
# Order Independent Transparency with Dual Depth Peeling

Louis Bavoil, Kevin Myers

February 2008

# Document Change History

| Version | Date | Responsible | Reason for Change |
|---|---|---|---|
| 1.0 | February 9 2008 | Louis Bavoil | Initial release |
| | | | |
| | | | |
| | | | |

# Abstract

Rendering semi-transparent surfaces correctly using the alpha blending equation requires sorting fragments from front to back or back to front. Depth peeling is a robust solution to the fragment sorting problem which renders the transparent objects multiple times, peeling one layer of fragments every time. We introduce dual depth peeling, an extension of depth peeling based on a min-max depth buffer which peels two layers at a time, one layer from the front and one from the back. Alpha blending is interleaved with dual depth peeling, using different blend equations for front peeling and back peeling. The algorithm guarantees that no fragments can be blended twice. We implement our min-max depth buffer using 32-bit floating-point blending, which is fast on GeForce 8. We also describe a sort-independent method which replaces the RGBA color in the alpha blending equation by the per-pixel weighted average over the pixel, using alpha values as weights. This approximate method can render plausible order independent transparency with a single geometry pass.

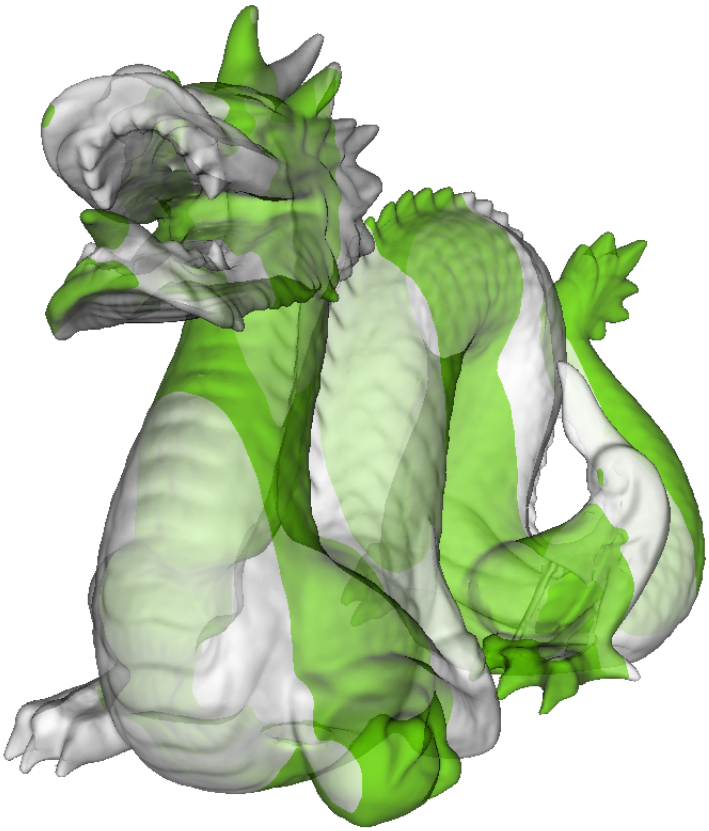


Figure 1. Dual depth peeling peels 4 color layers in 2 geometry passes. Image rendered with alpha=60%.

# Introduction

Rendering semi-transparent materials accurately with non-uniform RGBA colors requires composing fragments in depth order. Depth peeling [Everitt01] [Mammen84] is a robust image-based solution to this problem which captures one layer of fragments each time the geometry is rendered (geometry pass). Besides order-independent transparency, depth peeling is useful for generating layered depth images and tracing bundles of rays. The major drawback of the original depth peeling algorithm is that it requires one geometry pass per peeled layer. Thus, to capture all the fragments of a scene of depth complexity N, depth peeling requires N geometry passes. This is prohibitive for GPU-bound applications.

Dual depth peeling reduces the number of geometry passes from N to N/2+1 by applying the depth peeling method from the front and from the back simultaneously. The one pass overhead is due to the fact that our algorithm needs to work on two consecutive layers at a time. Besides dual depth peeling, we also describe how the peeled fragments can be blended on the fly based on the alpha blending equation.

We utilize new features of the GeForce 8: the min-max depth buffer for dual depth peeling is an RG32F color texture with MAX blending (previously 32-bit float blending was not supported), and the depth buffer for front-to-back depth peeling is a 32-bit depth buffer (previously the maximum depth precision was 24 bits per pixel).

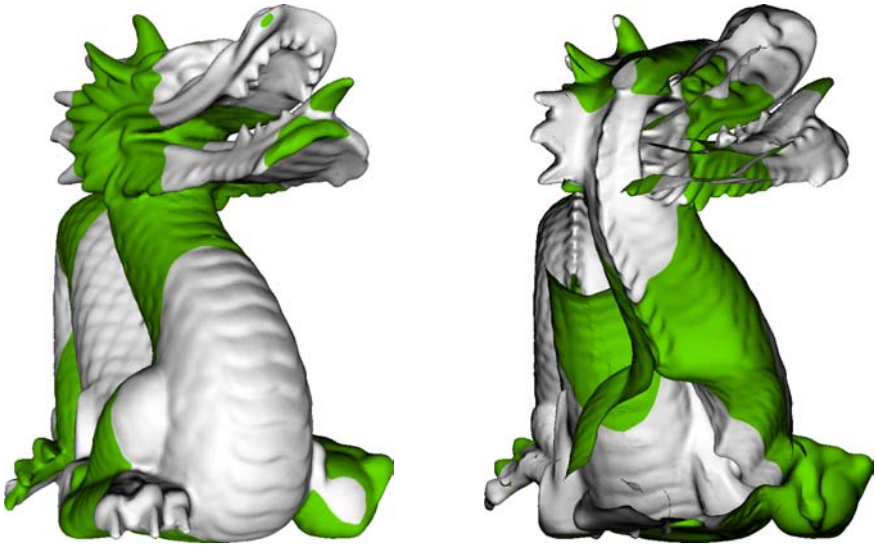


Figure 2. Front-to-back depth peeled layers.

# Overview

## Depth Peeling

Depth peeling [Everitt01] starts by rendering the scene normally with a depth test, which returns the nearest fragments to the eye. In a second pass, the depth buffer from the previous pass is bound to a fragment shader. To avoid read-modify-write hazards, the source and destination depth buffers are swapped every pass (ping ponging). If the fragment depth is less or equal to the associated depth from the previous pass, then this fragment is discarded and the next layer underneath is returned by the depth test. See Figure 2.

## Dual Depth Peeling

Conceptually, dual depth peeling performs depth peeling in both back-to-front and front-to-back directions, peeling one layer from the front and one layer from the back every pass. This would be simple to implement if the GPU had multiple depth buffers and if each depth buffer was associated with a specific color textures. There would be one RGB texture for back-to-front blending and one RGBA texture for front-to-back blending. However, in the case of Figure 4, the layer in the middle would be peeled and blended both in the front-to-back and the back-to-front blending textures. We solve this issue by working on a sliding window of two consecutive layers. In addition, this makes it possible to implement a min-max depth test with blending combined in the same shader ("blending on the fly"). We implement a min-max depth buffer with an RG32F color texture, by using MIN blending to perform the depth comparison. We essentially turn off the hardware depth buffer, and use blending to perform the read-modify-write part of our custom depth test. The peeled fragments from the front layers need to go though the under-blending equation, and the fragments from the back layers through the over-blending equation. Finally, the two blended images for the front and the back directions are blended together using under blending.

Figure 3. Dual depth peeling advancing fronts.

The algorithm works on sets of consecutive layers, two layers from the front and two layers from the back. In the first pass, the outside set is initialized to the min and max depths and no color is peeled yet. In the second pass, the shader processes the fragments matching the depths of the previous outside set (dashed layers on Figure 3), and performs depth peeling on the layers inside the outside set (plain layers on Figure 3).

When the front and back advancing fronts meet, fragments may be processed as either front or back fragments. We process them as front fragments as in the following pseudo code:

```
if (fragDepth == nearestDepth) {
    // process front fragment
} else {
    // process back fragment
}
```

# Implementation Details

## Depth Comparisons

The depth value written in the depth buffer is directly available in the fragment shader as `gl_FragCoord.z`. So we want to compare `gl_FragCoord.z` with the depth stored in the depth buffer at `gl_FragCoord.xy`. Because both `gl_FragCoord.z` and our depth buffer are 32-bit floats, we can compare them without any bias.

## Under Blending

There are two ways to perform alpha blending. The most common way is to composite fragments from back to front using the blending equation

$$C_{dst} = A_{src}\, C_{src} + (1\text{-}A_{src})\, C_{dst}$$

This equation does not use the alpha channel of the destination color buffer. When peeling layers of fragments from the front and compositing from front to back, the alpha blending equation must be modified. A fragment color ($C_1$, $A_1$) blended over a color ($C_2$, $A_2$) over a background color $C_0$, generates the following $C_2$' color:

$C_1' = A_1\, C_1 + (1\text{-}A_1)\, C_0$

$C_2' = A_2\, C_2 + (1\text{-}A_2)\, C_1'$

$C_2' = A_2\, C_2 + (1\text{-}A_2)\, (A_1\, C_1 + (1\text{-}A_1)\, C_0)$

$C_2' = A_2\, C_2 + (1\text{-}\, A_2)\, A_1\, C_1 + (1\text{-}A_2)\, (1\text{-}A_1)\, C_0$

The above equation shows that the fragments can be blended in front-to back order with the following separate blending equations:

$$C_{dst} = A_{dst}\, (A_{src}\, C_{src}) + C_{dst}$$

$$A_{dst} = (1\text{-}A_{src})\, A_{dst}$$

where $A_{dst}$ is initialized to 1.0. This is performed efficiently by pre-multiplying $C_{src}$ by $A_{src}$ in a shader and using separate blend equations for RGB and alpha with:

```
glEnable(GL_BLEND);
glBlendEquation(GL_FUNC_ADD);
glBlendFuncSeparate(GL_DST_ALPHA, GL_ONE,
                    GL_ZERO, GL_ONE_MINUS_SRC_ALPHA);
```

Finally, the blended fragments are composited with the background color $C_{bg}$ using a fragment shader implementing $C_{dst} = A_{dst} C_{bg} + C_{dst}$.

# Render Targets

The GeForce 8 supports 32-bit floating-point blending. By using MAX blending and writing float2(-depth, depth) in a shader, we can render the min and max depths for every pixel. This way, one can implement depth peeling on (-depth) and (depth), effectively peeling from the front and form the back simultaneously.

Our dual depth peeling fragment shader (shaders/dual_peeling_peel_fragment.glsl) writes to 2 color textures. This render to multiple render targets is implemented in OpenGL using framebuffer objects (FBOs). Render target RT0 is for the min-max depth buffer, RT1 for dumping the front fragments and RT2 for dumping the back fragments.

The GeForce 8 does not support independent blending equation for different render targets. Since we need either MIN or MAX blending for the min-max depth buffer, we need to use MIN or MAX blending with all our render targets. We use MAX blending and initialize the dumping color textures to 0.0f every pass. This essentially allows us to perform MAX blending on one render target and REPLACE blending on the two other targets. The internal texture formats are RG32F for RT0 (min-max depth buffer), RGBA8 for RT1 (front fragments to be blended under) and RGBA8 for RT2 (back fragments to be blended over).

# Blending on the Fly

With DirectX 10.1, it will be possible to setup the under-blending equation on render target 1, and the over blending equation on render target 2. We can work around by observing that every pass, at most one front fragment and one back fragment need to be blended. Thus, we can dump the fragments to textures using REPLACE blending as explained above, and blend them in a full screen pass.

In addition, because under blending is essentially additive blending, the RGB color with under blending increases every pass. So we can use MAX blending and our default pass-through, and perform the additive blending in the dual depth peeling shader. Besides, we use the third component of RT0 to store the $(1-A_{dst})$ component necessary for under blending, which works because $A_{dst}$ decreases every pass (multiplied by $(1-A_{src})$ every pass). Thus, MAX blending is applied to float3(-depth, depth, $1-A_{dst}$).

# Single-Pass Approximations

## Weighted Average

At GDC 2007, [Meshkin07] presented the idea of simplifying the alpha blending equation to make it order independent. The proposed strategy was to expand the alpha blending equation and to ignore the order-dependent terms. This method produces plausible images for low alpha values. However, it produces overly dark or bright images (according to what terms are ignored) except for low alpha (< 30 % opaque).

Our weighted average technique is similar but does not ignore any term. It builds on the following observation. If all the RGBA colors were identical for a given pixel, then the result would be independent on the order in which the fragments are alpha blended. Now, in the case where the RGBA colors are not uniform, what if we replaced the multiple colors per pixel by a uniform color per pixel, such as the average color per pixel? To handle non-uniform alphas, we use an alpha-weighted average of the colors for every pixel.

The average RGBA color for each pixel is generated by rendering the transparent geometry into an accumulation buffer implemented as a 16-bit floating-point texture. The result is an accumulated RGBA color and the number of fragments per pixel (depth complexity). After this geometry pass, the accumulation buffer is passed to a post-processing full screen pass, which performs a weighted average of RGB by alpha, yielding a weighted average color C:

$$C = \sum[(RGB)\ A] \ / \ \sum A.$$

The average alpha is also computed from the accumulated alpha and the depth complexity n:

$$A = \sum A \ / \ n$$

Then the alpha blending equation is approximated by replacing all the terms with the average color C and the average alpha A. Since $\sum[1\text{-}A, n=0..n\text{-}1] = (1\text{-}(1\text{-}A)^n)/(1\text{-}(1\text{-}A))$, the blended color $C_{dst}$ is only a function of C, A, n, and the background color $C_{bg}$:

$$C_{dst} = C\ A \sum[(1\text{-}A)^k, k=0..n\text{-}1] + C_{bg}\ (1\text{-}A)^n$$

$$C_{dst} = C\ A\ (1\text{-}(1\text{-}A)^n)/A + C_{bg}\ (1\text{-}A)^n$$

The color and alpha sums are computed using additive 16-bit float blending, adding together the RGBA colors in one render target, and the depth complexity n in another target.

## Weighted Sum

We compare our weighted average method with the weighted sum formula introduced at GDC 2007. There are two versions of this technique. Here is the simple formula which is in fact a weighted sum [Meshkin07]:

$$C_{dst} = \sum[A_{src}\ C_{src}] + C_{bg}\ (1 - \sum A_{src})$$

# Running the Sample

| Control | Action |
|---|---|
| Right Mouse Button | Application menu |
| Left Mouse Button | Rotate object |
| Control + Left Mouse Button | Pan object in front of the camera |
| Shift + Left Mouse Button | Dolly the object toward/away from the camera |
| A/D | Change the alpha value |
| 1 | Use dual depth peeling |
| 2 | Use front-to-back depth peeling |
| 3 | Use weighted average |
| 4 | Use weighted sum |

By default, occlusion queries are enabled for dual depth peeling and front-to-back depth peeling. Occlusion queries can be toggled on and off by pressing the 'Q' key. When occlusion queries are disabled, a fixed number of layers is used which can be configured using the '+' and '-' keys. The overlay on the bottom left bottom of the screen displays the number of geometry passes, as well as the current technique. With the default model (Stanford Dragon, 800k+ triangles), the GPU is mainly vertex bound, so the performance is inversely proportional to the number of geometry passes.

# Implementation Notes

## GLSL Separate Compilation

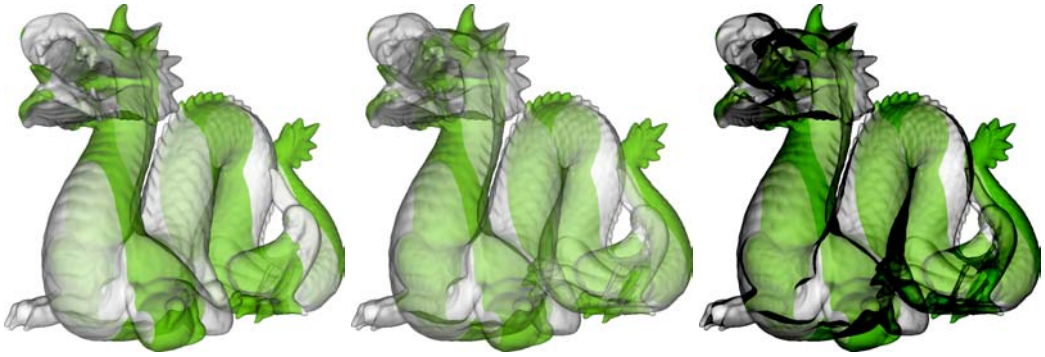
The sample uses the separate compilation mechanism of GLSL to share GLSL code between shaders (shade_vertex.glsl and shade_fragment.glsl). This is a preferred alternative to using #includes, which are not officially part of the GLSL language.

## RG32F Rendering

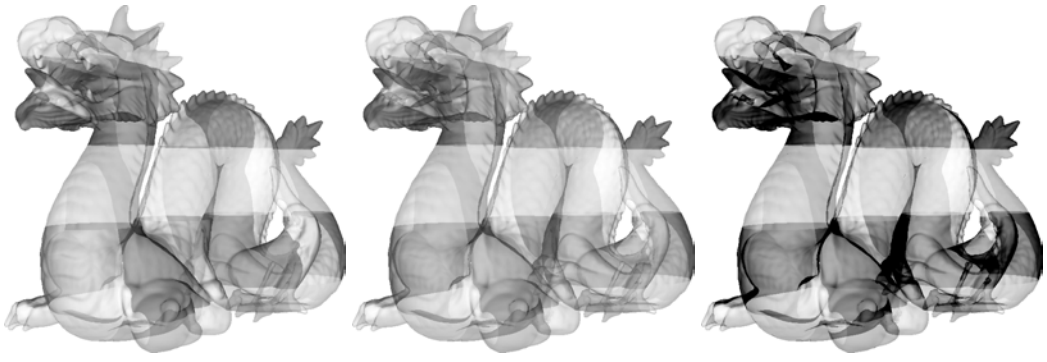
We render to RG32F by using a rectangle texture with the internal format GL_FLOAT_RG32_NV.

# Images

Alpha = 20%



Alpha = 60%



Non-uniform Alpha

Figure 4. (left) Dual Depth Peeling, (middle) Weighted Average, (right) Weighted Sums.

# Conclusion

The advantage of depth peeling is that the primitives can be rendered in any order and it guarantees correct fragment ordering from back to front or front to back. However, it requires one geometry pass per layer.

Dual depth peeling reduces the number of passes from N to N/2+1, which may speed up performance by 2x for geometry bound applications. For N=2, dual depth peeling extracts the front and back layers, while front-to-back peeling extracts the first two front layers. The back layer is important for the transparency look of convex-like objects, so this is a nice property of dual depth peeling. However, since dual peeling does not use any depth test, it disables the early-z optimizations of the hardware.

An alternative technique is the stencil routed k-buffer [Myers07] which can capture up to 8 layers per geometry pass on GeForce 8. A drawback of the k-buffer is that because the fragments are captured in rasterization order, they need to be sorted in a post-processing pass. Dual depth peeling also has the advantage that exiting early before capturing all the fragments always captures the foremost layer, whereas the stencil routed k-buffer needs to capture all the layers first, to make sure that the foremost layer is captured.

The weighted average and weighted sum techniques render the geometry only once per frame, so they have a much higher performance. Our weighted average technique works nicely for high and low alpha values, whereas the weighted sum technique darkens the colors significantly for alpha > 20%.

# References

[Mammen84] A. Mammen. “Transparency and antialiasing algorithms implemented with the virtual pixel maps technique”. IEEE Computer Graphics and Applications, 9:43–55, July 1984.

[Everitt01] Cass Everitt, “Interactive order-independent transparency”, Technical report, NVIDIA Corporation, 2001.

[Meshkin07] Houman Meshkin, “Sort-Independent Alpha Blending”, Perpetual Entertainment, GDC Session, March 2007.

[Myers07] "Stencil Routed A-Buffer", Kevin Myers, Louis Bavoil, ACM SIGGRAPH Technical Sketch Program, 2007.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com