Interactive Visibility Ordering and Transparency Computations among Geometric Primitives in Complex Environments

Naga K. Govindaraju Michael Henson Ming C. Lin Dinesh Manocha
University of North Carolina at Chapel Hill
{naga,henson,lin,dm}@cs.unc.edu
http://gamma.cs.unc.edu/SORT

Abstract

We describe a novel algorithm for visibility ordering among non-overlapping geometric objects in complex and dynamic environments. Our algorithm rearranges the objects in a back-to-front or a front-to-back order from a given viewpoint. We perform comparisons between the primitives by using occlusion queries on the GPUs and exploit frame to frame coherence to reduce the number of occlusion queries. Our visibility ordering algorithm requires no preprocessing and is applicable to all kind of models, including polygon soups and deformable models. We have used our algorithm for order-independent transparency computations in high-depth complexity environments and performing N-body collision culling in dynamic environments. We have implemented our algorithm on a PC with a 3.4 GHz Pentium IV CPU with an NVIDIA GeForce FX 6800 Ultra GPU and applied it to complex environments with tens or hundreds of thousands of polygons. Our algorithm can compute a visibility ordering among the objects and triangles at interactive frame rates.

CR Categories: I.3.1 [Computing Methodologies]: Hardware Architecture—Graphics Processors; I.3.7 [Computing Methodologies]: Three-Dimensional Graphics and Realism—Visible surface algorithms, animation, virtual reality; I.3.5 [Computing Methodologies]: Computational Geometry and Object Modeling—Geometric algorithms;

Keywords: Visibility ordering, transparency computations, sorting, collision detection, graphics hardware

1 Introduction

The problem of computing a visibility ordering among geometric objects is important in many interactive 3D graphics applications. Given a set of disjoint objects in a complex and dynamic environment, our goal is to compute a front-to-back or a back-to-front object-level ordering from a given viewpoint. This problems arises in rendering with transparency [Mammen 1989; Everitt 2001], volume rendering of unstructured grids [Williams 1992; Cook et al. 2004], collision detection in large environments [Govindaraju et al. 2003], special effects generation including motion blur and depth of field generation [Max and Lerner 1985; Przemyslaw 1993], image-based rendering [Snyder and Lengyel 1998], occlusion culling [Cohen-Or et al. 2001], etc. The underlying objects in these applications are represented as polygonal models or height fields or deformable models.

Visibility ordering has been studied in computer graphics and related areas for almost four decades. Different algorithms can be classified into object-space and image-space techniques. The object-space approaches perform a 3D sorting among objects and compute an object-level ordering. However, the resulting algorithms either perform considerable preprocessing (e.g. BSP-trees) or are limited to simple shapes (e.g. convex polytopes). The image-space algorithms rasterize the primitives and compute a visibility ordering among the resulting pixel fragments, as opposed to an object-level ordering. As a result, they are not directly applicable to some applications such as collision detection. Moreover, current implementations on commodity graphics processors perform selection sort at the fragment level and may not be able to exploit temporal coherence in many interactive applications.

Main Results: We describe a novel algorithm that computes an object-level visibility ordering in complex and dynamic environments. We assume that different objects are non-overlapping and a visibility order exists between them. The sorting algorithm proceeds over the list of objects in multiple iterations and computes a sequence of *consecutive quantiles* based on the ordering between the objects. The overlap tests and comparison operations between the objects are performed using occlusion queries on the graphics processing unit (GPU). Our algorithm utilizes the temporal coherence in interactive applications and performs incremental computations to reduce the number of occlusion queries used during each frame. Overall, our algorithm computes an object-level ordering by performing image-space occlusion computations. The accuracy of the occlusion computations is governed by the image-space resolution for rendering applications (e.g. rendering with transparency) and by object-space precision for geometric applications (e.g. collision detection). We have implemented the algorithm on a PC with 3.4 GHz CPU and NVIDIA GeForce FX 6800 GPU. We have tested its performance for rendering with transparency and N-body collision culling on environments with hundreds of thousands of polygons. Our algorithm is able to compute an ordering between the 3D objects at interactive rates. Some of the main benefits of our approach include:

- **Generality:** Our algorithm makes no assumptions about the input objects or their motion and is applicable to all geometric or sampled primitives.
- **Efficiency:** The comparison operations between the primitives are performed by using occlusion queries on the GPUs. As a result, we are able to handle environments with tens or hundreds of thousands of objects and high depth complexity at interactive rates.
- **Exploitation of Coherence:** Our algorithm exploits temporal coherence between successive frames and reduces the number of occlusion queries.

Organization: The rest of the paper is organized as follows. We survey some related work on visibility ordering and related appli-

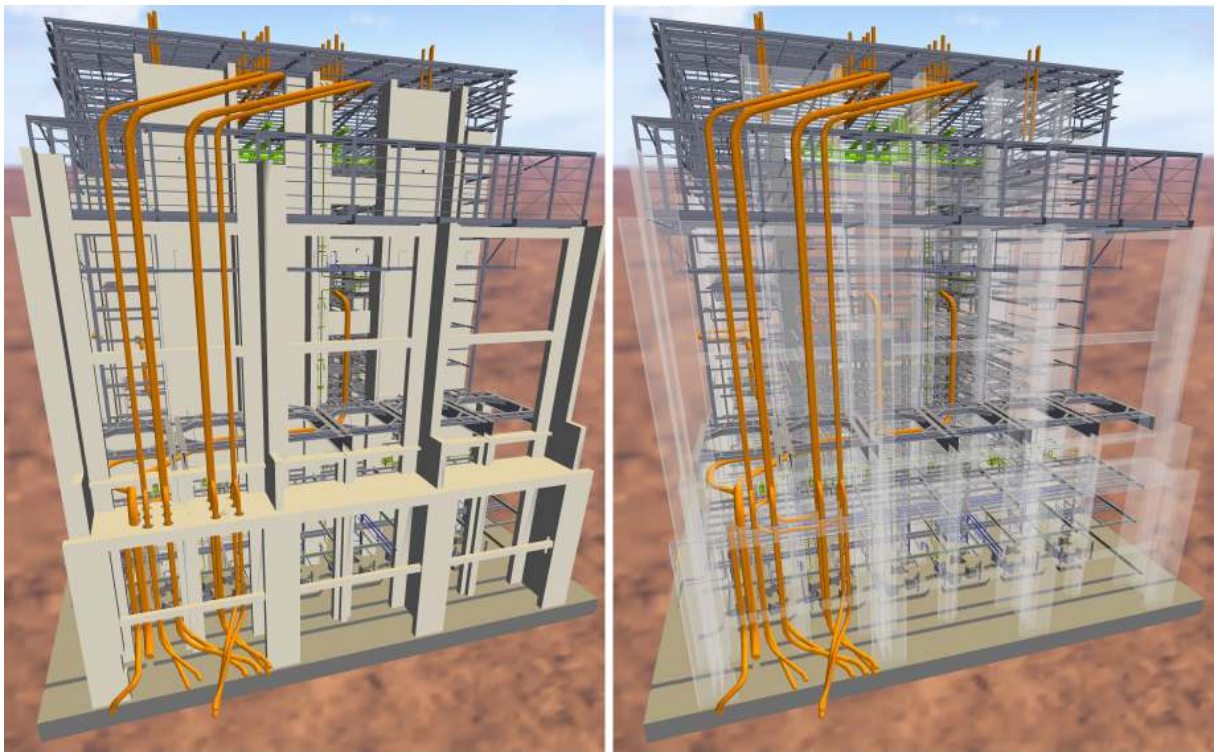

Figure 2: *These images demonstrate the performance of our visibility ordering algorithm on a CAD model with 820K triangles and high depth complexity. The left image shows the original model rendered with opaque objects. The outer walls and structures (represented with 91K triangles) of the powerplant are rendered with transparency in the right image. Our algorithm computes a back-to-front ordering of the transparent primitives at* $7 - 10$ *frames on a* 3.4 *GHz PC with NVIDIA GeForce FX 6800 GPU and renders them with a vertex shader.*

cations in Section 2. Section 3 gives an overview of 1D visibility-based sorting algorithm and we extend it to sorting 3D objects in Section 4. We describe our implementation in Section 5 and highlight its performance on rendering objects with transparency and N-body collision culling. We compare some features of our algorithm with prior techniques in Section 6 and discuss some of the limitations.

2 Related Work

In this section, we give a brief overview of related work in visibility ordering, sorting and their applications. Visibility computation is a classic problem in computer graphics, computational geometry and related areas. Many algorithms have been proposed for hidden surface removal and visible surface computation [Sutherland et al. 1973], before the depth-buffer hardware became widely available.

At a broad level, algorithms for visibility ordering can be classified into object-space or image-space algorithms. The object-space algorithms compute an ordering of the primitives in 3D space. Berg et al. [de Berg et al. 1994] present an $O(n^{(\frac{4}{3}+\epsilon)})$ object-space algorithm for any fixed $\epsilon > 0$ to compute a visibility order of n primitives or to determine the existence of a cycle. However, we are not aware of any practical implementation of the algorithm. Some of the earliest object space algorithms were proposed by Schumacker [1969] and Newell et al. [1972]. Later Fuchs et al. [1980] developed the binary space partitioning (BSP) tree that represented a hierarchical convex decomposition of a given space. BSP-trees can be used for computing a visibility ordering of a set of objects. Most of these algorithms work well on static environments. A few algorithms have been proposed for visibility computations in dynamic environments. Torres [1990] used dynamic BSP trees to compute visibility ordering of polygons in the scenes. Sudarsky and Gotsman [1996] described an output-sensitive algorithm which minimizes the time required to update a hierarchy for dynamic objects for visibility culling. Snyder and Lengyel [1998] presented an efficient algorithm for visibility sorting of geometric primitives in dynamic scenes. Their main goal is to compute a sequence of image layers which can be composed to produce the final image. The algorithm works well on environments composed of a few hundred convex polytopes (or union of convex polytopes). A major limitation of object-space algorithms is that they can't be used for interactive visibility ordering in complex, dynamic scenes.

Image-space algorithms are used for visibility ordering of pixel fragments as opposed to object fragments. Check out a recent survey [Cohen-Or et al. 2001]. Image-space algorithms rasterize the primitives and perform some kind of of per-pixel sorting. Carpenter [1984] proposed the A-buffer algorithm that saves all the fragments and their depth values in per-pixel linked lists and uses them for sorting. Wittenbrink [2001], Jouppe and Chang [1999] and Aila et al. [2003] have proposed extensions to the A-buffer algorithm and used the resulting techniques for transparency. Other class of image-space algorithms for interactive order-independent transparency are based on *depth peeling*, which is a fragment-level depth sorting technique described by Mammen [1989] using *virtual pixel maps* and by Diefenbach [1996] using a *dual depth buffer*. Depth peeling can be implemented using shadow mapping hardware on commodity GPUs [Everitt 2001]. All these image-space algorithms are applicable to general models and dynamic environments. However, they can not be directly used to compute an object-level visibility ordering of primitives.

Many specialized visibility ordering algorithms have been proposed for volume rendering of unstructured grids and polyhedral cell complexes [Williams 1992; Max 1993; Cignoni et al. 1998].

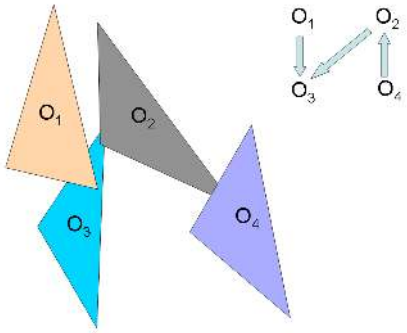

Figure 3: *Occlusion graph and 3-D visibility ordering: This figure illustrates a simple scene with four mutually overlapping primitives and its occlusion graph. Each node in the graph corresponds to a 3D primitive. A directed edge exists between the nodes of two primitives P_1 and P_2 if P_1 occludes P_2 from a given viewpoint. 3D visibility ordering is equivalent to performing topological sorting on the occlusion graph.*

These algorithms sort the cells along a ray direction and compute the order of the resulting ray segments. Some recent volume rendering algorithms use GPUs to generate the fragments based on their visibility order [Krishnan et al. 2001; Cook et al. 2004; Callahan et al. 2005].

The algorithms for coarse-grained or N-body collision detection use sorting techniques to prune away pairs of primitives that are not in close proximity. The sweep-and-prune incremental algorithm bounds each object with an AABB (axis-aligned bounding box) and checks for overlap between them by sorting their projections along the three axes [Cohen et al. 1995]. Govindaraju et al. [Govindaraju et al. 2003] have presented a two pass collision culling algorithm that renders the objects in two passes and checks for collision by performing occlusion queries. These algorithms only prune the object pairs that are not close to each other and do not compute a visibility ordering between them.

Sorting is a well studied problem in itself and extensive surveys are given in [Knuth 1973; Estivill-Castro and Wood 1992]. In terms of using GPUs for sorting, Purcell et al. [2003] described an implementation of bitonic merge sort on the GPUs. The algorithm is implemented as a fragment program and each stage of the sorting algorithm is performed as one rendering pass. Recently Govindaraju et al. [2004c] have used blending and texture mapping functionalities of GPUs to implement sorting network algorithms efficiently.

3 Visibility Ordering

In this section, we formulate the problem of visibility ordering and present an algorithm for sorting 1D elements that maps well to the GPUs. Our sorting algorithm is based on Vis-sort [Govindaraju et al. 2004b] that can sort 1D and 3D elements. We show that Vis-sort can exploit the computational capabilities of the GPUs and can compute an ordering between 1D objects. We extend it to compute an ordering between 3D objects in Section 4. Given a collection of n acyclic 3D primitives, our sorting algorithm rearranges the geometric primitives either in a front-to-back or a back-to-front order. In order to perform visibility ordering, we define a pairwise occlusion relation between two 3D primitives P_1 and P_2 based on whether P_1 occludes P_2 or not. More specifically, $P_1 \preceq P_2$ if for every eye-ray intersecting P_1, the eye-ray intersects P_1 before P_2. Given n acyclic 3D geometric primitives, P_1, P_2, ..., P_n, we assume that a visibility ordering can be defined between the primitives and there are no cycles.

The set of occlusion relations between the different pairs of objects form a directed graph between the geometric primitives (see Fig. 3). The resulting directed graph is also defined as an occlusion graph [Schumacker et al. 1969; Snyder and Lengyel 1998] and a directed edge exists in the graph from P_1 to P_2 if $P_1 \preceq P_2$. Computing a 3D visibility ordering is equivalent to performing topological sort on the occlusion graph. Our algorithm computes such an ordering by performing the comparisons using GPUs (graphics processing units). Furthermore, our algorithm exploits temporal coherence between successive instances. The rest of the section is organized in the following manner. We first introduce the terminology used in the rest of the paper. Next we highlight some issues in sorting object-level primitives on GPUs and present the 1D sorting algorithm.

3.1 Terminology

In this section, we introduce some of the terminology and notation used in the rest of the paper. We use lower case letters to represent the elements of sequences and upper case letters to represent the sequences and 3-D primitives. Moreover, we use bold face symbols to represent a list of elements, like $\mathbf{I}$ or $\mathbf{O}$. A sorting algorithm operates on a list of data values $\mathbf{I}$, and rearranges the data values in a monotonically increasing order or a monotonically decreasing order. A sequence $\mathbf{O}$ is defined as monotonically increasing if $x_i \leq x_j$ for $i < j$, $x_i, x_j \in \mathbf{O}$. A monotonically decreasing sequence is defined in a similar fashion. Without loss of generality, we assume that our algorithm orders the data values in a monotonically increasing order and computes an ordered sequence $\mathbf{S}$ as the output. Each element in the ordered sequence $\mathbf{S}$ is associated with a rank which indicates its position in $\mathbf{S}$. We use the following definitions in the rest of the paper.

Quantile: An element with rank k in $\mathbf{S}$ is defined as the k-th quantile or the k-th largest number.

Nearly-Sorted Sequence: A nearly-sorted sequence is defined by measuring the disorder in the sequence. We use Knuth's measure of disorder [Knuth 1973]. Given an input sequence $\mathbf{I}$, the measure of disorder is defined as the minimal number of elements that need to removed for the rest of the sequence to remain sorted. It can be trivially proved that if Y is a set of such disordered elements, then $I - Y$ is a longest ascending sequence. A sequence is nearly sorted if $sizeof(Y) \ll n$.

3.2 Object-Level Sorting using GPUs

One of the major goals of our work is to handle general and dynamic environments at interactive rates, without performing any precomputation. In particular, we would like to use the high computational performance of GPUs for topological sorting and to compute an object-level ordering. There are two main benefits of using GPUs for sorting:

- **Efficient comparison operation:** The performance of any sorting algorithm depends on the cost of the comparison operation between the underlying elements. On a GPU, a comparison operation can be performed between a pair of geometric primitives by rasterizing the primitives using occlusion queries. GPUs are optimized for rasterization and their rasterization performance has been increasing at a rate faster than Moore's Law. As a result, reducing the comparison operation to rasterization can result in significant performance gains.

- **Number of comparison operations:** The performance of a sorting algorithm depends upon the number of comparison operations performed to order the input elements. GPUs are optimized to perform spatial sorting on projections of geometric primitives onto the image space. Therefore, a geometric primitive is compared only against its overlapping primitives in the screen-space. Moreover, GPUs compute an image-space fusion of rendered primitives i.e., the representation

generated in the Z-buffer by rendering a group of primitives is the same as the one obtained by rendering each primitive separately. Therefore, a new primitive can be compared against a collective group of spatially coherent primitives that have been rasterized into the Z-buffer. Effectively, this capability can be used to reduce the number of pairwise comparison operations on the primitives.

Issues: GPUs offer many advantages in terms of performing 3D visibility ordering among primitives. However, there are many issues in using them directly for sorting. Although GPUs are well-suited for performing visibility sorting in image-space, it is difficult to map current sorting algorithms onto the GPUs. The Z-buffer representation used for performing comparison operations stores only one value at each pixel which represents either the minimum or the maximum depth value of primitives that project onto the pixel. Furthermore, once *a portion* of the Z-buffer is updated, it may not be possible to access the prior stored values. Therefore, the class of sorting algorithms that can map well to the GPUs are the ones that perform comparisons between an input value against the *current* minimum or the *current* maximum of a subset of input primitives. An example of such an algorithm is selection sort [Knuth 1973], but that has an asymptotically worst-case complexity ($O(n^2)$) for all input sequences. Some of the most popular algorithms like insertion sort or quick sort do not map well to the GPUs in terms of performing comparison operations.

3.3 1D Sorting Algorithm

We describe a sorting algorithm that maps well onto the GPUs and exhibits linear-time performance in environments with high coherence. We first present our algorithm for 1D primitives.

Our sorting algorithm proceeds in multiple iterations. Initially the input sequence is an unsorted sequence of 1D elements and the output sequence is an empty set. During each iteration, our algorithm operates on a list of unsorted data values and computes a sequence of *consecutive* quantiles beginning with the *minimum* of the unsorted list. We append these sorted values to the ordered sequence, and iterate on the remaining data values. Thus, the size of the unsorted list decreases during each iteration, and upon termination, the output sequence is the sorted list. The set of operations performed by our algorithm map well to the GPUs.

The consecutive quantiles in each iteration are determined by computing a monotonically increasing sequence in the first pass. In the second pass, the first few values in the monotonically increasing sequence are classified as consecutive quantiles. The following notation is used in the rest of the paper.

- **I** = input list for the sorting algorithm
- **S** = current sorted sequence
- **M** = monotonically increasing sequence

The pseudo-code to describe our sorting algorithm is given in Fig. 3.1.

In a companion paper [Govindaraju et al. 2004b], we formally prove that the algorithm sorts any input sequence. We now show a simple example of our sorting algorithm operating on a list of 4 values $I = \{1, 3, 2, 4\}$. We initialize the monotonically increasing sequence **M** and the sorted sequence **S** to an empty sequence. In the first iteration, we perform a scan of **I** from the last element to the first element in the first pass. Each element is compared against the minimum and **M** = {1,2,4} is computed. In the second pass of the first iteration, we determine that the elements 1 and 2 are sorted and are appended to the sorted list $S = \{1, 2\}$. Similarly, in the second iteration, the remaining data values are sorted. The series of sorting operations are indicated in Table 1.

1D HardwareSort

1 **M** ={}, **I**= *Unsorted input,* **S** ={}
2 *while(***I** *is not empty)*
3 *do*
First Pass:
4 *min*=∞
5 *for each element* $x_i \in \mathbf{I}$, $i = sizeof(\mathbf{I}), \ldots, 1$
6 *if*($x_i \notin \mathbf{M}$ *and* $x_i \leq min$ *)*
7 *append it to the beginning of* **M**
8 *if* $x_i \leq min$, $min = x_i$
9 *end for*
Second Pass:
10 *min*=∞
11 **T** = **I**
12 *for each element* $x_i \in \mathbf{I}$, $i = 1, ..., sizeof(\mathbf{I})$
13 *if*($x_i \in \mathbf{M}$ *and* $x_i \leq min$)
14 *remove* x_i *from* **M** *and* **T**, *and append it to the end of* **S**
15 *if*($x_i \in \mathbf{T}$)
16 *if* $x_i \leq min$, $min = x_i$
17 *end for*
18 **I** = **T**
19 *end do*
20 *return* **S**

ALGORITHM 3.1: *Pseudo code for our novel 1D Sorting Algorithm:Given an input sequence I, our algorithm outputs a sorted output* **S**. *During each iteration, we perform a first pass on the elements in* **I** *in the reverse order and compute a monotonically increasing sequence* **M**. *In the second pass of each iteration, we scan the elements of* **I** *in order, and determine the elements in* **M** *that can be sorted (lines 12-13). These elements are appended to* **S**, *and at the end of the second pass, removed from* **I** *(lines 14 and 18). During each iteration, at least one element is guaranteed to be removed from* **I** *and placed in* **S**. *The sorting algorithm terminates when* **I** *is empty.*

3.4 Analysis

Our sorting algorithm has a best-case run-time complexity of $O(n)$ and a worst-case run-time complexity of $O(n^2)$. In particular, our algorithm has some properties that are useful for many interactive applications. These are efficient handling of nearly sorted lists, and mapping well to the GPUs.

3.4.1 Nearly-sorted sequences and Coherence

Nearly-sorted sequences of primitives occur often in computer graphics applications due to the coherent motion of the viewer or the coherent motion of the objects. The depth values as well as the relative order of the objects do not vary much due to the spatial and temporal coherence in many interactive applications. If the primitives are ordered in one frame, the sequence remains nearly ordered in the successive frame. Therefore, if the input is nearly ordered, our algorithm takes advantage of the low level of disorder in the input sequence and achieves a linear-time performance [Govindaraju et al. 2004b].

We provide an informal justification that our algorithm sorts nearly-sorted sequences in linear time. Given an input sequence **I**, let **Y** be the subset of disordered elements. During each iteration, we can show that our algorithm sorts all the elements in **I** that are less than or equal to the min(**Y**) including $min(\mathbf{I})$, where $min(\mathbf{X})$ denotes the minimum of a sequence **X**. Based on this property, it is easy to see that at most two successive iterations are required to sort all the elements in **I** that are less than min(**Y**). In one iteration, all the elements in **I** that are less than min(**Y**) are sorted, and if $min(\mathbf{I}) \neq min(\mathbf{Y})$, a second iteration is required to sort all the elements in **I** equal to $min(\mathbf{Y})$. Our algorithm removes all the sorted elements from **I**. Thus the size of **I** as well as **Y** decreases by at least 1 during two successive iterations, till **Y** becomes empty. In the worst case, our algorithm needs $(2 * sizeof(Y))$ iterations to sort all the elements in $I \leq max(Y)$, where $max(Y)$ is the max-

Iteration 1 **First Pass** Current Element Scanned in **I** ={1,3,2,4} $min = \infty, M = \{\}$	4 min=4, **M**={4}	2 min=2, **M**={2,4}	3 min=2, **M**={2,4}	1 min=1, **M**={1,2,4}
Second Pass Current Element Scanned in **I**={1,3,2,4} $min = \infty, \mathbf{M} = \{1, 2, 4\}$ **S**={}, **T** ={1,3,2,4}	1 $min = \infty$, **M**={2,4} **T** ={3,2,4}, **S**={1}	3 min=3, **M**={2,4} **T**={3,2,4}, **S**={1}	2 min=3, **M**={4} **T** ={3,4}, **S**={1,2}	4 min=3, **M**={4} **T** ={3,4}, **S** = {1,2}
Iteration 2 **First Pass** Current Element Scanned in **I**={3,4} $min = \infty, \mathbf{M} = \{4\}$	4 $min = 4, \mathbf{M} = \{4\}$	3 min=3, **M**={3,4}		
Second Pass Current Element Scanned in **I**={3,4} $min = \infty, \mathbf{M} = \{3, 4\}$ **T**={3,4}, **S**={1,2}	3 $min = \infty, \mathbf{M} = \{4\}$ **T**={4}, **S** ={1,2,3}	4 $min = \infty, \mathbf{M} = \{\}$ **T** ={} **S** ={1,2,3,4}		

Table 1: *In this table, we illustrate our 1D hardware sorting algorithm on an input sequence* $\mathbf{I} = \{1, 3, 2, 4\}$. *Initially, the monotonically increasing sequence M and the sorted sequence* **S** *are set to* $\{\}$. *Our algorithm proceeds in multiple iterations and each iteration performs two passes on* **I**. *At the beginning of each pass, the current minimum* **min** *is initialized to* ∞. *In the first pass, we scan the elements in* **I** *in the reverse order* $\{4, 2, 3, 1\}$ *and compute* **M**. *During the scan, we compare the elements* **not in M** *against the current minimum and only those less than or equal to the current minimum are added to the beginning of* **M**. *The minimum is updated accordingly, and at the end of the pass, a monotonically increasing sequence* $\mathbf{M} = \{1, 2, 4\}$ *is computed. In the second pass, the elements in* **I** *are copied into a temporary buffer* **T**, *and scanned in order* $\{1, 3, 2, 4\}$. *While scanning, the elements in* **M** *are compared against the current minimum, and if less than or equal to the minimum, are removed from* **M** *and* **T**, *and placed in* **S**. *In this pass, the current minimum is updated for only the elements in* **T** *and not in* **M**. *At the end of the second pass,* **I** *is set to* **T**. *In this example,* $\mathbf{I} = \{3, 4\}$, $\mathbf{M} = \{4\}$, *and* $\mathbf{S} = \{1, 2\}$ *at the end of the first iteration. In the second iteration, the remaining values are sorted as shown in the table.*

imum of $\mathbf{Y}$. At this stage, all the elements in $\mathbf{Y}$ are sorted in their proper place. As there are no inversions in the rest of the elements, the rest of the elements in $\mathbf{I}$ remain sorted and our algorithm orders these elements in a single final iteration. As a result, it requires at most $(2 * sizeof(\mathbf{Y}) + 1)$ iterations to terminate. If the number of disordered elements is $k \ll n$, then the run-time complexity of our algorithm has an *upper* bound of $(4k + 2)n$ which is linear in the number of elements in the input sequence $\mathbf{I}$.

3.4.2 Mapping to GPUs

Our algorithm maps well to the commodity graphics hardware, as the comparisons of data elements are performed against the *current minimum* of a subset of data values. The minimum value is stored in the depth buffer, and the comparison operation is performed using the depth test functionality of the GPUs. Unlike prior implementations of sorting algorithms on the GPU, our algorithm exhibits linear-time performance on environments with high coherence, thus enabling us to handle high depth complexity scenes.

4 Object-Level 3D Visibility Ordering

In this section, we show how the 3D version of Vis-sort [Govindaraju et al. 2004b] maps well to the GPUs and can be used to compute a visibility ordering between 3D objects. We perform comparisons between 3D objects using occlusion queries.

3D sorting is more intricate as compared to 1D sorting. In 1D sorting all the primitives are overlapping when viewed along the 1D dimension and ordering is performed on these overlapping primitives. On the other hand, given two non-overlapping 3D objects, we may need to perform ordering between such objects to compute a 3D visibility sort. For example, in Fig. 3, primitive O_4 is overlapping with primitive O_2 but not with O_3. In terms of 3D visibility ordering, O_4 must come before O_3 in the front-to-back ordering.

Object-level 3D visibility sorting can be performed by constructing the occlusion graph, and applying a depth-first search (DFS) algorithm on the occlusion graph to compute the finishing time of each primitive. A 1D sort of the finish times computes the topological sort of the occlusion graph. This can be proved easily based on the following property of the finish times computed by DFS: *If a directed path exists from P_1 to P_2, then the finish time of P_1 is greater than the finish time of P_2.*

We do not compute the occlusion graph explicitly but rather use a variation of the 1D sorting algorithm presented in Section 3 along with the property listed above. In particular, we incorporate two overlap constraints between geometric primitives to perform topological sort on the occlusion graph implicitly. The constraints are:

- **Constraint 1:** If two primitives P_1 and P_2 overlap in screen-space, $P_1 \preceq P_2$ if and only if P_1 is fully visible with respect to P_2. That is, for every eye-ray intersecting P_1, the eye-ray intersects P_1 on or before P_2.

- **Constraint 2:** If two primitives P_1 and P_2 overlap in screen-space, and $P_1 \preceq P_2$, then P_1 occurs before P_2 in the output list of the visibility ordering algorithm.

Constraint 1 ensures that a comparison operation is performed between pairs of overlapping primitives in the scene, thus constructing the occlusion graph implicitly. A comparison operation between any two primitives can be implemented on the GPU using an occlusion query. In order to compare P_1 against P_2, we render P_2 into the depth buffer. If P_1 is fully visible with respect to P_2, then $P_1 \preceq P_2$. We check for this condition by reversing the depth test to $GL\_GREATER$, disable the depth writes and render P_1 using an occlusion query. If the pixel pass count returned by the occlusion query is zero, then P_1 is fully visible with respect to P_2 and we place it accordingly in the output list.

Constraint 2 classifies a primitive P as sorted if and only if all the primitives that occlude P are sorted. Intuitively, constraint 2 ensures that the finish times of the geometric primitives are properly ordered. The detailed description of the object-level 3D sorting algorithm is given in [Govindaraju et al. 2004b]. Note that if a cycle exists in the scene, our algorithm can detect the cycle.

Multi-Stage Ordering: We further improve the performance of the algorithm by using a multi-stage ordering approach. We assume that the scene consists of multiple objects and each object is composed of spatially coherent triangles. In the first stage, we compute the visibility at an object level. After the first stage, several of these objects are ordered with respect to each other and a local visibility sort is sufficient to order the triangles within each of these objects.

It is possible that a few objects in the scene may result in cycles and the algorithm is not able to order all the objects during the first stage. Our algorithm detects these cycles and resolves the visibility order by applying our sorting algorithm on the triangles in these objects. Multi-stage sorting is also useful in scenarios with low coherence and lets us avoid the worst case scenarios with $O(n^2)$ complexity.

5 Implementation and Applications

In this section, we describe the implementation of our algorithm and highlight its application to order-independent transparency and collision culling in large environments.

5.1 Implementation

We have implemented our algorithm on a PC with a 3.4 GHz Pentium IV CPU running Windows XP operating system. We have tested the performance of our algorithm using an NVIDIA GeForce FX 6800 Ultra GPU. The occlusion queries are performed on an offscreen buffer using the OpenGL extension GL_NV_occlusion_query. The color writes are disabled to reduce the internal memory bandwidth within the video card to improve the runtime performance. We increase the rendering throughput by copying the dynamic geometric primitives into a vertex array stored in the video memory. We have reduced the stalls due to occlusion queries by batching several queries together. We have measured the performance of our algorithm using a screen resolution of 1600×1200 and with $4X$ anti-aliasing mode.

5.2 Order-Independent Transparency

We have used our algorithm to generate transparency effects in complex CAD model and dynamic environments. During each frame, our algorithm computes a back-to-front order of the primitives in the scene. We exploit frame-to-frame coherence by using the sorted order of the objects in the previous frame as an input sequence of the primitives for the current frame. In most interactive applications, our algorithm operates on nearly-sorted lists and performs almost a linear number of comparisons. We compute the final output image by first rendering the opaque portions of the scene. We then generate a visually accurate transparent effect by blending the transparent primitives in a strict back-to-front order.

In order to improve the quality of transparency, we have applied membrane shading on the transparent objects. A simple vertex shader is used while rasterizing the transparent objects.

5.3 N-Body Collision Culling

We have applied our visibility ordering algorithm to perform fast collision culling in dynamic environments. Given a large environment with multiple moving objects, our goal is cull away all pair of objects that are not in close proximity. Ultimately the exact collision detection test is applied to those pairs of objects that are almost colliding. The culling algorithm is used to avoid performing $O(n^2)$ exact collision detection tests.

Given a list of 3D objects, we modify our 3D sorting algorithm, so that it returns two lists. They are:

- **S:** It is a sorted list of all non-overlapping and acyclic 3D objects.
- **C:** It is a list of all the objects that are either overlapping or form a cycle.

All the elements in $\mathbf{S}$ are non-overlapping and can be culled away in terms of exact collision tests. On the other hand, the objects in $\mathbf{C}$ are potentially colliding.

Given an environment with multiple static and dynamic objects, we compute a visibility ordering from different view directions. Initially, we choose an axis or random direction and compute the two lists, $\mathbf{S}$ and $\mathbf{C}$. We choose a different viewing direction and apply the algorithm on the objects contained in $\mathbf{C}$. We repeat the procedure until the list of potentially colliding objects does not decrease any further. Eventually, we use exact collision detection tests on the set of objects in the potentially colliding set.

We also use temporal coherence to improve the performance of our algorithm and maintain the sorted order of the objects along the chosen view directions in each frame. These sorted lists are used in the next frame to quickly prune non-colliding objects along the view direction.

The occlusion queries are performed at the image-space resolution on the GPUs. This can lead to sampling and precision errors. In order to overcome these errors, we use orthographic projections along the viewing directions for visibility ordering and use conservative fattened representations of objects. The fattened representations are computed as a union of the bounding representations of the triangles of the object. The bounding representation of each triangle tightly encloses the Minkowski sum of the triangle and a pixel-sized sphere [Govindaraju et al. 2004a].

In practice, our culling algorithm based on visibility ordering is less conservative than CULLIDE [Govindaraju et al. 2003] for the same number of view-directions. As a result, our algorithm results in fewer false positives and reduces the number of exact collision tests between the primitives.

5.4 Performance

We have applied our algorithm on two complex benchmarks:

1. **CAD Model:** We have used a portion of the powerplant model composed of 91 K transparent polygons and 732 K opaque triangles (see Fig. 1). This scene exhibits a depth complexity of $5 - 12$ in different view configurations and we have observed an average frame rate of $7 - 10$ frames per second as the user moves around the model. It includes the time to compute an ordering between the primitives and rendering the primitives.

2. **Dynamic Scene:** We applied our algorithm on a dynamic scene composed of deforming bunnies (see Fig. 4. The overall geometric complexity is about 25 K triangles and all of them are rendered with transparency. In this environment, 18 deforming bunnies are moving randomly in a cube and colliding with each other. The depth complexity in the scene varies in the range $7 - 10$ from most view directions. We have tested its performance in two scenarios: a stationary viewer in the dynamic environment, and a moving user in the dynamic environment. We compute an ordering between the triangular primitives and render the scene with transparency. In both the scenarios, we have observed a frame rate of $8 - 10$ frames per second. In addition to transparency computations, we applied our algorithm to perform N-body collision culling on this dynamic environment. On an average, we were able to compute all the collisions in 40 ms.

5.5 Nearly Sorted Lists and Coherence

The underlying comparison operations in our ordering algorithms are performed using occlusion queries. The performance of the overall algorithm is dominated by the number of occlusion queries and our goal is to minimize the number of queries.

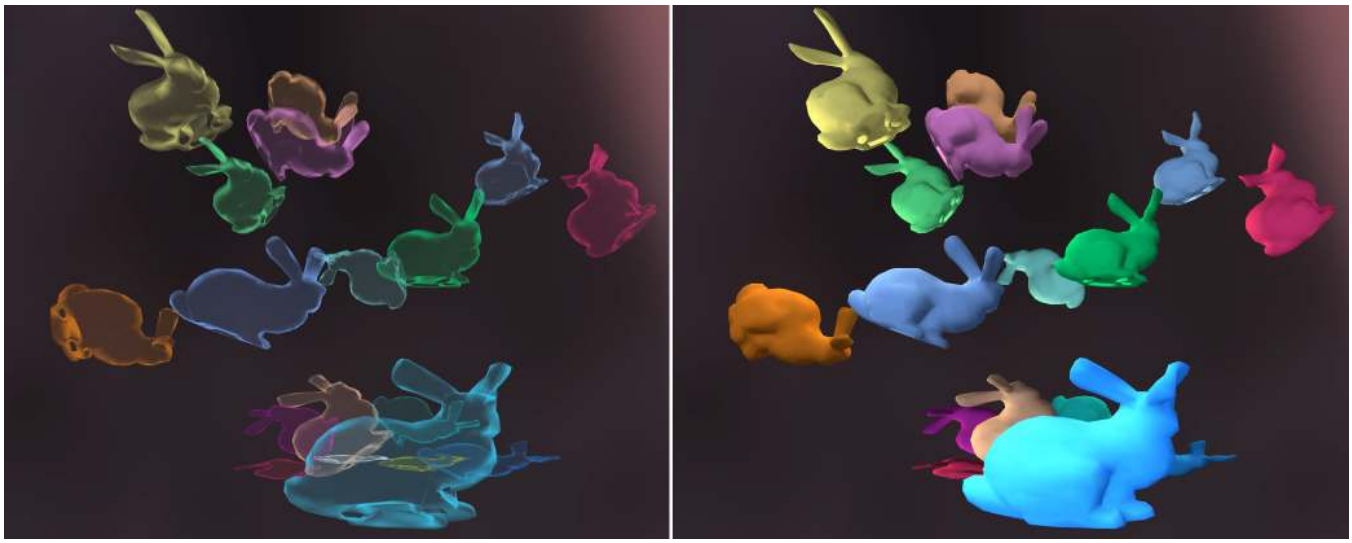

Figure 4: *Dynamic Scene: This scene is composed of* 18 *deforming bunnies moving in a room. The scene consists of* $25K$ *triangles and has a high depth complexity of* $8 - 10$ *in many view directions. One such view is shown in the left image where the scene is rendered with transparency using our visibility ordering algorithm (at* 10 *frames a second). In the right image, the same scene rendered with opaque objects is shown. In this dynamic environment, we are able to perform interactive collision computations at* 25 *frames per second.*

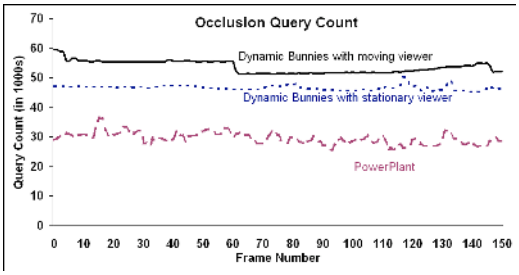

Figure 5: *Occlusion query count: This graph highlights the near-linear time performance of our algorithm on a sample path in the Powerplant model, and the dynamic scene with a stationary viewer and a moving viewer. Our benchmark scenarios exhibit high frame-to-frame coherence and our sorting algorithm exhibits an almost linear-time performance in such benchmarks.*

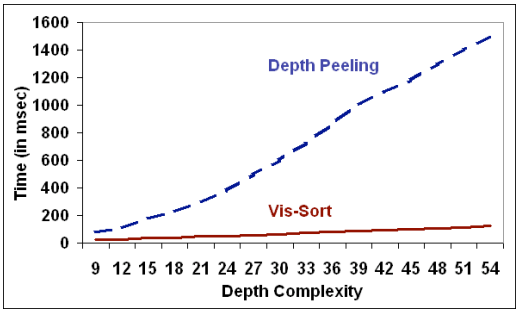

Figure 6: *Comparison with Depth Peeling: This graph compares the performance of our algorithm (Vis-sort) in a dynamic scene with depth peeling algorithm as a function of the scene depth complexity. The timings were gathered at a resolution of* 1600×1200. *Vis-sort does not have the overhead of texture copy operations.*

In Fig. 5, we highlight the number of queries performed in different benchmarks. These scenarios exhibit high frame-to-frame coherence and we observe almost linear time performance in terms of number of queries.

6 Comparison and Limitations

In this section, we compare some of the features of our approach with prior approaches and highlight some of its limitations. Several algorithms have been proposed to perform visibility ordering between geometric primitives for volume rendering and transparency computations. We compare the features of our algorithm with other algorithms.

Object-space algorithms: These algorithms perform ordering of primitives in object-space. Williams [Williams 1992] proposed a visibility ordering algorithm for arbitrary acyclic polyhedral models. However, the algorithm is limited to connected convex polyhedra. Several authors have proposed algorithms to handle non-convex disconnected meshes [Silva et al. 1998]. In practice, these algorithms work well on static scenes and could involve considerable overhead in handling dynamic environments. BSP-tree-based algorithms order fragments of primitives in a back-to-front or a front-to-back order and also work well in static environments.

Image-space algorithms: Many of these algorithms are based on depth peeling or A-buffer functionalities within the GPUs to order the *fragments* of primitives in a front-to-back or back-to-front order. These algorithms may not be able to perform visibility ordering of geometric primitives efficiently, and therefore, may not extend well to applications such as collision computations where primitive-level ordering is essential. Depth peeling [Everitt 2001] performs selection sort on the fragments of primitives on each pixel, and does not exploit temporal coherence or the near-sorted order of these primitives. Therefore, it may not work well in environments with high depth complexity (see Fig. 5). Several issues exist in the architectural implementation of A-buffer [Molnar et al. 1992] and many graphics vendors currently do not support A-buffer functionality in the hardware. Recently, Callahan et al. [Callahan et al. 2005] implemented a fixed-size A-buffer, called k-buffer, using the programmable pipeline and multiple render target functionality (MRT) of the current graphics cards. However, the current GPU implementation of MRT limits the A-buffer size to six and it may not be sufficient for environments with large depth complexity. In practice, these implementations can be texture-bandwidth limited and there is considerable overhead of using supersampling. Moreover, current GPUs do not support supersampling of depth textures. On the other hand, our algorithm is a hybrid algorithm and uses image-space computations for object-level ordering instead of fragment-level ordering. Our algorithm uses the supersampling functionality of frame buffers to generate higher quality images and exploits both spatial and temporal coherence to order nearly-sorted sequences in

almost linear time. Our algorithm involves no preprocessing and can directly handle dynamic environments.

Limitations: Our visibility ordering algorithm has some limitations. First of all, we assume that the input objects are non-overlapping and there is a sorting order among them. Our algorithm can detect cycles in the input but does not split the objects to resolve the cycles. A possible solution is to use other known algorithms to resolve these cycles. Secondly, the occlusion queries used for the comparison operations have additional overhead. The current implementation of occlusion queries on the GPUs is not optimized and reading back the result of an occlusion query from the GPU can stall the pipeline. Finally, the comparison operations for object-level ordering are performed at image-precision for rendering applications.

7 Conclusions and Future Work

We have presented a novel algorithm for computing an object-level visibility ordering among geometric primitives in complex environments. Our algorithm is general and makes no assumptions regarding input models or their motion. We describe a novel sorting algorithm that performs comparisons among objects by performing image-space occlusion computations on the GPUs. Our algorithm exploits temporal coherence between successive frames to improve its performance. We demonstrate its application to rendering with transparency and N-body collision culling in complex environments.

There are several avenues for future work. We would like a more detailed evaluation and comparison of our algorithm with image-based techniques. In order to handle render complex environments with transparency, we would like to combine our algorithm with LOD techniques and also develop techniques to handle cycles in the input. Finally, we would like to use our algorithm for other applications including volume rendering of unstructured grids and image-based reconstruction.

Acknowledgements

Our work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134, 0429583, 0404088 and 0118743, ONR Contract N00014-01-1-0496, DARPA Contract N61339-04-C-0043 and Intel. We would like to thank NVIDIA corporation for their hardware and driver support. We would like to acknowledge Avneesh Sud for video editing, Brandon Lloyd for lighting shaders, and members of UNC Walkthrough and GAMMA groups for useful discussions.

References

AILA, T., MIETTINEN, V., AND NORDLUND, P. 2003. Delay streams for graphics hardware. *ACM Trans. on Graphics 22*, 792–800.

CALLAHAN, S. P., IKITS, M., COMBA, J., AND SILVA, C. 2005. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Trans. on Visualization and Computer Graphics*. to appear.

CARPENTER, L. 1984. The A-buffer, an antialiased hidden surface method. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, H. Christiansen, Ed., vol. 18, 103–108.

CIGNONI, P., C.MONTANI, AND SCOPIGNO, R. 1998. Tetrahedra based volume visualization. In *Mathematical Visualization - Algorithms, Applications and Numerics*, 3–18.

COHEN, J., LIN, M., MANOCHA, D., AND PONAMGI, M. 1995. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proc. of ACM Interactive 3D Graphics Conference*, 189–196.

COHEN-OR, D., CHRYSANTHOU, Y., DURAND, F., GREENE, N., KOLTUN, V., AND SILVA, C. 2001. Visibility, problems, techniques and applications. *SIGGRAPH Course Notes # 30*.

COOK, R., MAX, N., SILVA, C., AND WILLIAMS, P. 2004. Image-space visibility ordering for cell projection volume rendering of unstructured data. *IEEE Trans. on Visualization and Computer Graphics*.

DE BERG, M., OVERMARS, M., AND SCHWARZKOPF, O. 1994. Computing and verifying depth orders. In *sicomp*, 437–446.

DIEFENBACH, P. 1996. *Multi-pass pipeline rendering: Interaction and realism through hardware provisions*. PhD thesis, University of Pennsylvania.

ESTIVILL-CASTRO, V., AND WOOD, D. 1992. A survey of adaptive sorting agorithms. *ACM Computing Surveys*.

EVERITT, C. 2001. Interactive order-independent transparency. Technical report, NVIDIA. http://developer.nvidia.com/object/Interactive_Order_Transparency.html.

FUCHS, H., KEDEM, Z., AND NAYLOR, B. 1980. On visible surface generation by a priori tree structures. *Proc. of ACM SIGGRAPH 14*, 3, 124–133.

GOVINDARAJU, N., REDON, S., LIN, M., AND MANOCHA, D. 2003. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. *Proc. of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 25–32.

GOVINDARAJU, N., LIN, M., AND MANOCHA, D. 2004. Fast and reliable collision detection using graphics hardware. *Proc. of ACM VRST*.

GOVINDARAJU, N., LIN, M., AND MANOCHA, D. 2004. Vis-sort: Fast visibility ordering of 3-d geometric primitives. Tech. rep., Department of Computer Science, University of North Carolina.

GOVINDARAJU, N., RAGHUVANSHI, N., AND MANOCHA, D. 2004. Fast and approximate stream mining of quantiles and frequencies using graphics processors. Tech. rep., University of North Carolina at Chapel Hill.

JOUPPE, N. P., AND CHANG, C. F. 1999. An economical hardware technique for high-quality antialiasing and transparency. *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 85–93.

KNUTH, D. E. 1973. *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA.

KRISHNAN, S., SILVA, C., AND WEI, B. 2001. A Hardware-Assisted Visibility-Ordering algorithm with applications to volume rendering. In *Proc. of Data Visualization*, 233–242.

MAMMEN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications 9*, 4 (July), 43–55.

MAX, N. L., AND LERNER, D. M. 1985. A two-and-a-half-D motion-blur algorithm. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, vol. 19, 85–93.

MAX, N. L. 1993. Sorting for polyhedron compositing. *Focus on Scientific Visualization*, 259–268.

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. PixelFlow: High-speed rendering using image composition. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, E. E. Catmull, Ed., vol. 26, 231–240.

NEWELL, M. E., NEWELL, R. G., AND SANCHA, T. L. 1972. A solution to the hidden surface problem. In *Proc. ACM Nat. Mtg.*

PRZEMYSLAW, R. 1993. Fast genration of depth of field effects in computer graphics. *Computer and Graphics 17*, 5, 593–595.

PURCELL, T., DONNER, C., CAMMARANO, M., JENSEN, H., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, 41–50.

SCHUMACKER, R., BRAND, B., GILLILAND, M., AND SHARP, W. 1969. Study for applying computer-generated images to visual generation. Tech. rep., AFHRL-TR-69-74, US Air Force Human Resources Lab.

SILVA, C. T., MITCHELL, J. S. B., AND WILLIAMS, P. L. 1998. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *IEEE Symposium on Volume Visualization*, 87–94.

SNYDER, J., AND LENGYEL, J. 1998. Visibility sorting and compositing without splitting for image layer decompositions. *Proc. of ACM SIGGRAPH*.

SUDARSKY, O., AND GOTSMAN, C. 1996. Output sensitive visibility algorithms for dynamic scenes with applications to virtual reality. *Computer Graphics Forum 15*, 3, 249–58. Proc. of Eurographics'96.

SUTHERLAND, I. E., SPROULL, R. F., AND SCHUMACKER, R. A. 1973. Sorting and the hidden-surface problem. In *Conf. Proc. Natl. Computer Conf.*

TORRES, E. 1990. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. In *Eurographics '90*, North-Holland, C. E. Vandoni and D. A. Duce, Eds., 507–518.

WILLIAMS, P. L. 1992. Visibility-ordering meshed polyhedra. *ACM Trans. on Graphics 11*, 2, 103–126.

WITTENBRINK, C. 2001. R-buffer: A pointerless a-buffer hardware architecture. *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 73–80.