CHAPTER 26

Imaging Lingo

In This Chapter

Learning about the image object

Copying images directly to the Stage

Creating your own images in memory

Drawing lines and shapes

Macromedia Director has been missing a feature coveted by developers for years. Programmers from other languages struggled to learn Director's paradigm of Stage, sprites, and Score. There was no way to draw directly to the Stage, to get or set a single pixel color, to composite several cast members off-screen, or to get the image of the Stage without resorting to third-party Xtras. As of version 8, developers need covet no longer—imaging Lingo has arrived! Imaging Lingo is not implemented by an Xtra (third-party or Macromedia), but is built right into Director and its playback engines (Shockwave and projectors).

The Image Object

The addition of imaging Lingo is achieved through the creation of a new datatype called the *image object*. You may have used the picture property of a bitmap or text member in the past. Although there are some similarities, you probably won't use the picture property as often after reading this chapter. The image object gives you far greater flexibility and much faster access time to the property (*hundreds* of times faster). The only place where using the picture property is faster is setting the picture of a member to that of another member (it's about twice as fast). Image properties can be found in the Stage, movies in a window (MIAWs), bitmap, text, vector shape, and Flash members, but they can also be created from scratch and stored in a variable.

Each image object has a series of properties (see Table 26-1). The `useAlpha` property is the only one you can set. To access an image object property, you can use standard or dot notation, as follows:

```
set w to the width of the image of member 1
-- or dot notation
w = member(1).image.width
```

Table 26-1
Image Property Properties

Property	*Get*	*Set*
width	X	
height	X	
rect	X	
depth	X	
useAlpha	X	X

Several functions, shown in Table 26-2, are associated with an image object. In Table 26-2, *myImage* is an image object and parameters surrounded by <> are optional. The functions in Table 26-2 are shown in dot notation, but they can be expressed in standard notation by moving the object to the first parameter of the function, as in the following example:

```
copyPixels(myImage, sourceImage, destRect_or_quad, sourceRect,
<paramList>)
```

Table 26-2
Image Object Functions

Functions
myImage .copyPixels(*sourceImage*, *destRect_or_quad*, *sourceRect*, <*paramList*>)
myImage .createMask( )
myImage .createMatte(<*alphaThreshold*>)
myImage .crop(*cropRectangle*)
myImage .draw(*left*, *top*, *right*, *bottom*, *color*, <*parameters*>)
myImage .duplicate( )
myImage .extractAlpha( )
myImage .fill(*left*, *top*, *right*, *bottom*, *color*, <*parameters*>)
myImage .getPixel(*x*, *y*)
myImage .setAlpha(*intOrImage*)
myImage .setPixel(*x*, *y*, *intOrColor*)
myImage .trimWhiteSpace( )

All the functions in Table 26-2 require an image object. One very important function not mentioned in the table is not a function of image objects, but one used to create them. To create an image object, you use the `image()` function.

image()

Use this function to create a new image object. It has three required parameters and two optional parameters.

Syntax:

```
newImage = image(width, height, bitDepth, <alphaDepth>,
<paletteMember>)
```

The first three parameters for the `image()` function should seem logical: An image you see on your screen has dimension (`width` and `height`) as well as a particular bit depth (`bitDepth`). These parameters are in the form of integers. `bitDepth` can be 1, 2, 4, 8, 16, or 32. The optional parameter `alphaDepth` can be 0 or 8 and is only used for 32-bit images; this parameter specifies the bit depth for the image's alpha channel. The last parameter, `paletteMember`, can be either a symbol for one of the built-in palettes (such as `#grayscale`, `#systemMac`, and so on) or a palette member in the Cast window (such as the member `myColors`).

An image object is not something you see onscreen. You see the actual image only when you copy it to the Stage's image or to a member that is being referenced by a sprite. Creating a new image object is simple. Try the following in the Message window:

```
myImage = image(64, 64, 32)
```

That's it! Not very impressive, though. What you now have is a reference to an image in memory. Try putting the `myImage` variable to the Message window, as follows:

```
put myImage
-- <image:8592670>
```

Now assign `myImage` to a new variable, `sameImage`, as shown here:

```
set sameImage to myImage
put sameImage
-- <image:8592670>
```

Image objects in variables work like lists in that if you assign `myImage` to another a variable, they will both reference the same image, as opposed to creating a duplicate. Note how both `myImage` and `sameImage` now reference `<image:8592670>`. If you need to create a duplicate, use the `duplicate()` function.

duplicate()

If you want to create a copy of an image and place it into a variable, you need to use the `duplicate()` function.

Syntax:

```
set anotherImage to duplicate(myImage)
anotherImage = myImage.duplicate()
```

The `duplicate()` function returns a copy of the image object. Enter the following in the Message window:

```
myImage = image(64, 64, 32)
put myImage
-- <image:8592670>
anotherImage = myImage.duplicate()
put anotherImage
-- <image:b39e094>
```

The preceding code first creates an image object, as was done previously. Next, it is put to the Message window so that we can see which image we are referencing. Then we duplicate `myImage` and put the reference to the duplicate into `anotherImage`. Last, we put `anotherImage` to the Message window to show that it truly does reference a difference chunk of memory.

Member, Stage, and MIAW images

Many graphic members have image properties, such as bitmaps, text (though not fields), Flash, and vector shapes (though not Tool Palette shapes). You can access the image property of the member directly or place it into a variable.

An image from a member, the Stage, or an MIAW, is a reference to the actual image, not a duplicate! Changes to this image variable using image functions change the actual graphic. The exceptions to this are the `crop()` function, which returns a cropped duplicate of the image, and, of course, the `duplicate()` function. If this is not what you want, then use the duplicate function, as explained earlier. If you plan to get and set information frequently to an image, it is faster to use an image object that does not reference a member, the Stage, or MIAW.

To illustrate how a member image reference in a variable will always be pointing to the member's image, try the following in the Message window:

```
myMembImg = member(1).image
alsoMyMembImg = myMembImg
put member(1).image
-- <image:b3881f8>
put myMembImg
```

```
-- <image:b3881f8>
put alsoMyMembImg
-- <image:b3881f8>
```

On one hand, this makes sense and "feels" consistent with other Lingo functionality, such as working with child objects and lists. When you relate its functionality to members, however, it is less clear. For example, if you are accustomed to working with the `vertexList` of vector shapes, it might seem odd, because putting the `vertexList` property of a vector shape member actually returns a copy of the `vertexList`. The important thing to remember is that the image you get from a member, the Stage, or MIAW is a reference to that image, not a duplicate.

The image property of a member is settable, but not the image of the Stage or MIAW. To draw graphics to the Stage or MIAW, you must use the `copyPixels()` function.

copyPixels()

Although the concept of sprites may not be new to a programmer unfamiliar with Lingo, the method of using them in the Score and controlling them with Lingo is alien. With the new image objects, Lingo may seem a little more programmer friendly. Using image objects, a programmer can build sprites or the whole screen image in an image object and then copy the image to the Stage. The `copyPixels()` function is the method to copy images to the Stage and MIAW image properties. It can also be used between other image objects.

Syntax:

```
copyPixels(myImage, sourceImage, destRect_or_quad, sourceRect,
<paramList>)
myImage.copyPixels(sourceImage, destRect_or_quad, sourceRect,
<paramList>)
```

You can find the movie world1.dir in the EXERCISE:CH26 (EXERCISE\CH26) folder on the CD-ROM that accompanies this book.

The best way to understand how `copyPixels()` works is to see it in action. Open the movie world1.dir from the CD-ROM. This movie has several images of a globe, but for now we'll just concern ourselves with getting one image onto the Stage. In the Message window, type the following code:

```
(the stage).image.copyPixels(member(1).image, member(1).rect,
member(1).rect)
updateStage
```

The `updateStage` command is necessary because the movie isn't running. If the movie were playing, the image would appear as soon as the playback head moved. Your screen should look something like Figure 26-1.

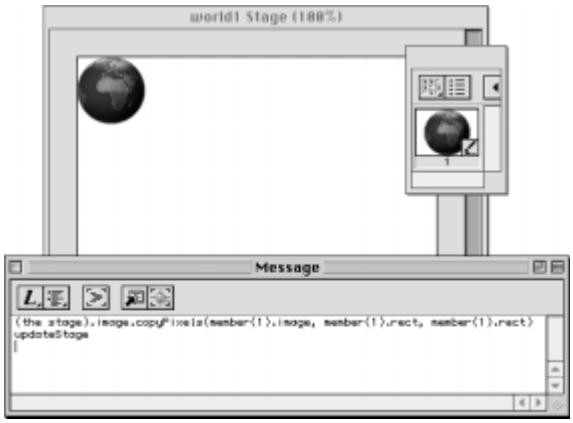

Figure 26-1: Using copyPixels( ) from the Message window

The image is drawn in the upper-left corner of the Stage because the `destRect_or_quad` parameter was the rect of the member, which is `rect(0, 0, 60, 60)`. The image will stay on the Stage until that area of the Stage is refreshed.

Now let's animate the globe. Create a frame behavior, as shown in Figure 26-2, in the script channel of frame 1.

The behavior has one property, `pCount`, which is used to keep track of which image the movie is on. `pCount` is initialized in the `beginSprite` handler:

```
property pCount

on beginSprite me
  pCount = 1
end
```

The real work is done in the `prepareFrame` handler. The first thing drawn is the image of a member onto the Stage; the member chosen is based on the value of `pCount`. The `copyPixels()` function will draw right over whatever was under it. After the image is drawn, `pCount` is checked to see if it exceeds the number of globe images in the move, and resets to 1 if it does:

```
on prepareFrame me
  (the stage).image.copyPixels(member(pCount).image, \
                               member(pCount).rect, \
                               member(pCount).rect)
  pCount = pCount + 1
  if pCount > 6 then
    pCount = 1
  end if
end
```

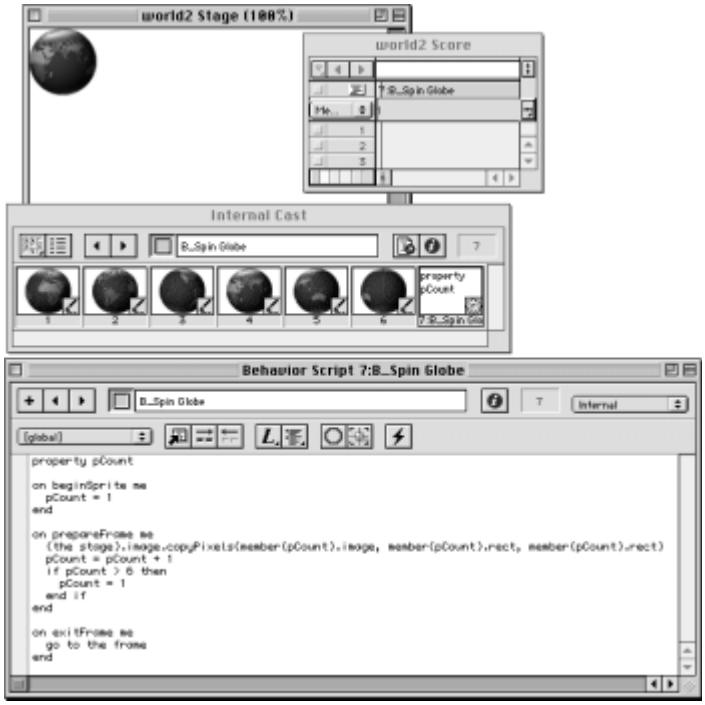

Figure 26-2: Animating the global by using copyPixels( )

The last handler in the behavior is the `exitFrame` handler, which just keeps the playback head looping on the same frame:

```
on exitFrame me
  go to the frame
end
```

When you're finished, save this movie as **world2.dir**.

You can find the world2.dir and many_worlds1.dir movies on the companion CD-ROM in the EXERCISE:CH26 (EXERCISE\CH26) folder.

Next, open the movie many_worlds1.dir from the CD-ROM. The many_worlds1.dir example is a variation of the preceding example. The slight difference between this movie and the preceding one is that the graphics of the Earth has an alpha channel. When imaging Lingo composites a graphic onto another graphic, the alpha channel of the image is used (unless the `useAlpha` property of the image is set to 0).

Figure 26-3 shows the code and the Stage when the movie is played.

Figure 26-3: Drawing the graphic in different locations

The B_Fill Screen frame behavior increments `pCount` and then checks it to make sure it does not exceed the number of globe bitmaps. Next, two variables containing points are created. The first, `pt1`, uses the `random()` function to pick a number from 0 to 320, minus the width of the member for the `locH` of the point. The width of the member is subtracted so that it does not draw outside the Stage. The same is done for the `locV` of `pt1`. Next, `pt2` is based on the values of `pt1`, plus the dimensions of the member being used. Last, the image of the current member is copied to the Stage at the location determined by `pt1` and `pt2`:

```
property pCount

on prepareFrame me
  pCount = pCount + 1
  if pCount > 6 then
    pCount = 1
  end if
  pt1 = point(random(0, 320-member(pCount).width), \
              random(0, 240-member(pCount).height))
  pt2 = point(pt1.locH + member(pCount).width, \
              pt1.locV + member(pCount).height)
  (the stage).image.copyPixels(member(pCount).image, \
                               rect(pt1, pt2), \
                               member(pCount).rect)
end
```

Finally, we loop on the frame in the `exitFrame` event handler:

```
on exitFrame me
  go to the frame
end
```

When you're finished, save this movie as **many_worlds2.dir**. As with the preceding example, you can find the finished many_worlds2.dir on the CD-ROM that accompanies this book.

Using quads

As the syntax for `copyPixels()` suggests, when using `copyPixels()` you are not restricted to copying from rect to rect; you may also copy from rect to quad. This capability enables you to take all or part of an image from a graphic and display it in any shape of quad. You cannot take a quad shape from an image, though, only a rect. There is no quad data type, like there is a rect; the quad is a linear list of four points. As discussed in Chapter 15, quads are a good way to distort an image or to give an image a sense of perspective. A more utilitarian use is to use this property when you need to flip an image while using imaging Lingo.

The movie world_cube1.dir is in the EXERCISE:CH26 (EXERCISE\CH26) folder on the CD-ROM that accompanies this book.

Open the movie world_cube1.dir from the CD-ROM. In the frame script for frame 1, create the behavior shown in Figure 26-4. Then play the movie. Figure 26-4 shows the movie after it starts playing.

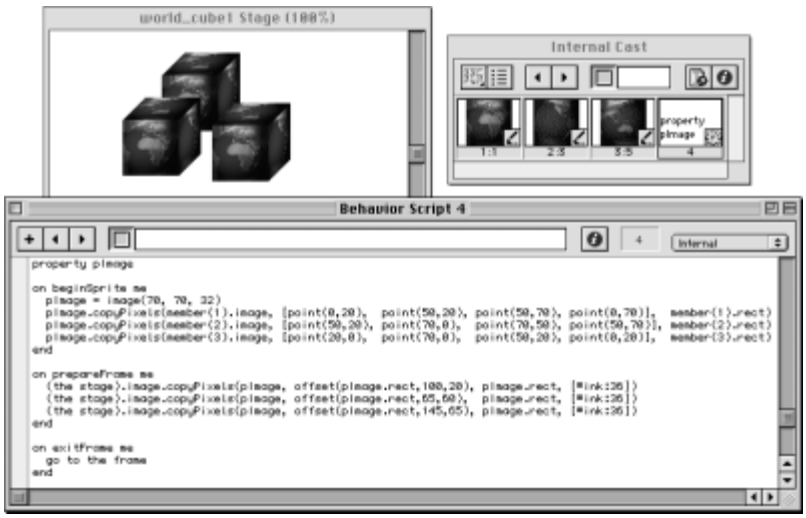

Figure 26-4: Using a quad as a parameter in the copyPixels( ) function

All of the action occurs in the frame behavior. First, in `beginSprite` an image object is created and placed into the property `pImage`, which is declared at the top of the script window:

```
pImage = image(70, 70, 32)
```

Then, the three faces of the cube are drawn by means of `copyPixels()`, with a quad as a parameter:

```
pImage.copyPixels(member(1).image, \
                  [point(0,20), point(50,20), point(50,70),
point(0,70)], \
                  member(1).rect)
pImage.copyPixels(member(2).image, \
                  [point(50,20), point(70,0), point(70,50),
point(50,70)],\
                  member(2).rect)
pImage.copyPixels(member(3).image, \
                  [point(20,0),  point(70,0), point(50,20),
point(0,20)], \
                   member(3).rect)
```

The pixels from `pImage` are copied in the `prepareFrame` handler (you cannot draw to the Stage from the `beginSprite` handler). The image is copied three different times into different locations (note the use of `offset()`). To achieve this, see Chapter 12 for more info on using `offset()`.

```
(the stage).image.copyPixels(pImage,
offset(pImage.rect,100,20), \
                             pImage.rect, [#ink:36])
(the stage).image.copyPixels(pImage, offset(pImage.rect,65,60),
\
                             pImage.rect, [#ink:36])
(the stage).image.copyPixels(pImage,
offset(pImage.rect,145,65), \
                             pImage.rect, [#ink:36])
```

You might notice a list, `[#ink:36]`, as the last parameter. It contains one item, specifying the ink type to use. Optional parameters are discussed next.

An easy and common mistake to make when coding is forgetting to add the `.image` to `(the stage)`. The following will cause a Handler not found in object error:

```
-- the below won't work!
(the stage).copyPixels(pImage,
offset(pImage.rect,100,20), \
                       pImage.rect, [#ink:36])
```

It looks a lot like the preceding code, and when you are getting bleary-eyed, it can be easy to miss.

Optional parameters for copyPixels()

The last parameter for `copyPixels()` is optional; it is in the form of a property list. Table 26-3 shows the properties that can be used with `copyPixels()`. You can use as many or as few of these parameters as you need.

Table 26-3
Properties in the copyPixels() Optional Parameter List

Property	*Type*	*Example*
`#bgColor`	color object	`rgb(0, 0, 0)`
`#blendLevel`	integer from 0 to 255	`128`
`#color`	color object	`rgb(255, 123, 0)`
`#dither`	Boolean value*	`TRUE`
`#ink`	symbol or integer	`#copy`
`#maskImage`	mask object	`imageObj.createMask()`
`#maskOffset`	point	`point(10, 25)`
`#useFastQuads`	Boolean value*	`FALSE`

* A Boolean value is either true or false. 1 or 0 (for TRUE or FALSE, respectively) can also be used.

This added parameter list gives you all of the compositing power you have when using the Director user interface or sprite Lingo. We'll examine each of these properties in more detail in the following sections.

#bgColor

The `#bgColor` property is the background color for the image. The type of value for this property is a color object. The default is white, `rgb(255, 255, 255)`. You can use other colors to create different effects.

#blendLevel

The `#blendLevel` property specifies the amount of transparency applied to the pixels being copied. This value is an integer from 0 to 255. 0 is completely transparent and 255 is completely opaque. Setting this value to any value other than 255 automatically switches the `#ink` property to `#blend` ink (`#blendTransparent` if the `#ink` was specified as `#backgroundTransparent`).

Alternatively, you can use `#blend` as a property in place of `#blendLevel`. Using `#blend` instead of `#blendLevel` enables you to use a value range of 0 to 100. 0 is transparent, and 100 is opaque.

#color

The `#color` property is the foreground color for the image. The type of value for this property is a color object. The default is black, `rgb(0, 0, 0)`. You can use other colors to create different effects.

#dither

The `#dither` property specifies whether the copied pixels will be dithered when copied to an 8- or 16-bit image. This property takes a value of either TRUE or FALSE, and the default is FALSE.

#ink

The `#ink` property is the type of ink applied when copying the pixels. The type of value for this property can be either a symbol or an integer. Table 26-4 shows the ink types in both symbol and integer format. Copy ink is the default.

Table 26-4
Ink Options

Ink Name	*Number*	*Symbol*
Add	34	`#add`
Add pin	33	`#addpin`
Background Transparent	36	`#backgroundTransparent`
Blend	32	`#blend`
Copy	0	`#copy`
Darken	41	`#darken`
Darkest	39	`#darkest`
Ghost	3	`#ghost`
Lighten	40	`#lighten`
Lightest	37	`#lightest`
Matte	8	`#matte`
Mask	9	`#mask`
Not Copy	4	`#notCopy`
Not Ghost	7	`#notGhost`
Not Reverse	6	`#notReverse`

Ink Name	*Number*	*Symbol*
Not Transparent	5	#notTransparent
Reverse	2	#reverse
Transparent	1	#transparent
Subtract	38	#subtract
Subtract Pin	35	#subtractPin

#maskImage

The #maskImage property enables you to mask images in the same manner you would use mask or matte ink on sprites. The value for this property is a mask image object. The mask image object can only be created by the createMask() and createMatte() functions. If the useAlpha property for the source image is set to TRUE, this property will be ignored.

#maskOffset

The #maskOffset property determines the position of the mask under the image. The value for this property is a point, and the default is point(0, 0). The offset is measured from the upper-left corner of the image.

#useFastQuads

The #useFastQuads property determines whether quads are rendered in higher or lower quality. This property takes a value of TRUE or FALSE, defaulting to FALSE. Setting it to TRUE is useful for quads that are just skewed or rotated; this will render the image more quickly. For all other cases, keep the property set to FALSE for a higher quality rendering of the image.

Getting/Setting Pixel Values

Another new feature of imaging Lingo is the capability to get or set the individual value of a single pixel in an image. Depending on your application, the getting or setting of individual pixels may not be as fast as you would hope. Oftentimes, you'll be better off using copyPixels() to set a larger area of pixels to get the job done more quickly, the fill function to fill an area, or the draw function to draw lines.

getPixel()

This function returns the color of the specified pixel.

Syntax:

```
myImage.getPixel(x, y <, #integer>)
myImage.getPixel(point(x, y) <, #integer>)
```

The first version of the `getPixel()` function takes two parameters: the horizontal and vertical location of the pixel whose value you are getting. The first pixel is in `point(0, 0)` not `point(1, 1)`. The optional third parameter, `#integer`, tells the function to return an integer value instead of an rgb color type. The second way to call `getPixel()` is to use a point instead of two integers. Generally, if you are calling this function excessively, it is faster to use the first version and include the optional parameter.

setPixels()

This function sets the color of the specified pixel.

Syntax:

```
myImage.setPixel(x, y, colorObj)
myImage.setPixel(x, y, int)
myImage.setPixel(point(x, y), colorObj)
myImage.setPixel(point(x, y), int)
```

The `setPixel()` function is similar to the `getPixel()` function in that the first parameter (or two) is the location of the pixel being set. The last parameter is either a color object or an integer specifying the color.

The movie stars1.dir is in the EXERCISE:CH26 (EXERCISE\CH26) folder on the CD-ROM that accompanies this book.

Using `setPixel()` is an easy way to create a star field. Open the movie stars1.dir from the CD-ROM. This example is our most complex, but is still pretty tame compared to some of the exercises creating the Lander game in Chapter 15! It still requires only one script. That script will create a star field, and then rotate the Earth, all without any sprites in the Score. Create the behavior shown in Figure 26-5 and place it in the frame script of frame 1. Play the movie and you'll see the unattractive stage color be replaced by a randomly created star field, with the Earth centered and spinning.

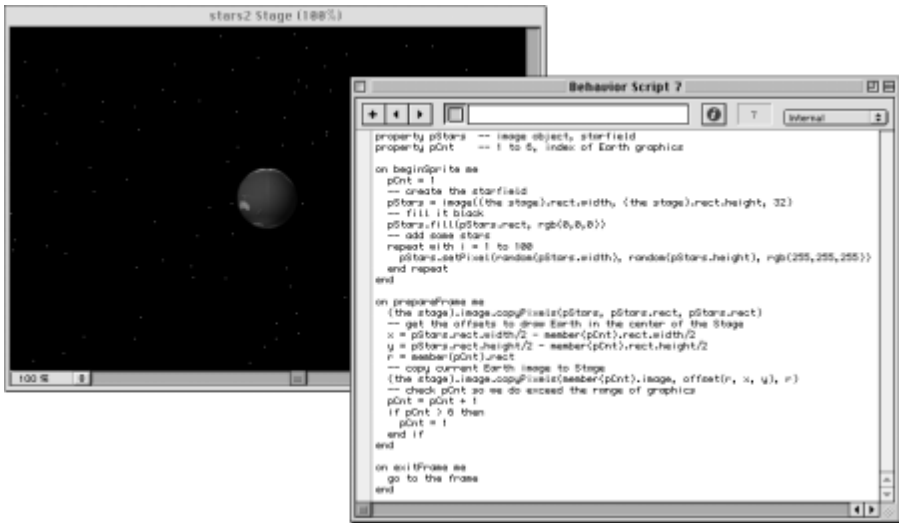

Figure 26-5: Creating a star field and an animated Earth

This script uses two properties, one for the image of the stars and one to keep track of which Earth bitmap we are on. These properties are:

```
property pStars  -- image object, starfield
property pCnt    -- 1 to 6, index of Earth graphics
```

In the `beginSprite` handler, `pCnt` is initialized to 1, which means that we'll draw the first Earth bitmap first. If you wanted to begin with another image showing, just change this assignment statement, as in this example:

```
pCnt = 1
```

Then we create an image object the size of the Stage and place it in the property `pStars`. You may want to change the size of the Stage for your movie, so the value is not hard-coded but instead is derived from the width of its rect:

```
pStars = image((the stage).rect.width, (the stage).rect.height,
32)
```

After that, we use the `fill()` function to fill that image with `rgb(0,0,0)`, which is black. This is a handy, though pedestrian, use for the `fill()` function, which we discuss in more detail later in this chapter:

```
pStars.fill(pStars.rect, rgb(0,0,0))
```

Last, we create 100 stars by using `setPixel()`.We randomly pick a location based on the height and width of the `pStars` image object and set that pixel to white:

```
    repeat with i = 1 to 100
      pStars.setPixel(random(pStars.width),
  random(pStars.height), rgb(255,255,255))
    end repeat
```

In the `prepareFrame` handler, we copy the star field over to the Stage first. If the star field was specified as smaller than the Stage, then the following line of code would place it in the upper-left portion of the Stage:

```
(the stage).image.copyPixels(pStars, pStars.rect, pStars.rect)
```

Then we need to figure out where to draw the Earth. We determine the *x* and *y* coordinates from the dimensions of the `pStars` image and the dimensions of the current member. To shorten the code, the rect of the current member is put into the variable `r`. These variables are used in the `offset()` in `copyPixels()`. While all of the Earth bitmaps happen to be the same dimensions, using the following code would work even if all the bitmaps were different sizes:

```
x = pStars.rect.width/2 - member(pCnt).rect.width/2
y = pStars.rect.height/2 - member(pCnt).rect.height/2
r = member(pCnt).rect
```

The Earth is copied to the Stage. You can imagine how much longer this line of code would be if we didn't create the preceding variables:

```
(the stage).image.copyPixels(member(pCnt).image, offset(r, x,
y), r)
```

The last thing to do is check to make sure `pCnt` has not exceeded the number of bitmaps we are using to do the Earth animation. Because we initialized `pCnt` in the `beginSprite` handler, we don't need to do the increment or checking until after we have used the current value of `pCnt`.

```
pCnt = pCnt + 1
if pCnt > 6 then
  pCnt = 1
end if
```

Drawing Lines and Shapes

Each image object supports drawing lines and shapes. This enables you to easily draw lines on your image, as well as ovals, rectangles, and round rectangles. There are two functions for drawing shapes, one is for drawing unfilled shapes and the other is for filled shapes.

draw()

With the `draw()` function, you can draw lines, as well as unfilled ovals, rectangles, and round rectangles.

Syntax:

```
myImage.draw(x1, y1, x2, y2, colorOrList)
myImage.draw(point(x, y), point(x, y), colorOrList)
myImage.draw(rect, colorOrList)
```

The `draw()` function comes in three forms. In each form, all but the last parameter describes a rectangle by using four integers, two points, or one rect. In each case, these values are used to draw a line unless the last parameter is a list that contains a `#shapeType` property. The last parameter can be a color object specifying the color of the line, or it can be a list of optional parameters.

Using the `draw()` function, it is easy to create a simple program that enables you to draw with the mouse.

You can find the movie drawline.dir on the CD-ROM in the EXERCISE:CH26 (EXERCISE\CH26) folder.

There is no starter movie for this example, but if you don't feel like typing in the code, you can open the finished version, drawline.dir, from the CD-ROM. Figure 26-6 shows the entire code that enables you to draw on the Stage. Just create the behavior shown in the frame script of frame 1, play the movie, and start drawing!

Figure 26-6: Drawing with the draw( ) function

This behavior requires two properties. One property keeps track of whether the mouse has been pressed, and the other remembers the previous location:

```
property pPressed
property pPrevLoc
```

In the `prepareFrame` handler, we check to see whether `pPressed` is TRUE. If it is, we check to see whether the mouse has moved by comparing `pPrevLoc` to `the mouseLoc`. If they are different, then we draw a line between those two points and set `pPrevLoc` to `the mouseLoc`:

```
if pPressed then
    if pPrevLoc <> the mouseLoc then
      (the stage).image.draw(pPrevLoc, the mouseLoc,
rgb(255,0,0))
      pPrevLoc = the mouseLoc
    end if
  end if
end
```

The `mouseDown` handlers are straightforward. If the mouse button has been pressed, then `pPressed` is set to TRUE and the location of the mouse is put into `pPrevLoc`. For both `mouseUp` and `mouseUpOutside`, `pPressed` is set back to FALSE so that no more drawing is done until the mouse button is pressed down again.

You might be wondering why we didn't do something simpler, such as the following:

```
on mouseDown me
  prevLoc = the mouseLoc
  repeat while the mouseDown
    if prevLoc <> the mouseLoc then
      (the stage).image.draw(prevLoc, the mouseLoc,
rgb(255,0,0))
      updateStage
      prevLoc = the mouseLoc
    end if
  end repeat
end
```

The preceding code works, and it will draw as fast as it can. The problem is that it is not very processor friendly; when a Director movie is running (including looping on a frame), it shares time with the processor. When it is in the middle of a `repeat` loop, that is the only task happening at that moment. If the clock on your computer displays seconds, you can easily see the difference. When drawing in the `prepareFrame`, you can see the seconds ticking on the clock. When you do the drawing within a `repeat` loop, the clock does not change until you release the mouse button.

List parameters for draw()

The `draw()` function's last parameter can be a property list containing properties describing the shape, line width, and color of the line.

#color

The `#color` property value is a color object that is the color of the line or border of the shape. The default is black.

#lineSize

The `#lineSize` property value is an integer specifying the width of the line. The default is 1.

#shapeType

The `#shapeType` property value is a symbol specifying the shape drawn. Only four values are available: `#oval`, `#rect`, `#roundRect`, or `#line`. The default is `#line`.

fill()

With the `fill()`, function you can create filled ovals, rectangles, and round rectangles with or without borders.

Syntax:

```
myImage.fill(x1, y1, x2, y2, colorOrList)
myImage.fill(point(x, y), point(x, y), colorOrList)
myImage.fill(rect, colorOrList)
```

List parameters for fill()

The list parameter for `fill()` can contain up to four different properties, specifying the background color, fill color, line width, and shape type.

#bgColor

The `#bgColor` property value is a color object that is the color of the line or border of the shape.

#color

The `#color` property value is a color object that is the color of the inside of the shape. The default is black.

#lineSize

The `#lineSize` property value is an integer specifying the width of the line. The default is 0.

#shapeType

The `#shapeType` property value is a symbol specifying the shape drawn. Only three values are available: `#oval`, `#rect`, or `#roundRect`. The default is `#rect`.

Summary

Some of the things you learned in this chapter include:

- You create new image objects with the `image()` function.
- Bitmaps, text, vector shape, and Flash members have image properties that you can get and set.
- The Stage and MIAWs have image properties you can get but not set. Changing those images requires the use of the `copyPixels()` function.
- You can get and set individual pixel values with the `getPixel()` and `setPixel()` functions.
- You can draw lines and shapes with the `draw()` and `fill()` functions.

✦ ✦ ✦