In-Depth Buffers

SONGFANG HAN, Hong Kong UST
GE CHEN, Hong Kong UST
DIEGO NEHAB, IMPA
PEDRO V. SANDER, Hong Kong UST

We present a new approach for rendering all triangles in a model in front-to-back order without the need for sorting at runtime. The method can be used for rendering order-dependent transparency effects, or to minimize overdraw, for example. The key distinguishing component in the approach is its negligible runtime cost and therefore the ease with which it can be incorporated into rendering engines. More specifically, given a viewpoint, the runtime simply selects a presorted triangle list, which we call *in-depth buffers*, to be rendered at full speed. These in-depth buffers are even optimized for post-transform vertex cache efficiency. The result is unmatched in front-to-back rendering performance. The difficulty is computing the smallest set of in-depth buffers required. This reduces to a graph problem that we prove to be NP-hard. Nevertheless, we have found an optimization heuristic that produces good results, particularly when visually imperceptible fragment ordering mistakes are allowed.

CCS Concepts: • **Computing methodologies** → **Rasterization**; **Visibility**;

Additional Key Words and Phrases: Efficient rendering, visibility, translucent rendering, vertex caching

ACM Reference Format:
Songfang Han, Ge Chen, Diego Nehab, and Pedro V. Sander. 2018. In-Depth Buffers. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 1, Article 2 (May 2018), 14 pages. https://doi.org/10.1145/3203194

1 INTRODUCTION AND RELATED WORK

One of the key features of the Z-buffer algorithm [Catmull 1974] is that it guarantees correct results when rendering opaque geometry regardless of the order in which the fragments are generated. Nevertheless, for performance reasons, it is still preferable to generate fragments front-to-back, since this minimizes the number of fragments that are shaded unnecessarily and then overwritten by fragments closer to the camera. In contrast, semi-transparent fragments must be composited over each other in depth order for correct results [Porter and Duff 1984]. Real-time depth-sorting is therefore a well studied problem in computer graphics, and several different approaches to the problem have been developed over the years.

Inspired by the A-buffer [Carpenter 1984], several modifications of the rendering pipeline have been proposed that collect and sort fragments before blending [Aila et al. 2003; Jouppi and Chang 1999; Liu et al. 2010; Mark and Proudfoot 2001; Wittenbrink 2001]. More recently, programmable graphics hardware has been used successfully to implement similar ideas in software [Knowles et al. 2012, 2014; Lefebvre et al. 2014; Lipowski 2010; Maule et al. 2014; Patney et al. 2010; Peeper 2008; Vasilakis and Fudos 2012; Yang et al. 2010].

Authors' addresses: Songfang Han, Hong Kong UST, shanaf@connect.ust.hk; Ge Chen, Hong Kong UST, gechen@connect.ust.hk; Diego Nehab, IMPA, diego@impa.br; Pedro V. Sander, Hong Kong UST, psander@cse.ust.hk.

2577-6193/2018/5-ART2 $15.00
https://doi.org/10.1145/3203194

One problem with these strategies is that they require an unbounded amount of memory. Early *screen-door transparency* techniques, though approximate, did not suffer from this problem [Foley et al. 1990; Mulder et al. 1998]. Newer algorithms perform even better on a limited memory budget [Enderton et al. 2010; Laine and Karras 2011; Maule et al. 2013; Salvi et al. 2011]. Some of these methods are particularly well suited for scenes with high depth-complexities [Callahan et al. 2005; Jansen and Bavoil 2010; Kim and Neumann 2001; Sintorn and Assarsson 2008, 2009].

Another alternative is to use the Z-buffer to progressively peel depth layers in order [Everitt 2001; Mammen 1989; Thibieroz 2008]. A variety of techniques have been developed that reduce the number of rendering passes required to peel all layers [Bavoil et al. 2007; Bavoil and Myers 2008; Carr et al. 2008; Huang et al. 2010; Liu et al. 2006, 2009; Wexler et al. 2005].

Our work belongs to yet a different category that attempts to precompute the depth order [Goad 1982; Govindaraju et al. 2004; Newell et al. 1972; Schumacker et al. 1969]. In particular, we attempt to all but eliminate runtime involvement of the CPU and GPU in depth-sorting [Chen et al. 2012; Han and Sander 2016; Nehab et al. 2006; Sander et al. 2007]. The key observation in this last set of techniques is that a single triangle list can be effectively depth-sorted when seen from an entire range of viewpoints.

Chen et al. [2012] start by dividing the sphere of viewpoints into a set of compact regions. Then, given a static mesh, they produce, for each view region, a single triangle list. The list merges all triangle orders needed to render the mesh in depth-sorted order from any viewpoint in that region. The trick is to include more than one instance of selected triangles in different positions in the list, each associated to a plane test. At runtime, the viewpoint is tested against the plane and the result is used to decide whether to output each triangle instance. This way, each triangle is rendered exactly once, and the resulting triangle list is depth-sorted correctly.

In our work, we do away with the repeated triangles and plane tests. Instead, we try to identify the regions on the sphere that can use exactly the same triangle list. We call such triangle lists *in-depth buffers*. The runtime simply selects the in-depth buffer appropriate for the current viewpoint and renders it at full speed. This simplifies the task of integrating our technique into existing rendering engines. Moreover, it reduces the amount of memory needed to store the buffers, as there is no duplication and no planes to test against.

Identifying the partition of the viewpoint sphere that results in the smallest number of in-depth buffers leads to an interesting graph problem that we prove to be NP-hard. On the other hand, once setup, this graph problem is purely combinatorial, whereas the optimization problem formulated by Chen et al. [2012] is geometric and therefore harder to deal with robustly. We propose a heuristic that results in an small number of in-depth buffers for a variety of typical 3D models, particularly when imperceptible fragment ordering errors are allowed. Finally, we propose a constrained vertex locality optimization that reorders the triangles in each buffer to maximize the effectiveness of the post-transform vertex cache.

In summary, we propose a high-quality approximate depth-sorting algorithm for static meshes that incurs essentially no runtime cost. Since we provide source-code for the preprocessing stage, and since there is no runtime component, we believe our method will be immediately useful in a variety of applications.

2 VIEW-DEPENDENT PARTIAL ORDERS

We start by partitioning the view space into a set V of viewpoints, view triangles, or view regions which are representative of regions from where the model can be viewed. We then associate a partial order graph to each $v \in V$ that represents the occlusion relationships between all triangles in the mesh when viewed from v. These partial order graphs will be the input to the clustering algorithm described in section 3. We will next elaborate on each of these steps.

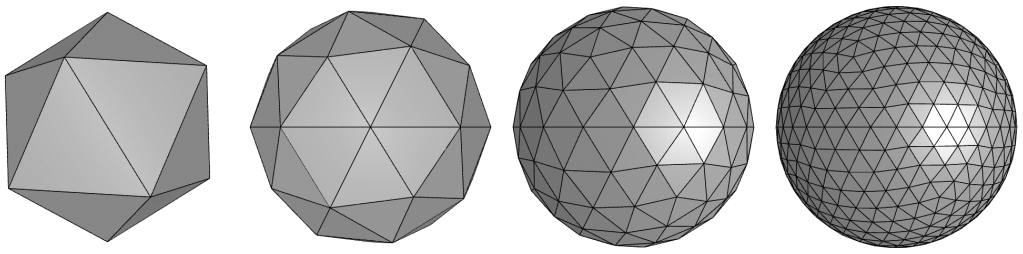

Fig. 1. A icosahedron with zero to three levels of subdivision. The resulting vertices can be used as a set of uniform viewpoints V surrounding the model. The number of subdivisions provides a tradeoff between quality of the orders and processing time and memory.

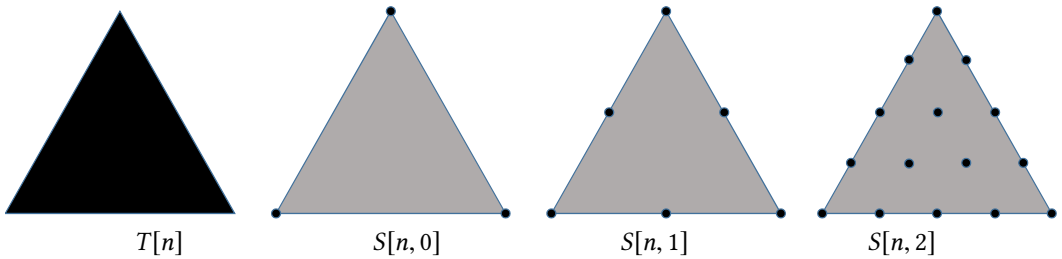

Fig. 2. View triangles $T[n]$ consider the continuous space of viewpoints within each triangle of the subdivided icosahedron. For sampled view triangles $S[n, m]$, we perform m additional levels of sudivision and use the resulting vertices as the viewpoints.

2.1 Partitioning the view space

Discrete viewpoints. One alternative is to use the *vertices* of a subdivided icosahedron as the set V. Figure 1 shows the subdivided icosahedra $P[n]$, where n denotes the number of midpoint subdivisions used. When n is small, this approach is more appropriate for overdraw reduction techniques, since they are fairly tolerant to errors in depth-sorting [Han and Sander 2016; Nehab et al. 2006]. Order-dependent transparency effects, however, require accurate results even away from the viewpoints in V in order to avoid visible rendering errors. Unfortunately, when n is large, this method becomes prohibitive in terms of memory and preprocessing time.

View triangles and view regions. We also explored using the set $T[n]$ of triangles of the subdivided icosahedron as set V. In this case, the partial order graph associated to each triangle considers the continuous space of *all* viewpoints inside the triangle. We have found that, although this virtually eliminates ordering errors, the resulting partial orders are dense enough to prevent significant sharing of in-depth buffers between different triangles. We also tried using the set of all viewpoints in 3D that project to each triangle of the subdivided icosahedron as set V. Although this eliminates rendering errors, it is even more restrictive.

Sampled view triangles. We then found a compromise between discrete viewpoints, which cause too many rendering artifacts, and the view triangles and regions, which severely restrict clustering. The idea is to sample viewpoints inside each triangle and associate to the triangle the union of all ordering relations found for the viewpoints sampled from it. Figure 2 illustrates the samples used in this approach for different subdivision levels of a sampled view triangle $S[n, m]$. Here, n denotes

the number of subdivisions used to generate the triangles, and m denotes the number of *additional subdivisions* used to define the sample points within the triangle. Note that several of the samples lie on the boundary of the view triangle and thus are shared with the neighboring view triangle, unlike with viewpoints where each $v \in V$ is independent. This approach of effectively "preclustering" with shared boundary samples avoids large *gaps* in the view space. It also significantly reduces the clustering time. Overall, it yields satisfactory results for sufficiently large values of n and m.

2.2 Generating the occlusion graph

We start from a triangle mesh consisting of non-intersecting triangles T, and a set of view partitions V, which can be viewpoints, view triangles, view regions, or sampled view triangles.

Like Chen et al. [2012], we explicitly add backfacing duplicates of each original mesh triangle to T with the goal of making the partial order sparser. Since we render with hardware backface culling enabled, only one instance of each triangle is rendered at runtime. (Naturally, triangles that are always backfacing from the relevant view partitions are removed after the clustering stage.)

For each view partition v in V, we generate a directed acyclic graph $G_v(E_v, T)$ over the set of triangles T in the following way. A directed edge (p, q) is included in E_v if and only if there exists a ray emanating from a point in view partition v that intersects triangle p before intersecting triangle q when both are front-facing. In other words, if any part of p occludes any part of q, when seen from v.

To compute the graph edges, we first determine the occlusion region $O_{i\to j}$ for each pair of triangles t_i and t_j. This occlusion region represents the partition of space where t_i occludes t_j [Chen et al. 2012]. We then include an edge from t_i to t_j in G_v, if and only if any viewpoint within view partition v intersects $O_{i\to j}$.

To render all triangles in T in front-to-back order relative to v, it suffices to render them in topological order relative to G_v. Since G_v is acyclic, any depth-first-search over G_v produces such an order.

Note that our clustering algorithm assumes that the input model does not contain any visibility cycle in each $v \in V$. If such a cycle exists, a preprocessing step can break it by splitting the offending triangles.

3 VIEW CLUSTERING

Computing and storing one in-depth buffer for each viewpoint is, naturally, impractical in terms of memory requirements. Our intuition suggests that it should be possible to render triangles in front-to-back order relative to multiple viewpoints *using the same order*. To decide if two viewpoints v and v' can share a single in-depth buffer this way, we check that the graph $G_{\{v,v'\}}(E_v \cup E_{v'}, T)$, formed by the union of edges in G_v and $G_{v'}$, is acyclic.

We can now precisely state the problem we wish to solve.

> Find the partition P of the viewpoints in V, with the smallest number $|P|$ of subsets, such that, for each subset S in P, graph $G_S(E_S, T)$ is acyclic, where $E_S = \bigcup_{v\in S} E_v$ is the union of the edges E_v associated to each viewpoint v in subset S.

For lack of a standard name in the literature, we call this the *view-clustering problem*.

3.1 View clustering is NP-hard

As is common with such combinatorial graph problems, the view-clustering problem is NP-hard. We can prove this by reducing the chromatic number problem to the view-clustering problem in the following way. Let $H(D, U)$ be an undirected graph for which we want to obtain the chromatic number $\chi(H)$. Associate one of our viewpoints to each node in $H(D, U)$, i.e., our "viewpoints" set V

equals the set of nodes U. Associate one of our triangles to each *pair* of nodes in $H(D,U)$, i.e., the "triangle" set T is the Cartesian product $U \times U$. Now, for each viewpoint v in V, associate a graph $G_v(E_v, T)$ such that E_v contains a directed edge $\big((v,w),(w,v)\big)$ from (v,w) to (w,v), seen as elements of T, if and only if D contains an undirected edge $\{v,w\}$ between nodes v and w, seen as elements of U.

We claim that, if partition P of V is a solution to the view-clustering problem on the set of graphs $G_v(E_v,T)$ we created from $H(D,U)$ as above, then $|P| = \chi(H)$. To see this, note that if there is an undirected edge $\{v,w\}$ between nodes v and w of $H(D,U)$, no subset S in P can contains both v and w. Otherwise, G_S would contain, at least, the cycle $(v,w) \to (w,v) \to (v,w)$. By the pigeonhole principle, it follows that $|P| \geq \chi(H)$. On the other hand, assume we have any coloring for $H(D,U)$, expressed as a partition Q of U, and take any subset R in Q. Since no two vertices in R are connected by an edge in D, all edges in E_v for v in R are of the form $\big((v,w),(w,v)\big)$ for w *not* in R. This means that for all u, v in R, with $u \neq v$, sets E_u and E_v reference entirely disjoint sets of vertices. There is no chance any cycle will be formed if we merge them. In other words, $|P| \leq |Q|$. Since $\chi(H)$ is an example of coloring, it follows that $|P| \leq \chi(H)$ as well. This completes the proof that $|P| = \chi(H)$.

3.2 A heuristic for view clustering

We now propose an effective heuristic for the view-clustering problem. We proceed in two stages: an initial clustering followed by a relaxation process.

The initial clustering. We start with one graph in each partition,

$$P^0 = \big\{\{v\} \;\big|\; v \in V\big\}, \tag{1}$$

and proceed iteratively. At each iteration, we try to form $P^{(k+1)}$ by merging two of the subsets $S_i^{(k)}$ and $S_j^{(k)}$ in $P^{(k)}$. That is, we set

$$P^{(k+1)} = \big\{S_i^{(k)} \cup S_j^{(k)}\big\} \cup P^k \setminus \big\{S_i^{(k)}, S_j^{(k)}\big\}, \quad i \neq j, \tag{2}$$

if the resulting graph $G_{S_i^{(k)} \cup S_j^{(k)}}\big(E_{S_i^{(k)}} \cup E_{S_j^{(k)}}, T\big)$ is acyclic.

The heuristic is in selecting the best pair of subsets to merge at each step. Recall our viewpoints are spatially distributed on the sphere. This spatial distribution equips the set of viewpoints with a neighboring relation that extends naturally to subsets of viewpoints, sets of subsets of viewpoints, etc. For example, two subsets S_i and S_j are neighbors if there exist viewpoints $u \in S_i$ and $v \in S_j$ that are spatial neighbors on the sphere. Our first heuristic decision is to only consider neighboring subsets for merging. There are two desirable consequences of this choice. First, it brings down the number of pairs considered for merging, from quadratic to linear on the number of subsets in the current partition. This reduces the computational complexity of selecting the best pair. Second, each partition defines a contiguous region on the sphere's surface. This reduces the number switches between different triangle lists when the viewpoint changes at runtime.

Our second heuristic decision relates to the order in which the subsets are merged. For that, we keep a priority queue of subset pairs. At each iteration, the highest-priority pair is removed from the queue and the graph that would be formed by merging the subsets is checked for cycles. If no cycles are found, the subsets are indeed merged. This involves updating the neighboring pairs and their priorities. If, on the other hand, merging the two subsets would create cycles, a new pair is removed from the priority queue. We repeat this process until no pairs can be merged.

We have tried several different pair priority functions, all based on the associated edge sets

$$I = E_{S_i^{(k)}} \quad \text{and} \quad J = E_{S_j^{(k)}}. \tag{3}$$

For performance reasons, we favor priority functions that can be updated incrementally when subsets are merged. These included maximizing $|I \cap J|$, minimizing $|I \cup J| - |I \cap J|$, and finally minimizing $|I \cup J|$. This last alternative gave us the best results.

The relaxation process. After the initial clustering, we proceed to the relaxation process. At each relaxation iteration, we select a random subset S from the current partition. The subset neighboring relation defines a 1-ring and a 2-ring of subsets around S. The relaxation proceeds in two stages. First, we try to move elements from the 1-ring out towards the 2-ring. Last, we try to move elements from S out towards the 1-ring. The hope is that we will be left with an empty set S, thereby reducing the number of subsets in the partition. Let us consider these stages in more detail.

Some viewpoints in the union of subsets of the 1-ring have neighboring subsets in the 2-ring. Among these, some viewpoints can be moved to one of the neighboring subsets without creating cycles. We select one of these viewpoints at random and make one of its allowed moves, also at random. This requires us to update the neighboring relations, which we do incrementally. We repeat this process until no viewpoint can be moved from the 1-ring to the 2-ring of S.

Now we look for viewpoints in S that have neighboring subsets in the 1-ring. Among these, some can be moved to one of their neighboring subsets without creating cycles. We select one of these viewpoints at random and make one of its allowed moves, also at random. This again requires us to update the neighboring relations, which we do incrementally. Finally, we repeat this process until no viewpoint can be moved from S to any subset in its 1-ring.

At some point, relaxing all subsets does not reduce the number of subsets in the partition. If this happens multiple times in sequence, we abort the process and return the results. In our experiments, we abort after 5–10 unfruitful iterations.

4 CONSTRAINED VERTEX CACHE OPTIMIZATION

Finally, we reorder the in-depth buffers of all partitions to maximize vertex locality and leverage the GPU vertex cache. To do so, we seek to minimize the *average cache miss ratio* (ACMR), which measures the ratio between processed vertices and rendered triangles. We assume a FIFO caching scheme, which is the traditional caching approach and leads to good results on modern hardware.

Optimizing an input triangle mesh to minimize the number of cache misses in is a well-studied problem in computer graphics. A number of heuristics have been proposed (e.g., [Bogomjakov and Gotsman 2002; Hoppe 1999; Lin and Yu 2006; Sander et al. 2007]). Most heuristics follow the same high-level structure:

(1) Select an (often random) unprocessed triangle on the mesh and add it to the list;
(2) Process and add nearby triangles following the proposed local heuristic until a *dead end* is reached;
(3) Repeat steps 1–2 until all triangles have been added.

This greedy approach, coupled with a carefully crafted order in which to pick triangles or vertices to process, achieves results that are close to the theoretical lower bound ACMR of 0.5 for many models. Unfortunately, our triangle lists cannot use these methods directly since reordering the mesh triangles arbitrarily would violate our partial ordering. Instead, we propose an adaptation that restricts the set of triangles that can be processed at each stage of the algorithm to ensure that the resulting order still adheres to the partial order graph. More specifically, for steps 1 and 2, we only consider triangles which have no remaining unprocessed ancestors in the partial order graph. Once a triangle is processed, it is removed from the partial order graph. Triangles that have ancestors are therefore skipped to be processed when revisited at a later point.

This general adaptation is applicable to many existing methods. We applied it to the *tipsify* approach of Sander et al. [2007] due to its simplicity, availability of source-code, and efficiency. As

evidenced in the results section, the constrained order has approximately a 0.7–0.8 ACMR, which is inferior to the original unconstrained results (0.6–0.7 ACMR), but far better than not using orders with vertex locality, which can be close to a 3.0 ACMR.

5 ORDER UPSAMPLING

To improve processing speed, we explored a strategy where we upsample our results from a simplified coarse mesh M_c solution to our target fine mesh M_f. First, M_f is simplified to generate M_c. This can be accomplished using one of many high quality mesh simplification techniques. In our implementation, we applied edge collapses using a quadric error metric [Garland and Heckbert 1997], which yields a faithful approximation. Next, we run our algorithm on M_c, resulting in a coarse solution. Then, for each triangle t_f in M_f, we find the point p in M_c that is closest to the centroid of t_f. We then assign t_f to the coarse mesh face t_c where p lies. Finally, we generate the buffers for M_f by replacing each face in the coarse solution by all faces assigned to it. We used an arbitrary order among fine mesh faces associated to each t_c. Since the upsampling factor is small (2–5×), vertex locality is still preserved and there is no noticeable decrease in cache efficiency. This can significantly reduce processing time while still achieving high quality results.

6 RUN-TIME SELECTION

At rendering time, the application must select the appropriate in-depth buffer based on the current viewpoint. We follow an approach similar to Han and Sander [2016], using a 2D lookup table IBid$[\theta, \phi]$ of sufficient resolution indexed by polar and azimuth angles of the current viewpoint rounded to the nearest valid parameter values. In the case of sampled view triangles, a view triangle ID is stored and the viewpoint is tested for containment against its one-ring neighborhood. Once the view triangle is identified, its associated in-depth buffer is used.

Although this parameterization is less uniform than the subdivided icosahedron, it is only used to store the index buffer IDs for the purpose of simplifying the lookup at runtime. Since it is a single array lookup for the entire mesh, this lookup time is negligible.

The selected buffer is then directly rendered by the application with backface culling enabled. Unlike Chen et al. [2012], which use plane tests in the shaders to further cull triangles, no additional modifications or processing are required from the client rendering application.

7 RESULTS

We evaluated the performance and quality of our results in a variety of meshes ranging from 10,000 to 115,000 triangles, which are typical resolutions for real-time rendering applications. All experiments were performed in an Intel Core i7 3.6GHz machine with 16GB RAM and an NVIDIA GeForce GTX 770 graphics card.

Figure 3 demonstrates the importance of using a sufficiently dense sampling of the view space in order to achieve accurate results. Note that as the number of sampled view triangle subdivisions progressively increases, so does the quality of the results. With $S[2, 2]$, the artifacts are imperceptible. The accompanying video demonstrates these results for a variety of settings. We also explored using viewpoints, view triangles, and view regions. In the case of viewpoints, unless we used an extremely dense sampling, there were significant artifacts away from these viewpoints. As for view triangles and view regions, the complexity of the partial order graph made it infeasible to cluster effectively. For example, for a 10,000-triangle dragon mesh, we could not achieve results with fewer than 200 clusters for either view triangles or view regions. Thus, for the majority of results, we adopted the dense $S[2, 2]$ sampled view triangle configuration.

Table 1 shows the overall results of our approach. We use the Depth-Presorted Triangle List (DPTL) method of Chen et al. [2012] as the baseline for comparison since it is the only other approach

Table 1. Final results of our approach on several models at varying input resolutions. Rows denoted by "↪" report results that are upsampled from the previous row to a higher resolution. All times are in minutes and memory in KB. Ratio of the result of Chen et al. [2012] over ours is shown in parenthesis. For the upsampled results, the ratios are relative to the their high resolution results.

input			occ gen	clustering		relaxation			overall statistics				
model	△s	views	time	time	clusters	iters	time	clusters	proc time	memory	ACMR	PSNR	IPR
bunny	10,000	S[2,2]	13.2	15.0	7	6	36.6	6	64.9(0.2×)	664(4.1×)	0.8(2.0×)	57.9	0.003
	↪ 50,000								64.9(3.5×)	3324(3.8×)	0.8(2.3×)	53.6	0.007
	30,000	S[2,2]	101.4	62.0	9	8	196.7	7	360.1(0.2×)	2280(3.4×)	0.8(2.4×)	60.1	0.003
	50,000	s[2,2]	262.8	158.4	9	8	777.3	8	1198.4(0.2×)	4202(3.0×)	0.7(2.6×)	61.1	0.002
dragon	10,000	S[2,2]	12.1	10.6	12	5	15.6	11	38.2(0.4×)	1092(2.8×)	0.8(2.8×)	55.0	0.004
	20,000	S[2,2]	36.9	36.9	13	11	177.2	11	250.9(0.2×)	2162(2.7×)	0.7(2.9×)	59.6	0.002
	↪ 40,000								251.0(0.6×)	4324(2.6×)	0.7(2.4×)	50.6	0.010
	40,000	S[2,2]	132.8	100.1	15	7	502.3	13	735.2(0.2×)	5131(2.2×)	0.7(2.4×)	59.6	0.002
feline	10,000	S[2,2]	12.5	8.6	10	5	11.3	9	32.4(0.3×)	928(3.2×)	0.8(2.1×)	55.5	0.004
	↪ 50,000								32.4(7.0×)	4642(2.8×)	0.8(2.4×)	45.5	0.018
	20,000	S[2,2]	46.9	25.7	11	5	53.4	11	125.9(0.3×)	2194(2.5×)	0.7(2.4×)	58.8	0.002
	50,000	S[2,2]	241.5	185.7	13	8	668.7	11	1095.9(0.2×)	5521(2.4×)	0.7(2.6×)	58.9	0.003
fandisk	10,000	S[2,2]	9.1	27.2	6	7	64.5	4	100.8(0.1×)	457(5.1×)	0.8(2.4×)	56.4	0.003
	↪ 50,000								100.9(1.6×)	2284(5.0×)	0.8(2.3×)	47.0	0.008
	50,000	S[2,2]	192.2	225.0	6	7	651.7	4	1068.9(0.1×)	2305(5.0×)	0.8(2.5×)	61.9	0.001

that performs runtime selection rather than sorting and has been shown to be significantly faster than the alternatives. The ratio of the result of DPTL over ours is reported for overall statistics. Next we discuss each aspect of the comparison in detail.

Processing. Statistics of all processing steps are detailed in table 1, including the time breakdown for the generation of the partial order graph, initial clustering, and relaxation. Note that a few iterations of the relaxation process can further reduce the number of clusters after the initial merging step. Overall, the processing time when using the dense $S[2, 2]$ configuration becomes significantly higher than that of DPTL (around 5–10×). However, we note that this processing only needs to be performed once for each static model. The performance deteriorates significantly as the mesh complexity increases beyond 50,000 triangles. The algorithm can still scale to complex scenes with a larger number of models since the algorithm is executed independently on each mesh. Thus, we have not explored further algorithm optimizations or a parallelized version as in DPTL. We instead explored upsampling the order from a coarse mesh to a more detailed mesh. In this case, our preprocessing time is significantly faster than DPTL, while still achieving high quality rendering results when using $S[2, 2]$.

Memory. The total memory for our approach is reported under the overall statistics. Whereas our approach requires a larger number of clusters than DPTL, it only needs to store the index data (we use 16 bits per index in our implementation). DPTL also stores one plane per triangle for the plane test, thus significantly increasing the memory requirement (2.4–5.1×).

Vertex cache efficiency. We applied our novel constrained vertex cache optimization algorithm to all in-depth buffers. We use a vertex cache size of 20 for these experiments. Note that, even with the constraints imposed by the partial order graph, the algorithm is able to considerably reduce the number of processed triangles, yielding results in the 0.70s. To provide a fair comparison, we also applied our algorithm on the DPTL buffers. However, their partial order graphs are more dense and involve more triangle duplication. As a result, their cache efficiency could not be significantly improved and their final ACMR is consistently over 2× that of ours, incurring significantly more vertex computation at runtime.

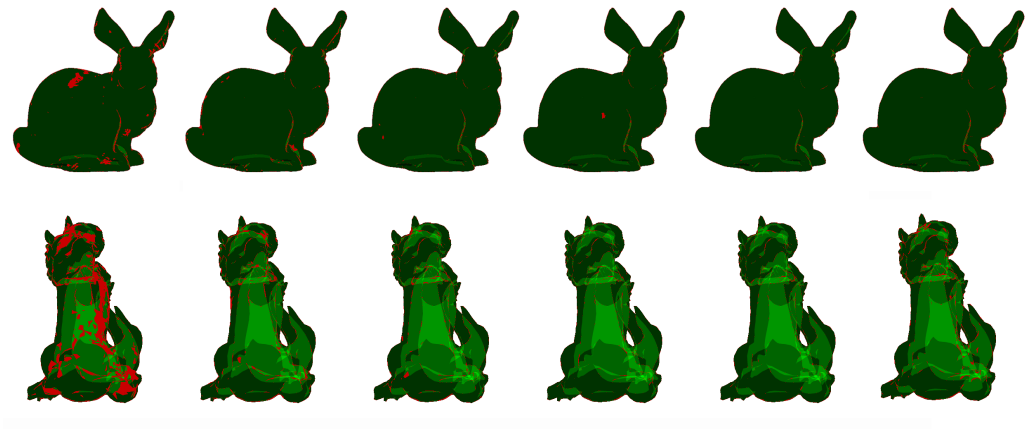

Fig. 3. Error visualization for different view configurations of the dragon [10,000] and the bunny [30,000]. $S[2,2]+$ denotes upsampled results from dragon [5,000] and bunny [10,000], respectively. Note that using more subdivisions avoids significant rendering artifacts caused by incorrect ordering (red pixels).

Quality. Unlike DPTL which can render *exact* front-to-back orders, our view space is sampled and therefore, away from these viewing locations, ordering errors can lead to rendering artifacts. The average *peak signal-to-noise ratio* (PSNR) of the foreground pixels for a set of 900 renderings from set of random viewpoints at 2–5× the radius of the model's bounding sphere is approximately 55–62db (down to approximately 45–55db for upsampled results). The average *incorrect pixel ratio* (IPR) is computed as the ratio of pixels with an incorrect order to the total number of foreground pixels and ranges between 0–0.4% for our processed results (0–2% for upsampled results). We disabled MSAA for these measurements. As discussed earlier, using densely sampled view triangles, most of the error occurs at silhouettes and becomes unnoticeable for semi-transparent rendering effects when viewed from all directions outside the model's bounding volume (figures 6 and 7). Note that the upsampling approximation can be prone to errors, especially if the coarse mesh has resolution that is too low. Table 2 shows this trend. As we increase the coarse mesh resolution, we get a higher quality fine mesh result. Once we reach a sufficiently high coarse mesh resolution that captures enough surface detail, note that we generally do no suffer any significant quality degradation as we increase the resolution of the target fine mesh.

For a more thorough set of results, please refer to the accompanying video, which shows a variety of renderings from multiple viewpoints and using different view configurations.

Rendering time. For the runtime comparison, we first render 900 instances of the model from multiple viewpoints using a complex shader effect and then average the results. Refer to table 3 for these rendering times. In this *simple scene*, we explored three combinations of shaders whose complexity vary in order to cater to vertex-bound (vs) and fill-bound (ps) applications, as well scenarios with expensive vertex and pixel shaders (both). Note that in-depth buffers do not require either sorting or selection in the shaders as in DPTL. So, once the buffer is determined, it is simply rendered directly. Furthermore, as discussed above, in-depth buffers have much better vertex locality, which is particularly helpful for the vs scenario. As a result, we observed a speedup of 1.3–2.7× over DPTL (we use their PS rendering mode, which is the fastest and also suitable for vertex caching).

Table 2. Upsampling results for different coarse and fine mesh resolutions. Consecutive rows with matching resolutions are shown in bold to facilitate comparisons.

input					overall statistics		
model		Δs		views	ACMR	PSNR	IPR
bunny	10,000	→	**50,000**	S[2,2]	0.8	53.6	0.007
	30,000	→	**50,000**	S[2,2]	0.8	58.5	0.003
	30,000	→	70,000	S[2,2]	0.8	58.3	0.004
dragon	10,000	→	**40,000**	S[2,2]	0.8	45.0	0.020
	20,000	→	**40,000**	S[2,2]	0.7	50.6	0.010
	20,000	→	70,000	S[2,2]	0.8	51.0	0.008
	20,000	→	80,000	S[2,2]	0.8	50.9	0.008
	20,000	→	115,000	S[2,2]	0.8	50.8	0.009
feline	10,000	→	**50,000**	S[2,2]	0.8	45.5	0.018
	20,000	→	**50,000**	S[2,2]	0.8	49.7	0.009
	20,000	→	70,000	S[2,2]	0.8	48.6	0.010
	20,000	→	80,000	S[2,2]	0.8	47.9	0.011
	20,000	→	100,000	S[2,2]	0.8	48.6	0.011
fandisk	10,000	→	**50,000**	S[2,2]	0.8	46.9	0.008
	30,000	→	**50,000**	S[2,2]	0.8	52.1	0.003
	30,000	→	70,000	S[2,2]	0.8	50.7	0.005
	30,000	→	90,000	S[2,2]	0.8	50.6	0.005
	30,000	→	110,000	S[2,2]	0.8	50.7	0.005

Table 3. Rendering time results for the models from table 1. All times are in miliseconds. Ratio of the result of Chen et al. [2012] over ours is shown in parenthesis.

input			simple scene			complex scene
model	Δs	views	vs	ps	both	simple
bunny	10,000	S[2,2]	1.2(1.9×)	3.5(1.3×)	3.7(1.3×)	4.8(1.4×)
	↪ 50,000		5.4(2.2×)	4.9(1.7×)	6.4(1.9×)	8.2(1.7×)
	30,000	S[2,2]	3.3(2.2×)	4.4(1.5×)	4.9(1.6×)	6.6(1.5×)
	50,000	S[2,2]	5.1(2.3×)	5.0(1.7×)	6.2(2.0×)	7.6(1.7×)
dragon	10,000	S[2,2]	1.1(2.5×)	3.8(1.5×)	4.0(1.5×)	5.2(1.4×)
	20,000	S[2,2]	1.9(2.6×)	4.3(1.6×)	4.7(1.6×)	5.4(1.5×)
	↪ 40,000		3.7(2.7×)	5.0(1.7×)	5.7(2.0×)	7.1(1.8×)
	40,000	S[2,2]	3.9(2.6×)	5.1(1.7×)	5.8(1.9×)	7.0(1.7×)
feline	10,000	S[2,2]	1.1(2.2×)	3.2(1.4×)	3.4(1.4×)	4.9(1.3×)
	↪ 50,000		4.8(2.4×)	4.5(1.7×)	5.7(2.2×)	7.1(1.8×)
	20,000	S[2,2]	2.0(2.4×)	3.6(1.5×)	3.9(1.6×)	5.3(1.6×)
	50,000	S[2,2]	4.7(2.5×)	4.5(1.7×)	5.7(2.2×)	7.2(1.9×)
fandisk	10,000	S[2,2]	1.2(2.0×)	3.2(1.3×)	3.4(1.3×)	5.0(1.3×)
	↪ 50,000		5.6(2.0×)	4.6(1.7×)	6.6(1.7×)	7.6(1.6×)
	50,000	S[2,2]	5.6(2.0×)	4.6(1.7×)	6.5(1.8×)	7.8(1.5×)

Rendering order discrepancies. Figure 4 shows three instances of rendering order errors. The top row uses a semi-transparent per-pixel lighting shader, whereas the bottom row uses a simpler and smoother depth-to-color shader where these artifacts are more clearly visible. The artifacts on the dragon are due to the coarse view configuration (S[2,0]), the feline uses an aggressive upsampling ratio, and the bunny shows an example of a small pixel discrepancy near an internal silhouette. As mentioned earlier, in practical scenarios, when using a S[2,2] view configuration, such artifacts are difficult to notice.

Complex scene. Finally, we tested our approach using the same complex room scene from DPTL. The scene consists of a physical simulation with multiple semi-transparent dragons interacting and colliding with one another. The right column of table 3 shows a direct rendering time comparison to DPTL. Our approach achieves a speedup of 1.3–1.9× with no visible difference in the rendering results. Figure 5 shows a rendered frame. For a full demonstration with a difference comparison to the exact method of DPTL, please refer to the accompanying video.

8 CONCLUSION

In this paper, we presented a new approach for efficient front-to-back (or back-to-front) rendering of static triangle meshes. The approach is useful to a wide range of real-time rendering applications as it can be trivially incorporated to any application that requires depth-sorted triangles: Given the viewpoint, a selected pre-sorted in-depth buffer is rendered. We prove that the problem of minimizing the total number of required in-depth buffers is NP-hard and describe a heuristic that achieves good results. We also show how to further optimize for vertex locality while respecting the constraints imposed by the front-to-back ordering. In the future, we would like to explore extensions to support models with bounded distortion or articulated rigid parts.

SOURCE CODE

https://github.com/hansongfang/idb

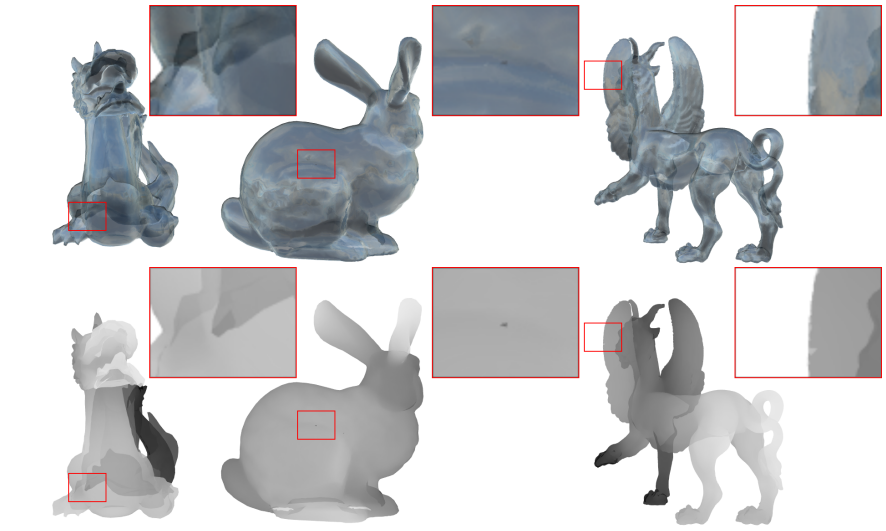

Fig. 4. Rendering order discrepancies due to a small number of viewpoints (left), a pixel inconsistency near an internal silhouette (middle), and aggressive upsampling (right). Renderings use a semi-transparent per-pixel lighting effect (top) and depth-to-color (bottom).

Fig. 5. A rendered frame from the complex room scene simulation.

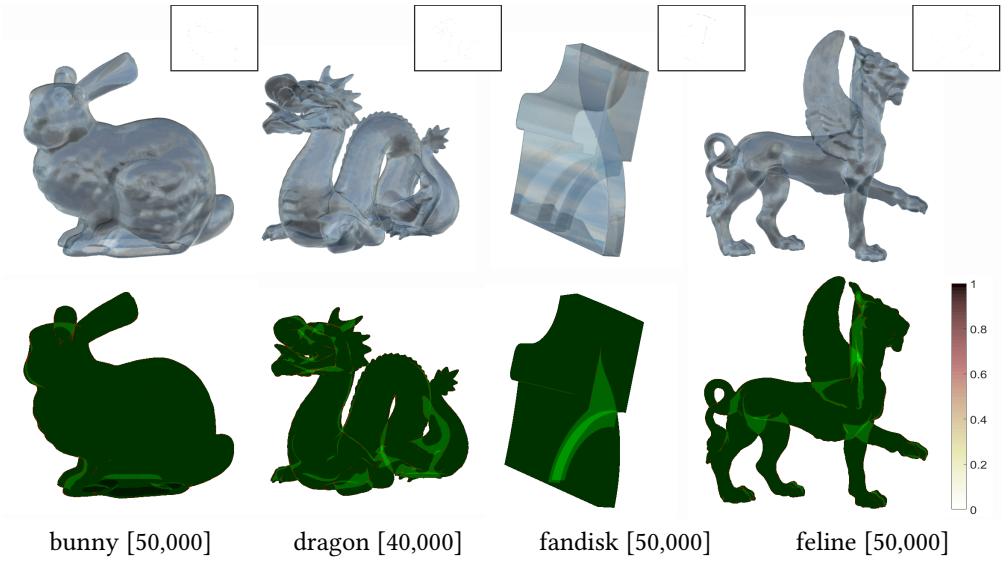

Fig. 6. Error visualization results of our approach for a variety of models using the $S[2, 2]$ view configuration (top row). Renderings using a semi-transparent per-pixel lighting effect are shown below. Difference images to ground truth are shown on top-right of each result.

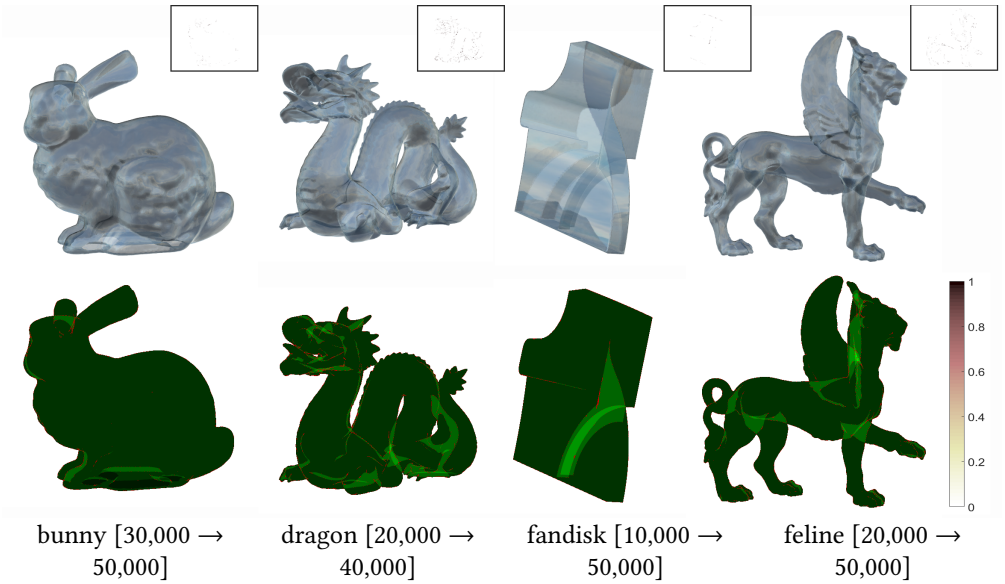

Fig. 7. Error visualization of the upsampling results using the $S[2, 2]$ view configuration (top row). Renderings using a semi-transparent per-pixel lighting effect are shown below. Difference images to ground truth are shown on top-right of each result.

ACKNOWLEDGMENTS

This work was partly supported by Hong Kong grants GRF16208814 and HKUST-DAG06/07.EG07, and by research grants from CNPq (PQ) and FAPERJ (JCNE).

REFERENCES

T. Aila, V. Miettinen, and P. Nordlund. 2003. Delay streams for graphics hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)* 22, 3 (2003), 792–800. https://doi.org/10.1145/882262.882347

L. Bavoil, S. P. Callahan, A. Lefohn, J. L. D. Comba, and C. T. Silva. 2007. Multi-fragment effects on the GPU using the *k*-buffer. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 94–104. https://doi.org/10.1145/1230100.1230117

L. Bavoil and K. Myers. 2008. Order Independent Transparency with Dual Depth Peeling. NVIDIA whitepaper.

Alexander Bogomjakov and Craig Gotsman. 2002. Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes. *Computer Graphics Forum* 21, 2 (2002), 137–148. https://doi.org/10.1111/1467-8659.00573

S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva. 2005. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics* 11, 3 (2005), 285–295. https://doi.org/10.1109/TVCG.2005.46

L. Carpenter. 1984. The A-buffer, an Antialiased Hidden Surface Method. *Computer Graphics (Proceedings of ACM SIGGRAPH 1984)* 18, 3 (1984), 103–108. https://doi.org/10.1145/964965.808585

N. Carr, R. Měch, and G. Miller. 2008. Coherent layer peeling for transparent high-depth-complexity scenes. In *Proceedings of Graphics Hardware*. 33–40. https://doi.org/10.2312/EGGH/EGGH08/033-040

E. Catmull. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.D. Dissertation. Dept. Computer Science.

Ge Chen, Pedro V. Sander, Diego Nehab, Lei Yang, and Liang Hu. 2012. Depth-presorted triangle lists. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2012)* 31, 6 (2012), 160. https://doi.org/10.1145/2366145.2366179

E. Enderton, E. Sintorn, P. Shirley, and D. Luebke. 2010. Stochastic transparency. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 157–164. https://doi.org/10.1145/1730804.1730830

C. Everitt. 2001. Interactive Order-Independent Transparency. NVIDIA whitepaper.

J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. 1990. *Computer Graphics: Principles and Practice* (2nd ed.). Addison-Wesley, Chapter 16.5.1.

Michael Garland and Paul S. Heckbert. 1997. Surface simplification using quadric error metrics. In *Proceedings of ACM SIGGRAPH 97*. 209–216.

C. Goad. 1982. Special purpose automatic programming for hidden surface elimination. *Computer Graphics (Proceedings of ACM SIGGRAPH 1982)* 16, 3 (1982), 167–178. https://doi.org/10.1145/965145.801277

N. K. Govindaraju, M. C. Lin, and D. Manocha. 2004. *Vis-Sort: Fast Visibility Ordering of 3-D Geometric Primitives*. Technical Report. UNC.

Songfang Han and Pedro V. Sander. 2016. Triangle Reordering for Reduced Overdraw in Animated Scenes. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 23–27. https://doi.org/10.1145/2856400.2856408

H. Hoppe. 1999. Optimization of mesh locality for transparent vertex caching. In *Proceedings of ACM SIGGRAPH 99*. 269–276. https://doi.org/10.1145/311535.311565

M.-C. Huang, F. Liu, X.-H. Liu, and E.-H. Wu. 2010. Multi-Fragment Effects on the GPU Using Bucket Sort. In *GPU Pro: Advanced Rendering Techniques*, W. Engel (Ed.). A K Peters, Chapter VIII.1, 495–508.

J. Jansen and L. Bavoil. 2010. Fourier opacity mapping. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 165–172. https://doi.org/10.1145/1730804.1730831

N. P. Jouppi and C.-F. Chang. 1999. Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Transparency. In *Proceedings of Graphics Hardware*. 85–93. https://doi.org/10.1145/311534.311582

T.-Y. Kim and U. Neumann. 2001. Opacity Shadow Maps. In *Proceedings of the Eurographics Workshop on Rendering*. 177–182. https://doi.org/10.2312/EGWR/EGWR01/177-182

Pyarelal Knowles, Geoff Leach, and Fabio Zambetta. 2012. Efficient Layered Fragment Buffer Techniques. In *OpenGL Insights*. A K Peters/CRC Press, 279–292. https://doi.org/10.1201/b12288-24

Pyarelal Knowles, Geoff Leach, and Fabio Zambetta. 2014. Fast sorting for exact OIT of complex scenes. *The Visual Computer* 30, 6 (may 2014), 603–613. https://doi.org/10.1007/s00371-014-0956-z

S. Laine and T. Karras. 2011. Stratified Sampling for Stochastic Transparency. *Computere Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2011)* 30, 4 (2011). https://doi.org/10.1111/j.1467-8659.2011.01978.x

Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. 2014. Per-Pixel Lists for Single Pass A-Buffer. In *GPU Pro^5: Advanced Rendering Techniques*, W. Engel (Ed.). A K Peters, Chapter I.1.

G. Lin and T. P.-Y. Yu. 2006. An Improved Vertex Caching Scheme for 3D Mesh Rendering. *IEEE Transactions on Visualization and Computer Graphics* 12, 4 (2006), 640–648. https://doi.org/10.1109/TVCG.2006.59

Jarosław Konrad Lipowski. 2010. Multi-layered Framebuffer Condensation: The l-buffer Concept. In *Computer Vision and Graphics*. Springer Berlin Heidelberg, 89–97. https://doi.org/10.1007/978-3-642-15907-7_12

B.-Q. Liu, L.-Y. Wei, and Y.-Q. Xu. 2006. *Multi-layer depth peeling via fragment sort.* Technical Report MSR-TR-2006-81. Microsoft Research Asia.

F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu. 2009. Efficient depth peeling via bucket sort. In *High Performance Graphics*. 51–57. https://doi.org/10.1145/1572769.1572779

F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu. 2010. FreePipe: A programmable parallel rendering architecture for efficient multi-fragment effects. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 75–82. https://doi.org/10.1145/1730804.1730817

A. Mammen. 1989. Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. *IEEE Computer Graphics and Applications* 9, 4 (1989), 43–55. https://doi.org/10.1109/38.31463

W. R. Mark and K. Proudfoot. 2001. The F-buffer: A rasterization-order FIFO buffer for multi-pass rendering. In *Proceedings of Graphics Hardware*. 57–64. https://doi.org/10.1145/383507.383527

Marilena Maule, João Comba, Rafael Torchelsen, and Rui Bastos. 2013. Hybrid transparency. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. https://doi.org/10.1145/2448196.2448212

M. Maule, J. L. D. Comba, R. Torchelsen, and R. Bastos. 2014. Memory-optimized order-independent transparency with dynamic fragment buffer. *Computers & Graphics* 38 (2014). https://doi.org/10.1016/j.cag.2013.07.006

J. D. Mulder, F. C. A. Groen, and J. J. van Wijk. 1998. Pixel masks for screen-door transparency. In *IEEE Visualization*. 351–358. https://doi.org/10.1109/VISUAL.1998.745323

D. Nehab, J. Barczak, and P. V. Sander. 2006. Triangle Order Optimization for Graphics Hardware Computation Culling. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 207–211. https://doi.org/10.1145/1111411.1111448

M. E. Newell, R. G. Newell, and T. L. Sancha. 1972. A solution to the hidden surface problem. In *Proceedings of the ACM annual conference*. 443–450. https://doi.org/10.1145/800193.569954

A. Patney, S. Tzeng, and J. D. Owens. 2010. Fragment-Parallel Composite and Filter. *Computere Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2010)* 29, 4 (2010), 1251–1258. https://doi.org/10.1111/j.1467-8659.2010.01720.x

C. Peeper. 2008. Prefix sum pass to linearize A-buffer storage. US Patent Application 11/766,091.

T. Porter and T. Duff. 1984. Compositing Digital Images. *Computer Graphics (Proceedings of ACM SIGGRAPH 1984)* 18, 3 (1984), 253–259. https://doi.org/10.1145/964965.808606

M. Salvi, J. Montgomery, and A. Lefohn. 2011. Adaptive transparency. In *High Performance Graphics*. 119–126. https://doi.org/10.1145/2018323.2018342

P. V. Sander, D. Nehab, and J. Barczak. 2007. Fast Triangle Reordering for Vertex Locality and Reduced Overdraw. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2007)* 26, 3 (2007), 89. https://doi.org/10.1145/1276377.1276489

R. A. Schumacker, B. Brand, M. G. Gilliland, and W. H. Sharp. 1969. *Study for Applying Computer-Generated Images to visual Simulation.* Technical Report AFHRL-TR-69-14. US Airforce Human Resources Laboratory.

E. Sintorn and U. Assarsson. 2008. Real-time approximate sorting for self shadowing and transparency in hair rendering. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 157–162. https://doi.org/10.1145/1342250.1342275

E. Sintorn and U. Assarsson. 2009. Hair self shadowing and transparency depth ordering using occupancy maps. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 67–74. https://doi.org/10.1145/1507149.1507160

N. Thibieroz. 2008. Robust Order-Independent Transparency via Reverse Depth Peeling in DirectX 10. In *ShaderX6: Advanced Rendering Techniques*, W. Engel (Ed.). Charles River Media, Chapter 3.7, 211–226.

Andreas A. Vasilakis and Ioannis Fudos. 2012. S-buffer: Sparsity-aware Multi-fragment Rendering. In *Eurographics Short Papers*, Carlos Andujar and Enrico Puppo (Eds.). https://doi.org/10.2312/conf/EG2012/short/101-104

D. Wexler, L. Gritz, E. Enderton, and J. Rice. 2005. GPU-accelerated high-quality hidden surface removal. In *Proceedings of Graphics Hardware*. 7–14. https://doi.org/10.2312/EGGH/EGGH05/007-014

C. M. Wittenbrink. 2001. R-buffer: a pointerless A-buffer hardware architecture. In *Proceedings of Graphics Hardware*. 73–80. https://doi.org/10.1145/383507.383529

J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. 2010. Real-Time Concurrent Linked List Construction on the GPU. *Computere Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2010)* 29, 4 (2010), 1297–1304. https://doi.org/10.1111/j.1467-8659.2010.01725.x