18. Springer Verlag, 1998.

[5] P. Cignoni, C. Montani, D. Sarti, and R. Scopigno. On the optimization of projective volume rendering. In *Visualization in Scientific Computing '95*, pages 58–71. Springer Computer Science, 1995.

[6] C. Wittenbrink. Cellfast: Interactive unstructured volume rendering. In *Proceedings IEEE Visualization'99, Late Breaking Hot Topics*, pages 21–24, 1999. also available as Technical Report, HPL-1999-81R1 , Hewlett-Packard Laboratories.

[7] C. T. Silva, J. S. B. Mitchell, and P. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *ACM Symposium on Volume Visualization*, pages 87–94, October 1998.

[8] João Comba, James Klosowski, Nelson Max, Joseph S. B. Mitchell, Claudio T. Silva, and Peter L. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. In *EUROGRAPHICS*, number 3, 1999.

[9] Shankar Krishnan, Claudio Silva, and Bin Wei. A hardware-assisted visibility-ordering algorithm with applications to volume rendering. Technical report, AT&T Laboratory — Research.

[10] M.E. Newell, R.G. Newell, and T.L. Sancha. Approach to the shaded picture problem. In *Proceedings of the ACM National Conference*, pages 443–450, 1972.

[11] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a-priori tree structures. *Computer Graphics*, 14:124–133, July 1980.

[12] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *Computer Graphics*, 24:27–33, November 1990.

[13] C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. In *SIGGRAPH Symposium on Volume Visualization*, pages 83–90, October 1994.

[14] Nelson Max, Peter Williams, and Claudio Silva. Approximate volume rendering for curvilinear and unstructured grids by hardware-assisted polyhedron projection. Technical report, 2000.

[15] Silo user's guide, revision 1. Technical report, Lawrence Livermore National Laboratories, Livermore, CA, 94550, August 2000. UCRL-MA-118751, ftp://ftp.llnl.gov/pub/meshtv/meshtv4.1.1/silo.ps.

[16] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. In *IEEE Computer Graphics and Applications*, pages 23–32, July 1994.

[17] R. Farias, J. Mitchell, and C. Silva. Zsweep: An efficient and exact projection algorithm for unstructured volume rendering. In *2000 Volume Visualization Symposium*, pages 91–99. ACM Press, October 2000.

[18] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification*. Silicon Graphics, Incorporated, April 1 1999. Version 1.2.1.