

## section 2: ActionScript Interfaces

### chapter 10: 3D with the drawing API

I remember when I was but a small lad and my parents took me to the movies to see the latest attempt at 3D film. It was absolutely awful in retrospect, but to my adolescent mind it was an extraordinary experience. I walked out of that cinema in awe and said to my parents, entirely in earnest, “I wish the *real* world was 3D, too!”

My point (other than to suggest I was a slow-minded youth) is that 3D work can easily inspire wonder and amazement, but if its only purpose is to attempt to inspire this wonder and amazement, chances are you'll end up with something like *Jaws 3D*. Like any effect or technique used in site design and animation, 3D should be included to support the content, not simply for eye candy. Keep this fact in mind as you work through these next two chapters, and think about how you might use Flash 3D elements to support and enhance your own designs.

#### Methods for 3D

Before we get into utilizing the drawing API to create 3D shapes through code, let's quickly take an important look at the alternative methods of incorporating 3D content in your Flash work.

#### Pre-rendered

This is the easiest method of the lot, if you have the resources. A program like Electric Rain's Swift 3D is dedicated to producing vector images for your Flash movies. Electric Rain also produces plug-ins to export Flash movies from Discreet's 3ds max, NewTek's LightWave, and Softimage. Electric Image's Amorphium Pro also produces Flash output that can be brought directly into your movies. Using these programs and others, you can quickly render stills or multiple frames of animation, and then use ActionScript to manipulate the images inside Flash. Drawbacks of these programs include the cost, of course, but also the file size of the exported

movies can be very large at times. Still, these programs enable you to produce images that are simply not possible within the limitations Flash, as we'll discuss shortly.

### Approximated illustrations

Believe me, artists were producing 3D imagery long before the invention of the microprocessor. If all you are looking to produce is a static 3D image, then there's no reason why you couldn't make this yourself without the aid of any expensive 3D software. With an image-editing program like Adobe Photoshop you can create faux-3D images that are startlingly real (check out Bert Monroy's web site at [www.bertmonroy.com](http://www.bertmonroy.com) to see some fantastic evidence of this). Even within Flash, you can still produce some terrific 3D effects and images using only the drawing tools and knowledge of perspective and lighting.

### Real-time 3D

And here's the reason you're reading this chapter in the first place. With a little math (um, or a lot of math), you can create 3D objects at run-time just using code. The drawing API is a big help here since the creation and filling of polygons at run-time is now possible. Though you could work around this in Flash 5 by skewing triangles, it was an imperfect solution as gaps often appeared in objects and the skewing calculations bogged down the processor.

The processing load is still a concern with 3D in Macromedia Flash MX, but there are ways to ease the strain to a certain extent. One way is to avoid attempting to create complete, animated, 3D environments using ActionScript alone. You're not going to make *Halo*, sorry. If you wish to explore fully interactive 3D worlds then look into Flash's big brother, Macromedia Director. Despite debates in the online Flash forums over the relevancy of Director and the equality between the two programs, Director is simply *more powerful* and the 3D capabilities of its latest version can perform astonishing things that are impossible in Flash.

But wait! Before you head off grumbling that you bought the wrong program (we both know that's not true), the benefits of Flash 3D can outweigh those of Shockwave 3D (produced in Director). For one, the Flash Player is simply more ubiquitous than the Shockwave Player. This means that more people will be able to view your Flash 3D content immediately, while they would have to download the relatively larger player to view your Shockwave 3D content. Also, because of the complexity of some of the Shockwave 3D environments, the file sizes are much larger, and sometimes too great for users on slower connections. Until broadband connections are the norm, Flash has the upper hand.

### 3D concepts reviewed

It's important to review the basic concepts of 3D before we start on the code that will produce our Flash 3D. Now, don't start flipping back through the pages to find where these 3D concepts were discussed initially; they weren't. But, if we're to get to the practical applications of coded, run-time 3D, we have to get started quickly. Because of this, and because friends of ED have

already published numerous books with chapters dealing with these concepts (such as Michael Bedar's comprehensive chapter in *Flash 5 ActionScript Studio*), I'm going to work with the assumption that you have at least explored some of these ideas yourself. In this way we can quickly get to the exciting new ground of utilizing the drawing API to extend these standard concepts and methods.

## Axes

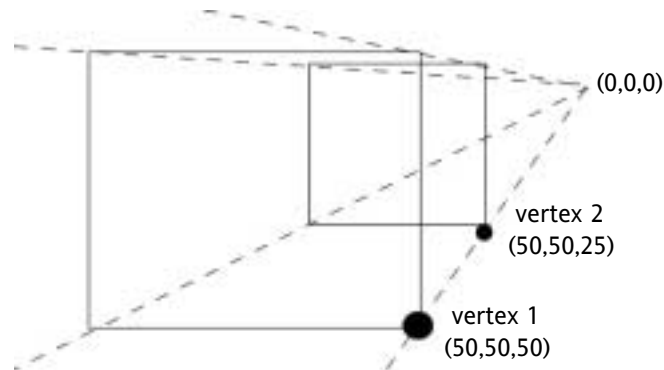
Your computer screen has two axes,  $x$  and  $y$ . You use these all the time when accessing the `_x`, `_y`, `_xscale` and `_yscale` properties of your objects. The  $x$ -axis runs horizontally across your screen while the  $y$ -axis runs vertically. In 3D, as the name implies, we have a third dimension, represented by the  $z$ -axis, which, for us at least, runs perpendicular to the screen. It's the placement of a point on the  $z$ -axis that gives us depth, and helps create the illusion of 3D. In our code, a positive value on the  $z$ -axis indicates that it is nearer to the viewer.

## Vertex

The smallest object in our 3D world is the vertex, which is simply a point in 3D space represented by its three coordinate values of  $x$ ,  $y$ , and  $z$ . Vertices are the building blocks of our 3D objects.

## Depth

Since we are our outputting to a 2D environment (the computer screen), we need a way to convert our 3D coordinates into 2D space. We do this by adjusting the screen `_x` and `_y` properties of our vertices (not to be confused with the vertices' 3D  $x$  and  $y$  coordinates) based on the vertices'  $z$  value. The larger the  $z$  value, the closer the vertex is to the viewer, and therefore the farther the vertex is moved away from world center, which serves as our 2D vanishing point. Here's a simple illustration of this process:



Although vertex 1 and vertex 2 have identical 3D  $x$  and  $y$  values (remembering that the 3D values are separate from the Flash screen coordinates), their  $z$  values differ, so we need to simulate depth between the two points. Vertex 1, having a larger  $z$  value, is moved further away from world

center, and depth simulation is achieved. The formula we will use to find out *how* we move a vertex from world center is:

```
scale = focalLength/(focalLength-z);  
_x = x*scale;  
_y = y*scale;
```

`focalLength` represents the distance between the viewer and the screen. Imagine in the above scenario that we set the `focalLength` to 100. For vertex 2, the `scale` variable will evaluate to approximately 1.33, which will set its Flash `_x` and `_y` properties to -66.5 and 133, respectively. This is the screen distance of the vertex from world center measured in pixels. Vertex 1, with its `scale` variable evaluating to 2, will have its Flash `_x` and `_y` properties set to -100 and 200, respectively. This sets the vertex's screen coordinates (again, separate from its 3D coordinates) further away from world center. By adjusting `focalLength`, we can increase or decrease this perspective distortion.

### Vectors

We'll discuss these further when we get into lighting, but you should at least be aware that a vector is a quantity in our 3D world consisting of a direction and a magnitude. Imagine a theater spotlight shining down from the ceiling on to a soloist on stage. This is a vector as it has a direction that it's pointing in and also has a specific length to its beam.

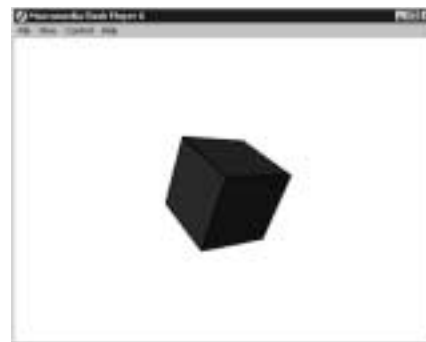
### Matrices

Remember these from the last chapter? Yes, they're back, and this time it's personal (sorry, *Jaws 3D* is still on my mind). Matrices, as you'll recall, are rectangular arrays of numbers. We'll use matrices to help us in transforming our 3D world in much the same way they helped us transform our gradients in the last chapter.

Those are the basic ideas and terms you need to have a firm grasp on, but hey, enough of my yakking! Let's move on and apply these ideas in Flash.

### Coding the cube

In this exercise we're going to create the ultimate Flash 3D exercise to demonstrate the concepts we've looked at so far and to give us a base to build on: it's the well-worn but underrated spinning cube.



## Setting the vertices

1. Create a new movie and save it as `spinning_cube_0.fla`. Create a new movie clip symbol with a central registration point called `vertex` and make it a small circle (mine's 12.5 in diameter). In the **Linkage Properties**, check **Export for ActionScript** and give it the identifier name `vertex`.

So now we have the **vertex** symbol in the Library. Eventually, the vertices will only be objects in memory and no longer actual movie clip symbols, but for this exercise we'll utilize them as symbols. Now on to the code.

2. Go back into the main timeline and rename the default layer `actions`. In frame 1 type the following code:

```
this.createEmptyMovieClip("center", 0);
center._x = Stage.width/2;
center._y = Stage.height/2;
focalLength = 400;
```

We've initially created a new movie clip that will hold our 3D world and place it at stage center. We've set our `focalLength` variable (to be used for our depth simulation) to 400.

3. We next create our cube model by defining the vertices. Add this beneath the previous code:

```
cube = {};
cube.vertexList = [];
cube.vertexList.push({x:-50, y:-50, z:50});
cube.vertexList.push({x:50, y:-50, z:50});
cube.vertexList.push({x:50, y:-50, z:-50});
cube.vertexList.push({x:-50, y:-50, z:-50});
cube.vertexList.push({x:-50, y:50, z:50});
cube.vertexList.push({x:50, y:50, z:50});
cube.vertexList.push({x:50, y:50, z:-50});
cube.vertexList.push({x:-50, y:50, z:-50});
vertices = [];
for (i=0; i<cube.vertexList.length; i++) {
    center.attachMovie("vertex", "v"+i, i);
    vertices.push(center["v"+i]);
}
```

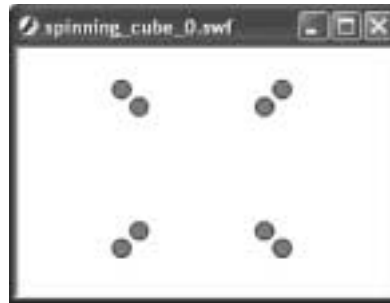
`cube` is a new object that we give a single property to: `vertexList`. This will be an array to hold each individual vertex. Over the following eight lines, we push the vertices into the array. Note that each vertex is an individual object made up of three properties: `x`, `y`, and

z. Using the length of our array, we place physical vertices (the symbol stored in the Library) on the stage to represent our code vertices. We place a reference to this movie clip in our `vertices` array. Again, this is something we'll dispense with later on as we'll exclusively use the drawing API to produce 3D visuals.

4. Now we need to write the code that will place the vertices in their correct screen positions:

```
render = function (model) {
    for (var i = 0; i < model.vertexList.length; i++) {
        var scale = focalLength / (focalLength - model.vertexList[i].z);
        vertices[i]._x = model.vertexList[i].x * scale;
        vertices[i]._y = model.vertexList[i].y * scale;
    }
};
render(cube);
```

`render` runs through our given model's `vertexList` and sets each vertex's screen position using the depth formula discussed earlier. Finally, we call our function. Go ahead and test the movie to see the result:



`spinning_cube_0.swf` shows perspective distortion applied to 8 vertices defining a cube. Simulating depth is necessary when attempting to display 3D on a 2D output device like a computer's monitor.

*Note that we are not altering the scale or swapping depths of our vertices as you might expect. This is because we are not attempting to draw perspective spheres as was often seen in Flash 5 experiments, but rather vertices that will define polygons. The vertices themselves do not scale, so we will not apply scaling at this early stage.*

Not too impressive yet: the cube isn't spinning so it's difficult to view the vertices' location. We'll change this, at least briefly, with the following function:

5. Type this directly after the render function:

```
rotateY = function (model, degree) {
    var sin = Math.sin(degree*Math.PI/180);
    var cos = Math.cos(degree*Math.PI/180);
    for (var i = 0; i<model.vertexList.length; i++) {
        var x = cos*model.vertexList[i].x-sin*model.vertexList[i].z;
        var z = cos*model.vertexList[i].z+sin*model.vertexList[i].x;
        model.vertexList[i].z = z;
        model.vertexList[i].x = x;
    }
};
```

This will rotate the vertices of our model about the world's y-axis. We first find the sin and cos of the angle sent (in degrees, so we must convert the number to radians) and then we go through each vertex in our model and move the vertex based on the angle. The formulas for rotation about the axes are established formulas, and to save pages I'll work with the assumption that you accept that they are true.

6. Finally, replace the last line `render(cube);` with the following code:

```
center.onEnterFrame = function() {
    rotateY(cube, 3);
    render(cube);
};
```

This final code rotates our cube by 3 degrees about the y-axis and then re-renders it to the screen. Test your movie to view the rotating cube. (If you get some unexpected results here you can check your file against our `spinning_cube_0.fla` on the CD).

## Adding meshlines

The first thing we need to do to more accurately simulate a 3D cube is to join up the vertices of the model with meshlines to create a wireframe. This could be accomplished in Flash 5, but is now far easier in Flash MX as we can use the drawing API.

1. Save your existing movie as `spinning_cube_1.fla`. What we'll do now is add some code to our render function in order to draw some lines between our vertices. To do this, we'll need to know which vertices need to be attached to each other. We'll accomplish this by assigning sides to our model made up of selected vertices. Add this code right after the

previous lines that defined the vertices of `cube` (beneath the last line which began `cube.vertexList.push`):

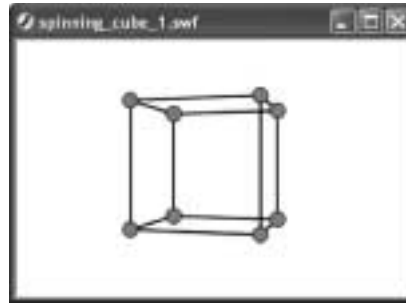
```
cube.side = [];
cube.side.push([0, 1, 2, 3]);
cube.side.push([2, 1, 5, 6]);
cube.side.push([1, 0, 4, 5]);
cube.side.push([5, 4, 7, 6]);
cube.side.push([0, 3, 7, 4]);
cube.side.push([3, 2, 6, 7]);
```

`side` is another array property of our model. Each index consists of four vertices that make up a cube side (try sketching it out from the vertex coordinates to see how these relate). The order that the vertices are placed in each `side` index is very important as you'll see when we come to **backface culling**, which is the process of making invisible any side that isn't facing the viewer (thus saving us from excess processing). We will discuss why in a few pages time. For the time being, just make sure you keep the same order of vertices for each side (the order of sides, however, is of no importance).

2. We now need to edit our `render` function to actually draw the lines between the vertices. With the sides already defined, this is a fairly straightforward process. The added code is in bold:

```
render = function (model) {
    center.clear();
    center.lineStyle(2, 0, 100);
    for (var i = 0; i < model.vertexList.length; i++) {
        var scale = focalLength / (focalLength - model.vertexList[i].z);
        vertices[i]._x = model.vertexList[i].x * scale;
        vertices[i]._y = model.vertexList[i].y * scale;
    }
    for (var i = 0; i < model.side.length; i++) {
        center.moveTo(vertices[model.side[i][0]]._x,
        vertices[model.side[i][0]]._y);
        for (var j = 1; j < model.side[i].length; j++) {
            center.lineTo(vertices[model.side[i][j]]._x,
            vertices[model.side[i][j]]._y);
        }
        center.lineTo(vertices[model.side[i][0]]._x,
        vertices[model.side[i][0]]._y);
    }
};
```

What we are doing (after clearing and redefining our `lineStyle`) is looping through all of our sides and drawing lines between each vertex. We start by moving our pencil to the vertex at index 0 of our current side. We then draw a line connecting all the remaining vertices on the side, ending by drawing a line back to the initial vertex. Test your movie now to see the drawn meshlines:



Wireframe lines drawn by the drawing API have been added to help define the edges of our model.

The thing is, now that we've got the meshlines for our model, the vertex movie clips have become a little superfluous. We can get rid of them altogether and make our vertices exist solely in the computer's memory. The only issue we have to deal with here is how to find the x and y screen coordinates of the vertices without being able to look at movie clip `_x` and `_y` properties.

3. Delete the vertex symbol from your Library, as it's no longer needed. Also delete the following code from your script (these should be lines 23-26 in the Script pane):

```
for (i=0; i<cube.vertexList.length; i++) {
    center.attachMovie("vertex", "v"+i, i);
    vertices.push(center["v"+i]);
}
```

4. Now update your `render` function like so:

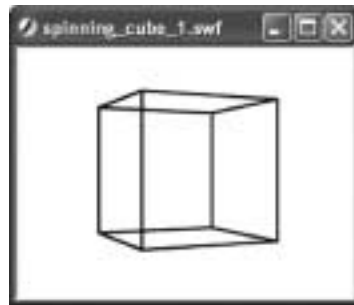
```
render = function (model) {
    center.clear();
    center.lineStyle(2, 0, 100);
    verts2D = [];
    for (var i = 0; i<model.vertexList.length; i++) {
        verts2D[i] = {};
        var scale = focalLength/(focalLength-model.vertexList[i].z);
        verts2D[i].x = model.vertexList[i].x*scale;
        verts2D[i].y = model.vertexList[i].y*scale;
    }
}
```

```

    }
    for (var i = 0; i<model.side.length; i++) {
        center.moveTo(verts2D[model.side[i][0]].x,
            ↪verts2D[model.side[i][0]].y);
        for (var j = 1; j<model.side[i].length; j++) {
            center.lineTo(verts2D[model.side[i][j]].x,
                ↪verts2D[model.side[i][j]].y);
        }
        center.lineTo(verts2D[model.side[i][0]].x,
            ↪verts2D[model.side[i][0]].y);
    }
};

```

There isn't too much difference. We create a new array called `verts2D` that stores objects made up of our adjusted screen coordinates. When we loop through our sides and vertices, we look into `verts2D` for these values. With only a few extra lines (really, we deleted more than we added), we've made our 3D spinning cube exist solely using ActionScript:



5. With the movie clip vertices removed, we have a 3D wireframe model spinning about its y-axis completely created using the drawing API.
6. Save your movie and leave it open, as we'll be modifying it again once we've looked at some more theory.

## Transformation matrices

I'd like to discuss another concept here and it concerns how we're handling our transformations. As it is, for each rotation about an axis (even though we only presently have rotation about the y-axis, you can probably surmise the code for the other two axes) we are looping through each vertex and adjusting its coordinates accordingly. Wouldn't it be nice to only have to do this once, after we have calculated all the necessary transformations? Well, we can, and the way to do it is by using matrices.

Although you might not have thought it at the time, each vertex can be described as a matrix too, without requiring us to adjust any code. A vertex with the coordinates (50,30,15) can be written just as easily as:

$$\left\{ \begin{array}{c} 50 \\ 30 \\ 15 \end{array} \right\}$$

You don't have to change a thing in your code (it's not as if you can format it this way in the ActionScript panel!), just as long as you understand what this matrix means. Um, so what does it mean? Well, you saw how easily we could perform transformations on our gradients using our transforms stored in matrices? Well, we can do the same thing for our 3D transforms.

In our last exercise, for example, in the `rotateY` function the transformation is handled by:

```
x = cos(?)*x - sin(?)*z;
//y = y;
z = cos(?)*z + sin(?)*x;
```

The commented out line was not in our previous code, but I've put it there as a placeholder so you can see where we're going with this.

Now what if I told you that this transform formula could be stored in a transform matrix that we could multiply with our 3D coordinates? What if I told you that these transform matrices could actually be multiplied together, creating a cumulative transform matrix that we would only have to multiply with our coordinates once at the end of our transformations? What if I told you that doing so could save us some processing and aid us in future transformations? What if I keep asking rhetorical questions and never get to show you how this is done? Nah, that'll never happen. The transform matrix for the above code looks like this:

$$\left\{ \begin{array}{ccc} \cos(\_) & 0 & -\sin(\_) \\ 0 & 1 & 0 \\ \sin(\_) & 0 & \cos(\_) \end{array} \right\}$$

## 10 Flash MX Studio

The result of multiplying this 3x3 matrix with our 3x1 coordinate matrix would be another 3x1 matrix:

$$\left\{ \begin{array}{l} \cos(\_) * x + 0 * y + -\sin(\_) * z \\ 0 * x + 1 * y + 0 * z \\ \sin(\_) * x + 0 * y + \cos(\_) * z \end{array} \right\}$$

Does this look familiar? That's right, it's the same formula as used by our `rotateY` function above! Now you might be thinking that it looks like a lot more work to implement this than its predecessor, but actually, if you set up the formula once (which only consists of simple multiplication and addition), you can use it for all of our transforms, the rest of which look would like this:

- Transformation about the x-axis:

$$\left\{ \begin{array}{lll} 1 & 0 & 0 \\ 0 & \cos(\_) & \sin(\_) \\ 0 & -\sin(\_) & \cos(\_) \end{array} \right\}$$

- Transformation about the z-axis:

$$\left\{ \begin{array}{lll} \cos(\_) & \sin(\_) & 0 \\ -\sin(\_) & \cos(\_) & 0 \\ 0 & 0 & 1 \end{array} \right\}$$

- Scaling of the object:

$$\left\{ \begin{array}{ccc} \%x & 0 & 0 \\ 0 & \%y & 0 \\ 0 & 0 & \%z \end{array} \right\}$$

Let's incorporate some of these into our code so that we might easily perform these transformations.

## Transforming with matrices

1. Still using your movie from the last exercise, save it as `spinning_cube_2.fla`. Then enter this code immediately after our `render` function:

```
rotateX = function (model, degree) {
    var rad = degree*Math.PI/180;
    var sin = Math.sin(rad);
    var cos = Math.cos(rad);
    var matrix = {a:1, b:0, c:0, d:0, e:cos, f:sin, g:0, h:-sin,
    ↪i:cos};
    transform(matrix, model);
};
rotateY = function (model, degree) {
    var rad = degree*Math.PI/180;
    var sin = Math.sin(rad);
    var cos = Math.cos(rad);
    var matrix = {a:cos, b:0, c:-sin, d:0, e:1, f:0, g:sin, h:0,
    ↪i:cos};
    transform(matrix, model);
};
rotateZ = function (model, degree) {
    var rad = degree*Math.PI/180;
    var sin = Math.sin(rad);
    var cos = Math.cos(rad);
    var matrix = {a:cos, b:sin, c:0, d:-sin, e:cos, f:0, g:0, h:0,
    ↪i:1};
    transform(matrix, model);
};
```

```

};
scale = function (model, percent) {
    var rad = degree*Math.PI/180;
    var matrix = {a:percent, b:0, c:0, d:0, e:percent, f:0, g:0, h:0,
        ↳i:percent};
    transform(matrix, model);
};

```

You could easily set these matrices up as linear arrays, but I've chosen to make them objects similar to our gradient transformation matrices from the last chapter so that you can see the relation. After we've made our matrix, we call our `transform` function to add this matrix to our full transform. Let's write this function next.

2. Under the matrix definitions we added in the last step, add the following code:

```

transform = function (matrix, model) {
    if (transformMatrix) {
        var a = matrix.a*transformMatrix.a+matrix.b*transformMatrix.d
        ↳+matrix.c*transformMatrix.g;
        var b = matrix.a*transformMatrix.b+matrix.b*transformMatrix.e
        ↳+matrix.c*transformMatrix.h;
        var c = matrix.a*transformMatrix.c+matrix.b*transformMatrix.f
        ↳+matrix.c*transformMatrix.i;
        var d = matrix.d*transformMatrix.a+matrix.e*transformMatrix.d
        ↳+matrix.f*transformMatrix.g;
        var e = matrix.d*transformMatrix.b+matrix.e*transformMatrix.e
        ↳+matrix.f*transformMatrix.h;
        var f = matrix.d*transformMatrix.c+matrix.e*transformMatrix.f
        ↳+matrix.f*transformMatrix.i;
        var g = matrix.g*transformMatrix.a+matrix.h*transformMatrix.d
        ↳+matrix.i*transformMatrix.g;
        var h = matrix.g*transformMatrix.b+matrix.h*transformMatrix.e
        ↳+matrix.i*transformMatrix.h;
        var i = matrix.g*transformMatrix.c+matrix.h*transformMatrix.f
        ↳+matrix.i*transformMatrix.i;
        transformMatrix = {a:a, b:b, c:c, d:d, e:e, f:f, g:g,
            ↳h:h,i:i};
    } else {
        transformMatrix = matrix;
    }
};

```

Though this might look a bit daunting, this is simply the formula for multiplying a 3x3 matrix by another 3x3 matrix. What we're doing in this function is first checking to see if a transform matrix already exists. If it does, we multiply the two matrices together, thus

forming a cumulative matrix of the two (if a transform matrix doesn't yet exist, we create one and set it equal to the matrix sent to the function). The result of this is that we can combine multiple transformations into one matrix *before* we apply it to our vertices. Let's do that now.

3. Add the following bold lines to the `render` function:

```
render = function (model) {
  if (transformMatrix) {
    for (var i = 0; i < model.vertexList.length; i++) {
      var vert = model.vertexList[i];
      var x = transformMatrix.a*vert.x+transformMatrix.b*vert.y
      ↪+transformMatrix.c*vert.z;
      var y = transformMatrix.d*vert.x+transformMatrix.e*vert.y
      ↪+transformMatrix.f*vert.z;
      var z = transformMatrix.g*vert.x+transformMatrix.h*vert.y
      ↪+transformMatrix.i*vert.z;
      vert.x = x;
      vert.y = y;
      vert.z = z;
    }
    delete transformMatrix;
  }

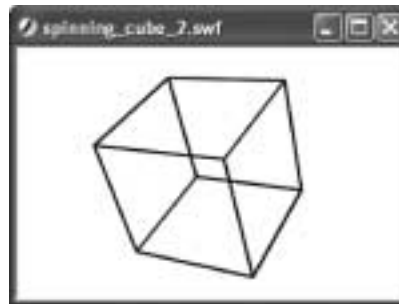
  center.clear();
  center.lineStyle(2, 0, 100);
  verts2D = [];
  for (var i = 0; i < model.vertexList.length; i++) {
    verts2D[i] = {};
    var scale = focalLength/(focalLength-model.vertexList[i].z);
    verts2D[i].x = model.vertexList[i].x*scale;
    verts2D[i].y = model.vertexList[i].y*scale;
  }
  for (var i = 0; i < model.side.length; i++) {
    center.moveTo(verts2D[model.side[i]][0].x,
      ↪verts2D[model.side[i]][0].y);
    for (var j = 1; j < model.side[i].length; j++) {
      center.lineTo(verts2D[model.side[i]][j].x,
        ↪verts2D[model.side[i]][j].y);
    }
    center.lineTo(verts2D[model.side[i]][0].x,
      verts2D[model.side[i]][0].y);
  }
};
```

Finally, before we render the model to the screen, we need to apply our transformations (which are conveniently stored in a single matrix) to our vertices. The extra code is the basic formula for multiplying a 3x3 matrix (our transform) with a 3x1 matrix (each vertex's coordinates). We then delete our transform to ready ourselves for further transformations.

4. To call our new rotation functions, alter the `onEnterFrame` function at the end of our code to read:

```
center.onEnterFrame = function() {  
    rotateX(cube, 3);  
    rotateY(cube, 6);  
    rotateZ(cube, 10);  
    render(cube);  
};
```

Notice that although we perform three separate transformations, they will not be applied to the model until the `render` function is called, once each frame. Go ahead and test the movie to see the results.



Using 3x3 transformation matrices, the rotation of our wireframe model about all three of its axes is made more efficient.

If you get any unexpected results then you can check your code against mine in `spinning_cube_2.fla` on the CD.

Now we're getting somewhere! Even though all we have is the same old spinning wireframe cube, we're building something that will provide a solid foundation for us to improve on.

### Depth sorting

One improvement that will be necessary once we begin filling the models with color is a **depth sorting** or **z-sorting** method. This is a method of placing models or the sides of the cube that are nearer to the viewer *in front of* objects and sides that are further away. Basically, you don't

want an object with a  $z=-200$  coordinate to be in front of an object that has a  $z=50$  coordinate. In Flash 5, you needed to swap the depths of your vertex movie clips in order to accomplish this. Now in MX, we can simply create an order to draw in, starting with the sides that are furthest away and working towards the viewer, effectively drawing over the sides that should be behind. This will all be accomplished in our `render` function, as we'll see in the following exercise.

### Filling the sides

1. Save your movie as `spinning_cube_3.fla`. Update your `render` function with the following new code:

```
render = function (model) {
    if (transformMatrix) {
        for (var i = 0; i<model.vertexList.length; i++) {
            var vert = model.vertexList[i];
            var x = transformMatrix.a*vert.x+transformMatrix.b*vert.y
                ↳+transformMatrix.c*vert.z;
            var y = transformMatrix.d*vert.x+transformMatrix.e*vert.y
                ↳+transformMatrix.f*vert.z;
            var z = transformMatrix.g*vert.x+transformMatrix.h*vert.y
                ↳+transformMatrix.i*vert.z;
            vert.x = x;
            vert.y = y;
            vert.z = z;
        }
        delete transformMatrix;
    }
    center.clear();
    center.lineStyle(2, 0, 100);
    verts2D = [];
    depthArray = [];
    for (var i = 0; i<model.side.length; i++) {
        var zDepth = 0;
        for (var j = 0; j<model.side[i].length; j++) {
            var whichVert = model.side[i][j];
            if (verts2D[whichVert] == undefined) {
                verts2D[whichVert] = {};
                var scale = focalLength/(focalLength
                    ↳-model.vertexList[whichVert].z);
                verts2D[whichVert].x = model.vertexList[whichVert].x
                    ↳*scale;
                verts2D[whichVert].y = model.vertexList[whichVert].y
                    ↳*scale;
            }
        }
    }
}
```

```

    }
    zDepth += model.vertexList[whichVert].z;
  }
  depthArray.push([model.side[i], zDepth]);
}
depthArray.sort(function (a, b) { return a[1]>b[1];});
for (var i = 0; i<depthArray.length; i++) {
  var sideVerts = depthArray[i][0];
  center.moveTo(verts2D[sideVerts[0]].x,verts2D
  ↳[sideVerts[0]].y);
  center.beginFill(0x666666, 100);
  for (var j = 1; j<sideVerts.length; j++) {
    center.lineTo(verts2D[sideVerts[j]].x,verts2D
    ↳[sideVerts[j]].y);
  }
  center.lineTo(verts2D[sideVerts[0]].x,verts2D
  ↳[sideVerts[0]].y);
  center.endFill();
}
};

```

Wow. That's a lot of changes, but they do quite a lot for us. We've some nested arrays in here that might look a little convoluted, so let's work through it and see what this code is doing.

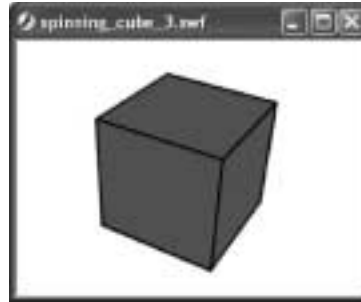
First, we create a new array called `depthArray`. This will hold references to our models' sides and the collective z position of each vertex in that side (this will give us a close enough z level for our purposes here). We then loop through each side in our model and convert its vertices into 2D coordinates (which is no different to our previous version) and add each vertex's z value to that side's total `zDepth`. The `if` statement is included so that we don't perform any unnecessary conversion operations more than once on a vertex, which would indeed happen since sides share vertices with other sides.

Once the vertices have been converted and the total `zDepth` of each side is recorded, we place a reference to each side (which holds that side's vertices, remember) and its `zDepth` into our `depthArray`, which we promptly sort using a sorting function. We used an almost identical sorting function in the previous chapter, but here we include it directly in the sort arguments instead of calling it externally. Once our `depthArray` is in the correct order (from the side furthest away through to the nearest side) we loop through it and draw our lines and fills.

The two most confusing variables in the above code might be `whichVert` and `sideVerts`, so let's look more closely at these. `whichVert` holds a single vertex number for a side in the model. So for `side[0]`, which is an array of the numbers `[0,1,2,3]`, `whichVert` will evaluate to each index on the corresponding iteration of the loop. `sideVerts`, on the

other hand, will hold that full array of vertex numbers. Remember that each index of `depthArray` holds a reference to one of the model's sides and its `zDepth`. By looking at the first index position in a particular index of `depthArray`, we can find that side's list of vertices, which we then place in `sideVerts`.

2. Test your movie. We now have filled sides! Thanks, drawing API!



`spinning_cube_3.swf` demonstrates a cube model with solid-filled polygons and visible edges, all created with the drawing API. Depth sorting of each polygon allows for nearer polygon sides to appear in front of further sides. The cube is spun about all three of its axes each frame using 3x3 transformation matrices.

3. Save your movie and leave it open for the next exercise.

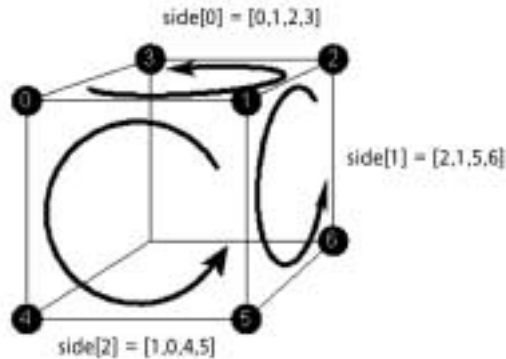
## Backface culling

Backface culling, as stated earlier, is simply preventing the rendering of polygons that are turned away from the viewer, saving the computer excess processing work (in fact, it's the polygon's *normal* directed away from the viewer, but unfortunately there's not much time to get into this here). This is desirable most of the time, though you can certainly make this option a toggled state for your models if you so choose. The way to determine if a polygon is facing the viewer or turned away is by examining the **screen** coordinates of its vertices, not the **3D** coordinates, since it's the model's relation to the screen and the viewer that is the important determining factor on whether the polygons can be seen or not. We also need to note if the vertices are in a clockwise or counter-clockwise order. With our method, it is imperative that the vertices are defined for the side in a counter-clockwise order in relation to the object center, in order for them to be seen at the correct angles.

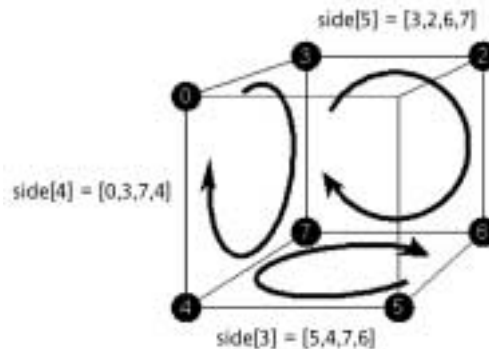
This all sounds a bit confusing, doesn't it? Well, once you get the hang of it, it's pretty easy. Trust me. In fact, explaining it is the most difficult part. The process I use to help me determine the correct order of vertices is to imagine the polygon (or side) to be directly facing me on the *near side of center* before I determine the vertex order. For our cube model, this would mean mentally spinning the cube until each side faced me so that I might see the order for the vertices better.

## 10 Flash MX Studio

If I then place the vertices into the `side` array in a counter-clockwise order at the current view, the side will be fully visible to the viewer while on the near side of center. To visualize this better, we'll look at our cube model with three of its sides defined so that the outside of the cube may be seen:



Now although it might appear that we simply have to define all sides in a counter-clockwise order in relation to our view, take a look at the order for our remaining three sides:



Ah-hah! Clockwise! This is because in this view we *don't* want these sides to be rendered, since they are *inside* the box. It's for this reason that I use the procedure described above, that of mentally spinning the model so that each side faces me before I record the vertex order. If I spin `side[5]` around to face me on the near side of center, then the order of its vertices would indeed be counter-clockwise.

OK, now that we know that whenever a side's vertices are in a clockwise order on the screen it should *not* be rendered, we need a way to determine the order in which the vertices' 2D coordinates are currently seen. (Full credit must go to Pavils Jurjans at [www.jurjans.lv/flash](http://www.jurjans.lv/flash) for his open source code and this fun section).

## Rendering only the necessities

1. Resave your movie from the previous exercise as `spinning_cube_4 fla`. Create this new function above the `render` function (really, it doesn't matter too much where you put it in our code but this helps to keep our code more logical and organized):

```
backface = function (x, y) {
  var cax = x[2]-x[0];
  var cay = y[2]-y[0];
  var bcx = x[1]-x[2];
  var bcy = y[1]-y[2];
  return (cax*bcy<cay*bcx);
};
```

What we're going to do is send this function three vertices from a side. Using the screen coordinates of the three vertices in the formula above, we can determine if the vertices run clockwise or counter-clockwise on the screen. All we need really is a `true` or `false`, and so we return just that: `true` if this side is turned away from the viewer, or `false` (i.e. *not* a backface) if the side is turned towards the viewer.

2. To call this function, we add just these few new lines to our growing render function:

```
depthArray.sort(function (a, b) { return a[1]>b[1];});
for (var i = 0; i<depthArray.length; i++) {
  var sideVerts = depthArray[i][0];
  if (!backface([verts2D[sideVerts[0]].x,
verts2D[sideVerts[1]].x,
↳verts2D[sideVerts[2]].x], [verts2D[sideVerts[0]].y,
↳verts2D[sideVerts[1]].y, verts2D[sideVerts[2]].y])) {
  center.moveTo(verts2D[sideVerts[0]].x,
↳verts2D[sideVerts[0]].y);
  center.beginFill(0x666666, 100);
  for (var j = 1; j<sideVerts.length; j++) {
    center.lineTo(verts2D[sideVerts[j]].x,
↳verts2D[sideVerts[j]].y);
  }
  center.lineTo(verts2D[sideVerts[0]].x,
↳verts2D[sideVerts[0]].y);
  center.endFill();
}
}
};
```

## 10 Flash MX Studio

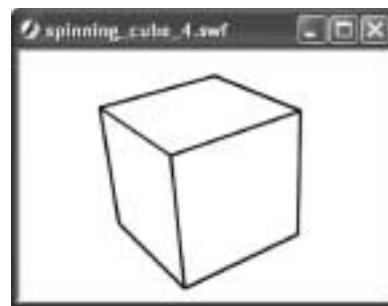
Now that's a long function call! Really, all we're doing here is nesting our line and fill drawing methods inside a conditional statement that ensures the side is not a backface. If we didn't run through each vertex here in the function call, we would have to do it in the `backface` function. I think it's a little cleaner this way. We put the first three vertices' x and y coordinates into separate arrays when we call our `backface` function (look back to that function to see how we access these arrays after they are sent). If the `backface` function determines that we do *not* have a backface (by returning `false`), we draw the side.

3. Test the movie. With the z sorting already taken care of, you might be wondering why it doesn't appear to be different:



Our spinning cube with backface culling utilized. This prevents the sides that are turned away from the viewer from being needlessly rendered.

4. However, if you go back into the code and change the fill alpha to 0 by updating the `beginFill()` call to `center.beginFill(0x666666, 0);`, you'll see that the backface culling has indeed been implemented when you test the movie again:



5. Save your movie and leave it open. We're going to be updating it in the next exercise.

## Lighting polygons

So far we've got a pretty nifty effect of a solid cube spinning on our stage, but we're missing one important factor necessary for a true 3D illusion: dynamic lighting of our model. To create a more realistic environment we need to provide a light source that will shine on our model and shade the sides based on their angle relative to the light source.

This is where some more advanced math comes into play, and so I'll preface this section by reassuring those who might feel daunted by the coming numbers. Before experimenting in Flash, my post-high school math consisted of doing my taxes and calculating a tip at a restaurant. The formulas we'll be using here are the result of many a night spent at math sites and the kind help of *ahab*, a moderator on the forums at [www.were-here.com](http://www.were-here.com). Some good math-minded friends of mine, Jeff Baldwin and Julie Bellanger, also aided me when the concepts went over my head. My point in saying this, other than to give credit and thanks where due, is to encourage you to use any and all resources at hand to help you, including any books that give you the formulas outright (like this one's about to do). The fantastic Flash work will come by combining these resources with your creativity, not by learning math alone.

### Lighting the cube

All right. The first thing we need to add is a light source. We could do this with more variables, but let's make it more object-oriented by making the light its own entity (we'll apply this same concept to the model next chapter).

1. Save your movie as `spinning_cube_5.fla`. Right at the start of your code, immediately before anything else, type:

```
LightSource = function(x, y, z, brightness) {
    this.x = x;
    this.y = y;
    this.z = z;
    his.brightness = brightness;
    this.calcMag = function() {
        this.magnitude =
            ↪Math.sqrt(this.x*this.x+this.y*this.y+this.z*this.z);
    };
    this.calcMag();
};
```

`LightSource` is a constructor function that creates a new light source for our world. As it is, its only properties are position and brightness, which are set on the object's creation (eventually, you'd want to set up methods to change these properties). The method `calcMag` calculates the magnitude of our light (the distance of the light from world center).

We'll need this value later, and it would also need to be recalculated if the *x*, *y*, or *z* properties were changed.

2. To create an instance of a `LightSource`, place the following code after the line `focalLength = 400;` (this is just after the section where we create the **center** movie clip near the start of our code):

```
light = new LightSource(-20000, -20000, 20000, 100);
```

So `light` is now the `LightSource` we use in the scene, and if we need to access any of its properties, we would do so with `light.property`. We've set its position as -20,000 on its *x*- and *y*-axes, and 20,000 on its *z*-axis. This will place it at the front, top, and left of our model, a nice traditional position for lighting. We also set its brightness to 100. Potential improvements on our original `LightSource` object could involve additional methods to alter the brightness or perhaps add color to our light.

We now need a variable in our model that will hold its color. We could easily set this for the whole model, but it might be more beneficial to provide a way to color individual sides or polygons. Let's improve our `side` property of the model to hold objects containing each side's information, as opposed to just references to the vertex numbers.

3. Alter the six lines that push vertex numbers into our `side` array to read:

```
cube.side = [];
cube.side.push({vertices:[0,1,2,3], sideColor:0x6600CC});
cube.side.push({vertices:[2,1,5,6], sideColor:0x6600CC});
cube.side.push({vertices:[1,0,4,5], sideColor:0x6600CC});
cube.side.push({vertices:[5,4,7,6], sideColor:0x6600CC});
cube.side.push({vertices:[0,3,7,4], sideColor:0x6600CC});
cube.side.push({vertices:[3,2,6,7], sideColor:0x6600CC});
```

Instead of merely inserting numbers into this property, we place objects containing two properties themselves: `vertices` to hold the vertex numbers and `sideColor` to hold the hex value of the side's color.

That takes care of storing our colors. We will have to change a few lines in our `render` function to allow for this new structure, but we'll actually house all the code to determine the light's effect on the color in two other separate functions.

4. Make the following changes to the second half of our `render` function:

```
center.clear();
center.lineStyle(2, 0, 100);
verts2D = [];
depthArray = [];
```

```

for (var i = 0; i<model.side.length; i++) {
  var zDepth = 0;
  for (var j = 0; j<model.side[i].vertices.length; j++) {
    var whichVert = model.side[i].vertices[j];
    if (verts2D[whichVert] == undefined) {
      verts2D[whichVert] = {};
      var scale = focalLength/(focalLength-
        ↪model.vertexList[whichVert].z);
      verts2D[whichVert].x =
        ↪model.vertexList[whichVert].x*scale;
      verts2D[whichVert].y =
        ↪model.vertexList[whichVert].y*scale;
    }
    zDepth += model.vertexList[whichVert].z;
  }
  depthArray.push([model.side[i], zDepth]);
}
depthArray.sort(function (a, b) { return a[1]>b[1];});
for (var i = 0; i<depthArray.length; i++) {
  var sideVerts = depthArray[i][0].vertices;
  if (!backface([verts2D[sideVerts[0]].x,
    ↪verts2D[sideVerts[1]].x,verts2D[sideVerts[2]].x],
    ↪[verts2D[sideVerts[0]].y,
    ↪verts2D[sideVerts[1]].y, verts2D[sideVerts[2]].y])) {
    center.moveTo(verts2D[sideVerts[0]].x,
      ↪verts2D[sideVerts[0]].y);
    center.beginFill(getSideColor(model,depthArray[i][0])
      ↪,100);
    for (var j = 1; j<sideVerts.length; j++) {
      center.lineTo(verts2D[sideVerts[j]].x,
        ↪verts2D[sideVerts[j]].y);
    }
    center.lineTo(verts2D[sideVerts[0]].x,
      ↪verts2D[sideVerts[0]].y);
    center.endFill();
  }
}
};

```

There's really not too much change here in comparison to what it allows us to do. You can see that we now have to access the sides' `vertices` property since that is where the vertex numbers are stored. Also, instead of providing a fill color in our `beginFill` method, we will call a function instead, sending it a reference to our current side (stored in `depthArray[i][0]`) and a reference to the model itself. We know that this will return a color, as that is the required first parameter for `beginFill`. Let's write this function now.

5. Add this new function, directly below the `render` function:

```
getSideColor = function (model, side) {
    var col = side.sideColor.toString(16);
    while (col.length < 6) {
        col = "0" + col
    }
    var verts = [model.vertexList[side.vertices[0]],
        ↪model.vertexList[side.vertices[1]],
        ↪model.vertexList[side.vertices[2]]];
    var lightFactor = factorLightAngle(verts);
    var r = parseInt(col.substr(0, 2), 16)*lightFactor;
    var g = parseInt(col.substr(2, 2), 16)*lightFactor;
    var b = parseInt(col.substr(4, 2), 16)*lightFactor;
    return r << 16 | g << 8 | b;
};
```

`getSideColor` is nearly identical to our `convertToRGB` function from the last chapter (a more explicit explanation was given there). Most of the function deals with separating our color into its red, green, and blue channels so that we can affect each value individually. The only different lines are those that put our side's first three vertices into a temporary variable array and then send that array to a function called `factorLightAngle`. By sending the vertices directly (remember, the objects are stored in our model's `vertexList`), it will be easier for `factorLightAngle` to deal with the numbers.

There's just one more function to go, the `factorLightAngle` function, but this is where all the power lies. It's also where the heavy math is waiting, so take a deep breath, and let's dive in!

6. Add these lines immediately beneath the last function:

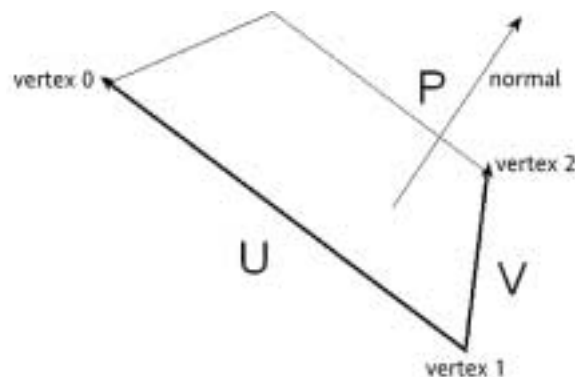
```
factorLightAngle = function (vertices) {
    var U = [(vertices[0].x-vertices[1].x), (vertices[0].y
        ↪vertices[1].y), (vertices[0].z-vertices[1].z)];
    var V = [(vertices[1].x-vertices[2].x), (vertices[1].y-
        ↪vertices[2].y), (vertices[1].z-vertices[2].z)];
    var p = [((U[1]*V[2])-(U[2]*V[1])), -((U[0]*V[2])-(U[2]*V[0])),
        ↪((U[0]*V[1])-(U[1]*V[0]))];
    var magP = Math.sqrt((p[0]*p[0])+(p[1]*p[1])+(p[2]*p[2]));
    var dP = ((p[0]*light.x)+(p[1]*light.y)+(p[2]*light.z));
    return ((Math.acos(dP/(magP*light.magnitude))/Math.PI)
        ↪*light.brightness/100);
};
```

7. We'll break down what's happening here in just a moment as I'm sure you want to test out all your hard work, so go on and test the movie to see the light shading take effect as your cube spins:



Dynamic lighting is applied to our spinning cube model, helping to better simulate depth.

Right, let's step back and examine that final section of code.  $\mathbf{u}$  and  $\mathbf{v}$  are simply 3D vectors we derive from our vertices (remember, a vector here represents a position and direction in space). You probably do similar things all the time for 2D vectors in your Flash movies, when you subtract one movie clip's `_x` property from the `_x` property of another clip (the only difference here being we have an extra dimension to calculate). Once we know  $\mathbf{u}$  and  $\mathbf{v}$ , we find the cross product of these two vectors, which is a vector perpendicular to the two vectors used in the operation. In 3D, this vector that is perpendicular to a surface's face is known as the polygon's **normal**. This normal tells us the orientation of the face of the polygon.



With this vector defined, we can find its relation to our light source and determine how much light it should receive. The formula we use to accomplish this is:

$$\cos(\angle) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}$$

*NOTE: U and V are here used to signify two vectors and do not relate to the U and V variables used in our above formula.*

So what does this mean then? Well, to find the cosine of the angle between our light source and our polygon's normal, we multiply the two vectors together (the dot product, which is the measure of the distance between the two vectors) and then divide this by the product of the length of the two vectors (their magnitudes, or distance from world center).

Another interpretation of this is that some extremely clever and diligent mathematicians worked very hard on this formula, and so who am I to dispute it? I can use it to find the angle at which my light source is hitting my polygon, and that's all that matters to me.

*Before we get to the code that utilizes this formula, it's important to understand the nature of the light we have placed in our 3D scene. In 3D software, our light would be akin to a directional light always set to point at world center. This saves us having to orient our light to point at our models when we first place it or if we translate it during an animation. However, because of this, if the light is translated to the world center, it cannot be directed out towards any models so the models would be black and unlit (black because we have no ambient light set in our scene, an adjustment you could make later).*

Finally, these are the last three lines of our function. `magP` is the magnitude of our normal, which is basically the polygon's distance from world center (just Pythagoras at work here). `dP` holds the result of the dot product operation, which you can see is just some multiplication and addition of the two vectors' components. Finally we determine the angle by finding the arccosine of our dot product divided by the product of the two vectors' magnitudes. This will give us a value in radians between 0 and pi, so we divide by pi to give us a percentage. We use this to find a percentage of our light's brightness (we divide by 100 again because of the way we set up the brightness value of the light – I wanted to use integers to express this, so we need to divide by one hundred here to return a percentage with this function).

Wow! All that to light our model! You'll be glad to know that we're finished though, so you can take a well-earned break. If however you want to go back over any elements of this exercise (or any of the earlier versions of the spinning cube), all of the source FLAs with fully commented code are on the CD to compare your files against.

In `spinning_cube_final.fla` I've provided text boxes so you may alter the cube's transforms and the light's properties at run-time. I've also added a `translate` function so that your models can be moved from center. Ideally, you'd not incorporate this in the way that I have, (which I've done for brevity's sake). One alternative method is to use 4x4 matrices for our transforms and add a fourth dimension for each vertex. We explore this in the following chapter. Obviously, there is a lot of work that can be done to improve the usefulness and modularity of our code, and this is one of the improvements I would suggest if you wish to develop this further.

## Summary

In this chapter we've looked at techniques for rendering 3D elements using the new drawing API. Moving past the wireframes and vertex models of Flash 5, we can now create and manipulate solid shapes that react dynamically to light using ActionScript alone. This opens the door to so many possibilities, but it will take your creativity and ingenuity to utilize these new capabilities in an interesting and useful manner.

In the next chapter we're going to look at ways to incorporate these 3D and drawing methods into Flash interface and game design, but of course this is the tip of the iceberg. Hopefully, the 3D seeds have been planted (and the formulas will be printed!) and you can take these concepts and run with them to places where only your own individual mind can go. I can't wait to see pictures of your trip!

