Chapter 61

Frames of Reference

Chapter 61

The Fundamentals of the Math behind 3-D Graphics

Several years ago, I opened a column in *Dr. Dobb's Journal* with a story about singing my daughter to sleep with Beatles' songs. Beatles' songs, at least the earlier ones, tend to be bouncy and pleasant, which makes them suitable goodnight fodder—and there are a *lot* of them, a useful hedge against terminal boredom. So for many good reasons, "Can't Buy Me Love" and "A Hard Day's Night" and "Help!" and the rest were evening staples for years.

No longer, though. You see, I got my wife some Beatles tapes for Christmas, and we've all been listening to them in the car, and now that my daughter has heard the real thing, she can barely stand to be in the same room, much less fall asleep, when I sing those songs.

What's noteworthy is that the only variable involved in this change was my daughter's frame of reference. My singing hasn't gotten any worse over the last four years. (I'm not sure it's *possible* for my singing to get worse.) All that changed was my daughter's frame of reference for those songs. The rest of the universe stayed the same; the change was in her mind, lock, stock, and barrel.

Often, the key to solving a problem, or to working on a problem efficiently, is having a proper frame of reference. The model you have of a problem you're tackling often determines how deeply you can understand the problem, and how flexible and innovative you'll be able to be in solving it.

An excellent example of this, and one that I'll discuss toward the end of this chapter, is that of *3-D transformation*—the process of converting coordinates from one coordinate space to another, for example from worldspace to viewspace. The way this is traditionally explained is functional, but not particularly intuitive, and fairly hard to visualize. Recently, I've come across another way of looking at transforms that seems to me to be far easier to grasp. The two approaches are technically equivalent, so the difference is purely a matter of how we choose to view things—but sometimes that's the most important sort of difference.

Before we can talk about transforming between coordinate spaces, however, we need two building blocks: dot products and cross products.

3-D Math

At this point in the book, I was originally going to present a BSP-based renderer, to complement the BSP compiler I presented in the previous chapter. What changed my plans was the considerable amount of mail about 3-D math that I've gotten in recent months. In every case, the writer has bemoaned his/her lack of expertise with 3-D math, and has asked what books about 3-D math I'd recommend, and how else he/she could learn more.

That's a commendable attitude, but the truth is, there's not all that much to 3-D math, at least not when it comes to the sort of polygon-based, realtime 3-D that's done on PCs. You really need only two basic math tools beyond simple arithmetic: dot products and cross products, and really mostly just the former. My friend Chris Hecker points out that this is an oversimplification; he notes that lots more math-related stuff, like BSP trees, graphs, discrete math for edge stepping, and affine and perspective texture mappings, goes into a production-quality game. While that's surely true, dot and cross products, together with matrix math and perspective projection, constitute the bulk of what most people are asking about when they inquire about "3-D math," and, as we'll see, are key tools for a lot of useful 3-D operations.

The other thing the mail made clear was that there are a lot of people out there who don't understand either type of product, at least insofar as they apply to 3-D. Since much or even most advanced 3-D graphics machinery relies to a greater or lesser extent on dot products and cross products (even the line intersection formula I discussed in the last chapter is actually a quotient of dot products), I'm going to spend this chapter examining these basic tools and some of their 3-D applications. If this is old hat to you, my apologies, and I'll return to BSP-based rendering in the next chapter.

Foundation Definitions

The dot and cross products themselves are straightforward and require almost no context to understand, but I need to define some terms I'll use when describing applications of the products, so I'll do that now, and then get started with dot products.

I'm going to have to assume you have *some* math background, or we'll never get to the good stuff. So, I'm just going to quickly define a *vector* as a direction and a magnitude, represented as a coordinate pair (in 2-D) or triplet (in 3-D), relative to the origin. That's a pretty sloppy definition, but it'll do for our purposes; if you want the Real McCoy, I suggest you check out *Calculus and Analytic Geometry*, by Thomas and Finney (Addison-Wesley: ISBN 0-201-52929-7).

So, for example, in 3-D, the vector **V** = [5 0 5] has a length, or magnitude, by the Pythagorean theorem, of

$$\|\mathbf{V}\| = \sqrt{v_1^2 + v_2^2 + v_3^2} = \sqrt{5^2 + 0^2 + 5^2} = 5\sqrt{2} \qquad \text{(eq. 1)}$$

(where vertical double bars denote vector length), and a direction in the plane of the x and z axes, exactly halfway between those two axes.

I'll be working in a left-handed coordinate system, whereby if you wrap the fingers of your left hand around the z axis with your thumb pointing in the positive z direction, your fingers will curl from the positive x axis to the positive y axis. The positive x axis runs left to right across the screen, the positive y axis runs bottom to top across the screen, and the positive z axis runs into the screen.

For our purposes, *projection* is the process of mapping coordinates onto a line or surface. *Perspective projection* projects 3-D coordinates onto a viewplane, scaling coordinates according to their z distance from the viewpoint in order to provide proper perspective. *Objectspace* is the coordinate space in which an object is defined, independent of other objects and the world itself. *Worldspace* is the absolute frame of reference for a 3-D world; all objects' locations and orientations are with respect to worldspace, and this is the frame of reference around which the viewpoint and view direction move. *Viewspace* is worldspace as seen from the viewpoint, looking in the view direction. *Screenspace* is viewspace after perspective projection and scaling to the screen.

Finally, *transformation* is the process of converting points from one coordinate space into another; in our case, that'll mean rotating and translating (moving) points from objectspace or worldspace to viewspace.

For additional information, you might want to check out Foley & van Dam's *Computer Graphics* (ISBN 0-201-12110-7), or the chapters in this book dealing with my X-Sharp 3-D graphics library.

The Dot Product

Now we're ready to move on to the dot product. Given two vectors **U** = [u_1 u_2 u_3] and **V** = [v_1 v_2 v_3], their dot product, denoted by the symbol •, is calculated as:

$$\mathbf{U} \bullet \mathbf{V} = u_1 v_1 + u_2 v_2 + u_3 v_3 \qquad \text{(eq. 2)}$$

As you can see, the result is a scalar value (a single real-valued number), *not* another vector.

Now that we know how to calculate a dot product, what does that get us? Not much. The dot product isn't of much use for graphics until you start thinking of it this way

$$\mathbf{U} \bullet \mathbf{V} = \cos(\theta) \|\mathbf{U}\| \|\mathbf{V}\| \qquad \text{(eq. 3)}$$

where q is the angle between the two vectors, and the other two terms are the lengths of the vectors, as shown in Figure 61.1. Although it's not immediately obvious, equation 3 has a wide variety of applications in 3-D graphics.

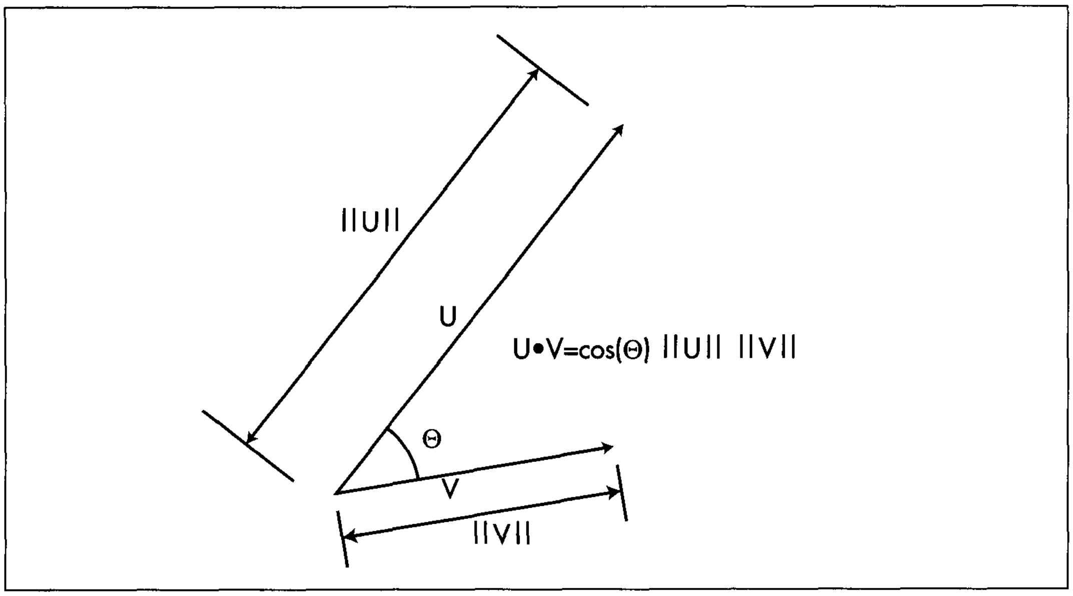
Dot Products of Unit Vectors

The simplest case of the dot product is when both vectors are *unit vectors*; that is, when their lengths are both one, as calculated as in Equation 1. In this case, equation 3 simplifies to:

$$\mathbf{U} \bullet \mathbf{V} = \cos(\theta) \qquad \text{(eq. 4)}$$

In other words, the dot product of two unit vectors is the cosine of the angle between them.

One obvious use of this is to find angles between unit vectors, in conjunction with an inverse cosine function or lookup table. A more useful application in 3-D graphics



The dot product.
Figure 61.1

lies in lighting surfaces, where the cosine of the angle between incident light and the normal (perpendicular vector) of a surface determines the fraction of the light's full intensity at which the surface is illuminated, as in

$$I_s = I_l\cos(\theta) \tag{eq. 5}$$

where I_s is the intensity of illumination of the surface, I_l is the intensity of the light, and q is the angle between $-\mathbf{D}_l$ (where $\mathbf{D}_l$ is the light direction vector) and the surface normal. If the inverse light vector and the surface normal are both unit vectors, then this calculation can be performed with four multiplies and three additions—and no explicit cosine calculations—as

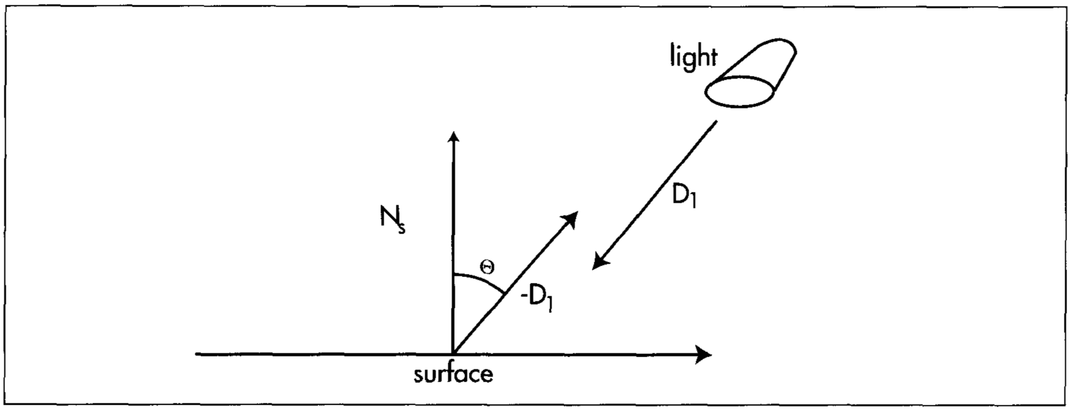
$$I_s = I_l(\mathbf{N}_s \bullet -\mathbf{D}_l), \tag{eq. 6}$$

where $\mathbf{N}_s$ is the surface unit normal and $\mathbf{D}_l$ is the light unit direction vector, as shown in Figure 61.2.

Cross Products and the Generation of Polygon Normals

One question equation 6 begs is where the surface unit normal comes from. One approach is to store the end of a surface normal as an extra data point with each polygon (with the start being some point that's already in the polygon), and transform it along with the rest of the points. This has the advantage that if the normal starts out as a unit normal, it will end up that way too, if only rotations and translations (but not scaling and shears) are performed.

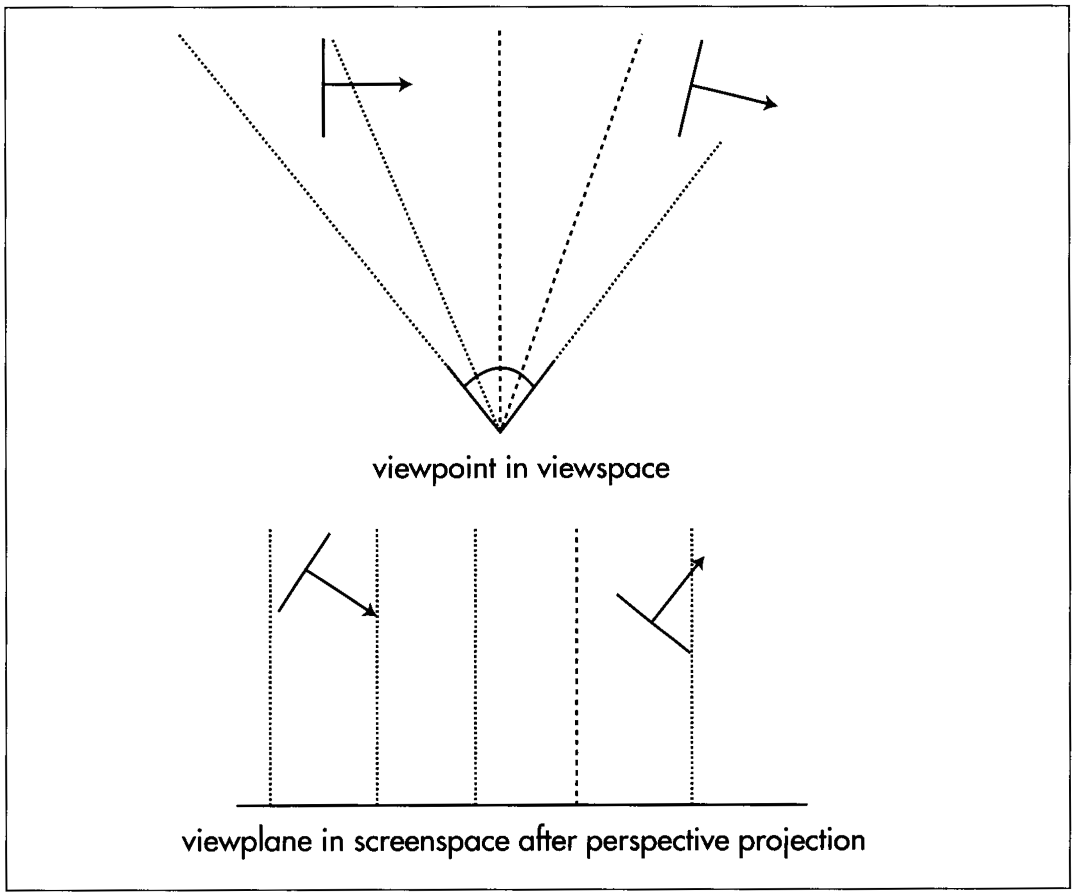
The problem with having an explicit normal is that it will remain a normal—that is, perpendicular to the surface—only through viewspace. Rotation, translation, and



The dot product as used in calculating lighting intensity.
Figure 61.2

scaling preserve right angles, which is why normals are still normals in viewspace, but perspective projection does not preserve angles, so vectors that were surface normals in viewspace are no longer normals in screenspace.

Why does this matter? It matters because, on average, half the polygons in any scene are facing away from the viewer, and hence shouldn't be drawn. One way to identify such polygons is to see whether they're facing toward or away from the viewer; that is, whether their normals have negative z values (so they're visible) or positive z values (so they should be culled). However, we're talking about screenspace normals here, because the perspective projection can shift a polygon relative to the viewpoint so that although its viewspace normal has a negative z, its screenspace normal has a positive z, and vice-versa, as shown in Figure 61.3. So we need screenspace normals, but those can't readily be generated by transformation from worldspace.



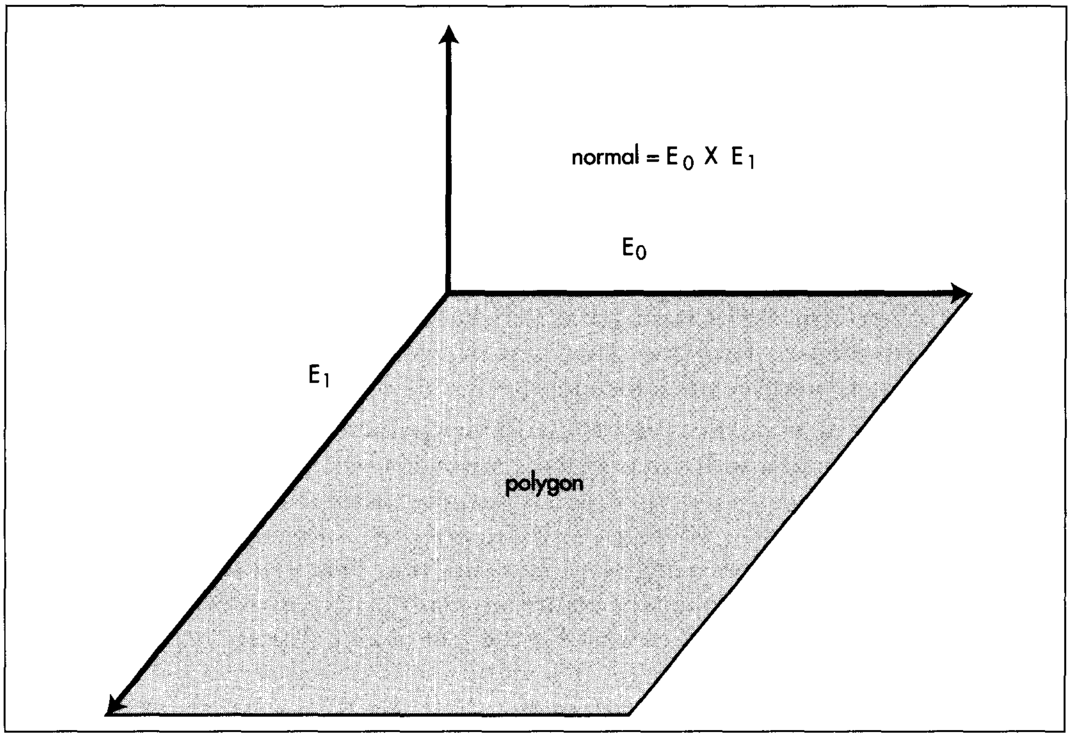
A problem with determining front/back visibility.
Figure 61.3

The solution is to use the cross product of two of the polygon's edges to generate a normal. The formula for the cross product is:

$$\mathbf{U} \times \mathbf{V} = \begin{bmatrix} u_2 v_3 - u_3 v_2 & u_3 v_1 - u_1 v_3 & u_1 v_2 - u_2 v_1 \end{bmatrix} \qquad \text{(eq. 7)}$$

(Note that the cross product operation is denoted by an X.) Unlike the dot product, the result of the cross product is a vector. Not just any vector, either; the vector generated by the cross product is perpendicular to both of the original vectors. Thus, the cross product can be used to generate a normal to any surface for which you have two vectors that lie within the surface. This means that we can generate the screenspace normals we need by taking the cross product of two adjacent polygon edges, as shown in Figure 61.4.

In fact, we can cull with only one-third the work needed to generate a full cross product; because we're interested only in the sign of the z component of the normal, we can skip entirely calculating the x and y components. The only caveat is to be careful that neither edge you choose is zero-length and that the edges aren't collinear, because the dot product can't produce a normal in those cases.



How the cross product of polygon edge vectors generates a polygon normal.
Figure 61.4

Perhaps the most often asked question about cross products is "Which way do normals generated by cross products go?" In a left-handed coordinate system, curl the fingers of your left hand so the fingers curl through an angle of less than 180 degrees from the first vector in the cross product to the second vector. Your thumb now points in the direction of the normal.

If you take the cross product of two orthogonal (right-angle) unit vectors, the result will be a unit vector that's orthogonal to both of them. This means that if you're generating a new coordinate space—such as a new viewing frame of reference—you only need to come up with unit vectors for two of the axes for the new coordinate space, and can then use their cross product to generate the unit vector for the third axis. If you need unit normals, and the two vectors being crossed aren't orthogonal unit vectors, you'll have to normalize the resulting vector; that is, divide each of the vector's components by the length of the vector, to make it a unit long.

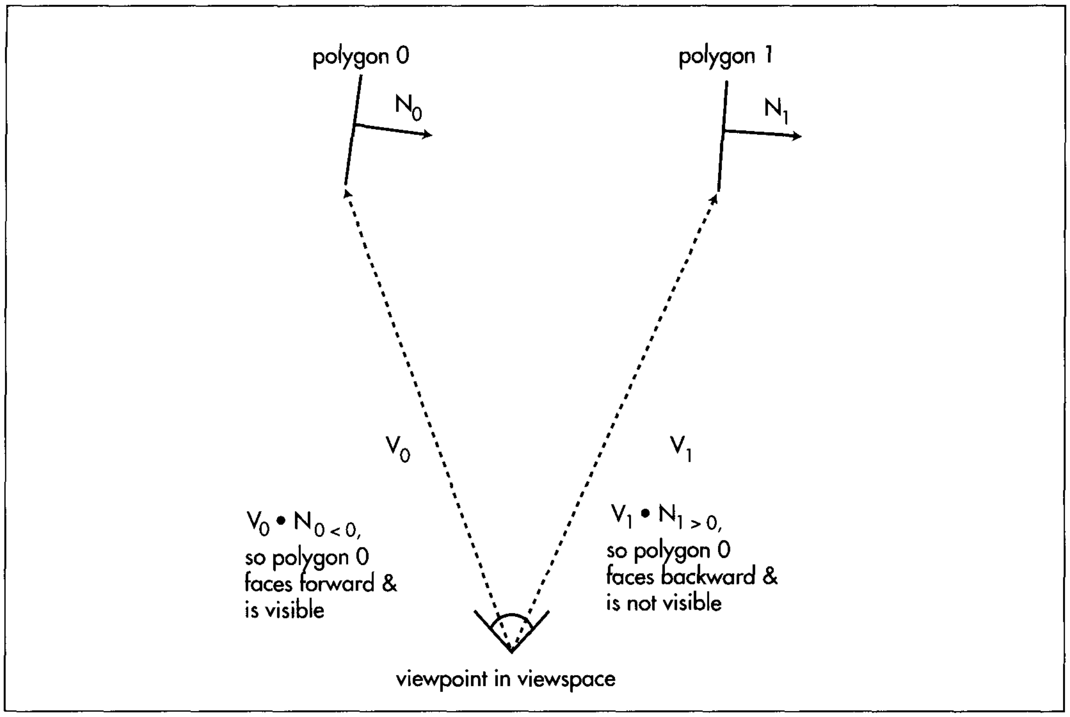
Using the Sign of the Dot Product

The dot product is the cosine of the angle between two vectors, scaled by the magnitudes of the vectors. Magnitudes are always positive, so the sign of the cosine determines the sign of the result. The dot product is positive if the angle between the vectors is less than 90 degrees, negative if it's greater than 90 degrees, and zero if the angle is exactly 90 degrees. This means that just the sign of the dot product suffices for tests involving comparisons of angles to 90 degrees, and there are more of those than you'd think.

Consider, for example, the process of backface culling, which we discussed above in the context of using screenspace normals to determine polygon orientation relative to the viewer. The problem with that approach is that it requires each polygon to be transformed into viewspace, then perspective projected into screenspace, before the test can be performed, and that involves a lot of time-consuming calculation. Instead, we can perform culling way back in worldspace (or even earlier, in objectspace, if we transform the viewpoint into that frame of reference), given only a vertex and a normal for each polygon and a location for the viewer.

Here's the trick: Calculate the vector from the viewpoint to any vertex in the polygon and take its dot product with the polygon's normal, as shown in Figure 61.5. If the polygon is facing the viewpoint, the result is negative, because the angle between the two vectors is greater than 90 degrees. If the polygon is facing away, the result is positive, and if the polygon is edge-on, the result is 0. That's all there is to it—and this sort of backface culling happens before any transformation or projection at all is performed, saving a great deal of work for the half of all polygons, on average, that are culled.

Backface culling with the dot product is just a special case of determining which side of a plane any point (in this case, the viewpoint) is on. The same trick can be applied whenever you want to determine whether a point is in front of or behind a plane,


polygon 0
polygon 1
N0
N1
V0
V1
V0 • N0 < 0, so polygon 0 faces forward & is visible
V1 • N1 > 0, so polygon 0 faces backward & is not visible
viewpoint in viewspace

Backface culling with the dot product.
Figure 61.5

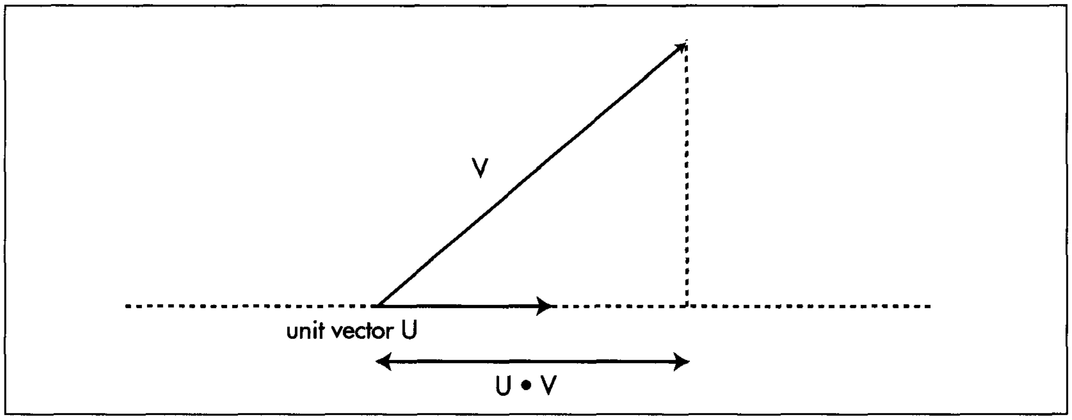
where a plane is described by any point that's on the plane (which I'll call the plane origin), plus a plane normal. One such application is in clipping a line (such as a polygon edge) to a plane. Just do a dot product between the plane normal and the vector from one line endpoint to the plane origin, and repeat for the other line endpoint. If the signs of the dot products are the same, no clipping is needed; if they differ, clipping is needed. And yes, the dot product is also the way to do the actual clipping; but before we can talk about that, we need to understand the use of the dot product for projection.

Using the Dot Product for Projection

Consider Equation 3 again, but this time make one of the vectors, say **V**, a unit vector. Now the equation reduces to:

$$\mathbf{U} \bullet \mathbf{V} = \cos(\theta)\|\mathbf{U}\| \qquad \text{(eq. 8)}$$

In other words, the result is the cosine of the angle between the two vectors, scaled by the magnitude of the non-unit vector. Now, consider that cosine is really just the



How the dot product with a unit vector performs a projection.
Figure 61.6

length of the adjacent leg of a right triangle, and think of the non-unit vector as the hypotenuse of a right triangle, and remember that all sides of similar triangles scale equally. What it all works out to is that the value of the dot product of any vector with a unit vector is the length of the first vector projected onto the unit vector, as shown in Figure 61.6.

This unlocks all sorts of neat stuff. Want to know the distance from a point to a plane? Just dot the vector from the point **P** to the plane origin $\mathbf{O}_p$ with the plane unit normal $\mathbf{N}_p$, to project the vector onto the normal, then take the absolute value

$$\text{distance} = |(\mathbf{P} - \mathbf{O}_p) \cdot \mathbf{N}_p|$$

as shown in Figure 61.7.

Want to clip a line to a plane? Calculate the distance from one endpoint to the plane, as just described, and dot the whole line segment with the plane normal, to get the full length of the line along the plane normal. The ratio of the two dot products is then how far along the line from the endpoint the intersection point is; just move along the line segment by that distance from the endpoint, and you're at the intersection point, as shown in Listing 61.1.

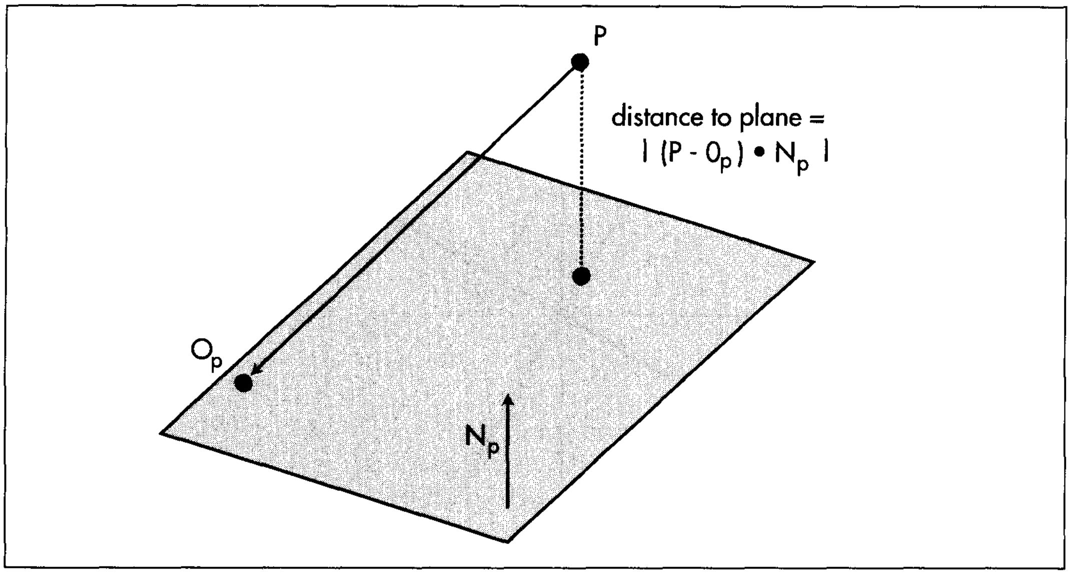
LISTING 61.1 L61_1.C

```
// Given two line endpoints, a point on a plane, and a unit normal
// for the plane, returns the point of intersection of the line
// and the plane in intersectpoint.
#define DOT_PRODUCT(x,y)   (x[0]*y[0]+x[1]*y[1]+x[2]*y[2])
void LineIntersectPlane (float *linestart, float *lineend,
   float *planeorigin, float *planenormal, float *intersectpoint)
{
   float vec1[3], projectedlinelength, startdistfromplane, scale;
   vec1[0] = linestart[0] - planeorigin[0];
   vec1[1] = linestart[1] - planeorigin[1];
```

```
    vec1[2] = linestart[2] - planeorigin[2];
    startdistfromplane = DOT_PRODUCT(vec1, planenormal);
    if (startdistfromplane == 0)
    {
        // point is in plane
        intersectpoint[0] = linestart[0];
        intersectpoint[1] = linestart[1];
        intersectpoint[2] = linestart[1];
        return;
    }
    vec1[0] = linestart[0] - lineend[0];
    vec1[1] = linestart[1] - lineend[1];
    vec1[2] = linestart[2] - lineend[2];
    projectedlinelength = DOT_PRODUCT(vec1, planenormal);
    scale = startdistfromplane / projectedlinelength;
    intersectpoint[0] = linestart[0] - vec1[0] * scale;
    intersectpoint[1] = linestart[1] - vec1[1] * scale;
    intersectpoint[2] = linestart[1] - vec1[2] * scale;
}
```

Rotation by Projection

We can use the dot product's projection capability to look at rotation in an interesting way. Typically, rotations are represented by matrices. This is certainly a workable representation that encapsulates all aspects of transformation in a single object, and is ideal for concatenations of rotations and translations. One problem with matrices, though, is that many people, myself included, have a hard time looking at a matrix of sines and cosines and visualizing what's actually going on. So when two 3-D experts, John



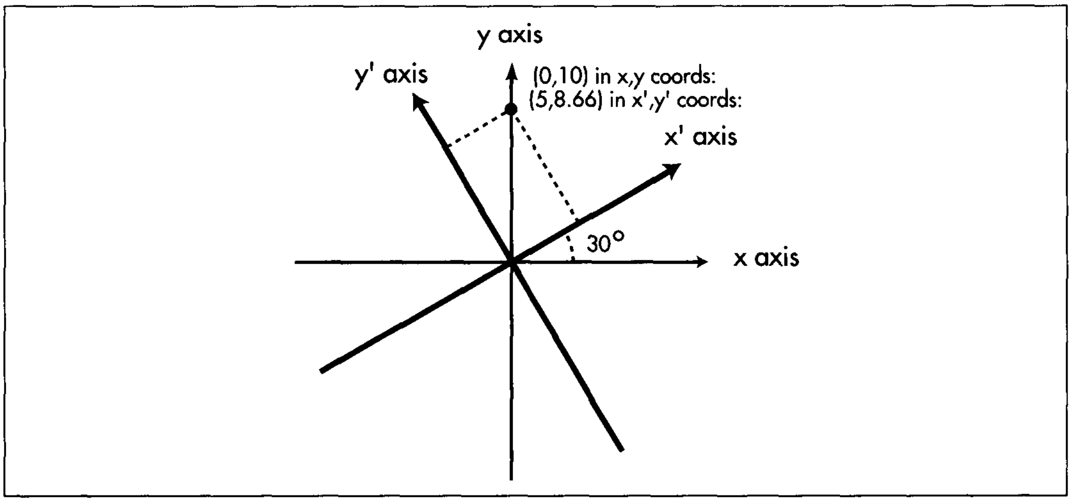
Using the dot product to get the distance from a point to a plane.
Figure 61.7

Carmack and Billy Zelsnack, mentioned that they think of rotation differently, in a way that seemed more intuitive to me, I thought it was worth passing on.

Their approach is this: Think of rotation as projecting coordinates onto new axes. That is, given that you have points in, say, worldspace, define the new coordinate space (viewspace, for example) you want to rotate to by a set of three orthogonal unit vectors defining the new axes, and then project each point onto each of the three axes to get the coordinates in the new coordinate space, as shown for the 2-D case in Figure 61.8. In 3-D, this involves three dot products per point, one to project the point onto each axis. Translation can be done separately from rotation by simple addition.

Rotation by projection is exactly the same as rotation via matrix multiplication; in fact, the rows of a rotation matrix are the orthogonal unit vectors pointing along the new axes. Rotation by projection buys us no technical advantages, so that's not what's important here; the key is that the concept of rotation by projection, together with a separate translation step, gives us a new way to look at transformation that I, for one, find easier to visualize and experiment with. A new frame of reference for how we think about 3-D frames of reference, if you will.

Three things I've learned over the years are that it never hurts to learn a new way of looking at things, that it helps to have a clearer, more intuitive model in your head of whatever it is you're working on, and that new tools, or new ways to use old tools, are Good Things. My experience has been that rotation by projection, and dot product tricks in general, offer those sorts of benefits for 3-D.



Rotation to a new coordinate space by projection onto new axes.
Figure 61.8