1

Understanding Flash3D

Flash programmers have always enjoyed a freedom of expression unparalleled by other programming platforms. And with the release of CS4, Adobe has propelled that freedom of expression into the 3rd dimension.

But 3D didn't start with AS3. Flash developers were experimenting with 3D long before. And applications like Papervision3D formalized these endeavors into a robust object-oriented class structure. The acceptance and popularity of Papervision3D has become a driver of change in the Flash developer community. But underneath the 300 classes that make up Papervision3D still beats the original algorithms used by the early developers of Flash 3D.

Understanding how to create 3D in Flash is essential to fully grasping Papervision3D and applications like it. As you learn how 3D engines were originally constructed, you'll gain both an insight into Papervision3D's core architecture and an appreciation of its robust structure. Its complexity will fade into a set of fundamental 3D algorithms.

Papervision3D, at its core, is a perspective projection engine, where projection simply means transforming a 3D "object" space into 2D Flash x and y screen space. And surprisingly, the whole process hinges on one equation: a perspective scaling equation derived from Thales Theorem:

```
T = scale = focal length/(focal length + z)
```

In the equation above, T is the perspective scale, and z is the distance from the projection plane. The focal length (or calculated "screen" location) determines the amount of perspective provided in the view. You use this equation to create an illusion of depth by scaling 2D surfaces.

In this chapter, you use Thales Theorem to create a simple 3D engine in just 19 lines of code. Then using Flash CS4 you rebuild your engine in just 13 lines of code. You cover the big 3: translation, rotation and scaling. And applying what you've learned you create a 3D torus worm, carousel, and image ball. Finally, you cover the basics of making believable animation, and how to turn a timeline animation into pure ActionScript.

But before you get started it's important that you understand the coordinate system you'll be rendering your objects in. It's a little different to the way you learned it in math class, and well worth reviewing.

3D Coordinates in Flash 10

Understanding the Flash coordinate system is vital to rendering in Papervision3D or CS4. With the transition to Flash Player 10, every ActionScript display object has a z property. Adding a 3rd dimension allows an object to move towards and away from the viewpoint of the user using perspective scaling.

As a side note, this extra dimension is easily handled using a 3D transformation matrix that incorporates the three fundamental transformations found in all physical systems, such as translation, rotation, and scaling. Every 3D object within Papervision3D has a Matrix3D. The Matrix3D class supports complex transformations of 3D geometry, such as rotation, scaling, and translation.

But unlike other 3D systems, especially those you might have met in math class, the y and z-axes are reversed as shown in Figure 1.1.

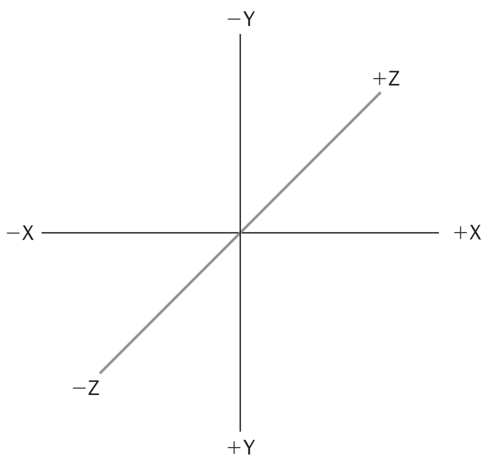

Figure 1-1

From the figure you can see that:

- x increases as you move to the right along the x-axis
- y increases as you move down along the y-axis
- z increases as you move away from the viewpoint.

In addition, in Flash 9 the coordinate origin (0,0) is not located at the center of the screen, but at the upper left corner of your screen. This becomes an issue when working with the asymptotic vanishing point, and you'll learn how to adjust for this later in this chapter.

Building a 3D Flash 9 Engine in 18 Lines of Code

In this example, you're going to build a 3D engine based on perspective scaling in just 19 lines of code. Before Flash 10 there was no native support for 3D and all you had were x and y coordinates to play around with. So you needed to add another dimension to get 3D (a z-axis). The trick to creating a z-axis was to use perspective scaling. Which means that as an object moves away from you it gets smaller and as it moves towards you it gets larger. But we need to quantify this idea (make some math out of it) so we can program it. And that come from Thales Theorem.

Applying Thales Theorem

The Greeks are famous for their geometry and Thales (a Greek Philosopher) was the first to propose similar triangles. From the concept of similar triangles you get simple linear projection: which is the heart of a Flash 3D engine. Of course perspective drawing really didn't take off until the Renaissance, and now again in Flash 3D.

Imagine that you're in your home looking out the window. As you approach the window objects outside look larger and as you get farther away from the window, objects outside look smaller. Your distance from the window is called your focal length, the window is your projection plane (or viewport), and your eye is the vanishing point.

Now remain stationary, this fixes your focal length; watch outside as objects move closer to and farther away from your window. As a bird flies closer to the window it looks larger and as it flies away it looks smaller. This is your z-axis: the distance between the outside object and your windowpane.

The equation that governs this behavior is:

```
T= scale = focal length/(focal length + z)
```

where T equals "one" when the outside object is at the window and "zero" when your object (bird) flies far away (off to infinity, also called vanishing point). This equation works well for Flash and is illustrated in the graphic below as a Blender monkey peers at a projection plane. Focal length is the distance from the vanishing point (monkey's eye) to the projection plane (see Figure 1.2).

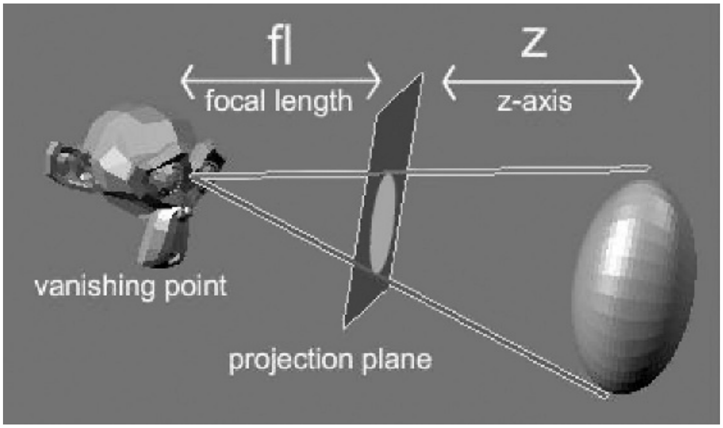

Figure 1-2

Creating the illusion of depth (or 3D) using perspective scaling is sometimes referred to as 2.5D.

Another term used to describe perspective projection is 2.5D. The term is usually used with computer graphics, especially video games, where a computer system uses 2D computer graphics to visually simulate 3D computer graphics. In Flash, you use the z perspective scale to create a perspective projection onto the Flash x, y screen.

Deriving the Scaling Equation

You're probably wondering how the scaling equation presented above was derived. As mentioned earlier, it came from Thales' idea of similar triangles. The figure below shows two nested (or similar) triangles. Small h is the size of an object at the computer screen and large H is the actual size of the object beyond the screen. And as described earlier fl (or focal length) is the distance from your computer screen to your eye, and z is the distance on the other side of your screen to your object as shown in Figure 1.3.

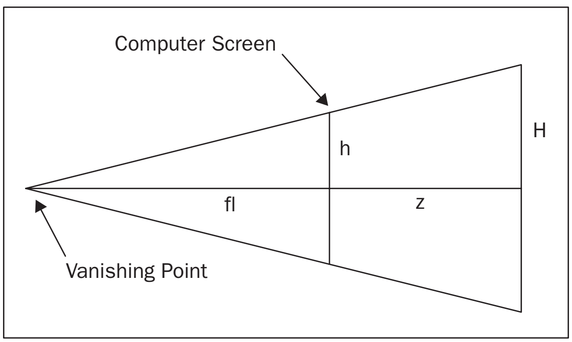

Figure 1-3

From the law of similar triangles, it follows that

$$\frac{h}{fl} = \frac{H}{fl + z}$$

By cross-multiplying, you get the scaling equation:

$$Scale = \frac{h}{H} = \frac{fl}{fl + z}$$

That's all there is to its derivation, but examining the equation reveals an interesting phenomenon: a singularity (blow up) in z. As z approaches –fl your equation's denominator goes to zero and your scale blows up to infinity. And as shown in the figure below after you pass –fl your figure goes through an inversion (it flips over).

You're probably wondering how fl (focal length) is assigned. Is it actually the distance of your eye to the computer screen? No! It's a virtual quantity that was used in the derivation of the scaling equation. But that doesn't mean it doesn't have meaning. It is the focal length of your camera, and its assignment changes the appearance of what you see on your computer screen.

The figure below illustrates a number of important occurrences in the life of the scaling equation:

- ❑ When z = 0 your image scale is 1 (image is its actual size)
- ❑ When z = fl your image scale is ½ (image is half its size)
- ❑ As z approaches−fl your scale approaches infinity (image is infinitely large)
- ❑ As z approaches infinity your scale approaches zero (image has vanished)

Of the four conditions listed above, two are used most by mathematicians to quickly define the behavior of a function, the singularity (blow up point) and asymptotic behavior (as z approaching infinity). The asymptotic behavior is important here since as your object moves towards infinity it vanishes. This is referred to as the vanishing point and occurs at a specific x, y position of your screen. In Flash, that position is at the origin that occurs at the upper left corner of your computer screen. Later in this chapter you'll find out how to change your vanishing point.

This may cause you a little confusion since you've already learned that the vanishing point occurs at the eye as shown in Figure 1.2 previously. So are there two vanishing points? Definitely, but the first one that occurred at the eye was used for derivation purposes and is where your singularity occurs (shown in Figure 1.4). The second vanishing point occurs at infinity. It's the one that you'll now be most interested in working with. It determines where your 3D object goes (on the x,y screen) as it gets further away from you.

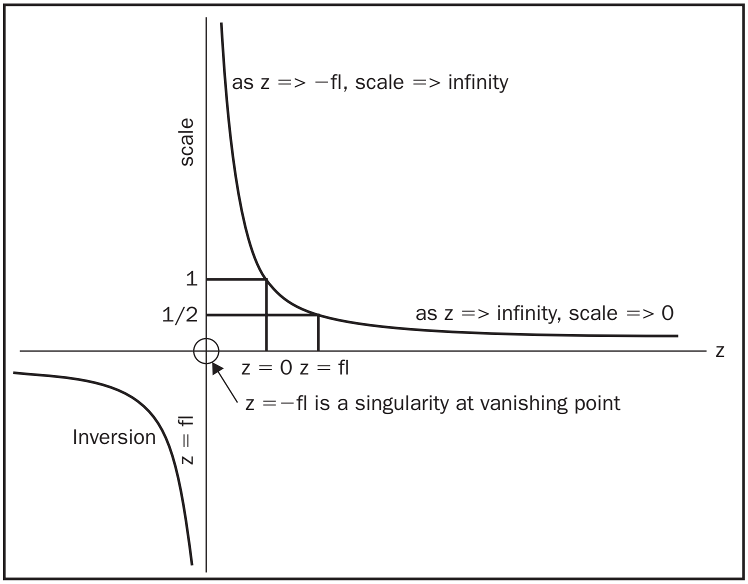

Figure 1-4

The curve above is very similar to the gravitational or electrical potential curves found in physics. And it further suggests that a size scaling force field could be constructed by taking the differential of the scaling equation. But that's beyond the scope of this book.

Rendering to the Screen

Using perspective scale you've picked up a 3rd dimension: z. But in order to render it to the Flash Stage you've got to get rid of it again. The process is called rendering and occurs by projecting a 3D object (created from perspective) to a 2D computer screen (or viewport).

Projection onto the Flash screen requires that you convert a 3D point (x, y, z) into a 2D (x, y) point. You accomplish this by using scaling to drop the z coordinate. A change in z affects the position and size of an object in 3D space as shown in the code below:

```
scale=focal length/(focal length + z);
x=x*scale;
y=y*scale;
xscale=yscale=scale*100.
```

In the code, the scale (based on focal length and z) changes the x and y position and the x and y scale, resulting in a 2.5D effect described earlier. And it's done the same way in Papervision3D. Throughout this book, you'll be drilling down into Papervision3D's classes and examining how it does what it does. Below is a code snippet taken from its `Camera3D` class.

In Chapter 2, you'll find out how to obtain the Papervision3D classes and how to drill down and examine its code. This particular code snippet is found in the `org/papervision3d/camera` folder.

```
if(screen.visible = ( focus + s_z > 0 ))
{
s_x = vx * m11 + vy * m12 + vz * m13 + view.n14;
s_y = vx * m21 + vy * m22 + vz * m23 + view.n24;

//perspective scaling in Papervision
persp = fz / (focus + s_z);
screen.x = s_x * persp;
screen.y = s_y * persp;
screen.z = s_z.
}
```

The code snippet above taken from Papervision3D's `Camera3D` class demonstrates how Papervision3D uses perspective scaling. You'll use this idea to create a 3D Flash engine in just 19 lines of code.

Coding Animation

Okay, you should now understand enough about perspective scaling to begin coding your 3D engine. In this example, you create your code in the Flash API, but many Papervsion3D developers use either Flex or Eclipse for their code development work. Flex and Eclipse offer superior code debugging and code hinting capabilities over Flash, and in many instances their use speeds your code development by a factor of 4. It's worth trying one of them out.

In this exercise, you'll create a circle that oscillates in and out of the screen in the z perspective direction. Whether using the timeline or pure ActionScript, Flash in essence is an animation machine. When using purely scripted animation, the animation process is based upon a looping (or updating) procedure shown in Figure 1.5.

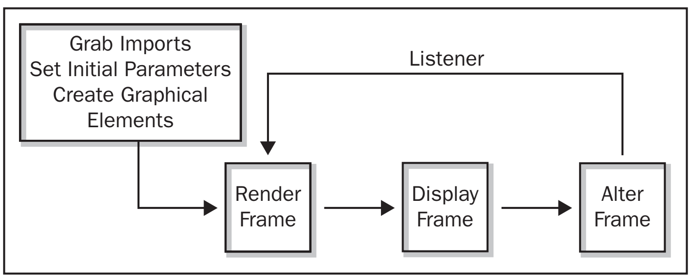

Figure 1-5

The typical flow of such a program involves first initializing the program and then using a listener to loop through a scripted animation routine. The whole process can be divided into three parts: initialize, listen and loop:

Initialize: Steps 1.3

Initializing an AS3 program can be a little overwhelming at first. The question that most people ask is how do I know what imports to bring in? There is no magical pool of knowledge that can help you. It just takes experience (or reading the documentation, or paying attention to syntax). As you work with the different Flash and Papervision3D methods you'll become experienced in spotting classes that need to be imported. For example, if you're creating a Wii game, it's obvious that you'll need to import some Wii classes.

But, which ones? It depends on which methods you're using to create your program. From experience, you'll begin to recognize and associate the right classes with their methods. Or better yet, Flex builder has an auto-complete that automatically adds the classes you need upon auto-completion of a method.

In the example below, you create a ball sprite, so it makes sense that you'll need to import the Sprite class. After importing the Sprite class, you initialize the position and angle of the ball and then set your focal length (from experience 300 works well for this example). Then in step 3, you create your ball, give it a color and set it on the stage using the `addChild` method.

1. Start by importing the Sprite class; this is where you're going to draw a ball that you will animate in 3D.

```
import flash.display.Sprite;//imports sprite class
```

2. Next declare your variable's zposition, angle, and focal length.

```
var zposition:Number = 0;//z position
var myAngle:Number =0;//Angle of ball
var fl:Number = 300; //focal length
```

3. Next create your ball and add it to the stage.

```
var ball:Sprite = new Sprite();//instantiates ball sprite
ball.graphics.beginFill(0xFF0000);//Assigns a ball color
ball.graphics.drawCircle(0, 0, 40);//draws your ball at (0,0)
ball.graphics.endFill();//ends the fill
addChild(ball);//adds the ball to the stage
```

Listen: Step 4

In AS3, the event architecture is bigger, better, and badder (in a good way). Written from the ground up, it's fast, powerful, and easy to use. It incorporates listener objects to listen for events. Since event listeners all work the same way, once you understand how to use one of them you understand them all. This is how they work:

- ❑ Call the method addEventListener to listen for an event
- ❑ Name the event you want to listen for
- ❑ Name the function you want to execute when the event occurs

So in the code below, you want to iterate your animation periodically. This can be done using the enter frame event listener or a timer. In this case, you'll listen for the enter frame event. Each time a frame event occurs the function onEnterFrame is called, as shown in step 4.

- ❑ Next create your onEnterFrame listener which loops through your equations of motion. This is the heart of all 3D engines.

```
addEventListener(Event.ENTER_FRAME, onEnterFrame);//loops equations
```

Loop: Step 5

Looping is the heart of creating a scripted animation. You're probably familiar with creating frame loops on Flash's timeline. But in Papervision3D, you don't use the timeline. Everything is done in ActionScript. But you can still create graphical animation by creating a frame ripper, or use animation script created in Flash by grabbing the animation elements as classes.

In Step 5, the animation is created by incrementing the angle on each loop. As the angle is incremented the ball oscillates and the z position changes.

- ❑ Create the function that will be looped. These are your equations of motion that govern the perspective as it is changed and converts 3D to 2D (or projects onto the viewport).

```
function onEnterFrame(event:Event):void{
var scale:Number = fl / (fl + zposition);//scale perspective
myAngle=myAngle+.1;//iterates angle
myB all.x = 300*Math.sin(myAngle)*scale; //ball orbit x
ball.y = 300*Math.cos(myAngle)*scale; //ball orbit y
ball.scaleX = scale;//scales perspective in x
ball.scaleY = scale;//scales perspective in y
zposition = 2000*Math.sin(myAngle/10);} //increments z and changes sign.
```

Upon every loop, the `myAngle` variable is iterated and the `ball.x` and `ball.y` positions oscillate sinusoidally. This creates the circling motion of the ball. In the last line of code the `zposition` variable oscillates sinusoidally as well, causing the ball to cycle in and out of the screen. By putting these two motions together your ball will spiral in and out of the screen in the z direction.

You now put it all together and run the code.

Running the Code

The code (all 18 lines of it) is listed below. Just type it in the Flash ActionScript editor and click control test. You'll see a red ball oscillating on the z-perspective axis. Not very exciting, but there are a number of concepts here that you'll use throughout the book.

```
import flash.display.Sprite;//imports sprite class
var zposition:Number = 0;//z position
var myAngle:Number =0;//Angle of ball
var fl:Number = 300; //focal length
var ball:Sprite = new Sprite();//instantiates ball sprite
ball.graphics.beginFill(0xFF0000);//Assign a ball color
ball.graphics.drawCircle(0, 0, 40);//draws your ball at 0,0
ball.graphics.endFill();//ends the fill
addChild(ball);//adds the ball to the stage
addEventListener(Event.ENTER_FRAME, onEnterFrame);//loops equations
function onEnterFrame(event:Event):void{
var scale:Number = fl / (fl + zposition);//scale perspective
myAngle=myAngle+.1;//iterates angle
myB all.x = 300*Math.sin(myAngle)*scale; //ball orbit x
ball.y = 300*Math.cos(myAngle)*scale; //ball orbit y
ball.scaleX = scale;//scales perspective in x
ball.scaleY = scale;//scales perspective in y
zposition = 2000*Math.sin(myAngle/10);} //increments z and changes sign
```

The code, though it's not Papervision3D, illustrates a number of important concepts that every 3D engine possesses:

- A frame looper/renderer
- Perspective (z-coordinate)
- Projection onto a viewport
- Primitive or basic shape
- Addition of a color (or material)

And all of this is done in just 19 lines of code. If only it had stayed this simple. Papervision3D started off with only 20 classes, now it's in the hundreds and growing. But as they say, no pain no gain.

Vanishing Point (Asymptotic Zero)

As your animation runs, the spiraling red ball approaches its vanishing point (asymptotic zero), which as mentioned earlier, is the Flash origin located at (0, 0) or upper left position of your screen. This is a little annoying, but can be easily adjusted by moving your vanishing point. This is done by adding a vanishing point coordinate value to your ball x and y positions. To center the vanishing point to the Flash stage you must import the stage class and use `stage.stageWidth/2` (which returns the x center point) and `stage.stageHeight/2` (which returns the y centerpoint) as shown here:

```
ball.x = 300*Math.sin(myAngle)*scale+stage.stageWidth/2;
ball.y = 300*Math.cos(myAngle)*scale+stage.stageHeight/2;
```

The results of this change are shown in Figure 1.6.

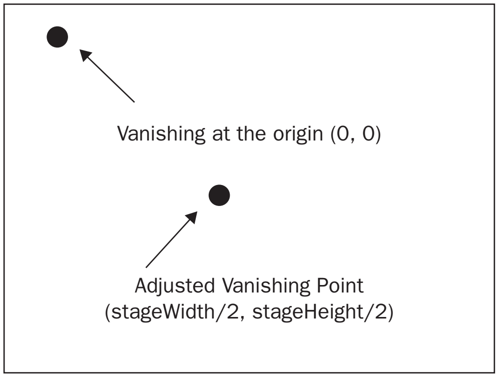

Figure 1-6

The stage class is very useful in positioning your Flash objects and will be addressed in more detail in the next chapter.

Adding a Camera

Along with 3D comes the concept of a camera. Theoretically a camera is just a point in 3D space that acts as a point of view of that space. Changing the position of your camera lets you change your view in your 3D world. But in reality your camera isn't moving, it's just offsetting everything else in relation to itself. So for example, if you want your camera to move through a maze, you have to move the maze around your camera – not the camera around the maze.

Programmatically you accomplish this by creating a camera object and giving it x, y, and z values. You use these values to adjust the position of your objects in your scene (represented by the statement below).

```
//create your camera object
camera = new Object();
camera.x = 10;
camera.y = 10;
camera.z = 100;

//loop through for all elements in your scene
scale = focalLength/(focalLength + this.z - camera.z);
this.x = (this.x - camera.x) * scale;
this.y = (this.y - camera.y) * scale;
this.xscale = this.yscale = 100 * scale;
```

It's important to understand that by putting the `camera.z` in the numerator of your scale equation you've actually changed the position of your singularity. This can produce some weird effects since at the singularity your object will blow up in size and then flip.

You could continue to extend this train of thought adding many other features such as rotation and scaling, as was done with Papervision3D. But there is a better way!

In CS4 the perspective scaling and rotation are automatically built into Flash. It's still important to know the concepts above when working with Flash 10 (that's why they were included), but it's just not necessary to do all this work. To illustrate the point, rebuild your spiraling 19 line animation engine in CS4.

Using CS4 (3D Flash 10 Engine in 13 Lines of Code)

Using CS4 you only need 13 lines of code to create the same animation that took 18 lines above. Saving 5 lines may not seem like much, but it enables you to drop your perspective scaling and let the Flash 10 engine do all the work.

```
import flash.display.Sprite;//imports sprite class
var myAngle:Number =0;//Angle of ball
var ball:Sprite = new Sprite();//instantiates ball sprite
ball.graphics.beginFill(0xFF0000);//Assign a ball color
ball.graphics.drawCircle(0, 0, 40);//draws your ball at 0,0
ball.graphics.endFill();//ends the fill
addChild(ball);//adds the ball to the stage
addEventListener(Event.ENTER_FRAME, myonEnterFrame);//loops equations
function myonEnterFrame(event:Event):void{
myAngle=myAngle+.1;//iterates angle
ball.x = 300*Math.sin(myAngle); //ball orbit x
ball.y = 300*Math.cos(myAngle); //ball orbit y
ball.z = 2000*Math.sin(myAngle/10);} //increments z and changes sign
```

Allowing Flash 10 to do the perspective scaling gives you much leverage in building 3D applications easily. Essentially, it's what Papervision3D does, but now that power is built directly into the Flash 10 player with no complicated set up or massive imports. Not that you're going to abandon Papervision3D . . . that's what this book is about. But whenever a better way to do it exists in CS4, we'll cover it as much as space allows.

Vanishing Point

After running the program above, you'll notice that the vanishing point moved. That's because Flash 10 automatically set the vanishing point. It's set upon initiation of the swf, based on the size set in the document properties panel. But there's a glitch in Flex. Flex Builder's default swf has a size of 500×375 pixels, and you'll probably never create an swf that size. As a result, all your vanishing points will be off. The solution is to use the PerspectiveProjection class to set your vanishing point. To set your display object to center stage, for example, you would use the following command:

```
 myDisplayObject.transform.perspectiveProjections.projectionCenter =
new Point(stage.stageWidth/2, stage.stageHeight/2);
```

Not only does it set the vanishing point for your display object (`myDisplayObject` in this case), but also its children. And that's the power of CS4's new display object. When applying transforms to a CS4 display object, its children are transformed as well. Papervision3D does the same thing with its DisplayObject3D, but now it's inherent in the Flash10 player. The `flash.display.DisplayObject` class contains the z property and new rotation and scaling properties for manipulating display objects in 3D space.

Using the Big 3

Fundamental to 3D graphics and physics are three fundamental motions: translation, rotation and scaling. All motions (at least for Newtonian motion) can be broken down into combinations of these three motions.

You can apply these 3D transformations all at once using a Matrix3D object. You can rotate, scale, and then move an object by applying three separate transformations or more efficiently by using one Matrix3D transformation. It's important to remember that these matrix operations aren't generally commutative: which means that applying a rotation and then translation won't necessarily give the same results as applying the reverse order (a translation then rotation). The following code snippet shows how to cascade a series of transformations: rotation, scale, translation, and then rotation again.

```
var matrix:Matrix3D = myDisplayObject.transform.matrix3D;
matrix.appendRotation(45, Vector3D.Y_AXIS);
matrix.appendScale(2, 1, 3);
matrix.appendTranslation(10, 150, -300);
matrix.appendRotation(10, Vector3D.X_AXIS);
myDisplayObject.transform.matrix3D = matrix;
```

Performing difficult matrix maths is unnecessary . . . that's great news! Adobe has made it pretty easy to do this just by applying the simple transformations shown above. And in many cases it's done automatically without the user even knowing it's being performed. Consider translation, for example, when you explicitly set the z property of a display object to a numeric value, the object automatically creates a 3D transformation matrix. It all happens behind the scenes giving you more time to concentrate on building stellar 3D experiences.

Translation

Adding a native z coordinate, in Flash 10, enables you to treat z just as you've treated x and y in the past. But using translation doesn't just mean you're traveling in a straight line. You can use z to constrain an element to a 3D trajectory path. As an example, consider a parametric path on a 3D surface such as a torus.

The parametric curves for a torus (and any other 3D surface) can be found at WolframMathWorld (www.mathworld.wolfram.com).

So what's a parametric equation?

Parametric equations are a set of equations that define the coordinates of the dependent variables (x, y and z) of a curve or surface in terms of one or more independent variables or parameters. That's a mouthful, but basically if you iterate over the range of your parameters (of the parametric equation) your torus will be plotted in 3D space. This is a very useful device as it gives you the vertices of your torus, and in Appendix A the parametric equation for a number of 3D objects are given.

The parametric equations for a torus are:

```
x=(c+a*cos(v))cos(u)
y=(c+a*cos(v))sin(u)
z=a*sin(v)
```

where c is the donut radius, and a is the tube radius. And u is the parameter that takes you around the larger radius (the donut) and v around the smaller tube radius as shown in Figure 1.7.

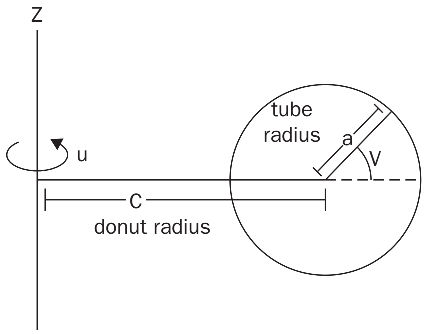

Figure 1-7

You now use these equations to create a parametric path on your torus. What you want to do is have your graphical element spiral around the torus. You can accomplish this by iterating the parameters u and v simultaneously adjusting v to get to the desired spiral velocity. The parametric path is extremely easy to execute. Just take the code from the 3D Flash 10 Engine in 13 lines and substitute your parametric equations for ball.x, ball.y, and ball.z as shown below:

```
import flash.display.Sprite;//imports sprite class
var myAngle:Number =0;//Angle of ball
var ball:Sprite = new Sprite();//instantiates ball sprite
ball.graphics.beginFill(0xFF0000);//Assign a ball color
ball.graphics.drawCircle(0, 0, 10);//draws your ball at 0,0
ball.graphics.endFill();//ends the fill
addChild(ball);//adds the ball to the stage
addEventListener(Event.ENTER_FRAME, myonEnterFrame);//loops equations
function myonEnterFrame(event:Event):void{
myAngle=myAngle+.1;//iterates angle
//ball parametric orbit x
ball.x = (100+50*Math.cos(2*myAngle))*Math.cos(myAngle/4)+200;
//ball parametric orbit y
ball.y = (100+50*Math.cos(2*myAngle))*Math.sin(myAngle/4)+200;
//ball parametric orbit z
ball.z = 50*Math.sin(2*myAngle);}
```

It works flawlessly, but a single ball isn't very exciting. Now add a few more balls to your parametric path.

Creating a Parametric Particle System (Torus Worm)

In the previous section, you learned how to get a single element orbiting a torus using the parametric equations of a torus. Here, you multiply that single element by 100 and create a parametric particle system, which orbits (or worms around) your torus.

You get a more formal treatment of particles in the "Gaming" section of this book. But for the most part, anything with more than one element can be treated as a particle system. Treating elements as particles has a number of advantages. Primarily, particles are easily handled mathematically.
You'll use particle systems extensively throughout this book.

Building a particle system in Flash is relatively easy, and in this section, you cover the basics. To build the orbiting worm around a torus (parametric particle system), follow the steps below:

1. Declare the number of particles to be created and create an array to place your particles in. This is the key to working with particles. Using the array "particles_ary" you're able to control position, color, and alpha of each particle.

```
var numOfParticles:uint = 100;
var particles_ary:Array = [];
```

2. Use a "for" loop to run the updateStage() function, which creates your particles. Once the particles are created run the addEventListener method, which starts updating the loop for each particle on every frame event.

```
 for(var i:uint = 0; i < numOfParticles; i++)
{
        updateStage();
        numVar++;
   //Start Looping
   if(numVar==numOfParticles){
   addEventListener(Event.ENTER_FRAME, myonEnterFrame);}

   }
```

3. Draw your balls (particles) to the stage and place them in a particle array. A random color is assigned to each ball using the Math.random()*0xffffff method.

```
//Draw a Ball
var ball:Sprite = new Sprite
//Assign a random ball color
ball.graphics.beginFill(Math.random()*0xffffff);
//draws your ball at 0,0
ball.graphics.drawCircle(0, 0, ballRadius);
ball.graphics.endFill();//ends the fill
//Add ball to the stage
addChild(ball);
//Push the ball into a particle array
particles_ary.push(ball);
```

4. With each onEnterFrame loop you update all your particle positions using myAngle+i/20 that separates the different particles in angle by the ratio of i/20. This causes your particles to line up on the torus and move across its surface as the myAngle variable is iterated. Use the

particles_ary[i].alpha method to change the alpha of each particle based on their ith position. Center the particle system on the stage by adding CenterX, and CenterY to the particle x, y positions.

```
      for(var i:uint = 0; i < particles_ary.length; i++)
      {
      //ball parametric orbit x
      var newAngle:Number=myAngle+i/20;
      particles_ary[i].alpha=1/(i+1)+.2;
      particles_ary[i].x = (100+60*Math.cos(2*newAngle))*Math.cos(newAngle/
4)+CenterX;
      //ball parametric orbit y
      particles_ary[i].y = (100+60*Math.cos(2*newAngle))*Math.sin(newAngle/
4)+CenterY;
      //ball parametric orbit z
      particles_ary[i].z = 60*Math.sin(2*newAngle);
      }
```

The results are shown in Figure 1.8 and result in a wormlike entity orbiting a torus. The key to creating particle systems and controlling them is to stuff the individual particles into an array upon creation. And then iterate over each particle during the frame loop of the animation sequence.

Figure 1-8

Putting it all together, the entire code is show below:

```
//imports sprite and stage class
import flash.display.Sprite;
import flash.display.Stage;
//Add Particle Array
var numOfParticles:uint = 100;
var particles_ary:Array = [];
var numVar:Number=0;

var ballRadius:Number=10;
//Stage Center
var CenterX:Number = stage.stageWidth/2;
var CenterY:Number = stage.stageHeight/2;
```

```
var myAngle:Number =0;//Angle of ball

   //Add Multiple Particles
   function updateStage():void
   {
      //Draw a Ball
      var ball:Sprite = new Sprite
      //Assign a random ball color
      ball.graphics.beginFill(Math.random()*0xffffff);
      //draws your ball at 0,0
      ball.graphics.drawCircle(0, 0, ballRadius);
      ball.graphics.endFill();//ends the fill
      //Add ball to the stage
      addChild(ball);
      //Push the ball into a particle array
      particles_ary.push(ball);
   }

   //Place Particles on the Stage
   for(var i:uint = 0; i < numOfParticles; i++)
   {
         updateStage();
         numVar++;
      //Start Looping
      if(numVar==numOfParticles){
            addEventListener(Event.ENTER_FRAME, myonEnterFrame);}

   }

   //Looping function
   function myonEnterFrame(event:Event):void{
   myAngle=myAngle-.1;//iterates angle

      for(var i:uint = 0; i < particles_ary.length; i++)
      {

            //ball parametric orbit x
            var newAngle:Number=myAngle+i/20;
            particles_ary[i].alpha=1/(i+1)+.2;
            particles_ary[i].x =
(100+60*Math.cos(2*newAngle))*Math.cos(newAngle/4)+CenterX;
            //ball parametric orbit y
            particles_ary[i].y =
(100+60*Math.cos(2*newAngle))*Math.sin(newAngle/4)+CenterY;
            //ball parametric orbit z
            particles_ary[i].z = 60*Math.sin(2*newAngle);


      }}
```

No discussion about particles would be complete without mention of Seb Lee Delisle, an expert in particle systems and one of the Papervision3D team members. Seb has a great site that covers particle creation in Papervision3D at http://www.sebleedelisle.com/. Next on the list of particle super stars is the

Flint-Particle-System (at http://flintparticles.org/), a robust open source project found on Google Code, which can be used to create impressive particle systems for Papervision3D.

However, the particle landscape is changing. Having a native 3D and pixel bender with the release of CS4 greatly simplifies the process of building particle systems in 3D (as shown in the code above).

Depth Sorting

Shuffling Objects so that the ones closer to your viewpoint appear on top of objects farther away is called depth sorting (or z-sorting). One big advantage that Papervision3D has over CS4 is automatic depth sorting. In CS4, sorting algorithms must be written and unfortunately they're dependent on object nesting.

When creating a 3D carousel for example you need to z-sort the objects as they spin to put the closest one on top of the stack. The technical name is transposition and in AS2 it was easily accomplished using the `swapDepths` method. But in AS3 it's a little more complicated.

Objects on the Stage

In AS3, the display list functions as an array and each display object has an index. The index starts at zero and goes up to the number of objects on your stage where index zero is the bottom object. So since the display object is a child you can change its position using the `setChildIndex` method.

As all your objects are in an array, you can sort that array by z and then set the indices of your array objects based on z. And that's how it's presently done! It uses the same code structure variables as the torus worm. Here's a sort code snippet that illustrates the concept of sorting:

```
//Sorting
function sortParticles():void
{
      particles_ary.sortOn("z", Array.DESCENDING|Array.NUMERIC);
      for(var i:int = 0; i < particles_ary.length; i++)
      {
      addChildAt(particles_ary[i] as Sprite, i);
      }
}
```

Using the pipe `Array.DESCENDING|Array.NUMERIC` forces the z value to be sorted in reverse numeric order. Particles are sorted from high to low based on their z value. Thus, the objects further away from your projection plane get a lower index value, placing them on the bottom of the stack.

This method needs to be called each time your 3D engine iterates the position of your objects. Typically this occurs on an `onEnterFrame` event. Unfortunately, you're stuck with manual sorting. But expect sorting to be native to the next version of the Flash Player.

Objects in a Display Container

In many examples in this book, you won't render multiple objects directly to the stage, but inside a display container. This approach works well when you rotate multiple objects. Instead of creating a complex trig algorithm to rotate each individual object you can just rotate the display container.

But in CS4 there's a problem with this. When using the sorting algorithm (above) you sort the elements on the z-axis internal to the container. But when the container is rotated the objects rotate with it and are no longer sorted correctly. Essentially, there are two z values: z inside the container and z outside the container. To sort these correctly, you must sort with respect to the z axis outside your container. You must transform the local coordinates inside your container to the stage coordinates (or z outside your container).

Keith Peters in his advanced book on AS3 Animation proposed the following solution for the container-sorting problem.

```
//Sorting
function sortParticles():void
{
      particles_ary.sort(particleZSort);
      for(var i:int = 0; i < particles_ary.length; i++)
      {
      myDisplayObject.addChildAt(particles_ary[i] as Sprite, i);
      }
}

//
function particleZSort(particle1:DisplayObject,particle2:DisplayObject):int
{
var zpos1:Vector3D = particle1.transform.matrix3D.position;
zpos1 = myDisplayObject.transform.matrix3D.deltaTransformVector(zpos1);
var zpos2:Vector3D = particle2.transform.matrix3D.position;
zpos2 = myDisplayObject.transform.matrix3D.deltaTransformVector(zpos2);
return zpos2.z - zpos1.z;
}
```

The `particles_ary.sort` method passes in the `particleZSort` function as a parameter. The `particleZSort` function is called multiple times with pairs of particles during a sort, returning a negative number if the first particle should be placed in front of the second. The `Matrix3D` and `Vector3D` methods are used to keep track of your objects' rotated positions, which we discuss in greater detail later in Chapter 3.

This method is computationally intensive, but works surprisingly well (as you can see from the Carousel and Image Ball examples next).

Rotation

You've already met basic sinusoidal functions, such as sine and cosine, in this chapter: it's these functions that give you the ability to rotate elements in space. Even though rotation is inherent in Papervision3D (and Flash 10), you still use these functions occasionally when manipulating elements in 3D space. For example, in the gaming section of this book, you use these equations to rotate a Rubik's cube.

3D Rotation

To rotate x, y, z points in 3D space you use the following equations:

xnew = xold*cos(angleZ) - yold*sin(angleZ)

ynew = xold*sin(angleZ) + yold*cos(angleZ)

xnew = xold*cos(angleY) - zold*sin(angleY)

znew = xold*sin(angleY) + zold*cos(angleY)

ynew = yold*cos(angleX) - zold*sin(angleX)

znew = yold*sin(angleX) + zold*cos(angleX)

There are actually two ways to rotate elements in Papervision3D; using the preceding matrix equations or using quaternions. Most 3D engines start with the preceding matrix equations.

Flash 10 Rotation

In Flash 10, rotation around the x, y, and z-axis is clockwise as angle increases. Rotation around the x, y, and z-axis are accomplished using the `rotationX`, `rotationY`, and `rotationZ` commands respectively. Just as in Papervision3D, the rotational values are given in degrees as opposed to radians, which had been the practice in Flash previously. But when using sine and cosine functions you still use radians (see Figure 1.9).

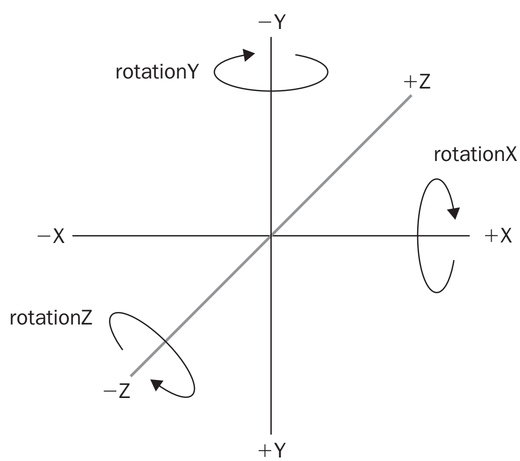

Figure 1-9

The conversion between radians and degrees and vice-versa is given below:

```
Radians = Degrees*PI/180
Degrees = Radians*180/PI
```

Although these conversions are very simple, you use them all the time.

Creating a Carousel and Image Ball

Two of the most popular navigation devices created using Papervision3D are the carousel and image ball. As done in the torus worm example, you'll treat the planes in the carousel and image ball as particles. Place these particles into a display object so you can rotate them with a single rotation command. But in this case (as opposed to the torus worm example), you'll need to sort your planes according to their z-positions. Otherwise, as you rotate your display container, your planes will not overlap correctly.

Carousel

Creating a carousel is very similar to creating the torus worm demonstrated earlier. As in the torus worm case, the planes of the carousel are treated as particles. And the planes are placed in Sprite display objects with the variable name `myDisplayObject`. Placing the planes in a display object gives you the ability to rotate all the planes at once using the `rotationY` method.

The individual planes are created using the `drawRect` method.

```
var plane:Sprite = new Sprite();//instantiates plane sprite
plane.graphics.beginFill(Math.random()*0xffffff);//Assign a plane color
plane.graphics.drawRect(-60, -60, 80, 80);//draws your plane at 0,0
plane.graphics.endFill();//ends the fill
```

Using a little trig to place your planes is at the heart of the code. The planes are set around a circle of radius 200 pixels using sine and cosine functions.

```
plane.x = Math.cos(angle) * 200;
plane.z = Math.sin(angle) * 200;
```

But the orientation of each plane is not correct. Each plane is facing the same direction. To align the planes around the carousel correctly you use the `rotationY` method. Dividing -360 by the number of particles, multiplying by the iteration variable i, and adding 90 degrees sets the planes to the right orientation.

```
plane.rotationY = -360 / numOfParticles * i + 90;
```

Once the planes are placed correctly in the carousel they're sorted using the container-sorting algorithm discussed in the previous section (see Figure 1.10).


Figure 1-10

Finally the display container is rotated based on mouse movement using a simple `mouseX` displacement algorithm.

```
function myonEnterFrame(event:Event):void
{
myDisplayObject.rotationY += (CenterX - mouseX) * .01;
sortParticles();

}
```

The complete carousel code is listed here:

```
//imports classes
import flash.display.DisplayObject;
import flash.display.Sprite;
import flash.display.Stage;
import flash.geom.Vector3D;
//Add Particle Array
var numOfParticles:uint = 12;
var particles_ary:Array = [];
var numVar:Number=0;
var myDisplayObject:Sprite= new Sprite();

var planeRadius:Number=20;
//Stage Center
var CenterX:Number = stage.stageWidth/2;
var CenterY:Number = stage.stageHeight/2;


//Place Planes on the Stage
for(var i:uint = 0; i < numOfParticles; i++)
{
//Draw a plane
var plane:Sprite = new Sprite();//instantiates plane sprite
plane.graphics.beginFill(Math.random()*0xffffff);//Assign a plane color
plane.graphics.drawRect(-60, -60, 80, 80);//draws your plane at 0,0
plane.graphics.endFill();//ends the fill

//Add plane to the stage
myDisplayObject.addChild(plane);
var angle:Number = Math.PI * 2 / numOfParticles * i;

plane.x = Math.cos(angle) * 200;
plane.z = Math.sin(angle) * 200;
plane.y =0;
```

```
plane.rotationY = -360 / numOfParticles * i + 90;
plane.alpha=1;
particles_ary.push(plane);

numVar++;
//Start Looping
if(numVar==numOfParticles)
      {
      addEventListener(Event.ENTER_FRAME, myonEnterFrame);
      addChild(myDisplayObject);
      myDisplayObject.rotationX=0;
      myDisplayObject.x=CenterX;
      myDisplayObject.y=CenterY;

      }
}

//Sorting
function sortParticles():void
{
particles_ary.sort(particleZSort);
      for(var i:int = 0; i < particles_ary.length; i++)
      {
      myDisplayObject.addChildAt(particles_ary[i] as Sprite, i);
      }
}

function particleZSort(particle1:DisplayObject, particle2:DisplayObject):int
{
var zpos1:Vector3D = particle1.transform.matrix3D.position;
zpos1 = myDisplayObject.transform.matrix3D.deltaTransformVector(zpos1);
var zpos2:Vector3D = particle2.transform.matrix3D.position;
zpos2 = myDisplayObject.transform.matrix3D.deltaTransformVector(zpos2);
return zpos2.z - zpos1.z;
}
//Looping function
function myonEnterFrame(event:Event):void
{
myDisplayObject.rotationY += (CenterX - mouseX) * .01;
sortParticles();

}
```

Image Ball

An image ball is just a bunch of images equally distributed around a sphere. The trick to distributing the images correctly is to use the parametric equation for a sphere, which can be found from WolframMathWorld. Iterating through the different rows and columns of the parametric equations distributes the planes to their correct positions. But they won't be oriented correctly.

```
plane.x = radius*Math.sin(i*pStep)*Math.sin(j*tStep);
plane.z = -radius*Math.sin(i*pStep)*Math.cos(j*tStep);
plane.y =-radius*Math.cos(i*pStep);
```

You must orient the planes to the correct angles so that they uniformly hug the sphere as shown in Figure 1.11.

```
plane.rotationX=phiTilt[i];
plane.rotationY=-j*thetaStep[i];
```

For this particular example the vectors for the plane positions and angles were obtained from Flash & Math (http://www.flashandmath.com/).

Figure 1-11

As in the previous example, you use the z-sorter and `mouseX` displacement algorithm code to sort the rectangles correctly in depth and move the entire container using the `rotationY` method.

The complete image ball code is listed below:

```
//imports sprite and stage class
import flash.display.DisplayObject;
import flash.display.Sprite;
import flash.display.Stage;
import flash.geom.Vector3D;
//Add Particle Array
var particles_ary:Array = [];
var myDisplayObject:Sprite= new Sprite();

//Image Ball Parameters
//Parameters adapted from Flash & Math
// http://www.flashandmath.com/
var jLen:Vector.<Number>=new Vector.<Number>();
jLen=Vector.<Number>([1,6,10,12,10,6,1]);
var thetaStep:Vector.<Number>=new Vector.<Number>();
thetaStep=Vector.<Number>([0,60,36,30,36,60,0]);
```

```
//The vertical angle between circles.
var phiStep:Number=30;
var phiTilt:Vector.<Number>=new Vector.<Number>();
phiTilt=Vector.<Number>([-90,-60,-30,0,30,60,90]);
//radius of the Sphere
var radius:Number=180;
//Stage Center
var CenterX:Number = stage.stageWidth/2;
var CenterY:Number = stage.stageHeight/2;

//Place Particles on the Stage
var i:int;
var j:int;
var tStep:Number;
var pStep:Number=phiStep*Math.PI/180;
for(i=0;i<7;i++){
tStep=thetaStep[i]*Math.PI/180;
for(j=0;j<jLen[i];j++)
  {

//Draw a plane
var plane:Sprite = new Sprite();//instantiates plane sprite
plane.graphics.beginFill(Math.random()*0xffffff);//Assign a plane color
plane.graphics.drawRect(-30, -30, 30, 30);//draws your plane at 0,0
plane.graphics.endFill();//ends the fill

//Add plane to the stage
      myDisplayObject.addChild(plane);
      //Use parametric equations of a sphere
      plane.x = radius*Math.sin(i*pStep)*Math.sin(j*tStep);
      plane.z = -radius*Math.sin(i*pStep)*Math.cos(j*tStep);
      plane.y =-radius*Math.cos(i*pStep);
      //Set rotations
      plane.rotationX=phiTilt[i];
      plane.rotationY=-j*thetaStep[i];
      particles_ary.push(plane);
   }
}

//Start Looping and Set Display Object
addEventListener(Event.ENTER_FRAME, myonEnterFrame);
addChild(myDisplayObject);
myDisplayObject.rotationX=0;
myDisplayObject.x=CenterX;
myDisplayObject.y=CenterY;

//Sorting
function sortParticles():void
{
      particles_ary.sort(particleZSort);
      for(var k:int = 0; k < particles_ary.length; k++)
      {
```

(continued)

(continued)

```
      myDisplayObject.addChildAt(particles_ary[k] as Sprite, k);
      }
}

function particleZSort(particle1:DisplayObject, particle2:DisplayObject):int
{
var zpos1:Vector3D = particle1.transform.matrix3D.position;
zpos1 = myDisplayObject.transform.matrix3D.deltaTransformVector(zpos1);
var zpos2:Vector3D = particle2.transform.matrix3D.position;
zpos2 = myDisplayObject.transform.matrix3D.deltaTransformVector(zpos2);
return zpos2.z - zpos1.z;
}

//Looping function
function myonEnterFrame(event:Event):void
{
    myDisplayObject.rotationY += (CenterX - mouseX) * .01;
    sortParticles();

}
```

Scaling (Basic Principles of Animation)

Many of the basic principles of animation, developed by Walt Disney, have their roots in scaling.

Scaling is the factor that adds realism to your 3D animations. For example, as a ball bounces against a hard surface it should exhibit a squash and stretch effect (accomplished by scaling). Walt Disney and his team of animators quantified the rules of cartoon animation. And no matter how great your images and 3D scenes are, if you don't apply these principles, your 3D animations will appear lifeless.

Here's a summary of Walt's Rules of animation. Applying them will bring your 3D animations to life.

- **Squash and Stretch** occurs when a soft object comes into contact with a hard surface. It deforms proportionally to its velocity. The deformation exhibits both a squash and stretch, but its volume is conserved. In 3D, x and z are the stretch axis and y the squash axis.
- **Exaggeration** brings attention to an activity that's being performed. Its motion is a little more flamboyant than in real life, but it draws the viewers in and adds character to your animation.
- **Anticipation** is the prep before the action. Prep-action before the action builds the excitement for what's about to happen. It's one of the great secrets of animation. Every animation should have anticipation injected in it – you should look for every opportunity to do so.
- **Follow-Through** is the action after the action. Once the action occurs, such as throwing a ball, or closing a car door, there's follow-through. It establishes the personality of your character. It's the aftermath of an action that really sells it.

- **Weight** is the concept most forgotten in animation. Animation software doesn't know how much something weighs. Communicating weight to your audience requires that your animated character shows strain when it picks up something heavy.

An execution of squash and stretch principle is illustrated in Figure 1.12. It shows a bouncing ball squash and stretch as it hits the floor. As the ball approaches the floor, both shadow and ball stretch.

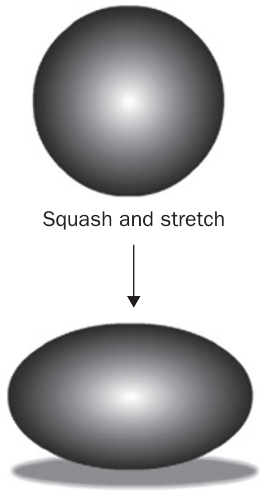

Figure 1-12

The bouncing ball above was created completely in the Flash CS4 motion editor and involves several motion edits. Here's a really cool trick that enables you to apply this animation sequence to any object. You now turn this timeline-based animation completely into ActionScript.

Turning Timeline Animation into ActionScript

One of the amazing additions to Flash 10 is the motion editor, which gives you powerful control over the individual properties of your 3D animations. All animation can be done completely from the motion editor and motions created can be saved as custom presets and added to any object. In addition, there are tons of presets to choose from.

A full treatment of the motion editor is beyond the scope of this book. A good place to learn about the motion editor is to view the CS4 video tutorial series by Todd Perkins on http://www.Lynda.com. Lynda.com is a video tutorial service that specializes in Multimedia and Adobe products. They're relatively inexpensive and fairly up to date.

Converting Your Animation

You can convert 3D animation, created on the timeline or in the motion editor, into ActionScript. It's pretty simple. Just right click on the timeline animation tween and choose Copy Motion As ActionScipt

3.0. The ActionScript is then copied to your clipboard and you can paste this code wherever you want. If you try this with the bouncing ball example (shown above) you get the following code:

```
import fl.motion.AnimatorFactory;
import fl.motion.MotionBase;
import flash.filters.*;
import flash.geom.Point;
var __motion_Ball_16:MotionBase;
//Is there animation running, if not start it
if(__motion_Ball_16 == null) {
import fl.motion.Motion;
__motion_Ball_16 = new Motion();
__motion_Ball_16.duration = 10;

// Call overrideTargetTransform to prevent the scale, skew,
// or rotation values from being made relative to the target
// object's original transform.
// __motion_Ball_16.overrideTargetTransform();
// The following calls to addPropertyArray assign data values
// for each tweened property. There is one value in the Array
// for every frame in the tween, or fewer if the last value
// remains the same for the rest of the frames.

__motion_Ball_16.addPropertyArray("x", [0,0,0,0,0,0,0,0,0,0]);
__motion_Ball_16.addPropertyArray("y", [0,7.38399,37.4766,112.504,269.275,333.95,22
1.075,47.6064,8.68268,0]);
__motion_Ball_16.addPropertyArray("scaleX", [1.000000,1.000000,1.000000,1.000000,1.
000000,1.250000,1.187556,1.092333,1.023972,1.000000]);
__motion_Ball_16.addPropertyArray("scaleY", [1.000000,1.000000,1.000000,1.000000,1.
000000,0.750000,0.812444,0.907667,0.976028,1.000000]);
__motion_Ball_16.addPropertyArray("skewX", [0,0,0,0,0,0,0,0,0,0]);
__motion_Ball_16.addPropertyArray("skewY", [0,0,0,0,0,0,0,0,0,0]);
__motion_Ball_16.addPropertyArray("rotationConcat", [-0.251785,-0.251785,-
0.251785,-0.251785,-0.251785,-0.251785,-0.251785,-0.251785,-0.251785,-0.251785]);
__motion_Ball_16.addPropertyArray("blendMode", ["normal"]);

// Create an AnimatorFactory instance, which will manage
// targets for its corresponding Motion.
var __animFactory_Ball_16:AnimatorFactory = new AnimatorFactory(__motion_Ball_16);
__animFactory_Ball_16.transformationPoint = new Point(0.746748, 0.778455);

// Call the addTarget function on the AnimatorFactory
// instance to target a DisplayObject with this Motion.
// The second parameter is the number of times the animation
// will play - the default value of 0 means it will loop.
// __animFactory_Ball_16.addTarget(<instance name goes here>, 0);}
```

The code above is relatively simple. If you've done this before in Flash 9, you'll immediately notice that it's not XML. The above code is built completely in ActionScript using the `addPropertyArray` method. This new approach is designed to work with the motion editor and uses property arrays of frame-by-frame motion tween data.

Here's how the code is put together:

- The code starts by bringing in a number of imports that set up the animation framework.
- The code uses an 'if' statement `if(__motion_Ball_16 == null)` to determine if animation is running or not. If it isn't, it imports all the necessary parameters needed to get the animation going. The name Ball comes from the name of the movie clip in your Flash Library.
- The `addPropertyArray` method is then executed as many times as needed to bring all the data created in the motion editor.

For example, in the bouncing ball example there were ten frames of animation, and comparatively, in each `addPropertyArray` there are ten values. As shown below for the y argument there are ten values corresponding to the position of your ball in the animation:

```
__motion_Ball_16.addPropertyArray("y", [0,7.38399,37.4766,112.504,269.275,333.95,22
1.075,47.6064,8.68268,0]);
```

- Finally, to apply this animation to another object you need to uncomment the final line of code:

```
// __animFactory_Ball_16.addTarget(<instance name goes here>, 0);
```

- Create a movie clip, put it on your Flash stage, and give it an instance name.
- Then place that instance name in the <instance name goes here> spot in the code above.

You can now apply this custom animation to as many objects as you want using only ActionScripting. The motion editor allows you to work with a large number of properties.

The following properties can be included in your animation using the motion editor:

- Basic Motion (X, Y, Z, Rotation X, Rotation Y, Rotation Z)
- Transformation (Skew X, Skew Y, Scale X, Scale Y)
- Color Effect (Alpha, Brightness, Tint, Advanced)
- Filters (Drop Shadow, Blur, Glow, Bevel, Gradient Glow, Gradient Bevel, Adjust Color)
- Eases (Simple, Start and Stop, Bounce, Spring, Sine, Sawtooth, Square, Random, Damped, and Custom)

Adding these properties is easy in the Flash 10 motion editor, very much like using curve editors found in sound and graphic programs.

Collada Files

Handling animation and bringing 3D models into Papervision3D is an important topic. It's also one of great frustrations to the Papervision3D user community. There's nothing worse than spending a month creating an incredible 3D model and not being able to get it to run in Papervision3D. Presently, Papervision3D brings 3D objects into its program using primitives or Collada files.

So what's a Collada File?

Essentially, it's an XML document that holds the parameters of your 3D model. It's designed to be both human readable and platform independent. Its incorporation into Papervision3D as the primary way to bring in 3D models may have been premature. Not because there's anything wrong with the Collada platform, but because most 3D modeling programs don't export a version of Collada that works with Papervision3D.

At the time of writing of this book, there wasn't a Blender Collada exporter for Papervision3D, and the 3DSMax exporters for Papervision3D only worked with certain versions of 3DSMax. But there are other options. In this book, you find out how to create XML exporters (essentially pared down Collada files) that work for both Papervision3D and Flash 10. Combining an XML exporter with the motion editor code presented earlier enables you to create a dynamic 3D animation engine.

Another way of dealing with 3D modeling is to use Papervision3D or Flash10 in which to create your 3D models. This requires you to create a modeling application in Papervision3D, but doing so allows you the freedom of working with your models natively. You can find out more about Collada and XML exporters in Chapter 5. But keep in mind that Flash offers great flexibility. If something isn't working for you just try another way.

Summary

In terms of innovation and "coolness factor", Flash 3D is one of the fastest-moving areas in web technology. In this chapter, you discovered how to build a simple 3D engine in both CS3 and CS4. Using what you learned about 3D engines you created a torus worm, carousel, and image ball. Finally, you examined Disney's rules for creating realistic animation, and converted a timeline animation into ActionScript.

You've begun the most important step in learning Papervision3D – understanding how 3D is made!