Graphics Hardware (2008)
David Luebke and John D. Owens (Editors)

Coherent Layer Peeling for Transparent High-Depth-Complexity Scenes

Nathan Carr   Radomír Měch   Gavin Miller

Adobe Systems Inc.

Abstract

We present two new image space techniques for efficient rendering of transparent surfaces that exploit partial ordering in the scene geometry. The first technique, called hybrid layer peeling, combines unordered meshes with ordered meshes in an efficient way, and is ideal for scenes such as volumes with embedded transparent meshes. The second technique, called coherent layer peeling, efficiently detects and renders correctly sorted fragment sequences for a given pixel in one iteration, allowing for a smaller number of passes than traditional layer peeling for typical scenes. Although more expensive than hybrid layer peeling by a constant factor, coherent layer peeling applies to a broader class of scenes, including single meshes or collections of meshes. Coherent layer peeling does not require costly clipping or perfect sorting. However, the performance of the algorithm depends on the degree to which the data is sorted. At best, when the data is perfectly sorted, the algorithm renders a correct result in a single iteration. At worst, when the data is sorted in reverse order, the algorithm mimics the performance of layer peeling but with a higher cost per iteration. We conclude with a discussion of a modified form of coherent layer peeling designed for an idealized rasterization architecture that would match layer-peeling in the worst case, while still exploiting correctly sorted sequences when they are present.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

Efficient rendering of transparent surfaces has long been a problem in computer graphics. Classically such rendering has been handled by a well known technique known as the *painter's algorithm*. The painter's algorithm starts by sorting geometry in a back-to-front manner as viewed from the camera. The furthest away surfaces are drawn first and each successive transparent layer can be blended over the top, much the same way a painter goes about forming a painting. Two primary drawbacks exist with this approach. Firstly, sorting is required on the surfaces and secondly, geometry must be split when either geometric intersections happen or *visibility cycles* arise during sorting. The latter of the two drawbacks can make this process particularly expensive. The painter's algorithm can be classified as an *object level* approach to the rendering of transparent surfaces since it works by sorting geometric primitives such as triangles.

An alternative class of algorithms work at the *pixel level*. These algorithms work by sorting surfaces viewed through each pixel center. Probably the most well known pixel level technique is *layer peeling* [Mam89, Eve01]. Layer peeling works by rendering the geometry multiple times to accumulate a final result. Each iteration of rendering peels off a single surface depth layer visible through each pixel. The core algorithm has the distinct advantage that it does not require the geometry to be sorted up front. Furthermore, geometric splitting is not needed as in the case of object level transparency techniques. Layer peeling, however, does require as many iterations as the worst pixel's transparent depth complexity. Given n transparent surfaces viewed through a pixel, the worst case algorithmic complexity of layer peeling is $O(n^2)$, since n peels are required and all n surfaces are rendered on each peel.

Our paper presents two new pixel level techniques for transparent surface rendering. They are both enhancements of layer peeling that exploit ordered structure in the geometry. One algorithm is ideal for combining unsorted meshes

© The Eurographics Association 2008.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

with sorted meshes. The second algorithm exploits correctly sorted sequences of layers at a given pixel for a partially sorted collection of meshes. The combined approach of approximate model-space sorting with enhanced peeling logic per pass enables dramatic speed-ups for high-depth complexity scenes. In closing we provide inspiration for hardware designers by detailing an even more efficient algorithm with only minimal changes to existing hardware.

2. Previous Work

Much research has gone into improving the performance of traditional peeling. Wexler et al. [WGER05] noted that the asymptotic complexity of layer peeling in many cases can be reduced to $O(n)$ by decomposing the problem into smaller sub-problems. They start by placing their objects in a heap effectively sorting them in $O(n \lg n)$ from front-to-back. A small fixed number of objects are extracted from the heap in sets to form a set of batches. If the depth range of each batch does not overlap then classic layer peeling is performed on each batch and compositing (front-to-back) is done to accumulate the solution. In the case of overlapping batches, clipping planes are used to define a z-range for the current batch. Any object (even objects not in the batch) that span this range must also be included when processing the current batch. This algorithm works well when batches do not overlap each other much in depth. Since perfect object level sorting is not required by this algorithm, we note that heap sort could be replaced with a linear time bucket sort to achieve a running time of $O(n)$ for many scenes.

Another approach is to perform object level sorting using the graphics hardware and the occlusion query extension [GHLM05]. Such an approach works well when the data can be properly ordered, however, in the presence of visibility cycles, clipping is still necessary to resolve surface ordering. Numerous extensions to hardware have been proposed to allow more efficient handling of transparent surfaces. Aila et al. proposed the idea of delay streams [AMN03] to improve the efficiency of handling transparent surfaces in graphics hardware.

Another approach is to store lists of fragments that arrive at a given pixel along with their colors and depths. Mark et al. [MP01] introduced the concept of the F-buffer which captures streams of fragments through each pixel in an intermediate representation. The ordering of these fragments can be resolved in additional passes. Houston et al. [HPS05] developed a practical hardware implementation of the F-buffer. Bavoil et al. [BCL*07] detailed a more restricted form of this concept they refer to as the *k*-buffer. The *k*-buffer restricts the size of fragments that can be stored at each pixel and it provides a limited read-modify-write operation to this list. Another alternative is to use the existing multi-sampling feature of graphics hardware along with stencil routing to store lists of fragments per pixel [MB07]. By ignoring read-write hazards, Liu et al. [LWX06] was able to use multiple render targets simultaneously bound as textures to implement a *k*-buffer strategy.

Section 3 introduces hybrid layer peeling. Section 4 describes an overview of coherent layer peeling. Section 5 illustrates how it may be implemented on a legacy GPU with stencil and z-buffering. Section 6 explores how coherent layer peeling may exploit floating point blending operations to reduce the number of passes. Section 7 discusses how the algorithm might be implemented on an ideal rasterization architecture of the future. Section 8 compares performance of GPU implementations of the hybrid and coherent layer peeling algorithms relative to traditional layer peeling. Section 9 draws conclusions and outlines areas for future work.

3. Hybrid Layer Peeling

It is relatively common to have easily sorted transparent geometry, such as slices through a volume data-set or screen-aligned sprites. Often, however, these may intersect geometrically with other meshes, such as semi-transparent iso-surfaces or treatment beams. For perfectly sorted geometry only a single rendering pass is required. However, as soon as the scene includes overlapping unsorted geometry, a more general technique such as layer peeling is required. This can lead to a very large number of passes for high-depth-complexity scenes.

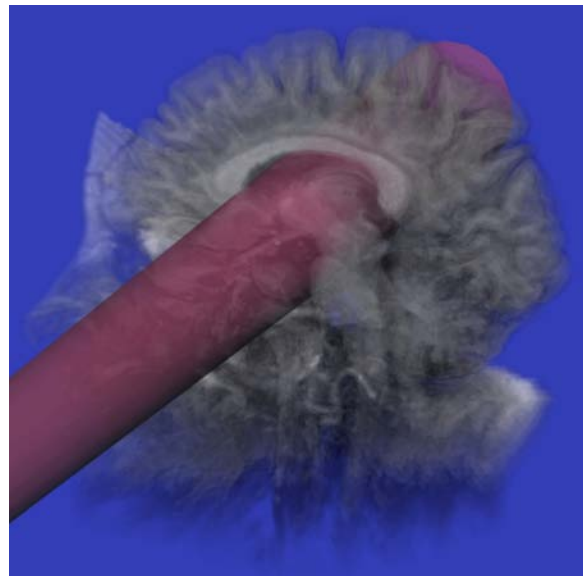

Figure 1: *Traditional layer peel requires 257 iterations to render the image versus a hybrid layer peeling approach needing only 3 iterations. This yields a 62x speed-up in rendering performance.*

A straightforward extension of layer peeling is to make the

© The Eurographics Association 2008.

observation that we can segment the algorithm into alternating steps. In one step, a conventional layer peel is performed for the unsorted geometry. In the second step the sorted geometry is rendered but clipped to a z-range between the previous peel depth and the current one. This simple algorithm, called hybrid layer peeling, is ideal for example scenes such as that shown in figure 1, where a semi-transparent surface, representing a beam, intersects a trivially sorted stack of volume slices. The algorithm is described more formally in Algorithm 1.

Algorithm 1 Hybrid Layer Peeling

```
 1: for all pixel sample locations do
 2:     zPeel ← zNear
 3: end for
 4: gChanged ← true
 5: while gChanged = true do
 6:     gChanged ← false
 7:     for all pixel sample locations do
 8:         zPeelNext ← zOpaque
 9:     end for
10:     for all unsorted geometry fragment locations do
11:         if zPeel < zFragment < zPeelNext then
12:             zPeelNext ← zFragment
13:             gChanged ← true            ▷ occlusion query
14:         end if
15:     end for
16:     for all sorted geometry fragment locations do
17:         if zPeel < zFragment < zPeelNext then
18:             Composite fragment into color buffer
19:         end if
20:     end for
21:     if gChanged = true then
22:         for all non-sorted geometry fragments do
23:             if zFragment = zPeelNext then
24:                 Composite fragment into color buffer
25:             end if
26:         end for
27:         swap(zPeel,zPeelNext)
28:     end if
29: end while
```

The number of total iterations of the algorithm is determined by the number of layer peels required for just the unsorted geometry, rather than the total depth complexity of the worst case pixel, including the sorted fragments. This can lead to asymptotic performance improvements proportional to the depth complexity of the sorted geometry, which in the case of a volume stack, can be very high. Unfortunately, only a limited class of scenes can be accelerated by hybrid layer peeling, namely those in which one set of scene geometry is perfectly sorted. A more general class of algorithm would automatically exploit correctly sorted fragment sequences at each pixel where they exist and correctly render unordered fragment sequences as they arise. Such an algorithm is described in the next section.

4. Algorithm Overview of Coherent Layer Peeling

Ideally we wish to develop an algorithm for correctly-rendered transparency that runs in linear time. Suppose we are given a list of transparent surfaces S that intersect the line segment emanating out of the camera through a given pixel center between the near and far plane. To properly composite these surfaces together they must be blended in front-to-back or back-to-front order. In either case this operation requires sorting of the surfaces in S based on depth. For a set of arbitrarily placed surfaces we know that sorting can be done in $O(n \lg n)$. When bucket sorting works, these surfaces can be sorted in $O(n)$ [WGER05]. This places assumptions on the placement of surfaces.

Rather than making such assumptions, we use the property of sorted coherency as done by Naga et al. [GHLM05]. We assume that the surfaces in S are mostly in sorted order to begin with and that we only need to fix the ordering of a small constant number of surfaces. In practice this assumption works well since data can be sorted up front based on a given camera view and updated periodically as the camera moves and the scene changes. For a practical scene with a smoothly moving camera and geometry it is unlikely that the surfaces will become completely out of order from frame to frame. The asymptotic performance of our algorithm is tied to the number of correctly sorted uninterrupted spans in the list S.

Let $s_0 \ldots s_{n-1} \in S$ be the list of mostly ordered surfaces that intersect the line segment emanating out of the camera through a given pixel center between the near and far plane. For any $s_i \in S$, let $D(s_i)$ denote the z-depth of the surface. Let us assume that our surfaces are mostly sorted in a front-to-back manner. Suppose we have done coherent layer peeling up to some surface s_c. Ideally we want to find the depth of the next out-of-order surface in the z-range $(D(s_c), \infty]$. Figure 2 shows one such sequence of mostly sorted surfaces. The last peeled surface is given by s_4. The next out of order surface in the sequence is s_9. All surfaces within the peel range $[D(s_3), D(s_{10})]$ are by definition guaranteed to be in the correct order.

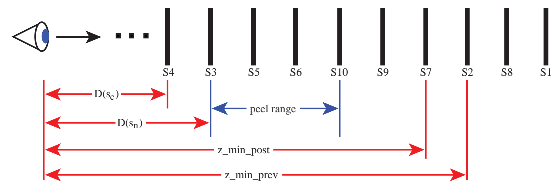

Figure 2: *Example of coherent peeling an out of order list of surfaces.*

We break the algorithm down into a series of steps. Let $D(s_n)$

© The Eurographics Association 2008.

be the first surface in depth following the last peeled layer s_c (see Figure 2).

4.1. Step 1: Examine the surfaces preceding s_n

In this step we find the nearest surface in the depth range $(D(s_n),\infty]$ that precedes s_n in sequence. By definition *any* surface that comes before s_n in the list whose depth is greater than s_n is an out-of-order surface. We define this list of surfaces as follows: $\hat{S} = \{s : s_i \in S, i < n, D(s_n) < D(s_i)\}$. Thus we need to find the nearest depth of any such surface if one exists. This is given by:

$$z_min_prev = \begin{cases} arg\min_{\hat{s}\in\hat{S}} D(\hat{s}), & \text{if } \hat{s} \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

The depth *z_min_prev* provides us a loose bound on the depth range of surfaces that are in order. Any coherent list of surfaces must now fall within the range $[D(s_n), z_min_prev)$.

4.2. Step 2: Examine the surfaces that come after s_n

In this step we find the nearest out-of-order surface in the depth range $[D(s_n),\infty)$ that follows s_n in sequence. This list of surfaces $\hat{S}$ is given by: $\check{S} = \{s_i : s_i \in S, i > n, D(s_n) \leq D(s_i), D(s_i) \leq D(s_{i-1})\}$. The nearest depth of any surface that occurs after s_n in the list is given by:

$$z_min_post = \begin{cases} arg\min_{\check{s}\in\check{S}} D(\check{s}), & \text{if } \hat{s} \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

The depth *z_min_post* provides a loose bound on the depth range of coherent surfaces. A tight bound on our peel range is now given by: $(D(s_n), min(z_min_prev, z_min_post)]$.

4.3. Step 3: Compositing the depth layers

Given the peel range interval, we can render our surfaces S clipped to the peel range. The surfaces within the peel range are guaranteed to be in correct front-to-back order so compositing can be done in a single pass through the data.

5. A GPU Algorithm

We now describe how both steps in section 4 can be mapped onto modern graphics hardware using five passes.

5.1. Initializing the algorithm

We start by separating our opaque surfaces from our transparent ones for rendering efficiency. We render the opaque objects into a depth map. We call this map the *opaque_depth*. We then initialize another depth map (which we call $D(s_c)$) to the z-near plane. These two buffers encode the depth range between which we must correctly resolve transparent ordering.

Following step 1 from section 4.1, we find the nearest out-of-order surface in the depth range $(D(s_n), opaque_depth)$ that precedes s_n in sequence. This step is broken down into two passes given by pass 1 and 2. As input to this step we are given the interval $(D(s_c), opaque_depth)$ stored in maps. The output of this step is the interval $(D(s_n), z_min_prev)$.

Pass 1: We start by counting the number of surfaces that update the z-buffer value and whose depth values lie in the range $(D(s_c), opaque_depth)$. We use the stencil buffer to do that. The depth range clipping is done in a simple fragment shader. The OpenGL state required for rendering this pass is as follows:

```
glClearStencil(0);
glClearDepth(1.0);
glStencilMask(0xFF);
glClear(DEPTH_AND_STENCIL_BUFFER_BITS);
glEnable(DEPTH_TEST);
glDepthFunc(LESS);
glEnable(STENCIL_TEST);
glStencilFunc(ALWAYS,0,0xFF);
glStencilOp(KEEP,KEEP,INCR);          ▷ fail,zfail,zpass
drawTransparentObjects();
```

Since we are clipping to the range $(D(s_c), opaque_depth)$ in the fragment shader, the stencil counting process is only impacted by surfaces s_i where $i <= n$. As soon as surface s_n flows through, $D(s_n)$ is written to the depth buffer and all subsequent surfaces ($e.g.\ s_i : i > n$) fail the z-test. At the end of this pass the stencil holds a count of the number of surfaces that updated the z-buffer. This count includes the update to the z-buffer by s_n. The depth buffer will hold $D(s_n)$, required in subsequent steps.

In the case illustrated in Figure 2 the following surfaces update the stencil buffer: s_1, s_2, and s_3. Thus the stencil buffer is set to 3 and the depth buffer to $D(s_3)$.

Note that the stencil count will be invalid if there is more than 255 surfaces that are in the opposite order and the count will overflow. Although that situation may happen in a very complex scene, in our case we also help the algorithm by providing a high level sorting of the surfaces thus virtually eliminating the chance of the stencil overflow. A more robust algorithm, which requires floating point blending, is given in Section 6.

Pass 2: By counting the stencil buffer back down to 1 we can find the minimum depth of any surface $s_i : i < n$ in the range $(D(s_c), opaque_depth]$. To do this, we render the geometry again, using the same depth comparison logic, counting down the stencil and only passing the stencil test if the stencil count is greater than one. The result of this process records *z_min_prev* into the z-buffer. Below is the OpenGL state required for the rendering pass:

In the case of Figure 2 we skip s_1 and stop at s_2, because

© The Eurographics Association 2008.

```
glClearColor(1.0,1.0,1.0,1.0);
glClear(COLOR_AND_DEPTH_BUFFER_BITS);
glEnable(DEPTH_TEST);
glDepthFunc(LESS);
glEnable(STENCIL_TEST);
glStencilFunc(LESS,1,0xFF);
glStencilOp(KEEP,KEEP,DECR);
drawTransparentObjects();
```

after rendering s_2 and setting the depth to $D(s_2)$ the stencil count reaches 1 and the pixel is not updated any more.

Following step 2 from section 4.2, we find the nearest out-of-order surface in the depth range $[D(s_n), min(z_min_prev, opaque_depth)$ that follows s_n in sequence. This requires two passes given by pass 3 and 4. For the next two passes we clip all surface fragments that fall outside the range $[D(s_n), min(z_min_prev, opaque_depth)$ using a simple fragment shader. It follows from step 1, that all surfaces s_i with $i < n$ fall outside this range so they play no role in Pass 3 or Pass 4.

Pass 3: We start by counting the number of correctly ordered surfaces that directly follows s_c in the range $[D(s_n), min(z_min_prev, opaque_depth)$. We set the z-test to GL_GREATER and each time a surface passes the z test we increment the stencil count. The first surface that fails the z-test will invert the stencil value and cause the stencil test to cull away all remaining surface fragments. The result of this pass is a stencil value that gives an upper bound on the number of surfaces that are in the correct depth order following s_c. The pseudo-code for this pass follows:

```
glClearStencil(0);
glClearDepth(0);
glClear(DEPTH_AND_STENCIL_BUFFER_BITS);
glEnable(DEPTH_TEST);
glDepthFunc(GREATER);
glEnable(STENCIL_TEST);
glStencilFunc(GREATER,128,0xFF);
glStencilOp(KEEP,INVERT,INCR);        ▷ fail,zfail,zpass
drawTransparentObjects();
```

In Figure 2 the stencil buffer is incremented for surfaces s_3, s_5, s_6, and s_7. The surface s_9 inverts the stencil value of 4 to 251 and sets the depth to $D(s_9)$.

There are no overflow issues in this pass. If there are more than 128 surfaces in proper order, when the stencil count reaches 128, the counting stops and we will assume that the last surface was out of order. Consequently, we can process at most 128 surfaces in one iteration of the algorithm.

Pass 4: In Pass 3 we counted the number of sorted z-coherent span of surfaces that directly follow $D(s_c)$. There may be other surfaces $s_j : j > n$ that interrupt this coherent span (for example, surface s_{10} in Figure 2). In this pass we compute the minimum depth of such surfaces. This value is *z_min_post*. The logic in Figure 5.1 culls away the first k surfaces that form the coherent span, counted in pass 3, using the stencil test and incrementing the stencil each time the stencil test fails. Once the stencil value reaches 255, the remaining surfaces flow through the pipeline and the one with the minimum depth is found using z-buffering with GL_LESS. The pseudo-code for pass 4 is as follows:

```
glClearColor(1.0,1.0,1.0,1.0);
glClearDepth(1.0);
glClear(COLOR_AND_DEPTH_BUFFER_BITS);
glDepthFunc(LESS);
glStencilFunc(EQUAL,255,0xFF);
glStencilOp(INCR,KEEP,KEEP);        ▷ fail,zfail,zpass
drawTransparentObjects();
```

In our example in Figure 2, we skip surfaces s_3, s_5, s_6, and s_7 and render surfaces s_9 and s_{10}. The depth value is set to $D(s_{10})$.

5.2. Step 3 (2-passes): Compositing the n depth layers

The surfaces that lie in the interval $[D(s_n), min(z_min_post, z_min_prev, opaque_depth)$ are guaranteed to be in an uninterrupted correct depth order. A standard compositing pass can use a fragment shader to clip to this range to accumulate these surfaces into the final image (pass 5).

Note that the surface at the upper bound of the peel range may be an out of order surface (in case the *z_min_post* is not set – for example, if the surface s_3 in Figure 2 was directly followed by surface s_2). Since we have this depth we can peel this surface away as well using a z equals test in an additional compositing pass (pass 6). Thus our algorithm will peel away (in the worst case) two layers per iteration of the process.

Once the coherent range of surfaces have been composited, we update $D(s_c)$ to be the min of *z_min_prev*, *z_min_post*, and $opaque_depth$ and return to Step 1. This process is repeated until no surfaces remain to be peeled. We can detect when this occurs using occlusion queries during the compositing phase.

6. Advanced Formulation using Floating Point Image Blending

Both pass 1 and 2 can be replaced by a single pass bringing the total count to five. This is done by using floating point blending to store the depth of the surface that was previously closest to the camera in the color channels. To achieve that we initialize the depth and color channels to have a value of z-far. When we render the geometry we set the color to 1.0, 1.0, 1.0 and the alpha to the fragment depth. By using a blend method that multiplies the source color by the destination

© The Eurographics Association 2008.

alpha, we move the depth of the previous surface in the frame buffer to the color channels of the frame buffer. The result of this pass will store the value of *z_prev_min* in the color channels. The OpenGL state required for rendering this pass is given as follows:

```
glClearStencil(0);
glClearDepth(1.0);
glClearColor(1.0,1.0,1.0,1.0);
glClear(DEPTH_BUFFER_BIT);
glClear(COLOR_AND_DEPTH_BUFFER_BITS);
glEnable(DEPTH_TEST);
glDepthFunc(LESS);
glEnable(BLEND);
glBlendFuncSeparate(ZERO,DEST_ALPHA,ONE,ZERO);
drawTransparentObjects();
```

7. An Ideal GPU Algorithm

The algorithm in section 6 requires five geometry passes per iteration. It peels away at least two layers per iteration. We now detail an algorithm that reduces the number of geometry passes down to two, with only minor modifications required to existing hardware. An ideal rasterization engine would allow for efficient rasterization of geometry while permitting access to multiple destination buffers in a rasterization shader. The rasterization shader could either be part of a general shader, computing color and alpha at the same time, or it could be run on a finer scale, at the level of sub-pixel samples (such as in the rasterization of pixels unit), with the surface material shader being amortized over several pixel subsamples, as is done in current GPUs. In current GPUs the rasterization of pixels unit (ROP) is restricted to a limited frame-buffer state machine. The proposed algorithm could also be implemented as an extension of such a state machine.

The rasterization shader would have available interpolated Z for the fragment at the sample location as well as multiple z values and additional integer values (equivalent to stencils) that could be read and written to in the equivalent of render targets. The rasterization shader would be executed for each fragment in rasterization order for each pixel subsample. Such a system would enable several of the passes for conventional GPUs to be collapsed into a single pass on this ideal GPU.

Available to the rasterization shader would be:

zFragment : the interpolated depth of a fragment

zPeel : the peel depth from the previous pass of the algorithm

zOpaque : the depth of the nearest opaque surface or zFar, whichever is nearest to the camera

z1, z2, z3 : a set of full precision buffers used to store z-values (either floating point or fixed point)

Algorithm 2 Ideal Coherent Layer Peel

```
 1: for all pixel sample locations do
 2:     z3 ← zNear
 3: end for
 4: gChanged ← true
 5: while gChanged = true do
 6:     gChanged ← false;
 7:     for all pixel sample locations do
 8:         z1 ← zOpaque;
 9:         z2 ← zOpaque;
10:         zPeel ← z3;
11:         zPostFlag ← false
12:     end for
13:     for all geometry sample locations do                ▷ Pass 1
14:         if ( zPeel < zFragment < z2) then
15:             gChanged ← true;
16:             if zFragment < z1 then
17:                 z2 ← z1
18:                 z1 ← zFragment
19:                 z3 ← z1                    ▷ Reset the state for post
20:                 zPostFlag ← false
21:             else if zPostFlag = true then
22:                 if zFragment < z3 then
23:                     z3 ← zFragment
24:                 end if
25:             else
26:                 if zFragment < z3 then
27:                     zPostFlag ← true
28:                 end if
29:                 z3 ← zFragment
30:             end if
31:         else if zFragment = zPeel then
32:             Composite fragment into color buffer
33:         end if
34:     end for
35:     if gChanged = true then
36:         for all geometry sample locations do            ▷ Pass 2
37:             if ( zPeel < zFragment < z3) then
38:                 Composite fragment into color buffer
39:             end if
40:         end for
41:     end if
42: end while
```

8. Performance

To assess the value of the different algorithms, we considered two metrics. One was absolute timings of performance on current state-of-the-art hardware (NVIDIA 8800 GTX), and the second was the number of geometry passes required to render a particular scene with correct transparency. Absolute performance is practically interesting in the short term for obvious reasons. The number of geometry passes is more theoretically interesting since it indicates the value of a hybrid or coherent layer peeling algorithm assuming further

© The Eurographics Association 2008.

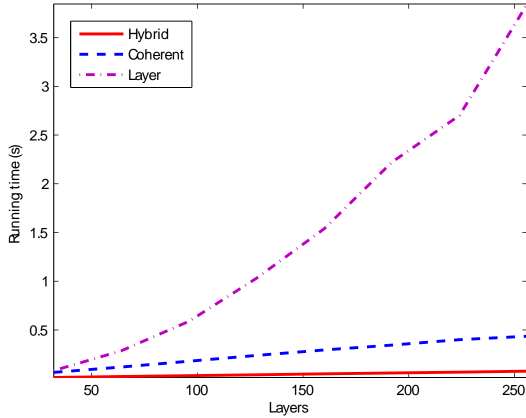

Figure 3: *Volume rendering with transparent intersecting cone as shown in figure 1.*

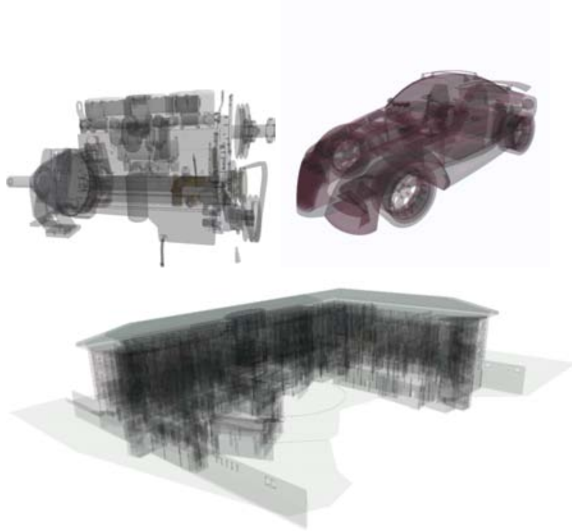

Figure 4: *Semi-transparent engine, car, and Siebel Center models.*

optimization of the rasterization hardware. This optimization could be done by enabling an ideal coherent layer peeling algorithm or by improving or merging passes of the current GPU algorithm.

All scenes that may be rendered by hybrid layer peeling may also be rendered by coherent layer peeling and traditional layer peeling. To determine how the algorithms scale with depth complexity we used a stack of alpha-modulated planes representing a volume intersecting a conical shell representing a treatment beam. Since the conical shell has very low depth complexity, this is an ideal case for hybrid rendering

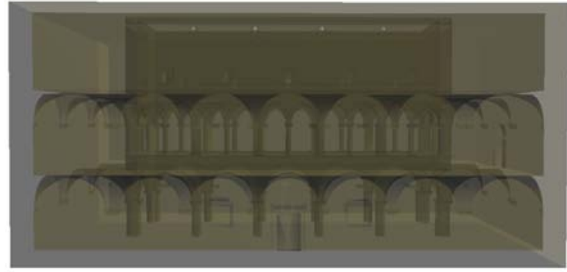

Figure 5: *Semi-transparent Sponza, Lucy, and dragon model.*

model Figure	faces vertices	coherent peel A	coherent peel B	layer peel
Engine (Fig.4)	133K f 120K v	22 fps 7 it,42 g	23 fps 7it,35 g	21 fps 22it,44 g
Dragon front	150K f 749K v	51 fps 3 it,18 g	53 fps 3 it,15 g	35 fps 14it,28 g
Dragon side (Fig.5)	150K f 749K v	51 fps 3 it,18 g	53 fps 3 it,15 g	35 fps 14 it,28 g
Lucy end on	250K f 125K v	37 fps 3 it,18 g	41 fps 3 it,15 g	15 fps 22 it,44 g
Lucy front (Fig.5)	250K f 125K v	36 fps 3 it,18 g	40 fps 3 it,15 g	27 fps 12 it,24 g
Sponza end on	66K f 61K v	36 fps 3 it,18 g	40 fps 3 it,15 g	12 fps 26 it,52 g
Sponza front (Fig.5)	66K f 61K v	56 fps 3 it,18 g	50 fps 3 it,15 g	35 fps 16 it,32 g
Car front (Fig.4)	730K f 365K v	0.28 fps 11 it,66 g	0.34 fps 11 it,55 g	0.27 fps 36 it,72 g
Siebel Center front (Fig.4)	357K f 444K v	0.71 fps 20 it,120 g	0.83 fps 20 it,100 g	0.86 fps 50 it,100 g

Table 1: *Frame rates, number of algorithm iterations (it), and geometry passes (g) for various models rendered using the coherent layer peel algorithm A, the advanced version with floating point blending B, and the traditional layer peeling algorithm.*

and coherent layer peeling. Figure 3 shows the test scene timings, with the number of slices spanning the volume being a settable parameter. It shows the absolute timings and number of required peels for the different algorithms. As expected, the number of peels increases linearly with the number of volume layers for traditional layer peeling. The total

© The Eurographics Association 2008.

time for traditional layer peeling increases faster than linear time. In contrast, the number of passes required by hybrid and coherent layer peeling remain constant, with the rendering times increasing linearly with the number of layers, as expected. For 256 layers, hybrid layer peeling achieves a 52 to 1 speed improvement over layer peeling. Coherent layer peeling is slower by a constant factor of approximately 6. This is due to the larger number of geometry passes per iteration of the algorithm as well as the need for rendering the volume layers in each pass. For hybrid layer peeling the volume slices only need to be rendered for the compositing pass, not the depth peel pass.

To explore the performance of coherent layer peeling versus traditional layer peeling for more general scenes, we used a variety of models. The total number of layers required by traditional layer peeling depends on the location of the camera. We compare a top and side view of some models. These typically represented the range in performance when rendering these models. Results are given in Table 1. Coherent layer peeling based on the algorithm in Section 5 shows up to a 3x speed improvement over traditional layer peeling. The floating point variant of the algorithm is slower and faster than the stencil-based version depending on the model. The variation is due to the relative cost of fewer geometry passes for the floating-point blending algorithm versus the more expensive bandwidth to the framebuffer required by floating point blending.

9. Conclusions and Future Work

We introduced two new approaches to layer peeling that take advantage of the available sorted structure in a scene. Hybrid layer peeling is somewhat narrow in applicability, but shows great speed-ups relative to traditional layer peeling and is simple to implement on current GPUs. Coherent layer peeling applies to a broader class of partially sorted scenes, and shows an up to 3x speed up on current hardware for typical test models. However, it holds the promise to be the basis of improved rendering hardware in the future that could allow it to match traditional layer peeling in the worst case, and have asymptotically better performance for high-depth complexity scenes with good but not perfect sorting.

This paper focused on the rasterization logic of hybrid and coherent layer peeling. Coherent layer peeling benefits from well-sorted scenes. Determining the ideal sorting algorithms and other heuristics, such as geometry dicing, when coherent layer peeling is the core rasterization algorithm is a rich area for future work. Our example implementation used a simple bucket sort of triangle centroids, so there is probably scope for improvement. Investigations of coherent layer peeling on future GPUs as they become available may yield significant speed-ups. Exploring how framebuffer state-machines might be enhanced to reduce the number of passes is also a promising line of research, as will be actual implementations of the ideal algorithm if suitable hardware becomes available. Z-culling is key to the efficiency of traditional layer peeling, and exploitation of z-culling in the context of coherent layer peeling is still an area of future work. We expect an ideal system would combine the approaches of coherent layer peeling and that of Wexler et. al [WGER05] to further enhance performance. Finally, as it becomes practical to render scenes with high transparent depth complexity, it will be important to explore how such rendering may be used effectively in visualization.

References

[AMN03] AILA T., MIETTINEN V., NORDLUND P.: Delay streams for graphics hardware. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), ACM, ACM, pp. 792–800.

[BCL*07] BAVOIL L., CALLAHAN S. P., LEFOHN A., COMBA J. L. D., SILVA C. T.: Multi-fragment effects on the GPU using the k-buffer. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2007), ACM, ACM, pp. 97–104.

[Eve01] EVERITT C.: *Interactive Order-Independent Transparency*. Tech. rep., NVIDIA Corporation, May 2001. http://developer.nvidia.com/object/Interactive_Order_Transparency.html.

[GHLM05] GOVINDARAJU N. K., HENSON M., LIN M. C., MANOCHA D.: Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2005), ACM, ACM, pp. 49–56.

[HPS05] HOUSTON M., PREETHAM A., SEGAL M.: *A Hardware F-Buffer Implementation*. Tech. Rep. CSTR 2005-05, CS Dept., Stanford U., May 2005.

[LWX06] LIU B., WEI L.-Y., XU Y.-Q.: *Multi-Layer Depth Peeling via Fragment Sort*. Tech. rep., Microsoft, June 2006. http://research.microsoft.com/research/pubs/view.aspx?tr_id=1125.

[Mam89] MAMMEN A.: Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Comput. Graph. Appl. 9*, 4 (1989), 43–55.

[MB07] MYERS K., BAVOIL L.: Stencil routed k-buffer. In *ACM SIGGRAPH 2007, sketches* (New York, NY, USA, 2007), ACM, p. 21.

[MP01] MARK W. R., PROUDFOOT K.: The F-buffer: A rasterization-order FIFO buffer for multi-pass rendering. In *Graphics Hardware 2001* (2001), EUROGRAPHICS.

[WGER05] WEXLER D., GRITZ L., ENDERTON E., RICE J.: Gpu-accelerated high-quality hidden surface removal. In *Graphics Hardware 2005* (July 2005), pp. 7–14.

© The Eurographics Association 2008.