CHAPTER 14

Building Buttons

In This Chapter

Building your own button behaviors

Mouse events

Swapping cast members with Lingo

Creating your own handlers within the behavior to be called by mouse event handlers

Creating a Parameters dialog box

If you've been reading sequentially, you've just finished some pretty intense chapters on learning to program using Lingo. This chapter is a bit of a breather for you. It is focused on a very common and specific task—building buttons.

Director 8 comes with a wide assortment of prebuilt behaviors for handling buttons. If you open one of those behaviors, you might be daunted by the amount of code it takes. It is easier to understand these complex behaviors after you've written some of your own. Maybe you need to do something that falls outside the scope of the prebuilt behaviors; maybe you prefer that your projects have your logic; or maybe you just want to understand how they work. This chapter is useful whether you are interested in building your own behaviors or just want to understand how something that seems so simple can be so complex.

Mouse Events

Buttons typically have a minimum of two states: up and down. Another common state is the roll-state—the state of the button when a mouse rolls over it but the button has not yet been clicked. A button can also be grayed-out, a state in which it is inert. Events (and a little scripting) cause these states to change. Standard events for a button are the following:

- `mouseEnter` occurs when the cursor enters a sprite's bounding rectangle.
- `mouseWithin` occurs repeatedly while the cursor is inside a sprite's bounding rectangle.
- `mouseDown` occurs when the user clicks a sprite.
- `mouseLeave` occurs when the mouse leaves the bounds of the sprite (whether the mouse is up or down).

- `mouseUpOutside` is a sprite that received a mouseDown but the mouse button was released outside of its bounds.
- `mouseUp` occurs when the mouse button is released over a sprite that received a mouseDown event.

These events are only sent to sprites that have behaviors attached to them. To make use of these events, you need to create a behavior. Figure 14-1 shows a screen with eight buttons. The top row is for illustrative purposes, showing all the available states. You apply the scripts to the bottom row of buttons.

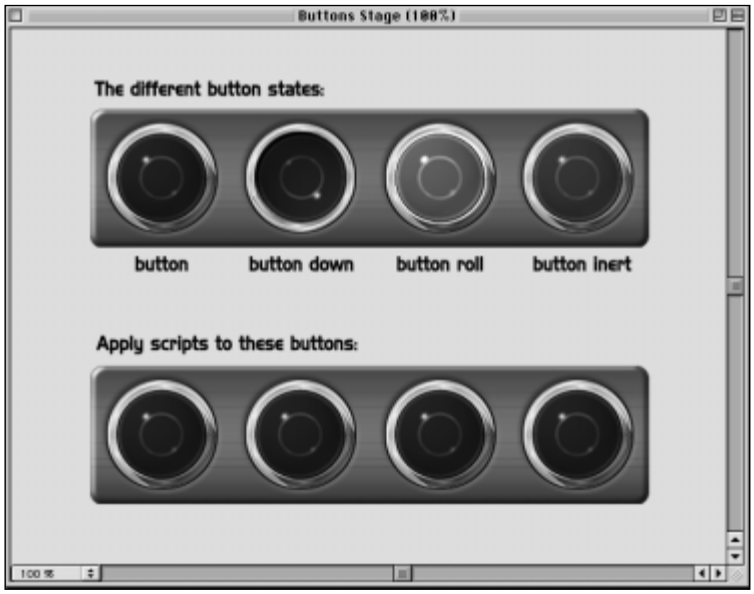


Figure 14-1: A bunch of "dumb" buttons waiting for behaviors.

Creating and applying a behavior

To better understand event handlers and how they work with sprites, work though the following steps. You are creating a behavior that has all of the most common sprite-related events, when those handlers have a single `put` statement so you can see them execute in the Message window.

All of the various Stages of the buttons.dir movie are on the CD-ROM in the folder EXERCISE:CH14 (EXERCISE\CH14). Start from scratch with buttons.dir, and if you get stuck anywhere along the line, check out button1.dir through button5.dir.

Creating a Behavior

1. Open the movie buttons.dir.
2. Select the four bottom buttons by clicking the first one and holding down the Shift key while clicking the others. Alternatively, you can open the Score window and select sprites 5 through 8.
3. In the Behaviors pop-up menu on the Score or in the Sprite Inspector, choose New Behavior. Alternatively, you can Ctrl+click (right-click) one while they're all selected, and then choose Script from the pop-up menu.
4. You're presented with an almost empty behavior, just an `on mouseUp` handler waiting for more code. Enter the code shown in Figure 14-2. Name it **B_mouse events**.
5. Open the frame script for frame 1 and add a `go to the frame` statement to the `on exitFrame` handler.
6. Open the Message window.
7. Play the movie. As it plays, roll the mouse over the buttons, and click and release over the buttons and off them. Note the feedback in the Message window.

Figure 14-2: A script that puts the sprite reference and the name of each mouse event to the Message window as the event occurs.

Sprite references

In Figure 14-2, the `beginSprite` handler sets a property called `pSprite`. As Chapter 13 explains, the `me` parameter contains an instance of the behavior for the sprite to which it is attached. Every behavior has one property built-in, and that is the `spriteNum`. The `spriteNum` is the number of the sprite channel for the sprite to which the behavior is attached.

The statement `pSprite = sprite(me.spriteNum)` gets the value of `spriteNum` from `me`, and then the sprite keyword returns a reference to the sprite, such as `(sprite 5)`. A good way to understand this is to go back to our trusty Message window. In it, type the following:

```
mySprite = sprite 5
put mySprite
-- (sprite 5)
```

What's the value of this? Well, instead of typing one of the following:

```
put the loc of sprite 5
-- or
put sprite(5).loc
```

You can give it a meaningful name and skip typing the word *sprite*:

```
put the loc of mySprite
-- or
put mySprite.loc
-- point(135, 363)
```

It also works well with dot syntax, as shown here:

```
put mySprite.member.name
-- "button"
```

In the case of the mouse event behavior, we are calling the sprite reference `pSprite`. As mentioned in Chapter 13, we start all property variables with a *p* to remind us that they are properties. This is just a convention we use; you can name it any legal variable name.

Your Message window probably looks something like the following:

```
-- "(sprite 6) mouseEnter"
-- "(sprite 6) mouseDown"
-- "(sprite 6) mouseUp"
-- "(sprite 6) mouseLeave"
-- "(sprite 7) mouseEnter"
-- "(sprite 7) mouseDown"
-- "(sprite 7) mouseLeave"
-- "(sprite 8) mouseEnter"
-- "(sprite 7) mouseUpOutside"
-- "(sprite 8) mouseUp"
```

```
-- "(sprite 8) mouseDown"
-- "(sprite 8) mouseLeave"
-- "(sprite 8) mouseUpOutside"
```

Try uncommenting the statement in the `mouseWithin` handler and playing the movie again. You'll understand why we commented it out when you roll over one of the buttons. Every time the playback head moves, if the mouse is over the sprite, it is sent a `mouseWithin` event. If your frame rate is set to 15 fps, you have 15 `mouseWithin`s `put` to your Message window every second. Remember that the playback head is moving, even when it is looping on a single frame. See the "Sprite References" sidebar for more information on the topic.

Adding behavior to the behavior

Continuing with the Buttons example, let's now add some functionality. We're going to use three of the buttons in the cast to create a button with an up, down, and roll state. Open the script window for the script named B_mouse events and add the code shown in Figure 14-3. You can keep the `put` statement in there; it's nice to have feedback while you're developing.

Figure 14-3: A button behavior with three states

Now when you play the movie, your buttons behave like buttons — almost (we'll address this "almost" in a little bit).

In the script's `beginSprite` handler, it assumes that the member currently in the sprite channel is the one you want to put in `pUpMember`. The next two — `pDownMember` and `pRollMember` — are derived on the assumption that they have the same name as `pUpMember` but with *down* and *roll* added, respectively.

Remember that the `&&` operator concatenates two strings *and* puts a space between them. So, when the name of `pUpMember` is accessed in the following line:

```
pDownMember = member(pUpMember.name && "down")
```

`(pUpMember.name && "down")` evaluates to "button down." Then "button down" is used as a parameter to `member()` to return a reference to the member "button down". See the "Member References" sidebar for more information on the subject.

This naming convention is usable even if the name of the button consists of multiple words. It is also good because all button states of the same group start with the same name. Using this convention, you can apply this behavior to the button sprite and it automatically uses all three cast members. If you made a mistake in your cast member naming convention, you'll get a syntax error warning that Director couldn't find the cast member.

Better Button Behavior

You might think we're about finished, and for some purposes, maybe we would be. But there is a problem with the way Director handles mouse events.

1. Play your movie and put the mouse over the button at the far right.
2. Click the button.
3. While holding down the mouse button, roll the cursor to the left.
4. The second button from the left becomes highlighted.
5. Release the mouse button over the second button. *That* button gets the `mouseUp`!

Your Message window should say:

```
-- "(sprite 8) mouseEnter"
-- "(sprite 8) mouseDown"
-- "(sprite 8) mouseLeave"
-- "(sprite 7) mouseEnter"
-- "(sprite 8) mouseUpOutside"
-- "(sprite 7) mouseUp"
```

Member references

First there were sprite references, and now there are member references. In Figure 14-3, three new properties were added to the script B_mouse events—`pUpMember`, `pDownMember`, and `pRollMember`. These properties were initialized in the `beginSprite` handler.

To understand what a member reference is, go back to the Message window. In it, type the following:

```
myMember = member "button"
put myMember
-- (member 11 of castLib 1)
```

The `myMember` holds a value that refers to `member 11` of `castLib 1`. Putting the member reference into a variable saves us from having to write code like this:

```
myMemName = "button"
put the type of member myMemName
-- #bitmap
```

Now you can do things like the following:

```
put myMember.type
-- #bitmap
put myMember.name
-- "button"
put myMember.rect
-- rect(0, 0, 98, 97)
```

You might prefer this sort of action. The built-in buttons on the Tool Palette respond in the same manner. We find several problems with this functionality:

- If you press the mouse button down on the Stage, and release over a button, it does *not* receive a `mouseUp` event, even though it gets a `mouseEnter` event.
- If you press the mouse button down on any other sprite with a script and release the mouse over a button, it *does* receive a `mouseUp` event.
- Given good user interface standards, one would expect only the button that received the `mouseDown` event to respond to the `mouseUp` event. Even Director's own interface follows this standard. Try clicking the Save button on the Toolbar that runs across the top of the screen. Then, with the mouse button still down, roll over the Save All button. Note that it does not become highlighted, and when you release the mouse button over it, it does not activate. Try some of the buttons in the Paint window or Toolbar as well.

Controlling events a sprite receives

It seems only appropriate that the buttons we create in Director should respond in the same manner as buttons in other programs. There is an easy way and a more involved way to get around this problem. The easy way is to set the system property `the buttonStyle` to 1 (the default is 0). A good place to do this is in an `on prepare Movie` event handler. Now the buttons you create will work like most buttons.

The easy way is not as educational, nor does it take into consideration the poss-ibility that you want both styles of button behavior in the same movie. Another situation where you may want to go the more involved route is when you are modifying an existing movie that assumes `the buttonStyle` will be set to its default value. Given the obscurity of this property, this is not unusual. So let's take a look at solving this problem the hard way. Due to the length, you add the code in two parts (see Figure 14-4).

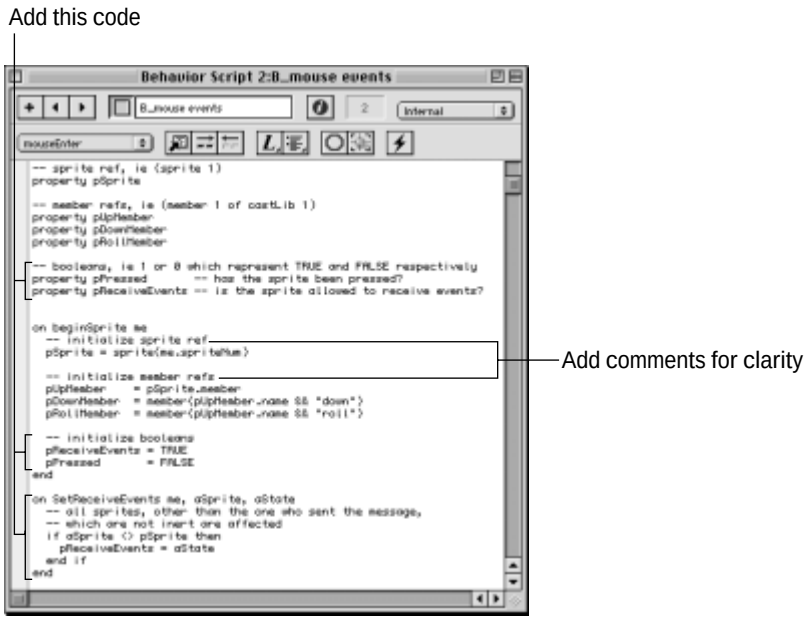


Figure 14-4: Two new properties, plus a handler, have been added.

First, add the new properties `pPressed` and `pReceiveEvents`. These both hold Boolean values (TRUE or FALSE). They are initialized in the `beginSprite` handler. A new handler is defined for this behavior as well; it is called `SetReceiveEvents`. `SetReceiveEvents` is not a Lingo keyword; it is just a user-created handler, so we could have called it anything. The purpose of this handler is to change the value of

the property `pReceiveEvents`. This handler is called from other objects. `SetReceiveEvents` has one parameter: `aState`. `aState` holds a Boolean value of TRUE or FALSE, which is assigned to `pReceiveEvents`. It also checks to make sure that it isn't the sprite that sent the `SetReceiveEvents` message.

Now, some code needs to be added to the mouse event handlers, as shown in Figure 14-5. For each event, we test whether or not the object should be receiving events by testing the value of the property `pReceiveEvents`.

Add this code

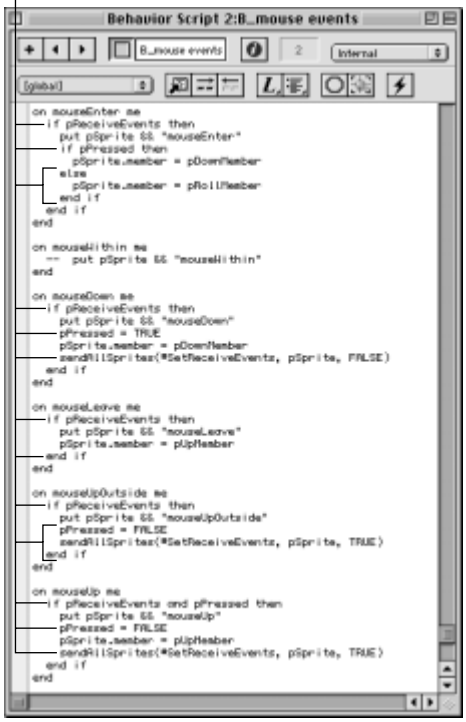


Figure 14-5: The mouse events are beginning to grow in complexity.

Starting with the `mouseEvent` handler, another `if` statement is used to check whether `pPressed` is TRUE. If it is, the button goes down. If it is not, then it turns into the roll state.

Next is the `mouseDown` handler. Assuming it receives events, the `mouseDown` handler sets `pPressed` to TRUE. It changes the member. Last, it sends the `SetReceiveEvents` message to all of the sprites, passing its sprite reference and FALSE (which prevents any of the other buttons from receiving mouse events).

"Sending a message" means calling the handler of an object. The command `sendAllSprites` enables you to call a specific handler in every behavior on every sprite. If the sprite does not have the handler, it ignores the message.

`mouseLeave` doesn't do anything different other than check to see whether it is receiving events. `mouseUpOutside` sets `pPressed` to FALSE, which makes sense — if the mouse button is up, it is no longer pressed. It also sends the `SetReceiveEvents` message to all sprites, this time passing TRUE to again enable them to receive events.

Last, `mouseUp` checks not only whether it can receive events, but also whether `pPressed` is true. `pPressed` is TRUE only if it was set to TRUE in the `mouseDown` event. It then switches the member and tells all the other sprites that they can get events once more. Now play the movie and test the buttons as you did at the beginning of this section. Don't forget to have the Message window open. Now these buttons are behaving better!

These new changes to the code give us some added capability as well. Just to show you it's worth the extra work, play the movie and try this in the Message window:

```
(sprite 5).mouseEnter()
-- "(sprite 5) mouseEnter"
```

The movie must be playing when you try this and the next example, because behaviors are not instantiated until the movie actually plays and the playback head enters the frame the sprite is on. Our handy `put` statement lets us know the Message was received; we also know this because the button changes to a roll state. You can now control button presses from Lingo.

```
(sprite 5).mouseDown()
-- "(sprite 5) mouseDown"
sprite(5).mouseUp()
-- "(sprite 5) mouseUp"
```

It's like a ghost pressing the buttons.

Adding an inert state

You might be asking yourself "Are we done yet?" We're glad you asked! Of course not; you can always add more code! Writing scripts is like creating art; you're never really finished (unless you've got a deadline). Besides, we still haven't used the member-named button inert yet.

The *inert* state is a state that receives no events. In our example, a button becomes inert after it receives a `mouseUp` event.

To add this functionality, we need to add two more properties — `pInertMember` and `pInert`. `pInertMember` is a reference to the bitmap member-button inert. Add the properties as shown in Figure 14-6. Also, make the changes to `beginSprite`, `SetReceiveEvents`, and `mouseUp`, as shown in Figures 14-6 and 14-7.

Add this code

Figure 14-6: pInertMember and pInert properties are added to the behavior and added to beginSprite. SetReceiveEvents now also tests pInert.

Add this code Change this code

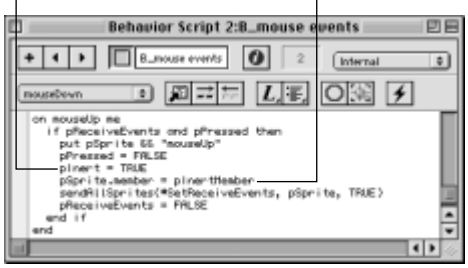


Figure 14-7: pInert is set in the mouseUp handler. pInertMember is also used.

Play the movie again and click one of the buttons. It becomes gray and no longer responds to mouse events. Being a little obsessive is the sign of a good programmer, so let's just make one more little addition. At the moment, we have no way to reenable an inert button. You *can* do it through Lingo, of course.

Click a button to make it inert. You see the following in the Message window, for example, if you click the button in sprite channel 8:

```
-- "(sprite 8) mouseEnter"
-- "(sprite 8) mouseDown"
-- "(sprite 8) mouseUp"
```

Now, in the Message window, type the following:

```
(sprite 8).pReceiveEvents = TRUE
(sprite 8).pInert = FALSE
```

You could also write it like this:

```
sprite(8).pReceiveEvents = TRUE
sprite(8).pInert = FALSE
```

Now you can roll over the button and click it. We just need to insert the code in the behavior. Let's put it into the `mouseDown` handler, as shown in Figure 14-8.

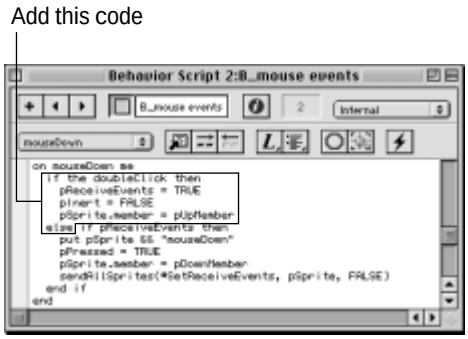


Figure 14-8: A few statements added to reenable the button when it is double-clicked.

When you work on successive generations of a script (where each generation is saved separately), sometimes it's helpful to compare the code from one version to another. Director does not support this, but programs such as Microsoft Word and BBEdit do. All you need to do is copy and paste the script into separate documents for comparison.

Creating a Parameters Dialog Box

At this point, you have a great button behavior. It's easy to use—just make sure you've named your cast members properly, and it automatically finds them. It behaves well. You might like it enough that you want others to use it . . . maybe even nonprogrammers. Nonprogrammers might find adhering to the naming convention too rigid, and might find going into the script—to add functionality in the `mouseUp` handler—scary. Here's how Director makes your life (and their lives) easier. You get to write pages of fun code, and they just get to see what they need.

Building a dialog box

First, we need to change our naming convention slightly. The naming convention is optional, but if you make this change and adhere to it, you'll like the results. Earlier, the assumption was made that the up state for a button was just whatever name you wanted to call the button. Suppose that you called it "green button." The subsequent states would have the same name, plus *down*, *roll*, and *inert* added to them. From this point, the up state should have *up* after it. So, we have the following:

```
"green button up"
"green button down"
"green button roll"
"green button inert"
```

When the user drags the behavior onto a sprite, we see the dialog box in Figure 14-9. The goal is to have the box take a good guess, based on naming conventions. If naming conventions weren't followed, the first graphic in the movie appears in the pop-up menu for each button state (it just doesn't look as cool). If the right member is not selected, the user can always click the pop-up menu next to the state and choose the desired member. Also included are check boxes to decide whether a button should be inert initially, whether it should be inert after it is used, and an Action pop-up menu to let the user choose from one of the Lingo commands you decide to put in it.

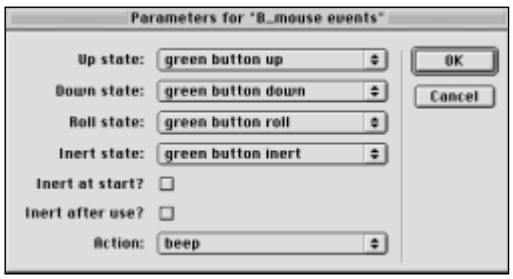


Figure 14-9: We want this dialog box to pop up when a user drags the behavior onto a sprite.

That's what we want, but how do we get there? By using the event handler `getPropertyDescriptionList`. This handler runs when a behavior is applied to a sprite. In it, you define a property list that is used to build the dialog box shown in Figure 14-9.

Add the handler shown in Figure 14-10 to the behavior B_mouse events. It doesn't matter where you put it; we put it after the `beginSprite` handler. The handler returns the property list `myList`. `myList` is composed of smaller property lists. Each property list in `myList` corresponds to a property defined in the behavior. The first property added to `myList` is `#pUpMember`, which corresponds to the property `pUpMember`. When a user chooses a member for the up state of the button, that member is placed into `pUpMember`. The list for `pUpMember` list includes the following:

- `#comment`, which is used to the left of the control being created.
- `#format` for the value. This is the value's type. We could have been more specific and chosen `#bitmap`, `#vectorShape`, or `#filmloop`, for example, but `#graphic` is handy because you are not limited to the type you want to use for each state of the button. This enables you to have an animated . GIF for the up state (if you want), and bitmaps or other graphics for the other states.
- `#default` is the value that appears when the dialog box first opens.

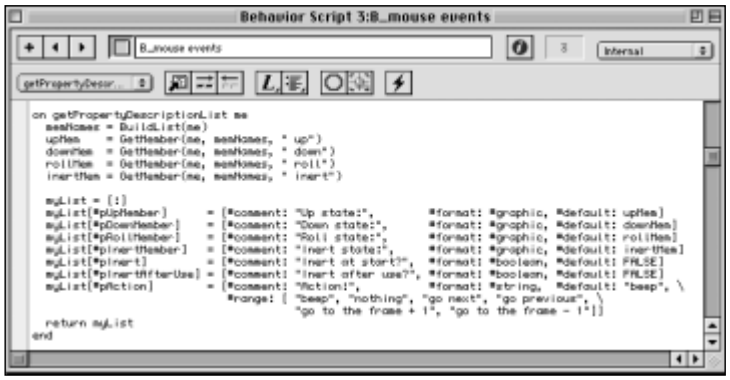


Figure 14-10: Use getPropertyDescriptionList to build the dialog box shown in Figure 14-9.

Farther down, you can see that `pInert`'s `#format` is set to `#boolean`. This presents the user with a checkbox. If it gets checked, TRUE is returned; if it doesn't, FALSE is returned.

- The last list added is for the `pAction` property (we'll declare that in a moment). It's `#format` is `#string`. It also has a `#range` value, which is a linear list. Because the `#format` is `#string`, the linear list is filled with strings. This list is used to create a pop-up menu for the user. Here, we've added several common Lingo commands to choose from. See Figure 14-11 to see how it appears to the user in the Property Inspector.
- Not used in our program, but still useful is a variation of `#range`. Its format is either `#float` or `#integer`. The list `#range` has a property list with two properties: `#max` and `#min`. This creates a slider on the interface. The syntax is `#range: [#max: 5, #min: 1]`.

Figure 14-11: The #range property enables you to specify what appears in a pop-up menu, as shown here. You can also use #range to create a slider.

At the top of the `getPropertyDescriptionList` handler, we create some variables. The first is `memNames`. It is a list of the names of all the members in the movie. This list is built from the function `BuildList()`, shown in Figure 14-12. Go ahead and add it right after the `getPropertyDescriptionList` handler.

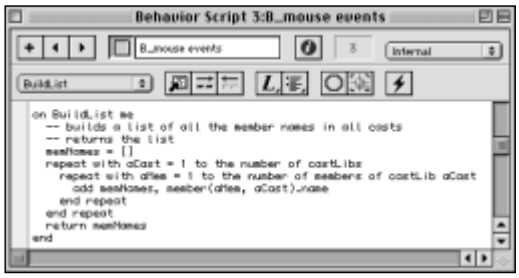


Figure 14-12: BuildList() returns a list of every name in every cast. It is called from getProperty DescriptionList.

`BuildList()` could easily be made into a movie handler, such as `prepareMovie`, that is only called once at the beginning of a movie and sets a global variable that this behavior would access. For convenience and not efficiency, we put it right in the behavior. This way, if you ever want to give this behavior to someone, you don't have to worry about remembering the movie script as well. Plus, it makes it easier to integrate into other movies by not having to merge one `prepareMovie` with another `prepareMovie`. It is all encapsulated in this one behavior (which is one of the attributes of object-oriented programming).

Then, second through fifth statements in the `getPropertyDescriptionList` handler shown in Figure 14-10 are variables that hold member references. They get these member references from `GetMember()`. `GetMember()` takes two parameters (in addition to the me parameter): `memNames` and `aName`. `memNames` is a list of member names, and `aName` is a string. Specifically, `aName` is expected to be a substring of one of the member names. The second statement,

```
upMem = GetMember(me, memNames, " up")
```

passes the `memName` list as an argument as well as the string literal "up." `GetMember()`, shown in Figure 14-13, searches through `memNames` looking for a string that has (in this case) *up* in it. If it finds a string in the list containing it, a reference to that member is returned, breaking out of the loop and the handler. The variable `upMem` is then used to set the default for the `#pUpMember` property list in the `myList` property list.

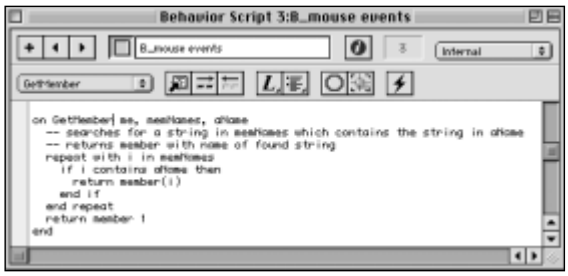
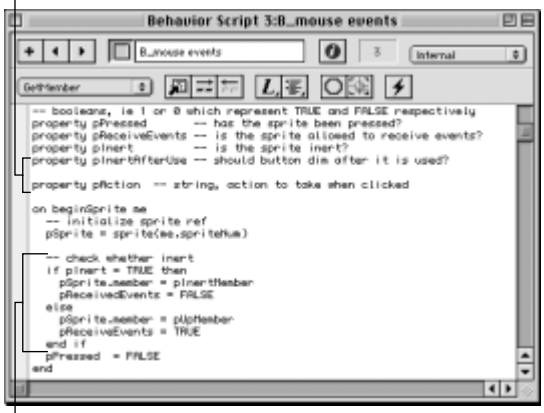


Figure 14-13: GetMember searches through a list for a substring. It is called from getPropertyDescriptionList.

`BuildList()` and `GetMember()` enable us to vary the naming of the buttons. Try renaming the buttons, but keep the suffix on each. The Parameters dialog box displays the first member it finds with a name containing *up* in the "Up state" pop-up menu, the first member it finds with a name containing *down* in the "Down state" pop-up menu, and so on.

To accommodate these changes, we needed to add the properties `pInertAfterUse` and `pAction`, as shown in Figure 14-14. The `beginSprite` function was changed drastically. We removed the initializations of all the properties that are given values by the Parameters dialog box.

Add this code



Change this code

Figure 14-14: Two new properties and a very different beginSprite handler

The `mouseUp` handler has some changes as well, as shown in Figure 14-15. It uses both of the new properties declared in Figure 14-14.

Add this code

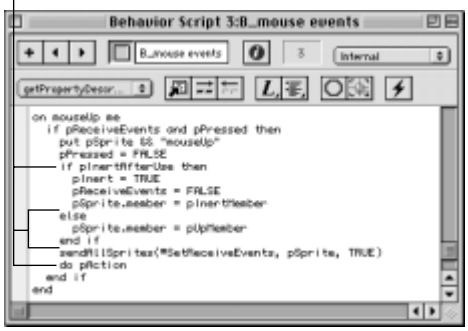


Figure 14-15: The restructured mouseUp handler makes use of both new properties: pAction and pInertAfterUse.

Documenting behaviors

There are two event handlers that aid in the documentation of a behavior's functionality: `getBehaviorDescriptonList` and `getBehaviorToolTip`. These are shown

in Figure 14-16. The `getBehaviorDescription` event handler returns a string that appears in the Behavior Inspector. The `getBehaviorToolTip` event handler returns a string used as a ToolTip when this behavior is in the Library Palette.

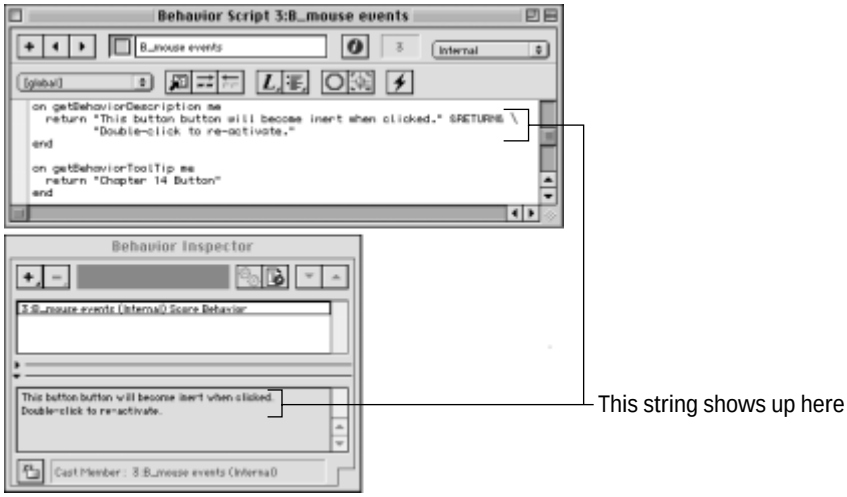


Figure 14-16: Documenting a behavior is easy with the getBehaviorDescription event handler. The string you specify it to return appears in the Behavior Inspector.

Summary

This chapter focused on the task of making a button that responds in a way that users have come to expect. It also illustrated the many events to which a sprite can respond. Specifically, you learned that:

- ✦ Button behaviors can be made quickly, but buttons that follow standard user-interface guidelines take a little more work.
- ✦ Sprites can send messages to other sprites.
- ✦ Mouse events are called by Director or from Lingo.
- ✦ You can put references to members and sprites in variables.
- ✦ You can create you own handlers within behaviors.
- ✦ Behaviors provide a way to create online documentation through the Behavior Inspector.

The next chapter discusses how you can control sprites by using Lingo.

✦ ✦ ✦