

# A Hardware-Assisted Visibility-Ordering Algorithm With Applications to Volume Rendering

Shankar Krishnan, Cláudio T. Silva, and Bin Wei

AT&T Labs-Research  
180 Park Avenue  
Florham Park, NJ 07932  
{krishnas, csilva, bw}@research.att.com  
<http://www.research.att.com>

**Abstract.** We propose a hardware-assisted visibility ordering algorithm. From a given viewpoint, a (back-to-front) visibility ordering of a set of objects is a partial order on the objects such that if object  $A$  obstructs object  $B$ , then  $B$  precedes  $A$  in the ordering. Such orderings are useful because they are the building blocks of other rendering algorithms such as direct volume rendering of unstructured grids. The traditional way to compute the visibility order is to build a set of visibility relations (e.g.,  $B <_p A$ ), and then run a topological sort on the set of relations to actually get the partial ordering. Our technique instead works by assigning a *layer* number to each primitive, which directly determines the visibility ordering. Objects that have the same layer number are independent, and can be placed anywhere with respect to each other. We use a simple technique which exploits a combination of the  $z$ - and *stencil* buffers to compute the layer number of each primitive. As an application of our technique, we show how it can be applied to obtain a fast unstructured volume rendering algorithm, which improves recent reported results. In this paper, we present our new algorithm and its implementation in OpenGL. We also discuss its performance and some optimizations we performed for some recent graphics hardware architectures.

## 1 Introduction

The original motivation for this work comes from volume rendering, but our work has other applications, which include image-based rendering acceleration, animations with selective display, efficient rendering with transparency [19]. The main contribution of this paper is a technique for computing an ordered set of (acyclic) primitives which can be rendered by painter's algorithm.

There are primarily two main approaches for exploring graphics hardware in volume rendering. One approach is to build new hardware, specialized for volume rendering. Quite possibly, the most visible example of this approach is VolumePro [14], which is based on the Cube-4 architecture of Pfister and Kaufman [15]. Another approach is to leverage existing graphics hardware, such as the texture-mapping based technique of Cabral *et al.* [1]. Although different, these two techniques have something fundamentally in common: the volumetric data model used is the same, that is, each volumetric grid is basically a regularly spaced 3D matrix of voxels.

An alternative technique for exploring graphics hardware for volume rendering is the Projected Tetrahedra (PT) algorithm of Shirley and Tuchman [17], which uses the traditional 3D polygon-rendering pipeline. This technique renders a volumetric grid by breaking the volumetric grid into a collection of tetrahedra. Then, each tetrahedra is rendered by *splatting* its faces on the screen. This technique explores the graphics hardware for approximating the volume rendering lighting computations and generates high-quality images. One of the nice properties of the Shirley and Tuchman’s approach is that it is not specific to a regular volumetric grid. Another nice property is that it is quite efficient in terms of the number of triangles it needs to render per primitive. Wittenbrink [27] found experimentally that, on average, one needs 3.4 triangles per tetrahedron. On a fast graphics board, such as the recent Nvidia GeForce, one can potentially render several million tetrahedra per second.

In the domain of rendering of digital terrain models, the trends have been towards converting the data into some form of adaptive tessellations [12] instead of rendering a large collection of small triangles. Extending this notion to three dimensions, PT seems like the adaptive analog for volume rendering as opposed to approaches based on texture mapping hardware. In this sense, PT is conceivably a superior approach, even though it is currently being used to render only unstructured grids [20].

In order to apply PT, one needs to compute a visibility-ordering of the cells. Williams’ Meshed Polyhedra Visibility Ordering (MPVO) algorithm [25] developed in the early 1990s provides a very fast visibility-ordering algorithm suitable for use in real-time rendering of unstructured grids. MPVO, which runs in linear time, works by exploiting the intrinsic connectivity of the unstructured grids and works well for well-behaved meshes (acyclic and **convex**). MPVO has recently been extended for general acyclic meshes by Silva *et al.*’s XMPVO [18], which lead to an  $O(n + b^2)$  algorithm (where  $n$  is the total number of cells, and  $b$  is the number of cells in the boundary of the mesh). The work of Silva *et al.* relies on being able to compute a visibility-ordering of the boundary cells by first performing a sufficient set of ray shooting queries, then running a topological sort on the visibility relations found to infer the ordering. Comba *et al.* [6] further improved these results with BSP-XMPVO to  $O(n + bp)$  (where  $p$  is the size of a small subset of the boundary cells), and leading to an order of magnitude improvement in sorting times over XMPVO. This technique requires a view-independent preprocessing which amounts to building a BSP tree of the boundary faces. Unfortunately, even BSP-XMPVO is not able to sort cells at millions of cells per second, which is necessary to drive high-end graphics boards at full speed. Another one of BSP-XMPVO’s disadvantages is the fact that it is not possible to handle visibility ordering of dynamic meshes efficiently, which might arise from the extension to volumetric meshes of techniques such as the continuous level of detail algorithm of Lindstrom *et al.* [12] (these techniques usually require the geometry being rendered to change continuously as to match the user movement). Furthermore, the implementation of BSP-XMPVO is relatively complex.

The fundamental computation which XMPVO and BSP-XMPVO are built on is the ability to visibility-order the boundary cells of the mesh. It is this problem that we attack in this paper.

In this paper, we propose a new hardware-assisted visibility-ordering algorithm. At a high-level, our algorithm can be seen as a hardware implementation of the XMPVO algorithm, but there are some significant differences. XMPVO (and most traditional visibility ordering algorithms) first build a sufficient set<sup>1</sup> of pairwise visibility relations (e.g.,  $B <_p A$ ), and then in a second phase, a topological sort is needed on the set of relations to actually get the ordering. Our technique instead works by assigning a *layer* number to each primitive, which directly determines the visibility ordering. To compute the layer number of each primitive, we make extensive use of the graphics hardware. In particular, we exploit a combination of the *z*- and *stencil* buffers.

In the rest of this paper, we first describe some related work in Section 2. In Section 3, we describe our new algorithm and some optimizations. In Section 4, we report some experimental results, including how our technique compares to XMPVO and BSP-XMPVO. We finish the paper in Section 5 with final remarks and our plans for future work.

## 2 Related Work

We let  $v$  denote the viewpoint and let  $\rho_u$  denote the ray from  $v$  through the point  $u$ . A *visibility ordering*,  $<_v$ , of a set of primitives  $\mathcal{P} = \{p_1, p_2, \dots, p_n\}$  from a given viewpoint,  $v \in \mathbb{R}^3$ , is a linear order on  $\mathcal{P}$  such that if  $p \in \mathcal{P}$  visually obstructs  $p' \in \mathcal{P}$ , partially or completely, then  $p'$  precedes  $p$  in the ordering:  $p' <_v p$ . In general,  $p' <_v p$ , if and only if there exists a ray  $\rho$  from the viewpoint  $v$  such that  $\rho \cap p \neq \emptyset$ ,  $\rho \cap p' \neq \emptyset$  and the intersection point of  $\rho$  with  $p$  is before the intersection point with  $p'$  along the ray. Theoretical results on exact visibility ordering are described by de Berg, Overmars, and Schwarzkopf [7], who give an algorithm requiring worst-case time  $O(n^{4/3+\epsilon})$  (for any fixed  $\epsilon > 0$ ) for determining an order or reporting that none exists (because of a cycle in the “behind” relation). 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).

Work on visibility ordering in computer graphics was pioneered by Schumacker *et al.* [21]. An earlier (complete) solution to computing a visibility-order was given by Newell, Newell, and Sancha (NNS) [13] which is the basis for several recent techniques [20]. The NNS algorithm starts with a rough ordering in *z* (depth) of the primitives, then for each primitive, it fine tunes the ordering by checking whether other primitives actually precede it in the ordering.

Building on [21], Fuchs, Kedem, and Naylor [9] developed the Binary Space Partitioning tree (*BSP-tree*), which is a data structure that represents a hierarchical convex decomposition of a given space (in our case,  $\mathbb{R}^3$ ) (see [8, 9, 16]). Each node  $\nu$  of a BSP-tree  $\mathcal{T}$  corresponds to a convex polyhedral region,  $P(\nu) \subset \mathbb{R}^3$ ; the root node corresponds to all of  $\mathbb{R}^3$ . Each non-leaf node  $\nu$  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,  $\nu^+$  and  $\nu^-$ , of  $\nu$ . Here,  $h^+(\nu)$  (resp.,  $h^-(\nu)$ ) is the

<sup>1</sup> *Sufficient* in the sense that it is possible to extend such pairwise relations into a valid partial order. In general, one has to formally show that this is the case. See [18].

halfspace of points above (resp., below) plane  $h(\nu)$ . Fuchs *et al.* [9] 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  $\nu^-$ , then the object fragments  $S(\nu) \subset h(\nu)$ , and then (recursively) the fragments stored in the leaves of the subtree rooted at  $\nu^+$ .

It is important to note that the BSP-tree does not actually generate a visibility order for the original primitives, but for *fragments* of them. Comba *et al.* [6] show how to recover the visibility order from the sorted fragments. There are a few issues in using BSP-trees for visibility-ordering. Building a BSP-tree is a computationally intensive process. Thus, handling dynamic geometry is a challenge. Using techniques from the field of “kinetic” data structures, Comba [5] developed an efficient extension of BSP-trees for handling moving primitives. At this time, his technique requires apriori (actually analytical) knowledge of the motion of the geometry to efficiently perform local changes on the BSP-tree as the primitives move.

Another technique for visibility order is described in Silva *et al.* [18]. In that paper, a well-chosen (small) set of ray shooting queries are performed, which compute for each primitive (at least) its successor and predecessor in the visibility ordering. By running a topological sort on these pairwise relations, it is possible to recover a visibility-order. One of the shortcomings of this technique is that it might actually compute a larger portion of the visibility graph than necessary to compute the ordering. Since the ray shooting queries are relatively expensive both in time and memory, this can be inefficient.

Another class of sorting techniques are based on power-sorting, see the work of Cignoni *et al.* [2, 4, 3]. These techniques are quite fast, since they reduce the 3D sorting problem to a one dimensional sort, which can be done quite efficiently with quicksort. Unfortunately, these techniques make limiting assumptions about the shape of the actual grids (*e.g.*, a Delaunay triangulation, see [27]) and their use for general meshes would, in general, cause visibility-ordering problems. For highly tessellated unstructured grids, these errors in visibility-ordering are mostly imperceptible, but for adaptively sampled volumetric grids where big cells would be close to small cells, sorting errors might be large.

Snyder and Lengyel [19] present an incremental visibility sorting algorithm, similar in some respects to the NNS algorithm [13]. Their algorithm, despite having a worst-case running time of  $O(n^4)$  is shown to be quite fast in practice. In order to cull the number of visibility relations they need to maintain, Snyder and Lengyel employ several optimizations, such as the use of kd-trees, and the tracking of overlaps of the convex hulls of the geometric primitives. Their algorithm is able to explore temporal coherency, and in fact is optimized for dynamic geometry. They also propose a technique for correct rendering in the presence of cycles.

The VBuffer technique of Westermann and Ertl [22] is related to our work. In their algorithm, they exploit the graphics hardware for performing depth-sorting of volumetric primitives by rendering the cells on a plane perpendicular to the scanline; then they

use the imprinted cell ids, and their geometric relationship, to guide the volume integral calculation. Our volume rendering technique is quite different, since we do not use the hardware to sort all the volumetric primitives as they do, but only the boundary, and use MPVO relations for the interior of the volume. Because of this, we require adjacency information, which they do not. Although the two techniques are quite different, both of them share several of the same implementation issues, such as the use of puffers, the disabling of all lighting calculations, and the reading back of the OpenGL buffers to get primitive ids. Our experimental results show that our technique is considerably faster than the VSbuffer. Quite possibly, this is due to the fact that for typical datasets, we require a much smaller number of buffer reads.

Building complicated data structures to solve the visibility-ordering problem is a fairly difficult task. Given that interactivity is of utmost importance in most applications, it would be prudent to try and solve this problem in hardware at some pre-specified resolution. As other researchers have found (see, for instance, Hoff *et al.* [10], Westermann and Ertl [23, 22]) exploiting the ever-faster graphics hardware available in workstations and PCs, can lead to simpler, and more efficient solutions to our rendering problems. Our work is motivated by this trend.

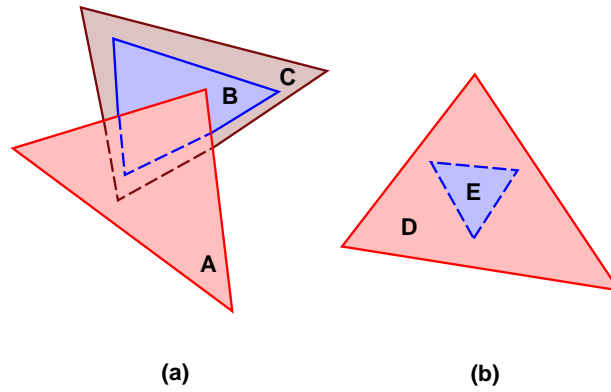
### 3 Our Algorithm

For the sake of argument, assume that we are trying to extract a front-to-back visibility order. The basic idea is to start with the complete collection of primitives, and extract the primitives in *layers*, that is, maximally independent set of polygons which do not relate to each other in the visibility order. The algorithm works by extracting a single layer from the current set of primitives. We basically keep doing this until no more primitives can be removed. At this point, if the set of primitives not assigned a layer number is not empty, one of the following two conditions are true: (a) the remaining (un-classified) primitives are either orthogonal to the viewing direction, hence we can not really classify them with respect to each other or the rest of the polygons, or (b) they contain a cycle, and our algorithm does not handle cycles. (See Snyder and Lengyel [19] for a technique which can be used to handle the cycles.)

We now explain our algorithm. We assume we have access to the *z*-, *stencil*, and *color* buffers. Also, for the sake of simplicity in presentation, we assume the input is composed of triangles, and all the transformation matrices have been handled by code that is outside of this subroutine. We start with some basic notation.  $\mathcal{T}$  is used to denote the set of triangles which have not been classified (notice that it changes over time);  $\mathcal{F}$  is the current layer being extracted;  $\mathcal{T}_i$ , for a given  $i$ , is the set of triangles assigned to be in the  $i$ th layer. During our algorithm, the stencil buffer is sometimes disabled, but whenever it is enabled, it is set to increment any time a triangle would have been projected into those pixels. In OpenGL, the stencil buffer would be configured as such:

```
glStencilFunc(GL_ALWAYS, ~0, ~0);
glStencilOp(GL_KEEP, GL_INCR, GL_INCR);
```

In our algorithm, we make extensive use of the item buffer technique, where triangles are rendered with different colors, which can be used to identify them by reading



**Fig. 1.** (a) In this situation, triangle A occludes parts of triangles B and C, while B is completely occluded by C from the opposite direction. During the first scan, pixels covering B and C are present in the top layer,  $\mathcal{F}$ . Note that in step 4, we remove triangles from back to front. Since C completely occludes B, we have to go through step 4 multiple times to extract the correct layering. (b) Simple case where the depth complexity of  $\mathcal{F}$  is always 1.

back the color buffer. We refer to this process as *reading and scanning* the buffer in the rest of our discussion. Reading buffers refers to performing the `glReadPixels` call, while scanning buffers refers to the process of traversing the pixel arrays, and obtaining the primitive ids, and depth complexity. Here is our algorithm:

While  $\mathcal{T} \neq \emptyset$ , loop,

1. Clear the color buffer; disable the stencil buffer; configure z-test to `GL_LESS`, while clearing it to 1.0 (far).
2. Render  $\mathcal{T}$ .
3. Read back the color buffer, and assign to  $\mathcal{F}$  any triangle that belongs to the current color buffer. Note that these triangles are *potential* candidates to be in the current layer, but they do not necessarily belong to the current layer, since they might be obscured by some other triangle. (See Fig. 1.)

A necessary and sufficient condition for  $\mathcal{F}$  to be a layer is that the depth-complexity of  $\mathcal{F}$  can be at most one. The idea in the next phase of our algorithm is to use the stencil buffer to test for this condition. In fact, by properly setting the z-buffer, it is possible to identify exactly the triangles which do not belong to the current layer by looking at pixels in the stencil buffer which have a depth-complexity larger than one.

4. Do
  - (4a) Clear and enable the stencil buffer; clear the color buffer; configure z-test to `GL_GEQUAL`, while clearing it to 0.0 (near).
  - (4b) Render  $\mathcal{F}$ .
  - (4c) Read back the color and stencil buffers. For each pixel in the stencil buffer which is larger than one, remove the corresponding triangle from  $\mathcal{F}$ , and re-insert it in  $\mathcal{T}$ . Since we rendered the scene from the back, we are necessarily removing a triangle that is covered by one or more other triangles.

Note that if we never find a pixel which has depth-complexity higher than two, we can leave the loop at this point. Otherwise, we need to keep removing triangles from the back of  $\mathcal{F}$ , until the depth-complexity of each pixel is at most one.

- (4d) Assign  $\mathcal{T}_i = \mathcal{F}$  for the current layer number, and increment the layer number.

While depth-complexity of  $\mathcal{F} > 1$ .

5. In case no triangles have been removed from  $\mathcal{T}$  since step (1) of the algorithm (that is, the number of elements in  $\mathcal{T}$  has not changed), we can stop the algorithm, and say the remaining triangles either are part of a cycle, or they are orthogonal to the view direction.

### Optimizations in the number of buffer reads and scanning.

It is straightforward to turn the description of our algorithm above into working C++ code. If we assume we have  $n$  triangles in a scene, the worst-case performance of our algorithm is  $O(n^2)$ , since all the triangles can be behind a single pixel. But this is rarely the case. Assuming the depth complexity of the scene is  $d$ , the complexity of the algorithm is much closer to  $O(nd)$ . Each triangle is rendered multiple times, and can potentially be rendered  $O(d)$  times. Often, rendering is not the bottleneck. As we show in Section 4, almost all of the time is spent in reading the color and stencil buffers, and scanning them (depending on image size, triangle count, and architecture limitations). Also, as layers are extracted, the actual footprint of a typical layer decreases quite rapidly (see Fig. 5). Thus, most of this time is wasted. We propose a simple modification of our algorithm which greatly improves the overall performance. It is based on the simple fact that once a pixel is not covered by a triangle after being rendered in step (2), it will never be covered again. Using this fact, it is simple to use a subdivision scheme of dividing the image into blocks, which keeps track of pixel coverage in every block, and avoids reading and scanning it the next time such an operation is needed. In most architectures, the larger the block size, the better the bandwidth that is achieved in reading back the buffers, although this tends to max out usually somewhere around a  $512 \times 512$  block. Based on our experiments, a  $64 \times 64$  blocking scheme works best on various hardware platforms.

## 4 Experimental Results

We use OpenGL to implement the depth sorting algorithm. We tested the performance on several workstations, including an SGI Octane, and an HP PC. We are only presenting the data collected from the faster Octane and the HP PC. We will briefly describe the performance of other machines after our discussion on the two representative ones. The SGI Octane we use has 300MHz MIPS R12000 CPU and 512MB main memory running IRIX 6.5 with an EMXI graphics board. The HP workstation we use has dual 450Mhz Pentium II Xeon processors and 384MB main memory running windows NT 4.0. The graphics subsystem is HP fx6. There are two versions of our algorithm. One is the naive implementation of the depth sorting algorithm and the other is the optimized

model	# of vertices	# of triangles	depth win256	depth win512
Bones	2156	4204	19.7	18
Mannequin	689	1355	10.8	12.4
Phoenix	8280	2760	9.6	11.1
Sphere	66	129	2.8	2.5
Spock	1779	3525	17.7	18.9

**Table 1.** Characteristics of the five models and their average depths for the window sizes of  $256 \times 256$  and  $512 \times 512$  over 30 frames.

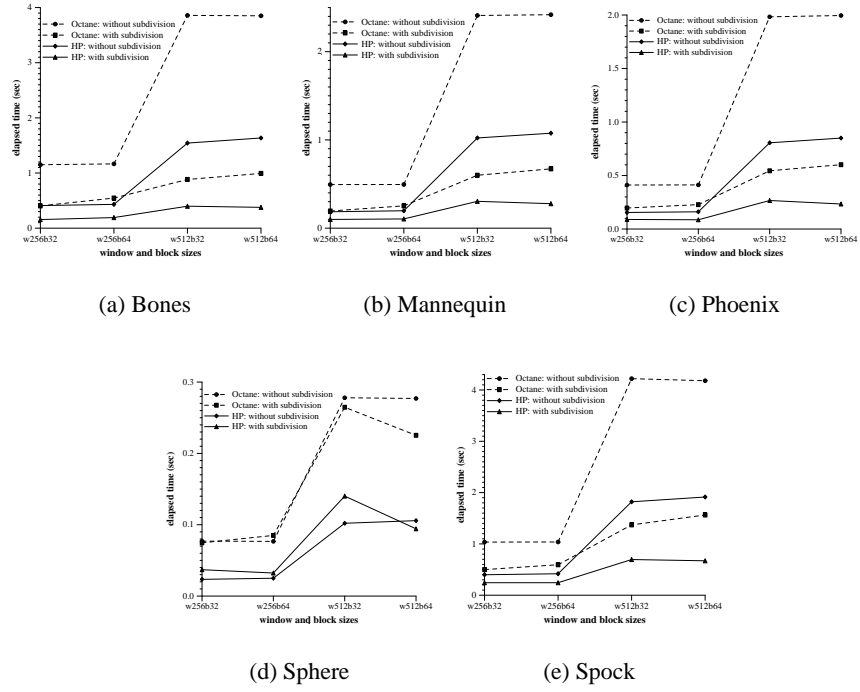
version with the subdivision scheme (see previous section) for better performance. We performed our experiments on two different window sizes:  $256 \times 256$  and  $512 \times 512$ . For the optimized version of our algorithm, we also vary our block sizes. We use  $32 \times 32$  and  $64 \times 64$  for our experiments. There are five data sets in our experiments (Bones, Mannequin, Spock, Phoenix, and Sphere, see Fig. 6 and Fig. 7(a)). We ran our program over a precomputed set of transformations. We collected the data over 30 frames. Table 1 lists some of the characteristics of these data sets.

Figures 2 (a), (b), (c), (d) and (e) shows the total computation time of the two algorithms over the five data sets. Generally speaking, the subdivision scheme reduces the total computation time. Figures 3 (a) and (b) show the speedup of the subdivision-based algorithm. This is because the image layers after the top-layer extraction tend to be smaller and smaller in the frame buffer. With the subdivision scheme, we can read a fraction of the frame buffer as necessary and at the same time, the scanning area gets smaller. However, there are a few models like the *sphere* which are too symmetric for us to observe any performance improvement with our scheme.

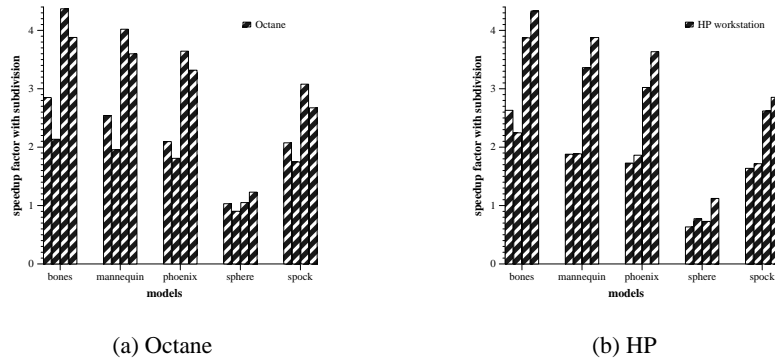
Figures 4 (a), (b), (c) and (d) list the percentage of the time spent on scanning layers and reading buffers for the two algorithms on the two machines. Scanning layers and reading buffers take most of the computation time. While the total percentage of time spent in scanning and reading the buffers is similar on the two architectures, we observe from the Figure 4 that the scanning time dominates in the SGI Octane, while in the HP, reading time is significantly higher. The most important reason for this discrepancy can be attributed to significant difference in the processor speeds. In most cases, the subdivision scheme speeds up the performance, sometime over 4 times.

#### 4.1 Unstructured Grid Volume Rendering

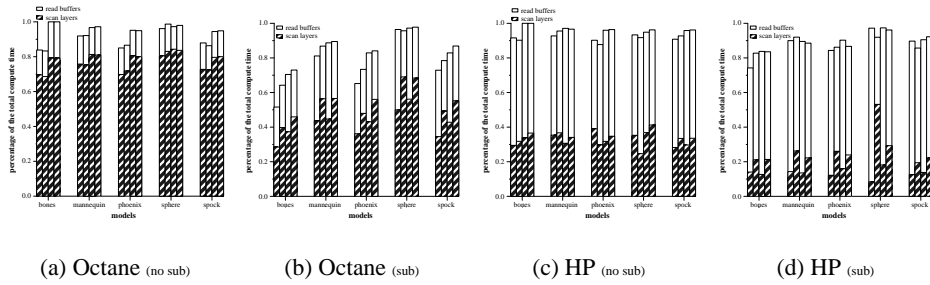
XMPVO [18] and BSP-XMPVO [6] are two volume rendering techniques based on extending MPVO [25] by sorting the boundary cells. The actual sorting techniques proposed in XMPVO and BSP-XMPVO are quite different, and lead to substantially different results. It is quite simple to replace the XMPVO sorting with our new approach. We present the performance of our hardware-assisted visibility ordering algorithm compares to these techniques in Table 2. We also include two previous techniques for exact polyhedral cell sorting, the algorithm of Stein *et al.* [20], the multi-tiled sort of Williams *et al.* [26] in our comparison for completeness. We ran these experiments for three (irregular) datasets. The boundary of the meshes are shown in Figs. 7 (Phoenix, NASA



**Fig. 2.** Compute times for various models. In the x-axis of each figure, we present the different rendering configurations of window and block sizes used.



**Fig. 3.** Speedups with subdivision-based algorithm. The four bars in each group are based on the window size and block size in our experiments. The four bars in each group of the plot, from left to right represent window size and block size (leftmost: window size  $256 \times 256$ , block size  $32 \times 32$ ; middleleft: window size  $256 \times 256$ , block size  $64 \times 64$ ; middleright: window size  $512 \times 512$ , block size  $32 \times 32$ ; rightmost: window size  $512 \times 512$ , block size  $64 \times 64$ ), of each configuration.



**Fig. 4.** Percentage of the overall execution time spent on scanning layers and reading frame buffers of the algorithm with and without the subdivision scheme on Octane and HP.

No. Cells	Stein Sort	Multi-Tiled Sort	XMPVO	BSP-XMPVO	HA-256	HA-512
13,000	14 sec.	7.2 sec.	3.5 sec.	0.37 sec.	0.10 + 0.07	0.33 + 0.07
190,000	2,880 sec.	162 sec.	25 sec.	2.5 sec.	0.09 + 0.70	0.18 + 0.70
240,000	N/A	475 sec.	48 sec.	2.9 sec.	0.14 + 0.90	0.25 + 0.90

**Table 2.** 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.* [26], (3) the XMPVO algorithm of Silva *et al.* [18], (4) the BSP-XMPVO algorithm of Comba *et al.* [6]. (5) and (6) are our results under two different window resolutions. We separate the time sorting the boundary cells took from the time it takes to the MPVO relations, as to highlight the different overheads. The first three timings were performed on an R10000 CPU of an SGI Power Onyx; BSP-XMPVO was timed on a 333MHz PowerPC 604. (5) and (6) were timed on an R12K CPU of an SGI Octane with MXE graphics.

Blunt Fin, and NASA Langley F117). The Phoenix data, shown in Fig. 7(a), has approximately 2,700 boundary faces and its boundary is the most interesting of the three. It has both convex and concave pieces with a complex topology (not homeomorphic to a sphere), and the number of layers our algorithm outputs is high (around 9 or 10 for most view directions). At a resolution of  $256 \times 256$ , we need to render between 11,000 to 14,000 triangles (triangles are rendered multiple times), taking on average 0.10 seconds. For a higher-resolution of  $512 \times 512$ , the number of triangles stay relatively the same but the overall time goes up to approximately 0.33 seconds.

The tetrahedralized NASA Blunt Fin shown in Fig. 7(b) has approximately 13,500 boundary faces. Its boundary is almost convex and has a topology homeomorphic to a sphere. The number of layers we generate is usually between 3 and 5, while rendering between 38,000 and 60,000 triangles. It takes usually less than 0.1 seconds for a resolution of  $256 \times 256$ , and only about double the time for the  $512 \times 512$  resolution despite the fact that the screen resolution actually goes up by a factor of four. Results for the NASA Langley F117 are similar to the Blunt Fin, although the numbers of layers is higher (in the 7-9 range).

The best overall sorting rates we get are a bit over 200,000 tetrahedra per second, considerably better than the ones achieved with the BSP-XMPVO technique. In fact, for these results, the MPVO relations start to dominate and account for about 80% of the overall time. It is not clear we are performing a fair comparison. BSP-XMPVO and XMPVO are truly “exact” techniques, while in our case, we could possibly miss generating ordering relations between cells that might need them. On the other hand, quite possibly the overall visibility-ordering generated changes little, because the inner relations that MPVO generates are highly constraining. Even by classifying a subset of the cells in a correct layer probably is enough to avoid generating any sorting error. We believe this is one of the reasons that the MPVONC heuristic proposed by Williams is so effective. For non-exact techniques, the fastest times are reported in [27], which can sort between 600,000 to 2 million tetrahedra per second, by exploiting temporal coherence.

## 5 Conclusion and Future Work

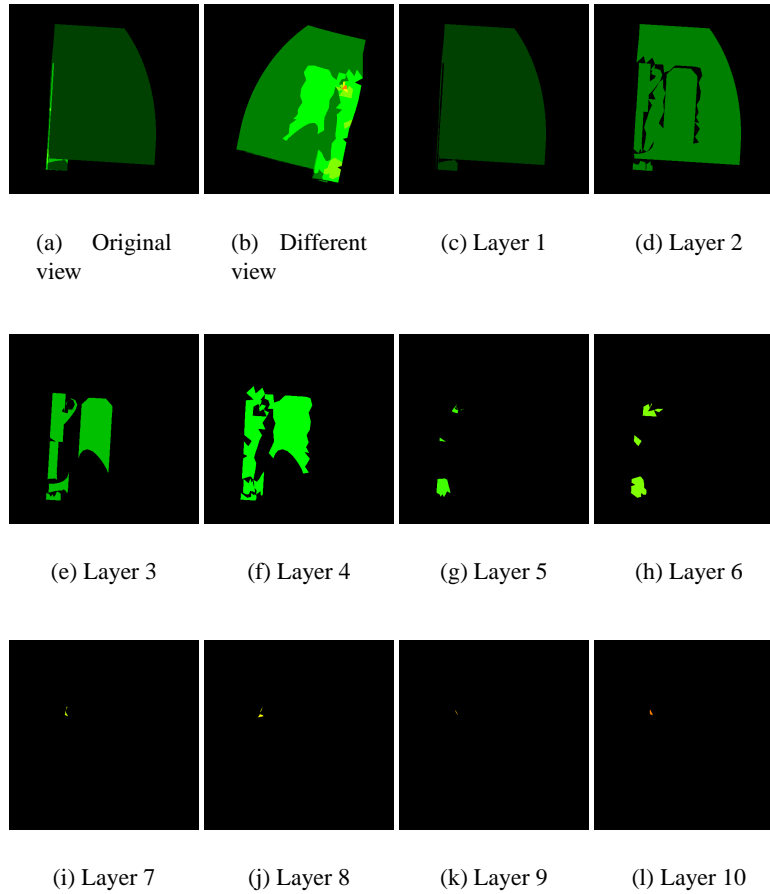
We have presented a hardware-assisted algorithm for visibility-ordering. From a given viewpoint and view direction, we compute a partial ordering of the primitives which can then be rendered using the standard painter’s algorithm. We have used a combination of the hardware *z*-, *stencil* and *color* buffers to achieve this ordering. Our experiments on a variety of models have shown significant speedups in the ordering times compared to existing methods.

The two main costs associated with our implementation are the cost of transferring the buffers to the host’s computer memory, and the time it takes the host CPU to scan them. It is possible to use the histogramming facility available in the ARB\_imaging extension of OpenGL 1.2 to make the graphics hardware perform those computations (we refer the reader to Klosowski and Silva [11], and Westermann et al [24] for details). Unfortunately, those pixel paths are not optimized, and are often slower than our current implementation. If future hardware optimizes that functionality, it would be possible to further improve the performance of our technique.

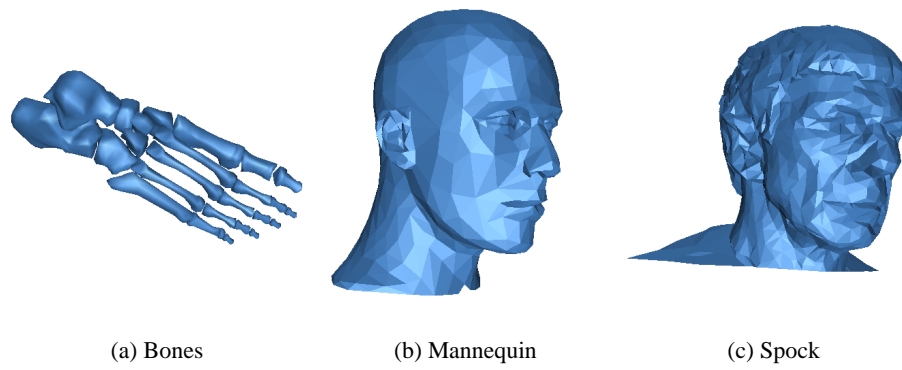
## References

1. B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *1994 Symposium on Volume Visualization*, pages 91–98. October 1994.
2. P. Cignoni and L. De Floriani. Power diagram depth sorting. In *10th Canadian Conference on Computational Geometry*, 1998.
3. 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.
4. P. Cignoni, C. Montani, and R. Scopigno. Tetrahedra based volume visualization. *Mathematical Visualization – Algorithms, Applications, and Numerics*, pages 3–18. Springer Verlag, 1998.
5. J. Comba. *Kinetic Vertical Decomposition Trees*. PhD thesis, Department of Computer Science, Stanford University, 2000.

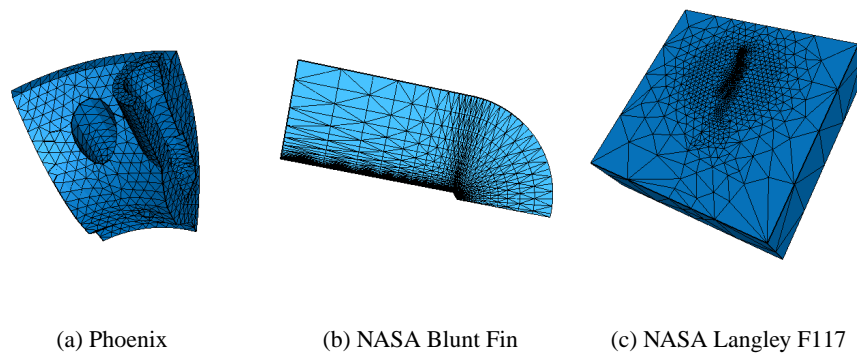
6. J. Comba, J. Klosowski, N. Max, J. Mitchell, C. Silva, and P. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum*, 18(3):369–376, September 1999.
7. M. de Berg, M. Overmars, and O. Schwarzkopf. Computing and verifying depth orders. *sicomp*, 23:437–446, 1994.
8. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.
9. 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.
10. K. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. *Proceedings of SIGGRAPH 99*, pages 277–286, August 1999.
11. J. Klosowski and C. Silva. Efficient Conservative Visibility Culling Using The Prioritized-Layered Projection Algorithm. *Unpublished manuscript*, 2000.
12. P. Lindstrom, D. Koller, W. Ribarsky, L. Hughes, N. Faust, and G. Turner. Real-Time, continuous level of detail rendering of height fields. *SIGGRAPH 96*, pages 109–118, 1996.
13. M. E. Newell, R. G. Newell, and T. L. Sancha. A new approach to the shaded picture problem. In *Proc. ACM Nat. Conf.*, page 443. 1972.
14. H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro real-time ray-casting system. *Proceedings of SIGGRAPH 99*, pages 251–260, August 1999.
15. H. Pfister and A. Kaufman. Cube-4 - A scalable architecture for real-time volume rendering. In *1996 Volume Visualization Symposium*, pages 47–54. October 1996.
16. H. Samet. *Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods*. Addison-Wesley, Reading, MA, 1989.
17. P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24, pages 63–70, November 1990.
18. C. Silva, J. Mitchell, and P. Williams. An interactive time visibility ordering algorithm for polyhedral cell complexes. In *Proc. ACM/IEEE Volume Visualization Symposium '98*, pages 87–94, November 1998.
19. J. Snyder and J. Lengyel. Visibility sorting and compositing without splitting for image layer decomposition. *Proceedings of SIGGRAPH 98*, pages 219–230, July 1998.
20. C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. *1994 Symposium on Volume Visualization*, pages 83–90. October 1994.
21. I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *Journal of the ACM*, March 1974.
22. R. Westermann and T. Ertl. The VSbuffer: Visibility ordering of unstructured volume primitives by polygon drawing. *IEEE Visualization '97*, pages 35–42, November 1997.
23. R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. *Proceedings of SIGGRAPH 98*, pages 169–178, July 1998.
24. R. Westermann, O. Sommer, and T. Ertl. Decoupling Polygon Rendering from Geometry using Rasterization Hardware. *Unpublished manuscript*, 1999.
25. P. L. Williams. Visibility-ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, April 1992.
26. P. Williams, N. Max, and C. Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):37–54, January-March 1998.
27. C. Wittenbrink. Cellfast: Interactive unstructured volume rendering. In *Proc. Late Breaking Hot Topics, IEEE Visualization*, 1999.



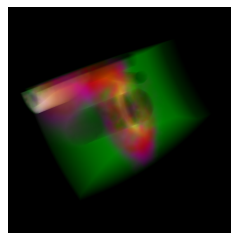
**Fig. 5.** Figures illustrates the layering computed with our algorithm. We color code the triangles according to the layer they belong to. In (a) we show the layering from the view it was computed. In (b), we rotated the object as to show the layering from the other side. Images (c)–(l) show the ten layers computed for this particular view. Note how the 2D footprint of the layers get smaller and smaller.



**Fig. 6.** Three of the five datasets used in our experiments.



**Fig. 7.** Boundary of various volumetric datasets.



**Fig. 8.** Volume rendered image of the Phoenix dataset. These images can be computed at about 5Hz on an SGI Octane.