5.5

Advanced Texturing Using Texture Coordinate Generation

Ryan Woodland

Because today's graphics processors are pushing more and more polygons, attention is starting to turn to the use of bandwidth to create compelling texture effects. With the addition of multitexture abilities to many processors, people are starting to wonder how to creatively use these features. Of course, artist-applied texture is the technique with which we're all the most familiar, but it's quickly being discovered that mapping textures at run time can produce some very interesting results. Developers are starting to use texture coordinate generation to perform animation, lighting, reflection, refraction, and bump mapping, to name a few techniques. This article discusses a few of the most common texture coordinate generation techniques.

The method of texture coordinate generation used most comfortably by most people is that of transforming some data (position, normals, texture coordinates) by some matrix to yield a set of texture coordinates. This method is fairly easy to adopt because most 3D programmers are familiar with the concept of matrix transformation and because matrix transformation is often accelerated by hardware. This article describes only techniques that can be performed using matrix math.

Simple Texture Coordinate Animation

Quite often, games use a simple rotation or translation of texture coordinates to simulate simple effects such as reflection or to give the appearance of water or some moving material. The concept is simply this: A texture coordinate can be thought of as a simple 2D point. Because programmers are accustomed to transforming points by matrices, it is easy to see that a texture coordinate can be rotated, translated, or scaled simply by transforming it into a 3 × 3 matrix. Just as with geometry, a homogenous coordinate must be added to the *s*, *t* pair to make the transformation possible. Therefore, the method of coordinate generation looks like this:

*(s, t, 1) * 3x3mtx = s', t'*

The illustrations in Figure 5.5.1 were generated using simple rotation and scale. The first picture shows a textured quad with no transformation applied. The second shows a rotation of the texture coordinates by 45 degrees. The third shows a texture coordinate translation of 0.5.

Texture Projection

Texture projection is useful for a number of effects. Most often it is used to simulate lighting effects such as spotlights or shadows. The result of texture projection is fairly straightforward: A texture is projected onto some geometry from some point in space. For example, we can define a spotlight at some point in a scene and project a texture (probably a light circle) onto the geometry, creating the illusion of a spotlight.

Again, the concept of texture projection has its roots in normal 3D geometry techniques. When simulating a camera in 3D, a projection matrix is used to project vertices in camera space onto the near clipping plane of the camera. These points are mapped in the range of –1 to 1 in both *X* and *Y*, and then they are transformed into screen space by a viewport transformation, which usually involves a translation and scale.

For texture projection, instead of modeling a camera in space, we are usually modeling a light. Light space vertices are projected back onto the near clipping plane of the light, and the resulting *X* and *Y* values are used as *S* and *T* values to map a texture onto the projected geometry.

A light should be modeled just as a camera usually would. The near clipping plane should be set to reflect the dimensions of the texture that is to be projected. For instance, a square near clipping plane should be used for projecting a square texture.

Now, as mentioned, geometry that is projected onto a texture needs to be in light space, just as geometry to be projected onto the screen needs to be in camera space. In order to do this, we need to first transform the geometry into world space. Once this is done, the light matrix (just like the camera matrix) must then transform the geometry. The geometry can then be projected by the light's projection matrix.

Once geometry has been projected, another problem arises. Projected geometry, as mentioned previously, falls in the range of –1 to 1 in both *X* and *Y*, with (0, 0)

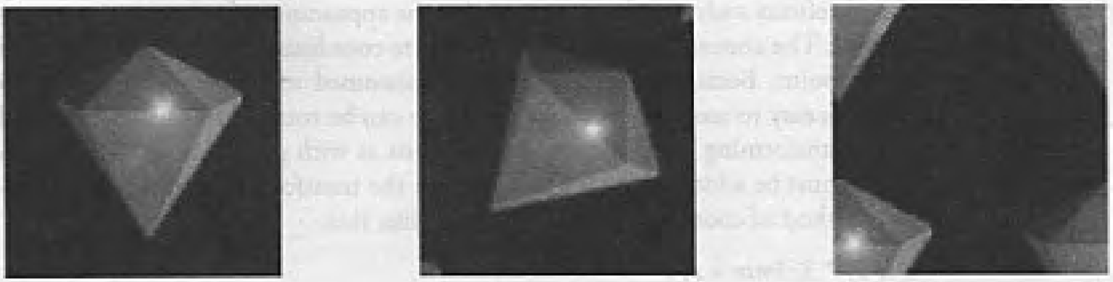
FIGURE 5.5.1. Examples of texture coordinate generation through texture matrix transformations.

being at the center of the plane of projection as it relates to the light. Texture coordinates usually run from 0 to 1 in both *S* and *T*, with the origin of that space being located at the upper-left corner of the texture. To map the projected coordinates into texture space, we must first scale them by 0.5 to put them in the range from –0.5 to 0.5 and then translate them by 0.5 to put them in the range from 0 to 1.

All these matrices can be concatenated together to form one final projection matrix for a given piece of geometry. The order is as follows:

*M_obj * M_light * M_proj * M_scale * M_trans * [x, y, 0, z] = [s, t, r, q]*

where:

- `M_obj` = the object's world space matrix.
- `M_light` = the light matrix used to transform the geometry from world space into light space.
- `M_proj` = the light's projection matrix.
- `M_scale` = 0.5 scale matrix.
- `M_trans` = 0.5 translation matrix.

The result of this calculation is a four-dimensional point. For simple texture projection, the *r* coordinate should be completely ignored, yielding an (*s, t, q*) triple. If the hardware allows, pass these three coordinates down for rasterization. The *q* coordinate is used to perform perspective correction; however, this must be done at rasterization time for it to be correct.

Figure 5.5.2 was generated using texture projection. It shows the frustum of the light that was used to project the circular highlight onto the sphere geometry.

By projecting geometry in this manner, a few unexpected results can occur. First, texture coordinates usually behave in a tiled manner. This means that there is really no

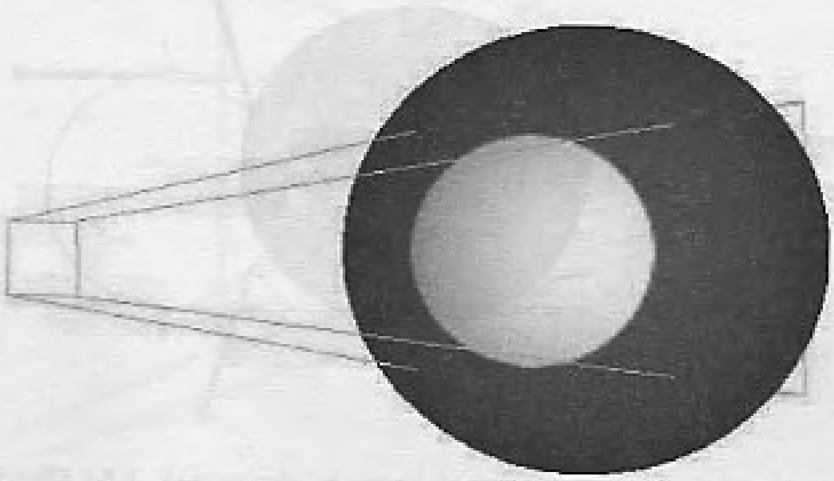

FIGURE 5.5.2. Texture projection example.

difference between a set of coordinates that ranges from 0 to 1 and a set that ranges from –1 to 0. Therefore, textures that are projected onto geometry should usually behave in a clamped manner, meaning that the outside border of the texture is repeated and applied to any texture coordinate less than 0 and greater than 1. For this reason, texture borders should be colored to behave correctly with the texture combine mode of choice.

The second and more complex problem is that of what I call *shine-through*. When we project a texture onto a sphere, for instance, the texture appears on both the front and back sides of the sphere as it relates to the light. This is because vertices on both the front and back of the sphere project into the correct texture space.

The image in Figure 5.5.3 highlights this problem. You can see that the spotlight texture projects correctly on the front of the sphere, but it also shines through to the geometry on the back of the sphere.

There are a couple of ways to fix this problem of shine-through. The first is to perform a dot product between the vertex normal and the light normal to determine whether the vertex is back facing. If the vertex is back facing, simply set the texture coordinate to something out of the range of 0 to 1.

Second, you can use the output of the standard lighting equation to determine whether a vertex is back facing. Place a parallel light at the location of the texture projector. If the color output from this parallel light for a vertex is black, you know that the vertex is back facing, because the only way for this vertex to become black is if the associated normal is facing away from the light.

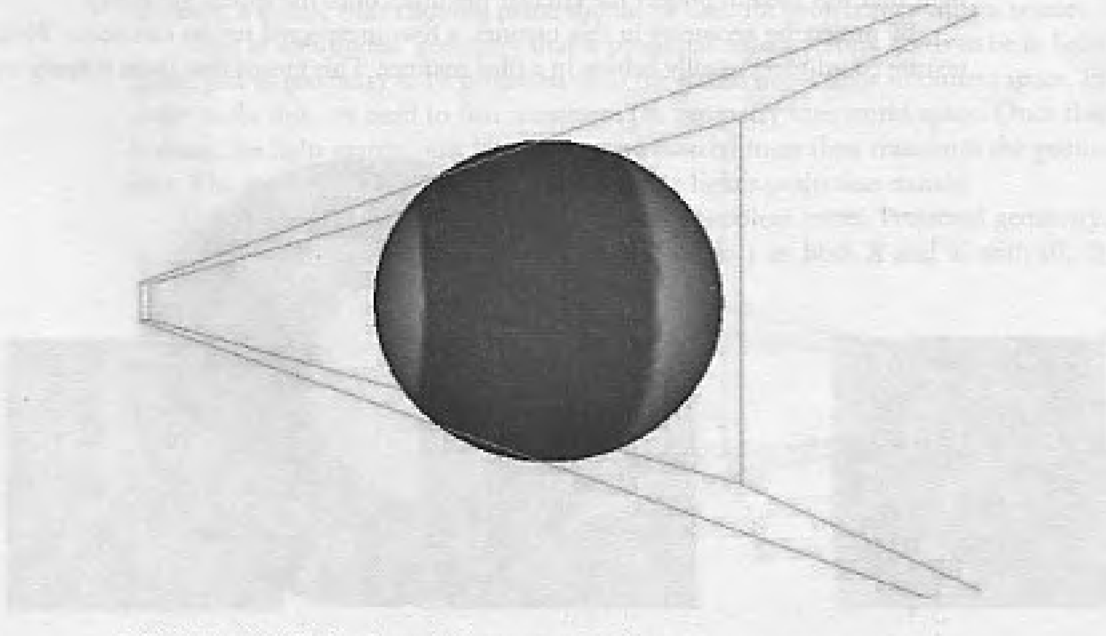

FIGURE 5.5.3. Shine-through in texture projection.

Reflection Mapping

To perform reflection mapping, I use a simple method called *sphere mapping*. The basic idea for this method makes two assumptions.

First, no matter the size and shape of the object being mapped, it is assumed to reflect the surrounding environment like a sphere. This concept is important because, logically, a point on a character's hand should reflect something different than a point on a character's foot with the same normal. With sphere mapping, these two points reflect exactly the same thing because they have the same normal.

Second, the reflective sphere like which the object will behave is assumed to be infinitely small. This means that all rays from the point of the eye in the scene to any point on the infinitely small sphere are parallel to each other.

Given these limitations, the sphere-mapping method operates on basic laws of reflection. Take, for instance, a ray from the eye point in a scene to a point on the reflective sphere. This ray should hit the sphere and reflect around the normal at the point of contact. Whatever this reflected ray hits should be seen reflected at the point of contact on the sphere. Figure 5.5.4 illustrates this concept.

Since it is not computationally feasible to perform one-bounce ray tracing for every point on a sphere, we instead create a texture map that contains the necessary environmental information. This map is called a *spherical reflection map*, or a *sphere map*.

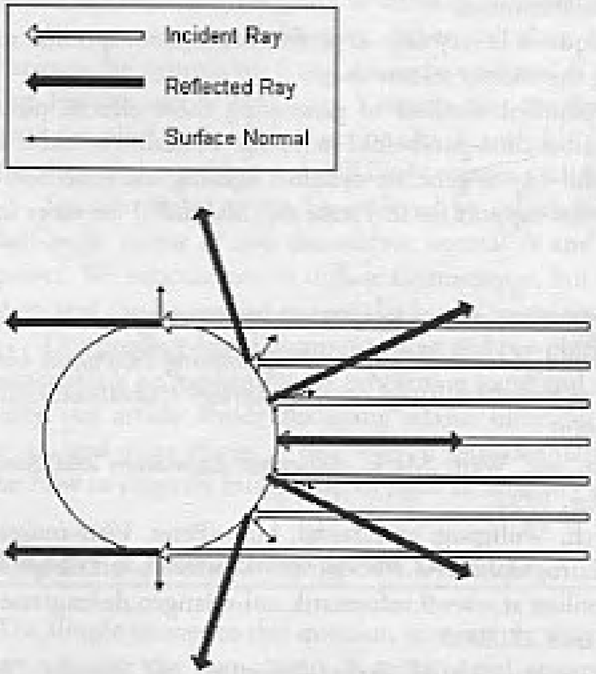

FIGURE 5.5.4. Rays are reflected around the surface normal at the point of contact on a reflective object.

The basic definition of a sphere map is a single texture map that contains a full 360-degree view of the environment surrounding a point in space. There is one big drawback to using sphere mapping for reflection: The texture used as the reflection map is viewpoint dependent. This means that to be completely correct, the texture map must be dynamically created each time the camera moves. I have found, however, that for some common effects such as generating a specular highlight on a car or creating lighting effects on a character, refusing to update the sphere map based on viewpoint is often not noticeable. (For an in-depth discussion of generating sphere maps see [Blythe99].)

Once you have an adequate sphere map, texture coordinate generation is a snap. Simply transform an object's normals into world space using the inverse transform of the object's model matrix. Then transform the normals into view space using the camera matrix. Finally, assuming your camera is looking down the $-Z$ axis, simply use the X and Y components of a normal as the S and T coordinates, respectively, for the associated vertex.

Obviously, using this technique, normals with a $-Z$ component generate the same S, T coordinate pair as the same normal with a $+Z$ component. This works out fine because any vertex with a $-Z$ component in its associated normal is by definition back facing and will not be seen, since this calculation is done in view space.

Color Plate 5 was produced simply by mapping a torus with a spherical reflection map of an outdoor environment.

Using this technique, it is very easy to perform reflection, specular mapping, and diffuse lighting using the correct texture maps.

For a view-independent method of generating these effects, please see [Heidrich98], which describes dual-paraboloid mapping. In addition, cubic environment mapping is a wonderful way to generate dynamic lighting and reflection effects if the target hardware provides support for it. Please see [Nvidia00] for more information.

References

[Blythe99] Blythe, David, *Advanced Graphics Programming Techniques Using OpenGL*, Available online at http://reality.sgi.com/blythe/sig99/advanced99/notes/node80.html, April 7, 2000.

[Watt92] Watt, Alan, and Watt, Mark, *Advanced Animation and Rendering Techniques*, ACM Press, 1992.

[Heidrich98] Heidrich, Wolfgang, and Seidel, Hans-Peter, *View-independent Environment Maps*, Eurographics/ACM Siggraph Workshop on Graphics Hardware 1998, available online at www9.informatik.uni-erlangen.de/eng/research/rendering/envmap/, March 22 2000.

[Nvidia00] NVIDIA technical brief, *Perfect Reflections and Specular Lighting Effects with Cube Environment Mapping*, available online at www.nvidia.com/Marketing/Developer/DevRel.nsf/WhitepapersFrame?OpenPage, March 10, 2000.