# Projective Texture Mapping

Cass Everitt
cass@nvidia.com

## Introduction

**Projective texture mapping** is a method of texture mapping described by Segal [3] that allows the texture image to be ***projected*** onto the scene as if by a slide projector. Figure 1 shows some example screen shots from the `projspot` demo, available in the NVIDIA OpenGL SDK. Projective texture mapping is useful in a variety of lighting techniques, including shadow mapping [4]. This document provides some background and describes the steps involved in projective texture mapping in OpenGL.

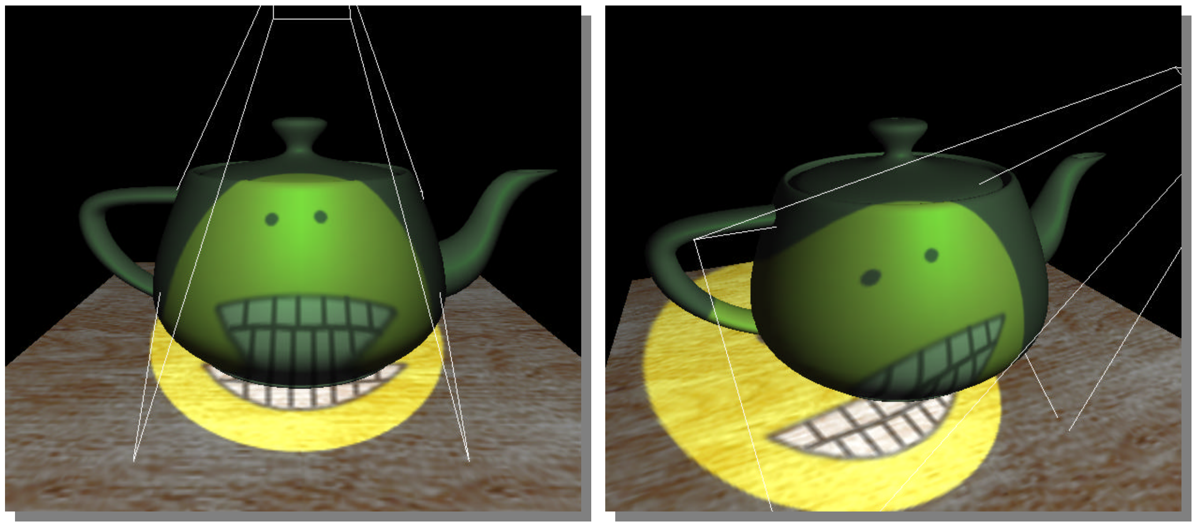


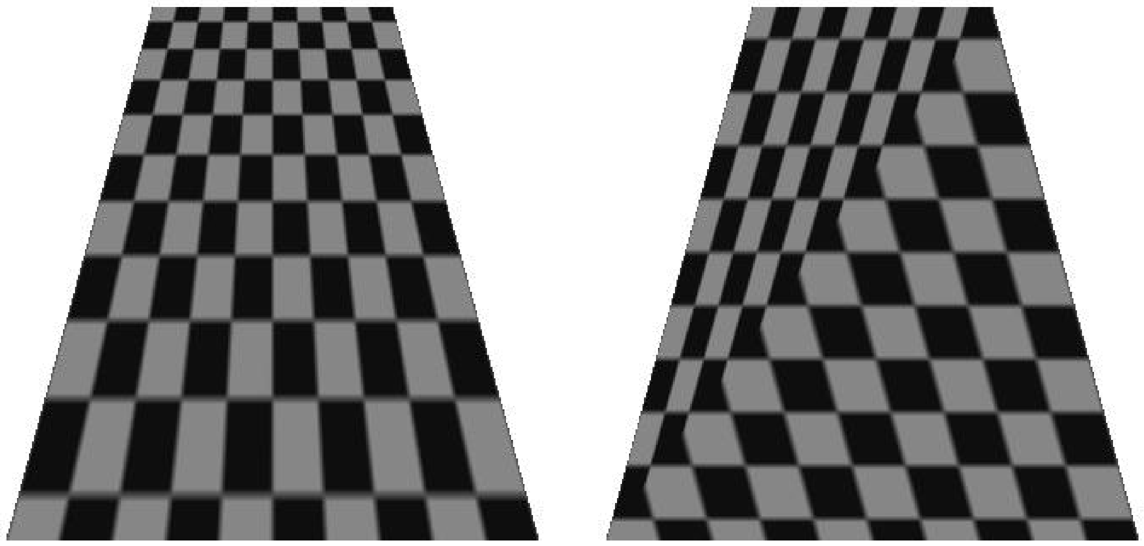
**Figure 1. Two different views of a smiley face texture projected onto the scene.**

Projective texture mapping refers both to the way texture coordinates are assigned to vertices, and the way they are computed during rasterization of primitives. We usually think of texture mapping as "the application of a texture image to a primitive," and while it certainly is that – there is more math going on than most folks think. If you have ever written your own rasterizer with support for mipmap filtered, perspective-correct, projective texture mapping, you no doubt became aware of the many subtle issues involved. We will begin by discussing the way that texture coordinates are computed during rasterization, and then we will discuss methods for assigning the texture coordinates to the vertices. We do not discuss filtering here, but there is a paper on ***Anisotropic Filtering*** at the NVIDIA developer web site that provides a good introduction to that topic.

## Rasterization Details

When performing projective texture mapping, we use *homogeneous* texture coordinates, or coordinates in *projective space*. When performing non-projective texture mapping, we use *real* texture coordinates, or coordinates in *real space*. For projective 2D texture mapping, the 3-component homogeneous coordinate (s,t,q) is interpolated over the primitive and then at each fragment, the interpolated *homogeneous* coordinate is projected to a *real* 2D texture coordinate, (s/q, t/q), to index into the texture image. For non-projective 2D texture mapping, the 2-component real coordinate (s,t) is interpolated over the primitive and used directly to index into the texture image.

The images in Figure 2 illustrate the difference between interpolating in projective space and real space. In projective space, the two triangles have the same screen-space coordinate gradients, but that is impossible in real space. These images were generated with the `qcoord` demo, which is available in the NVIDIA OpenGL SDK.



**(a) projective space interpolation** **(b) real space interpolation**

**Figure 2. These images illustrate interpolation in (a) projective space and (b) real space.**

One important note about the images in Figure 2 is that the geometry is a trapezoid that is parallel to the image plane. It is ***not*** a perspective projection of an elongated rectangle extending into the distance. This is an important distinction because the effect we see here looks like an illustration of the difference between **perspective-correct** and **non-perspective-correct** interpolation. The idea is related because perspective correction is about correctly interpolating in post-perspective real space (window coordinates), parameters that vary linearly in homogenous clip (or eye) space [1].

While the concepts of perspective-correct interpolation and projective texture mapping are closely related, they are, in fact, orthogonal. Perspective-correct (or non-perspective-correct) interpolation describes how each of s, t, q, and any other parameters that vary from vertex to vertex are interpolated over a polygon. The fact that s and t are divided by q for projective texture mapping is mostly irrelevant to the machinery responsible for interpolating these quantities. I say *mostly* because, early OpenGL implementations performed both projection and perspective correction with a single divide (by q/w). While this approach worked well for non-multitexture systems without perspective-correct color, it is less useful in modern multitexture-capable hardware with perspective-correct color because each texture unit has its own q coordinate.

## Assigning Homogeneous Texture Coordinates

The rasterization discussion above assumed that homogeneous texture coordinates had been assigned per-vertex. As the application programmer, that is our job. This section describes how to set that up in OpenGL [2].

Consider that the texture is being projected onto the scene by a *slide projector*. This projector has most of the same properties that cameras have – it has a **viewing transform** that transforms world space coordinates into *projector space* (or eye space), and it has a **projection transform** that maps the *projector space* view volume to clip coordinates. Finally, we have a scale and bias to apply a simple range mapping. In the case of the camera, x and y are mapped based on the current viewport settings, and z is mapped based on the current depth range. For projective texture mapping, the range mapping is typically [0,1] for each coordinate. The NV_texture_rectangle texture target is an exception to this case because it is indexed by s ∈ [0, width] and t ∈ [0, height].

Figure 3 compares the transforms that are applied to vertex position in order to compute window space positional coordinates and projective texture coordinates.

The key to assigning texture coordinates for projective texture mapping is to use the

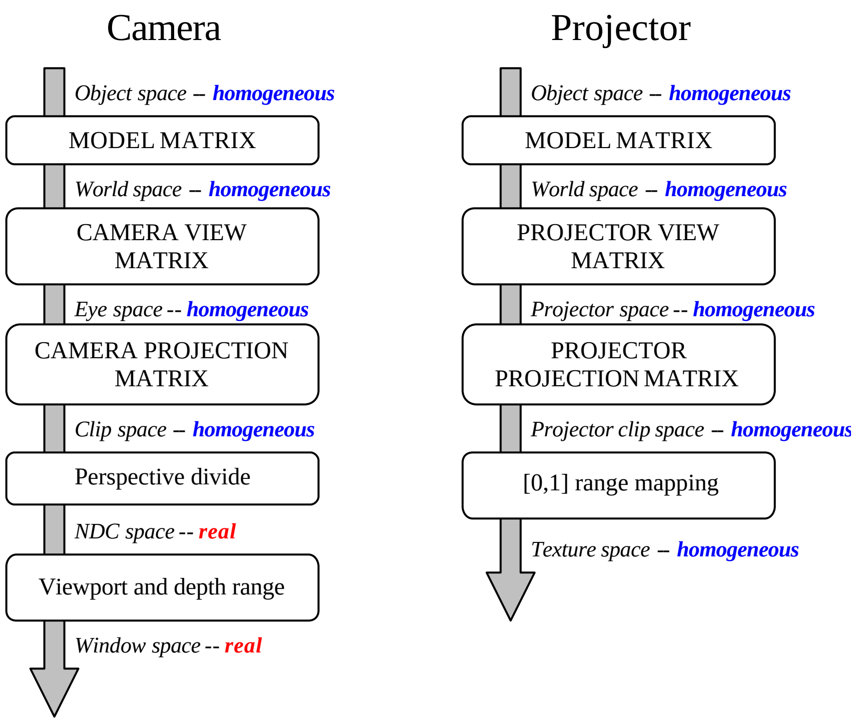


**Figure 3. The camera transforms applied to world space vertex position to generate window space coordinates are very similar to the projector transforms applied to world space vertex position to generate projective texture coordinates.**

OpenGL texture coordinate generation (texgen) facility. The texgen facility simply generates texture coordinates from other vertex attributes. In the case of GL_OBJECT_-LINEAR and GL_EYE_LINEAR texgen, the vertex position (in object space and eye space respectively) is used to generate the texture coordinate. Other forms of texgen use different attributes. GL_SPHEREMAP and GL_REFLECTION_MAP_ARB use the eye space vertex position and normal. GL_NORMAL_MAP_ARB simply assigns the eye space normal vector to the texture coordinates.

OpenGL keeps texgen state on a per-coordinate basis, which means, for example that you could assign the S texture coordinate with GL_SPHERE_MAP, the T coordinate with GL_REFLECTION_MAP_ARB, the R coordinate with GL_OBJECT_LINEAR, and the Q coordinate with GL_EYE_LINEAR. This flexibility is not particularly useful, though, and we typically use the same form of texgen for all coordinates.

The two types of texgen that are useful for projective texture mapping are *object linear* and *eye linear*. In both of these modes each component of the texture coordinate is computed by evaluating a plane equation at the vertex position. Because each coordinate has its own plane equation, and evaluating the plane equation is equivalent to a 4-component dot product, it may be more intuitive to think of the four texgen planes as forming a 4x4 matrix, **T**. Since texgen can be enabled per-coordinate, it would have been difficult to use this language in the OpenGL spec. Equations 1 and 2 illustrate how vertex positions are transformed into texture coordinates by the “texgen matrix”.

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \mathbf{T}_o \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix}_{object} \quad (1) \qquad \begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \mathbf{T}_e \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix}_{eye} \quad (2)$$

In the following two subsections, we will consider how to compute $\mathbf{T}_o$ (for *object linear* texgen) and $\mathbf{T}_e$ (for *eye linear* texgen). Also note that the texture coordinate generated by texgen is still transformed by the texture matrix. This can be attractive because it allows us to use standard matrix manipulation commands (glTranslate(), glRotate(), gluPerspective(), etc.), but it may be slightly less efficient than combining all transforms into the texgen matrix.

## Object Linear Texgen

*Object linear* texgen transforms the object space vertex position as specified to OpenGL. The concatenation of matrices required in this case is shown in Equation 3. In this equation, **M** is the ***model matrix*** (not to be confused with the OpenGL

$$\mathbf{T}_o = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}_p \mathbf{V}_p \mathbf{M} \qquad (3)$$

MODELVIEW matrix), $\mathbf{V}_p$ is the ***view matrix*** for the projector, and $\mathbf{P}_p$ is the ***projection matrix*** for the projector. The final matrix given performs the scale and bias to map the s, t, and r components of the texture coordinate to the [0,1] range.

Some considerations of using *object linear* texgen are that 1) you should keep track of your current ***model matrix*** – which is something OpenGL does not help you with, and 2) the object linear texgen planes, $\mathbf{T}_o$, must be updated every time the model matrix changes.

## Eye Linear Texgen

*Eye linear* texgen transforms the eye space vertex position, which is simply the object space vertex position transformed by the OpenGL MODELVIEW matrix. The concatenation of matrices required in this case is shown in Equation 4. The matrices in this equation are the same as in Equation 3, except that the ***model matrix***, M, has been replaced with the *inverse* of camera (or eye) ***view matrix***, $\mathbf{V}_e^{-1}$. This is logical, because we need to undo the eye's viewing transform to get back to world space.

$$\mathbf{T}_e = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}_p \mathbf{V}_p \mathbf{V}_e^{-1} \qquad (4)$$

Some considerations for using *eye linear* texgen are that 1) OpenGL transform and lighting **must** compute eye-space vertex position, and 2) a matrix inverse must be computed. OpenGL transform and lighting may already be computing eye space vertex position if we are using per-vertex lighting, so there may not be any significant per-vertex cost associated with *eye linear* texgen. OpenGL also tries to help out with the matrix inverse that is needed. It does this by transforming the eye planes (or rows of $\mathbf{T}_e$) by the inverse of the current MODELVIEW matrix. This is a common source of confusion when using *eye linear* texgen, so be aware. If you want to specify $\mathbf{T}_e$ directly, you should verify that the current MODELVIEW matrix is **identity**. To let OpenGL take care of the inverse view matrix, you can simply set the MODELVIEW matrix to contain *only* the ***view matrix***. Then, you would pass in the matrix from Equation 4 without including $\mathbf{V}_e^{-1}$.

## Caveats

There are a couple of issues to be aware of when using projective texture mapping. One is *reverse projection* and the other is texture image resolution and filtering.

### Reverse Projection

Unlike a real projector, the math of projective texture mapping actually produces a dual projection. One along the projector's view direction, and another in the opposite direction. Figure 4 illustrates this effect.

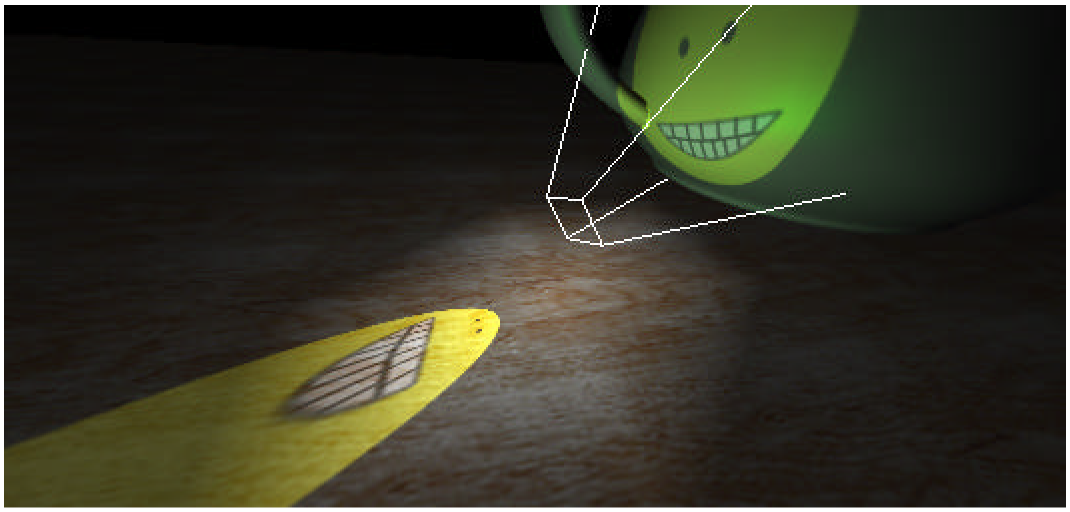


**Figure 4. Projective texture mapping produces a reverse projection as well.**

The sign of q becomes negative behind the projector, which inverts the texture image in the reverse projection. Typically applications use a 1D texture or other interpolated quantity to eliminate color contribution when the q coordinate of the projective texture becomes negative.

### Filtering Projective Textures

Another complication of projective texture mapping is the pathological variation in filtering requirements that can occur. Due to the nature of the two frusta involved, there can be portions of the texture image that require extremely anisotropic filter kernels, extreme minification, and extreme magnification of a single texture image in a single pass. Figure 5 illustrates this "deer in headlights" orientation of the two frusta.

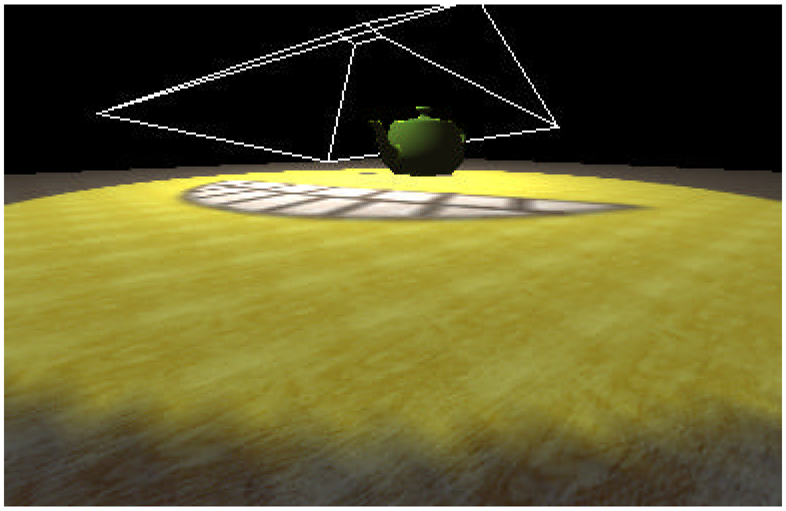


**Figure 5. The “deer in headlights” effect occurs when the projector’s frustum faces the eye’s frustum. This gross mismatch in sampling frequencies makes for complicated filtering.**

## Conclusion

Projective texture mapping has been supported in OpenGL since version 1.0. It is central to a number of interesting advanced rendering techniques, like per-pixel spotlight cone rendering and shadow mapping. The description provided here should provide enough background to enable OpenGL developers to implement projective texture mapping. Please visit http://www.nvidia.com/developer for more information, or send email to cass@nvidia.com.

## References

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

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

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

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