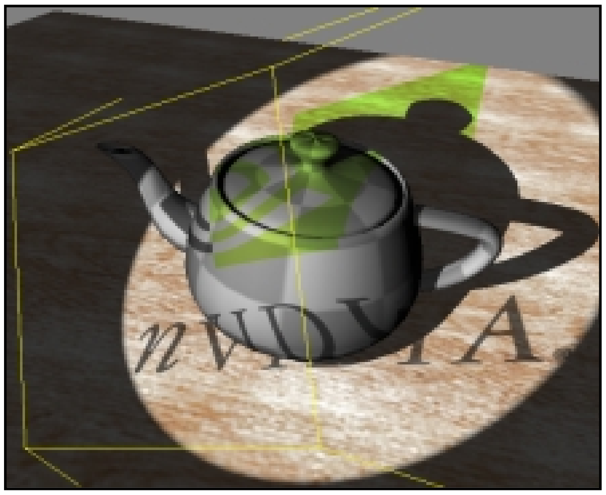
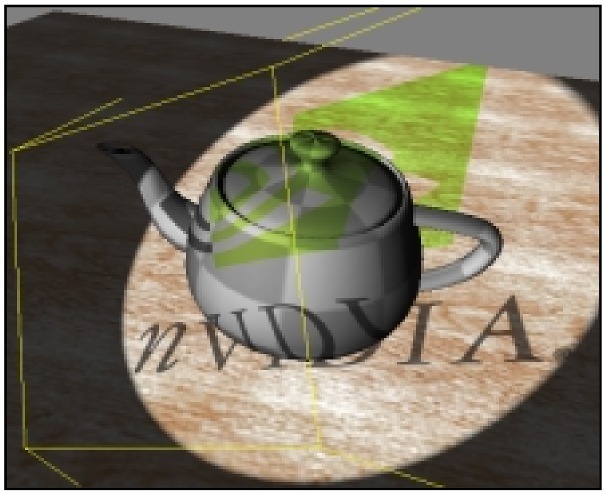
Hardware Shadow Mapping

Cass Everitt
cass@nvidia.com

Ashu Rege
arege@nvidia.com

Cem Cebenoyan
cem@nvidia.com





Introduction

Shadows make 3D computer graphics look better. Without them, scenes often feel unnatural and flat, and the relative depths of objects in the scene can be very unclear. The trouble with rendering high quality shadows is that they require a visibility test for each light source at each rasterized fragment. For ray tracers, adding an extra visibility test is trivial, but for rasterizers, it is not. Fortunately, there are a number of common cases where the light visibility test can be efficiently performed by a rasterizer. The two most common techniques for hardware accelerated complex shadowing are stenciled shadow volumes and shadow mapping. This document will focus on using shadow mapping to implement shadowing for spotlights.

Shadow mapping is an image-based shadowing technique developed by Lance Williams [8] in 1978. It is particularly amenable to hardware implementation because it makes use of existing hardware functionality – texturing and depth buffering. The only extra burden it places on hardware is the need to perform a high-precision scalar comparison for each texel fetched from the shadow map texture. Shadow maps are also attractive to application programmers because they are very easy to use, and unlike stenciled shadow volumes, they require no additional geometry processing.

Hardware accelerated shadow mapping [5] is available today on GeForce3 GPUs. It is exposed in OpenGL [4] through the SGIX_shadow and SGIX_depth_texture extensions [6], and in Direct3D 8 through a special texture format.

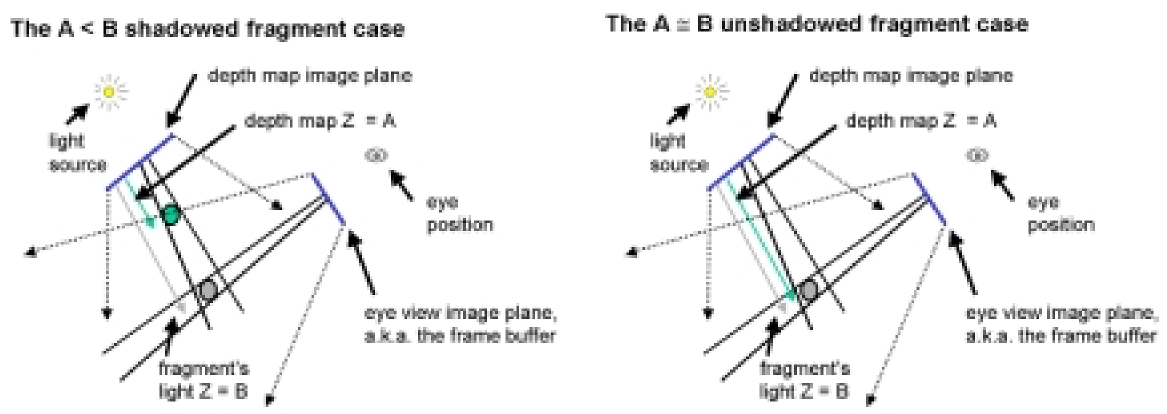


Figure 1. These diagrams were taken from Mark Kilgard's shadow mapping presentation at GDC 2001. They illustrate the shadowing comparison that occurs in shadow mapping.

How It Works

The clever insight of shadow mapping is that the depth buffer generated by rendering the scene from the light's point of view is a pre-computed light visibility test over the light's view volume. The light's depth buffer (a.k.a. the shadow map) partitions the view volume of the light into two regions: the shadowed region and the unshadowed region. The visibility test is of the form

$$p_z \le shadow\_map(p_x, p_y)$$

where p is a point in the light's image space. Shadow mapping really happens in the texture unit, so the comparison actually looks like:

$$\frac{p_r}{p_q} \le texture\_2D\left(\frac{p_s}{p_q}, \frac{p_t}{p_q}\right).$$

Note that this form of comparison is identical to the depth test used for visible surface determination during standard rasterization. The primary difference is that the rasterizer always generates fragments (primitive samples) on the regular grid of the eye's discretized image plane for depth test, while textures are sampled over a continuous space at irregular intervals. If we made an effort to sample the shadow map texture in the same way that we sample the depth buffer, there would be no difference at all. In fact, we can use shadow maps in this way to perform more than one depth test per fragment [2].

Figure 1 illustrates the depth comparison that takes place in shadow mapping. The eye position and image plane are shown, but they are not relevant to the visibility test because shadowing is view-independent.

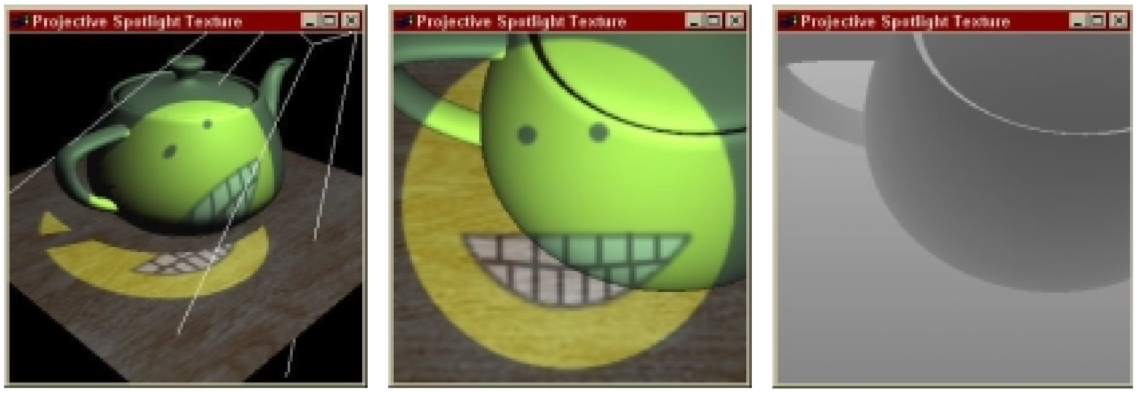

Projective Spotlight Texture
Projective Spotlight Texture
Projective Spotlight Texture

Figure 2. A shadow mapped scene rendered from the eye's point of view (left), the scene as rendered from the light's point of view (center), and the corresponding depth/shadow map (right).

How To Do It

The basic steps for rendering with shadow maps are quite simple:

- render the scene from the light's point of view,
- use the light's depth buffer as a texture (shadow map),
- projectively texture the shadow map onto the scene, and
- use "texture color" (comparison result) in fragment shading.

Figure 2 shows an example scene with shadows, the same scene shown from the light's point of view, and the corresponding shadow map (or depth buffer). Note that samples that are closer to the light are darker than samples that are further away.

Since applications already have to be able to render the scene, rendering from the light's point of view is trivial. If it is available, polygon offset should be used to push fragments back slightly during this rendering pass.

Why Is Polygon Offset Needed?

If implemented literally, the light visibility test described in the previous section is prone to self-shadowing error due to it's razor's edge nature in the case of unshadowed objects. In the hypothetical "infinite resolution, infinite precision" case, surfaces that pass the visibility test would have depth *equal* to the depth value stored in the shadow map. In the real world of finite precision and finite resolution, precision and sampling issues cause problems. These problems can be solved by adding a small bias to the shadow map depths used in the comparison.

If the problem were only one of precision, a constant bias of all the shadow map depths would be sufficient, but there is also a less obvious sampling issue that affects the magnitude of bias necessary. Consider the case illustrated in Figure 3. When the geometry is rasterized from the eye's point of view, it will be sampled in different

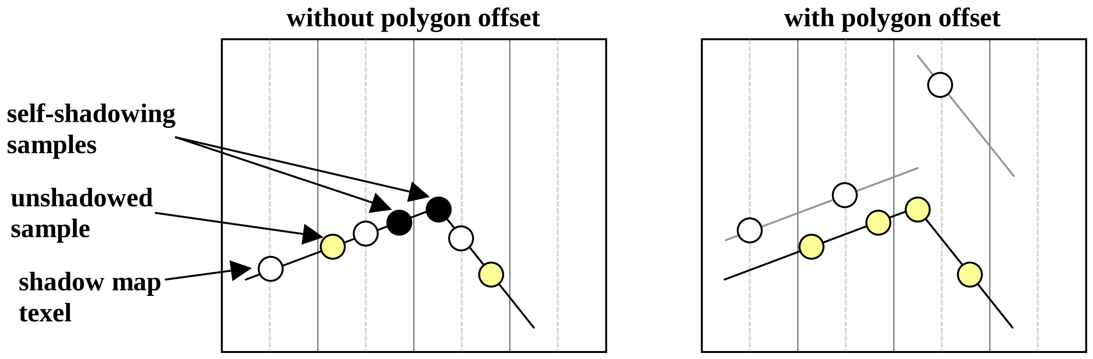


Figure 3. These figures illustrate the need for polygon offsetting to eliminate self-shadowing artifacts. The variable sampling location necessitates the use of z slope-based offset.

locations than when it was rasterized from the light's point of view. The difference in the depths of the samples is based on the slope of the polygon in light space, so in order to account for this we must supply a positive "slope factor" (typically about 1.0) to the polygon offset.

Direct3D does not expose polygon offset, so applications must provide this bias through matrix tweaks. This approach is workable, but because it fails to account for z slope, the uniform bias is generally much larger than it would otherwise need to be, which may introduce incorrectly unshadowed samples, or "light leaking".

The depth map as rendered from the light's point of view *is* the shadow map. With OpenGL, turning it into a real texture requires copying it into texture memory via glCopyTex{Sub}Image2D(). Even though the copy is card-local, it is still somewhat expensive. Direct3D's render-to-texture capability makes this copy unnecessary. You can render directly to the shadow map texture. This render-to-texture capability will also be available soon in OpenGL through extensions.

Once the shadow map texture is generated, it is projectively textured onto the scene. For shadow mapping, we compute 3D projective texture coordinates, where *r* is the sample depth in light space, and *s* and *t* index the 2D texture. Figure 4 shows these quantities, which are compared during rendering to determine light visibility.

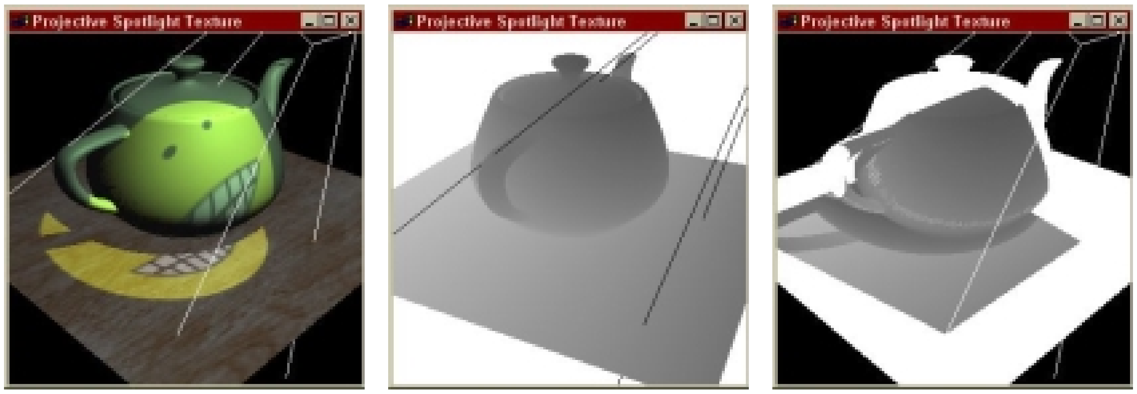

Projective Spotlight Texture
Projective Spotlight Texture
Projective Spotlight Texture

Figure 4. A shadow mapped scene (left), the scene showing each sample's distance from the light source (center), and the scene with the shadow map shadow map projected onto it (right).

The final step in rendering shadows is to actually factor the shadow computation result into the shading equation. The result of the comparison is either 1 or 0, and it is returned as the texture color. If linear filtering is enabled, the comparison is performed at the four neighboring shadow map samples, and the results are bilinearly filtered just as if they had been colors.

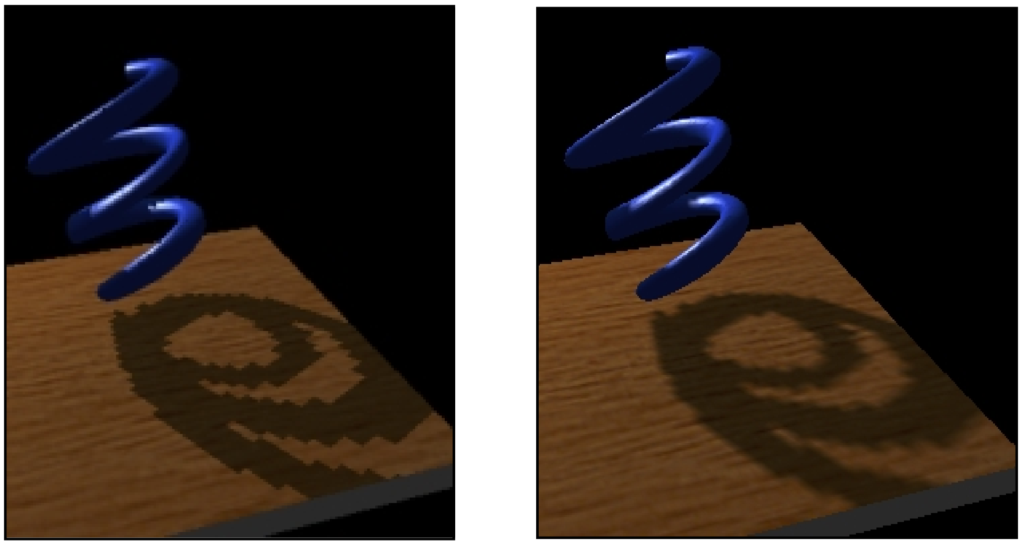


Figure 5. A very low resolution shadow map is used to demonstrate the difference between nearest (left) and linear (right) filtering for shadow maps. Credit: Mark Kilgard.

With GeForce3 hardware, it is easiest to use NV_register_combiners to implement the desired per-fragment shading based on the shadow comparison. One simple approach is to use the shadowing result directly to modulate the diffuse and specular intensity. Kilgard points out [3] that leaving some fraction of diffuse intensity in helps keep shadows areas from looking too “flat”.

OpenGL API Details

Support for shadow mapping in OpenGL is provided by the SGIX_shadow and SGIX_depth_texture extensions. The SGIX_shadow extension exposes the per-texel comparison as a texture parameter, and SGIX_depth_texture defines a texture internal format of DEPTH_COMPONENT, complete with various bits-per-texel choices. It also provides semantics for glCopyTex{Sub}Image*() calls to read from the depth buffer when performing a copy.

Direct3D API Details

Support for shadow mapping in Direct3D is provided by special depth texture formats exposed in drivers version 21.81 and later. Support for both 24-bit (D3DFMT_D24S8) and 16-bit (D3DFMT_D16) shadow maps is included.

Setup

The following code snippet checks for hardware shadow map support on the default adapter in 32-bit color:

```
HRESULT hr = pD3D->CheckDeviceFormat(
        D3DADAPTER_DEFAULT,        //default adapter
        D3DDEVTYPE_HAL,            //HAL device
        D3DFMT_X8R8G8B8,           //display mode
        D3DUSAGE_DEPTHSTENCIL,     //shadow map is a depth/s surface
        D3DRTYPE_TEXTURE,          //shadow map is a texture
        D3DFMT_D24S8               //format of shadow map
);
```

Note that since shadow mapping in Direct3D relies on "overloading" the meaning of an existing texture format, the above check does not guarantee hardware shadow map support, since it's feasible that a particular hardware / driver combo could one day exist that supports depth texture formats for another purpose. For this reason, it's a good idea to supplement the above check with a check that the hardware is GeForce3 or greater.

Once shadow map support has been determined, you can create the shadow map using the following call:

```
pD3DDev->CreateTexture(texWidth, texHeight, 1,
        D3DUSAGE_DEPTHSTENCIL, D3DFMT_D24S8, D3DPOOL_DEFAULT,
        &pTex);
```

Note that you **must** create a corresponding color surface to go along with your depth surface since Direct3D requires you to set a color surface / z surface pair when doing a SetRenderTarget(). If you're not using the color buffer for anything, it's best to turn off color writes when rendering to it using the D3DRS_COLORWRITEENABLE renderstate to save bandwidth.

Rendering

Rendering uses the same ideas as in OpenGL: you render from the point of view of the light to the shadow map you created, then set the shadow map texture in a texture stage and set the texture coordinates in that stage to index into the shadow map at (s / q, t / q) and use the depth value (r / q) for the comparison. There are a few Direct3D-specific idiosyncrasies to be aware of, however:

- The (z / w) value used to compare with the value in the shadow map is in the range [0..$2^{bitdepth}$-1], not [0..1], where ‘bitdepth’ is the bitdepth of the shadowmap (24 or 16 bits). This means you have to put an additional scale factor into your texture matrix.
- Direct3D addresses pixels and texels in different ways [1], where integral screen coordinates address pixel centers and integral texture coordinates address texel boundaries. You need to take this into account when addressing the shadow map. There are two ways to do this: either offset the viewport by half a texel when rendering the shadow map, or offset by half a texel when addressing the shadow map.
- As stated earlier, there is no native polygon offset support in Direct3D. The closest thing is D3DRS_ZBIAS, but this doesn’t help us when shadow mapping since it can only be used to bias depth a constant amount *towards* the camera, not away. Instead we can get similar functionality, albeit without taking into account polygon slope, by adding a small bias amount to our texture matrix.

Here is a sample texture matrix that takes into account these limitations:

```
float fOffsetX = 0.5f + (0.5f / fTexWidth);
float fOffsetY = 0.5f + (0.5f / fTexHeight);
D3DXMATRIX texScaleBiasMat( 0.5f,     0.0f,     0.0f,      0.0f,
                            0.0f,    -0.5f,     0.0f,      0.0f,
                            0.0f,     0.0f,     fZScale,   0.0f,
                            fOffsetX, fOffsetY, fBias,     1.0f );
```

Where fZScale is the ($2^{bitdepth}$-1) and fBias is a small negative value. Note that this matrix is applied post-projection, **not** in eye space.

Once the texture coordinates have been setup properly, the hardware will automatically compare (r / q) > shadowMap[s / q, t / q] and return zero to indicate in shadow or one to indicate in light (or potentially something in between if you’re on the shadow edge and using D3DTEXF_LINEAR). The following pixel shader shows a simple use of shadow mapping (but note that you don’t have to use pixel shaders to use shadow maps, DirectX7-style texture stage states work as well):

```
tex t0   // normal map
tex t1   // decal texture
tex t2   // shadow map
dp3_sat r0, t0_bx2, v0_bx2  //light vector is in v0
mul r0, r0, t2   //modulate lighting contribution by shadow result
mul r0, r0, t1   //modulate lighting contribution by decal
```

Advantages and Limitations

As with any technique, shadow mapping has certain advantages and limitations to be aware of. The fact that it is image-based turns out to be both an advantage and a limitation. It's advantageous, because it doesn't require additional application geometry processing, it works well with GPU-created and GPU-altered geometry and correctly handles fragment culling like alpha test. The associated limitation is that because it's image based, it works well for spotlights, but not point light sources. One could imagine a cube map –based shadow mapping system, but they would require six 90-degree frusta, which would each need to be fairly high resolution, and five more passes over the geometry to generate the shadow map.

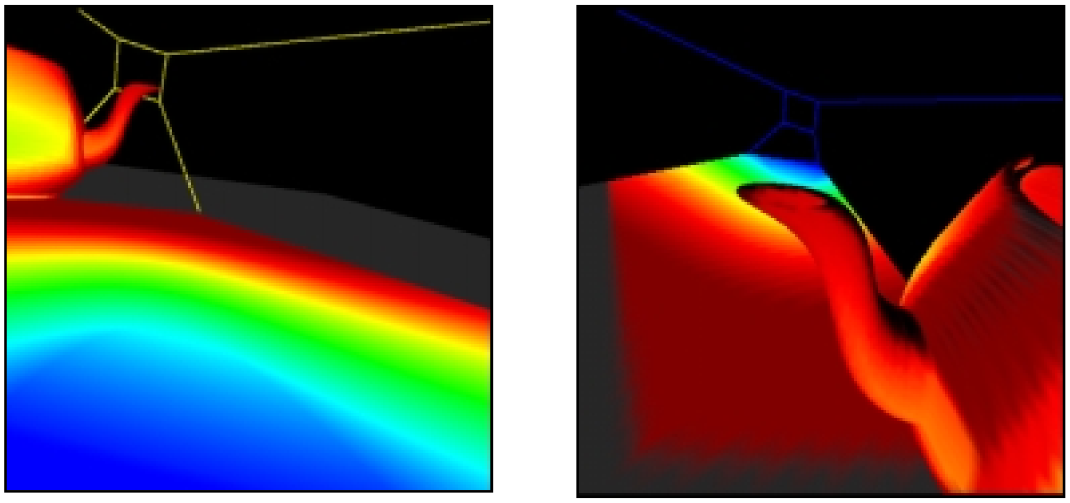


Figure 6. The "dueling frusta" problem occurs when the spotlight points toward the eye. The eye's view (left) shows the variation in sampling frequency of the shadow map, blue being the highest . The light's view (right) shows the very small portion of the light's image plane needs high frequency sampling.

Along the same lines, the quality of shadow mapping depends on how well the shadow map sampling frequency matches the eye's sampling frequency. When the eye and light have similar location and orientation, the sampling frequencies match pretty well, but when the light and eye are looking toward each other, the sampling frequencies rarely match well. Figure 6 illustrates this "dueling frusta" situation.

Another problem that comes up with any projective texture mapping is the phantom "negative projection". This is actually pretty simple to remove at the cost of an additional texture unit, or per-vertex color. The goal is just to make sure that the shadow test always returns "shadowed" for surfaces behind the light.

Finally, the polygon offset fudge factor, while quite adequate for virtually all uses of shadow mapping, can be a bit dissatisfying. Andrew Woo [9] suggested an alternative shadow map generation that is produced from averaging the nearest and second-nearest depth layers from the light's point of view. This technique can actually be implemented as a two-pass technique on GeForce3 hardware using the depth peeling technique described in [2] and with a slight twist. In the second pass, the shadow map is used to peel away the nearest surfaces, but all depths are computed as the average of the fragment's original depth and the nearest depth at that fragment's (x,y). The nearest surface (that is not peeled away), is then the average of the first and second nearest fragments!

Wang and Molnar introduce another technique to reduce the need for polygon offset [7]. Their technique works by rendering only back-faces into the shadow map, relying on the observation that back-face z-values and front-face z-values are likely far enough apart in z to not falsely self-shadow. This only helps front-faces, of course, but back-faces (with respect to the light) are, by definition, not in light, which helps hide artifacts. Note that this algorithm only works for closed polygonal objects.

Computing Transformations for Shadow Mapping

Computing the transformations required for shadow mapping can be somewhat tricky. This section provides details on the various transformations that need to be applied during the two render passes. While this section provides details for the OpenGL case, the transformations required for Direct3D are very similar with the main exception being that the texture coordinate generation is done directly via a matrix instead of the *texgen* planes. Also, keep in mind that the scale-bias matrix in Direct3D requires an additional offset to account for the discrepancy between pixel and texel coordinates as mentioned earlier, and that *eye linear* texgen is called D3DTSS_TCI_CAMERASPACEPOSITION.

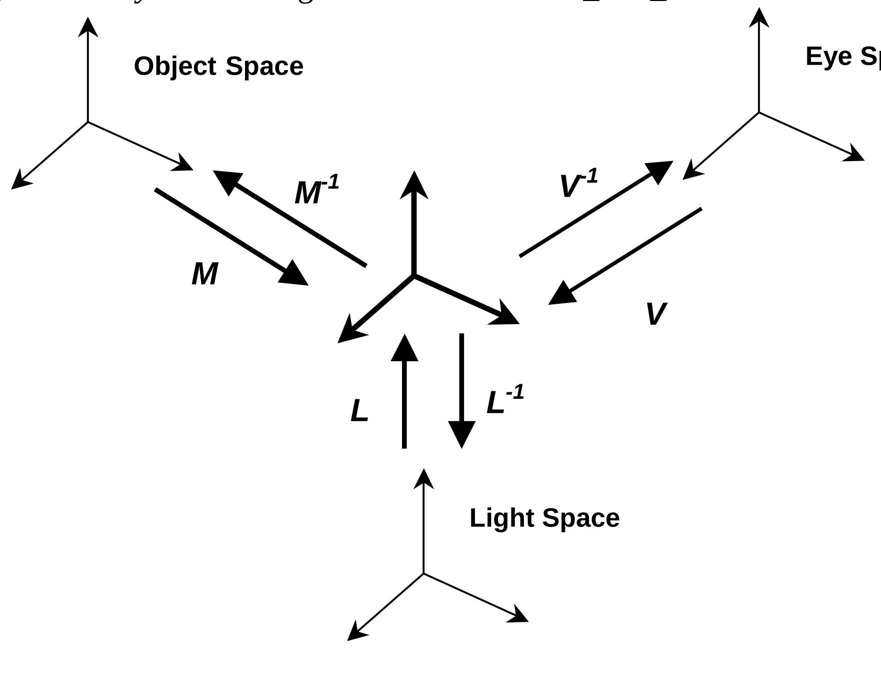


Figure 7: Schematic view of the basic transformations involved in shadow mapping.

Figure 7 shows the three primary transformations (and inverses) used in shadow mapping. Note that we use the convention of using the *forward* transforms as going *to* world coordinates. The standard 'modelview' matrix using the above notation will therefore be: $\boldsymbol{V^{-1}M}$. In addition to the above transformations, we also have to account for the projections involved in the two passes – these could be different depending on the frusta for the light and eye. The projection transformation will also be applied during the texture coordinate generation phase which is depicted in Figure 8 for OpenGL. As shown in the figure, two transformations are applied to the eye coordinates – the *texgen* planes, and the *texture matrix*. For *eye linear* texgen planes, OpenGL will automatically multiply the eye coordinates with the inverse of the modelview matrix *in effect when the planes are specified*. (See Appendix A for a more detailed explanation of the texgen planes in the eye linear case.)

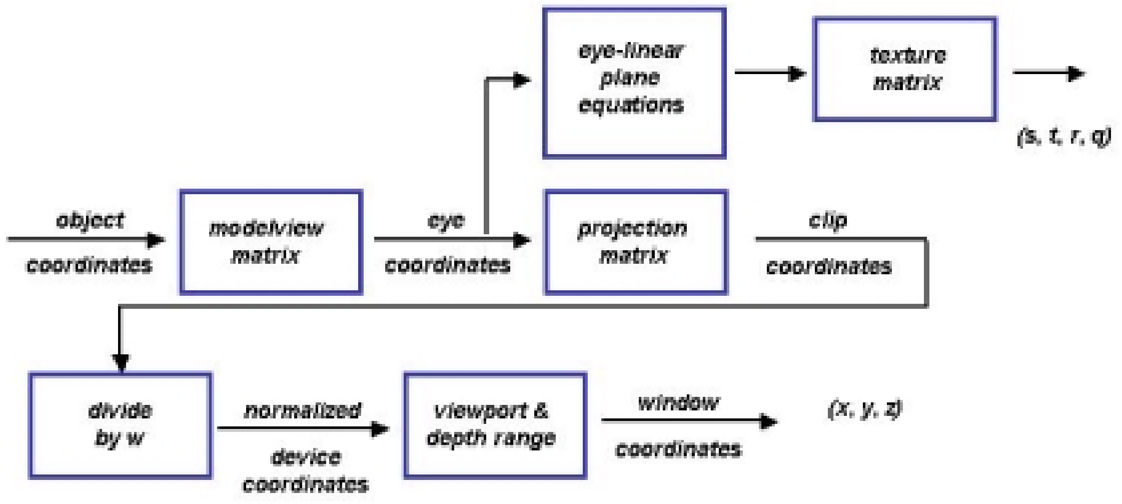


Figure 8: OpenGL Transformation Pipeline

The resulting texture coordinates are therefore computed as:

$$\mathbf{[x_e,y_e,z_e,w_e]^T = (modelview)\ [x_o,y_o,z_o,w_o]^T}$$

$$\mathbf{E_e = E_{po}(modelview_{po})^{-1}}$$

$$\mathbf{[s,t,r,q]^T = T\ E_e\ [x_e,y_e,z_e,w_e]^T}$$

Equation 1

Here the subscript '**o**' denotes object space coordinates, and the subscript '**e**' refers to eye space coordinates, $\mathbf{modelview_{po}}$ is the modelview matrix in effect when the eye linear texgen plane equations are specified, $\mathbf{E_{po}}$ is the matrix composed of the eye linear plane equations *as specified to OpenGL* (i.e. in their own object space), $\mathbf{E_e}$ is the matrix composed of the transformed plane equations (these are the plane equations that are actually stored by OpenGL), **T** is the texture matrix, and **modelview** is the modelview matrix when rendering the scene geometry.

Setting Up the Transformations

We want to set the transformations in Equation 1 to compute texture coordinates **(s,t,r,q)** such that **(s/q,t/q)** will be the fragment's location within the depth texture, and **r/q** will be the window-space z of the fragment relative to the light's frustum. In other words, we want to compute:

$$[s,t,r,q]^T = S\, P_{light}\, L^{-1}\, M\, [x_o,y_o,z_o,w_o]^T$$

Equation 2

Here, **S** is the scale-bias matrix, given by:

$$\begin{bmatrix} ½ & 0 & 0 & ½ \\ 0 & ½ & 0 & ½ \\ 0 & 0 & ½ & ½ \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

P_{light} is the projection matrix for the light frustum. The "texgen matrix" (E_e), however, is applied to *eye* coordinates $[x_e,y_e,z_e,w_e]^T$ but we want to generate the coordinates in *light* space, since that is where the depth map computation takes place. So we need to take $[x_e,y_e,z_e,w_e]^T$ back into world space by applying the transform **V**. That is, we want to compute $[s,t,r,q]^T$ as:

$$[s,t,r,q]^T = S\, P_{light}\, L^{-1}\, V\, [x_e,y_e,z_e,w_e]^T$$

Equation 3

Note that the right hand side of Equation 3 reduces to $S\, P_{light}\, L^{-1}\, M\, [x_o,y_o,z_o,w_o]$, precisely what we want. A straightforward way to compute Equation 3 is to set $\mathbf{modelview}_{po}$ to *identity* and set:

$$T\, E_{po} = S\, P_{light}\, L^{-1}\, V$$

Equation 4

The first observation is that we have two matrices **T** (the texture matrix) and E_{po} (the eye linear texgen planes specified to OpenGL) so we can compute Equation 4 in several ways. Since we are going to have to set the eye linear planes in any case, the less expensive thing to do is to not set the texture matrix at all, and use the texgen matrix **G** for the entire computation[†], i.e., set

$$\mathbf{E_{po}} = \mathbf{S}\ \mathbf{P_{light}}\ \mathbf{L^{-1}}\ \mathbf{V}$$

Equation 5

This assumes that the modelview matrix, $\mathbf{modelview_{po}}$, was identity at the time the texgen planes are set. Another improvement is to make use of the fact that OpenGL automatically multiplies $\mathbf{[x_e,y_e,z_e,w_e]^T}$ with $(\mathbf{modelview_{po}})^{-1}$ for eye linear texgen. The sole purpose of using **V** in Equation 5 is to eliminate $\mathbf{V^{-1}}$. If we set $\mathbf{modelview_{po}} = \mathbf{V^{-1}}$, then OpenGL will do the elimination for us and we can avoid having to compute **V**, the inverse of the view matrix. The steps can be summarized as follows:

First Pass (Depth Map Generation)

- Render from light's point of view. Set projection matrix to $\mathbf{P_{light}}$. Set the view portion of the modelview matrix to $\mathbf{L^{-1}}$.
- Render scene (with appropriate modeling transform(s) **M**).

Second Pass (Depth Map Comparison)

- Render from eye's point of view. Set projection matrix to be $\mathbf{P_{eye}}$. Set the view portion of the modelview matrix to be $\mathbf{V^{-1}}$.
- Set texgen to be EYE_LINEAR. Specify texgen planes as $\mathbf{E_{po}} = \mathbf{S}\ \mathbf{P_{light}}\ \mathbf{L^{-1}}$
- Render scene (with appropriate modeling transform(s) **M**)

Conclusions

Shadow mapping is an easy-to-use shadowing technique that makes 3D rendering just look better. It enjoys hardware acceleration on GeForce3 GPUs. There is example source code in the NVSDK (hw_shadowmaps_simple, hw_woo_shadowmaps) that demonstrate the technique, and the corresponding OpenGL extensions. Please direct questions or comments to cass@nvidia.com.

References

[1] Craig Duttweiler. Mapping Texels to Pixels in Direct3D. http://developer.nvidia.com/view.asp?IO=Mapping_texels_Pixels.

[†] Note that this technique of collapsing the texture and texgen matrices works in our case because we are setting all four planes, and using the same mode for all four planes. In general, each coordinate can have a different mode (eye linear, object linear, sphere map…) and the technique may not be applicable.

[2] Cass Everitt. Interactive Order-Independent Transparency. Whitepaper: http://developer.nvidia.com/view.asp?IO=Interactive_Order_Transparency.

[3] Mark Kilgard. Shadow Mapping with Today's Hardware. Technical presentation: http://developer.nvidia.com/view.asp?IO=cedec_shadowmap.

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

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

[6] OpenGL Extension Registry. http://oss.sgi.com/projects/ogl-sample/registry/.

[7] Yulan Wang and Steven Molnar. Second-Depth Shadow Mapping. UNC-CS Technical Report TR94-019, 1994.

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

[9] Andrew Woo, P. Poulin, and A. Fournier. "A Survey of Shadow Algorithms," IEEE Computer Graphics and Applications: vol 10(6), pages 13-32, 1990.

Appendix A: Another Way to Think about EYE_LINEAR planes in OpenGL

An unfortunate thing about EYE_LINEAR texgen in OpenGL is that the name implies that the plane equations are specified in eye space, when they are, in fact, specified in their own object space. There are two ways one can think about the planes specified in EYE_LINEAR texgen. As mentioned earlier, OpenGL will automatically multiply the planes specified with $(\mathbf{modelview}_{\mathbf{po}})^{-1}$, i.e. the inverse of the modelview matrix in effect when the planes are specified. From Equation 1 we see that the net effect is to map the vertex position in eye coordinates $[\mathbf{x_e},\mathbf{y_e},\mathbf{z_e},\mathbf{w_e}]^{\mathbf{T}}$ back to the 'object space' defined by $(\mathbf{modelview}_{\mathbf{po}})^{-1}$. The transformed coordinates are then evaluated at each plane in this object space to get the texture coordinates. An alternate way to think about the texgen planes is to consider the matrix $\mathbf{E_e} = \mathbf{E_{po}}(\mathbf{modelview})^{-1}$, which defines a map whose domain is *eye* space, with the planes $\mathbf{E_{po}}$ being specified in object space. $\mathbf{E_e}$ therefore defines the *transformed* planes in eye space. In either case, the planes are being *specified* in the 'object space' defined by $(\mathbf{modelview}_{\mathbf{po}})^{-1}$ and *not* in eye space.

In the shadow mapping case described earlier, the modelview matrix is set to $\mathbf{V}^{-1}$ when the texgen plane equations are specified. This is the same thing as saying that we are specifying the plane equations in *world* space. If the modelview matrix were set to identity, then we would be specifying the equations in *eye* space. The same is true if we were specifying vertex positions.

We could set the modelview matrix to $\mathbf{V}^{-1}\mathbf{L}$, and specify the plane equations in *light* space. This might be handy, because we would only need to update our plane equations if the light's projection ($\mathbf{P}_{\mathbf{light}}$) changed. We could even put the *whole* transformation into the modelview matrix as $\mathbf{V}^{-1}\mathbf{L}\mathbf{P}_{\mathbf{light}}{}^{-1}\mathbf{S}^{-1}$. In this case, the texgen planes are *always* just specified as identity ($\mathbf{E}_{\mathbf{po}} = \mathbf{I}$)!