CHAPTER 15

Controlling Sprites with Lingo

In This Chapter

Manipulating sprites and their members

Controlling navigation

Using parent scripts with sprites

Score recording

Puppeting sprites

Manipulating sprites through Lingo is perhaps the heart of programming in Director. You've learned how to put sprites onto the Stage in earlier chapters (or through previous experience) — you've animated them through the user interface. That is just "click and go" results (which, in the right hands, can be impressive), but now you want your artwork to have a life of its own. You want it to move based on your input or in relation to other sprites on the Stage.

In this chapter, you modify a movie that already has some elements in it (instead of creating them from scratch). We tell you which movie to open to work on, and periodically we tell you to save the movie with a new number (for example, mars landing 02.dir, mars landing 03.dir, and so on). Every time we tell you to save a movie, one that parallels yours is on the CD-ROM under the same name. These movies are useful if you become stuck at any point or simply want to skip ahead.

Moving Sprites

To move sprites through Lingo commands, you need an understanding of the coordinate system in Director. Given experience in working with sprites on the Stage, or even page layout programs such as QuarkXPress, you might already be familiar with how coordinates are handled.

The loc of a sprite

The Stage area is similar to the first quadrant of a Cartesian coordinate system, with one important exception: the values along the *y*-axis are flipped — values increase (become greater than zero) as you move toward the bottom of the Stage (see Figure 15-1). The upper-left corner is point(0, 0); assuming your Stage area is 640 × 480, the farthest point in the lower-right corner is point(640, 480).

The Lingo term loc is short for location. The term locH is the location horizontally; locV is the location vertically. The loc of a sprite is a point() that is made up of the sprites locH and its locV. These values do not have to fall within the rectangle that makes up the Stage. A sprite can have negative coordinates or positive coordinates that are well outside the Stage's width and height.

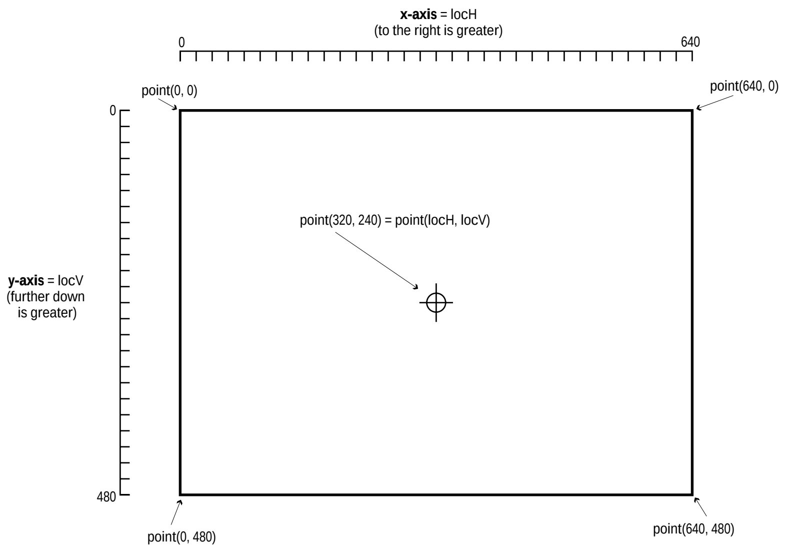


Figure 15-1: The coordinates of the Stage start at point(0, 0), in the upper-left corner.

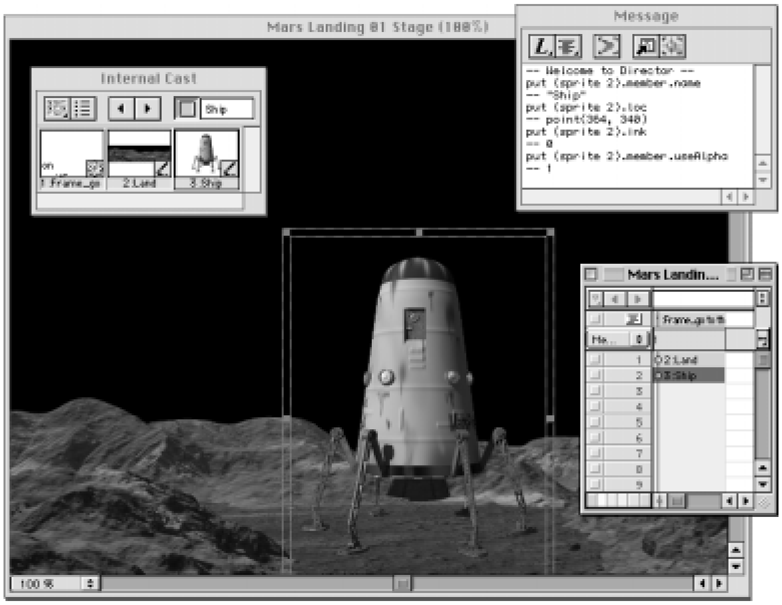
Note In the EXERCISE:CH15 (EXERCISE\CH15) folder on the CD-ROM you can find a movie called coordinates.dir. If you are unfamiliar with this coordinate system, open this movie and move the sprite around the screen with the mouse. A field shows you the location of the sprite as you move it. This should help you understand the relationship between the sprite's position on the screen and its loc.

On the CD-ROM

In the EXERCISE:CH15 (EXERCISE\CH15) folder on the CD-ROM, open the movie mars landing 01.dir.

Changing the locH

After you open up the movie mars landing 01.dir, note that it has two sprites on the Stage already. The background is the Martian terrain and a spaceship (see Figure 15-2).

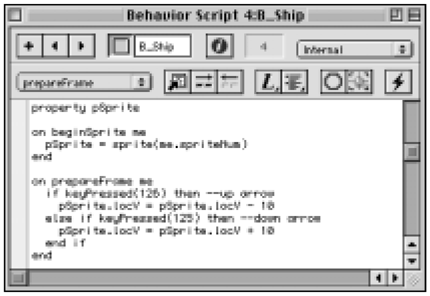


```
-- Welcome to Director --
put (sprite 2).member.name
-- "Ship"
put (sprite 2).loc
-- point(384, 340)
put (sprite 2).ink
-- 0
put (sprite 2).member.useAlpha
-- 1
```

Figure 15-2: The mars landing 01.dir movie

The only script in the movie is the frame script in frame 1. This script enables you to play the movie, and it will loop on the frame. For now, it does nothing other than loop on the frame.

Ctrl+click (right-click) on the ship and choose Script from the pop-up menu. Delete the `on mouseUp` handler that is automatically placed in the window, and type what is shown in Figure 15-3. Name it B_Ship. The *B* stands for behavior.



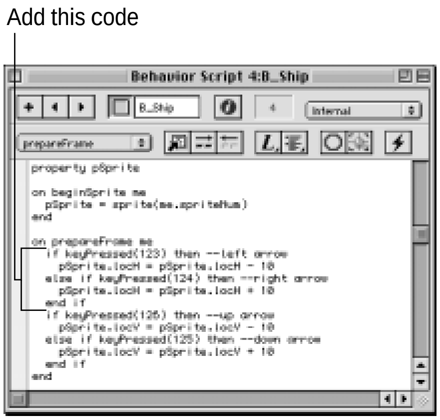
```
property pSprite

on beginSprite me
  pSprite = sprite(me.spriteNum)
end

on prepareFrame me
  if keyPressed(126) then --up arrow
    pSprite.locV = pSprite.locV - 10
  else if keyPressed(125) then --down arrow
    pSprite.locV = pSprite.locV + 10
  end if
end
```

Figure 15-3: The behavior for the ship sprite

In this behavior, you create the `pSprite` property, which is a reference to the sprite to which the behavior is attached. You also create a `prepareFrame` handler that is executed every time the playback head moves. Because we are looping on the frame, the playback head is moving at the tempo setting of the movie. Each time the `prepareFrame` executes, it checks whether the up arrow key or the down arrow key is pressed. If one is, then the sprite is moved 10 pixels up or down, respectively.

Changing the locV

Now let's add horizontal movement. Change the B_Ship behavior to include detecting right and left arrow presses, as shown in Figure 15-4. Play the movie and try the keys. You can press a left/right key and an up/down key at the same time to get diagonal movement. Before continuing, save the movie as **mars landing 02.dir**.

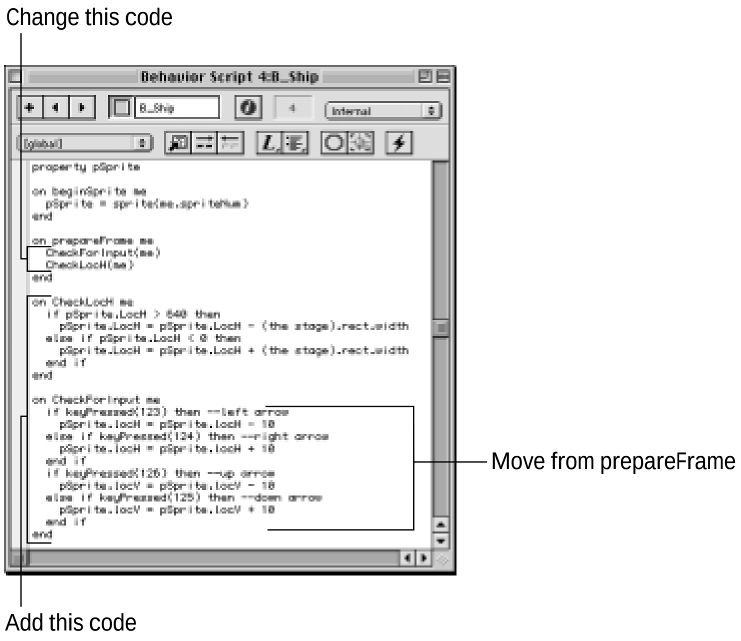


```
property pSprite

on beginSprite me
  pSprite = sprite(me.spriteNum)
end

on prepareFrame me
  if keyPressed(123) then --left arrow
    pSprite.locH = pSprite.locH - 10
  else if keyPressed(124) then --right arrow
    pSprite.locH = pSprite.locH + 10
  end if
  if keyPressed(126) then --up arrow
    pSprite.locV = pSprite.locV - 10
  else if keyPressed(125) then --down arrow
    pSprite.locV = pSprite.locV + 10
  end if
end
```

Figure 15-4: The behavior for the ship sprite expanded to support horizontal movement

Perhaps you want to have the ship wrap around when it reaches the other side of the screen. At each `prepareFrame`, you can check whether the sprite has moved beyond the width of the Stage. If it has, just move it over the width of the Stage.

Instead of further cluttering up the `prepareFrame` handler, now would be a good time to create some custom handlers. The first custom handler you create is `CheckForInput`. This handler should contain the contents of the `prepareFrame` handler. The next custom handler, called `CheckLocH`, checks the horizontal location of the sprite to determine if it needs to be moved. The `prepareFrame` handler calls each of these (see Figure 15-5). When finished, save the movie as **mars landing 03.dir**.



```
property pSprite

on beginSprite me
  pSprite = sprite(me.spriteNum)
end

on prepareFrame me
  CheckForInput(me)
  CheckLocH(me)
end

on CheckLocH me
  if pSprite.LocH > 640 then
    pSprite.LocH = pSprite.LocH - (the stage).rect.width
  else if pSprite.LocH < 0 then
    pSprite.LocH = pSprite.LocH + (the stage).rect.width
  end if
end

on CheckForInput me
  if keyPressed(123) then --left arrow
    pSprite.locH = pSprite.locH - 10
  else if keyPressed(124) then --right arrow
    pSprite.locH = pSprite.locH + 10
  end if
  if keyPressed(126) then --up arrow
    pSprite.locV = pSprite.locV - 10
  else if keyPressed(125) then --down arrow
    pSprite.locV = pSprite.locV + 10
  end if
end
```

Figure 15-5: A cleaned-up prepareFrame handler and two new custom handlers

Persistence of Sprite Changes

As you make changes to a sprite through Lingo, you might notice that each time you stop and rewind the movie, the sprites are restored to their original attributes (those specified in the Score).

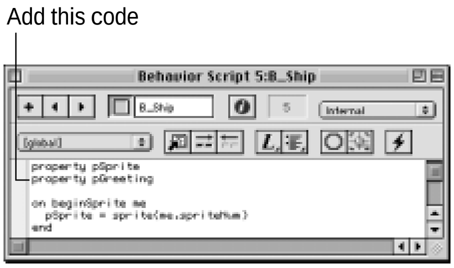
You also lose sprite changes if you go to a frame that is not within the span of the sprite, which is illustrated in the mars landing 04.dir movie (on the CD-ROM). Open the movie, play it, move the ship with the arrow keys. Then click anywhere on the Stage. Clicking will move you to frame 2; click again to return to frame 1. When you do, the ship is back in its original location.

Tip

When you go to a frame where the sprite with the behavior does not exist, and you then return, the sprite is back in its original position in the Score, and any properties the sprite had are reinitialized on return.

Jumping to different frames

To further demonstrate the loss of property values, declare a property at the top of the B_Ship script. Call it `pGreeting`, as in Figure 15-6.



```
property pSprite
property pGreeting

on beginSprite me
  pSprite = sprite(me.spriteNum)
end
```

Figure 15-6: Greeting is declared, but it is not given a value anywhere in the script.

Play the movie, and while it is playing, in the Message window, type the following:

```
put (sprite 2).pGreeting
-- <Void>
```

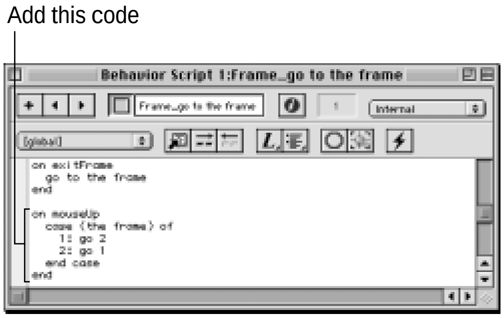
This is the expected result. `pGreeting` has not been given a value anywhere in the script (unlike `pSprite`). Give it a value in the Message window and test that value by using the `put` command:

```
(sprite 2).pGreeting = "Hello"
put (sprite 2).pGreeting
-- "Hello"
```

Now, open the frame script, `Frame_go to the frame` and add the `mouseUp` handler shown in Figure 15-7. Play the movie and click anywhere in the frame. This script is on both frame 1 and 2; the simple case statement in the `mouseUp` handler enables us to toggle between frames.

After clicking and going to frame 2, click again to return to frame 1. Type the following in the Message window:

```
put (sprite 2).pGreeting
-- <Void>
```



```
on exitFrame
  go to the frame
end

on mouseUp
  case (the frame) of
    1: go 2
    2: go 1
  end case
end
```

Figure 15-7: The case statement in the mouseUp handler enables us to toggle between frames 1 and 2.

The value of pGreeting was lost when we went to frame 2. Save the movie as **mars landing 05.dir**.

pGreeting lost its value because the sprite spans only frame 1. If it were extended to frame 2, it would retain its value. Do that now. Open the Score, click sprite 2, and Option+drag (Alt+drag) it to frame 2.

Caution Do *not* copy and paste the sprite in frame 2. If you do that, the sprite will not be continuous. It will be considered a new sprite and the values will be lost.

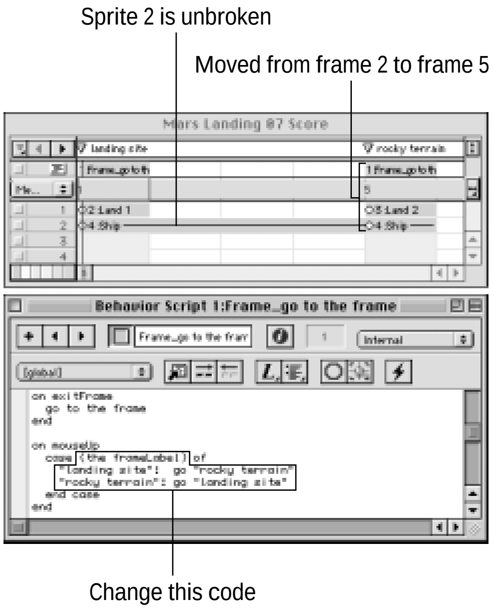
Try giving the pGreeting a value again and click the Stage to alternate between frames. It keeps its value as you navigate from one frame to the other. Save the movie as **mars landing 06.dir**.

Using markers

Sometimes, giving explicit frame numbers, as in Figure 15-7, can lead to problems. What if you moved those frames? You'd have to go back into the script and change the frame numbers. On larger projects, this can be a real hassle, especially if you used those numbers in several places.

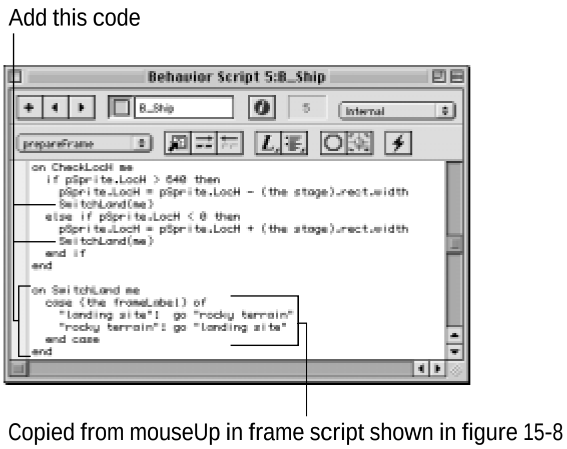
As Figure 15-8 illustrates, the frames no longer need to be next to each other. In fact, sometimes it's nice to have some space in the Score. Note that the ship sprite still needs to span both frames for it to retain its properties. Save the movie as **mars landing 07.dir**.

After we're sure the statements in the mouseUp handler are working, we can take them and use them in the B_Ship script. Create a new handler called SwitchLand, and copy and paste the code from the frame script into the SwitchLand handler. It is called from the CheckLocH handler. Every time the ship is moved to the other side of the screen, we jump to a different marker, as shown in Figure 15-9. Save the movie as **mars landing 08.dir**.



```
on exitFrame
  go to the frame
end

on mouseUp
  case (the frameLabel) of
    "landing site":  go "rocky terrain"
    "rocky terrain": go "landing site"
  end case
end
```

Figure 15-8: The mouseUp handler altered to work with markers (frame labels)



```
on CheckLocH me
  if pSprite.LocH > 640 then
    pSprite.LocH = pSprite.LocH - (the stage).rect.width
    SwitchLand(me)
  else if pSprite.LocH < 0 then
    pSprite.LocH = pSprite.LocH + (the stage).rect.width
    SwitchLand(me)
  end if
end

on SwitchLand me
  case (the frameLabel) of
    "landing site":  go "rocky terrain"
    "rocky terrain": go "landing site"
  end case
end
```

Figure 15-9: The code from the mouseUp handler in the frame script can be used in the B_ship script to make it appear to change landscapes.

More Sprite Manipulations

Moving a sprite a very common manipulation in Lingo, but there are many other manipulations that can be performed on sprites after you are comfortable with Lingo. In Chapter 14, you swapped cast members for a button sprite, but here you swap whole landscapes. You also use the distortion effects of rotation, skew, and scaling that were introduced in Director 7.

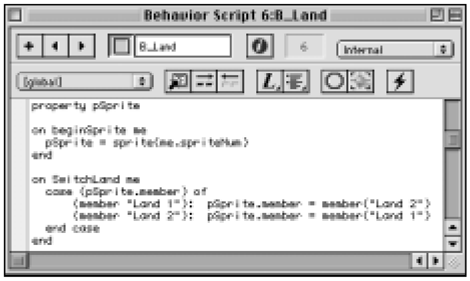
Changing the member

In Chapter 14, you learned how to swap sprites, changing states to make a button look like it was interacting with the cursor. Instead of putting different landscapes in different frames, why not do it all in one frame? This saves you the trouble of having to be scrolling all over the Score.

On the CD-ROM

The movie mars landing 09.dir is in the EXERCISE:CH15 (EXERCISE\CH15) folder on the accompanying CD-ROM.

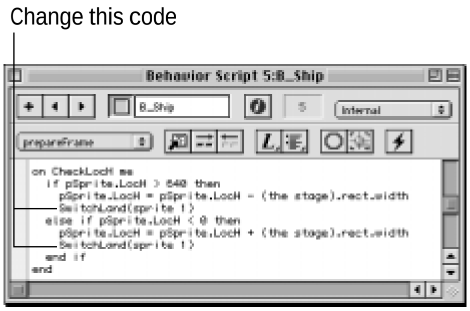
Open the movie mars landing 09.dir on the CD-ROM. Ctrl+click (right-click) sprite 1 (the landscape art) and choose Script to create a new behavior for that sprite. Name the behavior B_Land. Enter the code you see in Figure 15-10. Now the behavior on the sprite with the landscape takes care of switching its sprite's member, as opposed to a behavior on a different sprite (such as the ship) altering this sprite's member. For some, this logic may be more appealing. It seems natural that the behavior on sprite 1 is the only one that actually alters sprite 1.



```
property pSprite

on beginSprite me
  pSprite = sprite(me.spriteNum)
end

on SwitchLand me
  case (pSprite.member) of
      (member "Land 1"):  pSprite.member = member("Land 2")
      (member "Land 2"):  pSprite.member = member("Land 1")
  end case
end
```

Figure 15-10: The SwitchLand handler is now in a behavior on the background sprite.

Next, open the B_Ship script and add a call to the `SwitchLand` handler in sprite 1, as shown in Figure 15-11. The B_Ship behavior no longer has any idea how the land changes: it just tells sprite 1 that it is time and sprite 1 takes care of the details.



```
on CheckLocH me
  if pSprite.LocH > 640 then
    pSprite.LocH = pSprite.LocH - (the stage).rect.width
    SwitchLand(sprite 1)
  else if pSprite.LocH < 0 then
    pSprite.LocH = pSprite.LocH + (the stage).rect.width
    SwitchLand(sprite 1)
  end if
end
```

Figure 15-11: The ship's behavior tells the land when it needs to switch sprites.

Play the movie and try flying off the side of the screen. The background art changes as you do, and your ship appears on the opposite side of the screen. Save this movie as **mars landing 10.dir**.

On the CD-ROM

The mars landing 11.dir movie is in the EXERCISE:CH15 (EXERCISE\CH15) folder on the CD-ROM.

Changing the visibility and locZ

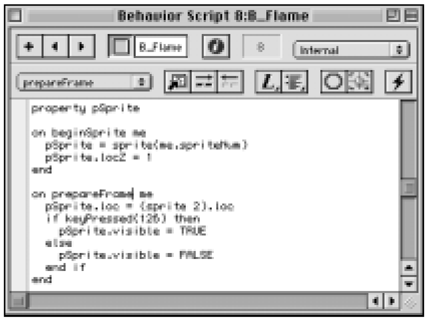
A handy and simple thing to do with Lingo is to change the visibility of a sprite. The ship in the Mars Landing movie needs a flame. Open the mars landing 11.dir on the CD-ROM. A flame has been added in sprite channel 3. We want the flame to appear only when the user is pressing the up arrow key, plus we want the flame to appear behind the ship.

One way to make the flame appear behind the ship is to move it to a lower sprite channel. That is a valid option, but we would miss the opportunity to show you the locZ feature of Director 8.

Tip

The locZ property of a sprite determines the order in which a sprite is drawn on the screen. The default value of the locZ is the sprite's channel number.

A good example of using the locZ property is with the flame sprite. Ctrl+click (right-click) the flame sprite and choose Script to create a new behavior. In it, type what is shown in Figure 15-12.



```
property pSprite

on beginSprite me
  pSprite = sprite(me.spriteNum)
  pSprite.locZ = 1
end

on prepareFrame me
  pSprite.loc = (sprite 2).loc
  if keyPressed(126) then
    pSprite.visible = TRUE
  else
    pSprite.visible = FALSE
  end if
end
```

Figure 15-12: The flame sprite follows the ship in sprite 2.

We have the flame in sprite 3, but we want it to appear below sprite 2. In the `beginSprite` handler for the B_Flame script, we set the locZ to 1. If we set it to 2, it would still appear above the ship. This is because when two locZ's are equal, the higher sprite channel number is drawn above the other. If we set the locZ of the flame to 2, and the ship is in 2, then Director looks to see which is in the lesser channel. The ship is, so it is drawn first, below the flame. That is why we set the flame to have a locZ of 1. It still appears above the background in sprite 1 because the flame is actually in sprite 3.

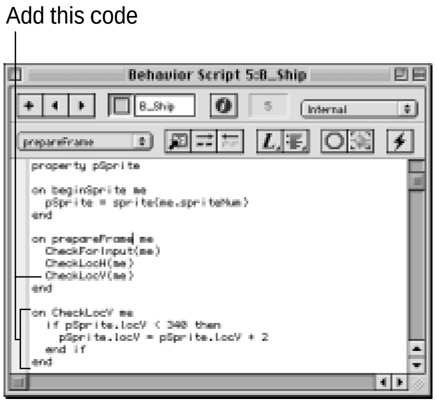
Another good reason to use the locZ in this situation is that the flame is following the ship sprite. With each `exitFrame`, the location of the ship is changed based on user input. Even though the locZ of the flame is set to be lower, the behaviors on the sprite still execute after the ship.

Tip

Behaviors execute in the order of the sprite channels, starting with 1 and continuing through 1000. The behaviors in sprite 3 always execute after those in sprite 2, even when sprite 3 is being drawn below sprite 2.

The way it works in our movie now is that the ship sprite sets its locH and locV, and then the flame sets its loc based on the loc of the ship sprite. If the flame were in a lower sprite, it would set its loc based on the ship sprite before the ship sprite's loc was changed for that `exitFrame`.

One more change, just to enhance the spaceship flying experience. This planet needs gravity! Add a `CheckLocV` handler to the B_Ship script, as shown in Figure 15-13. Now, whenever the ship's locV is less than 340 (higher on the screen), it will add 2 to the ship sprite's locV. Save this movie as **mars landing 12.dir**.



```
property pSprite

on beginSprite me
  pSprite = sprite(me.spriteNum)
end

on prepareFrame me
  CheckForInput(me)
  CheckLocH(me)
  CheckLocV(me)
end

on CheckLocV me
  if pSprite.locV < 340 then
    pSprite.locV = pSprite.locV + 2
  end if
end
```

Figure 15-13: Adding some gravity with CheckLocV

Rotating, skewing, flipping, and scaling sprites

One of the most exciting features in Director 8 is the capability to rotate, skew, and flip sprites. Not just bitmap sprites, either — you apply these transformations to text, vector shape, animated .GIFs, and Flash members. This feature, however, is not available to fields, nor is it available to the ellipses and rectangles on the tool palette.

Setting these properties is very simple. To rotate sprite 1 to 30 degrees, for example, you can use dot notation or traditional Lingo syntax, as follows:

```
sprite(1).rotation = 30
set the rotation of sprite 1 to 30
```

Skewing is also in degrees:

```
sprite(1).skew = 10
set the skew of sprite 1 to 10
```

Flipping is set to TRUE or FALSE (1 or 0). You can flip horizontally with `flipH` or vertically with `flipV`:

```
sprite(1).flipH = TRUE
sprite(1).flipV = 1
```

You can scale a sprite up or down, rotate it, skew it, and still get great performance. The image quality is very good, too (especially when just scaling).

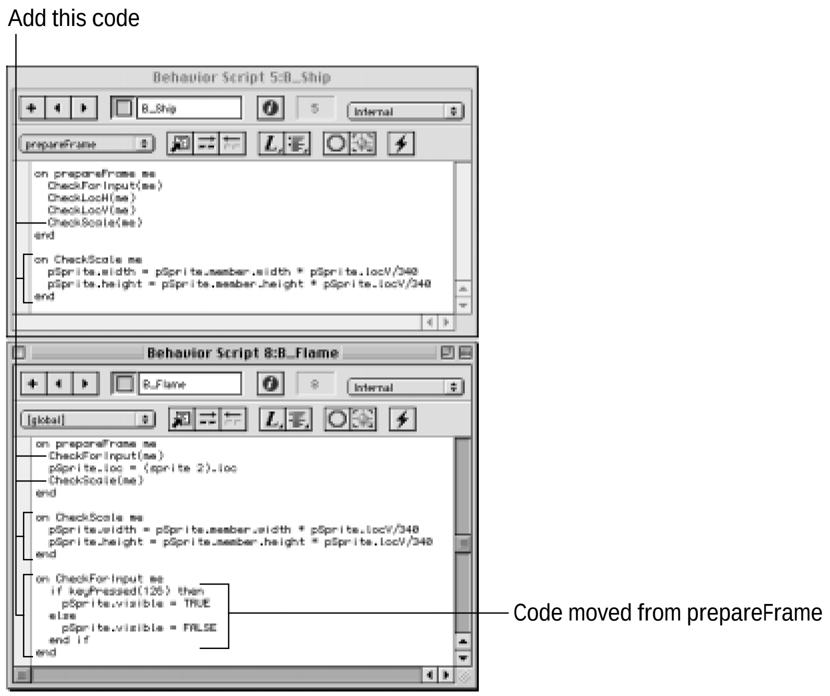
Scaling the ship and flame

Continuing with our Mars landing, you can make the ship smaller as it gets higher, to give the illusion of it flying off into space (or coming down from space). You need to add a scaling routine to the flame as well. The routine is called `CheckScale`. It sets the width and height of the sprite, based on the width of the member, multiplied by a ratio based on the locV of the sprite. The ratio is based on the assumption that the highest the locV (lowest on the screen) the sprite will go is 340. When it is at 340/340, it will be at 100 percent (see Figure 15-14).

While you're in the B_Flame script, take this opportunity to create a `CheckForInput` handler, and move the `keyPressed` detection into it. This keeps the main event loop of this handler, the `exitFrame`, less cluttered, and it is more consistent with how we've coded the B_Ship script. Consistency can be a great help when you come back and look at your code in six months, trying to find out what you were thinking when you wrote it. Now would be a good time to save your movie as **mars landing 13.dir**.

Tweaking speed

You might have noticed that the ship seems to zip into space faster as it gets higher. This may be partly due to the smaller screen area that Director has to redraw as the ship graphic is scaled down. But an even greater effect is the illusion of a much smaller ship moving 10 pixels versus a larger ship moving 10 pixels. The distance moved relative to its size is giving the perception of speed (or slowness). This effect can be improved by multiplying the distance to move the ship by the same ratio we are scaling the ship.



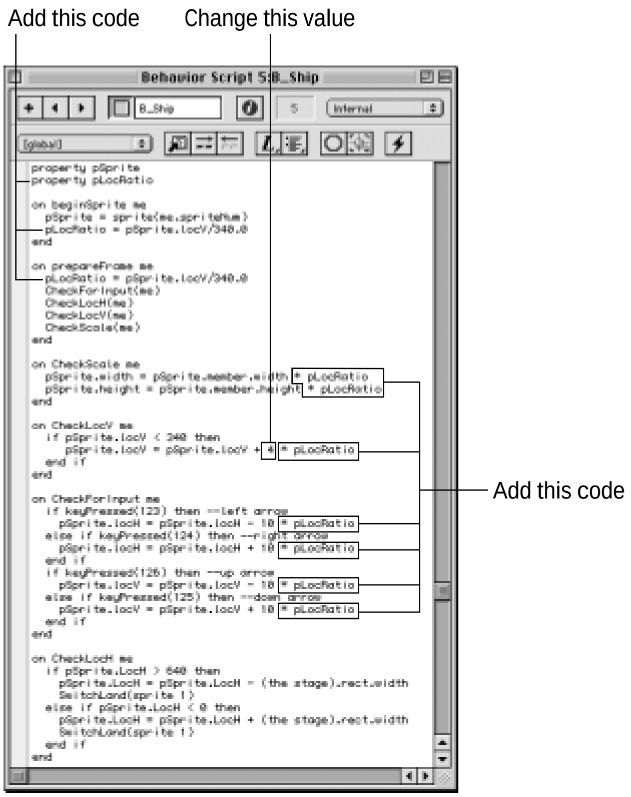
```
on prepareFrame me
  CheckForInput(me)
  CheckLocH(me)
  CheckLocV(me)
  CheckScale(me)
end

on CheckScale me
  pSprite.width = pSprite.member.width * pSprite.locV/340
  pSprite.height = pSprite.member.height * pSprite.locV/340
end
```

```
on prepareFrame me
  CheckForInput(me)
  pSprite.loc = (sprite 2).loc
  CheckScale(me)
end

on CheckScale me
  pSprite.width = pSprite.member.width * pSprite.locV/340
  pSprite.height = pSprite.member.height * pSprite.locV/340
end

on CheckForInput me
  if keyPressed(126) then
    pSprite.visible = TRUE
  else
    pSprite.visible = FALSE
  end if
end
```

Figure 15-14: CheckScale is added to both scripts, and CheckForInput is added to B_Flame.

Create a new property called `pLocRatio`, named after the fact that it is a ratio based on the location of the ship. This value will be the locV of the sprite divided by 340.0. Make sure there is a "point zero" after the 340 so that Director uses a float. As mentioned in Chapter 11, if you use integers in your multiplication and division, the result is integers, which would introduce rounding errors (try it and see what happens).

Your script should look like that in Figure 15-15. Don't worry about not seeing the `CheckLocH` handler; it is still in the script. There were no changes to it, so we moved it below the `CheckForInput` handler. `pLocRatio` is initialized in the `beginSprite` handler and then needs to be updated each time through the `exitFrame` loop. Play the movie and check out the difference this makes. Fly up high and note how much more time it takes to move across the black Martian sky. Save this movie as **mars landing 14.dir**.



```
property pSprite
property pLocRatio

on beginSprite me
  pSprite = sprite(me.spriteNum)
  pLocRatio = pSprite.locV/340.0
end

on prepareFrame me
  pLocRatio = pSprite.locV/340.0
  CheckForInput(me)
  CheckLocH(me)
  CheckLocV(me)
  CheckScale(me)
end

on CheckScale me
  pSprite.width = pSprite.member.width * pLocRatio
  pSprite.height = pSprite.member.height * pLocRatio
end

on CheckLocV me
  if pSprite.locV < 340 then
     pSprite.locV = pSprite.locV + 4 * pLocRatio
  end if
end

on CheckForInput me
  if keyPressed(123) then --left arrow
    pSprite.locH = pSprite.locH - 10 * pLocRatio
  else if keyPressed(124) then --right arrow
    pSprite.locH = pSprite.locH + 10 * pLocRatio
  end if
  if keyPressed(126) then --up arrow
    pSprite.locV = pSprite.locV - 10 * pLocRatio
  else if keyPressed(125) then --down arrow
    pSprite.locV = pSprite.locV + 10 * pLocRatio
  end if
end

on CheckLocH me
  if pSprite.LocH > 640 then
    pSprite.LocH = pSprite.LocH - (the stage).rect.width
    SwitchLand(sprite 1)
  else if pSprite.LocH < 0 then
    pSprite.LocH = pSprite.LocH + (the stage).rect.width
    SwitchLand(sprite 1)
  end if
end
```

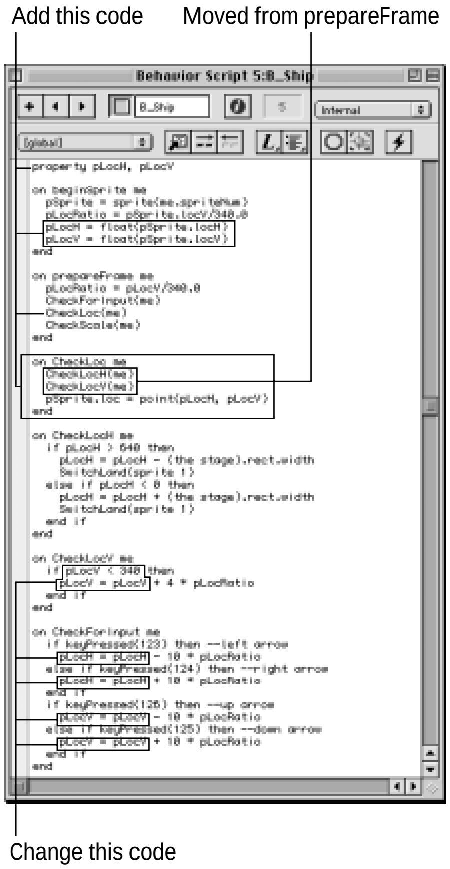
Figure 15-15: pLocRatio is added to all the handlers shown.

Tweaking movement

Although there is much better movement, it still doesn't seem like it flies up and down as smoothly as it could. It seems like it is moving in fits — subtle but noticeable. It also gets stuck near the top of the screen. What is happening?

The ship sprite's behavior has been using its own screen coordinates to modify its movements. This is a problem, because screen coordinates are integers. The smallest amount a graphic can move on the screen is 1 pixel. The float values from our calculations are being rounded. Suppose that the result of a calculation to determine the locV is 10.5; that will round to 11 on the screen. The next time through the loop, 11 is used in the calculation, because that is the value of the sprite's locV. That five-tenths of a unit is lost. To be more accurate, the calculation should have used the result from the previous calculation. What is needed is a property for both the locH and locV that is a float value. This provides for the appearance of subpixel movement. If we slow how often it moves in a consistent manner, it appears to move in smaller increments.

Add two more properties: `pLocH` and `pLoc`v. We declared them on the same line (separated by a comma) to save space. Then replace wherever a sprite's loc is set with the corresponding property. After those values have been set, they need to be applied to the sprite's loc. Create a new handler called `CheckLoc`. In it, put `CheckLocH`, `CheckLocV`, and the statement to set the sprite's loc (see Figure 15-16). Not shown are the other properties and handlers that do not have changes — they're still there, above and below the code shown. Now fly the Mars Lander and check out how smoothly it goes up (until it is just a speck) and moves through the sky. Save this movie as **mars landing 15.dir**.



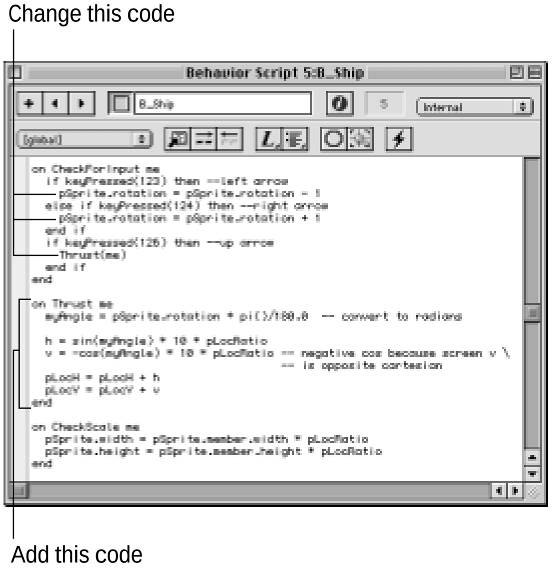
```
property pLocH, pLocV

on beginSprite me
  pSprite = sprite(me.spriteNum)
  pLocRatio = pSprite.locV/340.0
  pLocH = float(pSprite.locH)
  pLocV = float(pSprite.locV)
end

on prepareFrame me
  pLocRatio = pLocV/340.0
  CheckForInput(me)
  CheckLoc(me)
  CheckScale(me)
end

on CheckLoc me
  CheckLocH(me)
  CheckLocV(me)
  pSprite.loc = point(pLocH, pLocV)
end

on CheckLocH me
  if pLocH > 640 then
    pLocH = pLocH - (the stage).rect.width
    SwitchLand(sprite 1)
  else if pLocH < 0 then
    pLocH = pLocH + (the stage).rect.width
    SwitchLand(sprite 1)
  end if
end

on CheckLocV me
  if pLocV < 340 then
    pLocV = pLocV + 4 * pLocRatio
  end if
end

on CheckForInput me
  if keyPressed(123) then --left arrow
    pLocH = pLocH - 10 * pLocRatio
  else if keyPressed(124) then --right arrow
    pLocH = pLocH + 10 * pLocRatio
  end if
  if keyPressed(126) then --up arrow
    pLocV = pLocV - 10 * pLocRatio
  else if keyPressed(125) then --down arrow
    pLocV = pLocV + 10 * pLocRatio
  end if
end
```

Figure 15-16: To increase the precision of the sprite's movement, properties are kept for the locH and locV of the sprite.

Rotating the ship and diagonal movement

You're dying to turn that ship, aren't you? Rotating a sprite is easy; it's just a matter of setting the rotation to whatever increment in degrees you desire. There is a catch, of course: The ship needs to move in the direction that it is facing, and at the same angle. This involves dusting off your trigonometry books. Don't worry, it's not much work, and we tell you the answers. Another benefit is a more realistic diagonal movement. Currently, if you hold down both a horizontal direction key and a vertical direction speed, you move at a faster rate (10 horizontally and 10 vertically).

Fortunately, this involves the creation of only one handler and the altering of another. Because the ship is now rotated with the arrow keys instead of moving to the side, `CheckForInput` has to change. The new handler is called `Thrust`. We do away with checking for the down arrow key; gravity has been handling the downward movement for some time. Pressing the up arrow key calls the `Thrust` handler. Change `CheckForInput` and create the `Thrust` handler, as shown in Figure 15-17.



```
on CheckForInput me
  if keyPressed(123) then --left arrow
    pSprite.rotation = pSprite.rotation - 1
  else if keyPressed(124) then --right arrow
    pSprite.rotation = pSprite.rotation + 1
  end if
  if keyPressed(126) then --up arrow
    Thrust(me)
  end if
end

on Thrust me
  myAngle = pSprite.rotation * pi()/180.0  -- convert to radians

  h = sin(myAngle) * 10 * pLocRatio
  v = -cos(myAngle) * 10 * pLocRatio -- negative cos because screen v \
                                     -- is opposite cartesian
  pLocH = pLocH + h
  pLocV = pLocV + v
end

on CheckScale me
  pSprite.width = pSprite.member.width * pLocRatio
  pSprite.height = pSprite.member.height * pLocRatio
end
```

Figure 15-17: Rotating the sprite is easy, but moving it at the right angle takes some explaining.

The rotation of the sprite is straightforward. When the left or right arrow key is pressed, the sprite is rotated left or right one degree. The only thing that might surprise you is that turning to the right is incrementing the angle—it goes clockwise. In trigonometry, a positive increase in the angle is counterclockwise.

The angle of the sprite is converted to radians and put into the variable `myAngle`. Radians are another way of measuring angles. The sine and cosine functions use them. There are pi (3.14159. . .) radians in 180 degrees:

```
myAngle = pSprite.rotation * pi()/180.0
```

After we have the angle, the horizontal and vertical distances need to be determined. For that, we use the `sin()` and `cos()` functions, the Lingo version of sine and cosine. Because the zero position of an angle is usually horizontal, and our ship starts out vertical, we swapped the `sin()` and `cos()` functions. The number 10 is just the arbitrary number we've been using for acceleration, and `pLocRatio` adjusts the acceleration based on the locV of the ship:

```
h = sin(myAngle) * 10 * pLocRatio
v = -cos(myAngle) * 10 * pLocRatio
```

Last, we just increment `pLocH` and `pLocV`:

```
pLocH = pLocH + h
pLocV = pLocV + v
```

That wasn't too bad, was it? You now have a routine for diagonal movement that you can use over and over. Fly this baby now! Oh, and save the movie as **mars landing 16.dir**.

Scrolling backgrounds

An effect that is very popular in games is the scrolling background. The ship flies over the landscape as it moves beneath it in one continuous loop. The Mars Landing movie has almost everything it needs to implement a scrolling landscape.

What happens is that the sprites holding the pictures of the landscape move instead of the ship. As the ship accelerates, the land moves faster. Because we have two background images, each the width of the Stage, we need to be checking their location constantly. If they get too far to one side or the other, they need to be moved so that no gaps appear in the landscape (see Figure 15-18).

Open the movie mars landing 17.dir. It is the same as mars landing 16.dir, except that the handlers in the B_Ship script are reordered in the sequence in which they are called. If you want to continue with the movie that you have open, just be sure you make the changes to the correct handlers.

The first thing you should do is to create four new properties we use in the movie, `pSpeedH`, `pSpeedV`, `pMaxSpeed`, and `pAccel`. Declare them at the top of the Script window as you've done with other properties, as shown in Figure 15-19.You can use the `pSpeedH` and `pSpeedV` properties to increment the location of the ship; you no longer have instant acceleration. The speed increases in increments until the maximum speed is reached. `pMaxSpeed` holds the value for the maximum speed. `pAccel` holds the amount you can increase the speed or accelerate each time through an `exitFrame`.

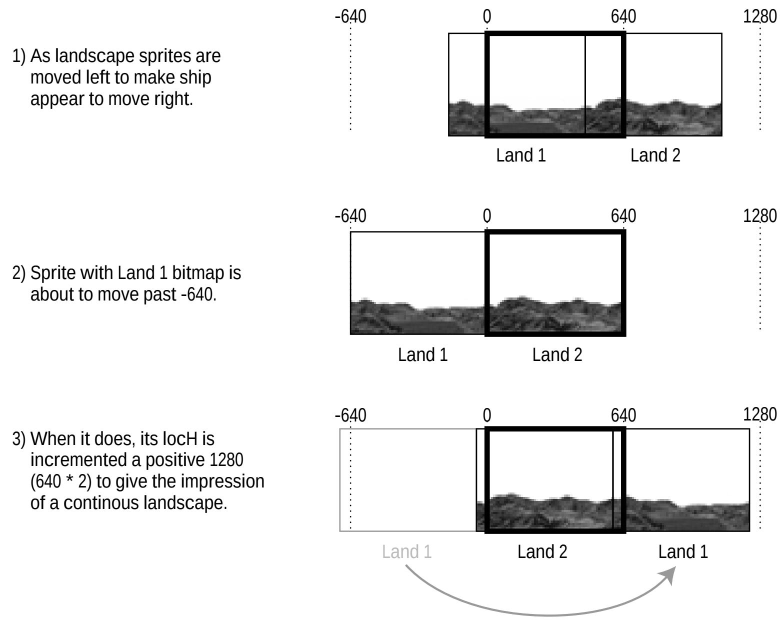
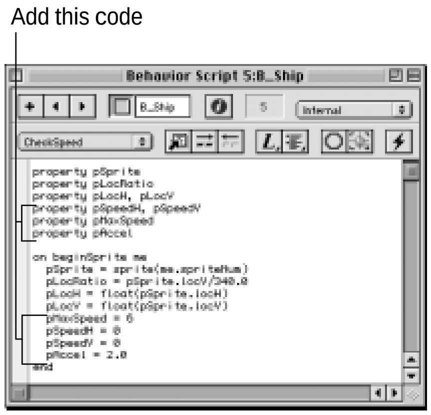

1) As landscape sprites are moved left to make ship appear to move right.
2) Sprite with Land 1 bitmap is about to move past -640.
3) When it does, its locH is incremented a positive 1280 (640 * 2) to give the impression of a continous landscape.
-640
0
640
1280
Land 1
Land 2

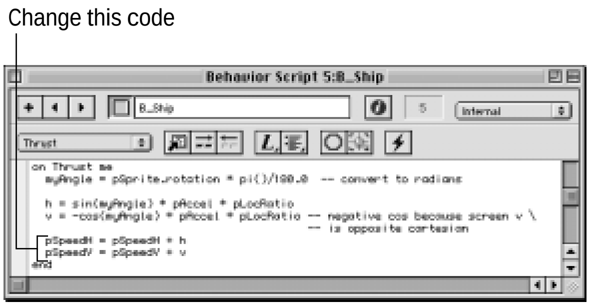
Figure 15-18: The mechanics of making the land appear continuous. The heavy black rectangle is what you see on the Stage.



```
property pSprite
property pLocRatio
property pLocH, pLocV
property pSpeedH, pSpeedV
property pMaxSpeed
property pAccel

on beginSprite me
  pSprite = sprite(me.spriteNum)
  pLocRatio = pSprite.locV/340.0
  pLocH = float(pSprite.locH)
  pLocV = float(pSprite.locV)
  pMaxSpeed = 6
  pSpeedH = 0
  pSpeedV = 0
  pAccel = 2.0
end
```

Figure 15-19: Four new properties are declared and initialized.

The two speed variables are altered in the Thrust handler. Thrust altering speed seems logical. pAccel is also used here (see Figure 15-20).



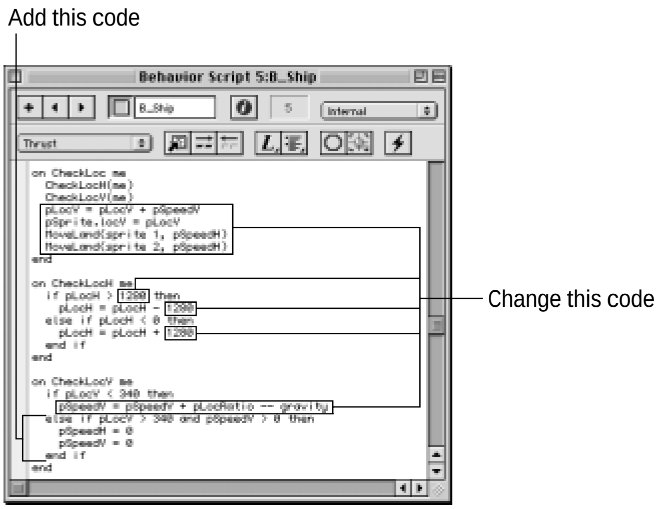
```
on Thrust me
  myAngle = pSprite.rotation * pi()/180.0  -- convert to radians

  h = sin(myAngle) * pAccel * pLocRatio
  v = -cos(myAngle) * pAccel * pLocRatio -- negative cos because screen v \
                                          -- is opposite cartesian
  pSpeedH = pSpeedH + h
  pSpeedV = pSpeedV + v
end
```

Figure 15-20: Thrust now affects speed, and the acceleration is no longer a literal, but the property variable pAccel.

In place of the literal 10, you can put the variable pAccel. Having a variable (property or otherwise) is better than having some arbitrary number appear more than once in the program. It is good practice to use variables instead of actual values in the code, especially when a value is used more than once. The number 10 is a good example of this. Seeing 10 doesn't necessarily say "acceleration," but the variable pAccel does. The 180.0 in the conversion of degrees to radians is all right, because it is only used once and it is a common value in a common equation.

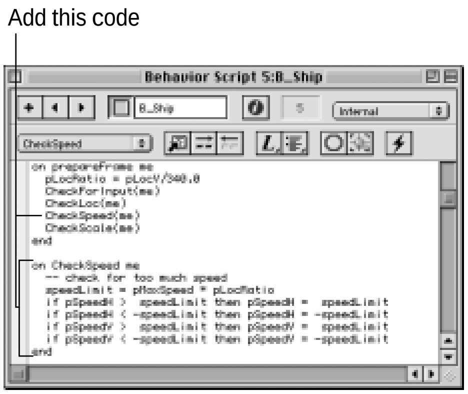
The handlers that deal with the ship's location need to be changed as well, as shown in Figure 15-21. CheckLoc increments pLocV with pSpeed before setting the sprite's locV. Then we need to move the land sprites (1 and 2) in relation to the speed. We call them MoveLand handlers (which we create shortly). CheckLocH needs to deal with the fact the ship is in a world 1280 pixels wide (640*2). If we had three 640-pixel-wide landscapes, then we would change this number to 1920 (640*3), and so on. CheckLocV needs to increment pSpeedV instead of pLocV. This makes more sense too, because gravity is affecting the speed with which the ship is falling. The speed properties are zeroed out if the ship is at 340 pixels or lower and pSpeed is greater than 0. This keeps the ship from going below the bottom of the screen, and it enables it to lift off.

Because the speed is being continually incremented, either by user input or by gravity, it needs to be checked against the maximum speed allowed, which is stored in pMaxSpeed. Create the CheckSpeed handler as shown in Figure 15-22. CheckSpeed is called from the exitFrame handler. It should be placed after CheckLoc in case CheckLoc exceeded the value in pMaxSpeed. Again, pLocRatio is used to make the speed relative to the position of the ship.



```
on CheckLoc me
  CheckLocH(me)
  CheckLocV(me)
  pLocV = pLocV + pSpeedV
  pSprite.locV = pLocV
  MoveLand(sprite 1, pSpeedH)
  MoveLand(sprite 2, pSpeedH)
end

on CheckLocH me
  if pLocH > 1280 then
    pLocH = pLocH - 1280
  else if pLocH < 0 then
    pLocH = pLocH + 1280
  end if
end

on CheckLocV me
  if pLocV < 340 then
    pSpeedV = pSpeedV + pLocRatio -- gravity
  else if pLocV > 340 and pSpeedV > 0 then
    pSpeedH = 0
    pSpeedV = 0
  end if
end
```

Figure 15-21: The handlers that deal with the ship's location need to be modified.



```
on prepareFrame me
  pLocRatio = pLocV/340.0
  CheckForInput(me)
  CheckLoc(me)
  CheckSpeed(me)
  CheckScale(me)
end

on CheckSpeed me
  -- check for too much speed
  speedLimit = pMaxSpeed * pLocRatio
  if pSpeedH >  speedLimit then pSpeedH =  speedLimit
  if pSpeedH < -speedLimit then pSpeedH = -speedLimit
  if pSpeedV >  speedLimit then pSpeedV =  speedLimit
  if pSpeedV < -speedLimit then pSpeedV = -speedLimit
end
```

Figure 15-22: The new speed properties pSpeedH and pSpeedV need to be checked against pMaxSpeed.

Currently, the movie has only one of the landscapes in the Score. Move the ship and flame sprites so that member Land 2 can be put in sprite 2. Then apply the B_Land script to sprite 2. After that, position sprite 2 at X:640 and Y:480, as shown in Figure 15-23.

Having changed the sprite the ship is in, that change needs to be reflected in the B_Flame script. This script refers to sprite 2, assuming the ship is in it. It needs to be changed to sprite 3. The locZ also needs to change to 2, so that the flame does not appear behind the new landscape in sprite 2 (see Figure 15-24). You can now save this movie as **mars landing 18.dir**.

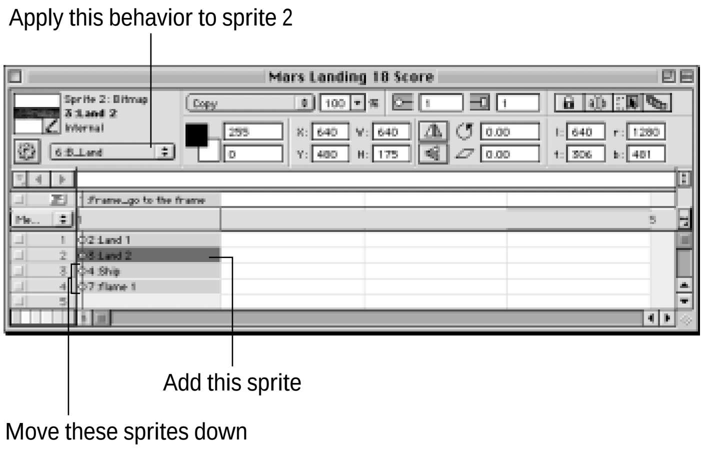
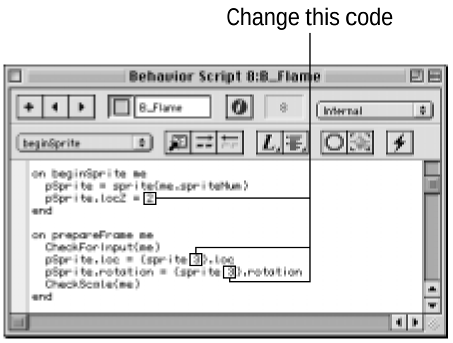


Figure 15-23: Member Land 2 in sprite 2 with the B_Land script applied



```
on beginSprite me
  pSprite = sprite(me.spriteNum)
  pSprite.locZ = 2
end

on prepareFrame me
  CheckForInput(me)
  pSprite.loc = (sprite 3).loc
  pSprite.rotation = (sprite 3).rotation
  CheckScale(me)
end
```

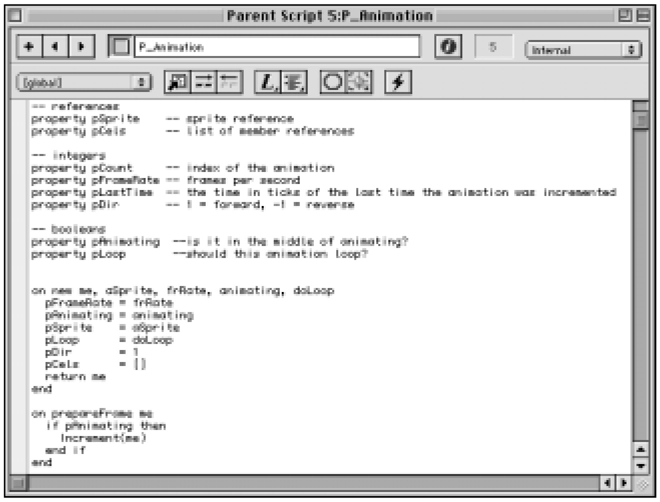
Figure 15-24: Changing the locZ for the B_flame sprite behavior

Using Parent Scripts with Sprites

You might not be too surprised to learn that the Mars Landing movie can use a parent script. Director has many ways to create an animation in a single frame. You can use a film loop, a QuickTime movie, an animated .GIF, or even a Flash movie. A film loop can be convenient, because all the artwork is created in Director, but it is limited by a fixed `regPoint`, and the tempo runs at the tempo of the movie. QuickTime gives you some flexibility when it comes to tempo, but they are often created outside of Director; the performance goes down when it is not direct to Stage. Animated .GIFs have a moveable `regPoint`, but the format itself is limited to 8-bit. Last, Flash movies can be manipulated quite a bit, but they, too, are developed outside Director. This leads us to a custom solution, a parent script that plays a series of cast members, as if they were animation cels, at a frame rate you specify.

Creating the parent script

Open the movie mars landing 19.dir. It has the same code as mars landing 18.dir, but added to it are 19 ship bitmaps to create an animation of the legs coming down (or going up). The ship in the Score is now called Ship 19, whereas before it was just Ship. Open a new Script window by choosing Window ➪ Script. If it opens an existing script, click the New Script button. Click the Cast Member Properties button to change this script to a parent script, and then name it **animation**. Type in the code that you see in Figure 15-25.



```
-- references
property pSprite    -- sprite reference
property pCels      -- list of member references

-- integers
property pCount     -- index of the animation
property pFrameRate -- frames per second
property pLastTime  -- the time in ticks of the last time the animation was incremented
property pDir       -- 1 = forward, -1 = reverse

-- booleans
property pAnimating  --is it in the middle of animating?
property pLoop       --should this animation loop?


on new me, aSprite, frRate, animating, doLoop
  pFrameRate = frRate
  pAnimating = animating
  pSprite    = aSprite
  pLoop      = doLoop
  pDir       = 1
  pCels      = []
  return me
end

on prepareFrame me
  if pAnimating then
    increment(me)
  end if
end
```

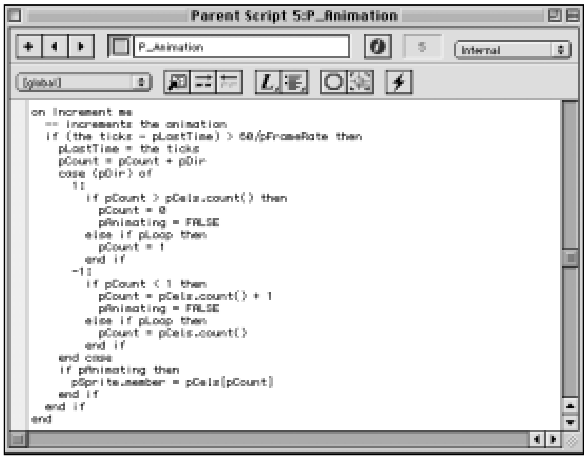
Figure 15-25: The beginning code of the animation script

The new function takes five parameters:

- ✦ `me`: the reference to the object.
- ✦ `aSprite`: a reference to the sprite this object will use.
- ✦ `frRate`: an integer of the rate the animation should play.
- ✦ `animating`: a Boolean value whether the animation is in the process of animating.
- ✦ `doLoop`: another Boolean value. This one indicates whether this is a looping animation.

The nice thing about all these values is that they are easily changeable on the fly through Lingo. Inside the new handler, the properties that correspond to the parameters are given their values. `pDir` and `pCels` are also initialized. `pDir` is either 1 or –1, indicating forward or backward, respectively. `pCels` is a list that will be given a value later. That list will hold a series of member references that will be used like cels in an animation.

The `prepareFrame` handler is quite simple. It tests whether the object is animating by testing the value of `pAnimating` in the `if` statement. If it is, then it increments the animation by calling the `Increment` handler. Below the `prepareFrame` handler, add the `Increment` handler shown in Figure 15-26.

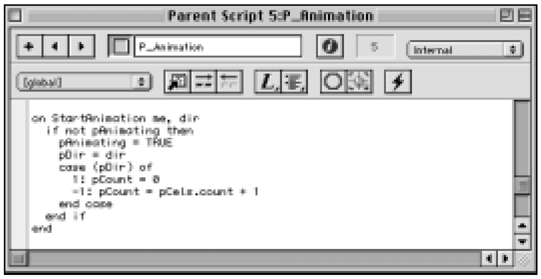


```
on Increment me
  -- increments the animation
  if (the ticks - pLastTime) > 60/pFrameRate then
    pLastTime = the ticks
    pCount = pCount + pDir
    case (pDir) of
      1:
        if pCount > pCels.count() then
          pCount = 0
          pAnimating = FALSE
        else if pLoop then
          pCount = 1
        end if
      -1:
        if pCount < 1 then
          pCount = pCels.count() + 1
          pAnimating = FALSE
        else if pLoop then
          pCount = pCels.count()
        end if
    end case
    if pAnimating then
      pSprite.member = pCels[pCount]
    end if
  end if
end
```

Figure 15-26: The Increment handler increments the animation in one direction or the other.

The `Increment` handler begins by checking the current time (in ticks) against the time in `pLastTime`. If the difference is greater than 60/`pFrameRate`, the animation increments one cel. 60 is the number of ticks in one second. `pFrameRate` was given the `frRate` value in the new handler. If `pFrameRate` is 15, then every 4 ticks the animation is updated. Inside the `if` statement, `pCount` is incremented; it keeps track of the cel it is on. `pLastTime` is also set to the current time in ticks. Based on the direction in `pDir`, the animation increments or decrements. After it has gone through all of the members in the `pCels` list, it stops, unless it is set to loop.

The last handler we need is one to get the animation started. Figure 15-27 shows the `StartAnimation` handler. It takes a direction as a parameter. This handler does nothing if the object is already animating. If it is not, then `pAnimating` is set to TRUE, `pDir` is set, and `pCount` is set based on the direction in `pDir`.



```
on StartAnimation me, dir
  if not pAnimating then
    pAnimating = TRUE
    pDir = dir
    case (pDir) of
      1: pCount = 0
      -1: pCount = pCels.count + 1
    end case
  end if
end
```

Figure 15-27: StartAnimation is the way to get this animation going (if it is not set to animate initially).

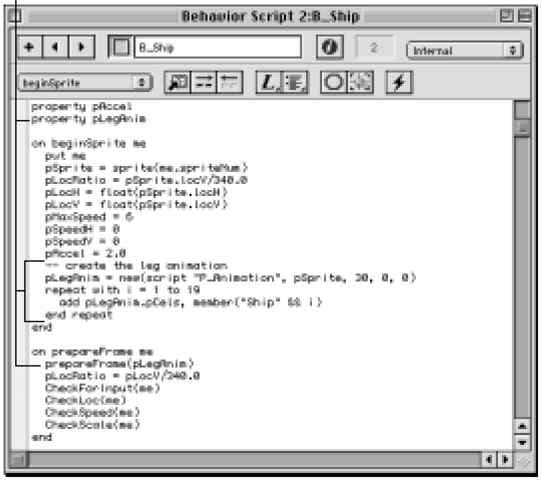
Instantiating the parent script

The B_Ship script creates an instance (child object) of the animation script. Open the script B_Ship and declare a new property called `pLegAnim` right after `pAccel`. In the new handler, it is given a value. That value is a child object of the animation script. Passed along as arguments to the new function is the sprite reference for the ship (`pSprite`), the frame rate (30 fps), 0 for not animating, and 0 for not looping. After `pLegAnim` is initialized, a repeat loop adds 19 member references (Ship 1, Ship 2, and so on) to the `pCels` list within the child object in `pLegAnim`.

Also shown is the `prepareFrame` handler. `prepareFrame (pLegAnim)` was added so that this child object gets a `prepareFrame` message every time the behavior that contains it gets one (see Figure 15-28).

A new handler, `ToggleLegs`, needs to be added to the B_Ship script. This handler sets the direction of the animation based on the member that is currently in the sprite. It then starts the animation by calling the object's `StartAnimation` handler.

Add this code

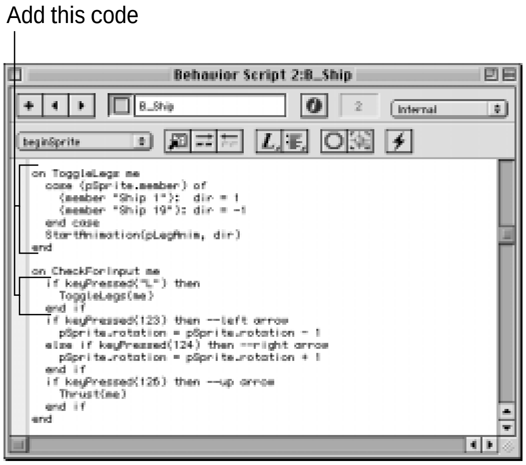


```
property pAccel
property pLegAnim

on beginSprite me
  put me
  pSprite = sprite(me.spriteNum)
  pLocRatio = pSprite.locV/340.0
  pLocH = float(pSprite.locH)
  pLocV = float(pSprite.locV)
  pMaxSpeed = 6
  pSpeedH = 0
  pSpeedV = 0
  pAccel = 2.0
  -- create the leg animation
  pLegAnim = new(script "P_Animation", pSprite, 30, 0, 0)
  repeat with i = 1 to 19
    add pLegAnim.pCels, member("Ship" && i)
  end repeat
end

on prepareFrame me
  prepareFrame(pLegAnim)
  pLocRatio = pLocV/340.0
  CheckForInput(me)
  CheckLoc(me)
  CheckSpeed(me)
  CheckScale(me)
end
```

Figure 15-28: Creating a child object of the animation parent script inside the B_Ship behavior.

`CheckForInput` needs an addition as well. It needs to check whether the user has pressed the L key. If so, `ToggleLegs` is called (see Figure 15-29).

Congratulations! You've come a long way since the beginning of the chapter when you moved a sprite 10 pixels up and 10 pixels down. You can save this movie as **mars landing 20.dir**.



```
on ToggleLegs me
  case (pSprite.member) of
    (member "Ship 1"):  dir = 1
    (member "Ship 19"): dir = -1
  end case
  StartAnimation(pLegAnim, dir)
end

on CheckForInput me
  if keyPressed("L") then
    ToggleLegs(me)
  end if
  if keyPressed(123) then --left arrow
    pSprite.rotation = pSprite.rotation - 1
  else if keyPressed(124) then --right arrow
    pSprite.rotation = pSprite.rotation + 1
  end if
  if keyPressed(126) then --up arrow
    Thrust(me)
  end if
end
```

Figure 15-29: The ToggleLegs handler, called from CheckInput, sets the direction of the animation.

Sprite Rotation Revisited

Sprite rotation is such a cool feature that it needs to be revisited now that the Mars Landing is complete.

On the CD-ROM The movie centipede.dir is in the EXERCISE:CH15 (EXERCISE\CH15) folder on the CD-ROM.

Open the movie centipede.dir on the CD-ROM. The next behavior you write is one to have a series of sprites follow the mouse, like a centipede. Figure 15-30 shows what the finished movie looks like.

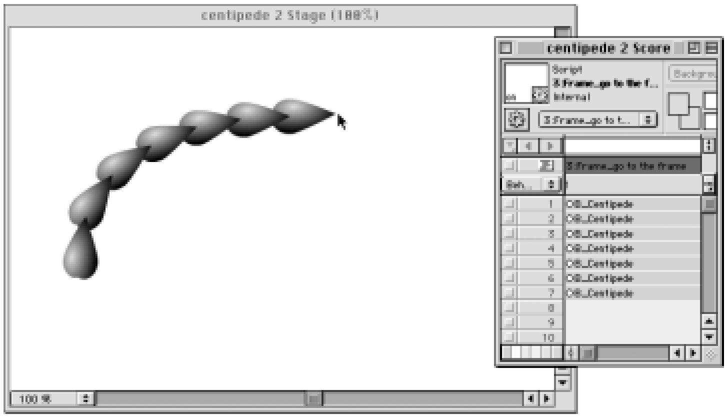
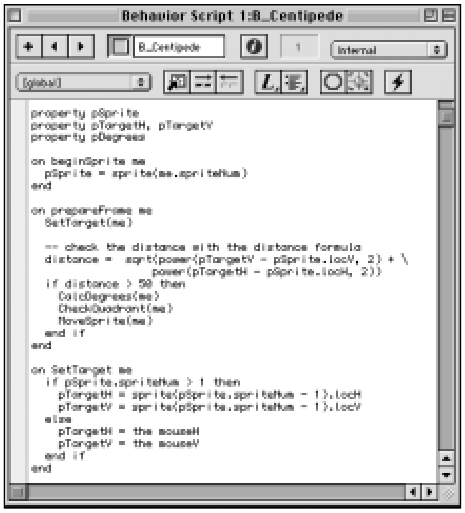


Figure 15-30: The first sprite follows the mouse, and each subsequent sprite follows the one before it.

Ctrl+click (right-click) sprite 1 and choose Script to create a new behavior. In it, recreate what you see in Figure 15-31. For the sprite to follow something, it must have a target. What it really needs is a pair of coordinates for that target. In the `prepareFrame`, `SetTarget` is called. `SetTarget` will choose the mouse as the target if it is sprite 1. Otherwise, it chooses the sprite in the channel right below it and uses that sprite's coordinates as the target.



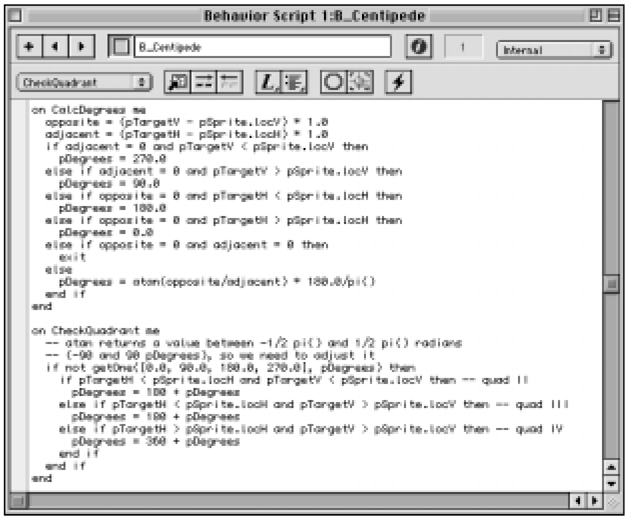
```
property pSprite
property pTargetH, pTargetV
property pDegrees

on beginSprite me
  pSprite = sprite(me.spriteNum)
end

on prepareFrame me
  SetTarget(me)

  -- check the distance with the distance formula
  distance =  sqrt(power(pTargetV - pSprite.locV, 2) + \
                power(pTargetH - pSprite.locH, 2))
  if distance > 50 then
    CalcDegrees(me)
    CheckQuadrant(me)
    MoveSprite(me)
  end if
end

on SetTarget me
  if pSprite.spriteNum > 1 then
    pTargetH = sprite(pSprite.spriteNum - 1).locH
    pTargetV = sprite(pSprite.spriteNum - 1).locV
  else
    pTargetH = the mouseH
    pTargetV = the mouseV
  end if
end
```

Figure 15-31: First, the sprite gets the location of its target.

Next it needs to check its distance from the target. If it is too close, it will do nothing to prevent the sprites from ending up on top of each other. The angle of the sprite to the target needs to be calculated. `CalcDegrees` does just that. It checks for the special cases first; if it is not a special case, it gets the degrees by using `atan()`, the arctangent function. `atan` returns a value in radians, so it is converted to degrees by multiplying by 180 and then dividing by pi.

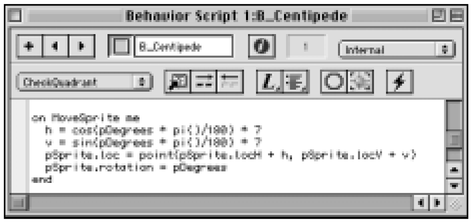
Because `atan` returns a value between –pi radians and pi radians (–90 degrees and 90 degrees), the value needs to be adjusted based on what quadrant the target is in relative to the sprite, as shown in Figure 15-32.

The last task is to move the sprite in the direction of the target; `MoveSprite` handles this. The amount the sprite moves is multiplied by 7. The value 7 is just an arbitrary number, so you can play around with different values to have it move at different speeds. The loc of the sprite is set and so is the rotation, as shown in Figure 15-33. Play your movie and watch the sprites snake around on the Stage like a centipede.



```
on CalcDegrees me
  opposite = (pTargetV - pSprite.locV) * 1.0
  adjacent = (pTargetH - pSprite.locH) * 1.0
  if adjacent = 0 and pTargetV < pSprite.locV then
    pDegrees = 270.0
  else if adjacent = 0 and pTargetV > pSprite.locV then
    pDegrees = 90.0
  else if opposite = 0 and pTargetH < pSprite.locH then
    pDegrees = 180.0
  else if opposite = 0 and pTargetH > pSprite.locH then
    pDegrees = 0.0
  else if opposite = 0 and adjacent = 0 then
    exit
  else
    pDegrees = atan(opposite/adjacent) * 180.0/pi()
  end if
end

on CheckQuadrant me
  -- atan returns a value between -1/2 pi() and 1/2 pi() radians
  -- (-90 and 90 pDegrees), so we need to adjust it
  if not getOne([0.0, 90.0, 180.0, 270.0], pDegrees) then
    if pTargetH < pSprite.locH and pTargetV < pSprite.locV then -- quad II
      pDegrees = 180 + pDegrees
    else if pTargetH < pSprite.locH and pTargetV > pSprite.locV then -- quad III
      pDegrees = 180 + pDegrees
    else if pTargetH > pSprite.locH and pTargetV > pSprite.locV then -- quad IV
      pDegrees = 360 + pDegrees
    end if
  end if
end
```

Figure 15-32: CalcDegrees and CheckQuadrant calculate the angle between the sprite and its target.



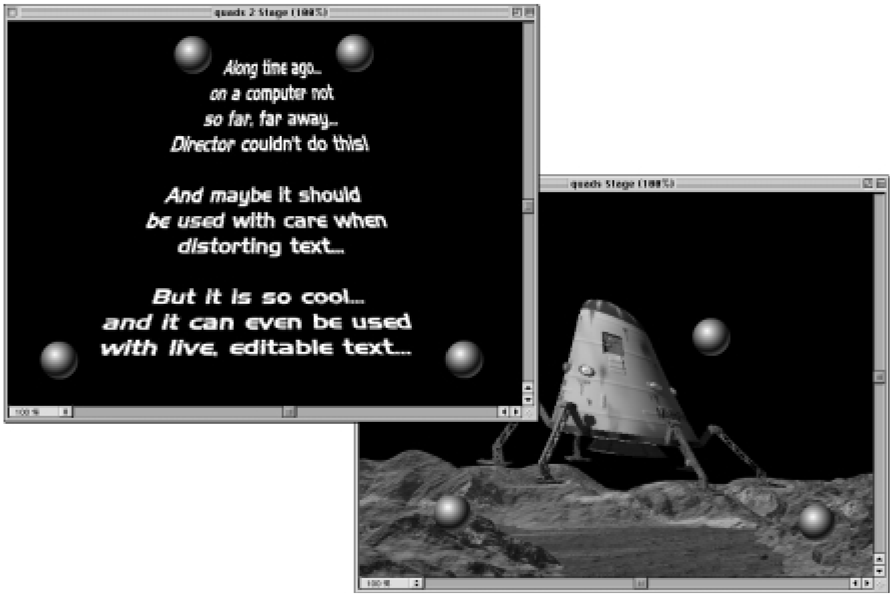
```
on MoveSprite me
  h = cos(pDegrees * pi()/180) * 7
  v = sin(pDegrees * pi()/180) * 7
  pSprite.loc = point(pSprite.locH + h, pSprite.locV + v)
  pSprite.rotation = pDegrees
end
```

Figure 15-33: MoveSprite sets the loc and rotation of the sprite.

Using Quads

Whereas you can use the Score to take advantage of sprite rotation, skewing, and flipping, you can take advantage of *quads* only through Lingo. The quad of a sprite is a list of points defining the shape of the sprite. With quads, you can distort the perspective of a sprite referencing a bitmap or text cast member. Figure 15-34 shows two screen shots. The center points of spheres in the pictures are used to set the points in the quad of the sprites. The first picture is using a text cast member, and it is even editable! The second is using our Mars Lander, which (if you recall) has an alpha channel.

A quad is a list of four points. Initially, the four points correspond to the corner points of a sprite's rectangle. Using quads is as easy as 1, 2, 3, 4.



Along time ago...
on a computer not
so far, far away...
Director couldn't do this!

And maybe it should
be used with care when
distorting text...

But it is so cool...
and it can even be used
with live, editable text...

Figure 15-34: Using quads to create cool distortion effects

Modifying the Quad of a Sprite Property

1. Put the quad of a sprite into a variable.

   ```
   myQuad = sprite(2).quad
   ```

   That variable now holds a list similar to the following:

   ```
   put the quad of sprite 2
   -- [point(195.0, 93.0), point(422.0, 93.0), point(422.0,
   419.0), point(195.0, 419.0)]
   ```

   (The `floatPrecision` for the preceding code was set to 1).

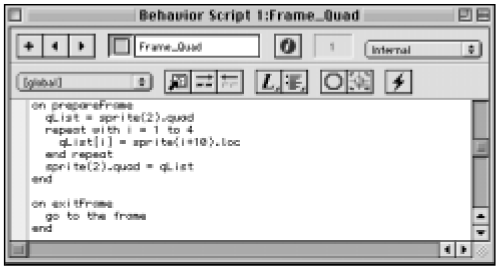
2. You then can alter the points in that list.

   ```
   myQuad[1] = point(0,0)
   ```

3. After finishing, you set the quad of the sprite to your list.

   ```
   sprite(2).quad = myQuad
   updateStage
   ```

Figure 15-35 shows the code for the `prepareFrame` in the movies shown in Figure 15-34. The graphic to be distorted is in sprite 2. The four spheres are in sprites 11 through 14; they are set to be moveable. While the movie plays, you can drag them to distort the graphic.



```
on prepareFrame
  qList = sprite(2).quad
  repeat with i = 1 to 4
    qList[i] = sprite(i+10).loc
  end repeat
  sprite(2).quad = qList
end

on exitFrame
  go to the frame
end
```

Figure 15-35: The quad of sprite 2 is changed from the frame script.

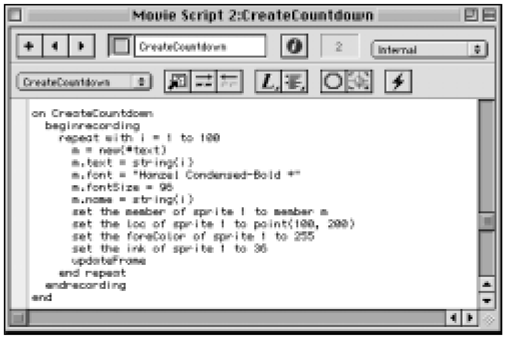
Score Recording

Score recording is very useful for automating authoring tasks. It enables to you make permanent changes to sprites in the Score. Previously discussed was the persistence of changes made from Lingo. Those changes are only when the movie plays. After you stop, the original values in the Score are still there. To make them permanent, you can use Score recording. With Score recording, you can create tools to author Director by Director. It is a better tool for authoring because you take a performance hit when you start recording.

On the CD-ROM

The movie score recording.dir is in the EXERCISE:CH15 (EXERCISE\CH15) folder on the CD-ROM that accompanies this book.

To illustrate the usefulness of Score recording, suppose that you have to create an animation that counts to 100, and for some reason it needs to be laid out in the Score (instead of just changing the contents of a field). Open the movie score recording.dir. Create a new movie script that looks like Figure 15-36.



```
on CreateCountdown
  beginrecording
    repeat with i = 1 to 100
      m = new(#text)
      m.text = string(i)
      m.font = "Hanzel Condensed-Bold *"
      m.fontSize = 96
      m.name = string(i)
      set the member of sprite 1 to member m
      set the loc of sprite 1 to point(100, 200)
      set the foreColor of sprite 1 to 255
      set the ink of sprite 1 to 36
      updateFrame
    end repeat
  endrecording
end
```

Figure 15-36: Laying out 100 frames with Score recording.

CreateCountdown starts recording with the beginrecording command. It then repeats 100 times. It creates a new text member and puts it into the variable m. It sets the text to i, which is the index of the iteration we are on. It sets the font to the compressed font in the movie, the fontSize, and the name of the member. Then it sets sprite properties of sprite 1. It sets the member, loc, foreColor, and ink properties. Last, it calls the updateFrame command. updateFrame updates the contents of the frame and moves the playback head to the next frame.

This handler is called from the Message window by typing:

```
CreateCountdown
```

Rewind the movie and play it. Open the Score and Cast windows and take a look. The Score has 100 different members in the first 100 frames. The Cast has 100 new text members. You can save this movie as **score recording 2.dir**.

Puppeting Sprites

Sometimes you want to make changes to a sprite and have those changes persist throughout the movie, but not stay fixed in the Score. One way to do this is to have a sprite span all the frames of the movie. Another way to do it is by using the puppetSprite command. When you use the puppetSprite or puppet command on a sprite channel, Director ignores any changes to the sprite that occur in the Score. It only displays changes to that sprite made from Lingo. When you puppet a sprite channel, if there is a member in the sprite, the puppeted sprite initially takes all the attributes of the current sprite. As shown in Chapter 13, you can also puppet empty score channels and place members into them. This is useful when you have no idea how many sprites you will be creating.

On the CD-ROM

You can find the movie puppetsprites.dir in the EXERCISE:CH15 (EXERCISE\CH15) folder on the CD-ROM that came with this book.

Open the movie puppetsprites.dir (shown in Figure 15-37) on the CD-ROM. We don't discuss this movie in depth, because many of the concepts were thoroughly examined in our discussion of the Mars Landing movie. If you open some of the scripts, many of the handlers should look familiar to you. They are just variations on what you did with the Mars Landing movie. Play the movie and fly around for a while.

Besides the puppeting, there are some interesting aspects of this movie. It features an animated .GIF as one of its landscape images (land 3 with the volcano). The ship is a Flash movie. It uses flipH to flip the ship sprite. In addition, objects control all the sprites.

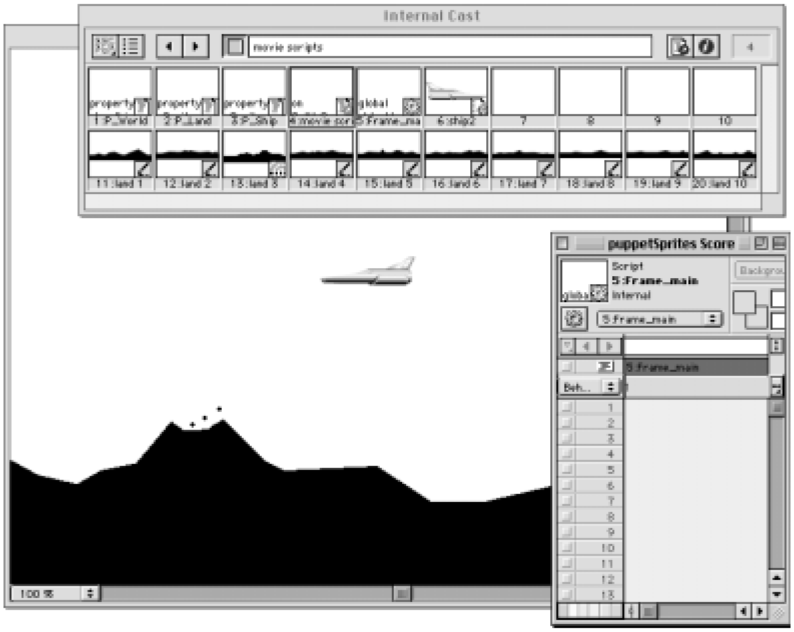
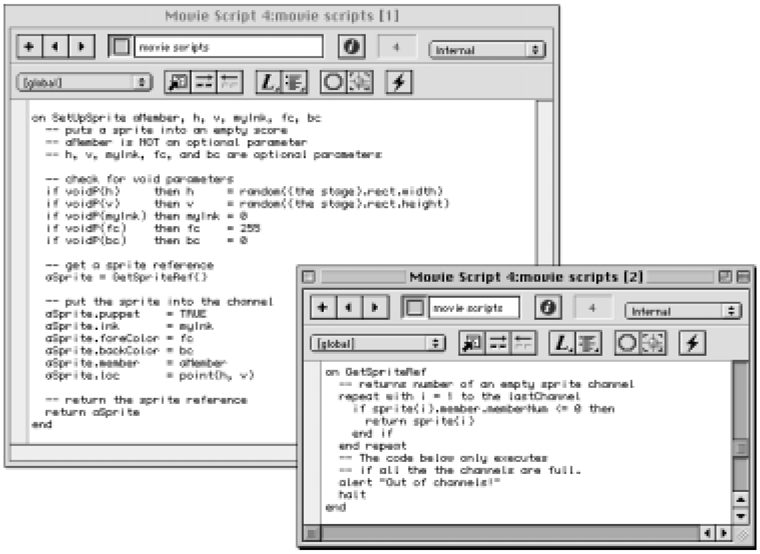


Figure 15-37: The puppetSprites movie has a single item physically in the Score, and the rest is puppeted.

Open the script named movie scripts. In it, you see two handlers — both are functions (they return a value), as shown in Figure 15-38.

Tip If you want to open two script windows of the same script, as is done in Figure 15-38, Option+click (Alt+click) on the Next or Previous Cast member buttons (the arrows at the upper left), and then click back to the script you were in. Holding the Option (Alt) key while opening any script window opens another window, instead of changing the script that you see in the current script window.

`SetUpSprite` is called from both the land and ship parent scripts. It has six parameters. The only one that is not optional is `aMember`, a reference to a member. If any of the other parameters have a void value, a default value is substituted. `SetUpSprite` calls `GetSpriteRef` to find an empty sprite channel. If all the channels are full, an alert pops up and the movie halts. Otherwise, `SetUpSprite` puppets the sprite and sets the necessary properties. You need to set at least the six properties shown (including the `puppet` property) in order for it to appear on the Stage.

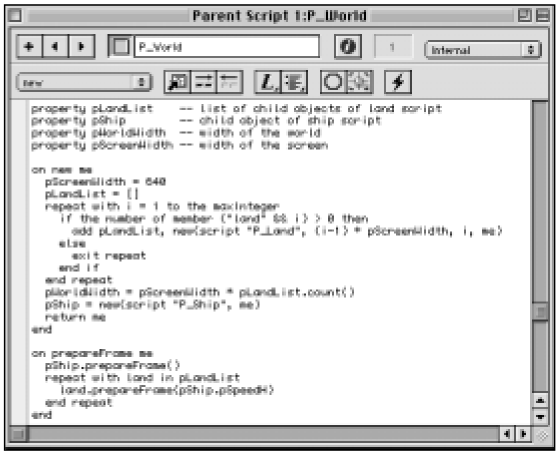

Movie Script 4:movie scripts [1]
movie scripts
4
Internal
[global]
Movie Script 4:movie scripts [2]

```
on SetUpSprite aMember, h, v, myInk, fc, bc
  -- puts a sprite into an empty score
  -- aMember is NOT an optional parameter
  -- h, v, myInk, fc, and bc are optional parameters

  -- check for void parameters
  if voidP(h)     then h     = random((the stage).rect.width)
  if voidP(v)     then v     = random((the stage).rect.height)
  if voidP(myInk) then myInk = 0
  if voidP(fc)    then fc    = 255
  if voidP(bc)    then bc    = 0

  -- get a sprite reference
  aSprite = GetSpriteRef()

  -- put the sprite into the channel
  aSprite.puppet    = TRUE
  aSprite.ink       = myInk
  aSprite.foreColor = fc
  aSprite.backColor = bc
  aSprite.member    = aMember
  aSprite.loc       = point(h, v)

  -- return the sprite reference
  return aSprite
end
```

```
on GetSpriteRef
  -- returns number of an empty sprite channel
  repeat with i = 1 to the lastChannel
    if sprite(i).member.memberNum <= 0 then
      return sprite(i)
    end if
  end repeat
  -- The code below only executes
  -- if all the the channels are full.
  alert "Out of channels!"
  halt
end
```

Figure 15-38: Functions that find an empty sprite channel and put a member in it.

Figure 15-39 shows our reason for choosing puppeting. When the `world` object generates the `pLandList`, it does not know how many members might make up that list. It keeps adding references to the list until no more can be found. In this case, when it looks for a member named land 11, `the number of member` will return a –1. It is assumed that the members are named in sequential order (land 1, land 2, and so on), with no gaps. This is a handy technique to keep your code flexible. If you were to add another landscape, all you need to do is give it the right name. On the other hand, you might decide you have too many landscapes and delete the last few. The code will still work without having to make any changes.



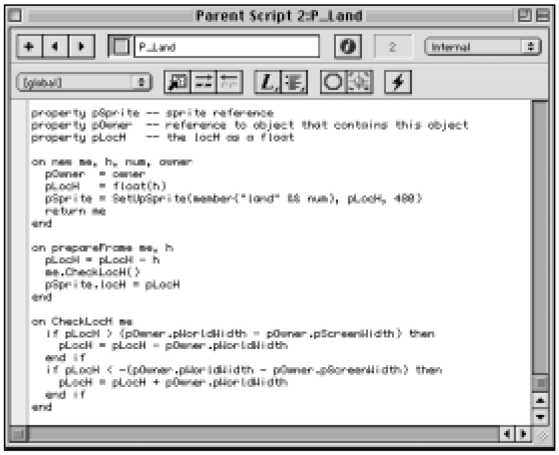
```
property pLandList    -- list of child objects of land script
property pShip        -- child object of ship script
property pWorldWidth  -- width of the world
property pScreenWidth -- width of the screen

on new me
  pScreenWidth = 640
  pLandList = []
  repeat with i = 1 to the maxinteger
    if the number of member ("land" && i) > 0 then
      add pLandList, new(script "P_Land", (i-1) * pScreenWidth, i, me)
    else
      exit repeat
    end if
  end repeat
  pWorldWidth = pScreenWidth * pLandList.count()
  pShip = new(script "P_Ship", me)
  return me
end

on prepareFrame me
  pShip.prepareFrame()
  repeat with land in pLandList
    land.prepareFrame(pShip.pSpeedH)
  end repeat
end
```

Figure 15-39: The New() function for the world parent script

The parent script P_Land is shown in Figure 15-40. The landscape moves in the `prepareFrame` handler. The event handler is initially called from the frame behavior Frame_main, shown in Figure 15-41. Then the `gWorld` object's `prepareFrame` handler, shown in Figure 15-39, manually propagates the `prepareFrame` event. The `prepareFrame` handler in the P_Land parent script (and the ship's, P_Ship) does not automatically get this event, as would a behavior; the event must be called from your code. `gWorld` propagates the message to the ship object `pShip` and to the land objects contained in `pLandList`. When it calls the `prepareFrame` for the land objects, it also passes along the `pSpeedH` property of the `pShip` object.

Tip The advantage of this technique is the capability to control when objects receive events and pass additional parameters.

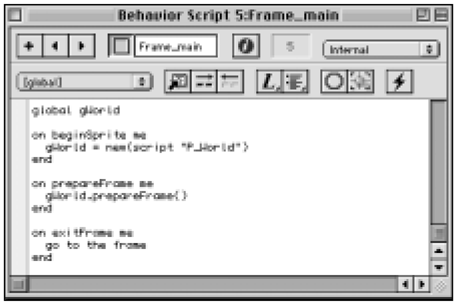
The only behavior in our movie is the script Frame_main (see Figure 15-41). It also happens to be the only item in the Score. It creates the child object of the parent script P_World in the `beginSprite` handler and puts it into the global variable `gWorld`. Every time the behavior gets a `prepareFrame` event, it propagates the message to the world object contained in `gWorld`.



```
property pSprite -- sprite reference
property pOwner  -- reference to object that contains this object
property pLocH   -- the locH as a float

on new me, h, num, owner
  pOwner  = owner
  pLocH   = float(h)
  pSprite = SetUpSprite(member("land" && num), pLocH, 400)
  return me
end

on prepareFrame me, h
  pLocH = pLocH - h
  me.CheckLocH()
  pSprite.locH = pLocH
end

on CheckLocH me
  if pLocH > (pOwner.pWorldWidth - pOwner.pScreenWidth) then
    pLocH = pLocH - pOwner.pWorldWidth
  end if
  if pLocH < -(pOwner.pWorldWidth - pOwner.pScreenWidth) then
    pLocH = pLocH + pOwner.pWorldWidth
  end if
end
```

Figure 15-40: The parent script used for the land objects

Tip Another advantage of this technique is that the entire game could be paused simply by stopping `gWorld` from getting `prepareFrame` events. Because Director would still be looping on the frame, it would continue to be processor friendly, handling events. This is a much better way to pause a game than locking up the processor in a repeat loop.



```
global gWorld

on beginSprite me
  gWorld = new(script "P_World")
end

on prepareFrame me
  gWorld.prepareFrame()
end

on exitFrame me
  go to the frame
end
```

Figure 15-41: The frame behavior on frame 1 is the only item in the Score. It creates the world in the beginSprite handler.

Summary

By the end of this chapter, you should feel much more comfortable taking control of sprites via Lingo. You learned a few things in this chapter including:

✦ The screen coordinate system is similar to the Cartesian coordinate system, except that the *y*-axis is flipped so that *y* increments increase as you go *down* the axis.

✦ Sprites can be moved, rotated, skewed, flipped, and scaled from Lingo.

✦ Changes from Lingo only persist for the length of the sprite's span in the Score, unless you puppet it or permanently change it through Score recording.

✦ Without Score recording, changes from Lingo revert to the Score after the movie stops.

✦ Using trigonometry functions to move the sprite at different angles can provide more realistic motion.

✦ Swapping a sprite's member to create an animated effect should be second nature by now.

✦ Multiple scrolling background images can be implemented to give the illusion of a large world.

In the next chapter, we look at how to manipulate all aspects of text members by means of Lingo.

✦ ✦ ✦