Bucket Depth Peeling

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

Figure 1: (a) Transparent effect on Stanford Dragon by UP1, 871K triangles. (b) Translucent effect on Lucy by UP2, 28.0M triangles. (c) Fresnel’s effect on Buddha by AP2, 1.0M triangles. (d) Transparent effect on the first 32 layers of UNC Powerplant by AP2, 12.7M triangles.

1 Introduction

Efficient rendering of multi-fragment effects has long been a great challenge in computer graphics. The classical depth peeling algorithm [Everitt 2001] provides a simple but robust solution by peeling off one layer per pass, but multi rasterizations will lead to performance bottleneck for large and complex scenes. The k-buffer [Bavoil et al. 2007] captures k fragments in a single pass but suffers from read-modify-write (RMW) hazards which can be alleviated by multi passes [Liu et al. 2006]. Our approach exploits multiple render targets (MRT) as bucket array per pixel. Fragments are scattered into different buckets and sorted by a bucket sort. We describe two efficient schemes to reduce collisions when multiple fragments are routed to the same bucket. Our algorithm shows up to 32 times speedup to depth peeling especially for large scenes and the results are visually faithful. Also it has no requirement of pre-sorting geometries or post-sorting fragments, and is free of RMW hazards.

2 Bucket Depth Peeling

Modern GPUs can support up to 8 MRTs with format RGBA32F, thus a bucket array of size 32 can be constructed per pixel. To guarantee correct concurrent updates of buckets, we utilize MAX blending, which compares the source and the destination values to keep the greater one. Each bucket is initialized to 0 so that the first update of any bucket will always succeed. When collision happens, the maximum fragment will survive finally. As for other buckets, we simultaneously update them by 0 to keep their values unchanged. This atomic operation assures the correct result and avoids RMW hazards. We then describe two schemes to alleviate collisions.

- Uniform bucket depth peeling. A bounding box or a coarse visual hull is first rendered to get the approximate depth range \([z_{Near}..z_{Far}]\) of each pixel. Then we group consecutive buckets into pairs and divide the depth range into 16 uniform subintervals. An incoming fragment with a depth value within the \(k_{th}\) subinterval will update the \(k_{th}\) pair of buckets by \((1-d_f, d_f)\) and the rest by 0. After the first pass, the minimum and maximum fragment depth values within the \(k_{th}\) subinterval \(d_{min_k}\) and \(d_{max_k}\) can be obtained from the \(k_{th}\) pair, with correct depth ordering: \(d_{min_0} \leq d_{max_0} \leq d_{min_1} \leq d_{max_1} \leq \cdots \leq d_{min_15} \leq d_{max_15}\). Other fragment attributes such as color can be packed into a floating point and captured in a similar way. For complex scenes, multi-pass approach would be a natural extension to further reduce collisions.

- Adaptive bucket depth peeling. We consider each channel of MRT as a vector of 32 bits and construct an array of size 1024 called depth histogram. The depth range is divided into 1024 subintervals which are tiny enough to distinguish almost any two close layers. In the first pass, fragments within the \(k_{th}\) subinterval will set the \(k_{th}\) bit to 1 to indicate their presence in that subinterval. The histogram is equalized in a fullscreen pass by searching for the nonzero bits and the corresponding upper bounds will be stored. In the second pass, these upper bounds will redivide the depth range into 32 subintervals and the fragments will be routed into corresponding buckets. The one-to-one correspondence between fragments and subintervals reduces most collisions and makes full use of the bucket array.

3 Results

<table>
<thead>
<tr>
<th>Model</th>
<th>Dragon</th>
<th>Buddha</th>
<th>Lucy</th>
<th>Powerplant</th>
</tr>
</thead>
<tbody>
<tr>
<td>UP1</td>
<td>215fps</td>
<td>212fps</td>
<td>10.93fps</td>
<td>24.15fps</td>
</tr>
<tr>
<td>UP2</td>
<td>128fps</td>
<td>106fps</td>
<td>5.71fps</td>
<td>12.79fps</td>
</tr>
<tr>
<td>AP2</td>
<td>106fps</td>
<td>91fps</td>
<td>3.57fps</td>
<td>12.31fps</td>
</tr>
<tr>
<td>[Liu 2006]</td>
<td>49fps/5g</td>
<td>39fps/6g</td>
<td>0.75fps/14g</td>
<td>0.83fps/2/g</td>
</tr>
<tr>
<td>DP</td>
<td>24fps/13g</td>
<td>20fps/13g</td>
<td>0.54fps/21g</td>
<td>0.76fps/32g</td>
</tr>
</tbody>
</table>

Table 1: Comparison of frame rates (fps) and geometry passes (g).

Figure 1 shows the results generated on Geforce 8800 GTX using the uniform scheme with a single pass (UP1) and two passes (UP2), and the adaptive scheme (AP2). Table 1 compares our performances with that of [Liu et al. 2006] and depth peeling (DP) at 512x512 resolution. For large scenes with N layers, UP1 can achieve nearly \(N\) times (up to 32) speedup since the cost of fragment shader is negligible. Both UP2 and AP2 produce better results at the cost of an extra pass. Our algorithm is more appropriate to handle large scenes with high depth complexity.

References

