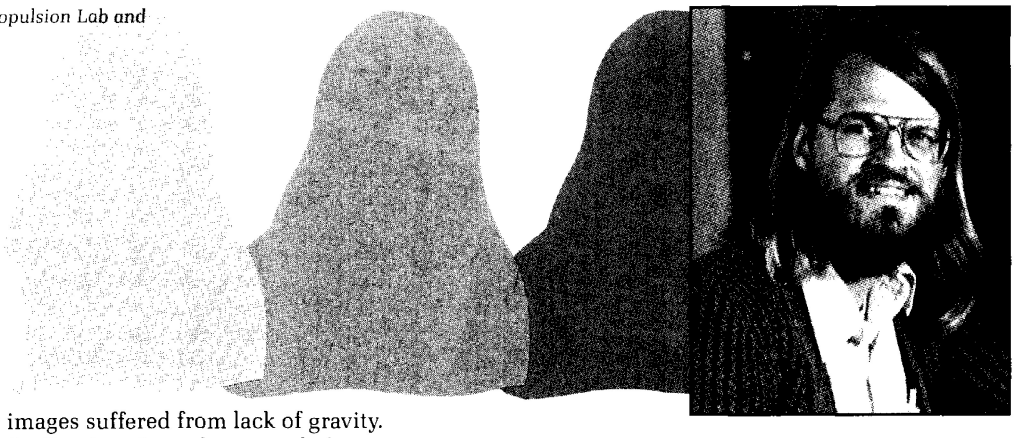
Jim Blinn's Corner

Me and My (Fake) Shadow

James F. Blinn, Jet Propulsion Lab and Caltech



Early computer images suffered from lack of gravity. Objects seemed to be floating above the ground plane. One way to solve this is to make the objects cast shadows. The shadows seem to "tie the object down." They also help to separate the object from the background. A general shadow-casting algorithm is not trivial, but it is often not necessary for the shadows to be perfect to give the right visual effect. Approximations to shadows are quite adequate, as conventional animators have known for years, and it is possible to generate approximate shadows easily by some mathematical tricks.

One of the most common tricks is to cast shadows on a flat ground plane by redrawing the objects of the scene scaled by zero in the vertical direction. I will present here two ways to do this: one that works for parallel light rays, and another that works for localized light sources. The latter illustrates some subtleties in the use of homogeneous coordinates and clipping.

I will use the notation developed in the October issue of *CG&A* for representing the scene. A subassembly called GROUND will define the ground plane, which lies at z=0. It is just a grid in the examples below. A subassembly called STUFF contains all the objects sitting on the ground, just a cube in the examples here. A scene without shadows would then be defined as

```
DEF  SCENE
DRAW GROUND     !ground plane at z=0
DRAW STUFF      !collection of objects
----
```

Simple shadows

If illumination comes from a single point source infinitely far away, all the light rays will be parallel to this direction, which we will call (x_L,y_L,z_L). A point (x_P,y_P,z_P) on some object of the scene will cast a shadow on the ground plane at $(x_S,y_S,0)$. The cast shadow starts at **P** and moves away from the light source in the opposite direction, toward **L** by some amount, until it hits z=0.

$$\mathbf{S} = \mathbf{P} - \alpha\mathbf{L}$$

Solve for α by the requirement that $z_S=0$. This results in

$$0 = z_P - \alpha z_L$$

or

$$\alpha = \frac{z_P}{z_L}$$

The x and y coordinates of the shadow point are then

$$x_S = x_P - \frac{z_P}{z_L}x_L$$

$$y_S = y_P - \frac{z_P}{z_L}y_L$$

While this is all very nice for computing individual points, it becomes exciting when written as a matrix multiplication:

$$\begin{bmatrix} x_S & y_S & 0 & 1 \end{bmatrix} = \begin{bmatrix} x_P & y_P & z_P & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -x_L/z_L & -y_L/z_L & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now shadow generation can be performed by the standard transformation and viewing process. Simply

0272-1716/88/0100-0082$01.00 1988 IEEE

draw the scene once normally, and then draw it again transformed by the above matrix.

```
DEF   SCENE
DRAW  GROUND
DRAW  STUFF
PUSH
ORIE  1, 0, -XL/ZL, 0, 1, -YL/ZL, 0, 0, 0,
DRAW  STUFF
POP
----
```

An example of this appears in Figure 1. The light comes from the direction (1.45,−.8,2). The cube is centered at the origin in x and y and stretches from 0 to 2 in z. Point **P** is at (1,1,2).

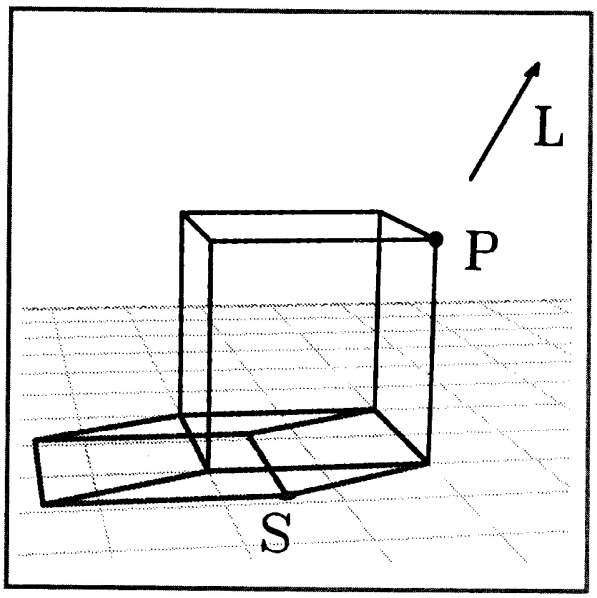


Figure 1. Light source at infinity.

Perspective shadows

Now what about local light sources? Here the light position (x_L,y_L,z_L) is an actual location in space, not a vector. The shadow is then a perspective projection of the object from the point of view of the light source.

The shadow of a point **P** is cast in the direction **P**−**L** so that

$$\mathbf{S} = \mathbf{P} - \alpha(\mathbf{P} - \mathbf{L})$$

Again, putting in z=0 and solving for α gives

$$\alpha = \frac{-z_P}{z_P - z_L}$$

The shadow coordinates turn out to be

$$x_S = \frac{x_L z_P - x_P z_L}{z_P - z_L}$$
$$y_S = \frac{y_L z_P - y_P z_L}{z_P - z_L}$$

Believe it or not, this can also be done with a matrix multiplication. The division can be accomplished by using coordinates and taking advantage of the implied division by the fourth coordinate. First, define

$$x_S = \widetilde{x_S}/\widetilde{w_S}$$
$$y_S = \widetilde{y_S}/\widetilde{w_S}$$

Then the shadow of point **P** is found by the matrix multiplication

$$\begin{bmatrix} \widetilde{x_S} & \widetilde{y_S} & 0 & \widetilde{w_S} \end{bmatrix} = \begin{bmatrix} x_P & y_P & z_P & 1 \end{bmatrix} \begin{bmatrix} -z_L & 0 & 0 & 0 \\ 0 & -z_L & 0 & 0 \\ x_L & y_L & 0 & 1 \\ 0 & 0 & 0 & -z_L \end{bmatrix}$$

Now there is a bit of a nuts-and-bolts problem. We know each element of the desired 4×4 matrix. The problem is expressing it to our transformation system. There is no command for explicitly specifying all 16 elements of a matrix. The most straightforward solution might be to implement a new command, similar to **ORIE**, which allows 16 parameters. However, I regarded it as an intellectual challenge to come up with a series of **PERS, TRAN, ROT,** and **SCAL** transformations that, when multiplied together, yield exactly the above matrix.

How to do this? First, reexamine the **PERS** transformation. (I have, in fact, received some questions concerning an ambiguity in its description in the October column.) By way of review and clarification, the perspective command

PERS θ, z_n, z_f

performs a perspective projection from the origin with a pyramid of view having an apex angle θ. The associated matrix is constructed by precalculating

$$s = \sin(\theta/2)$$
$$c = \cos(\theta/2)$$
$$Q = \frac{s}{1 - z_n/z_f}$$

The matrix is then

$$\begin{bmatrix} c & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & Q & s \\ 0 & 0 & -Qz_n & 0 \end{bmatrix}$$

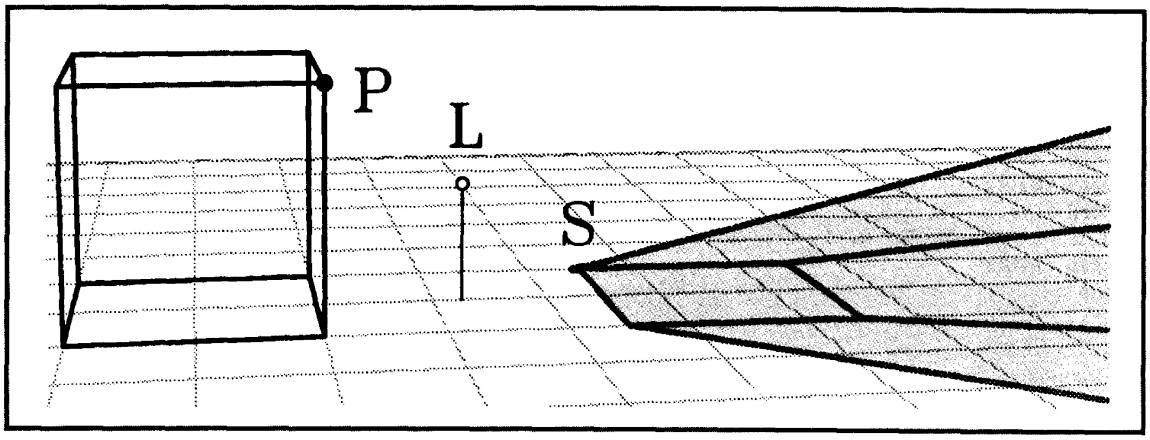


Figure 2. Antishadow.

The transformation effectively generates a screen x and y by dividing by z and then scaling by the cotangent of half the field of view (i.e., the angle subtended from the center of the screen to the edge). The z coordinate is also transformed in a way which won't concern us here. We will therefore simplify matters by setting $z_n = 1$ and $z_f = 1{,}000{,}000$, effectively making $Q = s$.

The basic strategy for using this transform to construct the shadow transform is

1. Translate the light source to the origin.
2. Apply a PERS transform by some angle.
3. Translate back to the original light source location.
4. Scale by 0 in z to squash the shadow down onto the ground plane.

To find the angle, I multiplied the above transformations together (symbolically) and solved for a θ that causes the net matrix to be equal to the desired one. The result is

$$\cot(\theta/2) = -z_L$$

This may seem a bit strange, since the field of view is negative, but it *does* generate the proper matrix. In fact, the trigonometry can be avoided by noting that you get the same effect using a perspective field of view of 90° and then scaling the result in x and y by the above cotangent, $-z_L$. The final transformed scene is then

```
DEF   SCENE
DRAW  GROUND
DRAW  STUFF
PUSH
SCAL  1, 1, 0,
TRAN   XL,  YL,  ZL,
SCAL  -ZL, -ZL, 1
PERS   90, 1, 1000000
TRAN  -XL, -YL, -ZL
DRAW  STUFF
POP
----
```

So, I coded it up, put it into the machine, and presto...no shadows. What went wrong?

The problem has to do with a subtlety in the clipping algorithm used with homogeneous perspective transformations. This problem is described in more detail in an old paper by Martin Newell and myself, (J.F. Blinn and M.E. Newell, "Clipping Using Homogeneous Coordinates," *Computer Graphics* [Proc. SIGGRAPH 78], Aug. 1978, p. 245). It concerns the situation when the *w* coordinate of a homogeneous point is negative. Notice that this will be the case if the light source is higher than the object ($z_L > z_P$), the expected situation.

Normally the clipping condition is defined so that a point is "visible" if its location after the homogeneous division lies within the standard screen boundaries, i.e.,

$$-1 < \frac{x}{w} < 1$$

Since clipping is done before division, this is rewritten to say that a point is visible if

$$0 < w - x$$
$$\text{and } 0 < w + x$$

This is not quite correct, however. It's only right if $w > 0$. If w is negative, a point might still project onto the visible region of the screen but be declared invisible by the above criterion. This ordinarily doesn't matter, because in the normal usage of the perspective transform, negative w's come only from perspective projections of points that were behind the viewer, and we want to clip them off anyway. Here, however, we have negative w's we want to keep.

The matrix we are using generates a sort of "antishadow." If you move the light source below the object, a projected image appears on the ground plane that is the object projected through the light point. This is shown in Figure 2. The local light coordinates are (2.25,0,1), and the light is sitting on a "lamppost" just to make it easier to see where it is situated in 3D. Notice that the example point $\mathbf{P} = (1,1,2)$ projects on a straight line through the light source to the point $\mathbf{S} = (3.5, -1, 0)$.

The solution to the antishadow problem is to avoid the negative w's while retaining the same geometrical projection. This is done by multiplying the entire fake perspective matrix by −1. Usually, any nonzero scalar multiple of a homogeneous quantity represents exactly the same quantity. However, due to the asymmetry in clipping, this is not the case here. The correct shadow transform matrix is

$$\begin{bmatrix} z_L & 0 & 0 & 0 \\ 0 & z_L & 0 & 0 \\ -x_L & -y_L & 0 & -1 \\ 0 & 0 & 0 & z_L \end{bmatrix}$$

Now we again have the intellectual challenge of constructing this matrix using only our defined operations. The sequence is

1. Translate light source to origin.
2. Turn around to look at the scene (rotate 180° in y).
3. Get perspective projection by positive angle, $2 \cot^{-1} z_L$.
4. Reverse the y rotation.
5. Reverse the translation.
6. Scale by 0 in z.

Translating it into our command language gives us

```
DEF  SCENE
DRAW GROUND
DRAW STUFF
PUSH
SCAL    1, 1, 0,
TRAN   XL,  YL,  ZL,
ROT   180, 2,
SCAL   ZL,  ZL,   1,
PERS   90, 1, 100000,
ROT   180, 2,
TRAN  -XL, -YL, -ZL
DRAW  STUFF
POP
----
```

If you multiply all these matrices together in the indicated order, you find that you do indeed get the desired matrix with all signs flipped properly. The result is shown in Figure 3. The light is at (2.25,.5,5).

Now I admit that I wasn't nearly that organized the first time I tried this. At first, I tried various hack-and-bash mechanisms for mixing available matrices to get the desired result. The solution I came up with is especially weird, since the near clipping plane is set farther away than the far clipping plane. Here is that first attempt, which does indeed generate the identical desired matrix, but in a far less intuitive way.

```
DEF   SCENE
DRAW  GROUND
DRAW  STUFF
PUSH
SCAL  1, 1, 0,
PERS  90, 1000000, 1,
TRAN  0, 0, ZL,
ORIE  ZL, 0, -XL, 0, ZL, -YL, 0, 0, -1,
DRAW  STUFF
POP
----
```

Notice that, in a complete scene, we are using the perspective transform twice: once as part of the viewing transformation, and a second time embedded in the model, to distort STUFF into the properly shaped shadow. When I first came up with this, I was very proud of myself. A while later I happened to be talking to Martin Newell on the phone and described it to him. He said, "Oh yes, that's how we used to make shadows in the late sixties in the CAD lab in England." Oh, well. I guess it just goes to show you that—

Nobody ever does anything *first*.

General shadows

It is interesting at this point to compare the perspective shadow matrix with a variant of the infinite-light-source shadow matrix. The latter can be rewritten as

$$\begin{bmatrix} z_L & 0 & 0 & 0 \\ 0 & z_L & 0 & 0 \\ -x_L & -y_L & 0 & 0 \\ 0 & 0 & 0 & z_L \end{bmatrix}$$

In other words, we can come up with a general-purpose matrix that works for both cases:

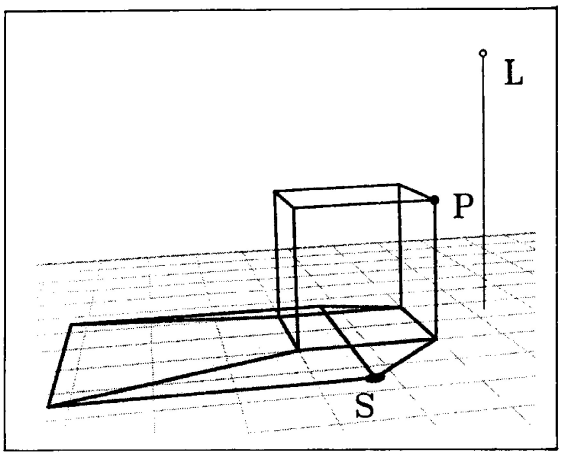


Figure 3. Correct perspective shadow.

$$\begin{bmatrix} z_L & 0 & 0 & 0 \\ 0 & z_L & 0 & 0 \\ -x_L & -y_L & 0 & -w_L \\ 0 & 0 & 0 & z_L \end{bmatrix}$$

where $w_L = 0$ for infinite light sources and $w_L = 1$ for local light sources.

How might we have found this to begin with? First, let's treat the relevant points **P, L,** and **S** as four-element, homogeneous row vectors. An arbitrary point on the line connecting **P** and **L** can be represented in homogeneous coordinates as

$$\alpha\mathbf{P} + \beta\mathbf{L}$$

where the pair (α,β) forms a sort of 1D homogeneous coordinate for points on the line.

Now represent the desired ground plane **G** as a four-element column vector. In our case, for z = 0, we have been using

$$\mathbf{G} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

but it can be any plane in general.

The point on the shadow line that intersects the ground plane is given by the (α,β) that satisfy

$$(\alpha\mathbf{P} + \beta\mathbf{L}) \cdot \mathbf{G} = 0$$

or

$$\alpha(\mathbf{P} \cdot \mathbf{G}) + \beta(\mathbf{L} \cdot \mathbf{G}) = 0$$

This can be satisfied by the pair of values

$$\alpha = (\mathbf{L} \cdot \mathbf{G})$$
$$\beta = -(\mathbf{P} \cdot \mathbf{G})$$

or any scalar multiple of it. The shadow point, **S**, then is

$$\mathbf{S} = \mathbf{P}(\mathbf{L} \cdot \mathbf{G}) - (\mathbf{P} \cdot \mathbf{G})\mathbf{L}$$

Now we must write this as a matrix multiplication. Behold the following trick: The expression

$$(\mathbf{P} \cdot \mathbf{G})\mathbf{L}$$

is a row vector times a column vector times a row vector. Matrix multiplication is associative, so it can be written

$$\mathbf{P}(\mathbf{GL})$$

Now realize that, while $\mathbf{L} \cdot \mathbf{G}$ is a number, the quantity **GL** is a 4×4 matrix according to the rules of vector product conformability. This is sometimes called the "outer product" of the vectors in contrast to the ordinary "inner product." The construction of **S** can then be written

$$\mathbf{S} = \mathbf{P}[(\mathbf{L} \cdot \mathbf{G})\mathbf{I} - \mathbf{GL}]$$

where **I** is a 4×4 identity matrix. The perspective matrix itself is

$$[(\mathbf{L} \cdot \mathbf{G})\mathbf{I} - \mathbf{GL}]$$

If you plug in the z = 0 plane for **G** you will get the matrix we derived above. In general, however, you can use this to perspectively project on any plane.

Correspondence about previous columns

I have received a letter from Nelson Max concerning the October column about Blobby Man. He points out that the rotation trick for making the foot always point forward does not keep it *exactly* forward (with an x component of 0). It still has some small sideways component. This is, of course, quite true. The intention was just to keep it approximately pointing forward (with a negative y component). This works best for the expected range of values $-90 < \mathbf{LOUT} < 0$ and $-90 < \mathbf{LHIP} < 90$. All other rotation combinations I tried made it too easy to get the foot pointing completely backward; amusing perhaps, but a real nuisance for animation.

In addition, several people have been sending me new circle-drawing algorithms. When I have had time to digest them, I will present any which I find are noticeably different from those in the original circle article, so keep those algorithms coming in. ■