Multi-Layer Depth Peeling via Fragment Sort

Baoquan Liu Li-Yi Wei Ying-Qing Xu

Microsoft Research Asia
MSR-TR-2006-81

Abstract

We present an accelerated depth peeling algorithm for order-independent transparency rendering on graphics hardware. Unlike traditional depth peeling which only peels one layer of transparent pixels per rendering pass, our algorithm peels multiple layers simultaneously per rendering pass. Our acceleration is achieved via our fragment program which sorts and writes multiple fragment colors and depths via MRT. A notable feature of our algorithm is that it is robust against the unreliable parallel read-after-write behavior in current graphics hardware, guaranteeing correct transparency ordering. For ordinary scenes rendered under RGBA8 color precision, we achieve up to 8× speed-up over conventional depth peeling with current generation graphics hardware. Our algorithm is simple to implement on current GPU without any hardware modification. In addition, it does not require applications to perform any pre-sorting of transparent geometry.

Keywords: order-independent transparency, depth peeling, multiple render target, concurrent read/write hazards, sorting, graphics hardware

1 Introduction

We present an algorithm for order-independent transparency rendering on commodity graphics hardware. Our algorithm is up to 8 times faster than traditional depth peeling [Everitt 2001], a simple and yet robust algorithm for rendering transparency for real-time applications. Similar to [Everitt 2001], our algorithm does not require hardware modifications and does not require applications to pre-sort transparent geometry.

The price we pay for such acceleration is several MRT (multiple-render-target) buffers, as well a more complex fragment program for sorting multiple layers of transparent fragments.

The basic idea of our algorithm is to perform M con-current depth peeling via MRT, where the M transparent layers are sorted via our fragment program. Since M is a constant, our fragment program can be implemented with great efficiency. The theoretical speed improvement by our algorithm is M; however, due to the unreliable read-after-write behavior in current hardware implementation of MRT (e.g. multiple transparent fragments written to the same pixel in a single rendering pass), the performance of our algorithm degrades slightly. However, our algorithm guarantees correct final transparency rendering even under this uncertainty.

Our algorithm is easy to implement, and provides significant speed improvement for rendering order-independent transparency in real-time applications.

1.1 Background

Since the conceptual introduction of alpha compositing by [Porter and Duff 1984], transparent objects has become an indispensable component for real-time graphics for either native transparency [Kelley et al. 1994; Diefenbach and Badler 1997; Akenine-Moller and Haines 2002] or volumetric effects [Weiler et al. 2002]. The major challenge for rendering transparency is that, unlike opaque objects which only the front-most one matters (a minimum-finding problem), correct rendering of transparent objects requires sorting in order to achieve the correct compositing effect; unfortunately, efficient sorting is difficult to implement on SIMD graphics hardware.

One simple method is to require applications to perform sorting. However, this can be time consuming and tedious, since the sorting needs to be done every time the view point or scene geometry change. In particular, application sorting could not take advantage of the massive computation power of z-buffer graphics hardware.

There exists a variety of techniques that exploits commodity graphics hardware for sorting transparency without requiring any hardware modifications. Among these techniques, depth peeling [Everitt 2001; Mammen 1989] has attracted significant interests due to its simplicity and robustness. The technique requires $O(N)$ rendering passes to render order-independent transparency with depth complexity N. Each render pass requires pass downing all the transparent geometry, with a resulting total time complexity $O(N^2)$. [Wexler et al. 2005] improves the time complexity of depth peeling from quadratic to linear, but requires the application to sort scene objects into depth batches.

A variety of techniques have been proposed to avoid sending geometry multiple times by re-circulating [Wittenbrink 2001] or delaying [Aila et al. 2003] transparent fragments. However, these techniques require adding features not available in current generation commodity graphics hardware.

1.2 Our Contribution

Our major contribution is the core concept allowing multiple concurrent and independent depth peelings via a simple insertion sort performed in fragment program. In particular, our algorithm guarantees correct result even under the unreliable concurrent read-after-write behavior in current graphics hardware. Despite the popularity of depth peeling, so far it has been restricted to peeling only one layer per rendering pass; to our knowledge, we are the first to allow parallel, multi-layer depth peeling with guaranteed correct result.

We believe our algorithm provides immediate benefit for interactive and real-time applications community.

2 Algorithm

Since our algorithm extends from depth peeling [Everitt 2001], for clarity, we begin with a brief summary of [Everitt 2001]. We then describe our extensions and improvements. For clarity, we summarize our algorithm in Table 1 with a simple example illustrated in Figure 1.

2.1 Brief Review of Depth Peeling

Essentially, depth peeling renders transparent fragments in the correct order similar to selection sort. The algorithm employs multi-pass rendering, and within i^{th} pass the i^{th} layer (nearest to the eye) is selected and rendered. The selection is controlled by proper z-range culling; specifically at each rendering pass the depth value of the recently rendered layer is recorded, preventing already rendered

fragments from being rendered again in the next pass. The peeling process stops until no more transparent fragments are rendered (this can be determined by occlusion query [Craighead 2002]). [Everitt 2001] provides more detailed implementation and extensions of depth peeling on GPU, but in general the algorithm needs $O(N)$ rendering passes where N is the number of layers of transparent fragments in the scene. Each pass requires a complete geometric transformation and rasterization of all transparent polygons.

2.2 Our Approach

The major disadvantage of depth peeling is the need for $O(N)$ rendering passes; in particular, each pass would require an occlusion query, which needs to wait for pipeline flushing.

In our approach, we provide a simple acceleration that reduces the required number of rendering passes up to a constant number M; even though this does not reduce the asymptotic time complexity of the algorithm, our approach does reduce the actual computation time significantly for real scenes. Our speed factor M depends on the frame-buffer color precision and number of MRT (multiple-render-target) used; in particular, $M == 8$ when using 4 MRT with RGBA8 color precision, providing $8\times$ speedup for transparency rendering, which is significant for real-time applications like gaming.

Our algorithm achieves at this speedup at the expense of several color+depth MRT buffers. Basically, we peel M transparent layers in one pass, storing color+depth for the i^{th} layer in MRT bucket i. (Each pass requires an occlusion query similar to [Everitt 2001].) Since M is a constant, we can achieve this via a simple insertion sort inside our fragment program. Assuming no concurrent read/write hazards, our algorithm would perfectly peel M layers per rendering pass. However, since this may not happen (see Figure 1 for examples), we might need multiple passes to correctly to peel off the current M layers. Fortunately, assuming that a fragment writes to MRT buffers are atomic (i.e. a fragment write to multiple buffers either all succeed or all fail), our algorithm is guaranteed to produce at least one more correct element per pass, so it will (1) terminate and (2) produce a final correct result.

After the current M layers are peeled, we composite the color results from the M MRT, and update the z-near buffer so that we could perform proper z-range culling to process the next M layers.

Below, we describe our algorithm in detail, as summarized in Table 1. A simple example is also illustrated in Figure 1. The algorithm presented below operates per frame.

Initialization Similar to depth peeling [Everitt 2001], we first render opaque objects into the color and z-far buffers. In addition, we clear the value of the z-near buffer to some minimum value. The z-near and z-far buffers define the proper z-range for the current active M transparent layers; specifically, fragments outside this range are culled away.

MRT Buffer Encoding Given M_{rt} MRT buffers with B_m bytes per pixel, we could store M buckets worth of color (with B_c bytes) and depth (with B_d bytes) values:

$$M = \frac{M_{rt} \times B_m}{B_c + B_d} \tag{1}$$

For example, given $M_{rt} = 4$ MRT with $B_m = 16$ bytes per pixel ($4\times$FP32), we can encode $M = 16$ buckets of $B_d = 4$ FP32 depth values, or $M = 8$ buckets of $B_c = 4$ RGBA8 color plus $B_d = 4$ FP32 depth values.

In our current implementation, we utilize $M = 8$ with $M_{rt} = 4$, $B_m = 16$, $B_c = 4$, and $B_d = 4$, but these numbers can be changed based on specific application needs and available hardware resources.

```
Initialization
  render all opaque objects into color and z-far buffers
  z-near buffer ← z-min
Transparency Rendering
  done_entire_scene ← false
  while(not done_entire_scene)
    done_entire_scene ← true
    done_peeling ← false
    clear MRT buffers
    while(not done_peeling)
      begin-occlusion-query
      depth peeling all transparent geometry
      // culled via z-near and z-far buffers
      // first M non-culled layers sorted in pixel shader
      end-occlusion-query
      done_peeling ← get-occlusion-query == 0
      if(not done_peeling) done_entire_scene ← false
    end while
    composite M MRT results into color buffer
    z-near buffer ← z values from the last MRT
  end while
```

Table 1: *Pseudocode of our algorithm. M is the maximum number of simultaneous depth peelings allowed by the color precision and number of MRT buffers used. The entire process is repeated for each frame.*

Transparency Rendering We now render the transparent objects and composite them in the correct order (front-to-back), regardless of their actual sequence of arrival. We achieve this by simultaneously peeling the front-most M active transparent layers by insertion sort in our fragment program, which writes the color and depth values of the i^{th} layer into the i^{th} MRT bucket.

Due to the concurrent read/write hazards as illustrated in Figure 1, our algorithm may require multiple passes to correctly render the current M layers. Assuming that fragment writes to multiple MRT are atomic (i.e. all succeed or all fail), then it can be easily shown that our algorithm will produce at least one more correct result per pass; as a result, our algorithm takes at most M passes (reducing to traditional depth peeling) and more importantly, the result is guaranteed to be correct.

Note that if the atomic write assumption fails, our algorithm might fall into an infinite loop, as illustrated in Figure 2. However, since we have never experienced such race condition during our extensive experiments, we believe our atomic commitment assumption holds true on real hardware.

After finish peeling the current M active voxels, we composite the results in the M MRT buckets into the final color/depth frame-buffers. We then update z-near buffer from the farthest MRT buffer, so that later on we will not touch layers already rendered. We iterate this process until the entire scene is rendered.

3 Implementation Details

We now describe further implementation details that are essential for achieving full efficiency and quality of our algorithm.

3.1 Stop criteria

As shown in Table 1, our algorithm requires two stop criteria, corresponding to the peeling of active layers (inner **while** loop via variable *done_peeling*) and the termination of the entire algorithm (outer **while** loop via variable *done_entire_scene*).

The termination of the inner loop can be easily determined via occlusion query similar to [Everitt 2001]; specifically, when no

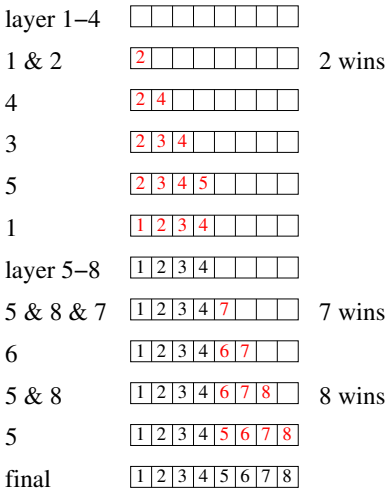


Figure 1: *A simple example illustration of our algorithm. Time progresses from top to bottom. Here we are allowed $M = 4$ con-current depth peelings. Notice the possibility of concurrent fragment read/write as illustrated.*

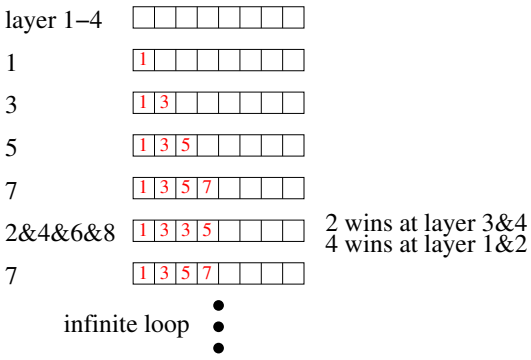


Figure 2: *A simple example illustration potential race condition. This illustration is similar to Figure 1, except that due to the possibility of non-atomic write (where both fragment 2 and 4 commits half of their target), the algorithm might run into a cycle. Note that in this example we assume the duplicated fragment 3 is cleaned out by the fragment program; if not, the final result will be incorrect.*

more transparent fragments are rendered in the current depth peeling pass, it signals that we have completed processing the current M active layers and are ready to move on to process layers further away.

The termination of the outer loop is trickier. Essentially, we can stop the algorithm when no transparent fragments exist within the current z-range defined by z-near and z-far buffers. Intuitively, this situation happens if and only if the GPU renders zero fragments after the last MRT compositing and z-near updating (as shown in the last steps in Table 1). This fact can be determined through the occlusion query already performed for the inner loop, as shown in the assignment of the *done_entire_scene* variable. In particular, once the entire scene is rendered, we will need only one more occlusion query to quit the entire process.

3.2 Fragment program

Since current FBO does not support multiple z-buffers, we simply disable all z testing and instead simulate z comparison in our fragment program, as shown in Table 2. Specifically, we write fragment z values as 32-bit floating point color values into MRT, and perform z-comparison by reading back the stored z-values inside our fragment program.

```
void main(
          in  float4 wpos      : WPOS,
          in  float4 color      : COLOR,
          uniform float4 layer,//tell us if this is the first time
          out float4    color_out0  : COLOR0,// output result 0
          out float4    color_out1  : COLOR1,// output result 1
          out float4    color_out2  : COLOR2,// output result 2
          out float4    color_out3  : COLOR3,// output result 3
          uniform samplerRECT bufferMap0: TEXUNIT0,
          uniform samplerRECT bufferMap1: TEXUNIT1,
          uniform samplerRECT bufferMap2: TEXUNIT2,
          uniform samplerRECT bufferMap3: TEXUNIT3,
          uniform samplerRECT LastZTex  : TEXUNIT4
          )
{
    if(round(layer.w)!=0)
    {//if first time, no need to compare with pre z
        float lastZ=texRECT(LastZTex, (wpos.xy)).a;//low
        if(wpos.z<=lastZ)
            discard;
    }
    float fp32color=pack_4ubyte(color);//4 ubyte to a float
    float4 ReadTarget[4];
    ReadTarget[0] = texRECT(bufferMap0, wpos.xy);
    ReadTarget[1] = texRECT(bufferMap1, wpos.xy);
    ReadTarget[2] = texRECT(bufferMap2, wpos.xy);
    ReadTarget[3] = texRECT(bufferMap3, wpos.xy);
    float2 frag_CZ[8];//color and Z
    int k=0;int i=0;
    for(i=0;i<4;i++)
    {
        frag_CZ[k]=ReadTarget[i].rg;
        k+=1;
        frag_CZ[k]=ReadTarget[i].ba;
        k+=1;
    }
    float temp=1;
    for(i=0;i<8;i++)
    {
        temp*=(wpos.z-frag_CZ[i].y);
    }
    if(temp==0)//current fragment is in the buffer already
        discard;
    int max_ind=-1;
    float maxz=-1;
    for (i = 0; i < 8; i++) //find max value
    {
        if(frag_CZ[i].y>maxz)
        {
            maxz=frag_CZ[i].y;
            max_ind=i;  //index
        }
    }
    if(wpos.z>=maxz)
        discard;
    else
    {
        frag_CZ[max_ind].x=fp32color;   //color
        frag_CZ[max_ind].y=wpos.z;  // Z
    }
    color_out0.rg=frag_CZ[0];   //write output
    color_out0.ba=frag_CZ[1];
    color_out1.rg=frag_CZ[2];
    color_out1.ba=frag_CZ[3];
    color_out2.rg=frag_CZ[4];
    color_out2.ba=frag_CZ[5];
    color_out3.rg=frag_CZ[6];
    color_out3.ba=frag_CZ[7];
}
```

Table 2: *Our fragment shader in HLSL.*

4 Results

Here we discuss our quality and performance results; all rendering and timing measurements shown below are performed on a NVIDIA Geforce 6800 card.

Ground truth verification with depth peeling

We have performed a variety of directed and random tests to ensure that our algorithm indeed produces identical results with respect to [Everitt 2001], which we consider as ground truth. Our

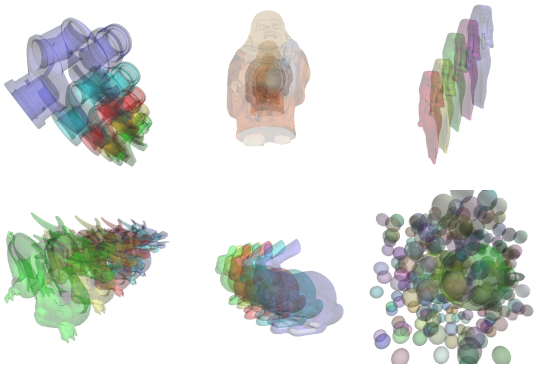
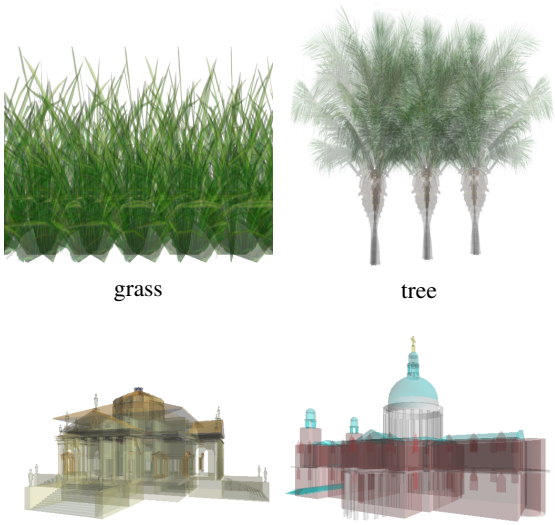


Figure 3: *Comparison with ground truth. The first 5 images demonstrate directed tests via objects with different geometry/topology, while the bottom-right image demonstrates random test.*

tests cover a variety of objects with different shape and complexity, mixed and arranged under different numbers and configurations, and viewed from different angles. Screen shots of a small subset of our tests are demonstrated in Figure 3. In all these tests, we have verified that our algorithm produces identical results with respect to the ground truth, indicating that our atomic write assumption might indeed hold true in real graphics chips.

Transparency rendering with real scenes

Figure 4 demonstrates transparency rendering for a variety of real scenes. In all these cases, our algorithm produces identical images with [Everitt 2001]. In addition, our algorithm achieves faster performance especially for scenes with greater depth complexity of transparent geometry.



Scene	# poly	# depth	our (fps)	[Everitt 2001] (fps)
grass	384	20	2.11	1.09
tree	384	20	2.23	1.18
villy	15472	31	4.51	2.62
stpaul	14780	23	4.72	3.15

Figure 4: *Transparency rendering for a variety of real scenes. # poly indicates the number of scene polygons, and # depth measures the maximum number of transparent layers.*

5 Conclusions and Future Work

We have presented an algorithm for depth peeling multiple layers of transparent pixels per rendering pass at the expense of multiple MRT buffers. Our core innovation is sorting multiple transparent layers via GPU fragment program, and the robustness of our algorithm under unreliable concurrent read/write hazards in current generation graphics chips. Despite the simplicity of our algorithm, it provides practical benefit for speeding up transparency rendering. In addition, the approach is easy to integrate with application pipelines as we require neither hardware modification nor application sorting of transparent geometry.

Currently, the speed of our algorithm degrades by the unreliable concurrent read/write in MRT. If future graphics hardware guarantees correct such read-after-write behaviors, the speed of our algorithm will be automatically and significantly improved. Unfortunately, since the future trend for GPU is wider and deeper pipelines, supporting this feature would incur significant performance cost. As a result, we envision alternative future hardware modification, instead of depth-peeling, a more viable approach for supporting order-independent transparency.

References

AILA, T., MIETTINEN, V., AND NORDLUND, P. 2003. Delay streams for graphics hardware. *ACM Trans. Graph. 22*, 3, 792–800.

AKELEY, K. 1993. Reality engine graphics. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, 109–116.

AKENINE-MOLLER, T., AND HAINES, E. 2002. *Real-Time Rendering (2nd Edition)*. AK Peters.

CARPENTER, L. 1984. The a -buffer, an antialiased hidden surface method. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 103–108.

CRAIGHEAD, M., 2002. Nv_occlusion_query. `http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt`.

DIEFENBACH, P. J., AND BADLER, N. I. 1997. Multi-pass pipeline rendering: realism for dynamic environments. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, 59–ff.

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

KELLEY, M., GOULD, K., PEASE, B., WINNER, S., AND YEN, A. 1994. Hardware accelerated rendering of csg and transparency. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, 177–184.

MAMMEN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Comput. Graph. Appl. 9*, 4, 43–55.

PORTER, T., AND DUFF, T. 1984. Compositing digital images. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 253–259.

WEILER, M., KRAUS, M., AND ERTL, T. 2002. Hardware-based view-independent cell projection. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, 13–22.

WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. 2005. Gpu-accelerated high-quality hidden surface removal. In *HWWS '05: 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 *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 73–80.