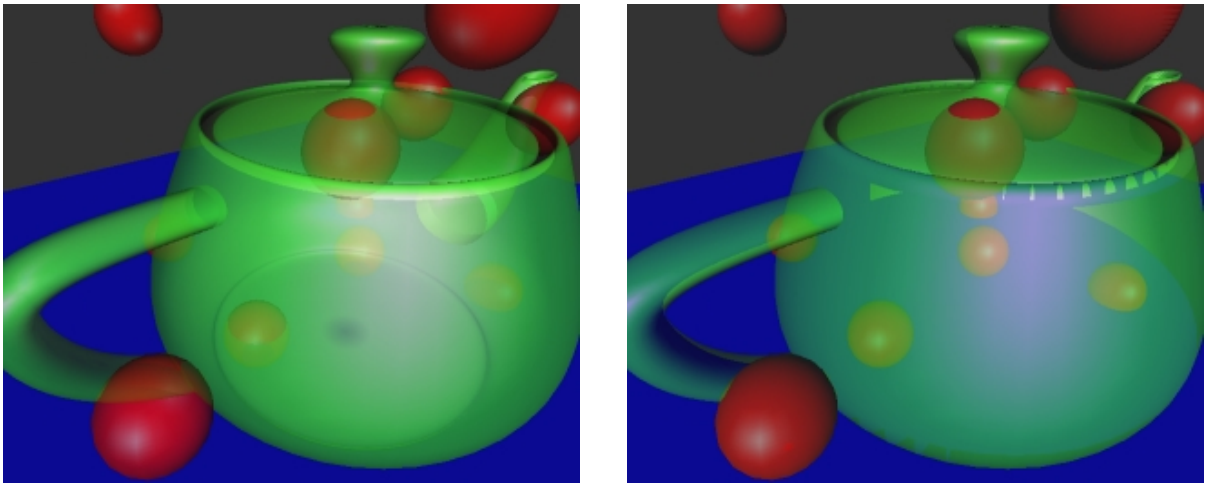
Interactive Order-Independent Transparency

Cass Everitt
NVIDIA OpenGL Applications Engineering
cass@nvidia.com



(a) (b)

Figure 1. These images illustrate correct (a) and incorrect (b) rendering of transparent surfaces.

Introduction

Correctly rendering non-refractive transparent surfaces with core OpenGL functionality [9] has the vexing requirements of depth-sorted traversal and non-intersecting polygons. This is frustrating for most application developers using OpenGL because the natural order of scene traversal (usually one object at a time) rarely satisfies these requirements. Objects can be complex, with their own transformation hierarchies. Even more troublesome, with advanced graphics hardware, the vertices and fragments of objects may be altered by user-defined per-vertex or per-fragment operations within the GPU. When these features are employed, it becomes intractable to guarantee that fragments will arrive in sorted order for each pixel. The technique presented here solves the problem of order dependence by using a technique we call *depth peeling*. *Depth peeling* is a fragment-level depth sorting technique described by Mammen using *Virtual Pixel Maps* [7] and by Diefenbach using a *dual depth buffer* [3]. Though no dual depth buffer hardware fitting Diefenbach's description exists, Bastos observed that shadow mapping hardware in conjunction with alpha test can be used to achieve the same effect [2]. Using this variation of depth peeling, each unique depth in the scene is extracted into layers, and the layers are composited in depth-sorted order to produce the correctly blended final image. The peeling of a layer requires a single order-independent pass over the scene. Figure 1 contrasts correct and incorrect rendering of transparent surfaces.

The goal of this document is to enable OpenGL developers to implement this technique with NVIDIA OpenGL extensions and GeForce3 hardware. Since shadow mapping is integral to the technique a very basic introduction is provided, but the interested reader is encouraged to explore the referenced material for more detail.

Shadow Mapping

Shadow mapping is a multi-pass shadowing technique developed by Lance Williams [11] in 1978. In the first pass, the scene is rendered from the light's point of view. The depth buffer generated in that pass is copied to a special "depth texture" or shadow map. In the second pass, the shadow map is projected onto the scene using projective texture mapping [10, 4]. Unlike regular 2D projective texture mapping where the *r* coordinate is unused, we use the *r* coordinate to compute the distance of the rasterized fragment to the light source. Then, the lookup of (*s*,*t*) is the distance to the *nearest* surface to the light source (along that direction). If $r \leq \texttt{lookup}(s,t)$, then the current fragment is *visible to the light source*, and therefore *not in shadow*. Essentially, we use depth-buffering in the first pass to determine which surfaces are visible from the light's point of view, and in the second pass we show those surfaces as illuminated. Figure 2 helps illustrate this concept.

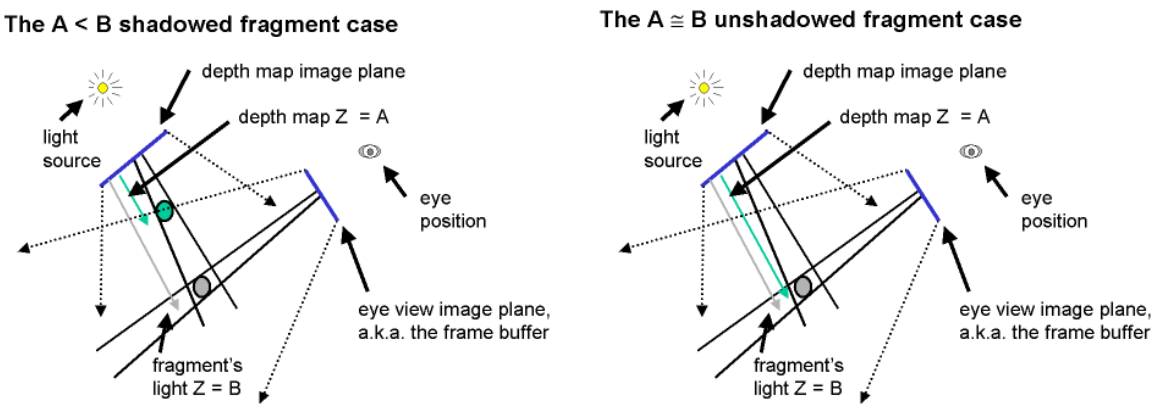


Figure 2. These diagrams were taken from Mark Kilgard's shadow mapping presentation at GDC 2001. They illustrate the shadowing comparison that occurs in shadow mapping.

We use the `SGIX_shadow` and `SGIX_depth_texture` extensions [8] to take advantage of GeForce3 shadow mapping hardware in OpenGL. The `SGIX_shadow` extension provides the ability to compute a comparison of the *r* texture coordinate with the results of the 2D lookup. The `SGIX_depth_texture` extension exposes `GL_DEPTH_COMPONENT` internal texture formats and defines semantics for `glCopyTex{Sub}Image2D` for fast copies from the depth buffer to a depth texture. These features are fully accelerated on GeForce3.

It has been shown by Heidrich [5] that multitexturing can be used to implement a limited form of shadow mapping. It is limited in that it requires multiple texture units and it only supports *nearest* filtering and 8-bit depth texels (16-bit depth on GeForce [6]). For depth peeling, we need full depth buffer precision (24 bits) that necessitates the use of the `SGIX` shadowing extensions.

Depth Peeling

Depth peeling is the underlying technique that makes this approach for order-independent transparency possible. The standard depth test gives us the nearest fragment at each pixel, but there is also a fragment that is second nearest, third nearest, and so on. Standard depth testing gives us the *nearest* fragment without imposing any ordering restrictions, however, it does *not* give us any straightforward way to render the second nearest or n^{th} nearest surface.

Depth peeling solves this problem. The essence of what happens with this technique is that with *n* passes over a scene, we can get *n* layers deeper into the scene. For example, with 2 passes over the scene, we can extract the nearest and second nearest surfaces in a scene. We get both the depth and color (RGBA) information for each layer.

The images we get from peeling away depth are shown in Figure 3. It can be quite confusing to make sense of the images of layer 1 and beyond, because the notion of a "second nearest surface" is unintuitive. To help distinguish the various surfaces, the teapot is rendered with two-sided lighting (outside is red and inside is green), and the ground plane is drawn in blue. Note that the image labeled 'Layer 2' is in the shape of a teapot, but most of the fragments in that layer are from the ground plane (they are blue). Without the coloring, this would be difficult to interpret.

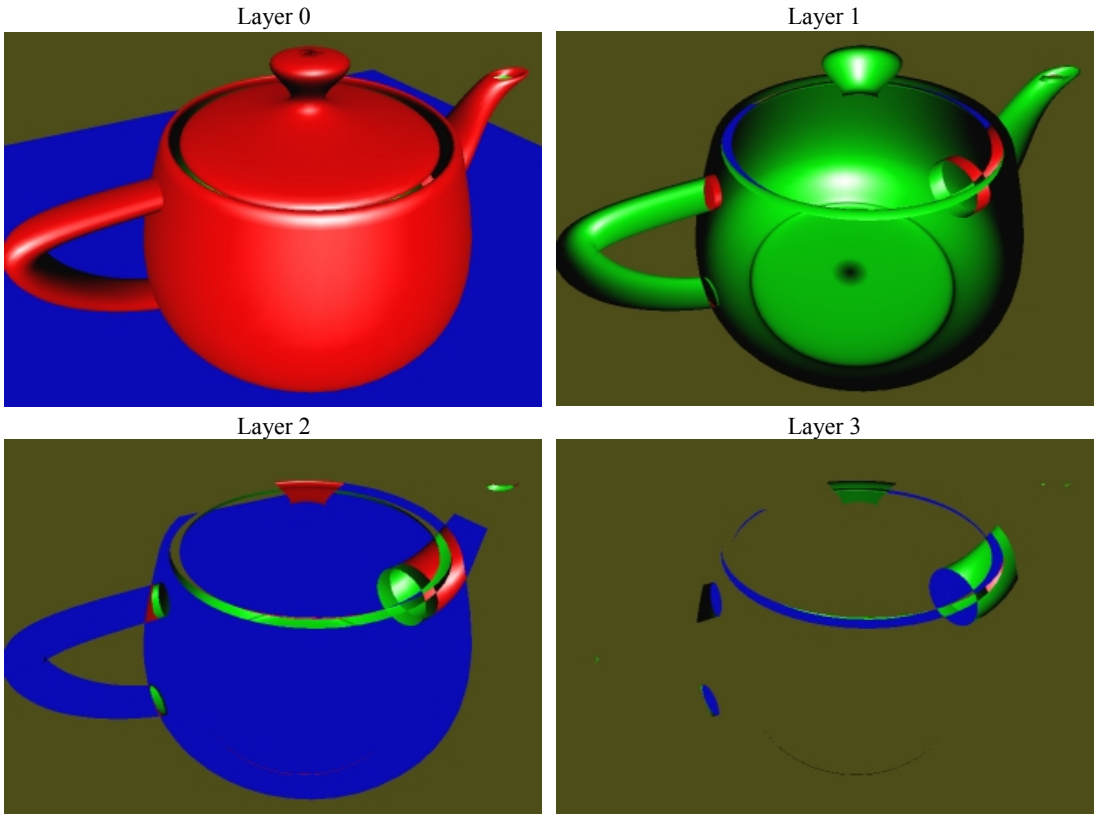


Figure 3. These images illustrate simple depth peeling. Layer 0 shows the nearest depths, layer 1 shows the second depths, and so on. Two-sided lighting with vivid coloring is used to help distinguish the surfaces.

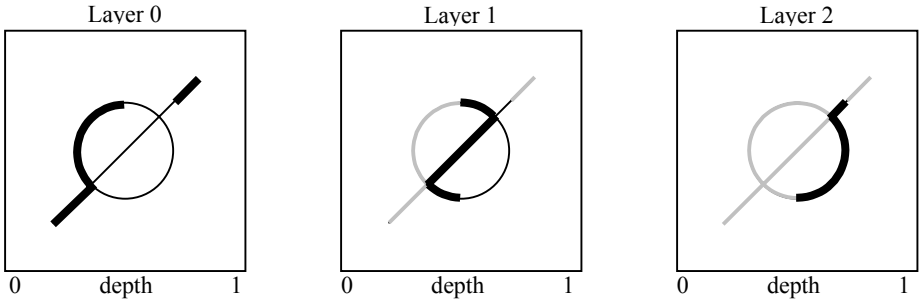


Figure 4. Depth peeling strips away depth layers with each successive pass. The frames above show the frontmost (leftmost) surfaces as bold black lines, hidden surfaces as thin black lines, and "peeled away" surfaces as light grey lines.

Figure 4 provides a more diagrammatic view of depth peeling. The diagrams there are analogous to the images in Figure 3, except we are now looking at a cross section of the view volume and highlighting each layer. It is evident from the view in Figure 4 that the depths vary within each layer, and the number of samples is decreasing. The peeling process clearly happens at the fragment level, so the pieces are generally not whole polygons.

The process of depth peeling is actually a straightforward multi-pass algorithm. In the first pass we render as normal, and the depth test gives us the nearest surface. In the second pass, we use the depth buffer computed in the first pass to "peel away" depths that are less than or equal to nearest depths from the first pass. The second pass generates a depth buffer for the *second* nearest surface, which can be used to peel away the first and second nearest surfaces in the third pass. The pattern is simple, but there is a catch. We need to perform *two depth tests per fragment* for it to work!

Multiple Depth Tests

The most natural way to describe this technique is to imagine that OpenGL supported multiple simultaneous depth units, each with its own depth buffer and associated state. We diverge from Diefenbach's dual depth buffer API in that we assume there are *n* depth units, all writeable, that are executed in sequential order. The first depth test to fail discards the fragment and terminates further processing. The pseudocode in Listing 1 implements depth peeling using two depth units.

In each pass except the first, depth unit 0 is used to peel away the previously nearest fragments while the depth unit 1 performs "regular" depth-buffering. We decouple the depth buffer from the depth unit because it simplifies the presentation of the

```
for (i=0; i<num_passes; i++)
{
    clear color buffer
    A = i % 2
    B = (i+1) % 2
    depth unit 0:
        if(i == 0)
            disable depth test
        else
            enable depth test
        bind buffer A
        disable depth writes
        set depth func to GREATER
    depth unit 1:
        bind buffer B
        clear depth buffer
        enable depth writes
        enable depth test
        set depth func to LESS
    render scene
    save color buffer RGBA as layer i
}
```

Listing 1. Pseudocode for depth peeling using multiple simultaneous depth buffers.

algorithm and more closely matches the semantics of `ARB_multitexture`. This decoupling is convenient because we need to use the depth buffer produced by depth unit 1 in pass *i* as the "peeling" depth buffer for depth unit 0 in pass *i*+1.

It is also worth mentioning that we only enable depth writes on depth unit 1. This will be important later.

Shadow Mapping as Depth Test

Shadow mapping *is* a depth test. For the purposes of our discussion, there are only a few major differences between shadow mapping and the depth-buffer algorithm:

- the shadow mapping comparison sets a fragment color attribute,
- the shadow mapping depth test is not tied to the camera position, and
- the shadow map (depth buffer) is not writeable during the shadow comparison (depth test).

It is not difficult to compensate for these differences. We write the results of the shadow mapping comparison to fragment alpha and use alpha test to discard fragments that fail the "depth test" we have chosen. We make the orientation and resolution of the shadow map identical to that of the camera. We can then use shadow mapping as a read-only depth test. This is good news, because this is all we needed to implement depth peeling as described in the previous section using our imaginary multiple depth test OpenGL. Except now, we can actually implement it using real OpenGL and *with hardware acceleration*!

An Invariance Issue

As simple as depth peeling sounds, it is actually pretty intolerant to variance. Due to the nature of the technique, many of the fragments generated in each pass will be on the razor's edge of the comparison. In our imaginary OpenGL that supports multiple depth tests, we would not expect variance to be a problem because we are re-using the same interpolator to compute depth the same way in each pass. Things are a little more complicated when we use shadow mapping as a depth test, though. This is primarily because

- z_w (window space z) is interpolated linearly in *window space* at the precision of the current depth buffer, and
- *r* and *q* are interpolated linearly in *clip space* (hyperbolically in window space) at high precision

The possible differences in precision and/or interpolation implementation are the hazards that cause variance. Consider the depth interpolation in Equation 1, which is linear in window space.

$$z_w = \frac{z_c}{w_c} = \alpha z_{w1} + (1-\alpha) z_{w2} = \alpha \frac{z_{c1}}{w_{c1}} + (1-\alpha)\frac{z_{c2}}{w_{c2}} \qquad (1)$$

Where z_w is window space z, z_c is clip space z, w_c is clip space w, and the numeric specifiers 1 and 2 indicate two points that are being interpolated. When we perform shadow mapping, we must interpolate quantities as texture coordinates which vary linearly in clip space, so we interpolate z_c and w_c as the r and q texture coordinates respectively, and use the r/q quotient to produce a value that varies linearly in window space. For the particular case we have been considering, shadow mapping from the camera's point of view we get Equations 2 and 3.

$$r = z_c = \frac{\alpha \frac{z_{c1}}{w_{c1}} + (1-\alpha)\frac{z_{c2}}{w_{c2}}}{\alpha \frac{1}{w_{c1}} + (1-\alpha)\frac{1}{w_{c2}}} \tag{2}$$

$$q = w_c = \frac{\alpha \frac{w_{c1}}{w_{c1}} + (1-\alpha)\frac{w_{c2}}{w_{c2}}}{\alpha \frac{1}{w_{c1}} + (1-\alpha)\frac{1}{w_{c2}}} \tag{3}$$

When we compute the r/q quotient, we recognize that the denominators in Equations 2 and 3 cancel, and that for our special case of shadow mapping from the camera's point of view, the numerator of Equation 3 is 1. This leaves only the numerator of Equation 2, which is *identical* to the expression in Equation 1. While this is algebraically true, the hardware may not be able to make some of these cancellations. For fragments with the same depth, hardware could evaluate the comparison shown in (4). The left side of the expression interpolates three quantities and performs four divides while the right simply interpolates one quantity.

$$\frac{\left(\left(\frac{z_c}{w_c}\right)\right)}{\left(\left(\frac{1}{w_c}\right)\right)} \leq \frac{\left(\left(\frac{w_c}{w_c}\right)\right)}{\left(\left(\frac{1}{w_c}\right)\right)} \frac{z_c}{w_c} \tag{4}$$

Luckily GeForce3's `NV_texture_shader` extension [8] supports a mode called `GL_DOT_PRODUCT_DEPTH_REPLACE_NV` that allows us to compute fragment depth using texture coordinates. The depth computed in this texture shader replaces the fragment depth that was computed in the rasterizer. This means that for GeForce3, we can

compute the depth that we store in the depth buffer in exactly the same way that we compute it when making the comparison. When we use this texture shader in generating our shadow map, there are no variances in the least significant bits. This is nice because it means we do not have to employ fudge factors to deal with LSB variance. The depth replace texture shader is very general, and this is a very simple use of it. Figure 5 illustrates the general operation of the depth replace texture shader.

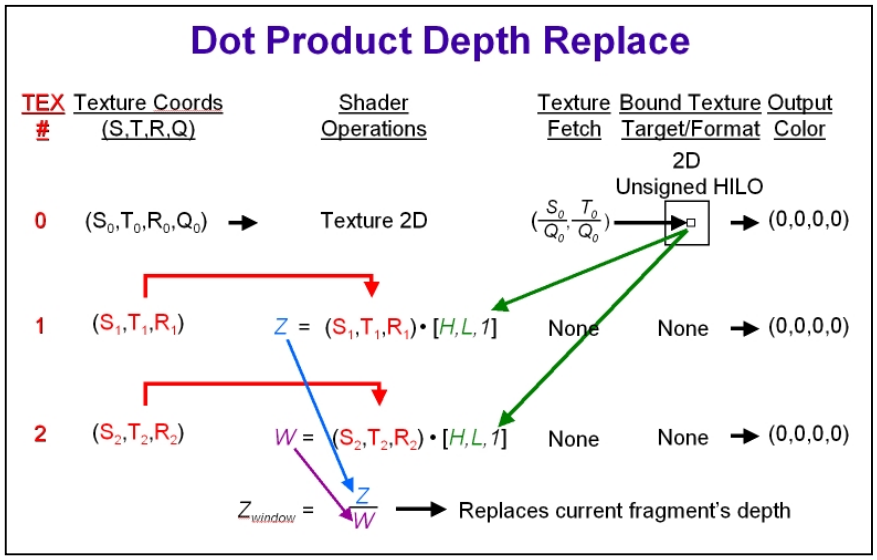


Figure 5. This diagram is a slightly modified slide taken from Dominé and Spitzer's GDC 2001 presentation on GeForce3 texture shaders. It describes the depth replace texture shader.

For our purposes, we really only want to interpolate z_c and w_c using a single texture coordinate for each, so we use a 1x1 `GL_UNSIGNED_HILO_NV` texture where H and L are zero. By definition, the 3rd component of an unsigned HILO is 1, so we perform a dot product of (S, T, R) with (0, 0, 1). In this way, we can interpolate the R coordinates of stages 1 and 2, and we use texture coordinate generation to make sure that R_1 is z_c and R_2 is w_c. When we perform the division z_c/w_c at each fragment, we are effectively interpolating window space depth in the same way that s/q does it in the subsequent shadow mapping pass.

There is one clarification we should make. When we consider the standard transformation pipeline, we often place the perspective divide before the viewport and depth range scale and bias. The depth replace texture shader and shadow mapping depth computation perform the divide (z_c/w_c and s/q respectively) as the *final operation*. This means that we must apply the depth range scale and bias *before* the perspective divide. Or, said another way, for depth replace and shadow mapping, we must transform coordinates into homogeneous window coordinates rather than homogeneous clip space.

```
glActiveTextureARB(GL_TEXTURE0_ARB);
simple_1x1_uhilo.bind();
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV, GL_TEXTURE_2D);

matrix4f m;
glActiveTextureARB(GL_TEXTURE1_ARB);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV, GL_DOT_PRODUCT_NV);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_NONE);
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();
eye_linear_texgen();  // set EYE_LINEAR texgen with identity planes
texgen(true);         // enable texgen on s,t,r, and q
glPopMatrix();
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glTranslatef( 0, 0,.5);
glScalef( 0, 0, .5);
reshaper.apply_perspective();  // apply the camera's perspective projection matrix
glMatrixMode(GL_MODELVIEW);

glActiveTextureARB(GL_TEXTURE2_ARB);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV, GL_DOT_PRODUCT_DEPTH_REPLACE_NV);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_NONE);
glPushMatrix();
glLoadIdentity();
eye_linear_texgen();  // set EYE_LINEAR texgen with identity planes
texgen(true);         // enable texgen on s,t,r, and q
glPopMatrix();
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
m(0,0) = 0;   m(0,1) = 0;   m(0,2) = 0;   m(0,3) = 0;
m(1,0) = 0;   m(1,1) = 0;   m(1,2) = 0;   m(1,3) = 0;
m(2,0) = 0;   m(2,1) = 0;   m(2,2) = 0;   m(2,3) = 1;  // move q to r
m(3,0) = 0;   m(3,1) = 0;   m(3,2) = 0;   m(3,3) = 0;
glMultMatrix(m);
reshaper.apply_perspective();  // apply the camera's perspective projection matrix
glMatrixMode(GL_MODELVIEW);

glActiveTextureARB(GL_TEXTURE3_ARB);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV, GL_TEXTURE_RECTANGLE_NV);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_NONE);

glActiveTextureARB(GL_TEXTURE0_ARB);
```

Listing 2. Example code for setting up depth replace texture shader for use in depth peeling.

The code in Listing 2 illustrates how to set up the `GL_DOT_PRODUCT_DEPTH_REPLACE_NV` texture shader to compute window *z* in a way that closely matches the standard projective texture mapping computation of window *z*. For illustrative purposes, we use *eye linear* texgen with an identity mapping for the *r* coordinate [0 0 1 0], and we use the texture matrix to perform the transforms. The most efficient approach would be to encode the transform in the texgen plane.

Another slightly odd aspect the depth replace texture shader is illustrated in the code in Listing 2. It is that the homogeneous window coordinate must be moved from the fourth row of the texture matrix into the *r* coordinate. This is because the dot product texture shaders only perform a 3-component dot product, so all quantities must be in the *s*, *t*, or *r* coordinates.

Putting It All Together

Now we have a way to compute the RGBA color for each unique depth at every pixel. These are stored as separate layers (or viewport-sized textures). All that remains is to compute the correct order-dependent color at each pixel by compositing the layers in order. Rendering each layer as viewport-sized textured quad does this. For back-to-front compositing a (`GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`) blending function is used.

Figure 5 illustrates the results of compositing of the layers into a final image. Note also that the bottom two images in Figure 5 look virtually (but not completely) identical. For completely correct results we should extract every semitransparent sample up to the first opaque sample, but in practice that is not necessary. The nature of the transparency computation is that samples further back have diminished effect, so truncation is a reasonable (and efficient) form of approximation. For example, the scene in Figure 5 is "good enough" after three layers.

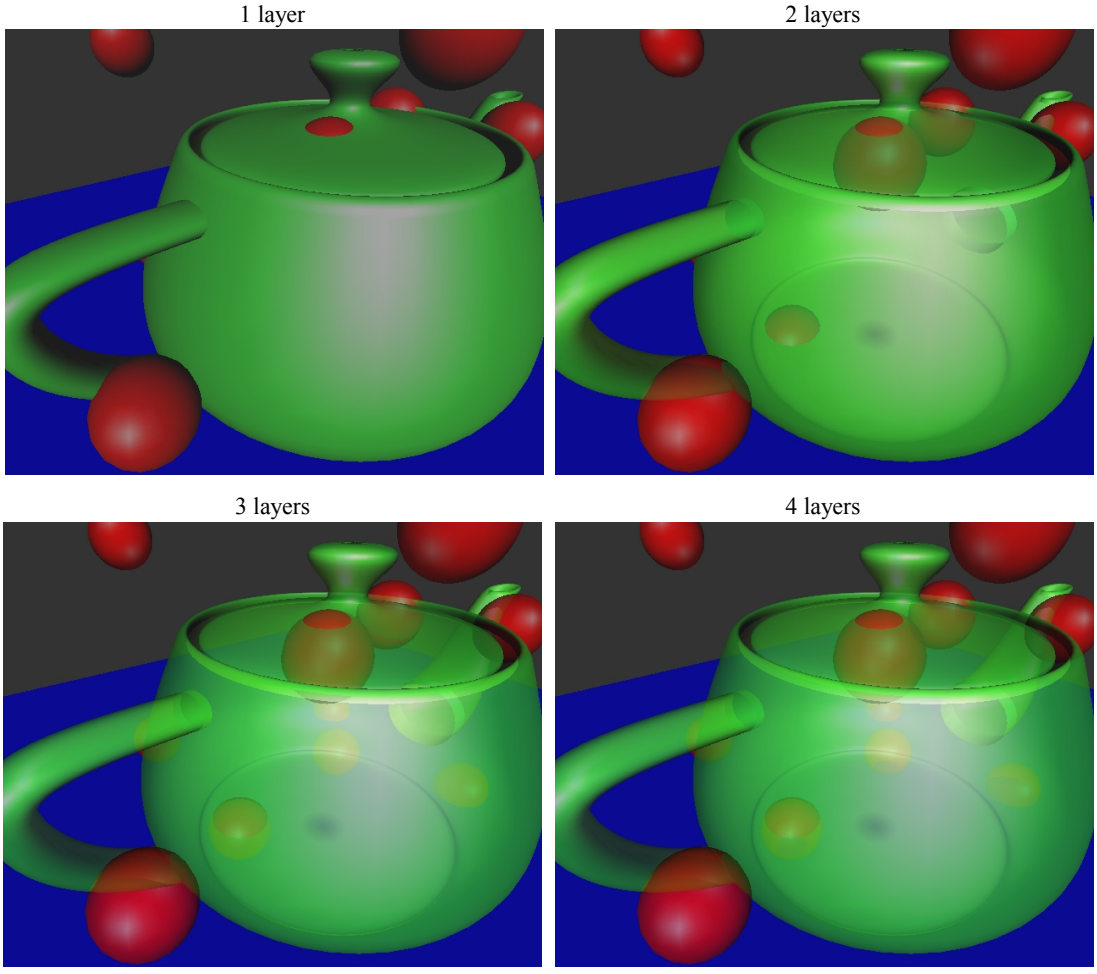


Figure 5. The depth peeled layers of the scene are correctly sorted per-fragment. If we simply save the color (RGBA) for each layer, we can composite them in depth-sorted order as a final pass. These images illustrate blending more layers for more correct transparency.

Conclusion

The technique presented is a straightforward and convenient way to render scenes with transparency because it does not require that the scene be rendered in sorted order, and it makes good use of graphics hardware. In addition, there may be no practical alternative to this approach of layer extraction and compositing for scenes that cannot be rendered in sorted order in a single pass.

Some of the figures in this paper come from the `layerz` and `order_independent_transparency` demos that can be found in the NVIDIA OpenGL SDK, which can be found at http://www.nvidia.com/developer. The demos only illustrate the technique described here, but many variations like those described in Diefenbach [3] are possible. The GDC 2001 presentations that were used in some figures are also available at the above web site.

Acknowledgements

Rui Bastos came up with the very clever idea of *depth peeling* using shadow mapping hardware support when he was considering hardware accelerated Woo shadow maps for GeForce3 (whitepaper on that topic to follow soon). I had help from Mark Kilgard on the appropriate texture coordinate generation setup for the *depth replace* texture shader that solves the invariance problem. He also provided invaluable feedback on early drafts of this paper.

References

[1] James F. Blinn. Hyperbolic interpolation. *IEEE Computer Graphics (SIGGRAPH) and Applications*, 12(4):89 94, July 1992.

[2] Rui Bastos. Personal communication. Feb 2001.

[3] Paul Diefenbach. Pipeline Rendering: Interaction and Realism Through Hardware-Based Multi-PassRendering. University of Pennsylvania, Department of Computer Science, Ph.D. dissertation, 1996.

[4] Cass Everitt. Projective texture mapping. http://www.nvidia.com/Marketing/developer/devrel.nsf/bookmark/BAB26B3133023C2088256A38007DE5E6. 2001

[5] Wolfgang Heidrich. High quality shading and lighting for hardware-accelerated rendering. http://www.cs.ubc.ca/~heidrich/Papers/phd.pdf. 1999.

[6] Mark Kilgard. GDC 2001 – Shadow mapping with today's OpenGL hardware. http://www.nvidia.com/Marketing/developer/devrel.nsf/bookmark/C89B7FC5F0497EFC88256A1800672176. March 2001.

[7] Abraham Mammen. Transparency and antialiasing algorithms Implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9(4): 43-55, July 1989.

[8] NVIDIA OpenGL Extensions Specifications. http://www.nvidia.com/Marketing/Developer/DevRel.nsf/bookmark/A86B9D846E815D628825681E007AA680. March 2001.

[9] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 1.2.1). www.opengl.org

[10] Mark Segal, et al. Fast shadows and lighting effects using texture mapping. In *Proceedings of SIGGRAPH '92*, pages 249-252, 1992.

[11] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of SIGGRAPH '78*, pages 270-274, 1978.