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 have no obstruction between 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. One application of our technique is to obtain a fast unstructured volume rendering algorithm. In this paper, we present our technique and its implementation in OpenGL. We also discuss its performance and some optimizations on 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 [20]. The main contribution of this paper is a technique for computing a visibility ordering of a set of (acyclic) primitives by using features of the graphics hardware.

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 [15], which is based on the Cube-4 architecture of Pfister and Kaufman [16]. 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 shared the same volumetric data model, that is, each volumetric grid is basically a regularly spaced 3D matrix of voxels.

A technique that is able to leverage existing graphics hardware for volume rendering is the Projected Tetrahedra (PT) algorithm of Shirley and Tuchman [18], which uses the traditional 3D polygon-rendering pipeline. This technique renders a volumetric grid by breaking it into a collection of tetrahedra. Then, each tetrahedron is rendered by *projecting* its faces on the screen. This technique explores the graphics hardware for approximating the volume rendering lighting computations and generates high-quality images. Different from the previous approaches, the Shirley and Tuchman's method is not specific to a regular volumetric grid. In addition, it is quite efficient in terms of the number of triangles it needs to render per primitive. Wittenbrink [28] 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 [21].

In order to apply PT, one needs to compute a visibility-ordering of the cells. Williams' Meshed Polyhedra Visibility Ordering (MPVO) algorithm [26] 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 [19], 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.* [7] 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).

The fundamental computation which XMPVO and BSP-XMPVO are built on is the ability to obtain a visibility-order of the boundary cells. In this paper, we focus on how to find an ordering of the boundary cells using graphics hardware.

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[1] 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 ρ_u denote the ray from v through the point u. A *visibility ordering*, $<_v$, of a set of primitives $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ from a given viewpoint, $v \in \Re^3$, 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 there exists a ray ρ from the viewpoint v such that $\rho \cap p \neq \emptyset$, $\rho \cap p' \neq \emptyset$ and the intersection point of ρ with p is before the intersection point with p' along the ray.

Work on visibility ordering in computer graphics was pioneered by Schumacker *et al.* [22]. 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 [21]. 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 [22], 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, $\Re^3$) (see [8, 9, 17]). Each node ν of a BSP-tree $\mathcal{T}$ corresponds to a convex polyhedral region, $P(\nu) \subset \Re^3$; the root node corresponds to all of $\Re^3$. Each non-leaf 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 $h(\nu)$. Fuchs *et al.* [9] demonstrated that BSP-trees can be used for obtaining a visibility ordering of 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^+.

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.* [7] show how to

[1] *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 [19].

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 [6] 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.* [19]. 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 obtain 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 [28]) 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 [20] 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 VSbuffer technique of Westermann and Ertl [23] 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 pbuffers, 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.

Another related technique is presented by Mammen [14], where he uses a multi-pass rendering technique with a “moving” depth buffer to render transparent objects.

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 [24, 23]) 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 discussion, we assume to obtain 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, a 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 without a layer number assigned 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 [20] for a technique which can be used to handle the cycles.)

We now explain our algorithm in more detail. We assume that 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 the 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 ith layer. During our algorithm, the stencil buffer is sometimes disabled. But whenever it is enabled, it is set to increase the values on the stencil buffer any time a triangle would have been projected into those pixels. In OpenGL, the stencil buffer can 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, from which the original triangles can be identified by reading back the color buffer. We name this process as *reading and scanning* the buffer in the rest of our discussion. Reading buffers refers to performing the `glReadPixels` call, while scanning a buffer refers to the process of traversing the pixel arrays, to obtain 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).

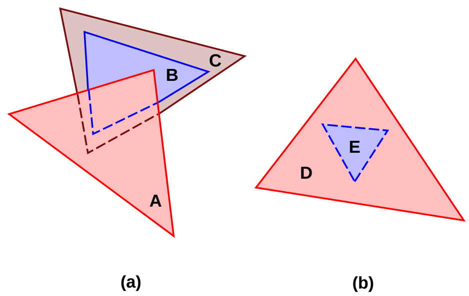


Fig. 1. (a) In this situation, triangle A occludes parts of triangles B and C, while B is completely occluded by C from the opposite direction. During the first scan, pixels (partially) covering B and C are present in the top layer, $\mathcal{F}$. Note that in step 4, we remove triangles from back to 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.

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, 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 claim that the remaining triangles contain a cycle, or they are orthogonal to the view direction.

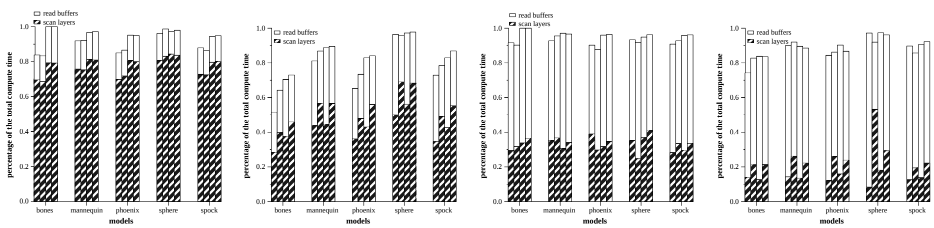
It is straightforward to turn the description of our algorithm above into working C++ code. If we have n triangles in a scene, the worst-case performance of our algorithm is $O(n^2)$, since all the triangles can be behind a single pixel. But this is rarely the case. Assuming the depth complexity of the scene is d, the complexity of the algorithm is much close 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, most 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. 3). Thus, reading and scanning the whole buffers is a waste of time. We propose a simple modification of our algorithm which greatly improves the overall performance. It is based on the 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 advantageous to use a subdivision scheme of dividing the image into blocks, and keeping track of pixel coverage in every block, to avoid unnecessary reading and scanning. In most architectures, the larger the block size, the better the bandwidth in reading back the buffers, although this tends to max out usually somewhere around a 512×512 block. On the other hand, large blocks may not effectively reduce the unnecessary reading and scanning operations. Based on our experiments, a 64×64 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 SGI Octanes and an HP PC. We are only presenting the data collected from the faster Octane and the HP PC. 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 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 version with the subdivision scheme (see Section 3) for better performance. We performed our experiments on two different window sizes: 256×256 and 512×512. For the optimized version of our algorithm, we also varied the block sizes. We used 32×32 and 64×64 for our experiments. There are five data sets in our experiments. 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. Generally speaking, the subdivision scheme reduces the total computation time. 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 2 (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 execution time. While the total percentage of time spent on scanning and read-

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 256×256 and 512×512 over 30 frames.



(a) Octane (no sub) (b) Octane (sub) (c) HP (no sub) (d) HP (sub)

Fig. 2. 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.

ing the buffers is similar between the two architectures, we observe from the Figure 2 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, sometimes over 4 times.

Unstructured Grid Volume Rendering. XMPVO [19] and BSP-XMPVO [7] are two volume rendering techniques based on extending MPVO [26] by sorting the boundary cells. The actual sorting techniques proposed in XMPVO and BSP-XMPVO are quite different, and lead to substantially different results. The XMPVO algorithm works by augmenting the visibility relation between cells by performing "ray shooting queries" between faces of the boundary cells. It is possible to replace the XMPVO sorting with our new approach, which essentially shoots *one ray per pixel*, and thus can lead to *inexact* sorting in some situations. The basic idea is to save the identity of a face that has been projected into a given pixel during the layering extraction. Then, while extracting higher layers later in the processing, add an arrow (ordering relation) to the face that projects in the same pixel, and has a higher layer. There are some choices on the actual accounting for the relations in a given implementation. One way would be to keep a number of relations equal to the number of projected pixels of each face. Again, note that we only need to care about boundary faces in this process, which in general is a very small number of faces compared to the total complexity of a given dataset (see [7] for details). Comparing with the results presented in [19] and accounting for the MPVO relations separately, our experiments indicate this discretized XMPVO is considerably faster than the one presented in [19], that is, about a factor of ten faster. 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 is probably enough to avoid generating any sorting error. We believe this is one of the reasons that the MPVONC heuristic proposed by Williams [26] is so effective. See Cook [5] for an alternative sorting technique also based on a discretization of XMPVO.

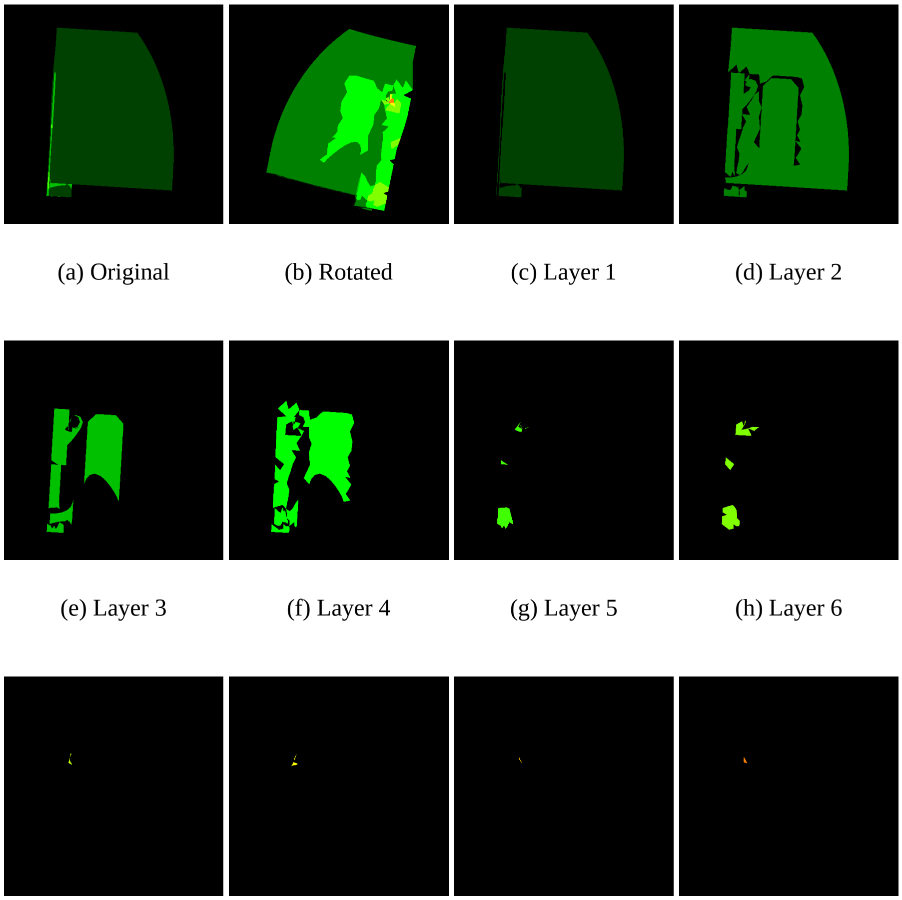
5 Conclusion and Future Work

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 compute this ordering. Our experiments on a variety of models have shown significant speedups in the ordering time compared to existing methods. The two main costs associated with our implementation are the cost of transferring the buffers to the host’s main 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 [25] for details). Unfortunately, those pixel paths are not optimized, and are often slower than our current implementation. If future hardware optimizes this functionality, it would be possible to further improve the performance of our technique.
Acknowledgements: The authors thank Nelson Max and Richard Cook for useful comments and discussions, and the referees for helpful suggestions.

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. R. Cook. Parallelizing and Implementing an Exact Topological Sort For The HIAC Volume Renderer. M.S. thesis, Department of Computer Science, University of California, Davis, 2001. (to appear)

6. J. Comba. *Kinetic Vertical Decomposition Trees*. PhD thesis, Department of Computer Science, Stanford University, 2000.
7. 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.
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. A. Mammen. Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. *IEEE Computer Graphics & Applications*, pages 43–55, July 1989.
15. 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.
16. 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.
17. H. Samet. *Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods.* Addison-Wesley, Reading, MA, 1989.
18. 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.
19. 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.
20. J. Snyder and J. Lengyel. Visibility sorting and compositing without splitting for image layer decomposition. *Proceedings of SIGGRAPH 98*, pages 219–230, July 1998.
21. 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.
22. I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *Journal of the ACM*, March 1974.
23. R. Westermann and T. Ertl. The VSbuffer: Visibility ordering of unstructured volume primitives by polygon drawing. *IEEE Visualization '97*, pages 35–42, November 1997.
24. R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. *Proceedings of SIGGRAPH 98*, pages 169–178, July 1998.
25. R. Westermann, O. Sommer, and T. Ertl. Decoupling Polygon Rendering from Geometry using Rasterization Hardware. *Unpublished manuscript, 1999.*
26. P. L. Williams. Visibility-ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, April 1992.
27. 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.
28. C. Wittenbrink. Cellfast: Interactive unstructured volume rendering. In *Proc. Late Breaking Hot Topics, IEEE Visualization*, 1999.



(a) Original (b) Rotated (c) Layer 1 (d) Layer 2

(e) Layer 3 (f) Layer 4 (g) Layer 5 (h) Layer 6

(i) Layer 7 (j) Layer 8 (k) Layer 9 (l) Layer 10

Fig. 3. 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.