CHAPTER 17

Vector Shape and Flash Lingo

In This Chapter

Creating vector shapes with Lingo

Modifying vector shapes with Lingo

Using Flash movies in Director

Director can now take advantage of vector-based art. Bitmaps are described by pixels, but vectors are described mathematically. This enables vectors to scale up without becoming pixelated. In other words, vector-based art shows more detail as it gets larger, whereas pixels have a finite resolution. Programs such as Photoshop and Painter work with pixels; programs such as Freehand, Illustrator, Flash, and In-Motion mostly use vectors.

Creating a Vector Shape with Lingo

Creating a new vector shape is easy. You do it the same way you create a new member of other types, as follows:

```
myVector = new(#vectorShape)
```

This creates a new member of the type `#vectorShape`, but it is empty. It is similar to creating a new text member on the fly — there is no text until you put it there. With a vector shape, you need to put in the vertices and handles. We'll show you how to use Lingo to add vertices later on in the chapter. First, we'll examine the `vertexList` of a shape created through the user interface.

The `vertexList` is a list of lists, defining the points and handles of a vector shape member. A good way to understand the `vertexList` is to create a couple of simple vector shapes and to examine their `vertexList` properties.

Examining the vertexList

1. Choose Insert ➪ Media Element ➪ Vector Shape. A blank vector shape window opens.
2. Select the Rectangle tool and draw a square by holding down the Shift key as you draw. Draw the square from the upper-left down to the lower right.
3. Open the Message window by choosing Window ➪ Message.
4. Type the following in the Message window:

```
put the vertexList of member 1
```

Figure 17-1 shows the result. Your `vertexList` might have different values, depending on the size of the square.

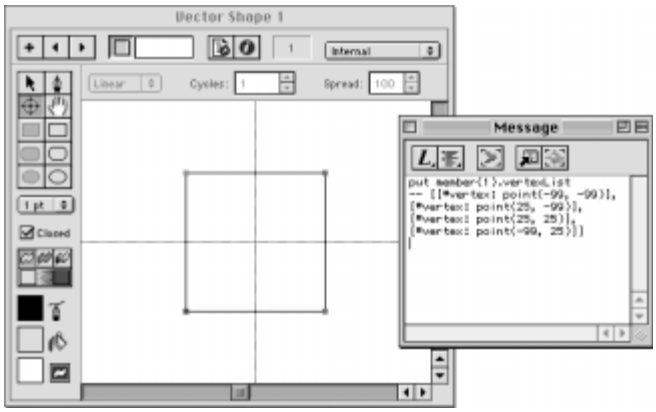

Figure 17-1: Examining the vertexList of a square vector shape

The `vertexList` is a linear list that holds a property list for each vertex. The first element in the list is the vertex in the upper-left corner. The next item in the list is the vertex at the upper right. The vertex at the lower right is the third item in the list, and the last item in the list is the vertex at the lower left. Pretty straightforward stuff. Now insert another vector shape, but this time draw a circle. In the Message window, put the `vertexList` for this member, and you should see something like Figure 17-2.

Tip To make all the handles visible at once in the Vector Shape Window, choose Select All from the Edit menu.

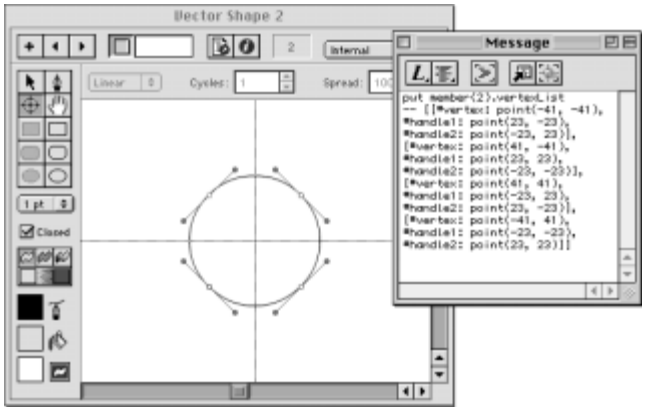

Figure 17-2: There's more to a vertexList than just vertices—curves have handles.

At first glance, this list might look a little intimidating. Upon closer examination, you see it is very similar to the `vertexList` for the square. It has four vertices, each one in its own property list within the linear `vertexList`. The big difference is now there are two additional elements in each of those lists: `#handle1` and `#handle2`. If you've worked with drawing programs, you've seen control handles on curves. `#handle1` and `#handle2` hold the location of the handle's control point relative to the vertex, but they are not absolute coordinates, as the vertices are. As you examine the `vertexList` from this member, note that all the handles are similar coordinates. This is because they are all similarly offset from their respective vertices. This would be much less obvious on a more arbitrary shape, as in Figure 17-3.

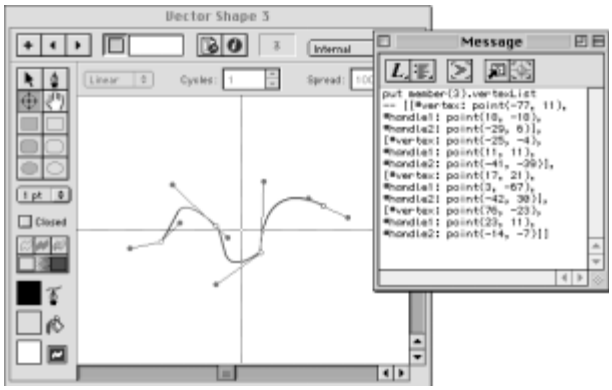

Figure 17-3: Vector shapes do not have to be closed shapes, nor do their handles need to be continuous.

The shape in Figure 17-3 illustrates two more aspects of vector shapes — they do not need to be closed shapes, and handles do not need to be continuous (meaning it does not have to be a straight line from one handle to another). Save this movie as **vector shapes.dir**.

The important thing to remember is that each property list in the `vertexList` has, at the most, three elements: `#vertex`, `#handle1`, and `#handle2`. A `vertexList` can contain any number of vertices. Each vertex can have 0, 1, or 2 handles.

Setting the vertexList

A powerful feature of Director 8 is the capability to set the `vertexList` of any vector shape. In the Message window, type the following:

```
myVector = new(#vectorShape)
myVector.vertexList = member(2).vertexList
```

`myVector` holds a reference to the member created with the new function. Because it was just created, the `vectorList` is empty. Next, the `vectorList` of member 2 (the circle) is assigned to it. If you've been following along, this new member should be in position 4 in the internal cast. It should look something like Figure 17-4.

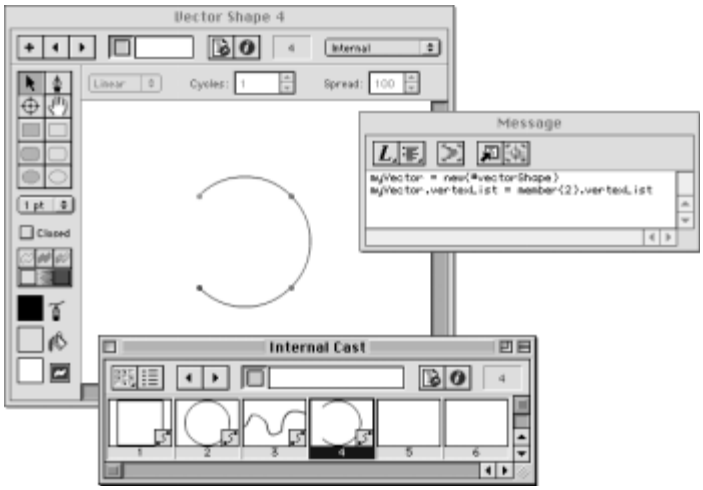

Figure 17-4: A new member is created and then the vertexList of one member is assigned to the other (what's wrong with this picture?).

The new circle is broken. Why? Because the `vertexList` property does not contain information about whether a vector shape is to be drawn closed or open. Every vector shape member has a `closed` property. To make this complete, you need to type the following:

```
myVector.closed = member(2).closed
```

The `closed` property is either 0 or 1. We don't need to know what it is in this case; we just need to set it to the same value as member 2's `closed` property.

As you learned in Chapter 12, when you set a variable to another variable that contains a list, both variables reference the same list unless you use the duplicate function to make a copy of the list. When you get the `vertexList` of a member, you are automatically getting a copy of that list, not a reference to it. After you put the `vertexList` into a variable and change it, it will not affect the vector shape from which you got it. You must reassign the changed list to the `vertexList` of the member for it to take effect. Type the following in the Message window:

```
set myVtList to the vertexList of myVector
set anotherVtList to the vertexList of member 3
```

`myVtList` now has a copy of the `vertexList` of member 4 (the member that `myVector` references). `anotherVtList` has a copy of the `vertexList` of member 3. Because we are manipulating a *copy* of the `vertexList`, not the actual vertices, we won't see the effects yet. Now add the points from one list to the other:

```
add myVtList, anotherVtList[1]
add myVtList, anotherVtList[2]
add myVtList, anotherVtList[3]
add myVtList, anotherVtList[4]
```

`myVtList` now contains eight vertices. Now assign this list to the `vertexList` of `myVector`, as follows:

```
set the vertexList of myVector to myVtList
```

You should see something similar to the shape in Figure 17-5.

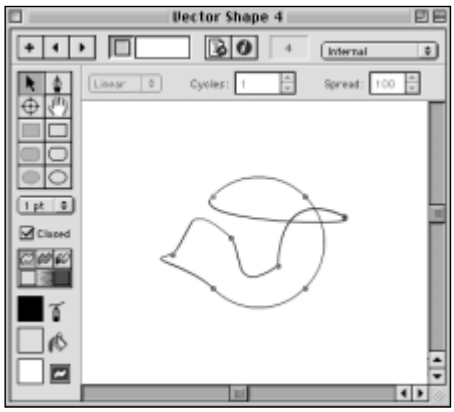

Figure 17-5: The circle in member 2 is joined with the curve in member 3.

Changing vertices

So far, the methods for changing vertices have been indirect: Put the `vertexList` of a member into a variable, change the variable, and reassign it to the `vertexList` of the member. There are also methods for changing vertices directly.

You can add vertices directly by using the `addVertex` command. Just pass in a vector shape member, an index, and a point. The *index* is where in the list you want to add the point. The *point* is the location of the new vertex you're adding, as in this example:

```
addVertex(member 1, 3, point(10, 10))
--or
member(1).addVertex(3, point(10, 10)
```

If you add vertices, you might also want to delete them You can use the `deleteVertex` command to delete a particular vertex. This command is even simpler. It takes a vector shape member and the index of the vertex you want to delete, as shown here:

```
deleteVertex(member 1, 3)
--or
member(1).deleteVertex(3)
```

Resizing vector shape data

In Chapter 15, you learned how to resize sprites. Resizing a vector shape sprite works great on the Stage, because vector shapes are resolution independent. But what if you want to resize the actual vector shape data in the member?

You can do it through the user interface by Command+Option+clicking (Ctrl+Alt+clicking) and dragging on the shape in the vector shape window. To do it at run-time, you need to do it using Lingo. This can be achieved quite easily by multiplying the `vertexList` of the member by your scaling factor and then setting the `vertexList` to the resulting value. The following line of code scales the vector shape by half:

```
member(1).vertexList = member(1).vertexList * 0.5
```

There are a few caveats with this method. If you are scaling down and up frequently, your shape may change slightly from rounding errors. You can avoid this situation by keeping the original `vertexList` in a variable and using that for your calculations.

Another problem is using vectors with multiple curves. This method will not work because your `vertexList` will contain a linear list that contains a symbol to delineate the different curves, `[#newCurve]`. The `vertexList` needs this information to know when a new curve begins. If you multiply a symbol by an integer, the result is an integer. *Multiplying a multi-curve `vertexList` by an integer corrupts the list.*

To change the location of a vertex, you can use the `moveVertex` command. It takes a vector shape member, an index, and two integers. The index is the vertex you want to move. The integers are the horizontal and vertical amount you are moving the vertex, as in the following:

```
moveVertex(member 1, 3, 25, -2)
-- or
member(1).moveVertex(3, 25, -2)
```

You can manipulate handles directly in a similar fashion. With the `moveVertexHandle` you need to specify the vector shape member, the index of the vertex, the index of the handle, and the horizontal and vertical amounts you are moving the handle, as follows:

```
moveVertexHandle(member 1, 3, 2, 10, 5)
--or
member(1).moveVertexHandle(3, 2, 10, 5)
```

Setting the fill

To fill a vector shape, you need to specify the color and the manner in which the shape is filled. The color should be of the type rgb. The `fillMode` of a vector shape can be set to `#solid`, `#gradient`, or `#none`. The shapes you've been drawing so far have been transparent. If you check their `fillMode` property, you see that they are set to `#none`.

This code makes the first member a red square:

```
member(1).fillMode = #solid
member(1).fillColor = rgb(255, 0, 0)
```

Creating a gradient

By setting the `fillMode` to `#gradient`, you can have a gradation in your vector shape:

```
member(1).fillMode = #gradient
```

To set a gradient, you need to specify a `fillColor` and an `endColor`. You already have set the `fillColor` to red, so now set the `endColor` to white, as follows:

```
member(1).endColor = rgb(255, 255, 255)
```

Your shape should now look like Figure 17-6.

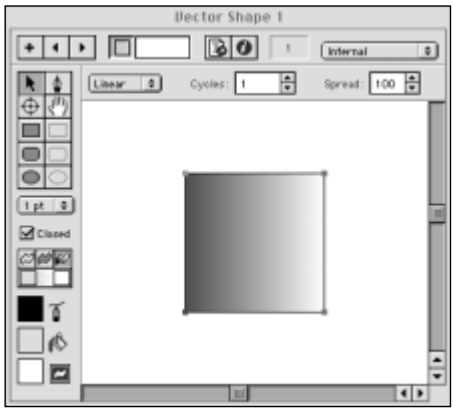

Figure 17-6: Changing the fill of a vector shape

There are several more Lingo commands that compare with the settings you can apply in the user interface shown in Figure 17-7:

- In the pop-up menu, you can specify the gradient: Linear or Radial. In Lingo, you would specify the `gradientType` property, and set it to `#linear` or `#radial`.
- The Cycles field corresponds to the `fillCycles` property — you can specify an integer from 1 to 7.
- The Spread field corresponds to the `fillScale` property.
- The Angle field corresponds to the `fillDirection` property — set it to the degree of the angle.
- The X Offset and Y Offset fields correspond to the `fillOffset` property. The `fillOffset` property takes a point so you can set both values in one step.

Figure 17-7: The different attributes that you can apply to a vector shape with a gradient.

Setting a stroke

As you might have noticed, there is still a stroke around the square vector shape in member 1. You can set the stroke by setting the `strokeWidth` property. A value of 0 is no stroke. You can set the thickness all the way to 100 through Lingo (as opposed to 12 in Director's user interface). You can change the color by setting the `strokeColor` property.

Mulitple curves

As you learned in Chapter 2, you can create vector shapes with multiple curves (Director 7 was limited to one path per member). This information is stored within the `vertexList` of the member. For Director to know a new path is beginning, an additional element is added to the `vertexList`, `[#newCurve]`. All subsequent vertices will belong to a new path. The following list contains two paths, forming a square within a square:

```
[[#vertex: point(-50, -50)], [#vertex: point(50, -50)],
[#vertex: point(50, 50)], [#vertex: point(-50, 50)],
[#newCurve], [#vertex: point(-30, -30)], [#vertex:
point(30, -30)], [#vertex: point(30, 30)], [#vertex:
point(-30, 30)]]
```

Try creating this vector shape from the Message window by following these steps:

1. Create the new member and place it in the variable `v` for easy access:

   ```
   v = new(#vectorShape)
   ```

2. Set the `vertexList` of the member:

   ```
   v.vertexList = [[#vertex: point(-50, -50)], [#vertex:
   point(50, -50)], [#vertex: point(50, 50)], [#vertex: point
   (-50, 50)], [#newCurve], [#vertex: point(-30, -30)],
   [#vertex: point(30, -30)], [#vertex: point(30, 30)],
   [#vertex: point(-30, 30)]]
   ```

3. Remember to close the shape:

   ```
   v.closed = 1
   ```

4. Then make it a filled shape:

   ```
   v.fillMode = #solid
   ```

5. Last, fill it with red:

   ```
   v.fillColor = rgb("#FF0000")
   ```

One of the nice things about Director is that you can have the Vector Shape window open while you do the preceding steps in the Message window. You can watch the effects of each line of code. If you have the Property Inspector open as well, select the new member after step #1 and watch the values change for the vector shape member (make sure you have the Vector tab selected and are viewing the Property Inspector in List View mode).

Vector and Flash Properties

There are many similarities between vector shapes and Flash members. Flash movies are made using vectors, just as the vector shapes. In fact, you use the Flash Xtra to draw the vector shapes. So, it is not surprising that quite a few Lingo commands work for sprites and members of each kind.

Antialias

When this property is set to 1 or TRUE, the member is antialiased. Setting it to 0 or FALSE makes the member aliased. The default is 1. Figure 17-8 shows the difference on two circle members.

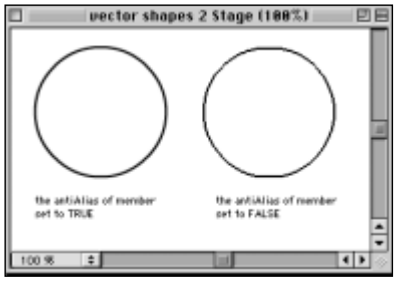

Figure 17-8: Antialiasing can make a big difference on static images.

Why would you ever want to turn it off? Images that do not have to be antialiased are rendered more quickly. You might have an option in your program that enables users to decide whether they prefer speed or quality. Static images benefit a great deal more from antialiasing than images that are moving quickly on the screen. The eye does not notice the flaws as much on a moving image as on one standing still just asking to be scrutinized.

defaultRect and defaultRectMode

The `defaultRect` of a member is the size it appears when it is on the Stage. When changed, all sprites referencing that member will change, unless they have been stretched.

The `defaultRectMode` is either `#flash` (default) or `#fixed`. After you change the `defaultRect`, the `defaultRectMode` changes to `#fixed`. To restore the member to its original size, set the `defaultRectMode` to `#flash`.

scale and scaleMode

Flash and vector shapes scale within their rects. The rect itself does not change size, just the image of the art. When a member is scaled 50 percent, then twice as much of it shows (see Figure 17-9). This property can be set for members or sprites.

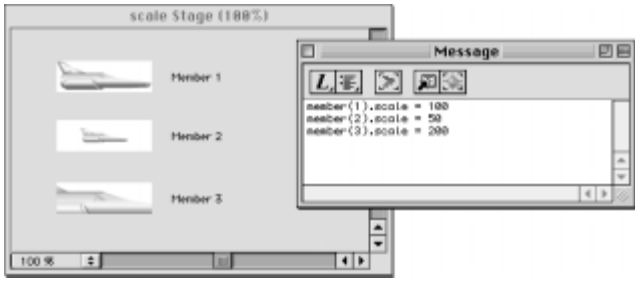

Figure 17-9: Scaling a Flash or vector shape member scales it within its bounding rectangle.

The scalemode.dir movie is in the EXERCISE:CH17 (EXERCISE\CH17) folder on the CD-ROM that accompanies this book.

The `scaleMode` is how the member or sprite scales. `Autosize` is the default. The `scaleMode` property is easier demonstrated than explained. For those of you not working through this at a computer, we'll do our best. The movie scalemode.dir, shown in Figure 17-10, has a Flash sprite and a vector shape sprite.

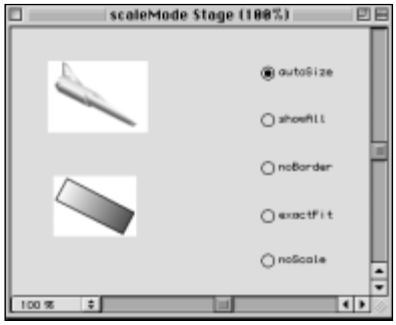

Figure 17-10: A movie to illustrate dynamically the differences that the scaleMode property can make. The copy ink on the images is intentional to show the bounding box as they rotate.

Each of the sprites in scalemode.dir rotates as the movie plays. When set to `#autoSize` — the default for Director — they react as you might expect. Because the Flash asset Xtra was first introduced with Director 6.5, movies that used it have the Flash members defaulting to `#showAll`.

- `#autoSize`: Rotates the image as you might expect, maintaining the same scale for the artwork.
- `#showAll`: Maintains the aspect ratio of the original cast member. Thus, as a rectangular shape rotates, it appears to shrink and grow.
- `#noBorder`: Maintains the aspect ratio also, but it crops the member so that the member appears to grow within its sprite's rect.
- `#exactFit`: Squeezes the shape into the rect, causing it to distort.
- `#noScale`: Looks the same as `#autoSize`, unless the sprite has been stretched. If the sprite is made smaller than the original member, then the artwork is cropped within that sprite's rect.

Figure 17-11 illustrates the different results based on the setting of the `scaleMode` property.

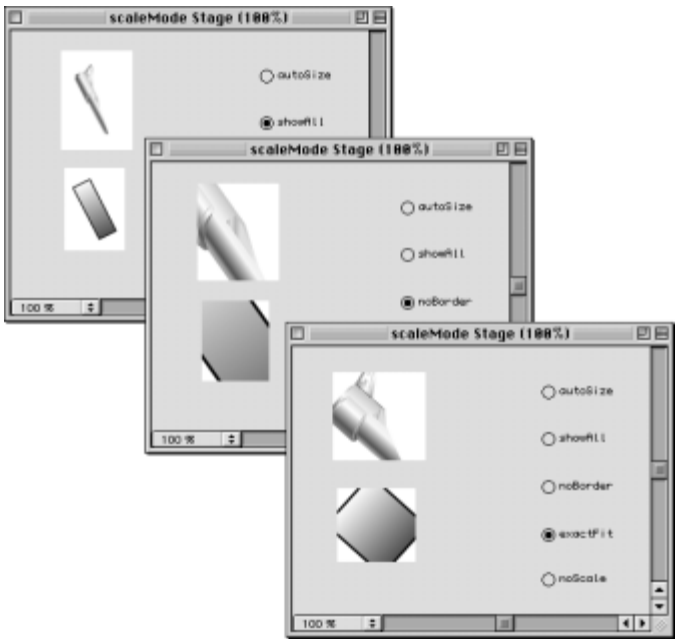

Figure 17-11: The scalemode.dir movie in action

Understanding view properties

The `viewScale` is similar to the `scale` property. Where setting the scale to 50 percent shows twice as much of the image within the rectangle (making it look smaller), setting the `viewScale` to 50 percent shows you half as much of the image within the rectangle, making it look larger. It is like looking through a camera. If you are looking at something and then cut the distance between you and it by 50 percent, it will be much larger in the picture. Figure 17-12 shows the results. Compare the difference in results shown by this illustration with that of Figure 17-9. This property can be set for members or sprites.

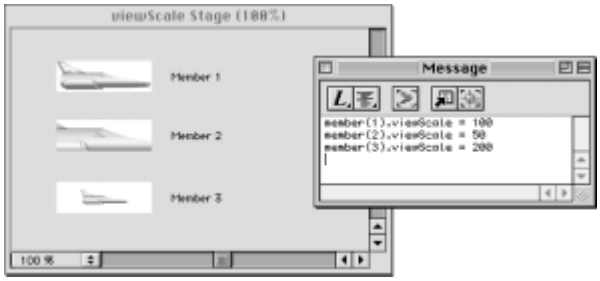

Figure 17-12: Changing the viewScale property

Keeping with the camera analogy, you could also move sideways, up or down, or some combination of those directions when viewing something. To do this with a Flash or vector shape, you set the `viewPoint`, `viewH`, or `viewV`. Moving 10 pixels is like stepping to the right, so the image moves to the left (see Figure 17-13). To move both directions at once, set the `viewPoint`.

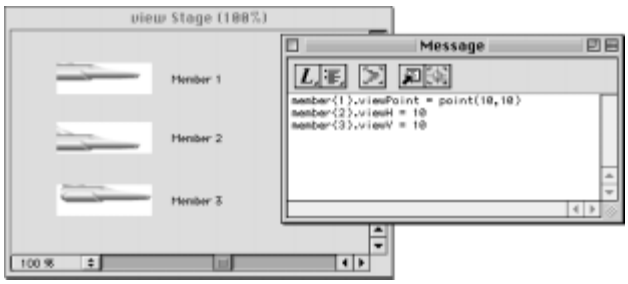

Figure 17-13: Changing the viewPoint property

Understanding origin properties

The originPoint property determines the point around which the Flash or vector shape scales or rotates. Like the viewPoint, viewH, and viewV, you have the originPoint, originH, and originV. When you change the origin, you also change the originMode. You can set the originMode as follows:

- #center is the default.
- #topleft causes the member to scale from the top-left corner.
- #point is a point you set with the originPoint, originH, or originV.

The images in Figure 17-14 have their scaleMode set to #noScale.

If you change the originPoint, you can restore the default by setting the originMode to #center.

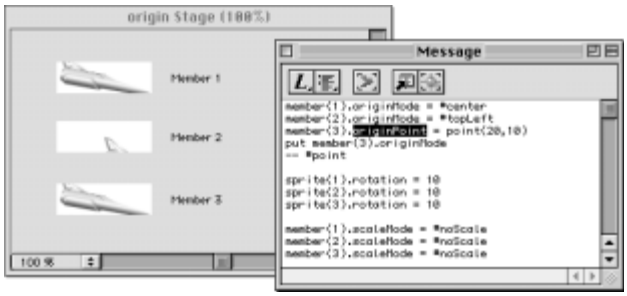

Figure 17-14: Changing the originMode property

Flash Only

The capabilities of Flash movies far exceed those of vector shapes, and there are many commands that apply only to Flash movies. Flash movies can be static or animated, they can have buttons, and they can stream.

Embedding or linking?

Director enables you to have Flash movies embedded in, rather than linked to the cast. There are advantages and disadvantages to both. It is convenient to have the media embedded, because you have fewer files to manage. Linking it enables you to update the Flash movie without having to modify the Director movie. Linking also enables you to take advantage of the commands regarding streaming. You can test

the linked property of a Flash member to determine whether it is linked or embedded. A nice aspect of the linked property is that you can both get it and set it. This enables you to create a new Flash member on the fly and a path to a Flash movie, such as a URL.

Controlling playback

A Flash movie can be a single frame of artwork or it can be many frames of animation. Director gives you control over the playback of these members. These commands operate on the Flash sprite, as opposed to the member:

- `play`: Starts a Flash movie sprite playing from the current frame.
- `the playing of sprite`: This property indicates whether the movie is playing. It is either 1 or 0—1 if it is playing; 0 if it is stopped. You can get this property, but you cannot set it.
- `the playBackMode`: You have three choices for this property: `#normal`, `#lockStep`, and `#fixed`. `#normal` plays the movie at the original tempo. `#lockStep` plays a frame of the Flash movie every time a Director frame plays. `#fixed` plays the movie at the rate to which the `fixedRate` property is set.
- `rewind`: This command rewinds the sprite back to frame 1.
- `hold`: This command stops the Flash movie from playing, but it doesn't stop the movie's audio.
- `stop`: This command stops the Flash movie from playing, and the movie remains on the current frame.

The box1.dir movie is in the EXERCISE:CH17 (EXERCISE\CH17) folder on the CD-ROM.

Open the box1.dir movie on the CD-ROM and play it (see Figure 17-15). You see the little green man slowly pop out of the box, even though the tempo is set to 60fps (see Figure 17-16). This is because the `playBackMode` of the sprite is set at its default value: `#normal`.

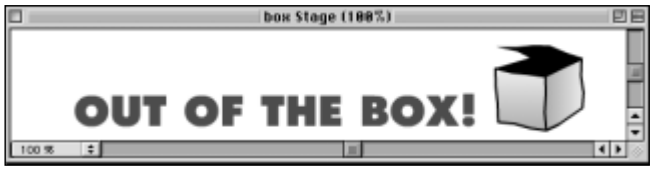

Figure 17-15: The beginning of box1.dir

Figure 17-16: The end of box1.dir

To make it move at the same rate as the movie, you need to set the `playBackMode` to `#lockStep`. A good way to do this is with a behavior attached to the sprite. This way, you can move the sprite to different places in the Score, if necessary, without having to worry about changing the code. Create and apply the behavior shown in Figure 17-17.

Figure 17-17: Setting the playBackMode

Now you can change the tempo of the movie, and the tempo of the Flash sprite changes accordingly. Setting the `fixedRate` of a sprite does not automatically change the `playBackMode` to `#fixed`. If you want the frame rate of the Flash movie to move at a rate different from the Director tempo, you need to set the `fixedRate` to the frame rate at which you would like it to play, and then you need to set the `playBackMode` to `#fixed`, as shown in Figure 17-18.

Figure 17-18: Setting the playBackMode

There are no commands to make a Flash movie play backwards, so how would you do it? Given `the frameCount of member` and `the frame of sprite` commands, you can write a short behavior that does just that (see Figure 17-19).

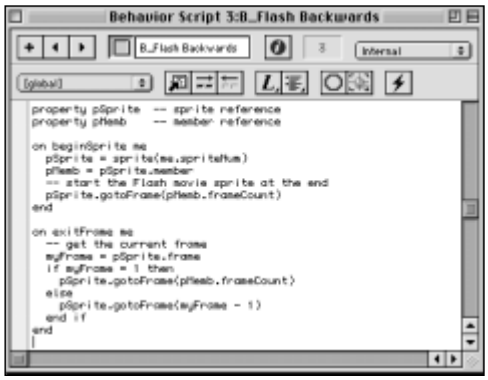

Figure 17-19: A behavior to play a Flash sprite backwards

The first part of the behavior in Figure 17-19 is for convenience. We create properties to hold a reference to the sprite and a reference to the member of the sprite. This enables you to use simpler syntax in the `exitFrame` handler. The last statement in the `beginSprite` handler `pSprite.gotoFrame(pMemb.frameCount)` might be a little confusing. The first part that executes in that statement is the part in parentheses, `pMemb.frameCount`. This returns the number of frames in the Flash member, because `frameCount` is a member property, not a sprite property. The next part that executes is the `gotoFrame` command, which is a Flash sprite command. The `gotoFrame` command takes an integer or string as an argument (a string would be the label to go to)—in this case, an integer is returned from `pMemb.frameCount`.

In the first statement of the `exitFrame` handler, the variable `myFrame` is created and given the value of the frame the Flash sprite currently is playing, using `the frame of sprite` property (the dot syntax form of it). If it is on the first frame, the sprite is told to go to the last frame; otherwise, it is told to go to the one just before the current one.

Streaming Flash

As mentioned earlier, Flash movies can stream. Whereas the playback commands apply to sprites, the streaming commands apply to members:

- `the bufferSize of member`: This property controls how many bytes can be streamed into memory at one time. The default it 32,768 bytes.
- `the bytesStreamed of member`: This is the number of bytes that have loaded.
- `clearError`: Resets the error state of a Flash member being streamed. Use this in conjunction with `the state of member` property.

- `state of member`: This property is the current state of the member that is streaming. 0 means it is not in memory. 1 means the header is loading. 2 means the header has finished loading. 3 means the member's media is loading. 4 means the member has finished loading. –1 is returned if there is an error.
- `stream`: This command/function enables you to stream a number of bytes of a Flash movie. It returns the number of bytes actually streamed, which might be fewer than the amount specified.
- `the streamMode of member`: This member property controls how a Flash movie is streamed. There are three options: `#frame`, `#idle`, and `#manual`. `#frame` is the default, it streams part of the member each time the playback head moves. `#idle` streams whenever there is an idle event, at least once per frame even if no idle event is generated. `#manual` streams a member only when the `stream` command is used.
- `the streamSize of member`: This property is the number of bytes in the stream for the member.

Controlling interaction

Flash movies have limited interactivity. You *can* have buttons that perform some predefined commands. Interaction of Director and the Flash movie is controlled on the sprite and member level.

- `the actionsEnabled of sprite or member`: This setting enables you to turn on or off the action in a Flash sprite or member. ON is the default. Possible values are 1 for ON and 0 for OFF.
- `the buttonsEnabled of sprite or member`: This setting enables you to control whether buttons in a Flash movie are active or inactive. ON is the default. Possible values are 1 for ON and 0 for OFF.
- `the clickMode of member`: This property is how a Flash member or sprite detects `mouseUp`, `mouseDown`, `mouseEnter`, `mouseWithin`, and `mouseLeave` events. It has three possible values: `#boundingBox`, `#opaque`, and `#object`. `#bounding box` is the rect of the sprite. `#opaque` is the default: It affects only opaque areas of the sprite if the ink is set to background transparent; otherwise, it acts like `#boundingBox`. `#object` works like opaque, but the ink for the sprite does not have to be background transparent.
- `the eventPassMode of sprite`: A Flash sprite can deal with mouse events one of four different ways: `#passAlways`, `#passButton`, `#passNotButton`, and `passNever`. `#passAlways` is the default; it always passes mouse events. `#passButton` passes mouse events when a button in a Flash movie is clicked. `#passNotButton` passes mouse events only when something other than a button is clicked. `#passNever` never passes mouse events.

- `hitTest`: Enables you to determine which part of a Flash movie is over a specific location. There are three possible return values: `#background`, `#normal`, and `#button`. `#background` means the location is over the background area of a Flash sprite. `#normal` means the location is over a filled object. `#button` means it is over a Flash button.
- `the mouseOverButton of sprite`: This property indicates whether the mouse is over a button in a Flash sprite.

Sometimes, the easiest way to understand something complex is to make a simple example that just illustrates what you are trying to figure out. Figure 17-20 shows a simple Flash movie. The rectangular shape is a button — the circle is just another Flash object. The white area is the bounds of the Flash member, and the gray around it is the color of the Stage.

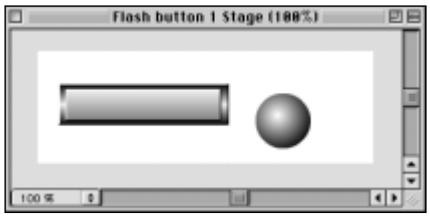

Figure 17-20: A simple Flash movie to demonstrate Lingo/Flash interaction

The button has an action applied to it in the Flash 3 application. Flash 3 is not included with Director 7, but this feature is important to mention, because the capability to interact with Director is not documented in the Flash manual. By selecting the button, choosing Modify ➪ Instance, clicking the Actions tab, and then choosing Get URL from the + pop-up menu, you are presented with the dialog box shown in Figure 17-21.

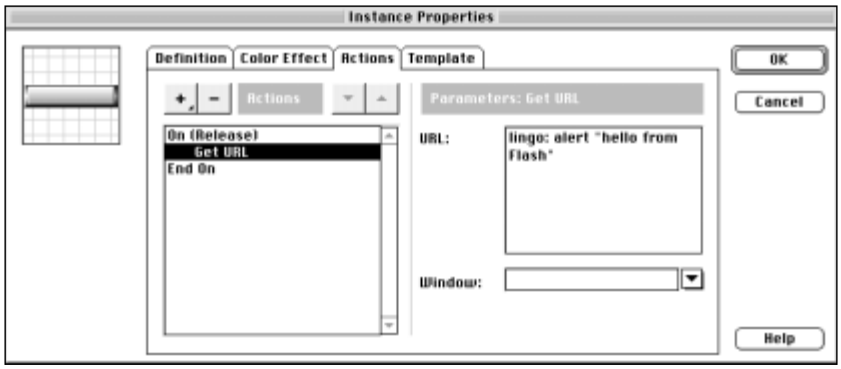

Figure 17-21: A Flash dialog box showing where you can enter Lingo to be executed in Director from a Flash button.

The URL text field is initially empty; in it, you can add the Lingo that you want to execute. You have three different options for what you put in the field. You can just put a string in there and then use an `on getURL` handler in Director. You can put a Lingo statement, as we did, but it must be prefaced with "lingo:". Alternatively, you can put a call to a handler, such as `event: MyHandler`. The word "event:" must be there and "MyHandler" needs to be a handler you defined in your Director movie.

When this button is clicked in Director, `alert "hello from Flash"` executes just as if it were written in a Director script. This occurs on a `mouseUp`. Flash enables you to have it happen on Release (`mouseUp`), Press (`mouseDown`), Release Outside (`mouseUpOutside`), Roll Over (`mouseEnter`), Roll Out (`mouseLeave`), Drag Over, and Drag Out, as shown in Figure 17-22. There aren't any corresponding Lingo event handlers for those last two, but you could create them easily enough.

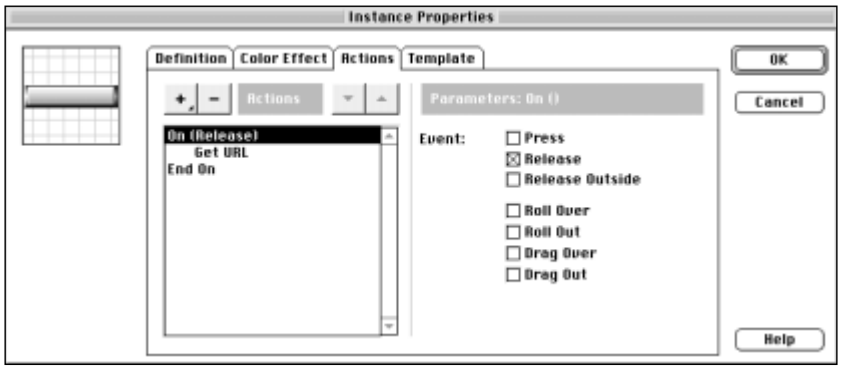

Figure 17-22: Flash enables you to select check boxes to determine when the GetURL statement is executed.

Now open the flash button 1.dir movie, which you can find in the EXERCISE:CH17 (EXERCISE\CH17) folder on the accompanying CD-ROM. It has the Flash movie already imported. Go ahead and play the movie and click the button. You get a "hello from Flash" alert dialog box. Now create a new behavior for the Flash sprite, as shown in Figure 17-23.

Figure 17-23: Create and then apply this behavior to your Flash sprite.

Play the movie and click the button and other parts of the Flash movie. When you click the Flash button, the "hello from Flash" dialog box pops up, and right after it comes the "Hello from Director" dialog box. If you click elsewhere on the Flash sprite, you still get the "Hello from Director" dialog box, because this script is applied to the sprite — button, background, and all. This is because the default `eventPassMode` for a sprite is `#passAlways`. This means that the sprite passes the `mouseUp` event to the sprite's behaviors. While the movie is playing, type the following in the Message window:

```
set the eventPassMode of sprite 1 to #passButton
```

Now when you click the Flash button, you still get both messages, but when you click elsewhere on the Flash sprite, you don't get any alerts. This is because you have told the sprite only to pass events when a button is clicked. Of course, if you click the Stage, no message for any of these examples appears, because we don't have a `mouseUp` in the Frame behavior. While the movie is still playing, type the following code in the Message window:

```
set the eventPassMode of sprite 1 to #passNotButton
```

Click the button and other areas of the sprite. When you click the Flash button, you see the Message "hello from Flash." When you click anywhere besides a button, you see "Hello from Director." This setting only passes events to a sprite's behaviors if the sprite was clicked, but not on one of its buttons. This is a good way to avoid having conflicting scripts execute, one from Flash and another from Director. Next, try this:

```
set the eventPassMode of sprite 1 to #passNever
```

Now, you've told the sprite never to pass events. It just sucks them up and uses them when it needs to, such as when you click one of its buttons. Finally, we have the following:

```
set the eventPassMode of sprite 1 to #passAlways
```

This sets it back to its default behavior. Next, try typing this code:

```
set the actionsEnabled of sprite 1 to 0
```

This statement disables the actions in the Flash sprite. The button still goes up, down, and has a rollstate. You can use this in conjunction with `#passButton` for the `eventPassMode` to have your Director scripts replace ones embedded in the Flash movie. If you want to disable all the buttons in a Flash sprite, just set `buttonsEnabled` to 0, or FALSE, as follows:

```
set the buttonsEnabled of sprite 1 to 0
```

You can use this movie to understand some of the other interaction-based events. Open the Watcher window and enter the statements shown in Figure 17-24. Remember that you don't enter the equals sign or anything after it; that information is what the expression evaluates to. Play the movie and watch as you roll the mouse over different areas of the Flash sprite. You see the value of the `hitTest` (sprite 1, the `mouseLoc`) expression change from `#background` to `#button` and to `#normal` when it is over a nonbutton Flash element. The `mouseOverButton` is a quick way sto test whether the cursor is over a button in a Flash sprite.

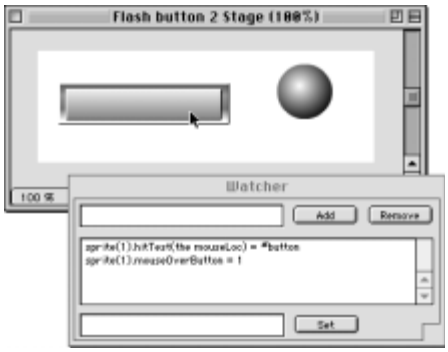

Figure 17-24: Viewing results using the Watcher window

Summary

Some of the things you learned in this chapter include the following:

- How to create and modify vector shapes.
- Vector shapes and Flash movies have many of the same commands.
- Flash movies can be embedded or linked.
- There are many Flash-only properties dealing with playback, streaming, and interactivity.
- Flash movies can have Lingo embedded in them.

The next chapter explores using Lingo with sound.

✦ ✦ ✦