Depth-Presorted Triangle Lists

Ge Chen[1] Pedro V. Sander[1] Diego Nehab[2] Lei Yang[1,3] Liang Hu[4]
[1]Hong Kong UST [2]IMPA [3]Bosch Research North America [4]Google Inc.

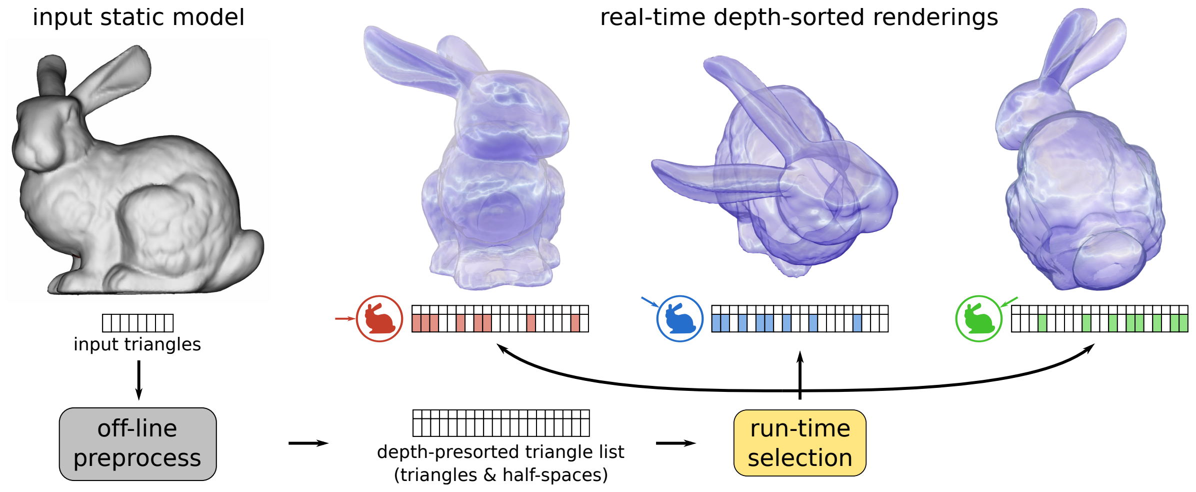

input static model
real-time depth-sorted renderings
input triangles
off-line preprocess
depth-presorted triangle list
(triangles & half-spaces)
run-time selection

Figure 1: *Our off-line preprocessing algorithm produces a depth-presorted triangle list from a static 3D input model. The list contains copies of the model that are depth-sorted relative to* any *viewpoint outside its bounding volume. Each triangle comes annotated with a half-space test that enables a fast run-time algorithm to select exactly the triangles needed to render a complete depth-sorted model for a given viewpoint.*

Abstract

We present a novel approach for real-time rendering of static 3D models front-to-back or back-to-front relative to any viewpoint outside its bounding volume. The approach renders depth-sorted triangles using a single draw-call. At run-time, we replace the traditional *sorting* strategy of existing algorithms with a faster triangle *selection* strategy. The selection process operates on an extended sequence of triangles annotated by test planes, created by our off-line preprocessing stage. Based on these test planes, a simple run-time procedure uses the given viewpoint to select a subsequence of triangles for rasterization. Selected subsequences are statically presorted by depth and contain each input triangle exactly once. Our method runs on legacy hardware and renders depth-sorted static models significantly faster than previous approaches. We conclude demonstrating the real-time rendering of order-independent transparency effects.

Links: DL PDF

1 Introduction

In real-time rendering applications that employ the Z-buffer for visibility determination [Catmull 1974], there are still many scenarios in which depth-sorting is necessary or desirable. The most common is order-independent transparency or translucency. Since the compositing operation is not commutative [Porter and Duff 1984], blending must happen in depth-sorted order.

As we discuss in section 2, a large number of techniques have been proposed for performing real-time depth sorting. In this work, we present a technique that possesses a unique combination of desirable properties. It can be implemented with the standard graphics pipeline, requires a single rendering pass, uses a fixed amount of memory, produces exact results, is very simple to integrate with existing rendering engines, and is extremely efficient.

These advantages come with certain limitations. Whereas most previous depth-sorting algorithms work seamlessly with deformable geometry at run-time, our method assumes static geometry viewed from outside the model's bounding volume. Finally, our preprocessing stage can take hours to complete when run on larger models, and the resulting data-structure consumes more memory than the input.

We target performance-critical applications that must render a number of moderately complex static objects with translucency effects, such as computer games. In this scenario, which we demonstrate in the results section, the relative order between objects is determined by the CPU, and our method ensures correct triangle ordering within each object. During game development, instant feedback can be provided to artists using earlier, less efficient methods. At the end of the release cycle, required models can be preprocessed and the engine set up to take advantage of the simplicity of our run-time component, and of the large performance gains that ensue. Since only the transparent components of objects with translucency effects must be preprocessed, the increase in run-time memory is not a significant limitation either.

Our key insight is that the space of different triangle orders that result from depth-sorting a triangle model under each possible viewpoint constitutes a tiny fraction of all triangle permutations. Moreover, this “space of depth-sorted orders” is extremely redundant in the sense that, with few modifications, the same order is valid for large portions of the viewpoint space.

We explore this insight in the following way. During a preprocessing stage, we create an extended sequence of triangles (i.e., a list that contains one or more instances of each input triangle). By construction, this extended sequence is such that there is a view-dependent subsequence of triangles that is depth-sorted relative to each and every viewpoint outside of a bounding volume. These view-dependent subsequences are guaranteed to include each input triangle exactly once. Within the extended sequence, each triangle is paired with a half-space. The entire extended sequence is sent for rendering by the GPU, at which point an extremely simple and efficient run-time procedure selects those triangles for which the associated half-space contains the viewpoint. The result is a depth-sorted rendering of the original input triangles, relative to the viewpoint.

Since there is no run-time sorting, only a trivial run-time selection procedure, our method runs at an order of magnitude faster than previous approaches. This selection happens in a single rendering pass, and can be implemented at any stage of the rendering pipeline (vertex, geometry, or fragment shaders). Finally, there is no CPU intervention and results are guaranteed to be exact.

In summary, our contributions include:

- The fastest single-draw-call, exact, real-time depth-sorted rendering algorithm for static models;
- A preprocessing algorithm for creating a compact extended sequence of triangles and associated half-spaces containing depth-sorted subsequences relative to all viewpoints;
- Three versions of the run-time algorithm for selecting the subsequence associated to a given viewpoint, all leading to state-of-the-art run-time performance.

The rest of the paper is organized as follows. In section 2 we position our method in context with previous work. Section 3 presents two simple examples that will be helpful in understanding the preprocessing algorithm, which is described in section 4. The run-time selection algorithms are presented in section 5. Section 6 provides relevant statistics specific to our method, as well as performance comparisons against previous approaches. We conclude by revisiting the strengths and weaknesses of the method and suggesting venues for future work.

2 Related work

The problem of depth-sorting is tightly connected to the problem of visibility, for which there is a vast amount of prior work. Here, we focus on the methods we believe are most related to ours. Note that many depth-sorting methods can be used to render dynamic (or even self-intersecting) geometry. Naturally, when comparing them against our approach, we assume the target application does not require this functionality.

The most well-known approach for rendering depth-sorted static geometry is the BSP tree [Fuchs et al. 1980; Paterson and Yao 1989]. Each node in a BSP tree includes a half-space test. The root node represents the entirety of space, and each subtree represents the fraction of the parent's space that reside in one of the two half-spaces. The BSP tree is created during a preprocessing stage, much like our extended triangle list. To render from a BSP, the tree is traversed recursively at run-time. For back-to-front rendering, when visiting a node, the subtree representing half-spaces that contain the viewpoint are visited last. In our method, the half-spaces are instead used to separate the viewpoints for which a given individual triangle instance must be drawn from those for which it must be ignored.

Other than the BSP, additional data-structures used for depth sorting include directed acyclic graphs [Williams 1992], feudal priority trees [Chen and Wang 1996], and Voronoi diagrams [Fukushige and Suzuki 2006]. In contrast, our extended list is a flat data-structure, potentially including multiple copies of each triangle, each of which is selected or not by the viewpoint. Our method is therefore better suited for modern GPUs, where recursiveness and pointer manipulation are difficult or otherwise inefficient to accomplish.

Early CPU-based approaches also generate flat or semi-flat data-structures. Newell et al. [1972] and Goad [1982] both describe automatic procedures that sort triangles into a priority order for given viewpoints. Schumacker et al. [1969] propose a method that separates the scene into convex clusters during preprocessing. The faces in each cluster can be assigned a fixed order which, after back-face culling, provides correct visibility from any viewpoint. These clusters must be mutually separable by planes, and the order of displaying these clusters is computed at run-time. Our method leverages GPU hardware in order to render the model using a single draw call without requiring ordering computations at run-time.

The most general techniques operate at the pixel level. The typical strategy is to generate per-pixel fragments lists and then sort each list, as in the A-buffer [Carpenter 1984]. This is challenging, since there is no way to guess the total number of fragment storage that will be needed, or the number of fragments needed per pixel. New hardware capabilities have recently enabled the use dynamic linked lists to collect fragments and blend them in order [Yang et al. 2010], or instead to count the number of fragments in a first rasterization pass and store them into individual arrays during a second pass [DX10 SDK 2010], before compositing them in parallel [Patney et al. 2010]. Our method requires a single pass, uses constant memory, and is significantly faster, particularly when multiple samples per pixel are used for anti-aliasing (e.g. MSAA).

Many modifications to the rendering pipeline have been proposed that support variations of the A-buffer. These include the R-buffer [Wittenbrink 2001], the F-buffer [Mark and Proudfoot 2001], Delay Streams [Aila et al. 2003], and the FreePipe architecture [Liu et al. 2010]. Our method runs on the standard rendering pipeline.

Another alternative is to render the scene as many times as required by the maximum depth complexity. At each pass, the Z-buffer is used to select the next closest fragment [Mammen 1989; Everitt 2001; Thibieroz 2008] (much like selection sort). For performance reasons, variations of this idea split the scene into pieces that are presorted [Wexler et al. 2005], peel multiple layers per pass [Liu et al. 2006; Bavoil and Myers 2008], or exploit any ordered structure that may already be present [Carr et al. 2008]. The algorithms are exact and the amount of memory needed is fixed. Our method is simpler and, depending on the depth-complexity, significantly faster.

Many methods rely on approximations for performance reasons. A common approach is to limit the maximum number of fragments per pixel [Jouppi and Chang 1999; Myers and Bavoil 2007a,b; Bavoil et al. 2007; Liu et al. 2009; Huang et al. 2010; Salvi et al. 2011], in which case some heuristic must be used to merge or evict overflowing fragments. In the case of order-independent transparency, ignoring order is sometimes acceptable [Meshkin 2007; Bavoil and Myers 2008]. Another method that avoids the need for sorting is stochastic transparency [Enderton et al. 2010; Laine and Karras 2011], a refreshing new take on *screen-door* transparency [Foley et al. 1990; Mulder et al. 1998]. Our method always produces noise-free renderings and is significantly faster. Some approximations work particularly well for hair, volumetric data, or both [Kim and Neumann 2001; Callahan et al. 2005; Yuksel and Keyser 2008; Sintorn and Assarsson 2008, 2009; Jansen and Bavoil 2010; Salvi et al. 2011]. Our method is not suitable for such high depth-complexities. Finally, in the context of overdraw reduction, some static orders may succeed in eliminating most of the overdraw [Nehab et al. 2006; Sander et al. 2007].

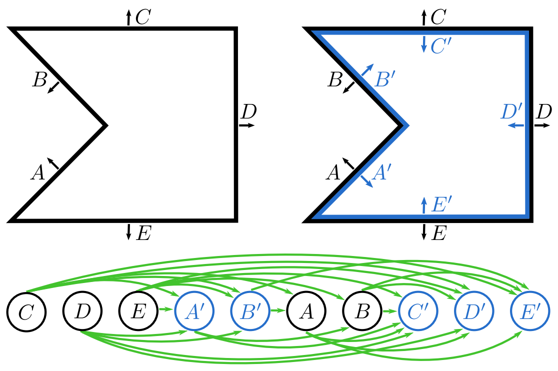


Figure 2: *A simple 2D example (left), after duplicating triangles (right), and the resulting occlusion graph (bottom).*

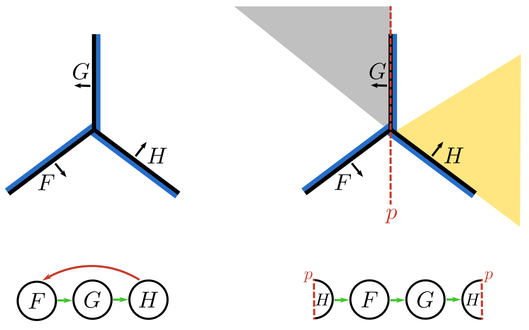


Figure 3: *A simple example with a cycle (left), and the result of duplicating triangle H using the cutting plane p (right).*

3 Motivating examples

We begin by presenting two simple examples that will motivate our strategy for building depth-presorted triangle lists. For simplicity, we show both examples in 2D (see figures 2 and 3), with triangles represented by line segments. The arrows next to each triangle represent the normal direction.

3.1 A view-independent depth-presorted model

Consider the non-convex model in figure 2 (top left). We will show that it is possible to produce depth-sorted renderings of this model (including back-facing triangles) from *any* viewpoint, using a fixed triangle ordering. At first sight, this seems difficult, given that the occlusion relationship between two triangles may be reversed when the viewpoint changes (i.e., when viewing from above, C occludes E, while when viewing from below, E occludes C).

The trick is to duplicate each triangle by including an additional instance with the opposite orientation. The augmented model, with newly instanced back-facing triangles in blue, is shown in figure 2 (top right). Although there are now twice as many triangles as in the original model, back-face culling ensures the hardware will rasterize at most one instance of each triangle. The additional freedom awarded by the duplication will allow us to create the triangle ordering we seek.

In the case of figure 2 (with back-face culling), triangles $C'D'E'$ never occlude other triangles, since there are no triangles in the half-space behind each of them. It is safe to place them in the end of the list. By the same token, CDE are never occluded by any triangle: they can be added to the front of the list. All that remains is to place the remaining triangles $AA'BB'$ between CDE and $C'D'E'$ (in some appropriate relative order). These triangles have more complex occlusion relationships. Specifically, A' may occlude B and B' may occlude A. Thus, we need to find an order such that A' is in front of B and B' is in front of A. Multiple such orders exist and in particular the following order satisfies all requirements: $CDEA'B'ABC'D'E'$. When processing the list of 10 triangles in this order, and from any viewpoint, back-face culling will select exactly the 5 triangles that render the model sorted by depth.

To solve the general case automatically, we first construct a graph that captures all occlusion relationships. This occlusion graph is shown in figure 2 (bottom). Each node represents a triangle, and there is an edge connecting a triangle X to a triangle Y if and only if there is a viewpoint in which triangle X *occludes* triangle Y. We seek a topological sort of the occlusion graph, i.e., an ordering in which there are no *back-edges* [Skiena 2008]. There is such an order whenever the directed graph is acyclic (i.e., it is a DAG), as is the case of the occlusion graph in figure 2. Surprisingly, it is possible to construct an ordering as described above for many simple objects.

3.2 A model requiring view-dependent selection

Unfortunately, in the general case, there may be cycles in the occlusion graph, even with back-face culling, so that no topological sort can be found. The simplest such case is shown in figure 3 (left). The corresponding occlusion graph containing a cycle is shown underneath it. (We have omitted the back-facing triangles from the graph to isolate this single cycle.)

The key observation is that H can only occlude F if the viewpoint is inside the yellow region, and G can only occlude H if the viewpoint is in the gray region. Furthermore, these regions can be separated by a cutting plane p (the red dashed-line). Thus, we can solve the cycle problem by creating an extra copy of H, as shown in figure 3 (bottom right), and using the cutting plane p to select the copy of H to render depending on which side of p the current viewpoint lies. This culling operation, which is in addition to back-face culling, can be performed very efficiently at run-time (see section 5).

4 Preprocessing algorithm

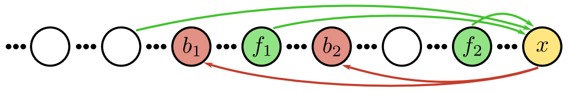
In this section, we describe an algorithm for automatically generating a depth-presorted triangle list for arbitrary input models. While in this description we assume a back-to-front ordering is desired, the algorithm can be trivially adjusted to create a front-to-back order. Since our method focuses on static models, we assume intersecting triangles have been split prior to invocation of our preprocessing algorithm, so the input list contains no intersecting triangles. Moreover, we assume there is a way to depth-sort the triangles from any viewpoint. In other words, we assume there are no single-viewpoint visibility cycles in the input model. Self-intersections, which are common in production models, must anyway be eliminated when rendering with transparency. Additional preprocessing can break cycles by splitting triangles when needed.

As in the example of section 3.1, we start by creating back-facing duplicates for each input triangle in order to relax the occlusion restrictions. Then, we compute the occlusion graph and generate a preliminary ordering. If the graph has no cycles, a topological sort completely solves the problem [Skiena 2008]. When there are cycles, a good preliminary ordering is one that minimizes the number of back-edges. Finding an optimal preliminary order is equivalent to solving the minimum feedback arc-set problem, which is NP-complete [Karp 1972]. Fortunately, the correctness of our algorithm does not depend on optimality, and therefore we use a fast heuristic that is detailed in section 4.2.

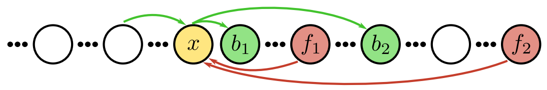
The final and most important preprocessing stage deals with back-edges in the preliminary order. To do so, we scan the ordering from start to finish, duplicating triangles and associating them with half-spaces whenever we find a back-edge. This step follows along the lines of the example of section 3.2, and is described below.

4.1 Duplicating triangles to work around back-edges

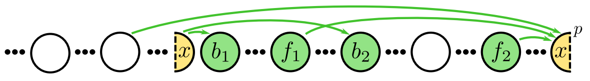
At each iteration, we process a graph node, moving along the preliminary order from start to finish. Nodes that send no back-edges can be safely ignored, but all back-edges must be dealt with. An example will guide us through the algorithm:



Here, moving along the list from right to left, the current iteration reaches a yellow node x with back-edges (in red) that must be eliminated. (Edges that are not adjacent to x are irrelevant to this iteration and have been omitted.) To remedy the two back-edges pointing to the b_* nodes, it often suffices to move x immediately before b_1. This works whenever there are no triangles between b_1 and x with edges that point to x. Unfortunately, this is not the case here and moving x gives rise to two new back-edges, coming from the f_*:

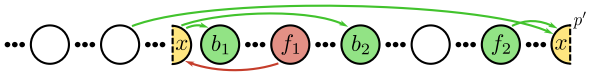


To address this issue, we proceed as in section 3.2, and look for a plane p that partitions the viewpoint space into two half-spaces. Plane p must be such that the set of viewpoints from which x occludes the b_* is contained in one of the half-spaces, whereas the set of viewpoints from which the f_* occlude x is contained in the *other* half-space. An algorithm for finding such a plane (when it exists) is given in section 4.1.1. With p, we can duplicate x as follows:



We then annotate the *right copy* of x with the half-space test for p, so that the run-time algorithm described in section 5 will only render this copy of x if the viewpoint is in the correct half-space. The left copy of x does not need to be annotated with the plane. Using Z-buffering with depth-test of *less* ensures that the fragments generated by the left copy of x are shaded and stored if and only if the right copy was skipped (Recall the back-to-front rendering proceeds from right to left.) This fact allows us to avoid dealing with multiple half-space tests in most practical cases.

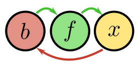
Whenever we can find p that completely separates the viewpoints associated to edges *to* the b_* from those associated to edges *from* the f_*, it is clear we can move on to the next offending node. However, such an ideal cutting plane may not exist. Thus, duplicating x with p' may introduce into the graph new back-edges emanating from a subset of the f_*, as shown below:



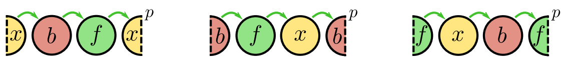
In that case, we proceed to the next iteration and postpone handling of new back-edges to the iteration that analyzes the nodes from which they emanate (f_1 in the example). It is therefore possible for the left copy of a node that has been duplicated to be further duplicated by the iteration that eventually processes it. As we noted earlier, in general only the right copy is assigned a half-space test: the Z-buffer deals with the left copy. Therefore, as we traverse the list from right to left, each node copy is processed only once and is assigned at most one plane.

It is important to notice that, as long as we manage to separate *at least one* of the edges between f_* and x from *at least one* of edges between x and b_*, we have made progress. This is because we have succeeded in reducing the total number of edges that must be considered in the future. Since that number is finite, the algorithm terminates. We now prove that, under mild assumptions, this is always the case.

We focus on three nodes in the graph: the current node x, the right-most node b to which x sends a back-edge, and the right-most node f, between b and x, such that f sends a forward-edge to x. We further assume that there exists a path from b to f, so that there is a cycle in the graph (otherwise we could simply move b to the right of x). Note that cycles of two nodes do not exist since there are no self-intersecting triangles.



If the viewpoints of edge (x, b) can be separated from the viewpoints of edge (f, x), we duplicate x as usual. If the viewpoints of edge (x, b) can be separated from those of edge (b, f), we duplicate b instead. Finally, if the viewpoints of edge (b, f) can be separated from those of edge (f, x), we duplicate node f. These configurations are shown below:



At least one of these separations *must* be possible. Otherwise, there would be a viewpoint from which the entire cycle is visible. This would preclude the existence of a depth-sorted triangle ordering for this viewpoint. But this contradicts our assumptions and therefore completes the proof.

Note that if we follow this strategy that duplicates b by moving it to the right, it is possible that b may have other back-edges and require further duplication. In that case, additional planes will have to be associated with the right copy of b. Fortunately, this must be relatively rare since we have never observed it in practice.

4.1.1 Defining the cutting plane

Let us begin with a few definitions. We associate an *occlusion region* $O_{i\to j}$ to each edge from node i to node j, defined as the set of viewpoints from whence i occludes j. Thus, from viewpoints outside of $O_{i\to j}$, it is as if the edge did not exist. Similarly, we associate to each node i (i.e., to each triangle instance) a *rendering region* E_i, representing the set of viewpoints from which i is rendered. As nodes are duplicated, rendering regions of new nodes are cumulatively restricted to their associated half-spaces and can therefore become rather small. Naturally, outside E_i or E_j, it is as if an edge between i and j did not exist either. Therefore, each edge is relevant only inside a reduced occlusion region

$$\bar{O}_{i\to j} = O_{i\to j} \cap E_i \cap E_j. \qquad (1)$$

With these definitions in hand, we can define the problem of finding an appropriate cutting plane p. Let x be the current node being analyzed. Assume x has back-edges pointing to a set of nodes b_*, as well as forward-edges arriving from n distinct nodes f_* that lie between x and b_1. We want to duplicate x, placing the new copy to the left of b_1. We hope to find a plane p that partitions the edges among the two instances of x: back-edges go to the new (left) instance, forward-edges remain with the current (right) instance.

Formally, the ideal plane p separates the regions

$$R_b = \bigcup_{b \in b_*} \bar{O}_{x \to b} \quad \text{and} \quad R_f = \bigcup_{f \in f_*} \bar{O}_{f \to x}. \tag{2}$$

Since the ideal plane may not exist, we *greedily* look for a good alternative. We start our search by separating the back-edge region R_b from an empty forward-edge region

$$R_f^{(0)} = \emptyset. \tag{3}$$

(The plane of x itself does the job.) Then, we try to progressively add forward-edge regions to R_f by setting

$$R_f^{(i)} = R_f^{(i-1)} \cup \bar{O}_{f_i \to x} \tag{4}$$

if we can find a separating plane between R_b and $R_f^{(i-1)} \cup \bar{O}_{f_i \to x}$. Otherwise, we set

$$R_f^{(i)} = R_f^{(i-1)} \tag{5}$$

and consider adding the occlusion region for the next forward-edge. When we are done considering all forward-edges (in order, for efficiency), we assign the last successful separating plane to the original x, and move the edges not accounted for the final region $R_f^{(n)}$ to its new copy. These will be addressed when the time comes, by duplicating the required subset of the f_*.

Although we can conceive of a situation where it is not possible to separate any of the $\bar{O}_{f_i \to x}$ from *all* of R_b, this has not happened in any of our test cases. As shown in section 4.1, it is in fact always possible to separate *at least one* $\bar{O}_{f_i \to x}$ from *at least one* $\bar{O}_{x \to b_j}$, and we can rely on this fall-back procedure to ensure the termination of the algorithm.

4.1.2 Computing the cutting plane

In this section, we detail the geometric operations used to efficiently compute R_f and R_b, and ultimately the cutting plane p that separates them. Computations involving convex-hulls, half-space intersections, and linear-programming are performed using the Qhull [Barber et al. 1996] and `lp_solve` [Berkelaar et al. 2004] libraries.

Intersection of viewpoint regions Determining the reduced occlusion region $\bar{O}_{i \to j}$ amounts to finding the intersection between three convex regions. To do so, we form the intersection between all half-spaces defining each of $O_{i \to j}$, E_i, and E_j. Since this intersection is often empty, as an optimization that avoids the costly half-space intersections, we first check if all vertices of either E_i or E_j are on the outside of one of the planes bounding $O_{i \to j}$. If so, the intersection is empty.

Union of viewpoint regions We conservatively approximate each region R_b and $R_f^{(i)}$ by their convex-hulls. Note that this does not compromise in finding the ideal cutting plane p, since any plane that separates $R_f^{(i)}$ from R_b also separates their convex-hulls.

Cutting plane Given the convex-hulls of R_b and $R_f^{(i)}$, we use linear programming to determine whether they intersect. If so, no cutting plane can separate them and the algorithm proceeds without updating $R_f^{(i)}$. Otherwise, we check if any of the bounding planes of the two regions is suitable as a cutting plane (i.e., all of the vertices of the other region lie on the opposite side of the plane). If that does not produce a valid cutting plane, then planes are formed using all possible combinations of vertices from one region and edges from the other region. At least one of these planes will necessarily separate the two convex-hulls and this becomes the candidate plane p.

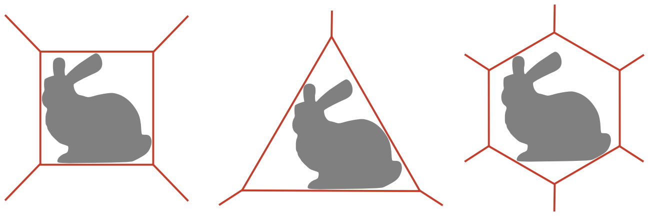


Figure 4: *Viewpoint partitioning optimization. To strike the proper balance between memory consumption and run-time performance, we generate independent depth-presorted triangle lists different parts of the viewpoint space.*

4.2 Viewpoint-space partitioning

In practice, we found that generating a single depth-presorted triangle list for the entire range of viewpoints requires far too many triangle duplications and leads to a list with many times as many triangles as the input model. By dividing the space of viewpoints into a small number of partitions and creating independent depth-presorted triangle lists for each partition, we get a much better trade-off between total memory usage and run-time performance (see results in section 6).

To divide the space of view points into partitions, we first enclose the model in a bounding polyhedron with a given number of faces (we have experimented with 4, 6, 16, and 64). Each partition is defined by one of the polyhedron faces, and by the boundaries with neighboring partitions, as in figure 4. By limiting the set of valid viewpoints to lie outside the bounding polyhedron, we further reduce the number of constraints in the occlusion graph. The preliminary order on which the preprocessing algorithm operates is simply a depth-sorted list of triangles relative to some viewpoint in the corresponding partition. The only further modification is that when computing the depth-presorted triangle list for each partition, all of the rendering regions E_* are further restricted to the corresponding partition. We experimented with different orientations for the polyhedron and obtained very similar results.

Depth-presorted lists for each partition can be computed simultaneously, leveraging the parallelism of multi-core CPUs. The lists are concatenated into an index buffer with multiple segments. At run-time, we issue a draw call that renders the segment of the index buffer corresponding to the partition that contains the current viewpoint. This can be easily accomplished by specifying the starting buffer index and total number of triangles in the command that issues the draw call. Thus, the CPU performs the coarse-level viewpoint selection (which is trivial) and the GPU completes the fine-level triangle selection, as described below.

5 Run-time selection algorithm

The task at run-time is to select those triangles from the depth-presorted list that pass a half-space test. Recall each triangle t is annotated by a test plane p_t (a 4D vector storing the plane equation). To decide whether to render a triangle t from viewpoint v_{xyz}, we use a single dot product, which is extremely efficient in modern GPUs:

$$\mathbf{dot}\big(p_t, [v_{xyz}, -1]\big) > 0. \tag{6}$$

For triangles with no associated test plane, we use $p_t = [0, 0, 0, -1]$, which causes the test to always succeed.

The test can be implemented in either the vertex, geometry, or fragment shader programmable pipeline stages. Although each alternative has advantages and disadvantages depending on the application and rendering configuration (see section 6), we favor the fragment shader test:

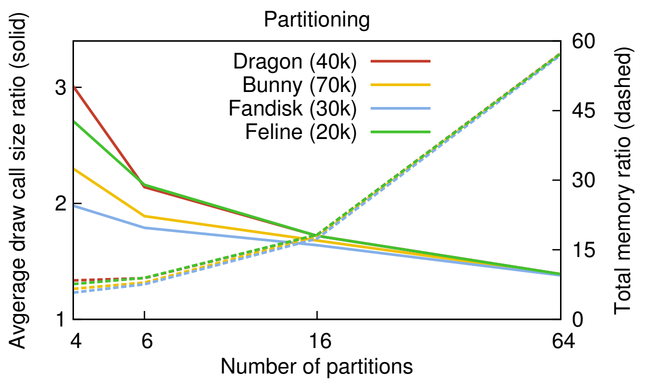


Figure 5: *Varying the number of segments as a trade-off between memory consumption and the size of each individual draw call.*

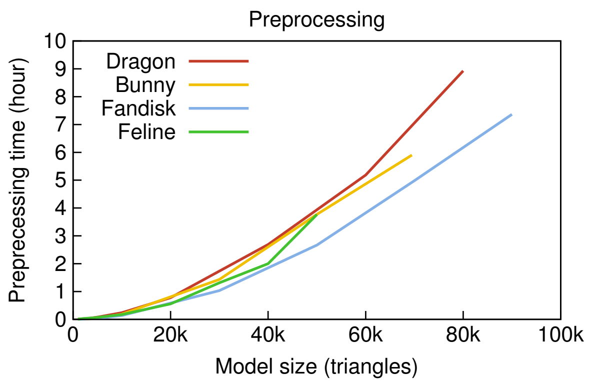


Figure 6: *Preprocessing time for different meshes at different resolutions (all meshes preprocessed with 6 segments).*

Option 1: Fragment shader (FS) The 4D plane vectors are kept in a 32-bit 4-channel texture indexed by triangle ID. (Although in practice the need has not arisen, it would be simple to encode an entire linked-list of test-planes in the texture.) The FS issues the lookup and performs the dot product. Since one dot product test is performed per triangle fragment, this method may suffer in pixel-bound scenes. Fortunately, the tests are inexpensive and the texture access is coherent.

Option 2: Vertex shader (VS) The 4D plane vectors are sent as vertex attributes. When the dot product test fails, the VS moves the vertex to the viewpoint position so that it gets culled by the near plane, otherwise it applies the standard vertex transformation. Since all three vertices of each triangle perform the same exact test, the result is consistent. The drawback of this approach is that triangles can no longer share vertices, which may impact vertex bound scenes.

Option 3: Geometry shader (GS) The 4D plane vectors are kept in a 32-bit 4-channel texture indexed by triangle ID. The GS emits only the triangles that pass the dot product test. Although this is the most natural implementation, the additional pipeline stage can impact performance if the application is not already using a GS.

6 Results

In this section we discuss the performance and memory consumption of our approach. All experiments were conducted on an Intel® Xeon® 2.27GHz E5520 CPU with 12GB of RAM and an AMD Radeon HD 6970 GPU. We used models with resolutions ranging from 1,000 to 100,000 triangles, which we believe are representative of the resolution and depth complexity of most static models found in games. All renderings used alpha blending for proper semi-transparency when rendering back-to-front as in figure 1. The run-time performance results in figures 7, 8, and 9 are averaged over 256 viewpoints around the model and reported as ratios relative to a baseline standard rendering, measured directly in clock ticks. The baseline renders all input triangles in a order optimized for vertex-cache locality (completely disregarding depth-sorting). For example, if rendering a depth-sorted model with our method is twice as expensive as the baseline rendering, we report the performance ratio of two.

Segment partitioning Figure 5 shows the effect of increasing the number of viewpoint partitions on the average number of triangles per segment (i.e, that are actually processed by the draw-call at run-time), as well as on the total memory consumption. The average draw call size is reported as a ratio to the number of triangles in the input triangle list. The total memory ratio is reported as a ratio to the total memory used by a standard triangle list with no duplicated triangles. The calculation assumes that each vertex contains a position, normal, and texture coordinates. Note that although increasing the number of partitions increases memory consumption, it reduces the average segment size. It is important to note that the total memory increase only happens for the transparent parts of models that have any transparency effect at all. Furthermore, the vast majority of memory in modern games is consumed by textures, with geometry lagging far behind. We found that using 6 viewpoint partitions provides a reasonable trade-off for the models we tested. The amount of memory used is not significantly higher than that of 4 partitions, but there is a marked decrease in average segment size. This translates to performance gains at rendering time. For the remainder of the experiments in this section we used 6 viewpoint partitions.

Preprocessing Figure 6 shows the preprocessing time of our method. Depending on model and number of triangles, it can take anywhere between a few minutes to several hours to complete. The computation of different segments was parallelized using multiple CPU cores. While the preprocessing algorithm can be slow on large input, it is important to point out that it only needs to be executed once for each static model. Further optimizations could improve the preprocessing time, but we instead concentrated our efforts on optimizing for better run-time results, which is the ultimate goal.

Rendering approaches Figure 7 compares performance of the three versions of our run-time selection procedure, each using a different stage of the pipeline. The geometry shader option is significantly slower due to the fact that it adds a new shading stage to the rendering pipeline. Therefore, we only see this option being viable when the rendering effect already requires a geometry shader, in which case it would require simply adding the plane test to an existing shader. The vertex shader and fragment shader versions are significantly faster. For very low-resolution models, the rendering is fill-bound, making the fragment shader option more costly. For medium- and high-resolution models, however, the vertex processing overhead of the vertex shader approach dominates. Thus, for most practical model sizes, the fragment shader option is the most efficient, with our algorithm being *only* 2–3$\times$ slower than baseline.

Overall performance We compared the overall performance of our fragment shader algorithm with a selection of state-of-the-art real-time depth-sorting algorithms: per-pixel dynamic linked lists (LL) [Yang et al. 2010], stochastic transparency (ST) [Enderton et al. 2010], and dual depth peeling (DDP) [Bavoil and Myers 2008]. ST is an approximate algorithm, while DDP and LL produce exact results. (DDP uses occlusion queries to ensure that no further passes are required and LL uses a sophisticated sorting operation that is not available in legacy hardware.) Tests were run for each of

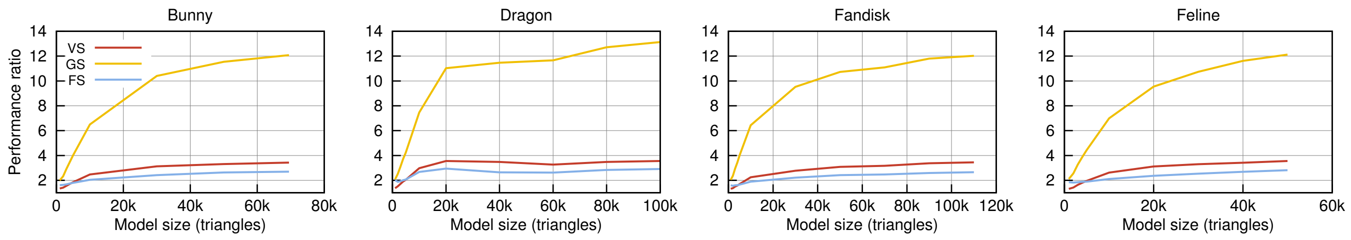


Figure 7: *Performance comparison between geometry shader, vertex shader, and fragment shader implementations.*

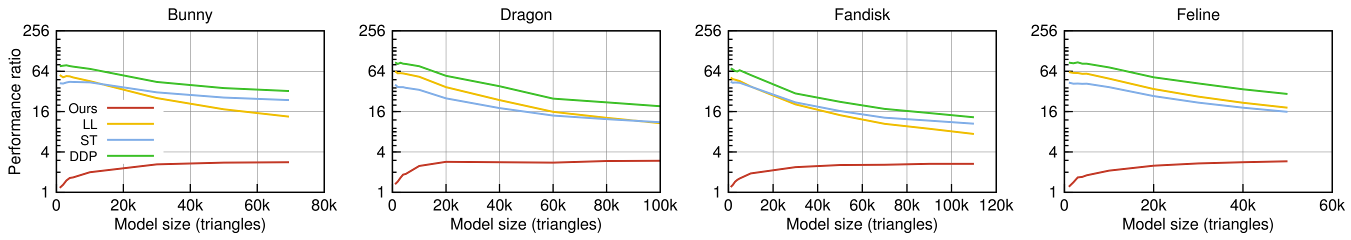


Figure 8: *Performance comparison with related techniques using a screen resolution of 640×480.*

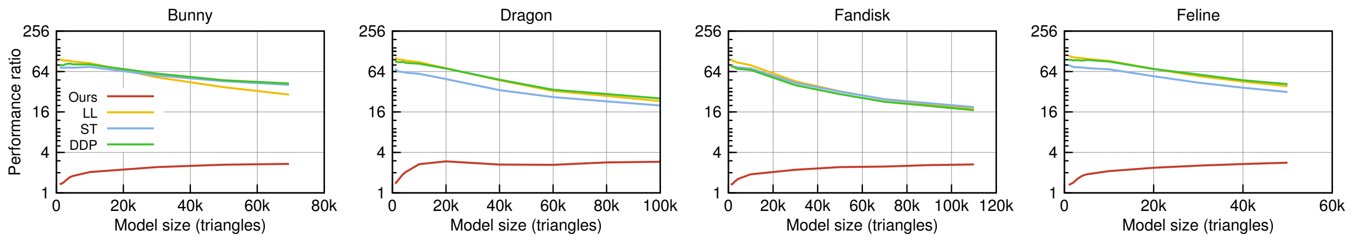


Figure 9: *Performance comparison with related techniques using a screen resolution of 1280×720.*

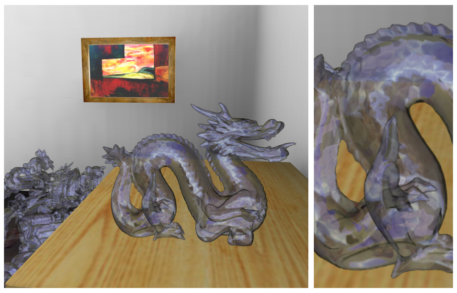
our models at 640×480 (figure 8) and 1280×720 (figure 9) screen resolutions to demonstrate the different trade-offs involved. Our algorithm is significantly faster than the alternatives, particularly at high screen resolutions. Conversely, the advantage of our method is even higher for smaller models. This is because the relative overhead of increasing pixel count is negligible in our method, in contrast to the effect of increasing input triangle count. Nevertheless, even at low screen resolutions and with larger models, our method is significantly faster than the alternatives. For our method to become slower than the alternatives, it would require the combination of an enormous amount of geometry with either small screen coverage (for pixel based approaches) or very small maximum depth-complexity (for depth-peeling approaches). Since GPUs are heavily optimized for batch processing with few large index buffers, our simpler, single-pass method is faster for practical scenarios of even up to several million of transparent triangles.

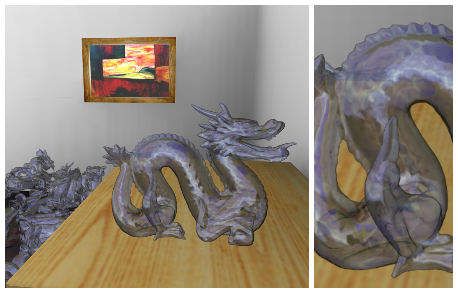
Complex scene We tested our approach on a complex *Room scene* (figure 10) consisting of a physical simulation with multiple semi-transparent dragons interacting and colliding with one another. We used a screen resolution of 1280×720 and 4X-MSAA. Refer to the accompanying video for the entire animation sequence. Inter-model depth-sorting was performed in the CPU using the models' convex bounding volumes, which are also used for collision detection. This is trivial and fast for a small number of objects. Since the convex bounding volumes are not allowed to inter-penetrate, the sorting results are guaranteed to be correct. Standard rendering (figure 10a) does not render the triangles of each model in depth sorted order and therefore yields an incorrect transparency effect. For example, the rear left foot of the dragon shown in the closeup appears very prominently even though it is behind the body. As expected, our method (figure 10b), LL, ST, and DDP generate correct results. We measured rendering time of the entire scene for a varying number of dragons (figure 10c), resulting in a scene geometric complexity of 40,000 to 2,000,000 triangles (1 to 50 dragons). Clearly, the performance is inversely proportional to the number of dragons in the scene and our method is significantly faster compared to other approaches. Furthermore, our method and DDP produce exact results, which is not the case for ST and LL with MSAA.

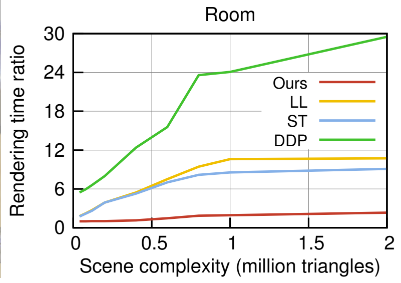
Game scene To show that our method also applies to a more typical *Game scene* (figure 11), we created another experiment including an animation sequence in which a character moves around a game scene with multiple solid objects and a semi-transparent model. Refer to the accompanying video for the entire animation sequence. To get a sense of the performance implications of our approach on these scenarios, we varied the resolution of our semi-transparent model in order to present our performance results as a function of the percentage of scene primitives that are semi-transparent. So, for instance, if approximately 15% of the game's primitives are semi-transparent, the slowdown for having the triangles in sorted order is just about 1.1×. On the other hand, in a more extreme scenario where the semi-transparent objects start to dominate by having 50% of the primitive count, the slowdown is a more significant 1.5×.

7 Conclusion

We presented a new algorithm for efficient, exact, depth-sorted rendering of static triangle models. Our method produces a depth-presorted triangle list in which each triangle is annotated by test planes. These lists can be rendered in depth-sorted order using a single draw call. Given a viewpoint, a simple run-time culling procedure executed by the GPU rasterizes a subsequence of the triangles that produce a depth-sorted rendering of the model relative to that viewpoint. We show that this approach is significantly faster than the alternative methods. The main limitations of our method is

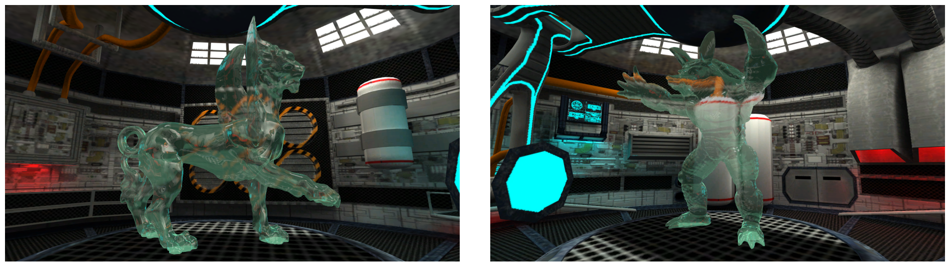

(a) Standard rendering

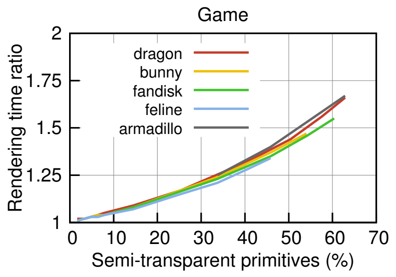

(b) Our result



(c) Performance comparison

Figure 10: *Results and performance comparison of the room scene with a screen resolution of 1280×720 and 4X-MSAA.*


(a) Renderings with our approach



(b) Performance comparison

Figure 11: *Results and performance comparison of the game scene with a screen resolution of 1280×720 and 4X-MSAA.*

that it is only suitable for static models, and for viewpoints outside of a bounding polyhedron. However, we feel that there exists a wide range of applications for which these limitations do not matter.

Finally, we believe that this novel *selection* based scheme using a single draw call is a significant departure from existing methods, most of which require either sorting or multipass rendering. We believe that this direction is worth further investigation, particularly on ways to handle deformable models. It would be interesting to consider generating a set of orders that, in conjunction, allow for a limited range of deformation at run-time. Alternatively, our fine-grained triangle-level technique can be combined with coarse-level dynamic sorting to enable animated characters with rigid parts.

Acknowledgments

This work was partly supported by Hong Kong's RGC GRF grant #619509 and by an INST grant from FAPERJ.

References

AILA, T., MIETTINEN, V., and NORDLUND, P. 2003. Delay streams for graphics hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 22(3):792–800.

BARBER, C. B., DOBKIN, D. P., and HUHDANPAA, H. 1996. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483.

BAVOIL, L., CALLAHAN, S. P., LEFOHN, A., COMBA, J. L. D., and SILVA, C. T. 2007. Multi-fragment effects on the GPU using the *k*-buffer. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, 94–104.

BAVOIL, L. and MYERS, K. 2008. Order independent transparency with dual depth peeling. NVIDIA whitepaper.

BERKELAAR, M., EIKLAND, K., and NOTEBAERT, P. 2004. `lp_solve` 5.5, open source (mixed-integer) linear programming system. Software.

CALLAHAN, S. P., IKITS, M., COMBA, J. L. D., and SILVA, C. T. 2005. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295.

CARPENTER, L. 1984. The A-buffer, an antialiased hidden surface method. *Computer Graphics (Proceedings of ACM SIGGRAPH 1984)*, 18(3):103–108.

CARR, N., MĚCH, R., and MILLER, G. 2008. Coherent layer peeling for transparent high-depth-complexity scenes. In *Proceedings of Graphics Hardware*, 33–40.

CATMULL, E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. Computer Science.

CHEN, H.-M. and WANG, W.-T. 1996. The feudal priority algorithm on hidden-surface removal. In *Proceedings of ACM SIGGRAPH 1996*, 55–64.

DX10 SDK. 2010. Microsoft Corporation. February 2010 release.

ENDERTON, E., SINTORN, E., SHIRLEY, P., and LUEBKE, D. 2010. Stochastic transparency. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, 157–164.

EVERITT, C. 2001. Interactive order-independent transparency. NVIDIA whitepaper.

FOLEY, J. D., VAN DAM, A., FEINER, S. K., and HUGHES, J. F. 1990. *Computer Graphics: Principles and Practice*, chapter 16.5.1. Addison-Wesley, 2nd edition.

FUCHS, H., KEDEM, Z. M., and NAYLOR, B. F. 1980. On visible surface generation by *a priori* tree structures. *Computer Graphics (Proceedings of ACM SIGGRAPH 1980)*, 14(3):124–133.

FUKUSHIGE, S. and SUZUKI, H. 2006. Voronoi diagram depth sorting for polygon visibility ordering. In *Proceedings of GRAPHITE '06*, 461–467.

GOAD, C. 1982. Special purpose automatic programming for hidden surface elimination. *Computer Graphics (Proceedings of ACM SIGGRAPH 1982)*, 16(3):167–178.

HUANG, M.-C., LIU, F., LIU, X.-H., and WU, E.-H. 2010. Multi-fragment effects on the GPU using bucket sort. In W. Engel, editor, *GPU Pro: Advanced Rendering Techniques*, chapter VIII.1, 495–508. A K Peters.

JANSEN, J. and BAVOIL, L. 2010. Fourier opacity mapping. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, 165–172.

JOUPPI, N. P. and CHANG, C.-F. 1999. Z^3: An economical hardware technique for high-quality antialiasing and transparency. In *Proceedings of Graphics Hardware*, 85–93.

KARP, R. M. 1972. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Proceedings of Symposium on the Complexity of Computer Computations*, 85–103.

KIM, T.-Y. and NEUMANN, U. 2001. Opacity shadow maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques*, 177–182.

LAINE, S. and KARRAS, T. 2011. Stratified sampling for stochastic transparency. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2011)*, 30(4).

LIU, B.-Q., WEI, L.-Y., and XU, Y.-Q. 2006. Multi-layer depth peeling via fragment sort. Technical Report MSR-TR-2006-81, Microsoft Research Asia.

LIU, F., HUANG, M.-C., LIU, X.-H., and WU, E.-H. 2009. Efficient depth peeling via bucket sort. In *Proceedings of the ACM Symposium on High Performance Graphics*, 51–57.

LIU, F., HUANG, M.-C., LIU, X.-H., and WU, E.-H. 2010. FreePipe: A programmable parallel rendering architecture for efficient multi-fragment effects. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, 75–82.

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

MARK, W. R. and PROUDFOOT, K. 2001. The F-buffer: A rasterization-order FIFO buffer for multi-pass rendering. In *Proceedings of Graphics Hardware*, 57–64.

MESHKIN, H. 2007. Sort-independent alpha blending. GDC Talk.

MULDER, J. D., GROEN, F. C. A., and VAN WIJK, J. J. 1998. Pixel masks for screen-door transparency. In *Proceedings of Visualization '98*, 351–358.

MYERS, K. and BAVOIL, L. 2007. Stencil routed A-buffer. ACM SIGGRAPH Technical Sketch.

MYERS, K. and BAVOIL, L. 2007. Stencil routed *k*-buffer. NVIDIA whitepaper.

NEHAB, D., BARCZAK, J., and SANDER, P. V. 2006. Triangle order optimization for graphics hardware computation culling. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, 207–211.

NEWELL, M. E., NEWELL, R. G., and SANCHA, T. L. 1972. A new approach to the shaded picture problem. In *Proceedings of the ACM National Conference*.

PATERSON, M. S. and YAO, F. F. 1989. Binary partitions with applications to hidden surface removal and solid modelling. In *Symposium on Computational Geometry*, 23–32.

PATNEY, A., TZENG, S., and OWENS, J. D. 2010. Fragment-parallel composite and filter. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2010)*, 29(4): 1251–1258.

PORTER, T. and DUFF, T. 1984. Compositing digital images. *Computer Graphics (Proceedings of ACM SIGGRAPH 1984)*, 18(3): 253–259.

SALVI, M., MONTGOMERY, J., and LEFOHN, A. 2011. Adaptive transparency. In *Proceedings of the ACM Symposium on High Performance Graphics*, 119–126.

SANDER, P. V., NEHAB, D., and BARCZAK, J. 2007. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2007)*, 26 (3):89.

SCHUMACKER, R. A., BRAND, B., GILLILAND, M. G., and SHARP, W. H. 1969. Study for applying computer-generated images to visual simulation. Technical Report AFHRL-TR-69-14, US Airforce Human Resources Laboratory.

SINTORN, E. and ASSARSSON, U. 2008. Real-time approximate sorting for self shadowing and transparency in hair rendering. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, 157–162.

SINTORN, E. and ASSARSSON, U. 2009. Hair self shadowing and transparency depth ordering using occupancy maps. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, 67–74.

SKIENA, S. S. 2008. *The Algorithm Design Manual*, chapter 15.2. Springer, 2$^{\text{nd}}$ edition.

THIBIEROZ, N. 2008. Robust order-independent transparency via reverse depth peeling in DirectX 10. In W. Engel, editor, *ShaderX6: Advanced Rendering Techniques*, chapter 3.7, 211–226. Charles River Media.

WEXLER, D., GRITZ, L., ENDERTON, E., and RICE, J. 2005. GPU-accelerated high-quality hidden surface removal. In *Proceedings of Graphics Hardware*, 7–14.

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

WITTENBRINK, C. M. 2001. R-buffer: a pointerless A-buffer hardware architecture. In *Proceedings of Graphics Hardware*, 73–80.

YANG, J. C., HENSLEY, J., GRÜN, H., and THIBIEROZ, N. 2010. Real-time concurrent linked list construction on the GPU. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2010)*, 29(4):1297–1304.

YUKSEL, C. and KEYSER, J. 2008. Deep opacity maps. *Computer Graphics Forum (Proceedings of Eurographics 2008)*, 27(2):675–680.