Efficient Depth Peeling via Bucket Sort

Fang Liu[‡] Meng-Cheng Huang[‡] Xue-Hui Liu[‡] En-Hua Wu[‡§]
Institute of Software, Chinese Academy of Sciences[‡] University of Macau[§]

Abstract

In this paper we present an efficient algorithm for multi-layer depth peeling via bucket sort of fragments on GPU, which makes it possible to capture up to 32 layers simultaneously with correct depth ordering in a single geometry pass. We exploit multiple render targets (MRT) as storage and construct a bucket array of size 32 per pixel. Each bucket is capable of holding only one fragment, and can be concurrently updated using the MAX/MIN blending operation. During the rasterization, the depth range of each pixel location is divided into consecutive subintervals uniformly, and a linear bucket sort is performed so that fragments within each subintervals will be routed into the corresponding buckets. In a following fullscreen shader pass, the bucket array can be sequentially accessed to get the sorted fragments for further applications. Collisions will happen when more than one fragment is routed to the same bucket, which can be alleviated by multi-pass approach. We also develop a two-pass approach to further reduce the collisions, namely adaptive bucket depth peeling. In the first geometry pass, the depth range is redivided into non-uniform subintervals according to the depth distribution to make sure that there is only one fragment within each subinterval. In the following bucket sorting pass, there will be only one fragment routed into each bucket and collisions will be substantially reduced. Our algorithm shows up to 32 times speedup to the classical depth peeling especially for large scenes with high depth complexity, and the experimental results are visually faithful to the ground truth. Also it has no requirement of pre-sorting geometries or post-sorting fragments, and is free of read-modify-write (RMW) hazards.

CR Categories: I.3.7 [Computing Methodologies]: Computer Graphics—Three-Dimensional Graphics and Realism

Keywords: graphics hardware; depth peeling; multiple render target (MRT); bucket sort; order independent transparency (OIT); Max/Min blending; histogram equalization.

1 Introduction

Multi-fragment effects play important roles on many graphics applications, including order independent transparency, volume rendering, collision detection, refraction and caustics, global illumination, etc., which require operations on more than one fragment per pixel. The classical depth peeling algorithm [Mammen 1989; Everitt 2001] provides a simple but robust solution by peeling off a single layer per geometry pass using the features of Z-buffer [Catmull 1974]. It works fairly well for simple scenes, but for large scenes with high complexity, multiple rasterizations of the geometry will become a performance bottleneck. The state-of-art single pass approach k-buffer gains great speed improvement but suffers from RMW hazards [Bavoil et al. 2007; Liu et al. 2006]. Our algorithm gains a great balance between performance and quality, and is capable to capture up to 32 fragments in a single geometry pass with great speed improvement and faithful results.

The basic idea of our algorithm is to perform a bucket sort on the fragment level. We exploit MRT buffers as a bucket array for each pixel location, and each bucket is capable of holding only one fragment. All the buckets can be concurrently updated by taking advantage of the new feature of 32-bit floating-point MAX/MIN blending of NVIDIA Geforce 8. This operation assures us to atomically update a specific channel of MRT while keeping the others unchanged, thus is free of RMW hazards. A bounding box or a coarse visual hull is first rendered to get the approximate depth range of each pixel location. While rendering the scene, the depth range is divided into consecutive subintervals uniformly and a linear bucket sort is performed on the pixel shader, i.e., fragments within each subinterval will be routed into the corresponding buckets. In a following fullscreen shader pass, the bucket array per pixel can be sequentially accessed to get the sorted fragments for post-processing. When multiple fragments are routed to the same bucket, i.e. a collision happens, the bucket will only hold the maximum one and the rest will be discarded, and artifacts will arise. We develop a two-pass approach to alleviate the problem, namely adaptive bucket depth peeling. In the first geometry pass, the depth range is divided into non-uniform subintervals to ensure the one-to-one correspondence between fragments and subintervals. In the following geometry pass, there will be only one fragment routed into each bucket, and collisions will be reduced substantially.

We believe that our algorithm is a great improvement for interactive rendering of multi-fragment effects, especially for large-scale scenes with high depth complexity. The contributions of our paper include:

- A novel linear solution is proposed to capture and sort multiple fragments by bucket sort in a single geometry pass, with significant speed improvement and faithful results. Collisions can be further reduced by multi-pass approach.
- An efficient two-pass approach namely adaptive bucket depth peeling is introduced, which can reduce the collisions substantially.

The rest of the paper is outlined as follows. Section 2 provides an overview of the previous work. Section 3 introduces the GPU implementation of the bucket array on graphics hardware, and describes our bucket depth peeling in detail. We also introduce an adaptive scheme to further reduce the collisions. Experimental results and applications are presented in Section 4. Finally, we provide a general discussion in Section 5 and conclude our paper in Section 6.

2 Related Work

The most intuitive idea of rendering multi-fragment effects is to pre-sort the geometries on the object level according to their distance to the eye, then render them in a back-to-front order, as typically shown in the painter's algorithm. The vis-sort [Govindaraju et al. 2005] exploits temporal coherence between successive frames and sorts the geometries on GPU before rasterization. For scenes with depth complexity N, the algorithm works in $O(N)$ for nearly sorted primitives and $O(N^2)$ for reverse order. The z-batches [Wexler et al. 2005] discretizes the scenes into grids and sorts the grids by the z value of the front face of the bounding box, then breaks the list of grids into batches and handles each batch with classical depth peeling. The best-case runtime complexity of the algorithm is $O(N)$ for sparsely distributed geometries and the worst-case is $O(N^2)$ for intensively overlapped grids. The slice map [Eisemann and Décoret 2006] approximately encodes the boundary representation of the geometries into a bitmask while voxelizing the scenes. It works well for volumetric effects, but one bit of information per voxel is far from enough for applications that require the precise depth value and other attributes of the fragments.

Another form of algorithm works on the fragment level. They try to sort fragments rather than geometries. The most well known technique is depth peeling [Mammen 1989; Everitt 2001], which peels off the front-most layer each geometry pass and naturally assures the correct depth ordering. It requires $O(N)$ geometry passes and $O(N)$ depth comparisons per pass, leading to a total complexity $O(N^2)$. Extensions include the A-buffer [Carpenter 1984], the R-buffer [Wittenbrink 2001], and the Z^3 algorithm [Jouppi and Chang 1999]. Unfortunately, all of these extensions need hardware modifications. The F-Buffer [Mark and Proudfoot 2001] maintains FIFO buffers to hold fragments in between rendering passes and need additional post-sort passes, with implementations available on ATI's graphics hardware [Houston et al. 2005]. Dual depth peeling [Bavoil and Myers 2008] succeeds to capture both the nearest and furthest layers in each pass by utilizing the 32-bit floating-point MAX/MIN blending, but the theoretical acceleration to classical depth peeling is limited to a factor of two.

K-buffer [Bavoil et al. 2007; Liu et al. 2006] presents a novel framework to capture up to k fragments in a single pass. The algorithm exploits MRTs as storage, and inserts each incoming fragment into an array maintained on MRTs. The theoretical acceleration rate of the algorithm is k times. However, it suffers from read-modify-write (RMW) hazards, which could be improved by dynamic pre-sorting of geometries, or triangle batches [Bavoil et al. 2007], but these modifications dramatically limit the performance of k-buffer. [Liu et al. 2006] tries to alleviate the RMW hazards by multi-pass approach, which guarantees to capture at least one fragment per pixel each geometry pass. But as the depth complexity of the scenes increases, the RMW hazards will also increase and the performance of the algorithm will degrade rapidly, which quite limits the utility of the algorithm. In addition, the benefit from the development of GPU is limited since the RMW hazards will also increase as the parallel degree of GPU increases, thus leads to more extra passes. The stencil routed a-buffer [Myers and Bavoil 2007] utilizes multisampling anti-aliasing (MSAA) buffers, and routes the fragments into sub-pixels of the MSAA using stencil test. An additional fullscreen pass is performed to sort the fragments using a bitonic sort in $O(N \times \log(N)^2)$ before rendering. It can completely avoid the RMW hazards, but the theoretical speedup to depth peeling is limited to 8 times due to the maximum number of MSAA samples. In addition, for scenes with high depth complexity, the bitonic sort in the post-processing pass will become the performance bottleneck.

The coherent layer peeling [Carr et al. 2008] exploits the property of sorted coherency between successive frames on the fragment level. It requires five geometry passes each iteration, and peels away only two layers per iteration in the worst case of reverse order. In addition, the sort of triangle centroids on CPU will degrade the performance of the algorithm especially for large scenes.

Ideally we wish to develop a linear algorithm for rendering multi-fragment effects, and the natural choice is the bucket sort. We perform a bucket sort on the fragment level to capture and sort multiple fragments in a single geometry pass. The algorithm will be described in the next section.

3 Our Algorithm

Our algorithm succeeds to capture and sort multiple fragments of each pixel in a single geometry pass via bucket sort on GPU. Specifically, a bucket array of fixed size K is allocated per pixel location in GPU memory, and the MAX/MIN blending operation is utilized to guarantee correct concurrent update of multiple fragments (Section 3.1). We divide the depth range of each pixel into K subintervals and route the fragments within the $k^{th}(k = 0, 1, \cdots, K-1)$ subinterval into the k^{th} bucket by a bucket sort (Section 3.2). In the following section 3.3, we also introduce an adaptive scheme to alleviate collisions when multiple fragments are routed to the same bucket.

3.1 GPU Implementation

The construction of bucket array requires a fixed size buffer per pixel location in GPU memory that can be updated concurrently on the pixel shader. We simply exploit MRT buffers to construct our bucket array. Given N MRT buffers with B_m bytes per pixel, we could store K depth values (with B_d bytes):

$$K = \frac{N \times B_m}{B_d} \tag{1}$$

Since modern GPUs can afford at most N = 8 MRTs with internal pixel format of GL_RGBA32F_ARB (B_m = 16 bytes) and depth value of format FP32 (B_d = 4bytes), the size of our bucket array can reach up to 32 (K= 8 * 16 / 4), which is often enough for most common applications.

The default REPLACE blending operation on MRTs will introduce a problem that we can not simply update a specific bucket individually on the pixel shader, since the update of a certain channel of MRT will result in the concurrent updates of all other $K-1$ channels. Thus the depth values which have already been captured and stored in other channels will be overwritten simultaneously by the default update value 0. It will produce unpredictable results under concurrent writes.

We solve this problem via the 32-bit floating point MAX/MIN blending operation which is available on recent commodity NVIDIA GeForce 8. MAX blending operation performs a comparison between the source and the destination values of each channel of MRTs, and keeps the greater of the two. We begin by initializing each bucket of the bucket array to 0. When updating a certain bucket, if the original value in the bucket is 0, the update will always succeed since the normalized depth values are always greater than or equal to 0; otherwise, the greater one will survive the comparison. As for other buckets, we implicitly update them simultaneously by the default value 0 so that their original values will always be kept unchanged. When multiple fragments are routed to the same bucket, i.e., a collision happens, the MAX blending operation assures that the maximum depth value will win all the tests and finally stay in the bucket. MIN blending is performed in a similar way except initializing and explicitly updating other buckets by 1. Using the MAX/MIN blending operation enables us to update a specific bucket of the bucket array, and guarantees correct results free of RMW hazards. Since the default update value for each channel of the MRT is 0, we prefer to utilize MAX blending in our implementation for simplicity.

3.2 Bucket Depth Peeling

The depth value of each fragment is normalized into a range [0,1] during rasterization. For each pixel, we denote the depth value of the nearest fragment as *zNear* and the furthest as *zFar*, and compose a precise depth range [*zNear*,*zFar*]. It can be easily obtained by rendering the scene in the same way as [Bavoil and Myers 2008]. But this extra pass over the geometry is expensive especially for large-scale scenes, so a bounding box or a much simplified visual hull is instead rendered to approximate the depth range of each pixel. In the first geometry pass, we bind consecutive buckets into pairs and divide the depth range into 16 subintervals: $[d_k, d_{k+1}), d_k = zNear + \frac{k}{16}(zFar - zNear), k = 0, 1, \cdots, 15$. For a fragment with depth value $d_f \in [d_k, d_{k+1})$, the corresponding bucket index k can be simply computed as follows:

$$k = floor(\frac{16 \times (d_f - zNear)}{zFar - zNear}) \tag{2}$$

Then the k^{th} pair of buckets will be updated by $(1 - d_f, d_f)$ and the rest pairs by $(0, 0)$. When the first geometry pass is over, the k^{th} pair of buckets will hold the minimum and maximum depth values in the k^{th} subinterval,i.e.,

$$dmin_k^1 = 1 - \max_{d_f \in [d_k, d_{k+1})} (1 - d_f), \quad dmax_k^1 = \max_{d_f \in [d_k, d_{k+1})} (d_f) \tag{3}$$

It is obvious that these fragments in the consecutive depth intervals are in correct depth ordering:

$$dmin_0^1 \le dmax_0^1 \le dmin_1^1 \le dmax_1^1 \le \cdots \le dmin_{15}^1 \le dmax_{15}^1 \tag{4}$$

If there is no fragment within the k^{th} subinterval, both $dmax_k^1$ and $dmin_k^1$ will remain the initial value 0 and can be omitted. And if there is only one fragment within the k^{th} subinterval, $dmax_k^1$ and $dmin_k^1$ will be equal and one of them can be eliminated. In a following fullscreen pass, the bucket array will be sequentially accessed as 8 input textures to get the sorted fragments for post-processing.

For applications that need other fragment attributes, taking order independent transparency as an example, we can pack the RGBA8 color into a 32-bit positive floating-point. The alpha channel will be mapped to the highest byte (bits 24-31) and scaled to [0,127] and the RGB channels will be mapped to the rest three bytes (bits 23-0) with each scaled to [0,255]. We then divide the depth range into 32 subintervals corresponding to the 32 buckets and capture the packed colors instead of the depth values in a similar way with MAX blending. In postprocessing, we can unpack the floating-point colors to RGBA8 in correct depth ordering for alpha blending.

For uniformly distributed scenes, most buckets will correspond to at most one fragment with few collisions, so the results will turn out to be a good approximation. However, for non-uniform scenes, collisions will happen more frequently especially on the silhouette or details of the model, and artifacts will arise. We can extend the algorithm to multi-pass approach for better results. In the second geometry pass, we allocate a new bucket array for each pixel and the bucket array captured in the first pass will be taken as 8 input textures. Those fragments with depth values $d_f \in [d_k, d_{k+1})$ satisfying condition $d_f \ge dmax_k^1$ or $d_f \le dmin_k^1$ have been captured in the previous pass, thus can be simply discarded. When the second pass is over, the k^{th} pair of buckets will hold the second minimal and maximum depth values $dmin_k^2$ and $dmax_k^2$ in the k^{th} subinterval. The depth values captured in these two passes are naturally in correct ordering:

$$dmin_0^1 \le dmin_0^2 \le dmax_0^2 \le dmax_0^1 \le dmin_1^1 \le dmin_1^2 \le dmax_1^2$$
$$\le dmax_1^1 \le \cdots \le dmin_{15}^1 \le dmin_{15}^2 \le dmax_{15}^2 \le dmax_{15}^1 \tag{5}$$

During post-processing, both bucket arrays can be passed to the pixel shader as input textures and accessed for rendering of multi-fragment effects.

Theoretically, we can obtain accurate results by enabling the occlusion query and looping in the same way until all the fragments have been captured. However, the sparse layout of depth values in the bucket arrays will lead to memory exhaustion especially for non-uniform scenes and high screen resolutions. Artifacts may also arise for the multi-pass approach when the packed color ordering is inconsistent with the correct depth ordering. Moreover, for applications that need multiple fragment attributes, such as the Fresnel's effect, simply binding the buckets into groups and updating each group simultaneously by the fragment attributes might result in undesirable mismatches. We instead propose a more robust scheme to alleviate these problems at the cost of an additional geometry pass, namely adaptive bucket depth peeling. The details will be described as follows.

3.3 Adaptive Bucket Depth Peeling

The uniform division of the depth range may result in some buckets overloaded while the rest idle for non-uniform scenes. Ideally, we prefer to adapt the division of subintervals to the distribution of the fragments per pixel, so that there is only one fragment falling into each subinterval. The one-to-one correspondence between fragments and subintervals will assure only one fragment for each bucket, thus can avoid the collisions.

Inspired by the image histogram equalization, we define a depth histogram as an auxiliary array with each entry indicating the number of fragments falling into the corresponding depth subinterval, thus is a probability distribution of the geometry. We allocate 8 MRT buffers with pixel format GL_RGBA32UI_EXT as our depth histogram. Considering each channel of the MRT as a vector of 32 bits, the depth histogram can be cast to a bit array of size 4*8*32=1024, with each bit as a binary counter for fragments. Meanwhile, the depth range is divided into 1024 corresponding subintervals: $[d_k, d_{k+1}), d_k = zNear + \frac{k}{1024}(zFar - zNear), k = 0, 1, \cdots, 1023$. The depth range is always on a magnitude of 10^{-1}, so the subintervals will be on a magnitude of 10^{-4}, which are often small enough to distinguish almost any two close layers. As a result, there is at most one fragment within each subinterval on most occasions, thus a binary counter for each entry of the depth histogram will be sufficient most of the time.

In the first geometry pass, an incoming fragment within the k^{th} subinterval will set the k^{th} bit of the depth histogram to 1 using the OpenGL's 32-bit logic operation GR_OR. After the first pass, each bit of the histogram will indicate the presence of fragments in that subinterval or not. A simplified example with depth complexity $N = 8$ (the maximum number of layers of the scene at all viewing angles) is shown in Figure 1. Suppose at a certain pixel location, the eye ray intersects the scene generating 4 fragments $f_0 - f_3$ within four different subintervals $[d_2, d_3], [d_8, d_9], [d_9, d_{10}], [d_{1022}, d_{1023}]$. They will set the 3rd, 9th, 10th, and the 1023rd bit of the depth histogram to 1 in the first geometry pass.

The depth histogram is equalized in a following fullscreen pass. For scenes with depth complexity N less than 32, the histogram is passed into the pixel shader as 8 input textures, and new floating point MRT buffers with N channels will be allocated as an equalized histogram for output. We can consecutively obtain the j^{th} bit

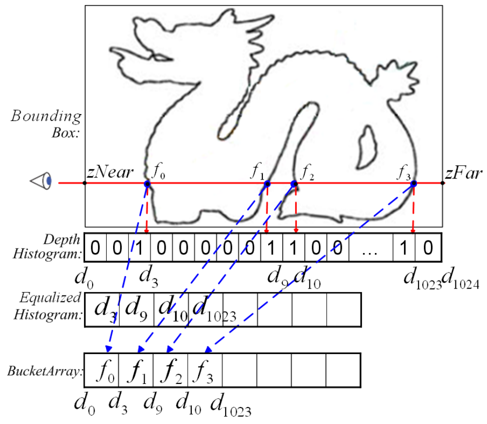

Figure 1: ***An Example of Adaptive Bucket Depth Peeling.*** *The red dash arrows indicate the operations in the first geometry pass, and the blue dash arrows indicate the operations in the second geometry pass.*

of the $i^{th}(i, j = 0, 1, 2, \cdots, 31)$ channel of the input depth histogram by a bitmask (equals to 2^j), using the logic AND operation. If the bit is zero, it means that there is no fragment falling into the $k^{th}(k = i * 32 + j)$ depth subinterval, thus can be simply skipped over; otherwise, there is at least one fragment within that subinterval, so we store the corresponding upper bound d_{k+1} consecutively into the equalized histogram for output. As for the example in Figure 1, two MRT buffers with 8 channels will be allocated as the equalized histogram, and the upper bounds d_3, d_9, d_{10}, and d_{1023} will be stored into it in the equalization pass.

We perform the bucket sort in the second geometry pass. The equalized histogram is passed to the pixel shader as input textures and a new bucket array of the same size N is allocated as output for each pixel. The upper bounds in the input equalized histogram will redivide the depth range into non-uniform subintervals with almost one-to-one correspondence between fragments and subintervals. As a result, there will be only one fragment falling into each bucket on most occasions, thus collisions can be reduced substantially. During rasterization, each incoming fragment with a depth value d_f will search the input equalized histogram (denoted as EH for short). If it belongs to the k^{th} subinterval, i.e., it satisfies conditions $d_f \geq EH[k-1]$ and $d_f < EH[k]$, it will be routed to the k^{th} bucket. In the end, the fragments are consecutively stored in the output bucket array, so our adaptive scheme will be memory efficient. As for our example in Figure 1, the upper bounds in the equalized histogram redivide the depth range into 4 subintervals: $[0, d_3), [d_3, d_9), [d_9, d_{10}), [d_{10}, d_{1023}]$. Fragment f_0 is within the first subinterval $[0, d_3)$, so it is routed to the first bucket. Fragment f_1 is within the second subinterval $[d_3, d_9)$, and is routed to the second bucket, and so on. After the second geometry pass, all of the four fragments are stored in the bucket array for further applications. The pseudocode of our adaptive scheme is listed in Table 1.

This adaptive scheme can reduce the collisions substantially, but collisions might still happen when two close layers of the model generate two fragments with a distance less than 10^{-4}, especially on the silhouette or details of the model. These fragments are routed to the same bucket and merged into one layer, thus result in artifacts. Theoretically, we can also resort to the multi-pass approach for better results.

For applications that need multiple fragment attributes, the one-one-correspondence between fragments and subintervals can assure

Algorithm: Adaptive Bucket Depth Peeling

```
Initialization
 1:  Enable(MAX_BLENDING);
 2:      $ZNearZFar \leftarrow$ DrawBoundingBox();
 3:  Disable(MAX_BLENDING);
Geometry Pass 1
 4:  Enable(LOGIC_OR);
 5:      $Histogram \leftarrow$ DrawModel($ZNearZFar$);
 6:  Disable(LOGIC_OR);
 7:  $EqualizedHistogram \leftarrow$ HistogramEqualization($Histogram$);
Geometry Pass 2
 8:  Enable(MAX_BLENDING);
 9:      $BucketArray \leftarrow$ DrawModel($EqualizedHistogram$);
10:  Disable(MAX_BLENDING);
11:  RenderScene($BucketArray$);
```

Table 1: *Pseudocode of our adaptive bucket depth peeling.*

the attributes consistency, so we can bind consecutive buckets into groups and update each group by the attributes simultaneously.

For scenes with more than 32 layers, we can handle the remaining layers by scanning over the remaining part of the histogram in a new fullscreen pass to get another batch of 32 nonzero bits. We then equalize it and pass the equalized histogram to the next geometry pass to route the fragments between layer 32 and 64 into corresponding buckets in the same way, and so on, until all the fragments are captured.

4 Results and Applications

We present the experimental results of our algorithm in comparison to the classical depth peeling, and k-buffer [Bavoil et al. 2007]. Our algorithm is implemented using OpenGL and Cg 2.1 shading language. All of our results are generated on a commodity PC of Intel Duo Core 2.4G Hz with 3GB memory, and NVIDIA Geforce 8800 GTX with driver 175.16.

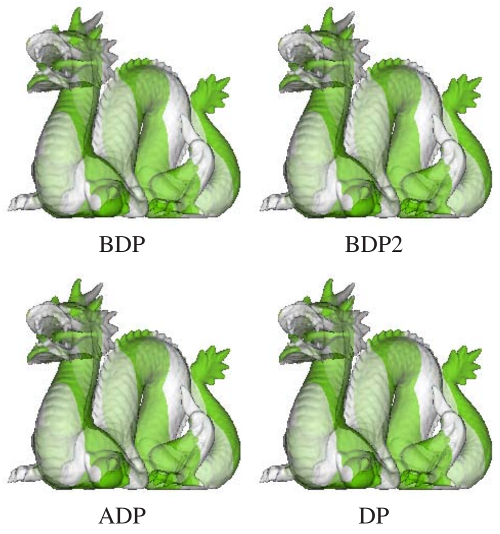

Figure 2: ***Transparent effect on Stanford Dragon (871K triangles).*** *The left top is rendered by BDP, the right top is by BDP2, the left bottom is by ADP, and the right bottom is the ground truth generated by DP.*

Transparency. Figure 2 shows the order independent transparency effect on Stanford Dragon rendered by our bucket depth peeling with a single pass (BDP) and its two-pass extension (BDP2) and the adaptive bucket depth peeling (ADP) in comparison to the classical depth peeling (DP). The differences between BDP and DP are visually unnoticeable, so one pass might be a good approximation when performance is more crucial, and two passes are preferred for nonuniform scenes with high depth complexity.

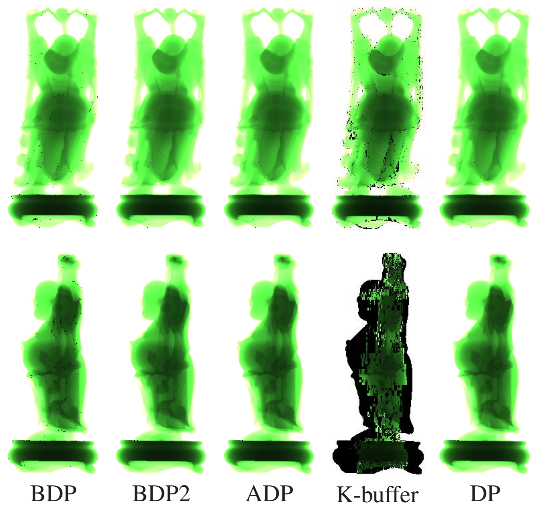

Figure 3: ***Translucent effect on Buddha (1,087K triangles).*** *The first column is rendered by BDP, the second and third are by BDP2 and ADP, the third is by the k-buffer of 16 layers without modifications, and the last one is the ground truth generated by DP.*

Translucency. The translucency effects can be rendered accounting only for absorption and ignoring reflection [NVIDIA 2005]. The ambient term I_a can be computed using Beer-Lambert's law: $I_a = \exp(-\sigma_t l)$, where σ is the absorption coefficient, and l is the accumulated distance that light travels through the material which can be approximated by accumulating the thickness between every two successive layers of the model per pixel. Figure 3 shows the translucent effect on Buddha using different methods. The results of k-buffer are obtained using the demo program on author's website. It is shown that for k-buffer the RMW hazards is severer on the side views with more layers, while in contrast, the collisions only affect our BDP on the sharp edges and detailed geometries of the model. The results produced by the single pass BDP provide a good approximation and the two-pass approaches are preferred for better visual quality.

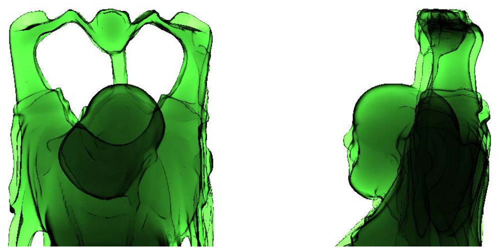

Figure 4: ***Fresnel's effect on Buddha(1,087K triangles) rendered by ADP.***

Taking into account the attenuation of rays (Fresnel's effect), Schlick's approximation can be used for fast calculation of Fresnel's transmittance of each fragment: $Ft = 1-(1-cos(\theta))^5$. Figure 4 shows the results of fresnel's effect rendered by ADP. In the second geometry pass, we transform the normal into the view space and pack it into a floating point. The buckets are grouped into pairs and each pair will be updated by the packed normal and the depth value simultaneously. In the deferred shading pass, the ambient term of each pixel can be obtained using Beer-Lambert's law. For a certain pixel, the eye direction can be restored by transforming the fragment position from the screen space back to the the view space. We then unpack the normal of each fragment and perform a dot product with the eye direction to get the incident angle θ on that surface. In the end, the Fresnel's transmittance of each fragment can be computed and multiplied together as the final attenuating factor to the ambient term on that pixel location. See more results generated by our techniques in Figure 5.

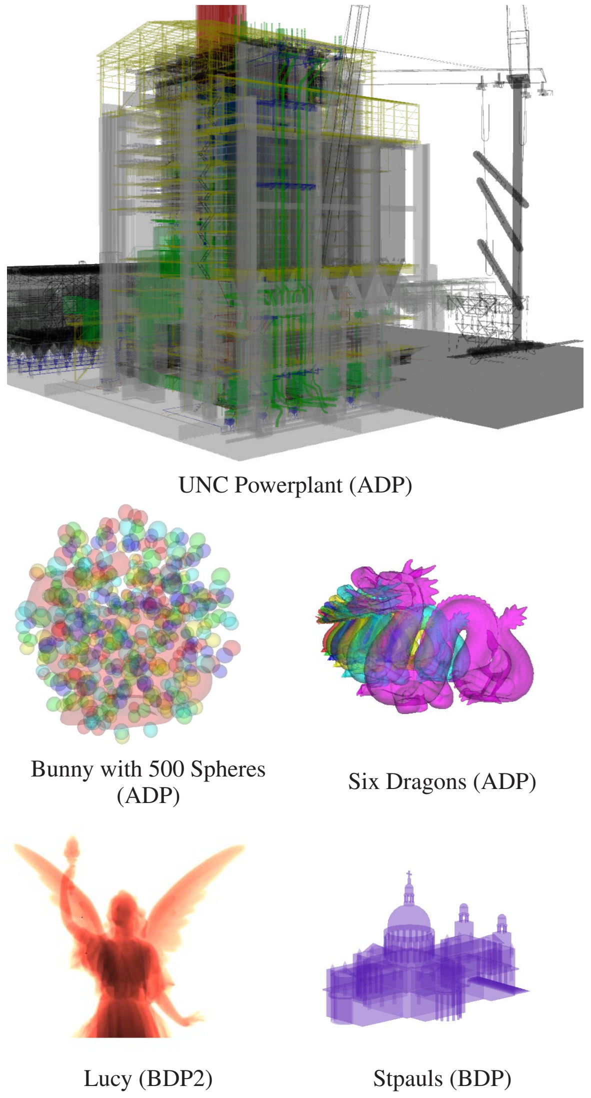

Figure 5: ***More results of our techniques.***

More applications such as constructive solid geometry (CSG), depth-of-field, shadow maps, refraction, and volume rendering will also benefit from our algorithms greatly in a similar way as [Eisemann and Décoret 2006; Bavoil et al. 2007].

Model	Dragon	Buddha	Powerplant	Lucy	Stpauls
Tri No.	871K	1,087K	12,748K	28,055K	14K
BDP	256fps	212fps	24.15fps	10.93fps	434fps
BDP2	128fps	106fps	12.79fps	5.71fps	256fps
ADP	106fps	91fps	12.31fps	5.37fps	212fps
K-buffer	206fps	183fps	23.98fps	10.49fps	468fps
[Liu 2006]	49fps	39fps	0.83fps	0.75fps	155fps
	5g	6g	27g	14g	22g
Dual DP	37fps	32fps	1.34fps	0.87fps	199fps
	8g	8g	16g	12g	14g
DP	24fps	20fps	0.76fps	0.54fps	242fps
	13g	13g	32g	21g	26g

Table 2: ***Frame rates(fps) and geometry passes(g) for various models with different methods.*** *All the scenes are rendered in a viewport of 512*512 by BDP, BDP2, ADP, k-buffer of 32 layers without modifications, the multi-pass extension of k-buffer described in [Liu et al. 2006], dual DP and DP .We only peel the first 16 and last 16 layers of powerplant by dual DP and the first 32 layers by other methods.*

5 Discussion

5.1 Performance Analysis

The performance of our algorithm is evaluated by comparison to k-buffer and its multi-pass extension [Liu et al. 2006], dual depth peeling, and the classical depth peeling on a set of common models. The analysis demonstrates that our algorithm produces high quality results comparable to the ground truth in great efficiency.

The linear complexity of bucket sort ensures our BDP work in $O(N)$ for uniformly distributed scenes with less than 32 layers. When the worst-case happens, i.e., most layers are clustered only into the first or the last depth intervals, multiple passes will be needed, and the performance will degrade to DP. In contrast, the ADP always run in $O(N)$ for arbitrary distributed scenes with less than 32 layers. Since two geometry passes are always needed, it will achieve a speed improvement up to 16 times.

Suppose the vertex rasterization needs time T_v for each geometry pass, and our fragment shader costs T_f, the accelerate rate is:

$$K = \frac{N \times T_v}{T_v + T_f} \tag{6}$$

For large-scale scenes with depth complexity N where $T_v \gg T_f$, such as powerplant and lucy, our BDP can get an accelerate rate of nearly N times to DP, and $N/2$ times to dual DP (See Table 2). It also outperforms the k-buffer and its multi-pass extension [Liu et al. 2006]. Both BDP2 and ADP generate better results than BDP but halve the speed, with ADP slightly slower because of the cost of histogram equalization. For simple scenes where the extra cost T_f on fragment shader can not be ignored, such as stpauls model, our algorithms may not perform as efficiently as large scenes, even slower than DP. However, all of other fragment level algorithms suffer from the same problem. So it will be more suitable for our algorithm to handle large-scale scenes with high depth complexity.

5.2 Quality Analysis

We perform an quality analysis of our algorithms by comparing the images generated by our algorithms to the ground truth generated by DP. Each pixel that has a different intensity with the corresponding location in ground truth image will be identified as an error pixel. Suppose at viewing angel θ, the total number of rendered pixels is $N_t(\theta)$ and the number of error pixels is $N_e(\theta)$, then the error pixel rate $EPR(\theta)$ can be computed as:

$$EPR(\theta) = N_e(\theta)/N_t(\theta) \tag{7}$$

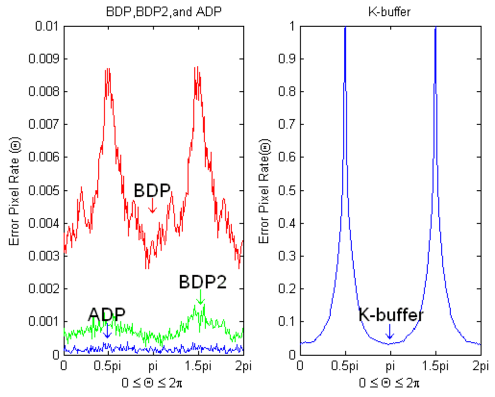

Figure 6: ***Error pixel rate of translucent effect on Buddha with different methods.*** *The results are sampled by 200 viewing angles horizontally around the model at a resolution 512*512. The number of rendered pixels N_t at each angle ranges around 30,000.*

Figure 6 illustrates the error pixel rate of different methods in rendering translucent effect on buddha. The *EPR* of k-buffer reaches its peaks of nearly 100% at the side views when the depth complexity is high. In comparison, the *EPRs* of BDP and BDP2 are two orders of magnitude smaller than that of k-buffer. And the *EPR* of ADP in contrast remains around a quite small constant of about 0.01%.

5.3 Limitation

Our algorithm produces a great speedup to the classic depth peeling especially for large scenes. It turns out to be a good approximation when performance is critical and the visual quality is less important; otherwise, we can resort to multi-pass approach for better results. However, we have to allocate new MRTs as bucket array in each additional pass, which might result in memory exhaustion, so the two-pass approach might be preferred as a compromise between visual quality and memory requirement. Meanwhile, the additional MRT buffers for histogram in our adaptive scheme will also be a non-trivial memory overhead especially for applications with high screen resolutions. When future graphics hardware is equipped with large amount of memory, the problem might be alleviated.

6 Conclusions and Future Work

In this paper, we provide a framework of bucket depth peeling, a highly efficient multi-layer depth peeling algorithm via bucket sort on GPU especially for large-scale scenes with high depth complexity. We describe the implementation of the bucket array on GPU, and perform a bucket sort on the fragment level. Collisions can be alleviated by multi-pass approach. We also introduce an efficient adaptive scheme to further reduce collisions. Experimental results show great credit for the performance and quality advantages of our approach.

In the future, we are interested in forming more efficient schemes to reduce collisions further more. For example, a depth histogram of two levels of details might be constructed, with the coarse level of 32 bins obtained by rendering the visual hulls first, then the precise level can be organized non-uniformly by skipping over those empty bins and allocating more bits for full bins at the coarse level. In addition, we will try to solve the memory problem of BDP by composing the fragments captured within each bucket per pass, and finally composing all the buckets after all the fragments have been captured.

Acknowledgments. The authors would like to thank anonymous reviewers for their constructive comments and suggestions. This work has been supported by National Basic Research Program of China (973 Program) (Grant No. 2009CB320802), National 863 High-Tec. Project (Grant No. 2008AA01Z301), National Science Foundation of China (Grant No. 60573155) and University of Macau Research Grant.

References

BAVOIL, L., AND MYERS, K. 2008. Order independent transparency with dual depth peeling. Tech. rep., NVIDIA Corporation.

BAVOIL, L., CALLAHAN, S. P., LEFOHN, A., AO L. D. COMBA, J., AND SILVA, C. T. 2007. Multi-fragment effects on the gpu using the k-buffer. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, 97–104.

CARPENTER, L. 1984. The a-buffer, an antialiased hidden surface method. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 103–108.

CARR, N., MECH, R., AND MILLER, G. 2008. Coherent layer peeling for transparent high-depth-complexity scenes. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 33–40.

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

EISEMANN, E., AND DÉCORET, X. 2006. Fast scene voxelization and applications. In *SIGGRAPH 2006 Sketches*.

EVERITT, C. 2001. Interactive order-independent transparency. Tech. rep., NVIDIA Corporation.

GOVINDARAJU, N. K., HENSON, M., LIN, M. C., AND MANOCHA, D. 2005. Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, 49–56.

HOUSTON, M., PREETHAM, A., AND SEGAL, M. 2005. A hardware f-buffer implementation. Tech. rep., Stanford University.

JOUPPI, N. P., AND CHANG, C.-F. 1999. z^3: an economical hardware technique for high-quality antialiasing and transparency. 85–93.

LIU, B.-Q., WEI, L.-Y., AND XU, Y.-Q. 2006. Multi-layer depth peeling via fragment sort. Tech. rep., Microsoft Research Asia.

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

MARK, W. R., AND PROUDFOOT, K. 2001. The f-buffer: a rasterization-order fifo buffer for multi-pass rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 57–64.

MYERS, K., AND BAVOIL, L. 2007. Stencil routed a-buffer. *ACM SIGGRAPH 2007 Technical Sketch Program.*

NVIDIA. 2005. Gpu programming exposed: the naked truth behind nvidia's demos. Tech. rep., NVIDIA Corporation.

WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. 2005. Gpu-accelerated high-quality hidden surface removal. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 7–14.

WITTENBRINK, C. M. 2001. R-buffer: a pointerless a-buffer hardware architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 73–80.