Memory-Efficient Single-Pass GPU Rendering of Multi-fragment Effects

Wencheng Wang*, and Guofu Xie*

Abstract—Rendering multi-fragment effects using GPUs is attractive for high speed. However, the efficiency is seriously compromised, because ordering fragments on GPUs is not easy and the GPU's memory may not be large enough to store the whole scene geometry. Hitherto, existing methods have been unsuitable for large models or have required many passes for data transmission from CPU to GPU, resulting in a bottleneck for speedup. This paper presents a stream method for accurate rendering of multi-fragment effects. It decomposes the model into parts and manages these in an efficient manner, guaranteeing that the parts can easily be ordered with respect to any viewpoint, and that each part can be rendered correctly on the GPU. Thus, we can transmit the model data part by part, and once a part has been loaded onto the GPU we immediately render it and composite its result with the results of the processed parts. In this way, we need only a single pass for data access with a very low bounded memory requirement. Moreover, we treat parts in packs for further acceleration. Results show that our method is much faster than existing methods, and can easily handle large models of any size.

Index Terms—Multi-fragment effects, depth ordering, fixed amount of memory, large models, accurate rendering.

1 Introduction

Rendering multi-fragment effects is important in computer graphics and is required in many areas such as rendering transparent and semi-transparent objects, anti-aliasing, and volume rendering. To utilize the great computational power of graphics processing units (GPUs), much research has focused on rendering multi-fragment effects on a GPU. Here, the main problem addressed is sorting the fragments efficiently by depth, a key requirement for many multi-fragment effects, because ordering fragments on GPUs is not easy and the GPU's memory may not be large enough to store the whole scene geometry. With regard to this, many methods such as depth peeling (DP) [1], dual depth peeling (DDP) [2], k-buffer [3], and bucket depth peeling (BDP) [4] attempt to sort a few fragments at a time while excluding the other fragments. As such, the model data are read many times for depth ordering of all fragments. As the bandwidth for transmitting data from the CPU to GPU is limited, data transmission has become a bottleneck in rendering multi-fragment effects using GPUs. To reduce the cost of data transmission, some methods sort fragments on the GPU using an allocated buffer, such as the Freepipe architecture [5] in CUDA. If the buffer is large enough, these methods require only a single pass over the data. However, owing to the limited memory on the GPU, large models cannot be dealt with efficiently, especially models with high depth complexity. Wexler et al. [6] suggested to partition the model into smaller data sets by the depth and so reduce the cost on data transmission. But it remains unsolved how to efficiently partition geometry into sets that are bounded in depth complexity to ensure that the model can be rendered using a fixed amount of memory.

This paper presents a new method for rendering multi-fragment effects on a GPU, addressing the challenge of accurately rendering large models in a fixed amount of video memory. The proposed method is memory-efficient and requires only a single pass over the data in a stream, thereby achieving considerable speedup, especially for models with high depth complexity. Moreover, it is very efficient in dealing with large models, even out-of-core models that are very difficult to handle using existing methods. Unlike some methods that sacrifice correct depth ordering for speedup, e.g., BDP [4], our method orders all fragments correctly for high quality rendering. The key idea is to decompose the model into parts, each of which has very few depth layers for any viewing direction, thereby guaranteeing the correct rendering of a part on the GPU, and to manage the parts using grids for efficiently ordering of the parts with respect to any a viewpoint. During rendering, we use grid cells to order the parts quickly by the occlusions between them according to the viewpoint, thus obtaining the order in which to transmit the parts one by one from the CPU to the GPU in a stream. Thus, their transmission order is guaranteed in accordance with their depth ordering. When a part is loaded onto the GPU, its fragments can be correctly ordered and rendered on the GPU using existing techniques, and the rendered results can be composited immediately with those of processed parts without any problems. In this way, the storage

- * *Wencheng Wang and Guofu Xie are joint first authors and are sorted by the alphabetic order of their last names.*
- *W. Wang and G. Xie are with State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China. E-mail: whn@ios.ac.cn, guofu@ios.ac.cn*
- *G. Xie is also with Graduate University of Chinese Academy of Sciences and University of Chinese Academy of Sciences, Beijing 100049, China.*

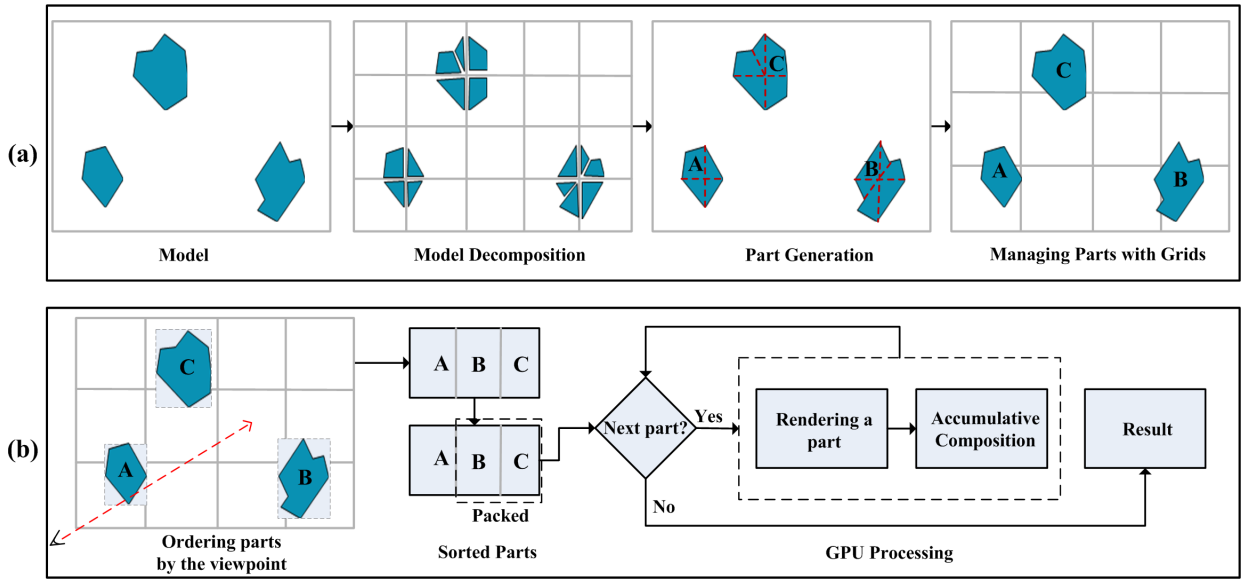


Fig. 1. Overview of our method: In preprocess (a), model decomposition is by clustering the facets in each grid cell respectively to form their related convex polyhedrons, then parts are generated by combining nearby convex polyhedrons, and at last a new grid is constructed with its cells each recording which parts are in it. For a rendering (b), the parts are first ordered by the viewpoint in a sequence, and the nearby parts causing no ordering errors in the sequence are further packed, so that the parts are transmitted one by one to GPU for rendering.

requirement can be reduced considerably, without dependence on the depth complexity, and the data can be transmitted in a single pass, alleviating the bottleneck problem of transmitting data from the CPU to the GPU. To further reduce the times of data transmission and flushing the rendering pipeline for result composition, we combine the parts that cause no ordering errors in a rendering in a pack to treat simultaneously. Results show that our method is much faster than existing methods, and using a common PC it is able to render very large models interactively, which has always been difficult for existing methods. Fig. 1 gives a conceptual overview of our method.

In the remainder of this paper, we first discuss related work in Section 2, and then present our techniques for model decomposition in Section 3. Management of the decomposed parts using grids is discussed in Section 4, followed by our rendering techniques in Section 5. Thereafter, experimental results are given and discussed in Section 6, and our conclusion is presented in Section 7.

2 Related Work

Rendering multi-fragment effects requires operations on multiple fragments at the same pixel location. Here, depth ordering of the fragments is a decisive factor for many effects such as translucency. For this, the popular Z-buffer technique keeps only the nearest (or the furthest) fragment at every pixel [7], while the A-buffer technique [8] tries to maintain an unbounded, sorted list of fragments per pixel. Many methods have extended the A-buffer method for execution on hardware, including the R-buffer method using a first-in-first-out strategy [9], as well as the F-buffer [10], stencil-routed A-buffer [11], Z^3 algorithm [12], and k-buffer methods [3], [13]. However, in these methods, the size of the per-pixel array is fixed, which prevents their use in many applications such as dealing with models with high depth complexity.

To sort fragments efficiently for rendering multi-fragment effects on a GPU, some methods, including the DP algorithm [1], [14], the DDP algorithm using a min-max depth buffer [2], and the algorithm to peel multiple layers simultaneously using multiple render targets (MRT) on the GPU [15], proposed dealing with one or more layers of fragments in each pass. Carr et al. [16] proposed using coherent layer peeling to exploit correctly sorted sequences of layers at a given pixel for a partially sorted collection of meshes between successive frames at the fragment level. Generally, this requires many data access passes to sort all the fragments.

To reduce the cost of depth ordering of fragments, some methods have proposed handling other primitives, thereby reducing the number of primitives to be processed or adopting efficient sorting techniques for the corresponding primitives. For example, Govindaraju et al. [17] proposed executing visibility ordering of the objects in a scene, Wexler et al. [6] proposed discretizing the scene into grids of pixel-sized quadrilateral micropolygons, while Eisemann et al. [18] tried to voxelize the scene into regular volume data. Though these methods increase the efficiency of depth ordering, they also need to read the data many times for high quality rendering.

In some cases, the rendering effect is determined mainly by the first few layers of fragments, while the

contribution of subsequent layers of fragments is ignored. Thus, some methods read the data only once to render multi-fragment effects by considering only the first few layers of fragments. As these methods execute ordering operations on the GPU, ordering errors may occur, thus reducing the rendering quality. For example, the k-buffer method [3] uses read-modify-write operations to implement ordering of the first k fragments. When multiple fragments at the same pixel have simultaneous read or write operations to the same entry of the k-buffer, read/write conflicts occur, causing ordering errors. To avoid such conflicts, multi-sample anti-aliasing buffers [11] have been proposed to work with the stencil routing operation for ordering fragments. Owing to the size limit of the buffer, it can only handle at most the first eight layers. Liu et al. [4] tried to render 32 layers of fragments by allocating a bucket array of 32 entries per pixel from the MRT buffer and dividing the depth range of the model into 32 intervals. Using this array, a fragment is inserted into its corresponding entry according to its ordering depth. However, fragments with their depth values in the same depth interval may be inserted into the same entry, causing ordering errors. Generally speaking, these methods work well when there are not many depth layers. If the model has high depth complexity, however, the data must still be read many times, mainly to reduce the sorting conflicts.

To reduce the number of data passes, Liu et al. [5] proposed implementing a complete rendering pipeline for multi-fragment effects in CUDA by exploiting CUDA atomic operations, where a buffer is allocated to sort all depth layers. In this method, a fixed-size array is used for every pixel, which prevents its use in handling models with a high depth complexity. Yang et al. [19] also proposed allocating a buffer to construct lists of fragments on the GPU, but with a more efficient way of using memory, which allows the lists at pixels to be of arbitrary length. It was reported that this method is much faster than the Freepipe method [5] for an order-independent transparency rendering. To summarize, using a single pass to obtain the data, these methods work well with smaller models. However, owing to the size limit of the memory on the GPU, they cannot deal efficiently with large models.

The deferred blending method [20] for point-based rendering adopts the strategy of separating the model data to promote multi-fragment effects rendering. However, this decomposition differs from ours in that the point data are decomposed into groups, guaranteeing that any pair of points in a group is far enough apart to be ordered correctly. As a result, the points in a group can be rendered correctly. But the rendered results of the groups are composited with weights, irrespective of the ordering relations between the points from different groups. Thus, this method cannot guarantee correct rendering of order-dependent effects, especially for models with high depth complexity.

From the above discussion, it is clear that existing methods cannot deal efficiently with large models, which may prevent application in many cases since large models are required more and more in practice. As for our method, it has a very low bounded memory requirement independently of the depth complexity and requires only a single data access pass. As such it is ideal for dealing with large models and is likely to promote widespread application as well.

Besides depth ordering fragments, some methods studied how to compute transmittance information of fragments efficiently for high quality rendering. Kim et al. [21] took opacity shadow maps to reuse pre-accumulated opacity on a regular grid along light rays. To remove layering artifacts, the method using deep opacity maps tries to shift the depth slices through the nearest layer to the light [22]. Recently, approximation-based techniques have attracted much attention. These include extending screen-door transparency with random sub-pixel stipple patterns to construct a stochastic representation of transmittance [23], using a Fourier series to represent a transmittance function [24], and building an adaptively compressed representation of the transmittance function in bounded memory [25]. In fact, since our proposed method focuses on depth ordering, which is helpful to efficient visibility computation, it can be used to improve transmittance computation, because accurate visibility is very important for high quality rendering, as discussed by Salvi et al. [25].

3 Model Decomposition

In our method, the model is decomposed into many parts, with the aim of correctly rendering each part on a GPU. As we use the stencil-routed A-buffer technique [11] for rendering, which can correctly handle at most eight layers of fragments at a same time, we assume that each decomposed part can have at most eight depth layers for any viewing direction, in order to be dealt with efficiently.

As is known, a convex polyhedron has at most two depth layers with respect to any viewing direction. Thus, we first decompose the model into convex polyhedrons, and then create a part composed of at most four nearby convex polyhedrons. In this way, every decomposed part has no more than eight depth layers of fragments. In our method, the number of decomposed parts determines the number of iterations for data transmission and result composition, which has a great influence on the rendering efficiency if there are many decomposed parts. Thus, we aim to produce the smallest number of decomposed parts. Unfortunately, this is an NP-hard problem. Considering this, we attempt to combine as many parts as possible, while executing model decomposition quickly for applying our method easily, as described in the following paragraph.

To decompose a model quickly, we construct a grid structure based on the bounding box of the model so that the facets in a grid cell can be treated locally and in

```
// Input all the facets of a grid cell, tagged unused.
void ConstructConvexs (in facets, out convexs)
    while(there are facets unused)
        Select a facet from unused facets as a seed;
        Constructing a convex polyhedron by greedily
            clustering the seed facet with adjacent facets;
    end while
```

Fig. 2. Pseudocode for constructing convex polyhedrons from the facets in a grid cell.

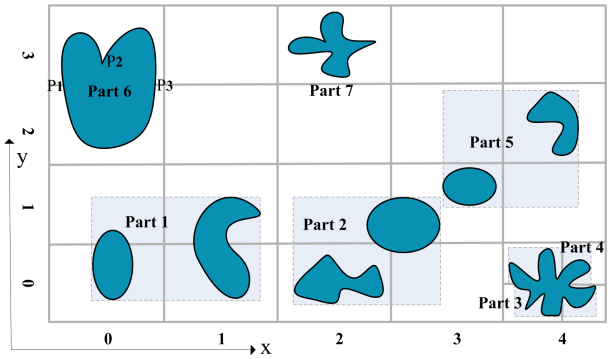


Fig. 3. Model decomposition. First, the facets in every grid cell are respectively clustered into convex polyhedrons with a seeding algorithm. Second, by the facets across neighboring grid cells, their related convex polyhedrons are combined, e.g. the facets in cells (0, 0) and (0, 1) are combined into a convex polyhedron. Afterwards, the grid cells are checked whether the convex polyhedrons in neighboring cells can be combined as a part. If there are no more than four convex polyhedrons in neighboring cells, they are combined, such as the formed Parts 1, 2 and 5. If a cell contains more than four convex polyhedrons, this cell should be subdivided iteratively until every subcell contains no more than four convex polyhedrons, as for treating the cell containing Parts 3 and 4. Here, the grid cells are swept from left to right and from bottom to up.

parallel. In a grid cell, we sample some facets as seeds to be extended individually to form their respective convex polyhedrons, which may be unclosed. The algorithm of constructing convex polyhedrons is described in Fig. 2. Then, from the facets across neighboring grid cells, their related convex polyhedrons in different cells are combined to form a larger convex polyhedron if possible, to reduce the number of decomposed parts. Thereafter, we find the grid cells that have no more than four convex polyhedrons, and allow each of these grid cells to form a decomposed part. The decomposed parts in nearby grid cells are further combined to form a larger decomposed part if there are no more than four convex polyhedrons in these grid cells. Here, a convex polyhedron overlapping several grid cells is allowed to be in only one decomposed part. As for a grid cell containing more than four convex polyhedrons, it is subdivided iteratively into smaller grid cells until no cell contains more than four convex polyhedrons. Clearly, in our decomposition process, some operations can be implemented in parallel to achieve greater speedup, as discussed in Section 6.

Without loss of generality, a 2D example is illustrated in Fig. 3, where the grid consists of 5×4 cells. The convex parts in cells (0, 0) and (0, 1) can be combined to form a larger convex part, which can be further combined with the parts in cells (1, 0) and (1, 1) to form an even larger decomposed part, Part 1. The convex part P1P2 in cell (0, 3) cannot be combined with the convex part P1P3 in cell (0, 2) to form a larger convex part, but the parts in these two cells can be combined to form the decomposed part, Part 6. Part 2 is composed of the parts in cell (2, 0) and a convex part in cell (3, 1), so that the convex parts in cell (3, 1) are separated into two decomposed parts, Part 2 and Part 5. Because there are more than four convex parts in cell (4, 0), this cell is subdivided, and Parts 3 and 4 are formed from the parts in the subdivided subcells.

4 Managing Parts With Grids

Our method manages the decomposed parts in grids, allowing the parts in different grid cells to be efficiently sorted, since it is very easy to order grid cells. If all grid cells contain at most one part, the ordering of parts is very easy. However, some grid cells may contain more than one part. To correctly order the parts in a grid cell, we define two procedures, for a grid cell containing two parts and more than two parts, respectively.

4.1 For a Cell Containing Two Parts

If two parts are interlocking, it is impossible to order them. In this case, we produce a slab structure between the two parts to separate them, thereby obtaining the correct ordering. Without loss of generality, we describe this measure using the 2D example illustrated in Fig. 4, where the rectangle in cyan represents a slab. A slab is determined by two parallel planes, as represented by lines 1 and 2 in Fig. 4, including the facets from the two parts that are in between these two parallel planes. As shown in Fig. 4, besides the facets within the slab, the remaining facets of Part 1 are on the left of the slab, while the remaining facets of Part 2 are on the right of the slab. Thus, the set of facets in Part 1 excluding its facets within the slab, the slab, and the set of facets in Part 2 excluding its facets within the slab, can be correctly ordered for any viewing direction because they can be separated by the planes between them. As a result, for a grid cell containing two parts, we decompose the two parts further with the slab between them, and also consider the slab as a part to be rendered. To ensure that the slab can be ordered conveniently, we use a cuboid to represent the slab. This is produced by projecting the facets of the slab onto one of its parallel planes orthogonally along the normal of the parallel planes

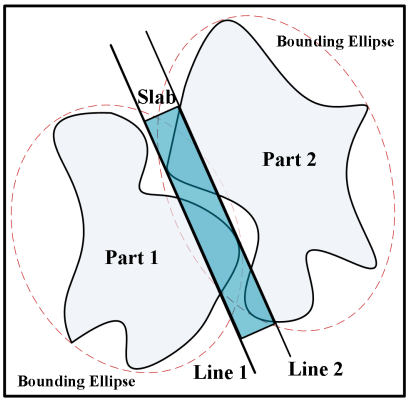


Fig. 4. A slab is constructed to separate two neighboring parts; it includes the facets from the two parts that lie in between the two parallel planes of the slab.

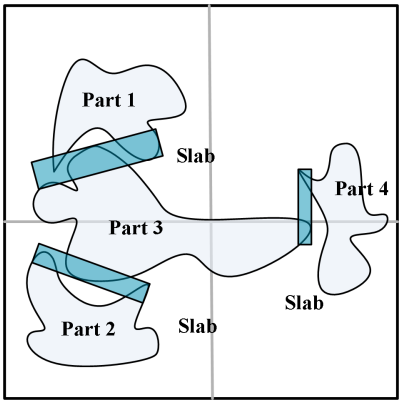


Fig. 5. Sub-grids are optimistically constructed by the distribution of slabs in a cell, where cyan rectangles represent slabs, and gray lines denote the partition lines for producing sub-cells.

to obtain the projection regions. Using the bounding box of the projection regions and the distance between the two parallel planes, the cuboid is obtained as the representation of the slab.

To produce the slab, we first need to find two parallel planes in between the two parts with as small a distance between them as possible. Our aim is to try to obtain a plane to separate the two parts to avoid the decomposition of these two parts as far as possible. Numerous methods exist for finding separating planes between shapes. These have been developed most in the context of collision detection. However, these methods are valid for convex polyhedrons, not suitable for our case of finding the plane that is most possible to separate two polyhedrons that are interlocked each other, as shown in Fig. 4. For example, the methods proposed by Chung et al. [26] and Wang et al. [27] can efficiently get separating planes between two convex polyhedron or two ellipsoids, but are invalid when the polyhedrons or ellipsoids have an overlapping region. Therefore, to deal with such a difficult problem, we use an approximate measure to estimate the normal of the planes, and then from the normal find a pair of parallel planes to produce a slab. The process comprises three steps. The first generates two bounding ellipsoids of the two parts, respectively, the second collects the facets of the parts that are inside the overlapping region of the two ellipsoids, and the third generates a small ellipsoid bounding the collected facets using principal component analysis. As a result, the direction of the shortest axis of the small ellipsoid is taken as the normal. Because the overlapping region of the bounding ellipsoids covers well the facets that are close to each other, and the largest section of the bounding ellipsoid for the collected facets is on the plane with the lowest quadric errors from the facets to the plane [28], it is highly likely that the estimated normal will produce a very thin slab. This process is illustrated in Fig. 4.

Though a part has at most eight depth layers, a slab may have more than eight depth layers for some viewing directions because it includes facets from two parts. Thus, every slab should be checked to see that it satisfies our requirement that each decomposed part should have no more than eight depth layers for any viewing direction. If it does not, the slab will have its representative cuboid subdivided iteratively by the mid-planes parallel to the faces of the cuboid, until every sub-cuboid contains no more than four convex polyhedrons. As the sub-cuboids can be ordered correctly for any viewing direction, there is no problem in rendering when the facets in a sub-cuboid are dealt with as a decomposed part.

4.2 For a Cell Containing More Than Two Parts

Any cell containing more than two parts is recursively subdivided to construct a local hierarchical grid in which a cell at a leaf node contains at most two parts. We then use the process in Subsection 4.1 to deal individually with those cells containing two parts. Iteratively dealing with the hierarchical grid cells causes no problems in ordering the parts.

4.3 Grid Resolution

Our method employs grid cells to facilitate ordering of decomposed parts. If too many cells are produced, this may seriously impact the ordering efficiency. To ensure that not too many cells are produced and to support the ordering computation, we adopt the following techniques to build grids and sub-grids.

The initial grid is constructed with its cell size as the average size of the bounding boxes of the decomposed parts. For this, we first compute the bounding box for every part, and then average the sizes of the bounding boxes along the three axes separately.

Each cell in the initial grid containing more than two parts is subdivided to construct a local hierarchy of grids until the cells at leaf nodes contain at most two parts. To subdivide a grid cell efficiently, we investigate the

distribution of the slabs in the cell to find a suitable grid resolution to construct its sub-grid, with the aim that every sub-cell contains at most one slab as illustrated in Fig. 5. Of course, if a sub-cell has more than one slab, it should be further subdivided iteratively using the same technique.

5 Rendering

Our rendering pipeline consists of three steps. First, according to the viewpoint, the parts are ordered from front to back or from back to front, and then transmitted orderly from the CPU to the GPU. Next, every part transmitted to the GPU is rendered using the stencil-routed A-buffer method [11] to obtain an individual rendered image. Finally, after the individual image for a part has been rendered, it is immediately composited with the previously rendered parts. These steps are discussed in the following subsections.

5.1 Ordering Parts

When a viewpoint is set, we can easily order the parts via the constructed grids. Without loss of generality, we discuss this with respect to ordering the parts from front to back. Ordering the parts from back to front can be dealt with similarly. For simplicity, we only discuss ordering the parts in a grid. As for a cell with local sub-grids, its hierarchical sub-grids are iteratively dealt with in a similar manner, and the results are used when the cell is processed in its located grid.

Our strategy is to order parts by their related grid cells. As we know, the grid cells can be processed sequentially slice by slice, row by row in a slice, and one by one in a row. Thus, from the sequence of cells sequentially from front to back, we can find the order to transmit these parts, thereby ensuring that every part is transmitted and processed earlier than its occluded parts. In this way, we can immediately composite the rendered image of a part with the rendered results of the processed parts, causing no problems in rendering multi-fragment effects.

We illustrate this using the 2D example in Fig. 6, which contains five decomposed parts. For viewpoint V, according to its relative position to the bounding box of the scene, we know that Row 3 is in front of Row 2, Row 2 is in front of Row 1, and Row 1 is in front of Row 0. Moreover, in each row, a grid cell is in front of the grid cell to its right. Checking the cells in Row 3, we find that Cell (0, 3) contains Part A and since Part A occupies no other cells, it is in front of the other parts. Next, on checking Cell (2, 3), we find that this cell is occupied by Part B, which also occupies several cells that have not yet been checked. In this case, we proceed to check the rows that have cells occupied by Part B. In checking the rows sequentially from front to back, we find that Cell (0, 2) is occupied by Part C, which also occupies some cells that have not yet been checked. We again proceed to check the rows that have cells occupied by Part C. After checking Cell (0, 1), we are sure there is

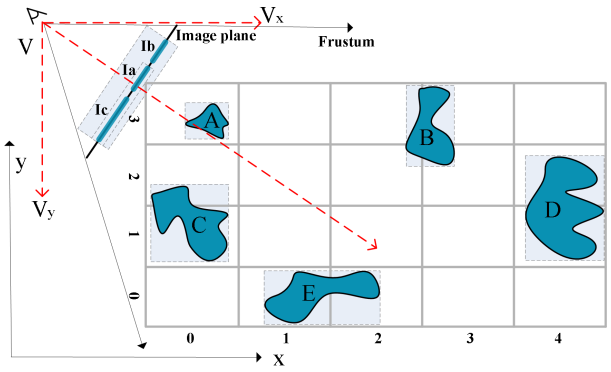


Fig. 6. The decomposed parts can be correctly depth ordered with respect to viewpoint V by ordering the grid cells containing the parts.

no other part in front of Part C, so we let Part C follow Part A. Then, we recursively check Cells (1, 2), and (2, 2) sequentially to ascertain that there are no other parts in front of Part B. At this time, we let Part B follow Part C. With similar recursive checks, we know that Part D follows Part B, with Part E being the last. Therefore, the order for transmitting these parts is Part A, Part C, Part B, Part D, and Part E.

By the ordered sequence for transmitting parts as discussed in the above paragraph, we can further pack the nearby parts in the sequence that cause no ordering errors in this rendering, and take the union as a part to transmit and render. With this, the times for data transmission and result composition can be reduced for more acceleration. As illustrated in Fig. 6, we project the bounding boxes of Part A, Part C and Part B orderly onto the image plane through the projection matrix in the CPU, and find no overlaps between Projection Ia and Ic, and then between the bounding box of Ia and Ic and Projection Ib, so that these three parts can be packed together. Similarly, Part D and Part E can be packed. As a result, these five parts are finally transmitted and rendered in two packs.

However, in determining the order of transmitting parts, a cyclic occlusion between the parts may occur, which means the involved parts each are occluded by at least another one, causing no part selected to transmit from the involved parts. For this, we transmit the parts gradually in packs by every time forming a pack of the unoccluded parts in the remained parts that have not been transmitted, and we know cyclic occlusions occur when no unoccluded part can be found in the remained parts, according to the discussion by Williams [29]. When a cyclic occlusion is met, we adopt the heuristic as suggested by Williams [29] to iteratively find the biggest part from the involved parts and decompose it in the middle, which may take a reverse procedure of generating the part from its facets, until the cyclic occlusion is erased, meaning some parts become unoccluded. In the worst case we decompose a part into its occupied cells with the

cells each having a sub-part of the part, because cyclic occlusions cannot occur between grid cells.

5.2 Rendering a Part

We render a part on the GPU using the method with stencil routed a-buffer [11]. First, all the fragments from the part are rasterized and a multi-sample texture is used to store a vector of fragments per pixel. Then, in the bounding quad of the projection of the bounding box of the part, a shader pass is carried out to order the fragments from the part using a bitonic sort. After the fragments have obtained their colors by illumination computation, they are blended to give colors to their corresponding pixels.

5.3 Composition

On the GPU, we allocate a buffer to store the colors at pixels in order to render the whole model. Each time a part is rendered, its rendered image is composited with the colors of the pixels in the buffer. After all the parts have been processed, we obtain the final image of rendering the whole model.

6 Implementation, Results, And Discussion

We performed our tests on a personal computer running Microsoft Windows 7 and installed with an Intel Xeon E5620 2.4 GHz CPU with 8 cores, 8 GB RAM, and an NVIDIA GTX 680 card with 2048 MB video memory. For comparison, we implemented our method using OpenGL, and also some existing methods that can execute order-dependent rendering, namely DP [1], DDP [2], BDP [4], bucket depth peeling in two passes (BDP2) [4], adaptive bucket depth peeling (ADP) [4], and two schemes for the Freepipe method in CUDA [5], that is, one using a multi-depth test (MDTS) and the other an A-buffer (ABS).

6.1 Preprocessing

Our preprocessing is executed on the CPU. For greater speedup, we use multiple threads to deal with certain operations for model decomposition in parallel. These operations include greedily clustering from a seed facet to form a convex polyhedron in a grid cell, combining convex polyhedrons via their shared facets across neighboring cells, detecting the grid cells that have no more than four convex polyhedrons, subdividing grid cells that have more than four convex polyhedrons, and so on.

The tested models together with statistics on their triangles and preprocessing are listed in Table 2. From these statistics, it can be seen that our preprocessing can handle a very large model in several minutes. This means that our preprocessing should not be an obstacle in adopting our method for various applications. Actually, managing parts instead of facets in representing a model can bring many benefits in model processing, such as treating facets in groups. This is an interesting issue for future study.

TABLE 1

Comparison between our scheme and kd-trees for managing parts from the Dragon model.

Schemes	Parts	*Storage (MB)	Preprocessing (s)	Performance (ms)
Kd-tree (facets)	$216K$	10.5	7	103.09
Kd-tree (parts)	52	3.9	17	4.88
Ours	36	3.7	15	3.98

*Storage (MB) *: the listed storage excludes that for the information representing the model, including vertices, normals, indices, and so on.*

6.2 Management

In our method, we use grids to manage decomposed parts for easy ordering, instead of using hierarchical management schemes such as the kd-tree, although these are popular for efficient queuing. This is because it is very time-consuming to construct a hierarchical tree, and the partition planes for tree construction may intersect parts, thereby increasing the number of parts and resulting in greater storage and time for parts.

We conducted tests on the Dragon model to compare our management scheme and a kd-tree structure. Two kd-trees were used for the facets and our decomposed parts, respectively. Leaf nodes in the kd-tree for facets each contain at most 8 facets so that the facets of a leaf node can be transmitted and rendered as a part, while leaf nodes in the kd-tree for our decomposed parts each contain one part. The statistics for the comparison are given in Table 1, where the rendered images are each 1024×1024 pixels and parts are all transmitted in packs, irrespective of the management scheme used.

From the statistical data in Table 1, it can be seen that managing parts instead of facets can considerably reduce storage requirements and rendering time, though the kd-tree for facets can be constructed much faster. As for managing parts, our scheme is faster than a kd-tree for preprocessing, which has more parts added by dividing our decomposed parts with partitioning planes, and achieves a speedup of (4.88-3.98)/3.98=22.6% for rendering. Thus, our management scheme is more efficient for rendering multi-fragment effects.

6.3 Rendering

We tested the efficiency of these methods in rendering the effects of transparency and translucency. For transparent rendering, depth ordering of fragments is only required for compositing the illumination colors via the opacity values. As for translucent rendering, the depth values of fragments are required to compute the attenuation effects. Here, two kinds of attenuation effects are taken into account. The first, caused by ray traversal through the model, is computed according to Beer-Lambert's law. The second is the Fresnel effect

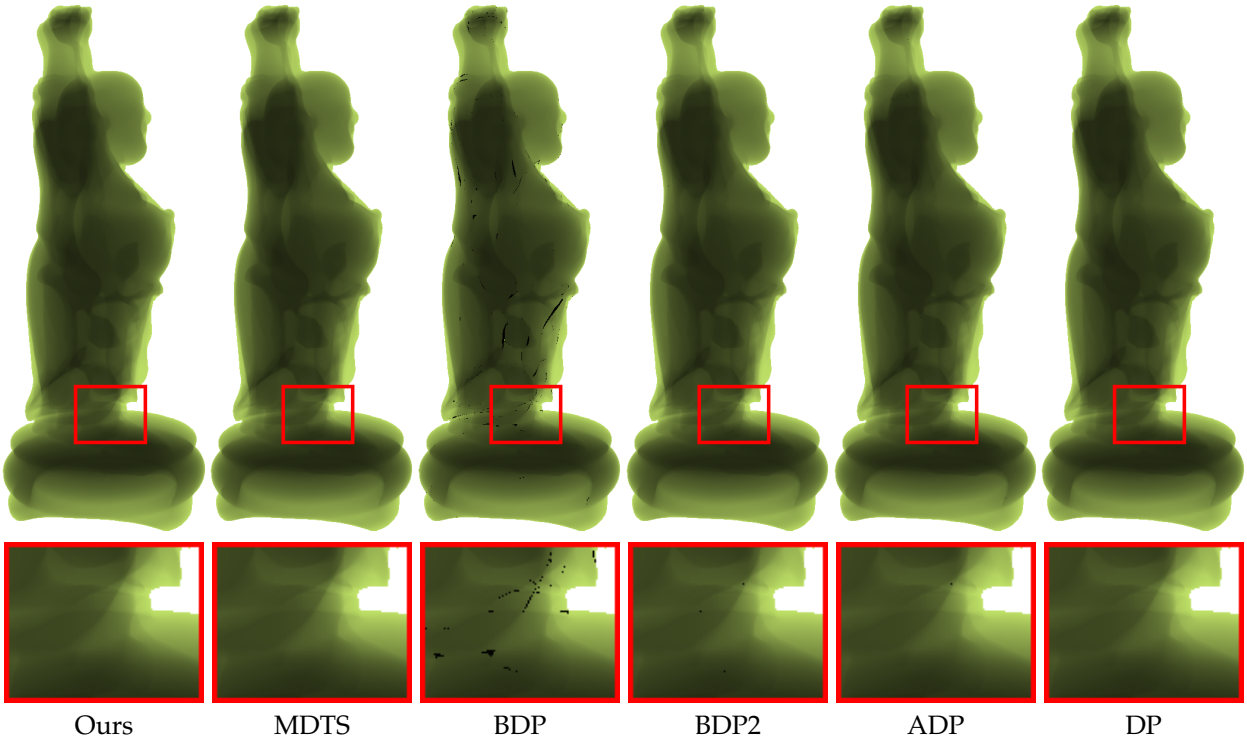


Fig. 7. Rendered images for the Buddha model with translucent effects using various methods for comparison. The compared methods are the Freepipe method using a multi-depth test (MDTS) [5], the methods with bucket depth peeling (BDP) [4], bucket depth peeling in two passes (BDP2) [4], adaptive bucket depth peeling (ADP) [4] and depth peeling (DP) [1]. As for the Freepipe method using an A-buffer (ABS) [5] and the method with dual depth peeling (DDP) [2], we don't displayed their rendered images, since the images by MDTS and ABS are the same, and similarly for DP and DDP.

caused by ray refraction, which is approximated using the Schlick formula [30].

Fig. 7 shows the rendered images when applying these methods to the Buddha model with translucent effects. We also present an enlarged rectangle for each image showing some of the rendering details by these methods. It is clear that our method produces the same high quality results to the DP and DDP methods, in which all fragments are ordered correctly, and superior results to the other methods. In Fig. 8, we display the rendered images by our method for the tested models with transparent or translucent effects. All images are of high quality. More results are presented in the supplemental video.

As for rendering efficiency, we give the statistics for rendering transparent and translucent effects by these methods in Table 2. The results show that our method is faster than existing methods, except in the case of rendering the transparent effect of the Powerplant model where it is a little slower than BDP, which only handles 32 depth layers compared with the 152 depth layers considered in our model. Of course, the rendered images by BDP are generally inferior to those obtained by our method owing to ordering errors caused by its limited depth layers. When the model has fewer depth layers, our method is much faster than BDP, for example, rendering the Horse, Dragon, and Buddha models. From the data in Table 2, our method achieves greater speedup in rendering translucent effects than in rendering transparent effects. This may be because our low memory requirement is more beneficial in supporting complex illumination, where it is required more data to transmit, such as normal information.

By investigating the relation between depth layers and speedup ratios, we find that our method tends to obtain greater speedup with more depth layers, compared with the MDTS, ABS, DP, and DDP methods. As for the BDP, BDP2, and ADP methods dealing with a fixed number of depth layers, our speedup ratios compared with theirs decrease with an increase in depth layers, because our cost increases as the number of depth layers increases.

As our method immediately composites the rendered result from a part and each part has at most eight depth layers, it can effectively restrict the length of the linked nodes at a pixel on the GPU, and so has a low bounded storage requirement. The memory requirements for rendering an image with 1024×1024 pixels of our method and the compared methods are given in Table 3. Obviously, our memory requirement is very low and bounded, not increasing with an increase in depth layers. Though BDP, BDP2, ADP, DDP, and DP do not require much memory for rendering, they need a buffer to store the whole model on the GPU. This prevents them from rendering large models, e.g., they may crash

TABLE 2

Statistical data for the tested models, and their decomposition and rendering efficiency using the compared methods, where these methods are the Freepipe method using a multi-depth test (MDTS) or an A-buffer (ABS) [5], the methods with bucket depth peeling (BDP) [4], bucket depth peeling in two passes (BDP2) [4], adaptive bucket depth peeling (ADP) [4], dual depth peeling (DDP) [2] and depth peeling (DP) [1]. The rendering time for a frame (milliseconds/frame) for a model using a particular method is obtained by averaging the rendering time to produce many images around the model, where the images all contain 1024×1024 pixels and are rendered with similar distances from the viewpoint to the model.

Model	Horse	Dragon	Buddha	Powerplant	Lucy	Xyzrgb_dragon
Triangles	$97K$	$871K$	$1,087K$	$12,748K$	$28,055K$	$43,313K$
Averaged max layers	11	13	16	152	26	29
Decomposed parts	14	36	48	2968	1131	2115
Preprocessing (seconds)	2	15	19	286	143	206
Transparent Effect						
Ours	2.05	3.98	4.24	64.18	79.55	114.03
MDTS	3.18(0.55)	6.61(0.66)	7.23(0.71)	127.39(0.98)	147.93(0.86)	221.24(0.94)
ABS	2.6(0.27)	5.54(0.39)	6.05(0.43)	96.9(0.51)	123.15(0.55)	190.11(0.67)
BDP*	2.97(0.45)	6.15(0.55)	6.98(0.65)	62.93(−0.02)	122.7(0.54)	187.27(0.64)
BDP2**	4.44(1.17)	8.6(1.16)	9.6(1.26)	96.43(0.5)	158.23(0.99)	300.3(1.63)
ADP	7.76(2.79)	12(2.02)	13.59(2.21)	202.43(2.15)	289.02(2.63)	476.19(3.18)
DDP	3.09(0.51)	5.37(0.35)	7.17(0.7)	4761.9(73.2)	4166.67(51.38)	6250(53.81)
DP	4.32(1.11)	10.83(1.72)	13.63(2.21)	10000(154.81)	8333.33(103.76)	12500(108.62)
Translucent Effect						
Ours	2.15	4.56	4.7	82.24	106.16	149.03
MDTS	3.35(0.56)	7.79(0.71)	8.29(0.76)	168.92(1.05)	204.5(0.93)	299.4(1.01)
ABS	2.89(0.34)	6.87(0.51)	7.22(0.54)	124.84(0.52)	160.51(0.51)	252.53(0.69)
BDP*	3.9(0.81)	7.17(0.57)	8.21(0.75)	88.81(0.08)	160(0.51)	249.38(0.67)
BDP2**	5.45(1.53)	10.18(1.23)	11.96(1.54)	125.47(0.53)	240.38(1.26)	413.22(1.77)
ADP	8.43(2.92)	11.73(1.57)	17.66(2.76)	266.67(2.24)	358.42(2.38)	636.94(3.27)
DDP	3.49(0.62)	6.13(0.34)	8.24(0.75)	6666.67(80.06)	5555.56(51.33)	8333.33(54.92)
DP	5.17(1.4)	12.31(1.7)	16.32(2.47)	14285.71(172.71)	11111.11(103.66)	16666.67(110.83)

Note 1: *The numbers in brackets are the speedup ratios of our method to the compared method, computed as (oldtime-newtime)/newtime, where newtime refers to the rendering time of our method and oldtime to that of the other method.*
Note 2: *"Averaged max layers" refers to the averaged maximum depth layers for rendering images of a model, where the maximum depth layer for an image is obtained by checking the depth layer at each individual pixel.*
Note 3: *BDP* and BDP2** only handle the first 32 and 64 layers, respectively, while the others deal with all the depth layers.*

TABLE 3

Memory requirements (MB) of the various methods for rendering an image (1024×1024 pixels) with different depth layers. These methods are the Freepipe method using a multi-depth test (MDTS) or an A-buffer (ABS) [5], the methods with bucket depth peeling (BDP) [4], bucket depth peeling in two passes (BDP2) [4], adaptive bucket depth peeling (ADP) [4], dual depth peeling (DDP) [2] and depth peeling (DP) [1].

Depth Layers	Ours	MDTS	ABS	BDP	BDP2	ADP	DDP	DP
20	36	80	81	128	256	208	16	8
60	36	240	244	128	256	208	16	8
100	36	400	404	128	256	208	16	8

if a large model cannot be loaded onto the GPU. On the contrary, our method is suitable for dealing with very large models of any size, as long as each decomposed part can be loaded onto the GPU. This is easy to achieve by partitioning a part into smaller ones if the part is too big to be loaded.

From the above discussion, it is clear that our method can efficiently speed up rendering multi-fragment effects with all fragments ordered correctly. As the model becomes larger with more depth layers, our method generally yields a higher speedup.

6.4 Limitations

In our method, we need preprocess to reorganize the model, which prevents ours from treating the dynamic case that the model changes its shape with time. It is an interesting issue to study efficient techniques for multi-fragment effects rendering of deforming models.

In our method, we try to reduce the times for data transmission and result composition, achieving much faster than existing methods. However, for a model with a large quantity of small-scale features, our decomposed parts will be in a large number, and so lowering the rendering efficiency. This should be further studied, e.g., we plan to integrate our method and instancing techniques to treat some cases.

7 Conclusion

This paper presents a new method for rendering multi-fragment effects on a GPU, focusing particularly on rendering order-dependent effects. The method decomposes the model into parts, each of which has very few depth layers for any viewing direction, and manages the parts using grids. Because each part can be correctly depth ordered on the GPU, and the depth ordering of the parts can be easily obtained via the grid cells, we can transmit parts one by one according to their depth order

Fig. 8. Images rendered by our method with transparent or translucent effects: (a) Horse with translucent effects; (b) Dragon with transparent effects; (c) Powerplant with translucent effects; (d) Lucy with translucent effects; and (e) Xyzrgb_dragon with transparent effects. Here, some thin objects in the Powerplant image are not displayed with high quality, because their width is much smaller than the size of a pixel. This aliasing problem is not addressed in this paper since it is beyond the scope of this research.

for rendering and compositing in time on the GPU. All fragments are guaranteed to be ordered correctly for rendering multi-fragment effects. In this way, we need only conduct a single pass over the data, with a very low bounded memory requirement, which is not sensitive to the number of depth layers. Thus, our method is much faster than existing methods, and yields even greater speedup for larger models with more depth layers. The results verify the effectiveness of our method and its potential to deal with very large models, which are generally very difficult for existing methods to handle.

There are some issues associated with our new method that need further investigation. The first is to reduce the times for data transmission even more by reducing the number of parts, or more efficiently packing parts that cause no ordering errors for simultaneous transmission. The second is to integrate our method with its orthogonal methods, such as the method using fragment-parallel composition and filtering [31], to further enhance the rendering of multi-fragment effects on a GPU. The third is to extend our method for treating dynamic scenes. Although our preprocessing takes much time, the larger portion (about 90% in our tests) is on model decomposition and part generation, and the results can be reused in some dynamic applications, such as for a rigid motion.

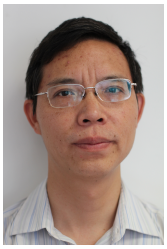
Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments, and Fang Liu and Mengcheng Huang for discussion and providing the source codes for bucket depth peeling and Freepipe. This work was partially supported by NSF of China (60773026, 60833007).

References

[1] C. Everitt, "Interactive order-independent transparency," NVIDIA Corporation, Tech. Rep., 2001.
[2] L. Bavoil and K. Myers, "Order independent transparency with dual depth peeling," NVIDIA Corporation, Tech. Rep., 2008.
[3] L. Bavoil, S. P. Callahan, A. Lefohn, J. L. D. Comba, and C. T. Silva, "Multi-fragment effects on the gpu using the k-buffer," in *Proc. of the 2007 symposium on Interactve 3D graphics and games*, 2007, pp. 97–104.
[4] F. Liu, M. C. Huang, X. H. Liu, and E. H. Wu, "Efficient depth peeling via bucket sort," in *Proc. of the Conference on High Performance Graphics 2009*, 2009, pp. 51–57.
[5] ——, "Freepipe: programmable parallel rendering architecture for efficient multi-fragment effects," in *Proc. of the 2010 symposium on Interactve 3D graphics and games*, 2010, pp. 75–82.
[6] D. Wexler, L. Gritz, E. Enderton, and J. Rice, "Gpu-accelerated high-quality hidden surface removal," in *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2005, pp. 7–14.
[7] E. Catmull, "A subdivision algorithm for computer display of curved surfaces," Ph.D. dissertation, University of Utah, Dept. of Computer Science, 1974.
[8] L. Carpenter, "The a-buffer, an antialiased hidden surface method," *Computer Graphics (Proc. of SIGGRAPH '84)*, vol. 18, no. 3, pp. 103–108, 1984.
[9] M. Wittenbrink, "R-buffer: a pointerless a-buffer hardware architecture," in *Proc. of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 2001, pp. 73–80.
[10] W. R. Mark and K. Proudfoot, "The f-buffer: a rasterization-order fifo buffer for multi-pass rendering," in *Proc. of the ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 2001, pp. 57–64.
[11] K. Myers and L. Bavoil, "Stencil routed a-buffer," in *Proc. of SIGGRAPH '07 sketches*, 2007.
[12] P. Jouppi and F. Chang, "z^3: an economical hardware technique for high-quality antialiasing and transparency," in *Proc. of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 1999, pp. 85–93.
[13] S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva, "Hardware-assisted visibility sorting for unstructured volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 3, pp. 285–295, 2005.
[14] A. Mammen, "Transparency and antialiasing algorithms implemented with the virtual pixel maps technique," *IEEE Computer Graphics and Applications*, vol. 9, no. 4, pp. 43–55, 1989.
[15] B. Q. Liu, L. Y. Wei, and X. Y. Q, "Multi-layer depth peeling via fragment sort," Microsoft Research Asia, Tech. Rep., 2006.

[16] N. Carr, R. Měch, and G. Miller, "Coherent layer peeling for transparent high-depth-complexity scenes," in *Proc. of the ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 2008, pp. 33–40.
[17] N. K. Govindaraju, M. Henson, M. C. Lin, and D. Manocha, "Interactive visibility ordering and transparency computations among geometric primitives in complex environments," in *Proc. of the 2005 symposium on Interactive 3D graphics and games*, 2005, pp. 49–56.
[18] E. Eisemann and X. Décoret, "Fast scene voxelization and applications," in *Proc.of the 2006 symposium on Interactive 3D graphics and games*, 2006, pp. 71–78.
[19] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-time concurrent linked list construction on the gpu," *Computer Graphics Forum (Proc. of the Eurographics Symposium on Rendering '10)*, vol. 29, no. 4, pp. 1297–1304, 2010.
[20] Y. ZHANG and R. PAJAROLA, "Deferred blending: Image composition for single-pass point rendering," *Computers & Graphics*, vol. 31, no. 2, pp. 175–189, 2007.
[21] T. Y. Kim and U. Neumann, "Opacity shadow maps," in *Proc. of the 12th Eurographics Workshop on Rendering Techniques*, 2001, pp. 177–182.
[22] C. Yuksel and J. Keyser, "Deep opacity maps," *Computer Graphics Forum (Proc. of EUROGRAPHICS '08)*, vol. 27, no. 2, pp. 675–680, 2008.
[23] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke, "Stochastic transparency," in *Proc. of the 2010 symposium on Interactve 3D graphics and games*, 2010, pp. 157–164.
[24] J. Jansen and L. Bavoil, "Fourier opacity mapping," in *Proc. of the 2010 symposium on Interactve 3D graphics and games*, 2010, pp. 165–172.
[25] M. Salvi, J. Montgomery, and A. Lefohn, "Adaptive transparency," in *Proc. of the ACM SIGGRAPH Symposium on High Performance Graphics*, 2011, pp. 119–126.
[26] K. Chung and W. Wang, "Quick collision detection of polytopes in virtual environments," in *Proc. of the ACM Symposium on Virtual Reality Software and Technology*, 1996, pp. 125–132.
[27] W. Wang, Y.-K. Choi, B. Chan, M.-S. Kim, and J. Wang, "Efficient collision detection for moving ellipsoids using separating planes," *Computing*, vol. 72, pp. 235–246, 2004.
[28] P. Lindstrom, "Out-of-core simplification of large polygonal models," in *Proc. of SIGGRAPH '00*, 2000, pp. 259–262.
[29] P. Williams, "Visibility-order meshed polyhedra," *ACM Transactions on Graphics*, vol. 11, no. 2, pp. 103–126, 1992.
[30] C. Schlick, "An inexpensive brdf model for physically-based rendering," *Computer Graphics Forum*, vol. 13, no. 3, pp. 233–246, 1994.
[31] A. Patney, S. Tzeng, and J. D. Owens, "Fragment-parallel composite and filter," *Computer Graphics Forum (Proc. of the Eurographics Symposium on Rendering '10)*, vol. 29, no. 4, pp. 1251–1258, 2010.



Wencheng Wang received his Ph.D. degree from Institute of Software, Chinese Academy of Sciences in 1998, where he is currently a professor of the State Key Laboratory of Computer Science. His research interests include computer graphics, visualization, virtual reality and expressive rendering and editing.

Guofu Xie received his BS degree in software engineering from Xiamen University in 2007. He is now a PhD student at State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. His major research interests lie in real-time photorealistic rendering and vector rendering.