

Lecture on Shadows and Fog

1 Shadows

(Based on the explanation in Edward Angel's *Computer Graphics: A Top-Down Approach with OpenGL*.)

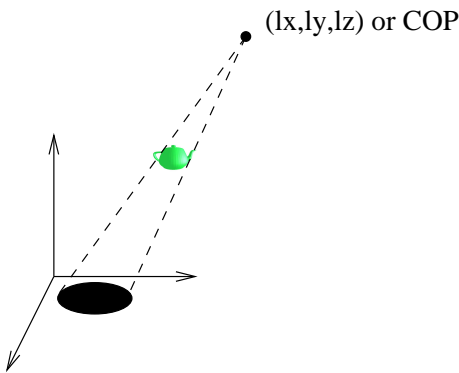
To do shadows generally is very hard. The renderer needs to know all the surfaces and light sources at once, so that it can trace rays and see where the shadows lie. As we have seen, this isn't possible because of the pipeline architecture.

We can do some special cases by hand without too much difficulty, however.

- Projecting only onto a flat plane, $y = 0$
- From one point light source only

This can be useful for outdoor scenes, say for the shadow of an airplane on the ground in a flight simulator.

1.1 The Basic Problem



This figure shows that the idea is to find the intersection of all rays from a point light source past the object onto a plane.

Our solution technique is to convert this problem into one that OpenGL can help us solve, namely a *projection* problem. In fact, there's really no conversion: a shadow is by definition the projection of a 3D object onto a 2D surface.

The conversion is simply to convert the light source to the COP. Note:

- the rays go the opposite direction (but the math works the same), and
- the object is in front of the projection plane, but again that doesn't matter.

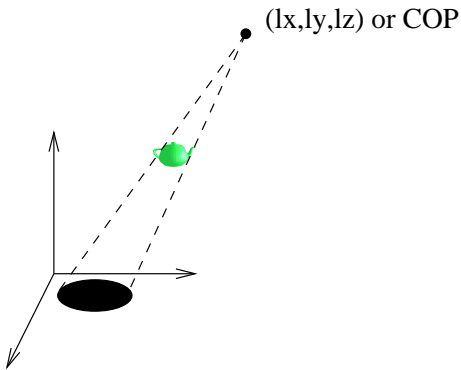
We're not going to make OpenGL do the projection for us (since that would get into issues like "near"); instead, we will use a custom projection matrix.

1.2 Projection

Let's translate things so that the COP is the origin. That means that the projection plane is the plane

$$y = -y_t$$

Consider the projection of an arbitrary point on the teapot (x, y, z) onto a projected point (x_p, y_p, z_p) . We can do this computation by similar triangles:



Thus:

$$\begin{aligned}\frac{x_p}{y_p} &= \frac{x}{y} \\ \frac{y_p}{y_p} &= \frac{y}{y} \\ \frac{z_p}{y_p} &= \frac{z}{y}\end{aligned}$$

(Yes, the middle equation reduces to $y_p = y_p$, but we do it for a symmetry that we'll find useful very soon.)

As we did once before, these rearrange to:

$$\begin{aligned}x_p &= \frac{x}{y/y_p} \\ y_p &= \frac{y}{y/y_p} \\ z_p &= \frac{z}{y/y_p}\end{aligned}$$

Thus, we have:

$$\begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix} = \begin{bmatrix} \frac{x}{y/y_p} \\ y_p \\ \frac{z}{y/y_p} \end{bmatrix}$$

Or, using homogeneous coordinates:

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ y/y_p \end{bmatrix}$$

We can accomplish that mapping with the following *projection matrix*:

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1/y_p & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

So, in summary, our steps are:

- move the origin to where we want the COP to be, namely, the location of the light source
- project
- move the origin back.
- draw the object

We move the origin back so that the object's shadow is drawn on the plane $y = 0$ and not on the plane $y = -y_l$.

Demo: The code is in `~cs307/pub/demos/16-shadowTeapot.cc`.

Note:

- The sequence of steps in the shadow rendering
- How the projection matrix is computed
- That we turn off lighting when we draw the shadow

The fact that we translate, multiply by a projection matrix, and then un-translate may seem odd. In what way are we projecting? We're not projecting at the moment we multiply by the matrix, of course; we are merely modifying the CTM. Thus, each vertex of the teapot is rendered:

$$V_{\text{rendered}} = M_{\text{twCamera}} M_{T(1)} M_{\text{Projection}} M_{T(-1)} V_{\text{original}}$$

When the vertices go through the projection matrix, their y coordinate gets squished down to $-y_l$, and the translation afterwards makes the y coordinate equal to zero.

This really should feel like the Road Runner painting a hole on the wall. We're drawing the teapot in a 50 percent gray color and getting its shadow. Very cool.

1.3 More Generality

While this is cool, it certainly seems very restrictive. However, it's not too bad:

- we could use blending and generalize it to more light sources
- we could use some transformations to move the shadow plane to another height (though we'd need to be careful about this)

2 Fog

(Based on the explanation in the OpenGL Programming Guide, first edition.)

Fog is a blending process, where the amount of blending depends on the distance of the fragment from the eye location. It can be used for haze, smoke, fog, and so forth. This is often useful in outdoor scenes where distant objects can just fade into the fog, instead of being rendered in sharp detail.

The first important step is to define your fog color. Ordinary fog might be gray. A hazy day might have a light blue tinge to it. Smoke might be black. The fog color is defined using RGBA as follows:

```
GLfloat fogColor = {0.5, 0.5, 0.5, 1.0};
glFogfv(GL_FOG_COLOR, fogColor);
```

You also have to enable the fog calculations, as follows:

```
glEnable(GL_FOG);
```

The fog process calculates a fraction, f , that depends on the distance of the fragment. The fraction is used to blend the incoming fragment color, C_i with the fog color, C_f , thus:

$$C = fC_i + (1 - f)C_f$$

So the f value needs to be *high* for things that show clearly through the fog and *low* for things that are hidden in the fog. This means that the fog function needs to start out near 1 and decrease to zero as the distance from the eye increases.

The f value is clamped by OpenGL to the range [0,1].

2.1 Fog Functions

There are many kinds of functions that monotonically decrease (actually, non-increase), as distance, z , increases. The OpenGL people chose three:

2.1.1 Linear

The linear function is mathematically defined as

$$f = \frac{\text{end} - z}{\text{end} - \text{start}}$$

Where “end” and “start” are the z distances where the fog starts and ends.

You can look at examples of this function by using `gnuplot` on the Unix machines. Here’s are some examples.

```
% gnuplot
gnuplot> plot [z=0:5] (5-z)/(5-0)
gnuplot> plot [z=0:5] (4-z)/(4-1)
```

This is specified in OpenGL as follows:

```
glFogi(GL_FOG_MODE, GL_LINEAR);
glFogf(GL_FOG_START, 1.5);
glFogf(GL_FOG_END, 5.0);
```

This function is nice because you can delay the fog setting in for a while (so that foreground stuff is entirely clear). You can also easily know when the fog will entirely obliterate objects.

2.2 Exponential

The exponential decay function is mathematically defined as

$$f = e^{-cz}$$

where c is a density constant.

You can use `gnuplot` to look at some examples:

```
gnuplot> plot [z=0:5] exp(-z)
gnuplot> plot [z=0:5] exp(-2*z)
gnuplot> plot [z=0:5] exp(-0.5*z)
gnuplot> plot [z=0:5] exp(-0.25*z)
gnuplot> plot [z=0:5] exp(-0.25*z), exp(-0.5*z)
```

This function is nice because it trails off at a decreasing rate, never really reaching zero, so it has a nice realism in that way. However, unless you use a pretty small density,

it drops very quickly, so I would suggest you look at a few functions first before you start guessing the density.

The exponential decay function is specified in OpenGL as follows:

```
glFogf(GL_FOG_MODE, GL_EXP);  
glFogf(GL_FOG_DENSITY, 0.25);
```

2.2.1 Gaussian Decay

Many of us have an intuitive picture of the “normal” or “bell” curve: starting at a peak in the middle (usually at zero), it drops off slowly at first, then more quickly, then trails off towards zero very slowly. The function, often called the “Gaussian” function, is mathematically defined thus:

$$f = e^{-cz^2}$$

In short, this is just the exponential decay using the square of the distance. The function can be plotted with gnuplot:

```
gnuplot> plot [z=0:5] exp(-z*z)  
gnuplot> plot [z=0:5] exp(-2*z*z)  
gnuplot> plot [z=0:5] exp(-0.5*z*z)  
gnuplot> plot [z=0:5] exp(-0.25*z*z)  
gnuplot> plot [z=0:5] exp(-0.25*z*z), exp(-0.5*z*z)
```

The Gaussian decay function is specified in OpenGL as follows:

```
glFogi(GL_FOG_MODE, GL_EXP2);  
glFogf(GL_FOG_DENSITY, 0.25);
```

2.3 Hints

In computer graphics, we can often define a tradeoff between speed and quality. It would be nice to be able to ask OpenGL which we prefer, and, indeed there is, in the form of *hints*. There are hints for many features (look at the manual page for `glHint`), but we will just look at one:

- If we prefer quality, we can ask that the fog calculation be done on a pixel-by-pixel basis (each pixel having a different depth or z value), as follows:

```
glHint(GL_FOG_HINT, GL_NICEST);
```

- If we prefer speed, we can ask that the fog calculation be done on a vertex basis, allowing the usual linear interpolation to handle all the intermediate pixels, as follows:

```
glHint(GL_FOG_HINT, GL_FASTEST);
```

- If we don't care, believe it or not, we can specify:

```
glHint(GL_FOG_HINT, GL_DONT_CARE);
```

Another hint is to clear the window to your fog color, so that things will be fading into the background.

Demos: `~cs307/pub/demos/16-FogKinds.cc`, `~cs307/pub/demos/16-FogOutdoors.cc`, and `~cs307/pub/demos/16-FogInside.cc`.