2: 3D and Flash

Michael Brandon Williams is a senior at Spring Woods High School in Houston, Texas, with many years of mathematics and computer science study in his curriculum vitae. His mathematics focus has been single and multivariable calculus, real analysis, linear algebra, ordinary differential equations, elementary combinatorics, and number theory. His computer science experience is based on programming design, object-oriented programming, and problem solving. His goal is to pursue a Ph.D. in Mathematics.

In his spare time, he helps run the math forum at Were-Here (www.were-here.com) under the name of ahab, and works for Eyeland Studios (www.eyeland.com) as a games programmer.

Danish freelance designer **Torben Nielsen** runs his own design company, SD Flash Studios, which specializes in Flash solutions for the Internet and other medias. He is based in Rome.

Throughout his childhood, Lego, Walt Disney cartoons, and his Comodore 64 computer sparked his interest in drawing and creativity. His popular comic strip for his high school newspaper reflected school life in a sarcastic way. Torben has never had formal design education, apart from a two-year art class in college.

When he discovered Flash 3 in 1998, that old Lego feeling came back to him, and it was love at first sight. Since then he has specialized in Flash; today it's the base of all of his projects.

More and more Flash developers are catching on to the power of 3D. Adding 3D effects to Flash movies can help spice up a navigation system, make impressive eye candy, or simply entertain viewers. In fact, the use of 3D in Flash has increased over the last few years: The trend began at front-running Web sites such as www.yugop.com and www.mano1.com, and has just kept growing.

But Flash is a 2D software program—it doesn't support 3D models. How, then, can you create 3D animation in Flash? Easy: You fake it.

There are two ways to create the illusion of 3D shape and motion: ActionScript 3D, in which you code the entire project, and Rendered 3D, in which you use third-party rendering software, Flash, and some ActionScript. This chapter will discuss these two very different methods.

ActionScript 3D

The first half of this chapter will focus on the concepts used to create 3D effects via ActionScript. All equations will be derived in the text, but we'll try to leave out as much mathematical rigor as possible. After we cover the core information, we'll look at a number of demos that show these concepts in action.

3D Basics

Let's start by reviewing and expanding on some concepts you already know. I'll talk about how points are represented in 3D space, show you a new type of 3D coordinate system, and describe how this system works in Flash.

Mathematical Coordinate System

Almost everyone is familiar with the 2D Cartesian coordinate system. The origin is a fixed point at the center of the graph, and the *x*- and *y*-axes go through the origin, perpendicular to each other. The *x*-axis runs from left (negative) to right (positive), and the *y*-axis runs from bottom (negative) to top (positive) (**Figure 2.1**).

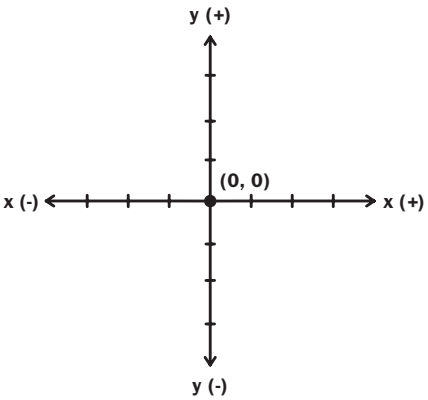

Figure 2.1 *The 2D Cartesian coordinate system.*

To represent a point in 3D space, we once again define the origin as a fixed point at the center, and choose three axes that go through the origin and are perpendicular to each other. These three axes are the *x*-axis, *y*-axis, and *z*-axis. Any two axes comprise what's called a coordinate plane. **Figure 2.2** shows the 3D coordinate system.

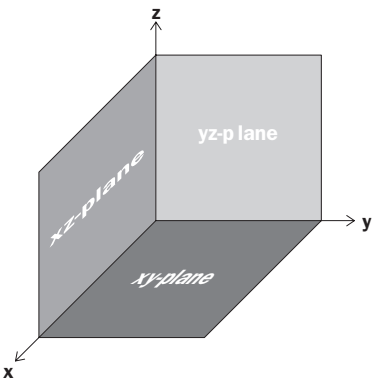

Figure 2.2 *The 3D coordinate system.*

Ordered Triplets

Students of analytical geometry and algebra plot graphs all the time. An ordered pair gives the position of each point on a graph. It contains separate values for a point's *x*-position (the *abscissa*) and *y*-position (the *ordinate*).

You use an ordered triplet to locate a point in 3D space. An ordered triplet has three separate values for the point's position along the *x*-, *y*-, and *z*-axis. Using the 3D coordinate system, we can plot a general point (**Figure 2.3**).

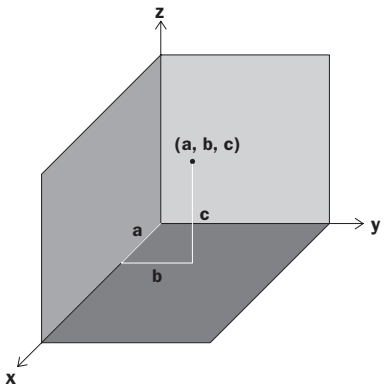

Figure 2.3 *An ordered triplet describes the position of a point in 3D space.*

An ordered triplet of real numbers (a, b, c) tells you to go *a* units along the *x*-axis, *b* units along the *y*-axis, *c* units along the *z*-axis, and plot a point.

Flash 3D Coordinate System

The 3D coordinate system shown so far is the standard in nearly all fields of mathematics. You'll notice, however, that it differs greatly from the 2D Cartesian system. The *x*-axis protrudes outwards, the *y*-axis goes from left to right, and the *z*-axis sticks up perpendicular to the *x*- and *y*-axis. In other words, the *x*- and *y*-axis have been switched around. This will only cause confusion when doing our calculations, so in Flash, we use a different coordinate system. The *x*-axis goes from left to right as usual, and the *y*-axis goes up and down as usual, but the *z*-axis protrudes from the screen (**Figure 2.4**).

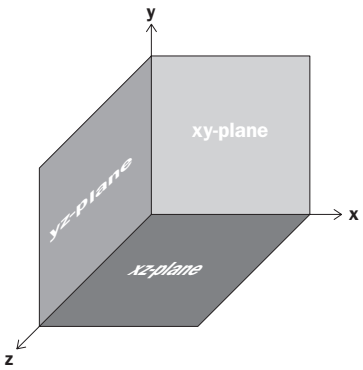

Figure 2.4 *3D coordinate system as it appears in Flash.*

In Flash, the *z*-axis represents how near or far an object is from the screen. When an object's *z*-position increases, the object moves toward the back of the screen; when an object's *z*-position decreases, the object comes closer (**Figure 2.5**).

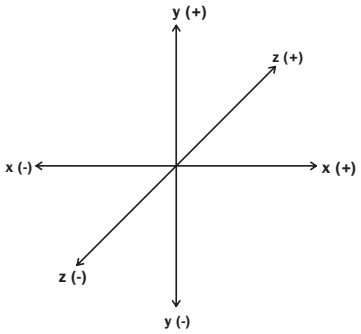

Figure 2.5 *How the three coordinate axes are oriented in Flash.*

Rendering 3D

Once you plot a few points in space, you'll want to render them, or show the points on the Stage. But how do you plot a 3D point on a 2D screen, and how do you size a point correctly so it looks as if it's in a true 3D environment? We can solve both of these problems with a little derivation.

Adding Perspective

Perspective helps you differentiate between 3D and 2D. Perspective is that omnipresent bend of the world that positions things in relation to their distance from you. If you stand in the middle of a road, looking down it as far as possible, you'll see that the road's edges begin to come together. Depending on how far you can see, the edges will appear to move infinitely closer to each other until they converge. Perspective also affects how movement is perceived. For example, even if objects in the foreground and background are moving at the same speed, those in the foreground will appear to move more quickly.

Changing ordered triplets into ordered pairs. The perspective problem boils down to something simple. You have an ordered triplet of a point, (x, y, z), and you want to find where that point would be placed on your screen if it didn't have a *z*-position. Because your computer screen doesn't have a *z*-axis, you must find a way to change the ordered triplet into an ordered pair while taking into consideration the point's *z*-position. The easy

way is to merely drop the *z* value, but if you do that, the road's edges will never converge. Instead, they will remain parallel as the road goes off into the distance.

We can solve this problem using simple geometry. But before we discuss the way to project a 3D point onto your 2D computer screen, there are a few quantities you should know.

Figure 2.6 shows a point in space positioned at (x, y, z). Its position when projected onto the computer screen is (x', y'). I used the apostrophe only so you can tell the difference between the point's *x*- and *y*-position in space and its *x*- and *y*-position on the screen. Because the image is two-dimensional, I had to leave out one dimension of the point; in this case, I left out *x*. The quantity labeled *d* is the distance from the viewer's eye to the computer screen. This value will be a key component of how your 3D environments look. The *z*-position of the point in space is measured from the computer screen because that's where the origin is. We can now say that the *z*-position of the viewer is at negative *d*, because the *z*-axis decreases as it goes toward the computer screen.

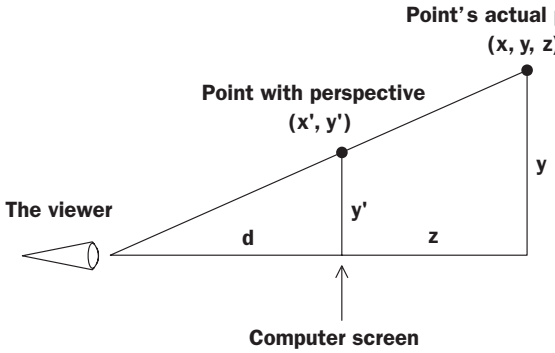

Figure 2.6 *A 3D point must be projected onto a 2D screen to be used in Flash.*

Now, let's use this figure to solve the perspective problem. We have to use the given information of *d*, *y*, and *z* to find *y'*. By looking at the figure we can see it contains two similar triangles. From simple laws of geometry, we can relate the sides of both triangles with the following proportion (**Figure 2.7**):

$$\frac{d}{y'} = \frac{d + z}{y}$$

Figure 2.7 *The sides of two similar triangles are proportional.*

Let's apply some elementary algebra to solve the equation for *y'*. We do this by multiplying both sides by *y* and *y'*, and dividing both sides by the sum of *d* and *z* (**Figure 2.8**):

$$\frac{d}{y'} = \frac{d + z}{y}$$

$$\Downarrow$$

$$d \bullet y = y' (d + z)$$

$$\Downarrow$$

$$y' = \frac{d \bullet y}{d + z}$$

Figure 2.8 *By solving the proportion for the y-position of an ordered triplet, we turned it into an ordered pair.*

We've just derived a crucial equation for perspective.

While the previous equation gives you the *y*-position of the point when it's projected onto the screen, we can use a similar method to find the projected point's *x*-position. We can now write our nearly finished equations for changing an ordered triplet into an ordered pair while taking into consideration the point's *z*-position (**Figure 2.9**):

$$x' = \frac{x \bullet d}{z + d}$$

$$y' = \frac{y \bullet d}{z + d}$$

Figure 2.9 *Equations for turning an ordered triplet into an ordered pair.*

When you work with equations in mathematics, you want them to be as neat and simplified as possible. If you have the expression *x^2 + x*, for example, you could simplify it to *x(x + 1)* by factoring out an *x*. Likewise, with perspective equations you can factor out *d/(d+z)* and set it as a separate variable before the perspective ordered pair is calculated. This variable will play an important role in sizing movies clips and will save you from doing extra math operations.

Now we can write the final form of the perspective equations (**Figure 2.10**):

$$perspective\_ratio = \frac{d}{z + d}$$

$$x' = x \bullet perspective\_ratio$$

$$y' = y \bullet perspective\_ratio$$

Figure 2.10 *Factoring out common terms.*

Let's apply our new knowledge to a script that will change an ordered triplet into an ordered pair.

You can choose any three numbers for the ordered triplet. Choosing a value for *d*, the distance from the viewer's eye to the screen, takes a little more thought. To use the example of the road again, if you choose too small a value, your points will take on a fish-eye look and the road's edges will converge quickly. If you choose too large a value, it may be hard to distinguish between close and far points and the road's edges will converge slowly. As a general rule, choose values between 200 and 500, although you'll have to experiment to better understand the different outcomes.

The following script is placed in the clip events of a movie clip. (The movie clip can be anything that you want to duplicate many times and place in space randomly.) The script will give the movie clip a random ordered triplet between –100 and 100, and then calculate the point's position as it appears on screen.

```
OnClipEvent (load)
{
// random ordered triplet
x = Math.random () * 200 – 100;
y = Math.random () * 200 – 100;
z = Math.random () * 200 – 100;
// used for perspective – distance from the viewer to the
screen
D = 400;
// calculate the perspective ratio
perspective_ratio = D / (D + z);
// calculate position of point on computer screen
perspective_x = x * perspective_ratio;
perspective_y = y * perspective_ratio;
}
```

Changing position and size on the stage. Although our script calculates the position of a 3D point on a 2D screen, we're still unable to render the point (*render* is used in the context of positioning and sizing the movie clip that will represent the point).

Positioning the point is the easiest part. If you have a point at (x, y, z) in space, and you transform it into a point at (x', y') on your screen, you still don't know where it is on Flash's Stage. To do this, you must first define the position of the origin, because Flash's origin (0, 0) is at the top-left hand

corner. The origin is the point that everything moves around; it's one of the most important variables that you can set. After you've found *x*' and *y*', you translate it along the *x*- and *y*-axis according to the position of the origin.

The following sample script positions a movie clip on the Stage given the position of the origin and the position of the point with perspective. The perspective point is chosen at random for now.

```
OnClipEvent (load)
{
// position of origin
origin_x = 275;
origin_y = 200;
// position of point with perspective
perspective_x = Math.random () * 200 – 100;
perspective_y = Math.random () * 200 – 100
/* Position movie clip on the stage – the minus sign
comes up when doing the y-position because Flash's
y-axis is flipped upside down. */
this._x = origin_x + perspective_x;
this._y = origin_y – perspective_y;
}
```

Once the movie clip is positioned on the Stage, the only other property to worry about is the size. You can size the movie clip perfectly by looking at the numerical values of the perspective ratio for different *z*-positions. When *z* becomes incredibly huge (the point goes far away), the denominator of the perspective gets very large, too. When you divide by a large number you get a very small number. So, in general, as *z* increases, the perspective ratio decreases. Conversely, when *z* decreases, the perspective ratio increases.

This is the exact behavior we want the size of a point to have. When an object moves away from you (*z* increasing), its size decreases. When it moves toward you (*z* decreasing), its size increases. The ratio doesn't fully describe this relationship, however: All objects would be sized the same because the perspective ratio doesn't take into consideration the size of the movie clip. By multiplying the perspective ratio by the size of the movie clip, you'll get the perspective size to render, as shown below. In this equation, `regular_size` is the size of the movie clip at the origin and `perspective_size` is the size of the movie clip as it appears with perspective.

```
Perspective_size = regular_size * perspective_ratio;
```

Going back to our first script, which picked a random ordered triplet and changed it into an ordered pair, we can render the point, too. Again, this script is placed in the clip events of any movie clip. You also need to initialize constants like the movie clip's size at the origin, the distance from the viewer to the screen, and the position of the origin.

```
onClipEvent (load)
{
// position of origin
origin_x = 275;
origin_y = 200;
// used for perspective - distance from the viewer to
→ the screen
D = 400;
// size of movie clip at the origin
regular_size = this._width;
// random ordered triplet
x = Math.random () * 200 - 100;
y = Math.random () * 200 - 100;
z = Math.random () * 200 - 100;
// calculate the perspective ratio
perspective_ratio = D / (D + z);
// calculate position of point on computer screen
perspective_x = x * perspective_ratio;
perspective_y = y * perspective_ratio;
// update position of movie clip on stage
this._x = origin_x + perspective_x;
this._y = origin_y - perspective_y;
// update size of movie clip
this._xscale = this._yscale = regular_size *
→ perspective_ratio;
}
```

Movie clip alpha. Objects in the distance often appear to be faded. You can achieve this effect in Flash by changing the `_alpha` of the movie clip that represents your point. Calculating the alpha is similar to calculating the point's size. When an object is at the origin, or between the origin and the viewer, it has an alpha of 100. As the object travels into the distance, its alpha becomes smaller, nearing 0. This is the same behavior of the perspective ratio value and the size of the point. So, by multiplying the perspective

ratio by 100 (the alpha value at the origin), you can change a movie clip's alpha depending on where it is along the *z*-axis.

```
// calculate alpha according to z-position
movie_clip. alpha = 100 * perspective_ratio;
```

Although the alpha is easy to implement, we won't use it in most of the upcoming demos.

Randomly positioned points in space. Now it's time for our first demo. This file will duplicate a number of points, place them randomly in 3D space, and render them.

Open random_3D_points.swf from the chapter 2 folder on the CD-ROM.

Try opening the file a few times to see how it creates a random field of points each time (**Figure 2.11**).

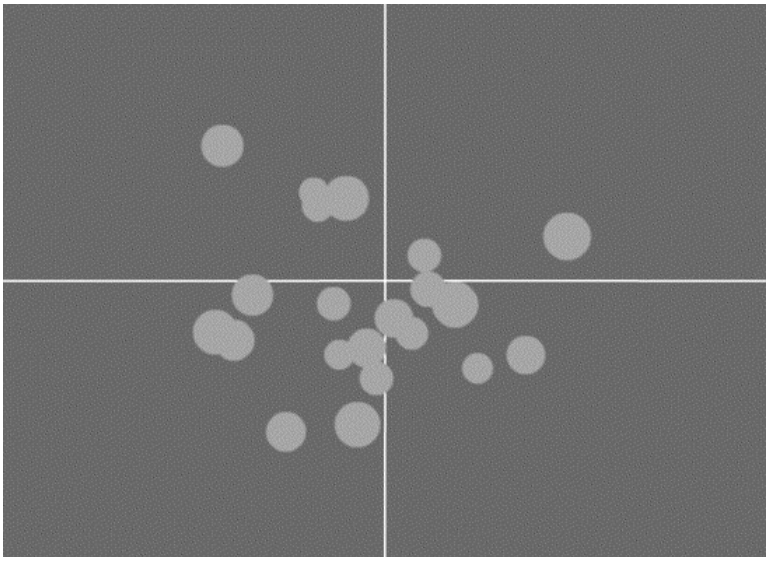

Figure 2.11 *A Flash demo that creates a field of randomly placed points in 3D space.*

Open random_3D_points.fla from the CD-ROM.

To create this file, I drew a circle, made it a movie clip, and placed the script we wrote for placing a point in space and rendering it. When this clip loads, it picks a random ordered triplet and renders itself due to perspective. All we have to do is duplicate a few of the movie clips so they can take care of their own business. These actions go in the first frame of the `_root` and should be executed only once.

The following code duplicates 20 movie clips. Once the movie clips are duplicated, the clip event load actions takes care of their position and size. The instance name of the movie clip to duplicate is called point.

```
// number of points to be placed out randomly
num_points = 20;
// duplicate points - the script inside the
// points will take care of the rest
for (var j = 0; j < num_points; j++)
{
this.point.duplicateMovieClip ("point" + j, j);
}
```

Z-Ordering

The previous demo has one small flaw in its rendering engine: making foreground objects appear over background objects. Because I used solid color circles in the demo file, it was nearly impossible to see that the movie clips' depths were ordered incorrectly; using a gradient would have created an eyesore, however. The solution is incredibly simple and effective, as well as easy on the CPU.

point.swapDepths (depth)

This built-in method of the movieClip object lets you easily set the movie clips' depth based on their *z*-positions. If you simply set the depth of a movie clip to its *z*-position, things still won't be correct: A movie clip with a large *z*-position will be placed on top, but a point with a large *z*-position should be in the background. Therefore, by inverting the point's *z*-position and setting the movie clip's depth to that value, you'll get the correct effect.

```
// set the depth of the clip based on its z-position
this.swapDepths (-z);
```

Now you can change the points to any type of graphic that you want and they'll overlap correctly, giving the file more depth.

Some people feel uncomfortable negating the *z*-position to find the clip's depth because it's possible to set it to a negative number. Although I have yet to run into this problem, you can simply subtract the *z*-position from a very large number to keep the depths sorted and ensure positive numbers.

Z-Sorted Random Points

Let's use this information to make a new file that will duplicate a few points, place them randomly in space, and then render them and sort their depths.

Open random_3D_points_depth.swf from the CD-ROM.

Open this file a few times to see how the larger points are placed above the smaller ones. It's much easier to distinguish between distance and close objects (**Figure 2.12**).

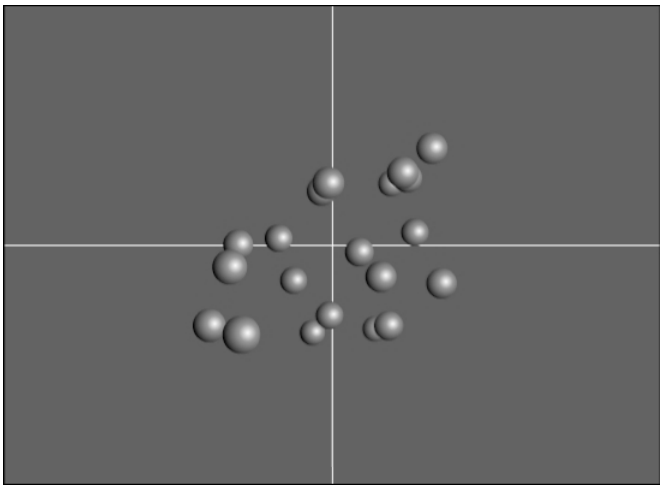

Figure 2.12 *A Flash demo that creates a field of randomly placed points in 3D space and sorts their depths.*

Open random_3D_points_depth.fla from the CD-ROM.

This movie is the same as random_3D_points.fla except for the graphic of the point and two lines of code. This code is placed in the clip events of the point movie clip. After all the other actions have been carried out, these two lines will keep the points at correct depths:

```
// set the depth of the clip based on its z-position
this.swapDepths (-z);
```

We've now nearly developed a full rendering engine.

Translation

To translate a point, you increment its *x*-, *y*-, or *z*-position by a certain value. For example, a point at (1, 3, 2) could be translated along the *x*-axis by 4 to (5, 3, 2), then along the *y*-axis by 3 to (5, 6, 2), and finally along the *z*-axis by - 4 to (5, 6, -2).

Random Movement

In Flash, your translations will be more on the fly. To create fluid movement, you must increment a point's position every frame. This means we need to change the layout of our previous script, which only placed and rendered points. The next file duplicates and places points in the same fashion as before, but this time the points move around in space in random directions.

Open random_3D_points_trans1.swf from the CD-ROM.

When you open this file, the points move away in a linear fashion. Every time you open the file, the points scatter in a different direction (**Figure 2.13**).

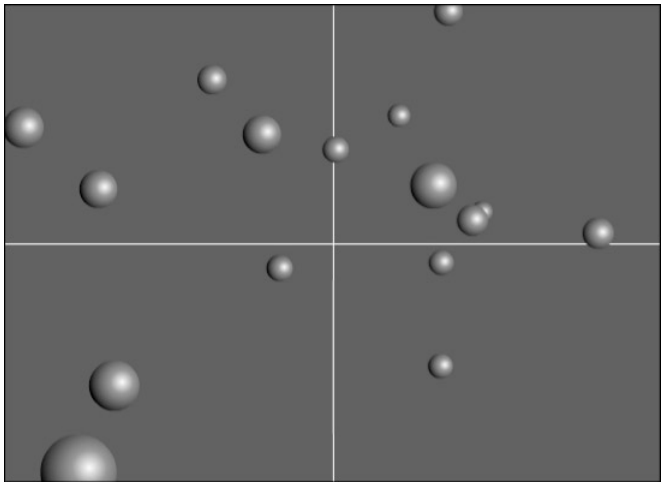

Figure 2.13 *The randomly placed points travel in different directions.*

Open random_3D_points_trans1.fla from the CD-ROM.

Instead of handling everything in the load clip event, this time we'll set only the position, constants, and random values for the increments at which the point will be translated. You can change the increments later on by using buttons, clip events, or any other interactive method. For now, I used small translation increments so you could see the gradual change in position.

```
onClipEvent (load)
{
// position of origin
origin_x = 275;
origin_y = 200;
// used for perspective - distance from the viewer to
→ the screen
D = 400;
```

```
// size of movie clip at the origin
regular_size = this._width;
// random ordered triplet
x = Math.random () * 200 - 100;
y = Math.random () * 200 - 100;
z = Math.random () * 200 - 100;
// increment to move along the x-, y-, and z-axes
inc_x = Math.random () * 2 - 1;
inc_y = Math.random () * 2 - 1;
inc_z = Math.random () * 2 - 1;
}
```

All of the rendering code has been taken out and three new variables have been added. The new variables are added to the point's position, moving it in a random direction. Although this is not the most useful effect, it helps us understand the movement.

The enterFrame clip event constantly translates and renders the point. The three random increment variables that were defined on `load` are added to the point's ordered triplet and, finally, the point is rendered using the same script from before.

```
onClipEvent (enterFrame)
{
// move the point according to the translate increment
x += inc_x;
y += inc_y;
z += inc_z;
// calculate the perspective ratio
perspective_ratio = D / (D + z);
// calculate position of point on computer screen
perspective_x = x * perspective_ratio;
perspective_y = y * perspective_ratio;
// update position of movie clip on stage
this._x = origin_x + perspective_x;
this._y = origin_y - perspective_y;
// update size of movie clip this._xscale = this._yscale =
→ regular_size * perspective_ratio;
// set the depth of the clip based on its z-position
this.swapDepths (-z);
}
```

Believe it or not, you've just written the script to another Flash demo! Duplicate a couple of the movie clips with these actions and you'll see a field of points moving around in random directions.

Controlled Movement

Having dots fly around you in random directions is probably not what you had in mind when you started this chapter. We can use simple translations, however, to immerse viewers into 3D by giving them control over the translations. The layout of this next script is crucial because of its interactive components, so that's what we'll focus on.

A handy trick for key presses. There's a handy trick to help keep track of which keys are pressed down during a frame loop without adding another clip event; it's mainly useful when variables need to be changed while the key is pressed. The `isDown()` method of the Key object returns a 1 when the key passed to it is down and a 0 when it's not. If you multiply another variable by the return of the `isDown()` function, you get the variable when the key is pressed and a 0 when it's not. This concept can be used in the following way:

```
x += Key.isDown (Key.someKey) * inc;
```

The variable x is incremented by `inc` every frame as long as `someKey` is held down. You can combine this idea with a variable decrement, too. At first, you might be inclined to write a whole new line for the decrement:

```
x -= Key.isDown (Key.someKey) * inc;
```

But you can write it all in one line, too. The line will constantly add some expression to the value of *x*. The expression will be the difference of the increment key multiplied by the increment value and the decrement key multiplied by the increment value.

```
x += Key.isDown (Key.inc_key) * inc – Key.isDown
→ (Key.dec_key) * inc;
```

When the `inc_key` is held down, *x* will increment by `inc` every frame. When the `dec_key` is pressed, *x* will decrement by inc every frame. If both keys are held down, nothing is added to *x* because the two terms cancel out.

Let's see this trick in action. The following script moves the movie clip around, depending on which arrow keys you press. Create a new Flash

movie, draw a simple graphic, and make the graphic a movie clip. Place the following actions in the load and enterFrame clip events of the movie clip:

```
onClipEvent (load)
{
// increment clip will move by when a key is down
inc = 4;
}
```

The load clip event initializes a variable that holds the value of how much you want the movie clip to move when a key is pressed. The `enterFrame` actions add the increments and decrements to the current *x*- and *y*-position of the movie clip; this uses the script we previously developed.

Before you can do this, though, you must think about which keys will cause an increment and which will cause a decrement. Because the *x*-axis in Flash is the same as in math class, we can safely say that the Left key will cause a decrement in the *x*-position and the Right key will cause an increment. But, because the *y*-axis is flipped in Flash, we'll use the Up key for the *y*-position decrement and the Down key for the increment.

```
onClipEvent (enterFrame)
{
// movie clip according to which key is pressed
this._x += Key.isDown (Key.RIGHT) * inc - Key.isDown
→ (Key.LEFT) * inc;
this._y += Key.isDown (Key.DOWN) * inc - Key.isDown
→ (Key.UP) * inc;
}
```

Open key_move_sample.fla from the CD-ROM.

This file demonstrates what we just formulated. There are only 13 lines of code in the entire movie and it's all placed within the clip events of the movie clip. The script is identical to what we just wrote.

Using the handy trick. Let's apply what we've learned to create another 3D application. The points will be duplicated and placed in the same fashion as before, except this time you can move around them.

Open random_3D_points_trans2.swf from the CD-ROM.

Use the Arrow keys and the Shift and Enter keys to move through the field of points (**Figure 2.14**).

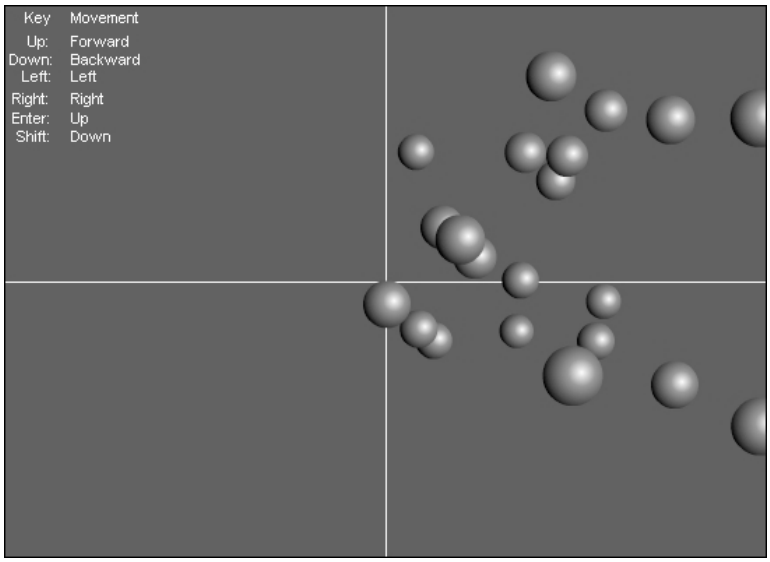

Figure 2.14 *Navigate through a field of points.*

Open random_3D_points_trans2.fla from the CD-ROM.

In this file, we'll put all scripting, except for the duplicate preliminaries, in the point movie clip. We'll replace the three random translate increments we defined earlier with one value, which will be how much the points are translated when a key is held down. Therefore, the load clip event actions have changed slightly:

```
onClipEvent (load)
{
// position of origin
origin_x = 275;
origin_y = 200;
// used for perspective - distance from the viewer to
→ the screen
D = 400;
// size of movie clip at the origin
regular_size = this._width;
// random ordered triplet
x = Math.random () * 200 - 100;
y = Math.random () * 200 - 100;
z = Math.random () * 200 - 100;
// increment the point will move by when a key is down
inc = 4;
}
```

The constants, such as the position of the origin, the size of the movie clip, and the perspective distance, are still initialized first. The script gives the point a random ordered triplet and a constant value for the rate at which the point is translated when a key is down. In the previous file, we initialized a random value for the translation increments in the load clip event. Here, the translation increment is determined by which keys are pressed; this is done in the enterFrame clip event.

The enterFrame clip event actions take care of the rendering and movement. In our previous file, the random translation increments were added to the point's ordered triplet every frame. Here, we replace the old increment actions with actions that increment and decrement variables from key presses. Because we aren't dealing with 2D movement any more, I had to change which keys changed which translation axis. In this file, the Left and Right arrow keys control the *x*-movement, the Enter and Shift keys control the *y*-movement, and the Up and Down keys control the *z*-movement. The rendering part of the script remains unchanged.

```
onClipEvent (enterFrame)
{
// move the point according to which keys are pressed
x += Key.isDown (Key.LEFT) * inc - Key.isDown (Key.RIGHT)
→ * inc;
y += Key.isDown (Key.SHIFT) * inc - Key.isDown (Key.ENTER)
→ * inc;
z += Key.isDown (Key.DOWN) * inc - Key.isDown (Key.UP)
→ * inc;
// calculate the perspective ratio
perspective_ratio = D / (D + z);
// calculate position of point on computer screen
perspective_x = x * perspective_ratio;
perspective_y = y * perspective_ratio;
// update position of movie clip on stage
this._x = origin_x + perspective_x;
this._y = origin_y - perspective_y;
// update size of movie clip
this._xscale = this._yscale = regular_size *
→ perspective_ratio;
// set the depth of the clip based on its z-position
this.swapDepths (-z);
}
```

Although the script may not look like much, we've created a truly impressive file!

A problem with the rendering engine. When you give viewers complete control over the 3D environment, you're more likely to find small bugs in the rendering engine. A problem may arise if you try to move too far into the points. If a point's ordered triplet is behind you, its position on screen (ordered pair) goes out of whack. The points will come back out in front of you and will be flipped across each axis. The solution: Set the visibility of the movie clip to false when it's behind you and true when it's in front of you.

How do you check if a point is in front of or behind you? Let's refer to the picture that helped us derive the perspective equations earlier in this chapter (**Figure 2.15**).

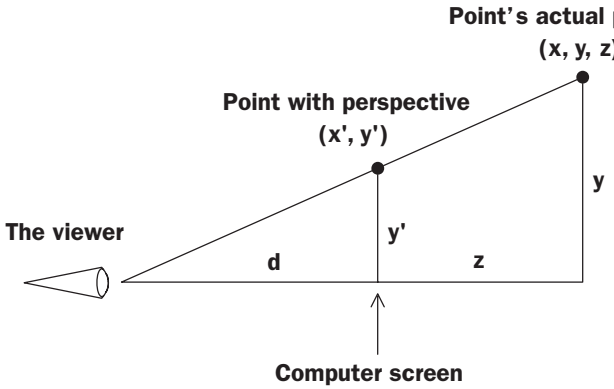

Figure 2.15 *A point is behind the viewer when it goes less than negative d.*

The origin of 3D space is placed right on the surface of your computer screen. Anything on your side of the computer screen has a negative *z*-position; anything inside the computer has a positive *z*-position. The distance from your eye to the computer screen is *d*, the perspective distance. In other words, the *z*-position of your eye is - *d*. Therefore, if any point has a *z*-position less than - *d*, you know it has traveled behind you.

Using this information, we can make a conditional statement that tests to see if a point's *z*-position has gone behind you; if it has, then it makes the movie clip invisible. Because the clip doesn't have to be rendered on the Stage at all if it's behind you, you can save Flash a few processor calculations by placing the rendering script inside the `else` of the condition statement.

The following script will be more insightful than my words; it translates and renders the point in the enterFrame actions.

```
onClipEvent (enterFrame)
{
// move the point according to which keys are pressed
x += Key.isDown (Key.LEFT) * inc - Key.isDown (Key.RIGHT)
→ * inc;
y += Key.isDown (Key.SHIFT) * inc - Key.isDown (Key.ENTER)
→ * inc;
z += Key.isDown (Key.DOWN) * inc - Key.isDown (Key.UP)
→ * inc;
// check if point is behind viewer
if (z < -D)
{
// make clip invisible
this._visible = false;
}
else
{
// show and render clip
this._visible = true;
// calculate the perspective ratio
perspective_ratio = D / (D + z);
// calculate position of point on computer screen
perspective_x = x * perspective_ratio;
perspective_y = y * perspective_ratio;
// update position of movie clip on stage
this._x = origin_x + perspective_x;
this._y = origin_y - perspective_y;
// update size of movie clip
this._xscale = this._yscale = regular_size *
→ perspective_ratio;
// set the depth of the clip based on its z-position
this.swapDepths (-z);
}
}
```

The following file is similar to the previous one, except for the conditional that removes points behind the viewer. You'll find it on the CD-ROM as random_3D_points_trans3.fla.

Rotation

Rotation is the epitome of 3D. It gives you the sense of truly being in a 3D environment because you can twist and turn inside it. Although it's by far the most complicated aspect of 3D covered in this chapter, it's easy for even math novices to understand.

Rotation Preliminaries

Before we get to the code, let's review a few underlying concepts, mainly based on trigonometry.

The two trigonometric functions that play the most important role from now on are sine and cosine. Both return ratios when given an input of an angle. To picture what type of ratio the functions return, you must create a right triangle from the angle (**Figure 2.16**).

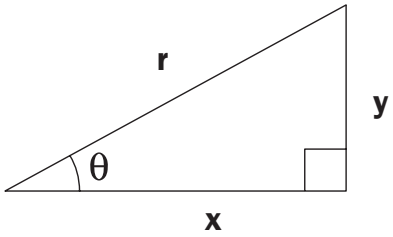

Figure 2.16 *The sine and cosine ratios are derived from a right triangle.*

The sine of the angle, θ, is the opposite side of the angle, *y*, divided by the hypotenuse, *r*. The cosine of the angle, θ, is the adjacent side to the angle, *x*, divided by the hypotenuse, *r.* These definitions are better understood with the following written mathematical equation (**Figure 2.17**):

$$\sin\theta = \frac{y}{r}$$

$$\cos\theta = \frac{x}{r}$$

Figure 2.17 *Equations for calculating the sine and cosine ratios.*

We can formulate our first applicable equations from these simple definitions. The quantities *x* and *y* represent the base and height of the right triangle made from the angle and *r*. Placing the origin at the vertex from

which we're measuring the angle, we can also interpret *x* and *y* as the position of the non-right angle vertex (**Figure 2.18**).

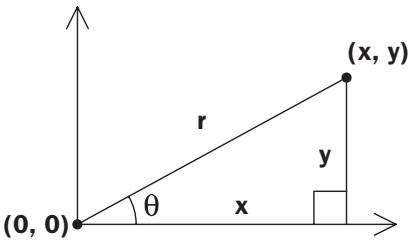

Figure 2.18 *A right triangle as points on a graph.*

Now, by rearranging the sine and cosine equations we can solve for *x* and *y* (**Figure 2.19**).

$$\cos\theta = \frac{x}{r} \longrightarrow x = r\cos\theta$$

$$\sin\theta = \frac{y}{r} \longrightarrow y = r\sin\theta$$

Figure 2.19 *Solving the sine and cosine ratios for x and y presents two more useful equations.*

This last set of equations is important for finding the *x* and *y* position of a point given *r* and θ. But what significance do *r* and θ hold? Picture a straight stick lying flat on the floor with one end at the origin (0, 0) and the other at the point (r, 0) where *r* is the length of the stick (**Figure 2.20**).

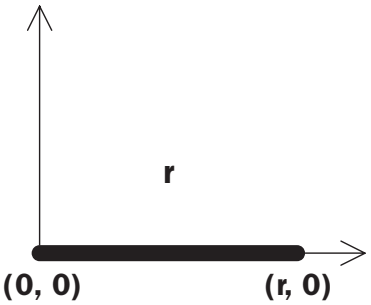

Figure 2.20 *Here's a stick with one end at the origin and the other at a point on the x-axis.*

If you were to rotate the stick around the origin by an angle of θ, one end would stay at the origin while the other would move somewhere out into the coordinate plane. Where exactly in the plane is a question we can now answer. The sine and cosine equations we solved for *x* and *y* helps us find the position of the point by simply plugging in *r* and θ (**Figure 2.21**).

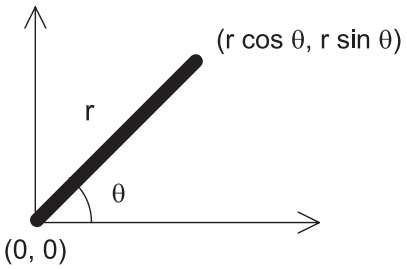

Figure 2.21 *You can use sine and cosine to find the position of the stick when it's rotated.*

From the picture above, we can conclude that if we were to go out a distance of *r* units in the direction of θ, we could find the position of that point by multiplying *r* with the sine or cosine of θ. In other words, another way of writing an ordered pair (x, y) is (r cos θ, r sin θ).

Trigonometric identities come up all the time in trigonometry and calculus. A trigonometric identity is an equation containing a trigonometric function that any angle will solve. They're particularly useful when you need a general formula for finding the sine and cosine of the sum, difference, or product of two angles. The two identities we're concerned with are called the sum identities for sine and cosine (**Figure 2.22**).

sin (a + b) = sin a cos b + cos a sin b
cos (a + b) = cos a cos b - sin a sin b

Figure 2.22 *Sum identities for sine and cosine.*

If you're curious, you can find the proofs to these identities in most textbooks. We'll apply these identities shortly.

Now that we're dealing with angles, sine, and cosine, I should mention Flash's idiosyncrasies in dealing with the trigonometric functions. There are two ways to measure an angle: in degrees and in radians. Almost everyone is familiar with degrees and knows there are 360° in a circle. Radians, however, aren't so straightforward. A radian is a ratio of the radius of a circle to an arc length of a circle (**Figure 2.23**).

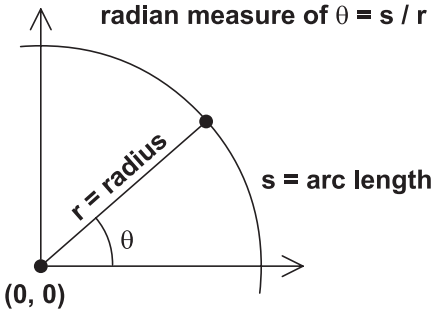

Figure 2.23 *Measuring an angle with radians.*

To make things easier, let's assume that the radius of the circle is 1. That means the radian angle measure reduces to only the arc length *s*. The arc length for an entire circle (also known as the *circumference*) is 2πr. If the radius is 1, then the circumference is 2π. This implies that an angle of 360° is equal to 2π radians. So, if we were given 360°, we could convert that to radians by multiplying it by 2π divided by 360, which reduces to π divided by 180. This is your conversion factor to translate between angle units (**Figure 2.24**).

radians = degrees • π / 180

⇓ ⇑

degrees = radius • 180 / π

Figure 2.24 *Converting between degrees and radians.*

Flash's ActionScript uses radians so you must convert all angles before you can use the Math object. Later you'll see that it's better to keep track of our rotation angles in degrees, and then change to radians when we're ready to use Flash's sine and cosine functions.

What Rotation Boils Down To

If you want to derive equations to rotate a point in space, you must first know how you're going to formulate the problem. You must also know what is meant by a rotation around the *x*-axis, or any other axis for that matter.

When you rotate a point around the *x*-axis, you literally move a point around the *x*-axis; there's no other way to put it. The same goes for the other axes. It's important to note that a point rotated around the *x*-axis doesn't change its *x*-position; only the *y*- and *z*-positions change. When an object is rotated around the *z*-axis it doesn't change *z*-positions, either. A point rotating around the *z*-axis is the same as a point rotating around the origin in a regular 2D Cartesian graph (**Figure 2.25**).

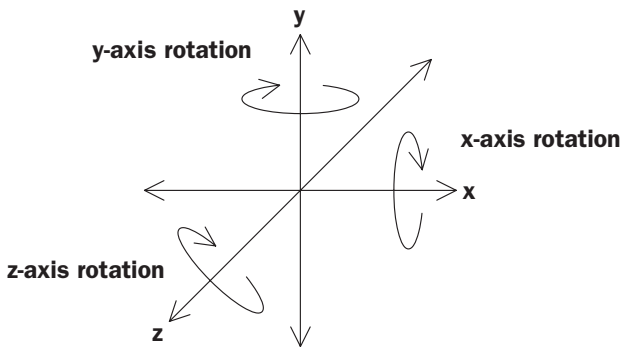

Figure 2.25 *How a point rotates around each of the three coordinate axes.*

Z-axis Rotation

Let's formulate the equation for rotating a point around the *z*-axis; the other axes will follow easily from that. Picture a point in the *xy*-plane with no *z*-position whatsoever. The point is placed at (x, y, 0), is at a distance of *r* from the origin, and makes up an angle of *a* with the origin and *x*-axis (**Figure 2.26**).

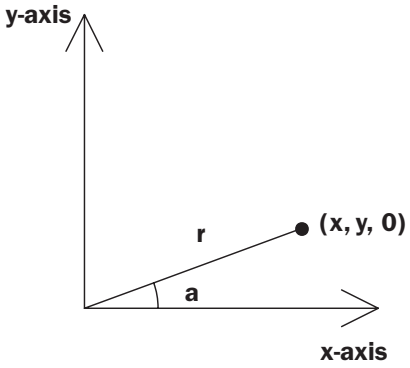

Figure 2.26 *An ordered pair makes an angle with the x- and y-axes on a 2D graph.*

Now, rotate that point by an angle of *b*, to the point (x', y', 0), while maintaining a distance of *r* from the origin (**Figure 2.27**).

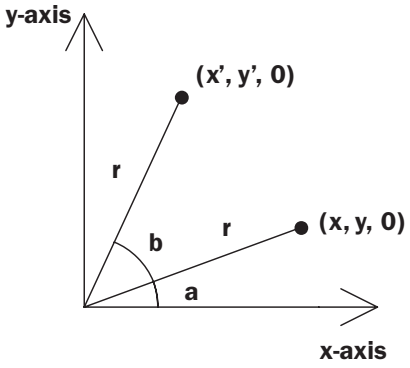

Figure 2.27 *An ordered pair rotated by an angle around the z-axis.*

Note that the angle made up from the rotated point and the *x*-axis is *a* plus *b* rather than just *b*. The angle *b* is by how much you rotated from the original position. So, the rotation problem boils down to finding (x', y', 0) by knowing *a*, *b*, *r*, and the original ordered triplet (x, y, 0).

Using Trigonometry for Rotations

Let's start by using our definitions of sine and cosine that were solved for *x* and *y* to find the position of the original point and the rotated point (**Figure 2.28**).

```
x = r cos a
y = r sin a
z = 0
```

Figure 2.28 *Equations for calculating the original position and rotated position of the point.*

```
x' = r cos (a + b)
y' = r sin (a + b)
z' = 0
```

If you're paying attention, you might start to see why the sum identities were shown earlier. These equations don't say much more than what we already know. It's important that you remember the rotated point makes an angle of *a* plus *b* with the *x*-axis. Although these equations might be of some use in rotating a point, they aren't as good as they could be. We'd have to constantly keep track of how far the point is from the distance and from the angle made from the original point and the *x*-axis. This is possible, but certainly not necessary. By substituting our sum identities, we come up with a few new equations (**Figure 2.29**).

```
x = r cos a
y = r sin a
z = 0
```

Figure 2.29 *Substitute the sum identities for sine and cosine.*

```
x' = r cos (a + b) = r (cos a cos b - sin a sin b)
y' = r sin (a + b) = r (sin a cos b + cos a sin b)
z' = 0
```

Remember that the sum identities had no quantities such as *r* involved, so *r* must stay outside the parentheses. But to simplify the expressions, we can distribute the *r* into the parentheses (**Figure 2.30**).

x = r cos a
y = r sin a
z = 0

x' = r cos (a + b) = r cos a cos b - r sin a sin b
y' = r sin (a + b) = r sin a cos b + r cos a sin b
z' = 0

Figure 2.30 *Distribute the term throughout the equation.*

If you've got a keen eye, you might see quite a few repetitions in this last set of equations. The equations that solve for (x', y', 0) have some terms that appear to be exactly the same as the equations for (x, y, 0). The terms of particular interest are those that involve the sine or cosine of *a* because these are the ratios used in (x, y, 0). Therefore, we can substitute all terms with an *r* and a sine or cosine with just *x* or *y* (**Figure 2.31**).

x = r cos a
y = r sin a
z = 0

x' = r cos (a + b) = x cos b - y sin b
y' = r sin (a + b) = y cos b + x sin b
z' = 0

⇓

x' = x cos b - y sin b
y' = y cos b + x sin b

Figure 2.31 *Simplify the equation by substituting like terms.*

You've just derived the equations for rotating a point around the *z*-axis. If you know the current position of a point (x, y, 0) and you want to rotate it around the *z*-axis by *b* degrees, simply use this last set of equations. Note that we got rid of all the awkward quantities like *r* and *a*.

Rotation Around the Z-axis

Rotation around the *z*-axis doesn't involve any 3D (because the points stay in the *xy*-plane), so let's go through a few small demos to get the hang of it.

Open z_axis_rotation1.swf from the CD-ROM.

Every time you press the Rotate button, the point will rotate around the origin by the rotation angle (**Figure 2.32**).

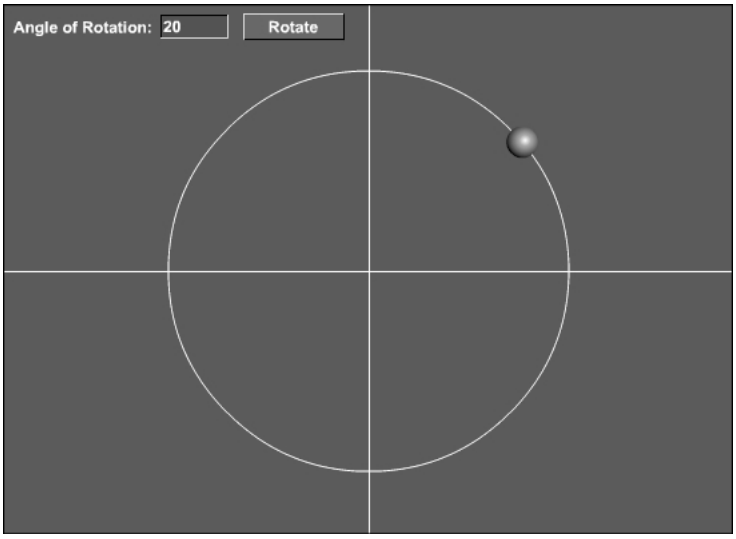

Figure 2.32 *Rotating a point around the z-axis by a user-defined angle.*

Open z_axis_rotation1.fla from the CD-ROM.

I created a textbox (called `rotation_angle`) for entering the rotation angle, a button for rotating the point, and a movie clip (called point) to represent the point. The only actions in the entire movie are placed in the first frame of the `_root` for declaring the initial variables and in the button for the rotation actions.

I added another layer to the `_root` (used only for putting some actions), and declared the initial variables in the first frame. A variable translates degrees to radians for the Math object, two variables for the initial position of the point, and two variables for the position of the origin. I also positioned the point so it would be in the right place immediately when you open the file.

```
// used for translating between angle units
trans = Math.PI / 180;
// position of origin
origin_x = 275;
origin_y = 200;
// starting position of point
x = 150;
y = 0;
// place the point for the beginning
point._x = origin_x + x;
point._y = origin_y - y;
```

The rotation actions go in the button's `on(release).` When the button is pressed and released, you must first find the sine and cosine of the rotation angle (in the textbox). Because the angle needs to be in radians only when using the Math object, I do the conversion directly in the function's parameters.

```
// find the sine and cosine of the rotation angle
sin_angle = Math.sin (Number (rotation_angle) * trans);
cos_angle = Math.cos (Number (rotation_angle) * trans);
```

Note that `rotation_angle` is run through the Number object before it's used. This is because numbers are taken from textboxes sometimes as strings, which means you can't perform mathematical operations on them. It's just a safety measure taken when dealing with textboxes.

The next few lines rotate the point. However, you can't directly change the *x* and *y* values with the equations we derived earlier. If you were to update the *x*-position of the point before you calculated the *y*-position, the identities wouldn't work. Therefore, two temporary variables are set to hold the rotated point's position, and then the *x* and *y* variables are updated.

```
// Calculate the rotated position of the point.
// Use sum identities for sine and cosine.
rotated_x = x * cos_angle - y * sin_angle;
rotated_y = y * cos_angle + x * sin_angle;
// update the position of the point
x = rotated_x;
y = rotated_y;
```

Once you've rotated the point's position, all you can do is render the point. Using the previously defined position of the origin, the code is the same as before:

```
// render point
point._x = origin_x + x;
point._y = origin_y - y;
```

Putting everything together, we have the actions for the button that will rotate a point (x, y) by the angle of `rotation_angle` around the origin:

```
on (release)
{
// find the sine and cosine of the rotation angle
sin_angle = Math.sin (rotation_angle * trans);
cos_angle = Math.cos (rotation_angle * trans);
// Calculate the rotated position of the point.
```

```
// Use sum identities for sine and cosine.
rotated_x = x * cos_angle - y * sin_angle;
rotated_y = y * cos_angle + x * sin_angle;
// update the position of the point
x = rotated_x;
y = rotated_y;
// render point
point._x = origin_x + x;
point._y = origin_y - y;
}
```

A variation of rotation around the z-axis. The next demo lets you use the Arrow keys to rotate a point continuously around the origin. The longer you hold down a key, the faster the point will spin in that direction. This file also uses what we learned about key presses earlier.

Open z_axis_rotation2.swf from the CD-ROM.

Holding down the keys lets you speed up and slow down the point's rotation. Before you look at the code, think of how you might accomplish this. How would you extrapolate the rotation angle from the user's key presses and rotate the point?

Open z_axis_rotation2.fla from the CD-ROM.

This file is similar to the first *z*-axis sample, as far as graphics go. For this file you can delete the button, the extra layer for the actions, and textbox; all the actions are placed in the movie's clip events.

The load clip event declares many variables. The constant that translates between angle units, the position of the origin, the initial position of the point, and the current rotation angle are all still declared. Also, the value that the rotation will increment by when a key is pressed down is initialized. Because we update *x* and *y* every time the point is rotated, you don't need to increment the rotation angle to keep the point moving. In other words, the `rotation_angle` quantity controls the angular speed at which the point goes around the origin. If you increment or decrement that value, you're only changing its speed.

```
onClipEvent (load)
{
// used for translating between angle units
trans = Math.PI / 180;
// position of the origin
origin_x = 275;
```

```
origin_y = 200;
// starting position of point
x = 150;
y = 0;
// increment the rotation will change by when a key is down
→ inc = .1;
// current rotation angle
rotation_angle = 0;
}
```

The enterFrame actions don't look much different than the actions that were in the button for the last file. Rather than jumping right into the rotation script, we must first increment the rotation angle. Because this is controlled by the user's keys, we resort to the script we used before. Once we've incremented the rotation angle, you find the sine and cosine of the angle, plug the numbers into the rotation equations, and render the point like before.

```
onClipEvent (enterFrame)
{
// increment rotation angle according to which key is down
rotation_angle += Key.isDown (Key.LEFT) * inc - Key.isDown
→ (Key.RIGHT) * inc;
// find the sine and cosine of the rotation angle
sin_angle = Math.sin (rotation_angle * trans);
cos_angle = Math.cos (rotation_angle * trans);
// Calculate the rotated position of the point.
// Use sum identities for sine and cosine.
rotated_x = x * cos_angle - y * sin_angle;
rotated_y = y * cos_angle + x * sin_angle;
// update the position of the point
x = rotated_x;
y = rotated_y;
// render the point
this._x = origin_x + x;
this._y = origin_y - y;
}
```

Rotation Around the Other Axes

The other axes don't need to be covered in as much depth as the *z*-axis because they're almost identical. Once the equations are derived for all three axes, you'll see that they differ only by a few values.

We concluded earlier that a rotation around the *x*-axis doesn't affect the *x*-position of a point. Therefore, to imagine a point rotating around the *x*-axis, we turn our view to the side so that the *x*-axis is pointing at us and the other axes are vertical and horizontal. Then, we have a point somewhere in the *yz*-plane that makes up an angle of *a* with the *x*-axis and we wish to rotate it to a new position by an angle of *b*. This is the same situation we came across before, except only the *y*- and *z*-positions are affected (**Figure 2.33**).

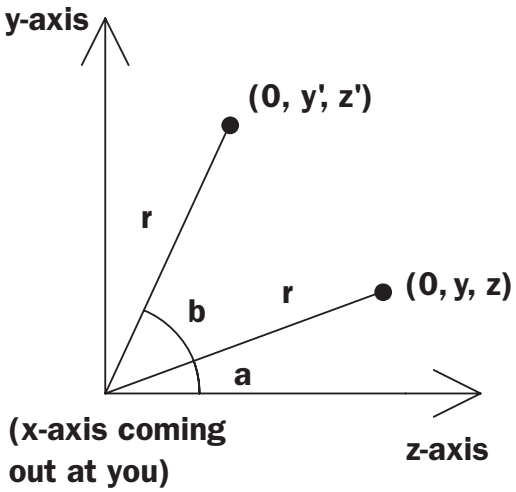

Figure 2.33 *A point rotated around the x-axis.*

From the picture above, we can formulate our first set of equations, much like we did before. Again, note that the angle created by the rotated point and the x-axis is *a* plus *b* (**Figure 2.34**).

Figure 2.34 *Position of the original and rotated point.*

```
x = 0
y = r sin a
z = r cos a

x' = 0
y' = r sin (a + b)
z' = r cos (a + b)
```

We'll skip most of the work here. By substituting in the sum identities again and simplifying, you get these steps (**Figure 2.35**):

x = r cos a
y = r sin a
z = 0

x' = 0
y' = r sin (a + b) = r sin a cos b + r cos a sin b
z' = r cos (a + b) = r cos a cos b - r sin a sin b

⇓

y' = y cos b - z sin b
z' = z cos b + y sin b

Figure 2.35 *Equations used to rotate a point around the x-axis.*

You've now derived the equations for rotating a point around the *x*-axis. If you know the current position of a point (x, y, z) and you want to rotate it around the *z*-axis by *b* degrees, simply use this last set of equations. Once again, the *r* and *a* terms dropped out because they aren't necessary to rotate a point.

The derivations of the equations that rotate a point around the *y*-axis are almost entirely the same. Therefore, I've left out the geometric representations, and am showing only the algebraic work (**Figure 2.36**).

x = r cos a
y = 0
z = r sin a

⇓

x' = r cos (a + b)
y' = 0
z' = r sin (a + b)

⇓

x' = r cos (a + b) = r cos a cos b - r sin a sin b
y' = 0
z' = r sin (a + b) = r sin a cos b + r cos a sin b

⇓

x' = x cos b - z sin a
z' = z cos b + x sin a

Figure 2.36 *Equations used to rotate a point around the y-axis.*

We've now derived all the equations for rotating a point around the three axes. In summary, here are all six equations (**Figure 2.37**).

Given Point: (x, y, z)
Point rotated around the x-axis: (x, y', z')
Point rotated around the y-axis: (x'', y', z'')
Point rotated around the z-axis: (x''', y''', z'')

x-axis rotation

```
y' = y cos b - z sin b
z' = z cos b + y sin b
```

y-axis rotation

```
x'' = x cos b - z'  sin b
z'' = z' cos b + x sin b
```

z-axis rotation

```
x''' = x'' cos b - y'  sin
y''' = y' cos b + x'' sin
```

Figure 2.37 *A reference list of all the equations used to rotate a point around each of the three axes.*

Although the apostrophes may be confusing, it's crucial that you understand what they mean. These equations demonstrate an important 3D concept: Use the newest position of a point when you want to rotate it. This means that if you want to rotate a point on the *x*- and *y*-axis, you would first rotate around the *x*-axis, and then use the rotated position in the equations to rotate around the *y*-axis.

The outline of a script that rotates many points around each axis by angles of *a*, *b*, and *c* might look something like this:

1. Find the sine and cosine of the rotation angles a, b, and c.
2. Use a loop to perform a rotation on and render each point.

   Translate the point on all three axes (if applicable).
3. Rotate the point by the *x*-axis.
4. Use the current position from the *x*-axis rotation and rotate the point by the *y*-axis.
5. Use the current position from the *y*-axis rotation and rotate the point by the *z*-axis.
6. Calculate the position of the point on the screen.
7. Set all the properties of the movie clip representing the point.

Optimizations

We now have all the information we need to create some spectacular effects in Flash. Making these effects run well in Flash, however, is equally important. Most sophisticated methods for reducing CPU calculations quickly become too burdensome in Flash for two reasons: Flash can't handle the optimization calculations, and there aren't enough objects in the environment to fully use the algorithms. We'll cover the few optimizations that are practical in Flash.

Hidden Objects

Our previous files adhered to a popular 3D optimization philosophy: *If you can't see it, don't render it*. When a point went behind the viewer, for instance, we didn't render it. This time, we'll remove a point if it goes off the viewing area.

An object is off the viewing area if its *x*- or *y*-position is greater than the Stage's width or height, and if its *x*- or *y*-position is less than 0. A few conditionals can test this problem and set the point's `_visible` property to either `false` or `true`, accordingly.

```
OnClipEvent (load)
{
// width and height of the stage
stage_width = 550;
stage_height = 400;
}
onClipEvent (enterFrame)
{
// check if clip has gone off the stage
if ((this._x > stage_width) || (this._x < 0))
{
 // clip is off the left or right side of the stage
 this._visible = false;
}
else if ((this._y > stage_height) || (this._y < 0))
{
 // clip is off the top or bottom of the stage
 this._visible = false;
}
else
{
```

```
 // clip is on the stage
 this._visible = true;
}
}
```

Remember that you must do all the translations, rotations, and perspective calculations before you can check if the point is visible. Removing hidden objects may not save you some calculations, but it will minimize the number of objects that are simultaneously displayed on Flash's Stage.

Perform Fewer Calculations

This optimization applies to programming as a whole. To speed anything up, you must do the fewest calculations possible. This means using the Math object as little as possible, only doing actions that are absolutely necessary, and simplifying expressions to use fewer multiplies and divides.

The Math object. Although it's not incredibly slow, the Math object should only be called a few times as it can tax the user's CPU. Many people make the mistake of finding the sine and cosine of the rotation angles from within the loop that rotates all the points. Because the rotation angles shouldn't be changing from one point to the other, you need to find the sine and cosine loop only once.

Here's some code with comments that outlines this:

```
OnClipEvent (enterFrame)
{
// change rotation angles or translation increments
// calculate sine and cosine of rotation angles
for (var j = 0; j < num_points; j++)
{
// translate
// rotate
// calculate perspective
// check if on stage
// render
}
}
```

Some people create look-up tables for sine and cosine values so they don't have to use the Math object. While this used to be common practice in C++ and other computer languages, it's unnecessary with today's powerful personal computers.

Showcase 1: Rotating a Field of Points

My first showcase is similar to the demo we've been developing throughout the chapter. It creates a field of randomly placed points and lets you use buttons to control their rotation. The structure of the script and movie is entirely different, so let's start over with something new.

Open demo_one1.swf from the CD-ROM.

The demo clearly displays the rotation angles around each axis and continuously rotates the points by those increments (**Figure 2.38**).

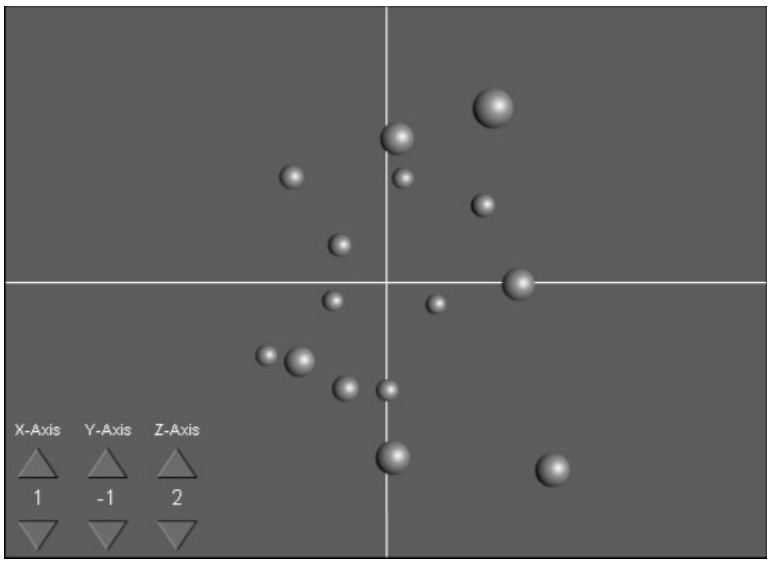

Figure 2.38 *Rotation points around each axis through the use of buttons.*

Open demo_one1.fla from the CD-ROM.

The Layout

This file has seven main graphics. Six are buttons that control the increment and decrement of each of the three rotation angles. The last graphic is the movie clip that represents the point. Unlike before, most of the code isn't placed in the clip events of the point movie. Due to my experience in C++, I prefer to have as much of my code in one place as possible. Therefore, I created a movie clip named dummy to place all of the main actions.

Variable Scope

The code is mainly contained in the dummy's clip events, except for three variables: the rotation angles around each axis. These variables are kept on the `_root` of the movie so the buttons can easily change their values. The variables are accessed from the dummy movie clip to find the sine and cosine of the angles.

Getting Rid of an Unwanted Movie Clip

The movie clip named point is duplicated many times to create the field of points. We only want the duplicated clips to be visible, however, because the script that rotates the points won't affect the base movie clip. Therefore, I put the following actions in the load clip event of the point movie to make the base clip invisible and the duplicates visible:

```
OnClipEvent (load)
{
if (this._name == "point")
{
this._visible = false;
}
}
```

Initializing the Movie

In the first frame of the `_root`, the variables `rotation_angle_x`, `rotation_angle_y`, and `rotation_angle_z` have been initialized to represent the rotation angles around each axis. These values are changed when one of the six buttons are pressed and are accessed from the dummy movie clip. Because no rotation should be taking place initially, they are set to 0.

I placed the rest of the actions in the dummy movie clip. Keeping track of the ordered triplet wasn't a problem in our previous files because all of the point's information was contained in itself. This time we're controlling everything from an external script, which means we have to keep track of many points. We could simply create many variables named `x1`, `x2`, `y1`, `y2` and so on for every point's ordered triplet, but that's not efficient as far as memory usage goes. Instead, I use an array of objects. Each element in the array represents a point and each member variable of the object represents a value of the ordered triplet.

Therefore, in the load clip event of the dummy movie, we create an array, loop through it, and create an object in each element. Then we set the initial ordered triplet for the point. In that same loop, the point movie clips are also duplicated.

```
onClipEvent (load)
{
// number of points to be placed out randomly
num_points = 15;
// array which will hold the position of the points
point_position = new Array (num_points);
// duplicate points and set initial position randomly
for (var j = 0; j < num_points; j++)
{
// duplicate new movie clip
  _parent.point.duplicateMovieClip ("point" + j, j);
// create object for x-, y-, and z-position of point
point_position[j] = new Object ();
// set point's initial position randomly
point_position[j].x = Math.random () * 200 - 100;
point_position[j].y = Math.random () * 200 - 100;
point_position[j].z = Math.random () * 200 - 100;
}
```

The rest of the actions in the load clip event declare all the variables we've used before: the position of the origin, the size of the point, the perspective distance, and a value for translating between angle units.

```
// position of origin
origin_x = 275;
origin_y = 200;
// used for perspective - distance from the viewer to
→ the screen
D = 300;
// size of movie clip at the origin
regular_size = this._width;
// used to translate between angle units
trans = Math.PI / 180;
}
```

Button Actions

The button actions are simple. When a button pointing upwards is pressed, the rotation value for that button is increased; when a button pointing downwards is pressed, the rotation value is decreased. It's up to you to decide by how much the rotation angles are incremented; I used the increment and decrement operators `++` and `--`, which will change the values by 1.

Here are the actions that are executed when the Up arrow for the *x*-axis rotation is pressed:

```
on (release)
{
// increase the x-axis rotation angle by one
rotation_angle_x++;
}
```

Main

The last set of actions in the movie clip is in the dummy's clip events. This takes care of all the rotations and renderings. Although there's nothing new in the 50 or so lines, there are a couple of pitfalls to watch out for. You'll also see one optimization technique used in this script.

Although it may seem like common sense, too many people make the mistake of rotating around every axis whether they need to or not. On many occasions, you only need to rotate around one or two axis. Rotating on more axes than you need to only adds unnecessary calculations to your script.

Because the file doesn't deal with any translations, we'll skip right to rotational movement. At the very beginning, the sine and cosine of the rotation angles are defined. Make sure to translate the angle units from degrees to radians.

```
onClipEvent (enterFrame)
{
// calculate the sine and cosine of the rotation angles
sin_a = Math.sin (_parent.rotation_angle_x * trans);
cos_a = Math.cos (_parent.rotation_angle_x * trans);
sin_b = Math.sin (_parent.rotation_angle_y * trans);
cos_b = Math.cos (_parent.rotation_angle_y * trans);
sin_c = Math.sin (_parent.rotation_angle_z * trans);
cos_c = Math.cos (_parent.rotation_angle_z * trans);
```

Now you must loop through every element of the `point_position` array and rotate the ordered triplet around each axis. I prefer to use a `for` loop because the number of elements in the array is not likely to change and because you need integers to access the elements.

```
// loop through all the points and rotate and render
for (var j = 0; j < num_points; j++)
{
// actions
}
```

The first things to be carried out in the `for` loop are the rotations. Remember that the rotation around the *x*-axis is done first, the *y*-axis second, and the *z*-axis last. Also, use the most current position of the point to do your next rotation and don't update the `point_position` array until you've performed all rotations. It's OK to refer back to the equation reference picture from earlier—it's not easy to memorize six equations!

```
// rotate around the x-axis
rx1 = point_position[j].x;
ry1 = point_position[j].y * cos_a - point_position[j].z
→ * sin_a;
rz1 = point_position[j].z * cos_a + point_position[j].y
V* sin_a; // rotate around the y-axis
rx2 = rx1 * cos_b - rz1 * sin_b;
ry2 = ry1;
rz2 = rz1 * cos_b + rx1 * sin_b;
// rotate around the z-axis
rx3 = rx2 * cos_c - ry2 * sin_c;
ry3 = ry2 * cos_c + rx2 * sin_c;
rz3 = rz2;
// update array that holds the position of the points
point_position[j].x = rx3;
point_position[j].y = ry3;
point_position[j].z = rz3;
```

The script has so far taken every point's ordered triplet and rotated it around the *x*-, *y*-, and *z*-axes. When all 3D movement is done, you're only

left with rendering the point. The perspective ratio is calculated first and the position of the point on a 2D screen is calculated next.

```
// calculate the perspective ratio
perspective_ratio = D / (D + rz3);
// calculate the position of point on computer screen
perspective_x = rx3 * perspective_ratio;
perspective_y = ry3 * perspective_ratio;
```

You might wonder what you're supposed to do after all these calculations. The perspective ratio is key to finding the position of the point on the Stage and sizing it appropriately. To calculate the perspective ratio, you need to know the point's ordered triplet. To find the most current ordered triplet, you must translate and rotate it by various values. So, once you get it down to the perspective ratio and the ordered pair, you simply set a few properties.

```
// update position of movie clip on stage
_parent["point" + j]._x = origin_x + perspective_x;
_parent["point" + j]._y = origin_y - perspective_y;
// update size of movie clip
_parent["point" + j]._xscale = _parent["point" + j]._yscale
→ = regular_size * perspective_ratio;
// set the depth of the clip based on its z-position
    _parent["point" + j].swapDepths (-rz3);
```

A Simple Variation

Instead of controlling the rotations with buttons, this time we use certain keys. Pressing the Up or Down arrow keys changes the *x*-axis rotation, pressing the Left or Right arrow keys changes the *y*-axis rotation, and pressing the Control key or the 0 key on the number pad changes the *z*-axis rotation.

Open demo_one2.fla from the CD-ROM.

We use the same technique from earlier, but we must slightly change our script from demo_one1.fla. First, take out the buttons and the actions on the first frame of the `_root`. The variables for the rotation angles will be kept in the dummy movie clip from now on. Therefore, the following lines need to be placed in the load clip event of the dummy movie:

```
// rotation angles of each axis
rotation_angle_x = 0;
rotation_angle_y = 0;
rotation_angle_z = 0;
```

We also need to initialize a variable in the load clip event to determine how much the rotation angles should increment or decrement by when a key is pressed. Play around with different values to find what you like best.

```
// increment the rotation angles will change when a key
→ is down
rotation_inc = .5;
```

Another variable that's initialized might not make sense at first. In the Key object there are many variables set. The variables have the names of various keys and hold numerical values. For instance, `Key.LEFT` has a numerical value of 37. The values are used to check if certain keys are being held down with the `isDown()` method. The Key object has predefined variables for every key we could use in this sample except for number pad 0. You'd have to pass the numerical value of 96 to the `isDown()` method. However, if you're following good programming practice, you should be using as few numerical values in your code as possible. Therefore, I set a `member` variable in the Key object for number pad 0 and simply pass that variable to the method.

```
// key code value for the number pad zero
Key.NUMPAD_0 = 96;
```

That's all of the extra variables this variation uses. In the enterFrame clip event we need to add the actions that increment or decrement the rotation angles, depending on which key is pressed. This is done before the sine and cosine of the angles are calculated because we want only the newest angles when calculating those ratios.

```
// increment rotation values according to which keys
→ are pressed
rotation_angle_x += Key.isDown (Key.UP) * rotation_inc -
→ Key.isDown (Key.DOWN) * rotation_inc;
rotation_angle_y += Key.isDown (Key.RIGHT) * rotation_inc -
→ Key.isDown (Key.LEFT) * rotation_inc;
rotation_angle_z += Key.isDown (Key.CONTROL) * rotation_inc
→ - Key.isDown (Key.NUMPAD_0) * rotation_inc;
```

The last thing to be changed is when using the Math object. Before, we had to access the rotation angles from the `_root`, which means we had to use a

target path. This time, the rotation angles are in the same timeline so we can drop the `_parent` path.

```
// calculate the sine and cosine of the rotation angles
sin_a = Math.sin (rotation_angle_x * trans);
cos_a = Math.cos (rotation_angle_x * trans);
sin_b = Math.sin (rotation_angle_y * trans);
cos_b = Math.cos (rotation_angle_y * trans);
sin_c = Math.sin (rotation_angle_z * trans);
cos_c = Math.cos (rotation_angle_z * trans);
```

That's it for the variation. You can control the rotation of the points around each axis with six keys.

Showcase 2: 3D World

The effects in this showcase may seem complex, but they're easy to create. We'll use all of the 3D knowledge we've gained so far, along with some extra information, to create a 3D world in Flash.

Open demo_two1.swf from the CD-ROM.

Use the arrow keys to explore the world (**Figure 2.39**). Take a second to ponder how you might achieve this effect without looking on. What kind of rotation is taking place? What kinds of translations are taking place?

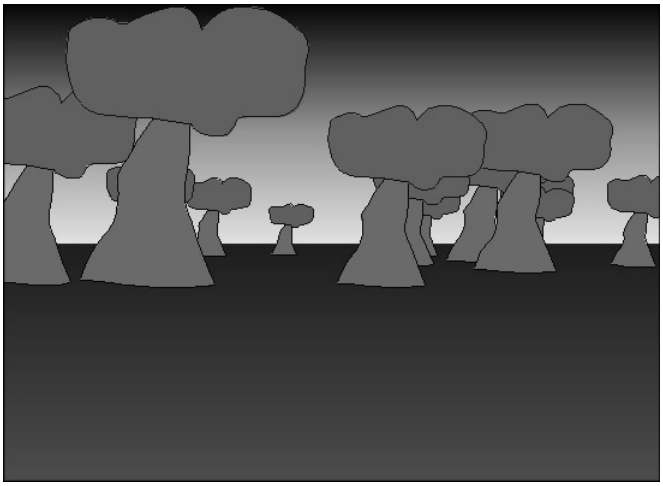

Figure 2.39
A 3D world created in Flash.

Open demo_two1.fla from the CD-ROM.

3D World in Flash

A 3D world in Flash must be simple in all aspects. Its objects should be static and look the same from all perspectives. It's possible to have an object pre-rendered from different angles so that, depending on your location in the world, the image could be set to a different frame. This is rarely worth doing, however, as it will great increase your file size and slow performance.

Due to these limitations, creating a 3D world in Flash requires using only a few techniques that we've learned. Because you want the world to encapsulate you from the sides, the only rotation possible in the *xz*-plane is around the *y*-axis. Also, the only axes you can move through are the *x*- and *z*-axes. Moving along the *y*-axis would mean to go up above the world, which would not look right because the objects are static.

In this showcase, a graphic of one tree is placed multiple times at random throughout the world. The instance name of the movie clip with the tree inside is objects. It's kind of hard to decide how to represent this tree in 3D space, as it's composed of an infinite number of points. To keep a sense of continuity in our world, we must choose only one point to represent the tree. The most defining point of the tree and the world is where the two meet. Therefore, the bottom of the tree is placed at the center of the movie clip and the ordered triplet will be the position of the bottom of the tree.

3D World Trickery

We use two simple translations to create the illusion of objects surrounding us. When we derived the equations for finding the position of a 3D point on a 2D surface, we assumed that the viewer was sitting at a distance (d) away from the screen. In an immersive environment, however, there's no distance between the viewer and computer screen. So, for our first translation, we'll translate every point in the world toward the viewer by an increment of d. Because this translation is for rendering purposes only, it shouldn't affect the points' actual positions.

The second translation gives the viewer height in the 3D world. Because the tree's position is determined by the position of its bottom, the viewer would see the world from ground level. To fix this, you translate all the points down the *y*-axis by a certain amount to make the trees' bottoms seem slightly below the viewer. This translation is also temporary and shouldn't affect the points' actual positions. Play around with height's value until you find what you like.

3D World Idiosyncrasies

A few things in the file can cause havoc in the overall production. The first is the background image. I drew a simple green ground that meets the sky in the middle of the Stage. The ground and sky don't meet in the middle by chance; the horizon is the vanishing point for all the objects in the world. This means that as the objects move farther away, they move closer to the horizon. The vanishing point is chosen when you pick a value for the *y*-coordinate of the origin. In general, make the horizon's *y*-position the same as the origin's.

This file also uses the technique we used last time by placing all the code in a dummy movie clip that does all the main calculations. All variables are kept in this movie clip and all 3D calculations and rendering actions are done in its clip events.

All the trees placed in the world will be duplicated from a main tree. We place actions in clip events to remove this original tree, as it's the only one that doesn't need to be rendered. The actions are similar to the ones we used before.

```
OnClipEvent (load)
{
if (this._name == "objects")
{
this._visible = false;
}
}
```

Initializing the 3D World

The variables, objects, duplicate movie clips, and initial conditions are set in the load clip event of the dummy movie clip. All of the values initialized are what we've already been using, with a few exceptions.

Our first task is to duplicate the trees and place them randomly. Each tree is also given a random size to create a more natural-looking forest. First, we initialize a variable for the number of objects to be duplicated and the array that will hold the position of each object:

```
onClipEvent (load)
{
// number of objects to create num_objects = 20;
// array which will hold the position of the objects
object_position = new Array (num_objects);
```

The script loops through and duplicates the trees. An object is created for each element in the `object_position` array. The object will have a variable for the *x*-, *y*-, and *z*-position of the point as well as a value that will hold the size of the regular size tree.

```
// duplicate objects and initialize random properties
for (var j = 0; j < num_objects; j++)
{
// duplicate object
_parent.objects.duplicateMovieClip ("objects" + j, j);
// create object for ordered triplet and size of object
object_position[j] = new Object ();
// set object's initial position randomly
object_position[j].x = Math.random () * 1500 - 750;
object_position[j].y = 0;
object_position[j].z = Math.random () * 1500 - 750;
// size of movie clip at the origin - set randomly
object_position[j].regular_size = Math.random () * 50 + 50;
}
```

The rest of the variables set are the constants of the world, the current rotation angle, and translation increment. Like all other 3D demos, you need values for the position of the origin, the distance from the viewer to the computer screen for perspective, and a constant for translating between angle units.

```
// position of origin
origin_x = 275;
origin_y = 200;
// used for perspective - distance from the viewer to
→ the screen
D = 500;
// used to translate between angle units
trans = Math.PI / 180;
```

A new value set is for the viewer's height. Play around with this value to see what you like best. You might want to let the viewer control this value. For instance, when the viewer holds down the space bar the value could be set to something smaller, which would create the effect of crouching down in the world.

```
// height of player walking around in world
player_height = 30;
```

The next three values help handle the translation movement. The first value is the increment at which the viewer moves when a key is pressed. Note that it's not the value that the viewer *is* moving at along the axes, but rather what the viewer *will* move at when a key is down. The other two values are the rates at which the viewer is currently moving along the *x*- and *z*-axes. These values are added to the viewer's *x*- and *z*-positions every frame.

```
// increment to move along the x- and z-axes when a key
→ is pressed
translation_inc = 10;
// current translation across the x- and z-axes
translation_x = 0;
translation_z = 0;
```

Basically, the same thing is done for the rotational movement as for the translations. A value is set for the rate at which the viewer will turn when a key is down. The second, however, is the increment at which the viewer is currently turning around the *y*-axis.

```
// increment to rotate around the y-axis when a key
→ is pressed
rotation_inc = 2;
// current rotation angle around the y-axis
rotation_angle_y = 0;
```

Finally, another value is set in the `Key` object for number pad zero. The key layout of the demo will be covered shortly.

```
// key code value for the number pad zero
Key.NUMPAD_0 = 96;
}
```

Key Layout

The key layout of the demo is important because it lets viewers easily navigate the world. We must let them control their turns around the *y*-axis, as well as their movement back and forth along the *z*-axis, and side to side along the *x*-axis. Choosing keys to control the *y*-axis rotation and *z*-axis translation movement is pretty straightforward: The Arrow keys make the most sense. Side-to-side movement is less obvious: I chose the Control and the number pad 0 keys.

Main

The rest of the actions in the demo are carried out in the dummy's clip events. These final 70 to 80 lines use every technique we've talked about so far, including the optimizations.

Before we can translate, rotate, or render any points, we must change the current translation and rotation increments. Once again, we'll use the technique from before.

```
onClipEvent (enterFrame)
{
// change the x- and z-axis translation of the objects
→ according to which keys are pressed
translation_x = Key.isDown (Key.CONTROL) * translation_inc -
→ Key.isDown (Key.NUMPAD_0) * translation_inc;
translation_z = Key.isDown (Key.DOWN) * translation_inc -
→ Key.isDown (Key.UP) * translation_inc;
// change the y-axis rotation angle according to which keys
→ are pressed
rotation_angle_y = Key.isDown (Key.RIGHT) * rotation_inc -
→ Key.isDown (Key.LEFT) * rotation_inc;
```

Once the current rotation angle is known you must find its sine and cosine for the rotation equations. Remember that these values need to be calculated only once before the rotation equations are used. Don't take the sine and cosine of the rotation angle in the `for` loop that rotates and renders all the points.

```
// sine and cosine of rotation angle
sin_y = Math.sin (rotation_angle_y * trans);
cos_y = Math.cos (rotation_angle_y * trans);
```

After the script calculates all the increments at which the world will be translated and rotated, it loops through every point and perform these transformations. I used a `for` loop again.

```
for (var j = 0; j < num_objects; j++)
{
// other actions
}
```

Inside the `for` loop is the meatiest part of the entire script. Before we can start rendering or checking if a point is off the Stage, we must translate and rotate the point. First, the object's position is translated and the translated values are placed in temporary variables; the `object_position` array is still not changed. Then, the translated point is rotated around the *y*-axis. Once the point has been translated and rotated, we update the values in the `object_position` array.

```
// translate the object across the x- and z-axes
tx = object_position[j].x + translation_x;
ty = object_position[j].y;
tz = object_position[j].z + translation_z;
// rotate the translated point around the y-axis
rx = tx * cos_y - tz * sin_y;
ry = ty;
rz = tz * cos_y + tx * sin_y;
// update object position array
object_position[j].x = rx;
object_position[j].y = ry;
object_position[j].z = rz;
```

In the earlier files, this was as far as we had to go to render a point. A 3D world, however, must completely surround the viewer. This calls for the two extra translations we discussed earlier. The translations affect only the *ry* and *rz* variables. The `object_position` should not be changed because you only want to temporarily translate the points for rendering purposes.

```
// translations used for making the world surround the
→ viewer
// and rendering – doesn't affect the point's actual
→ position.
// shift the rotated and translated point down the y-axis
→ to give user height
ry -= player_height;
// shift the rotated and translated point down the z-axis
→ to make first person perspective
rz -= D;
```

The points are now in their final positions; the script has performed all of the transformations needed. All we need to do now is render everything.

When rendering, we want to take every shortcut possible. The first shortcut is one we've used before: Check if an object is behind the view. If the most current value of the *z*-position of the point is less than the negative value of the perspective distance, then you don't need to render the point.

```
// check if object has gone behind viewer
if (rz < -D)
{
// object is behind viewer, make invisible
_parent["objects" + j]._visible = false;
}
else
{
// other actions
}
```

Even if the object is in front of the viewer, you should still check if it is on the Stage. If the object is placed far off screen, you still don't want the code to render it on the Stage. Before we can determine if the object is off the Stage, however, we must first calculate the values used in rendering, such as the point's position on the Stage and the size of the point with perspective. Remember that these actions are performed in the `else` of the conditional and are only executed when the object is in front of the viewer.

```
// object is not behind viewer but still may be off
→ the screen,
// so don't make visible yet
// calculate perspective ratio
perspective_ratio = D / (D + rz);
// calculate the position of point on computer screen
perspective_x = origin_x + rx * perspective_ratio;
perspective_y = origin_y - ry * perspective_ratio;
// perspective scale of object
perspective_scale = object_position[j].regular_size *
→ perspective_ratio;
```

Once the perspective values have all been calculated, you can check if the object is on the Stage or not. An object is off the left side of the Stage if its right-most point is less than 0; an object is off the right side of the Stage if its left-most point is greater than the Stage width. By taking the object's

x-position, adding or subtracting the size of the object, and checking if it has gone out of the bounds, you can determine whether the object is still on the Stage.

```
// check if object is off the side of the screen
if (((perspective_x + perspective_scale) < 0) ||
→ ((perspective_x - perspective_scale) > origin_x*2)))
{
// object is off the screen, make invisible
_parent["objects" + j]._visible = false;
}
else
{
// other actions
}
```

All the rendering actions go in the actions of the `else` of this conditional. First, the object's visibility should be set to `true` because it's in front of the viewer and on the Stage. After that, all the set properties are the same as before.

```
// the object is on the screen and in front of the view,
→ make visible
_parent["objects" + j]._visible = true;
// update position of movie clip on stage
_parent["objects" + j]._x = perspective_x;
_parent["objects" + j]._y = perspective_y;
// update size of movie clip
_parent["objects" + j]._xscale = _parent["objects" +
→ j]._yscale = perspective_scale;
// set the depth of the clip based on its z-position
_parent["objects" + j].swapDepths (-rz);
```

Close all the curvy braces to end your script, and you're done: You've just created a 3D Flash world. The number of lines for the entire script did not go above 150 and the effect is mesmerizing.

Showcase 3: 3D Menu System

The final showcase is a 3D-menu system with buttons that open new browser windows via URLs that you define in the code. This project works equally well for a gallery showcase.

Open demo_three1.swf from the CD-ROM.

This file randomly places a few menu items in 3D space and lets you navigate through them. We won't add any rotation to the environment because we can't control what the menu items look like at different angles (**Figure 2.40**).

Figure 2.40 *Navigable menu items placed randomly in space.*

Open demo_three1.fla from the CD-ROM.

Starting the Demo

This demo has very few components to it and no tricks in its mechanics. You'll probably want to create a background to cover the entire Stage. If you want to create the illusion of a horizon, be sure to make the Stage's *y*-position the same as the *y*-position of the origin. You should also create a movie clip (called menu_item) for the menu item names. I chose to create a long textbox (to accommodate long names) named field. I also created an invisible button behind it so that an URL is brought up when the item is clicked. A dummy movie clip handles the main actions.

Actions

There are only three places to look for code: the button inside the menu_item movie clip, the clip events of the menu_item movie clip, and the clip events of the dummy movie clip. The most important scripting is in the latter clip; only a few lines are in the other two.

Because the menu_item movie clip will be duplicated many times, I put actions in it to remove the clip if it's not a duplicated clip. This prevents you from having extra movie clips on the Stage.

```
onClipEvent (load)
{
// remove base menu item
if (this._name == "menu_item")
{
this._visible = false;
}
}
```

The main actions in the dummy clip events are slightly different from what we've been working with so far. First, we initialize a variable for the number of menu items we expect to have. For each menu item we must keep track of information like its name, the URL it's linked to, and its ordered triplet. To organize this information, we create an array with an object in each of its elements. The array elements will hold the menu items' information and the object will hold the miscellaneous pieces of information.

```
onClipEvent (load)
{
// number of menu items
num_menu_items = 9;
// array that will hold the information for each menu item
→ menu_item_info = new Array (num_menu_items);
// create an object in each element of the information
→ array for (var j = 0; j < num_menu_items; j++)
{
menu_item_info[j] = new Object ();
}
```

Next, we initialize the names and URLs of all the menu items. You can choose to enter them directly into Flash, or load them from an external text file. However you do it, make sure the information is put into the

menu_item_info array so it can be used in the rest of the script. I picked 10 random names and URLs.

```
// names of menu items
menu_item_info[0].name = "Home";
menu_item_info[1].name = "News";
menu_item_info[2].name = "Portfolio";
menu_item_info[3].name = "Locations";
menu_item_info[4].name = "Staff";
menu_item_info[5].name = "History";
menu_item_info[6].name = "Awards";
menu_item_info[7].name = "Business";
menu_item_info[8].name = "About";
menu_item_info[9].name = "Contact";
// urls to be executed when a button is pressed
menu_item_info[0].url = "http://www.home.com";
menu_item_info[1].url = "http://www.news.com";
menu_item_info[2].url = "http://www.portfolio.com";
menu_item_info[3].url = "http://www.locations.com";
menu_item_info[4].url = "http://www.staff.com";
menu_item_info[5].url = "http://www.history.com";
menu_item_info[6].url = "http://www.awards.com";
menu_item_info[7].url = "http://www.business.com";
menu_item_info[8].url = "http://www.about.com";
menu_item_info[9].url = "http://www.contact.com";
```

Once all the information is gathered, you must duplicate the menu items and set the initial conditions. Each item's name and URL values are sent to the movie clip so they can be displayed and used with the button's actions.

The items' ordered triplets are initialized in a certain way. The effect we're going for is to have items spread out in front of the viewer. This means the items should be placed along the *x*- and *z*-axes only. I added a small amount of random *y*-position, however, to space things out better.

```
// duplicate menu items and set out randomly
for (var j = 0; j < num_menu_items; j++)
{
// duplicate menu item
_parent.menu_item.duplicateMovieClip ("menu_item" + j, j);
// set text field inside movie clip for menu item name
→ parent["menu_item" + j].field = menu_item_info[j].name;
```

```
// pass url to movie clip for when the button is pressed
→ parent["menu_item" + j].url = menu_item_info[j].url;
// place randomly on the x- and z-axis
menu_item_info[j].x = Math.random () * 1000 - 500;
menu_item_info[j].y = Math.random () * 100 - 50;
menu_item_info[j].z = Math.random () * 500;
}
```

The only variables left to be initialized are the constants and values that change the position of the items. A variable for the size of the menu item at the origin has also been set. You most likely want the size of all the menu items to be the same at the origin, so you don't have to keep track of that value in the `menu_item_info` array.

```
// position of the origin
origin_x = 275;
origin_y = 200;
// used for perspective - distance from the viewer to
→ the screen
D = 300;
// used to translate between angle units
trans = Math.PI / 180;
// increment to move along the x- and z-axes when a key
→ is pressed
translation_inc = 10;
// current translation across the x- and z-axes
translation_x = 0;
translation_z = 0;
// size of menu items
menu_item_size = 200;
}
```

The rest of the actions in the dummy movie clip are in the enterFrame clip event. These actions take care of the translations and rendering. There are no rotations in this showcase, so there's no need to keep track of rotation angles and the sine and cosine of the angles.

First, we change the current translation increment along the *x*- and *z*-axes, depending on which keys are pressed. (You could also let viewers move along the *y*-axis if you don't think that would be too cumbersome.)

```
onClipEvent (enterFrame)
{
// change the x- and z-axis translation of the items
→ according to which keys are pressed
translation_x = Key.isDown (Key.LEFT) * translation_inc -
→ Key.isDown (Key.RIGHT) * translation_inc;
translation_z = Key.isDown (Key.DOWN) * translation_inc -
→ Key.isDown (Key.UP) * translation_inc;
```

When all the information is calculated for moving the points, the script will loop through every point and translate and render it. The translated points' positions are first put into two temporary variables, `tx` and `tz`, and then the `menu_item_info` array is updated. Putting the translated values into two new variables helps shorten the rest of the code because you don't need to type "menu_item_info" every time; using temporary variables also keeps you from having to index the array many times.

```
// loop through all the points and translate and render them
for (var j = 0; j < num_menu_items; j++)
{
// translate the object across the x- and z-axes
tx = menu_item_info[j].x + translation_x;
tz = menu_item_info[j].z + translation_z;
// update the array with all the item's information
menu_item_info[j].x = tx;
menu_item_info[j].z = tz;
```

Once the point has been translated, you do the first visibility check. When the point goes behind the viewer, there's no need to render it and calculate its position on the Stage. If it's in front of the viewer, you still have one more check left to do depending on whether or not the point is on the Stage.

```
// check if menu item has gone behind the viewer
if (tz < -D)
{
// item is behind viewer, make invisible
_parent["menu_item" + j]._visible = false;
}
else
{
// other actions
}
```

Inside the `else` of the conditional, the script will calculate the point's perspective position and size. Once you have this information, you can check if the point is still on Flash's Stage. Because the *y*-position of the items' will never change, we use the value in the `menu_item_info` array to calculate the perspective corrected *y*-position.

```
// object is not behind viewer but still may be off
→ the screen,
// so don't make visible yet
// calculate perspective ratio
perspective_ratio = D / (D + tz);
// calculate the position of point on computer screen
perspective_x = origin_x + tx * perspective_ratio;
perspective_y = origin_y - menu_item_info[j].y *
→ perspective_ratio;
// perspective scale of object
perspective_scale = menu_item_size * perspective_ratio;
```

We use the same method as before to determine if the point is on Flash's Stage. The right-most point of the object is used to determine if it has gone off the left side of the Stage, and the left-most point is used to determine if the object has gone off the right side.

```
// check if object is off the side of the screen
if (((perspective_x + perspective_scale) < 0) ||
→ ((perspective_x - perspective_scale) > origin_x*2)))
{
// object is off the screen, make invisible
_parent["menu_item" + j]._visible = false;
}
else
{
// other actions
}
```

If the script has gotten this far, it means the object is somewhere in front of the viewer. If the object is still on the Stage then you can set its visibility to `true` and carry on with setting its properties to render it.

```
// the object is on the screen and in front of the view,
→ make visible
_parent["menu_item" + j]._visible = true;
// update position of movie clip on stage
```

```
_parent["menu_item" + j]._x = perspective_x;
_parent["menu_item" + j]._y = perspective_y;
// update size of movie clip
_parent["menu_item" + j]._xscale = _parent["menu_item" +
→ j]._yscale = perspective_scale;
// set the depth of the clip based on its z-position
_parent["menu_item" + j].swapDepths (-tz);
```

After you close a few curvy braces, you can test your demo.

A Variation

There are plenty of ways to add even more interactivity to your menu system or gallery showcase. In this variation, we'll let viewers use their mouse to navigate the menu items.

Open demo_three2.swf from the CD-ROM.

Moving the mouse up and down the Stage lets viewers to go in and out of the world; moving the mouse to the sides lets them slide along the world (**Figure 2.41**).

Figure 2.41 *Try moving the authors' names with your pointer.*

Open demo_three2.fla from the CD-ROM.

The engine we've written so far is easy to adapt to this small variation. In fact, we only need to change the few lines that calculate the translation

increments, which were previously created from the key presses. This time, the increments will be calculated for the mouse position.

It's key that your navigation controls are intuitive. I chose to have the world slide along the *x*-axis in the opposite direction of the pointer. So, when the pointer is on the left side of the origin, the translation increment should be negative so the items move to the left. When on the right, the increment should be positive so the items move to the right. These simple statements are coded like this:

```
translation_x = origin_x - _parent._xmouse;
```

If the pointer is above the origin, then the translation increment for the *z*-axis should be negative so that the items slide toward the viewer. The items should slide away from the viewer when the pointer is below the origin, so the translation increment should be negative. This is expressed with the following line:

```
translation_y = _parent._ymouse – origin_y;
```

When you replace the previous method with these lines, you'll notice that the items move too fast. The lines calculate large numbers because the Stage is measured in pixels. To make the world move more slowly, divide the difference of the pointer and origin position by a certain number. The number depends on what kind of control you want to give the viewer. Experiment until you find something you like. Also, to keep from having too many numerical values in our code, I first set a variable in the load clip event to regulate the translation increments.

```
// used to regulate the translation increments based on the
→ mouse position
trans_regulate = 20;
```

With this variable defined, you can replace the two lines that calculated the translation increments with these two lines.

```
// change the x- and z-axis translation of the items based
→ on mouse position
translation_x = (origin_x - _parent._xmouse) /
→ trans_regulate;
translation_z = (_parent._ymouse - origin_y) /
→ trans_regulate;
```

That's it!

Open demo_three3.swf from the CD-ROM.

Here's another variation—one of the multitude of options left to explore. This one lines up all the items in a row down the *z*-axis and lets the viewer travel through them. The Up and Down keys control the *z*-axis movement and the pointer controls the slight movements along the *x*- and *y*-axes (**Figure 2.42**).

Figure 2.42 *You can create a multitude of variations on a simple theme.*

More Examples

I've included a few more examples of 3D in Flash on the CD-ROM. One demo is a rotating 3D menu with a sample site layout; another is a terrain generator with a ball rolling around on the terrain. I hope you enjoy exploring them. You'll find these files in the Extra Examples sub-folder in the chapter 2 folder.

Closing Thoughts

Congratulations: You're now in a position to create new and awe-inspiring effects. No doubt you'll have 3D questions or issues that you'd like to discuss with others in the Flash community. A good place to start is Were-Here (www.were-here.com). You can interact with some of the best Flash demo creators around at Were-Here's math forum; members have been known to discuss just about every aspect of 3D.

Rendered 3D

The second half of this chapter will take you on a tour of the Rendered 3D world. I hope this tour gives you a glimpse of the many possibilities that 3D and Flash provide, and I hope it offers you some insight and inspiration for your own future projects.

Third-Party 3D Software

Although there are many 3D applications on the market, two programs are particularly well suited for preparing 3D objects for use in Flash: Swift 3D from Electric Rain (www.swift3d.com) and Vecta3D from Idea Works (www.vecta3d.com). These two inexpensive programs are a boon for Flash developers. They let you modify and animate 3D objects, import files from more full-featured (and more expensive) 3D programs like 3ds max (formerly known as 3D Studio Max) or LightWave 3D, and import vector shapes from illustration applications like Macromedia FreeHand. More importantly, they're the only two applications I know of that can export SWF files. Both programs export SWFs as a series of vector images, which can be brought into Flash in individual keyframes (much as you see in cel animation).

Both Swift 3D and Vecta3D come in standalone versions for Windows and Mac. Swift 3D also ships as a plug-in for 3ds max (Windows) and LightWave 3D (Mac and Windows); Vecta3D comes as a Windows-only plug-in for 3ds max.

Aside from this, the similarities between the two programs are few. Each has a different workflow.

Swift 3D

Swift 3D employs a familiar 3D interface (**Figure 2.43**).

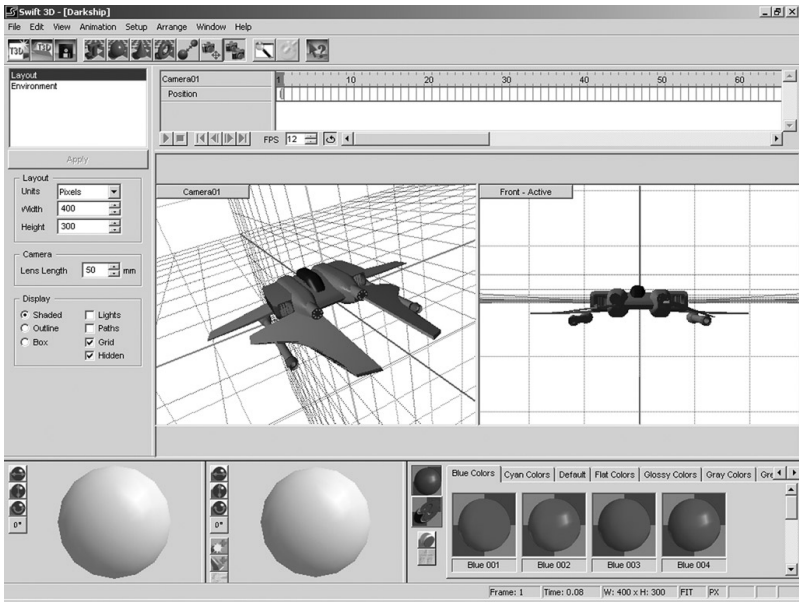

Figure 2.43 *The Swift 3D interface.*

Swift 3D lets you create animations in three ways. You can import a 3D mesh made with 3D Studio Max (.3DS); import vector shapes from programs such as FreeHand (.EPS) and convert them to 3D objects via Swift 3D's editing tools; and create 3D objects (including primitives such as spheres, cones, and toruses, as well as text) from scratch.

The program's easy-to-use interface offers controls for specifying colors, bevel types, depth, rotation, and position. Swift 3D comes with predefined color schemes and animation sequences that you can apply via drag and drop. It also offers an interesting rotating camera feature that lets you animate the camera instead of the object. (A good tutorial for this feature is at www.erain.com/tutorial.asp.)

Vecta3D

The Vecta3D interface is quite different from Swift 3D's (**Figure 2.44**).

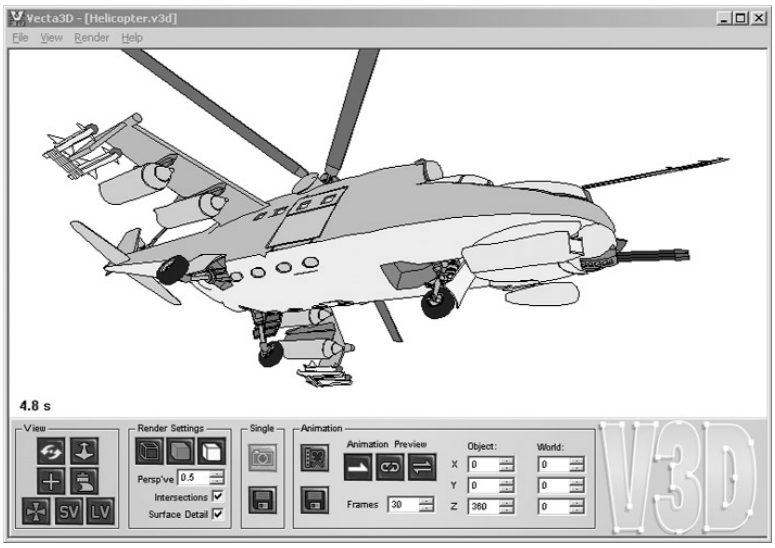

Figure 2.44 *The Vecta3D interface.*

Vecta3D doesn't let you create 3D objects from scratch and doesn't offer preinstalled animation schemes, so it requires a different workflow. The program displays a 3D object as a group of dots. To see the object with fills, you must render it.

Showcase 1: Spaceship

My first showcase is an animation of a spaceship that dives from the sky, levels out, fires a few shots, and then banks left and disappears from the screen. This animation is a trailer to a project of mine called the Stargate Interface.

To see the animation, open Spaceship_final.swf from the CD-ROM (**Figure 2.45**). You can also see the completed project at www.sdflash.net/trailer.htm.

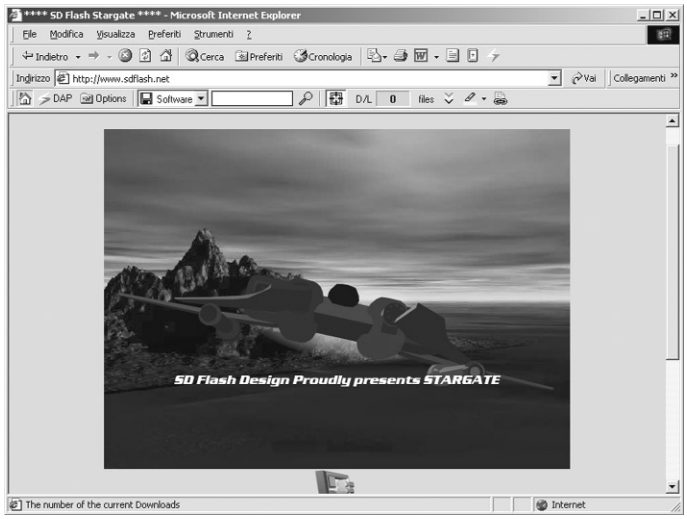

Figure 2.45 *The Stargate Interface.*

The animation is fairly straightforward with very little ActionScript—proof that you don't need to write tons of code to create cool 3D.

Storyboarding the Animation

Storyboarding your 3D animation is a crucial step to take before you begin your project. Seeing your project laid out in frames will help you analyze the action and decide which parts should be created with a rendering program and which should be created with Flash. This will save you lots of time and frustration later on.

In general, limit the number of frames that you render in a third-party application. This will help you minimize your Flash file size. (Each frame in a rendered animation is a separate drawing, which can quickly increase file size.)

Here's the storyboard of my animation (**Figure 2.46**):

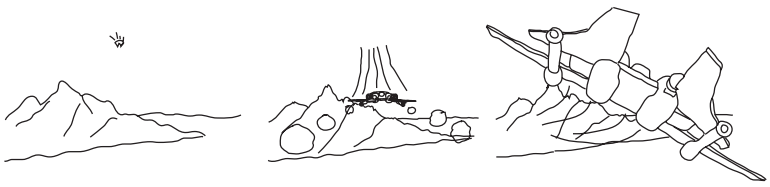

Figure 2.46 *Hand-drawn layout of the animation sequence.*

It's clear from the storyboard that Flash will run into problems when creating the spaceship's turns: specifically, when the spaceship changes from an almost vertical position to a frontal view, and later, when it banks to the left.

Flash can't create these different views because it can't change the visible part of the spaceship. With that in mind, here's how I divided the animation tasks between Swift 3D and Flash:

1. Spaceship turns from vertical to level flight. (Swift 3D)
2. Spaceship, while turning, dives towards ground level and moves towards viewer. (Flash)
3. Spaceship shoots at viewer. (Flash)
4. Spaceship banks left. (Swift 3D)
5. Spaceship, while banking left, moves toward the top right angle of the screen and disappears. (Flash)

Let's discuss the sequence that was created within Swift 3D.

Starting the Project in Swift 3D

Open Spaceship_Final.t3d from the CD-ROM (**Figure 2.47**). This is a Swift 3D file; you'll also find a demo version of Swift 3D on the CD-ROM.

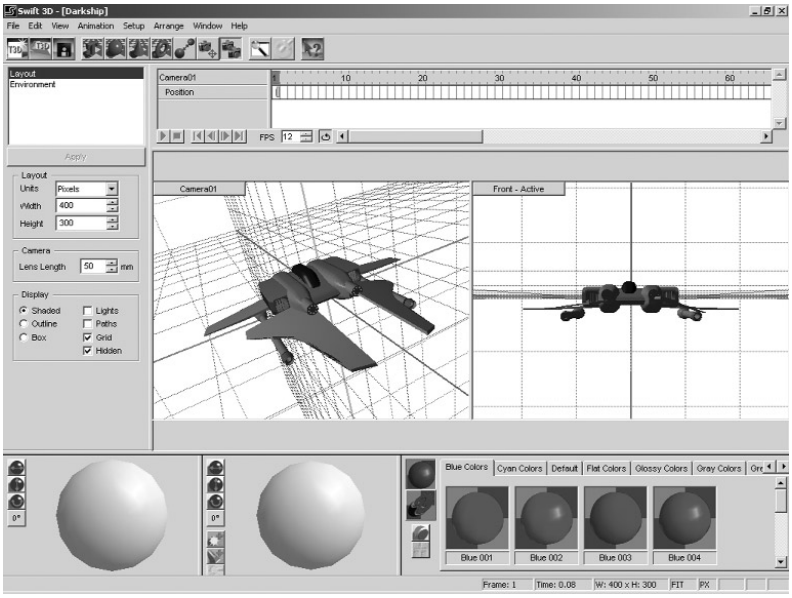

Figure 2.47
The final Swift 3D file that we'll export to Flash.

Press the Play button on the timeline to preview the animation. You'll see the spaceship go from vertical to level flight and then bank to the left. This covers the exact movement that I specified in the list above.

Note that the active viewpoint is called camera01. When Swift 3D imports a 3D Studio file, it inherits some of the 3D Studio preferences, such as camera views, lights, and colors, If you're a 3D Studio user, this is a big advantage because you can prepare your object completely before switching to Swift 3D.

The Internet offers plenty of free 3D meshes for users who don't have the time, patience or ability to create their own. Check out www.3dcafe.com or www.highend3d.com. I found the spaceship model for my showcase on 3dcafe.com.

As you can see in the timeline below, my animation has three keyframes (**Figure 2.48**). I used Swift3D's Object Trackball to flip and rotate the model into each position.

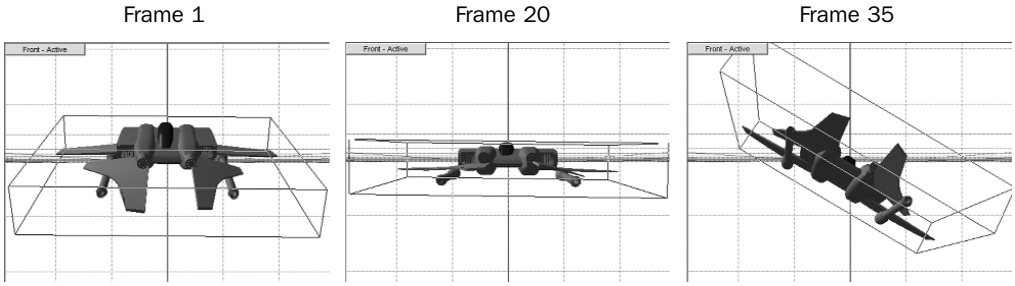

Figure 2.48 *Position of the spaceship in the three keyframes.*

After I set up my animation, it was time to export it as a SWF file. There are two important factors to consider during the export process: the number of polygons the renderer needs to draw each frame, and the type of fill you chose. Using more complex fills will increase your object's detail, but it will also create larger file sizes for each frame. **Figure 2.49** clearly illustrates this. You might need to experiment to find the optimal resolution.

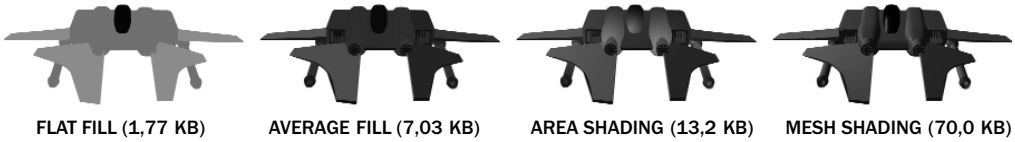

Figure 2.49 *The same object exported with various fill options.*

Open Spaceship_rendered.swf from the CD-ROM to see the exported SWF.

Flash is not the place for complex 3D objects. Import a 70,000-polygon object into your Flash project and you'll see what I mean—that is, if you can get the animation to load! Finding the right balance between file size and image quality requires patience: Sometimes you'll have to render an object several times before you find the optimal solution.

Finishing the Project in Flash

Once I brought the file intro Flash, my goal was to do a kind of cloning—to take the 3D genes from the Swift 3D file and clone them with the powerful Flash genes.

Open Spaceship_final.fla from the CD-ROM (Figure 2.50).

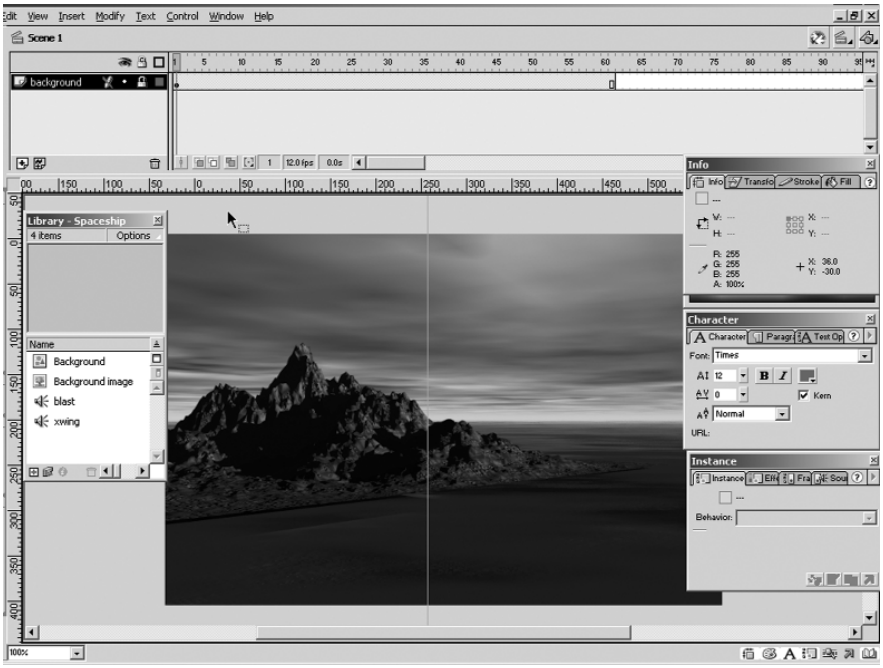

Figure 2.50 *Continuing the project in Flash.*

Importing the animation. In Flash, I began by creating a movie clip called ClipSpaceship to contain the rendered 3D object. I put the rendered animation on the movie clip's default layer, renamed the layer Spaceship and imported the rendered SWF file from Swift 3D. This brought 34 keyframes into the layer (**Figure 2.51**).

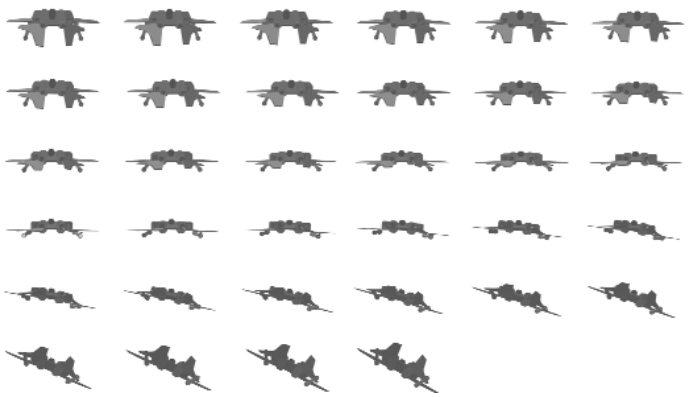

Figure 2.51 *The spaceship animation sequence.*

Creating the engine exhaust and laser shots. Next, I made a graphic symbol (a circle with a radial fill) that I used to create to some engine exhaust and laser shot effects for my movie. I placed this Laser symbol in the Engine Exhaust layer in the ClipSpaceship movie clip. It was important for this layer to be below the Spaceship layer in the timeline because it had to look like it was behind the spaceship. I scaled the symbol to make it look like realistic engine exhaust emitting from the spaceship (**Figure 2.52**). I was going for the effect seen in the Star Wars spaceships—that clear white-blue glowing light from the engines, just like on the Millennium Falcon with Han Solo and Chewie.

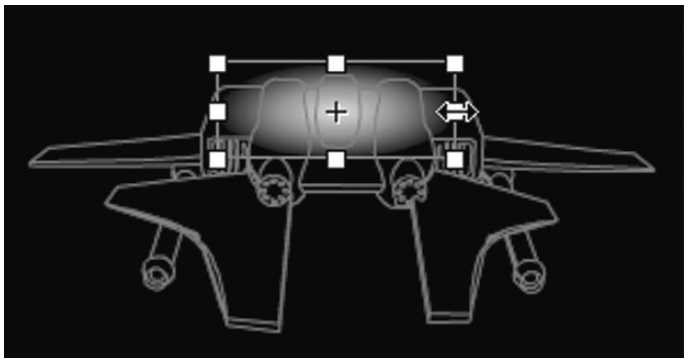

Figure 2.52 *Scaling the engine exhaust to fit the spaceship.*

The spaceship's movement is pretty linear over the first 20 frames of the movie clip, so I was able to use a normal tweened animation to make the exhaust follow the spaceship. In the last 15 frames, the spaceship moves more irregularly, so it becomes more difficult to control. Therefore, I made a new keyframe for each of the remaining frames, scaling and rotating the laser symbol into the right position in each frame.

Figure 2.53 shows how the timeline and animation appeared at this point.

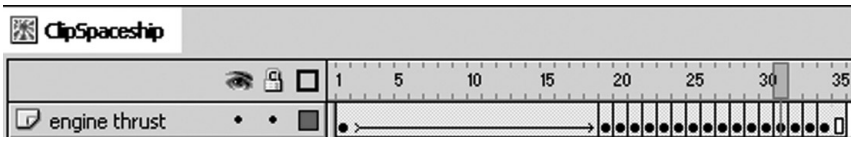

Figure 2.53 *The animation of the engine exhaust is a combination of tweens and keyframes.*

Next, I created the lasers that would shoot at the viewer. I created a movieclip symbol called ClipLaserRight, then positioned an instance of the Laser symbol at the center of the Stage and resized it (height: 13.6. width: 13.6). I made a new keyframe on frame 4 and resized the laser again (height 272, width: 272) and changed the *x*-position to -128. As a final step, I created a third keyframe on frame 7 and modified the laser again (height: 340, width 340, *x*-position: 176). By moving the *x*-position, I was able to achieve a more realistic effect in the final movie.

I then attached the following ActionScript to frame 7:

```
gotoAndPlay(1);
```

To make the laser for the left side. all I had to do was duplicate it and call the copy ClipLaserLeft. I then edited this duplicated movie clip, changed the *x*-position of the laser shot on frames 4 and 7, and changed the values in the Info panel from the current negative values to the equivalent positive values. This made the laser shot move in the opposite direction.

Adding sound. The final step was to add the sound of the laser cannon. Note that I added sound to only one of the two laser shot movie clips—because they fire simultaneously it's not necessary to insert sound into both. I prefer to use ActionScript to control my sounds, as Flash 5's Sound object offers excellent control. (For more information on using audio with Flash, see Chapter 4.)

I normally define all my sounds on the first frame of the main timeline of the movie. This makes it easier when debugging code. I made the sound symbols in my library available to ActionScript by setting the linkage option for each sound symbol. The identifier name that you give a sound symbol in the Linkage menu works like the instance name of a movie clip: You refer to it when you call it from within ActionScript. To keep things simple, I prefer to use the same name as the symbol has in the library.

I put the following script into the first keyframe in the timeline:

```
_root.blastSound.start();
```

Wrapping up the spaceship clip. Now back to the final preparations on the spaceship movie clip. I inserted a new layer above the Spaceship layer and called it Lasers. The lasers will fire their shots when the Spaceship levels out at horizontal flight, which means that they must appear on frame 19 of the movieclip timeline.

I created an empty keyframe in frame 19 and positioned the two lasershot movie clips exactly above each laser cannon. Here's how frame 19 looked at this point (**Figure 2.54**):

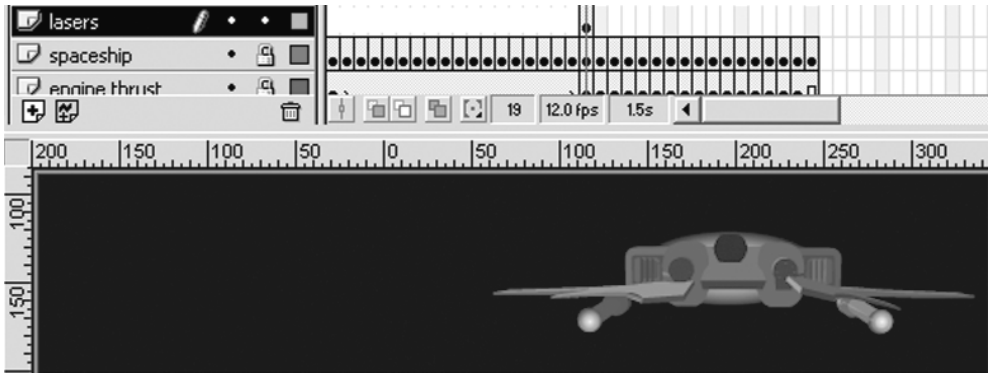

Figure 2.54 *The spaceship is now locked and loaded.*

Lastly, I inserted a few simple frame actions to help control the animation later on. I created a new layer called Frame scripts and created two empty keyframes, one on frame 19 and the other on frame 35. I inserted a `stop()` action in each one. (I know it may be difficult to see the logic in these two actions, but when I show you how I put together the animation on the Stage, it will become much clearer.)

Putting it all together. Now that the Spaceship movie clip was complete, it was time to put it all together on the Stage. In Scene 1, I created a new layer above the Background layer and named it Spaceship. To help me more precisely lay out the animation, I used Flash 5's ruler and guide tools. (If you're familiar with FreeHand, you've probably used these tools before.)

I positioned a guide centrally on the stage and locked it (**Figure 2.55**):

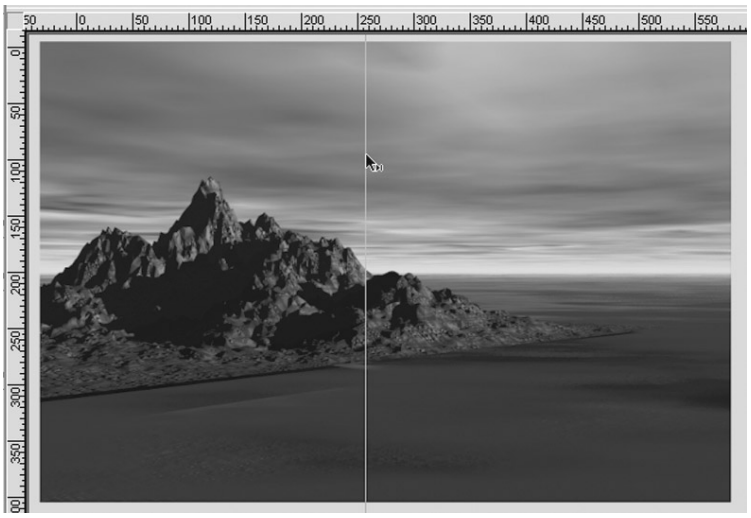

Figure 2.55 *Dragging and positioning a new guide on the Stage.*

With the spaceship layer active, I selected the ClipSpaceship movie clip from the library, dragged it onto the Stage and named it darkship_mc in the Instance panel.

I always append the names of my movie clip instances with `_mc`. This makes it easy to tell whether I'm referring to a movie clip or a variable. It also makes code more readable, which is a big help during debugging.

I scaled the movie clip to 10 percent of its original size, and then placed the spaceship's center exactly on the guide. I also put the spaceship close to the top of the screen setting the coordinates like this:

```
X position: 237.1
Y Posistion: 0.1
```

I made a new keyframe on frame 20. This is where the spaceship finishes its dive and reaches horizontal flight level.

When I putted the spaceship movieclip on the Stage, only the first frame of it is visible. This made it pretty difficult to position the spaceship because I had to imagine what frame of the spaceship movieclip would be visible at that specific moment of the animation. I have a trick for situations like this: I selected the ClipSpaceship movie clip and edited it. Then I created a new layer at the top of the movie clip timeline. I copied the spaceship from frame 19 in the Spaceship layer and pasted it into the first frame of the new layer.

Back to the Stage in Scene 1. Here's how the spaceship movie clip looked at this point (**Figure 2.56**):

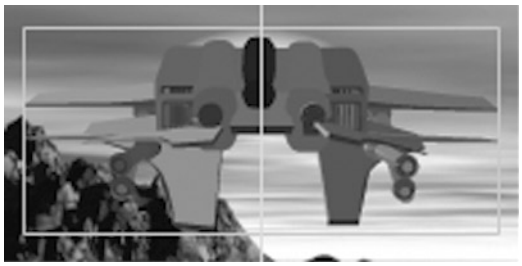

Figure 2.56 *Special editing view of the spaceship.*

In frame 20, I scaled the spaceship to 65 percent and changed the coordinates to this:

```
X position: 128.2
Y position: 54.0
```

I selected motion tween in the frame panel and kept the other default settings. Then I created the third keyframe on frame 40, scaled the spaceship to 150 percent, and changed the coordinates to this:

```
X position: -38.9
Y position: 43.9
```

Again, I selected motion tween and kept the other default settings. At this point, I needed to insert an action to tell the spaceship to move onto the first frame after the Stop command that I inserted in the movie clip on frame 19. I created a new layer in Scene 1 and named it Frame scripts.

I made a new keyframe on frame 1 and inserted the following script that defines the sounds used in the movie:

```
blastSound = new sound();
blastSound.attachsound("blast");
xwingSound = new sound();
xwingSound.attachsound("xwing");
```

To play the spaceship's sound, I created a new keyframe on frame 6 and inserted the following script:

```
xwingSound.start();
```

To control the spaceship, I insert an empty keyframe on frame 40 and added the following script:

```
_root.darkship_mc.gotoAndPlay(20);
```

This makes the movie clip begin the part of the animation where the spaceship banks to the left.

To ease the positioning of the spaceship in the next keyframe (frame 54), I used the same method as before. I replaced the graphic in frame 1 in the spaceship movie clip with frame 35 of the spaceship layer (**Figure 2.57**).

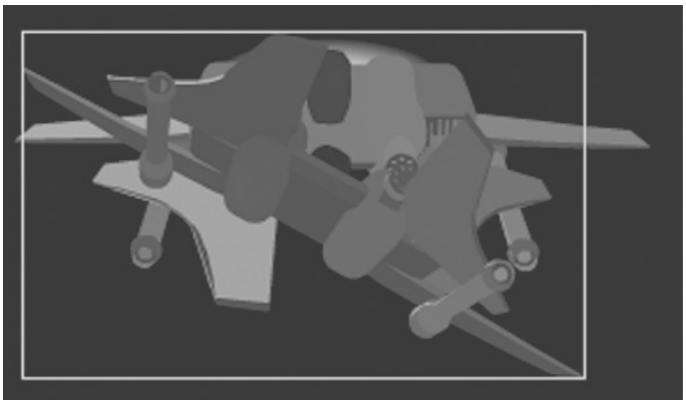

Figure 2.57 *Dragging and positioning a new guide on the Stage.*

Back at Scene 1, I created a new keyframe in the spaceship layer on frame 54, scaled the spaceship to 300 percent, and changed the coordinates to this:

```
X position: -252.8
Y position: -347.0
```

I created the last keyframe on frame 60 and gave the spaceship the following coordinates:

```
X position: -9.8
Y position: -499.0
```

That's it. Although this showcase may not seem that complex from a coding point of view, there are many other aspects to consider.

This showcase has taught you the basics, but I'm not finished yet. I want to show you how you can incorporate the principles of rendered 3D as seen in this showcase into a larger project.

Showcase 2: Stargate Interface

The science fiction movie Stargate inspired me to create the following project, called the Stargate Interface. This interface is a prototype, but it shows what you can do with a little more ActionScript and a lot of imagination. By the time this book hits the shelves, the project should be published on my site, www.sdflash.net.

The site's concept is built around a round gate with eight symbols, a Star Navigator that searches and finds the symbols for a specific code sequence, and a menu with six options. Each menu option has a code sequence of four symbols that must be activated by the gate to jump to one of the menu options. The symbols are individualized by the Star Navigator, which finds each symbol and tells the gate to lock on it. Once all the symbols in the code sequence have been locked, the movie takes you to the option you've selected.

Open Stargate.swf from the CD-ROM to see the final file.

All the 3D objects in this prototype were made with Swift 3D. I designed the Stargate as a flat vector drawing in FreeHand and then imported it into Swift 3D. All of the symbols on the gate made the output file kind of big, but I decided to go with it anyway because I really liked the result.

The interface is constructed around four movie clips that work together (**Figure 2.58**).

Figure 2.58 *The basic navigation setup.*

Initialization

All navigation begins in the Menu Navigator movie clip, where the viewer selects a destination. Each destination has its own code sequence of four symbols, which have been defined in six arrays at the beginning of the movie:

```
newsarray = new Array(5, 7, 4, 3);
aboutarray = new Array(3, 5, 6, 7);
flasharray = new Array(4, 1, 5, 3);
cdromarray = new Array(2, 5, 7, 3);
portfolioarray = new Array(2, 6, 1, 3);
contactarray = new Array(1, 5, 2, 6);
```

These arrays control the entire animation scheme. In the beginning of the movie, I defined some variables that are also crucial in controlling the interface:

```
//sets the value of the first symbol in the code sequence;
var symbol0 = 0;
//sets the value of the second symbol in the code sequence;
var symbol1 = 0;
//sets the value of the third symbol in the code sequence;
var symbol2 = 0;
//sets the value of the fourth symbol in the code sequence;
var symbol3 = 0;
//tells the stargate which of the four symbols in the code
sequence must be set; var step;
```

Selecting from the Menu

When a viewer clicks a button in the Menu Navigator, the two trace arrows search for the destination selected. A variable is set upon click; this tells the arrows where to stop. When they find the right destination, the following script is executed:

```
//first we check if the variable is set to the right value;
if (this.menuitem == "news") {
//Then we assign the values of the right array to the 4
→ symbol variables;
for (counter=0; counter<=3; counter++) {
root["symbol"+counter] = _root.newsarray[this.counter];
}
```

```
//tell the stargate to lock for the first symbol in the
→ code sequence;
root.step = 1;
//tell the star navigator to look for the first symbol in
→ the code sequence;
root.starnav_mc.counter = 0;
//activate the star navigator;
root.starnav_mc.play();
//set the right status message;
root.status_mc.message = "Symbol tracking  initiated......";
stop ();
}
```

The Star Navigator

As you can see in the code, we have activated the Star Navigator. In the Star Navigator I've defined a function that's being reused every time the trace arrows pass over a symbol. Here's the function:

```
//lock is used to define which symbol the trace arrows are
→ passing over;
function funcStarlock (lock) {
//check if the symbol is the right one and if step has the
→ right value;
if (_root["symbol"+counter] == lock && _root.step==1) {
//set the status message;
root.status_mc.message = "Symbol "+_root.step+"
→ located......";
//activate the stargate movieclip;
root.stargate_mc.play();
// activate the flashing light over the symbol;
setProperty (_root.starnav_mc["lock"+lock+"_mc"], _
→ visible, true);
root.locksound.start();
counter++;
stop ();
//this repeats itself for each symbol;
} else if (_root["symbol"+counter] == lock && _
→ root.step==2) {
root.status_mc.message = _ root.status_mc.message+"\nSymbol
→ "+_root.step+" located......";
```

```
root.stargate_mc.play();
setProperty (_root.starnav_mc["lock"+lock+"_mc"], _
→ visible, true);
root.locksound.start();
counter++;
stop ();
} else if (_root["symbol"+counter] == lock &&
→ root.step==3) {
root.status_mc.message = _root.status_mc.message+"\nSymbol
→ "+_root.step+" located......";
root.stargate_mc.play();
setProperty (_root.starnav_mc["lock"+lock+"_mc"], _
→ visible, true);
root.locksound.start();
counter++;
stop ();
} else if (_root["symbol"+counter] == lock &&
→ root.step==4) {
root.status_mc.message = "Symbol "+_root.step+"
→ located......\nLaunch sequence activated!!!!!";
root.stargate_mc.play();
setProperty (_root.starnav_mc["lock"+lock+"_mc"], _
→ visible, true);
root.locksound.start();
counter++;
stop ();
}
}
```

Each time the trace arrows reach a symbol, I put a reference to the code and assign a value for the lock variable like this:

```
funcStarlock(1);
//1 is the value that are being assigned to the lock
variable
```

Whenever a symbol is found, the Stargate (stargate_mc) is activated. Again, I use the powerful `Function` action. This time I have two different functions: one for the first three symbols, and the other for the last symbol in

the code sequence. I did this because the actions executed on the first three and the last symbol are different. Here are the two functions:

```
//Function for the first three symbols in the code sequence;
function funcStargate (symnum,symvalue,stepnum,clipname) {
if (_root["symbol"+symnum] == symvalue && _root.step ==
→ stepnum) {
//activate the right lock arrow on the Stargate;
root.stargate_mc[clipname+"_mc"].gotoAndPlay(10);
root.gatelocksound.start();
//restart the Star Navigator;
root.starnav_mc.play();
root.step++;
stop ();
}
}
//Function for the final symbol of the code sequence;
//notice the differences from the other function;
function funcStargate2 (symnum,symvalue,stepnum,clipname) {
if (_root["symbol"+symnum] == symvalue && _root.step ==
→ stepnum) {
root.stargate_mc[clipname+"_mc"].gotoAndPlay(10);
root.gatelocksound.start();
//jump to the beginning of the travelsequence;
root.gotoAndPlay("travelstart");
root.step++;
stop ();
}
}
stop ();
```

In the stargate_mc these functions are called when needed, and provide the right arguments. Their positions depend on the position of the symbols on the Stargate compared to the lock arrows next to it. The actions inserted into one of the frames look like this:

```
//calls the function for the first three symbols;
funcStargate(0,1,1,"topnav");
funcStargate(1,7,2,"rightnav"); //idem
funcStargate(2,5,3,"bottomnav"); //idem
//calls the function for the last symbol;
funcStargate2(3,3,4,"leftnav");
```

This ends the code sequence part. Now it's time to start the travel sequence.

The Travel Sequence

In the travel sequence, the star travel is mapped, similar to using a GPS unit in a car. You choose a destination and the computer maps your route. I made six destinations and 14 hubs that you can pass through on the way to the destinations. In a movieclip called mapping_mc I made six routes, one for each destination (**Figure 2.59**).

Figure 2.59 *The mapping_mc that maps our star travel.*

In the function funcStargate2, I send the playhead in the main timeline to a label called travelstart. In this frame I lay out the groundwork for the mapping sequence. Again, I defined a function earlier in the timeline that sorts out the route. The function script looks like this:

```
var destination;
var label="route";
function funcTravelstart () {
//sets  destination to a random integreter between 1 and 6;
destination = math.floor(math.random()*6)+1;
//set destination to a string that is made of a combination
→ of the variable label and itself;
destination = label+destination;
}
```

The function is called in the main timeline, and sends the playhead in the mapping_mc movieclip to the label that the function specifies. Here's the script:

```
funcTravelstart();
root.mapping_mc.gotoAndPlay(destination);
```

The mapping takes place, and off we go. At least that's my idea—I haven't gotten any further. For this reason, my little tour of the Stargate prototype finishes here. I hope this not only gives you a practical example of how you can combine rendered 3D with ActionScript in a real project, but also inspires you to create your own.

By the time this book hits the shelves I will have finished the entire project. You'll be able to go to www.sdflash.net and see which path I took toward the stars.

Conclusion

We have now reached the end of the line. I hope you've gained some knowledge about working with rendered 3D, and have learned how to create some nice visual effects to make your movies look more realistic. Happy 3D-ing!