INTRODUCTION

I love working with Director. By education, I am a computer scientist and a journalist. By nature, I am a problem solver. I love using both logic and imagination. Director requires me to use all these things.

If you have a job in which you work all day with Director, consider yourself lucky. You probably already do. In fact, you have probably said to yourself before: "I can't believe I get paid for this!"

Director is a great tool for creating software. On one hand, you can quickly bring to life your ideas. On the other hand, Director is an environment that inspires new ideas as you explore it.

The possibilities for animators are incredible, even if they never choose to do any programming. Director also has built-in programs called *behaviors* that expand the possibilities a thousand times for those looking to make interactive presentations.

If you are willing to go on to learn *Lingo*, the programming language of Director, the possibilities for your creations become virtually limitless. There are more than 800 Lingo keywords in Lingo including all standard programming language structures. This makes Lingo every bit as powerful as languages such as C++, Pascal, and Java. In some cases, it's even more powerful.

The results of your work can be easily distributed to the world. You can create standalone applications to send over the Internet or burn into CD-ROMs. You can embed your creations into HTML pages for the Web. You can even create a Java applet.

Many people in the computer industry still think of Director as it was in version 3: an animation tool with a simple scripting language. However, each version since then has added a huge array of powerful features. Director is now the most advanced animation tool ever, with one of the most advanced programming languages.

Anyone who says, "You can't do that in Director," doesn't really understand the power or depth of Director 8.

Nintendo Exhibit 1029
Page 00191

WHOM IS THIS BOOK FOR?

Chapter 1, "Animation with Director," assumes that you are a total beginner and have never used Director before. However, it does not assume that you lack intelligence.

The first two chapters move quickly through the basics. Those chapters, like the rest of this book, state concepts and techniques clearly, and never assume you know something about Director before it is taught in the book.

The idea is to not waste time by walking you step-by-step through basic and simple tasks. Instead, this book assumes that you are a motivated learner, wanting to read, absorb, understand, and then move on to the next piece of information.

If you are an animator, or are simply using Director to create presentations, Chapters 1 through 11 are for you. They go into detail about making animations and presentations without using any Lingo programming.

If you want to learn Lingo, Chapter 12, "Learning Lingo," and the following chapters teach you from the ground up. Like Chapters 1 and 2, these chapters do not assume you know anything about Lingo or programming in advance. However, they move quickly, and a motivated learner can be programming in Lingo in a matter of hours.

If you are interested in advanced Lingo techniques, the later chapters are filled with high-level Lingo programs. I have continued the practice of my earlier books in providing more advanced Lingo concepts than any of the other books on the market.

Finally, if you are seeking a good reference book on Director, this book will beat or rival any other. I have tried to cover all topics, even ones that other books do not explore. In addition, I have made the reference section comprehensive. It features a full Lingo dictionary and many other useful appendixes.

THIS BOOK IS ABOUT DIRECTOR 8

One thing to note is that this book is really for Director 8, not earlier versions of Director. In the past, many people have bought my Director 7 book and tried to use it to learn Director 5 or 6. They, of course, ran into problems.

The differences between Director 7 and Director 8 mean that many of the things in this book will not work in Director 7. In addition, all the examples on the CD-ROM are in Director 8 format and cannot be opened with Director 7.

Macromedia is definitely not one of those companies that fixes a few bugs and calls it an upgrade. It has added a large number of new features. Other companies should take a lesson from Macromedia about creating upgrades that are really worth the upgrade price.

If you consider yourself a serious developer, you should be using Director 8 by the time you read this. Director 7 is a great tool, but Director 8 is better. It will empower you to create better products.

Where Did This Book Come From?

This book is an update to *Special Edition Using Director* 7 by the same author and publisher. After writing a book on Director 5 and another on Director 6, there was little question in my mind that I would be working on one again when Director 7 was announced. The question was: Whom would I be writing for?

My old publisher was sadly no longer in business. So, I put the word out to other publishers.

The result was that I was contacted by Macmillan (Que) about updating the *Inside Director* series. The former author had declined to do an update, but was a friend of mine and recommended me.

After talking with the publisher, it was decided that a new direction was needed. The *Inside Director* books were very successful, but the material would be hard to update. Besides, I was up for writing a whole new tome myself.

So, the idea came up to write a new book under the *Special Edition Using* series. This series, which can be found in bookstores everywhere, is very well known and respected.

If you own one of the *Inside Director* books, you might recognize a sentence, paragraph, or phrase here and there. Probably 98% of this book is my own writing, but in a few cases the old *Inside Director* text stated something so well that I felt it was best to reuse it.

The updating of this book began during the Director 8 beta cycle, but I was able to work with the final version of Director 8 for the last month of writing. Hopefully, this will result in an error-free book that covers all the bases.

How Does This Book Differ from the Previous One?

I did several things to update this book. First, of course, I added sections on new features added with Director 8. Each new feature of Director 8 is marked as such throughout the book. In addition, I updated any part of the book that referred to a feature that somehow changed between Director 7 and 8.

In addition, I tried to improve the book throughout. This means that I added information and examples. Most of the chapters had changes; many of the chapters had significant changes.

I also added a new chapter on using the Shockwave Multiuser Server (Chapter 38, "Creating Multiuser Applications") and tried to improve the reference section of the book.

How to Use This Book

You can read this book through, or use it for random access reference. If you are just learning Director, or are looking to expand your skills, you can pick a point in the book that seems to match your current skill level and start reading there.

Each chapter makes the general assumption that you know the basics of the material in the previous chapters. However, enough context is given in each chapter for you to fill in the gaps in your knowledge as you go along. There are "See Also" markers throughout the book that refer you both forward and backward in the book to places where a similar topic is covered.

Chapters 1 and 2 are meant to teach the basics, from using Director as an animation tool to using Director as a presentation tool. Chapters 3 through 11 then add to those basics with information about each media type and some more advanced techniques. No Lingo knowledge is required or taught during these chapters.

Chapters 12 through 14 teach Lingo basics. Then, Chapters 15 through 20 build on that knowledge by showing how Lingo can control various types of media. Chapters 21 through 26 are about even more advanced Lingo techniques.

Chapters 27 to 32 are very different from the rest of the book, and from material found in other books. They provide examples of Director movies. An explanation of what each movie does and how it does it, plus the source code for each movie, is given. These are examples that you can use to see how Lingo code is put to use in real-life situations. These chapters also include suggestions for how you can alter and adapt them to make your own movies.

The rest of the chapters have information useful for completing a project, such as debugging, performance issues, and building projectors and Shockwave pages.

To round out the book as a reference guide, I have added a complete Lingo quick reference section and many tables and charts as appendixes.

The following list contains a few goals that developers might have for this book and how to go about using the book to accomplish them:

- **Learn Director as a beginner:** Start at Chapter 1 and read through Chapter 11, trying all of the examples along the way. Continue to learn Lingo in Chapter 12 when you feel you have mastered Chapters 1–11.
- **Learn to create animation with Director:** Read Chapters 1–10.
- **Learn to add interactivity to your animation:** Read Chapter 2, and then Chapters 9–11.
- **Learn to create presentations:** Read Chapters 1–11.
- **Learn how to create behaviors without using Lingo:** Read Chapter 11.
- **Learn how to program basic Lingo:** Read Chapters 12–15.
- **Learn how to create behaviors with Lingo:** Read Chapter 14.
- **Increase your Lingo skills beyond the basics:** First review Chapters 12–15, and then continue to read Chapters 16–26.
- **Learn advanced Lingo techniques:** Read Chapters 21–26.
- **Find useful source code that you can reuse:** Scan through Chapters 14–26, and then refer to Chapters 27–32 for larger examples.
- **Learn how to optimize and complete a project:** Read Chapters 33–36.

Nintendo Exhibit 1029
Page 00194

- **Learn how to translate your CD-ROM building Director knowledge to make Shockwave movies:** See Chapters 9, 22, 31, 35, and 36.
- **Fill in your existing Director knowledge with a greater understanding of members and techniques:** Read Chapters 3–8 to learn about members, 16–20 to learn about Lingo related to these members, and then 21–26 for advanced techniques.
- **Use this book as your Director 8 reference guide:** Familiarize yourself with the table of contents and index to learn where things are in the book. Also scan the appendixes to get an idea of the content there.

CONVENTIONS USED IN THIS BOOK

The following conventions are used to differentiate Lingo keywords from user-defined keywords and any other text with special emphasis:

- **Italic**—Used for official Lingo keywords. This includes handlers, commands, properties, and anything found in the official Macromedia documentation. Examples: *on mouseUp*, *on exitFrame*, *puppetSprite*, and *the ticks*.
- **Quotation marks**—Anything made up by the programmer or the author. These keywords can't be found in the official documentation, because they don't exist until the programmer creates them. Examples: "on myHandler" and "myVariable".

  Messages. A message is an invisible thing that is sent through the director environment. They have names only as a way to relate to the reader what is going on. Examples: "mouseUp" and "exitFrame". They look like official keywords, but are not.

  Values. When I want to say something such as: the value of this variable is "hello world", the value returned is shown in quotes.
- **Monospace**—Lines of code and commands that the programmer is asked to type into Director appear in monospace and bold, regardless of keyword type. For example: **`put 41+1.`**

This formatting reflects the importance of showing the difference between official Lingo keywords and made-up words.

At the end of most chapters you will find two sections: "Troubleshooting" and "Did You Know?" The first presents some common problems that developers face and how to avoid them. This is in addition to troubleshooting advice found throughout many chapters.

The "Did You Know?" section is something a little different. It contains extra information about the topic. Sometimes the information is a little more advanced than the level of the chapter. Other times it highlights little-known facts or undocumented Lingo. On occasion it simply contains an idea for an interesting application of the information taught in the chapter. In addition, "Tips" and "Notes" placed throughout the chapters provide interesting ideas and methods you might not have considered.

Nintendo Exhibit 1029
Page 00195

BEFORE YOU BEGIN...

There are several ways that I recommend beginning to use this book. The first suggestion is for first-time Director users. Play with Director first. Just open the program and play with it. Try all the menu commands and look at some of the tutorials that come with Director.

Another thing that beginners should do is check out as many Shockwave movies on the Web as you can. These give you an idea of what is possible. If you need a place to start, try `http://clevermedia.com`, a site created by my company, CleverMedia, and me.

Getting grounded by playing with Director and surfing the Web will give you some context as you start reading this book.

If you are a Director user who has just upgraded to Director 8, you should check the next section of this book, "What's New in Director 8." It has, of course, a list of most of the features new to Director 8.

There are so many new features, I honestly recommend that even an experienced Director 7 user start this book at Chapter 1. However, such a user would be able to move through most of the early chapters quickly.

Another way to use this book is to simply place it next to your computer as a reference. Need to know about using the Behavior Inspector? Chapter 11, "Advanced Techniques." Need to remember the basics for creating a behavior? Chapter 14, "Creating Behaviors." Need to create or alter a vector shape? Chapter 20, "Controlling Vector Graphics." Want to make a game? Chapter 32, "Games." You get the idea. All in all, there is a lot in this book for every level of Director user. I hope you enjoy reading this book as much as I enjoyed writing it!

Nintendo Exhibit 1029
Page 00196

CHAPTER 15

GRAPHIC INTERFACE ELEMENTS

In this chapter

Creating Display Rollovers 306

Using Check Boxes 308

Using Radio Buttons 310

Dragging Sprites 313

Creating Sliders 320

Creating Progress Bars 326

Creating Pop-Up Menus 328

Source movies for this chapter can be found on the CD-ROM in the "Book Movies" folder under folder 15.

Nintendo Exhibit 1029
Page 00197

Whether you are creating a simple presentation or a complex piece of software, it is primarily through user interface elements that the Director movie and the user interact. These elements can be as plain as pushbuttons, or as complex as slider bars. This chapter looks at the most common user interface elements and the behaviors that you use to create them.

CREATING DISPLAY ROLLOVERS

Chapter 14, "Creating Behaviors," contained a script that showed a simple rollover behavior. Typically, a rollover is a sprite that changes members when the cursor is "rolled" over it. However, there is another type of rollover as well, one that leaves the rolled-over sprite alone, but instead changes another sprite.

A typical use for this interface element is to present a list of items and then show more information about them in another part of the screen when users roll over it. Figure 15.1 shows a screen that does this. Rolling over the three items on the left brings up three different text members in the sprite to the right.

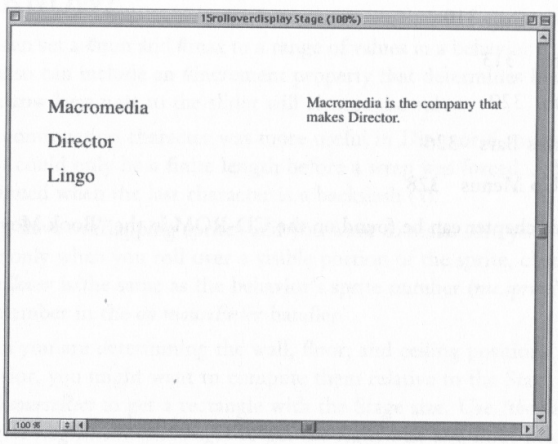


Figure 15.1 The screen shows three sprites on the left and one on the right. The one on the right changes depending on which sprite on the left the cursor is over.

The behavior to do this is similar to the rollover behavior discussed in Chapter 14. Rather than change the same sprite, it must change a different one. It also needs to know which member to use. One last parameter should be a default member that is displayed when the cursor is not over one of the rollover sprites. Here is the start of a behavior:

```
property pRolloverMember, pRolloverSprite, pDefaultMember

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pRolloverMember, [#comment: "Rollover Member",\
    #format: #member, #default: ""]
```

Nintendo Exhibit 1029
Page 00198

```
  addProp list, #pRolloverSprite, [#comment: "Rollover Sprite",\
    #format: #integer, #default: 0]
  addProp list, #pDefaultMember, [#comment: "Default Member",\
    #format: #member, #default: ""]
  return list
end
```

The "pRolloverSprite" parameter defines which sprite is changed. The actual sprite that the users must roll over is the one that the behavior is attached to. The "pRolloverMember" parameter is the member that the "pRolloverSprite" changes to. The "pDefaultMember" is the member that "pRolloverSprite" will show when the cursor is not rolled over the behavior's sprite anymore.

The default member should be the one in the rollover sprite on the Score. You can use *on mouseEnter* to change this sprite to the desired member. Then, when the cursor leaves the sprite, the *on mouseLeave* handler can switch it back to the default member.

```
on mouseEnter me
  sprite(pRolloverSprite).member = pRolloverMember
end

on mouseLeave me
  sprite(pRolloverSprite).member = pDefaultMember
end
```

As the cursor moves between the sprites, the "mouseEnter" and "mouseLeave" messages control which member the rollover sprite displays. Make sure that the sprites do not overlap; otherwise, the order of the messages can create undesired results. For instance, if sprites 1 and 2 overlap, the cursor can enter sprite 2 before it leaves sprite 1. The result would be that the default member would be displayed rather than the member specified by sprite 2.

To set this up in the Score, you need to attach this behavior to the sprites that are active and initiate the change in the nonactive sprite. Then, assign the same "pRolloverSprite" and "pDefaultMember" parameters to each, but a different "pRolloverMember" to each. This creates the basic effect discussed in this section. However, you can create more complex interfaces with different values of "pRolloverSprite" and "pDefaultMember" for each rollover.

Because you can attach multiple behaviors to a sprite, you can add a button behavior to these same sprites. That way, the rollovers are handled by the rollover display behavior here, but a click is handled by the button behavior. You can have sprites that display preview information, such as "Click this button to go to the index" in one sprite, but also react to the click to go to another frame.

→ For more information about creating rollover behaviors, **see** "Creating Simple Behaviors," **p. 277**

Nintendo Exhibit 1029
Page 00199

USING CHECK BOXES

A check box is a button that has two states: on and off. Director has a check box member type that is a variant on the button member. You can use this built-in member as a check box, or build your own check boxes with two members per selection.

A behavior for the built-in check box member is almost not needed. The member reacts to a mouse click by itself, placing a mark in the box when clicked and then removing it on a second click. Figure 15.2 shows a small group of check box members.



Figure 15.2 A group of three check box members.

One thing that a behavior can do is make it easier to determine which check boxes are checked. You can get this as a simple Lingo property of the member. For instance, in this example

```
member("apple check box").hilite
```

the *hilite* property returns *TRUE* or *FALSE*.

A behavior can use similar syntax, but because behaviors are assigned to sprites, not members, you can get the state of a check box according to the sprite number. Here is a simple behavior to do this:

```
on isChecked me
  return sprite(me.spriteNum).member.hilite
end
```

You can check the state of any check box by asking "isChecked(sprite *X*)", where *X* is the number of the sprite. Or you can do the same thing with "sendSprite(sprite *X*, #isChecked)". A value of *TRUE* indicates that the check box has a mark in it, and a value of *FALSE* indicates that the box is empty.

Because the built-in check boxes are rather limited in appearance, it is usually a good idea to create your own multistate buttons with bitmaps. Figure 15.3 shows three such buttons, with the middle one being checked.

Each sprite can contain one of two members: the on state and the off state. So, Figure 15.3 has a total of six members. In this example, the on states look the same as the off states, except for the addition of a check mark next to them.

Nintendo Exhibit 1029
Page 00200

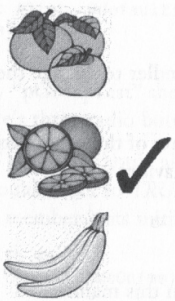


Figure 15.3
Three sprites are used to hold bitmap representations of check boxes. Behaviors control these sprites.

A behavior to handle these states needs to know what the on and off state members are. It also needs to know whether the sprite should start out in the on or off state. Three properties and parameters take care of this, as follows:

```
property pOnMember, pOffMember, pState

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pOnMember, [#comment: "On Member",\
    #format: #member, #default: ""]
  addProp list, #pOffMember, [#comment: "Off Member",\
    #format: #member, #default: ""]
  addProp list, #pState, [#comment: "Initial State",\
    #format: #boolean, #default: FALSE]
  return list
end
```

When the sprite begins, it should adjust itself to be the proper state, regardless of what is in the Score. This can be done by the *on beginSprite* handler. Because you need to reuse the code that changes the member of the sprite according to the "pState" property, it is a good idea to create a custom handler to do that. The *on beginSprite* handler needs to call that custom handler.

```
on beginSprite me
  setMember(me)
end

on setMember me
  if pState = TRUE then
    sprite(me.spriteNum).member = pOnMember
  else
    sprite(me.spriteNum).member = pOffMember
  end if
end
```

The sprite needs to change state when users click it. If it's on, it needs to go off, and vice versa. This can be done with one line using the *not* operator.

```
on mouseUp me
  pState = not pState
```

Nintendo Exhibit 1029
Page 00201

```
  setMember(me)
end
```

The *on mouseUp* handler calls the "on setMember" handler to change the member after the state changes.

One final handler can be a function that returns the state of the check box. This can be used by other Lingo handlers in movie scripts or other behaviors.

```
on isChecked me
  return pState
end
```

Note that you can also ask for this property directly in this manner:

```
sprite(X).pState
```

→ For more information about using check boxes, **see** "Creating Questionnaires," **p. 560**

USING RADIO BUTTONS

The topic of check boxes naturally leads into the topic of radio buttons. They are similar to each other. Whereas check boxes enable users to select one or more items from a list, radio buttons enable users to select only one item from a list.

Director also has a built-in radio button member in the form of a button member variant. It works the same way as the check box in that it already accepts mouse clicks and turns itself on and off accordingly. Figure 15.4 shows a small group of radio buttons.



Figure 15.4
A group of three radio buttons.

Whereas check boxes don't really need any Lingo to work, radio buttons definitely require some. The reason is that at this point the radio buttons don't know of one another's existence. In the example in Figure 15.4, the "Apples" button and the "Oranges" button do not interact in any way. This means that the users can select both buttons, which goes against the whole point of radio buttons. The users should be able to select only one of the buttons at a time.

A simple behavior works to restrict the three buttons and get them to work together. To start, the behavior needs to know what other sprites are in its group. It also needs to know whether it begins as selected, which is the equivalent of *TRUE*.

```
property pState, pGroupList

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pState, [#comment: "Initial State",\
    #format: #boolean, #default: FALSE]
  addProp list, #pGroupList, [#comment: "Group List",\
```

Nintendo Exhibit 1029
Page 00202

```
    #format: #list, #default: []]
  return list
end
```

The property "pGroupList" should contain a linear list of sprites in the group. For instance, if the example's three radio buttons are in sprites 1 to 3, "pGroupList" should be [1,2,3].

The "pState" property should also be set with care. One and only one of the radio buttons in the group should be set to *TRUE*. Then, in the *on beginSprite* handler, the sprite needs to be set to the on member if its initial "pState" is set to *TRUE*.

```
on beginSprite me
  if pState then turnMeOn(me)
end
```

The custom handler "on turnMeOn" is used to set the member of the sprite. Because these are radio buttons, this handler also has to make sure that the other sprites in this group are turned off.

```
on turnMeOn me
  pState = TRUE
  sprite(me.spriteNum).member.hilite = TRUE
  repeat with i in pGroupList
    if i <> me.spriteNum then
      sendSprite(sprite i,#turnMeOff)
    end if
  end repeat
end
```

The *repeat* command in this handler uses the form *repeat with i in*. This special form of the repeat command can be used only with lists. Instead of the variable "i" counting from one number to the next, it moves through the values of the list. If the list is [5,8,14], the loop runs three times and the value of "i" is set to 5, 8, and 14 for those times through the loop.

The *sendSprite* command is used to send the message "#turnMeOff" to each of the sprites in the list. An *if* statement makes sure that this message isn't sent back to the current sprite. The "on turnMeOff" handler is a simple one:

```
on turnMeOff me
  pState = FALSE
  sprite(me.spriteNum).member.hilite = FALSE
end
```

Now that there is a handler to turn on the current radio button, a mouse click is easily handled. It just calls the "on turnMeOn" handler, which turns on the current sprite and turns off all the others.

```
on mouseUp me
  turnMeOn(me)
end
```

One last handler can be used to determine which sprite in the group is currently selected. This handler uses the same repeat loop as the "on turnMeOn" handler, but rather than change the sprite, it just finds one that is on and returns that value.

Nintendo Exhibit 1029
Page 00203

```
on selected me
  repeat with i in pGroupList
    if sprite(i).pState = TRUE then return i
  end repeat
end
```

This handler relies on the fact that one and only one sprite is turned on. There should never be a time when none, or more than one, are on. The handler also has the unusual capability to return the same answer no matter which sprite in the group is used to call it. If a radio button group is in sprites one to three, a "sendSprite(1,#selected)" and a "sendSprite(2,#selected)" should return the same answer because they should have the same "pGroupList" property value.

Although this behavior takes care of the complexity of the radio button group, it does not offer the opportunity to use custom bitmaps as radio buttons rather than the boring built-in radio button member. However, it can be easily modified to do so.

The first step is to add two new properties to represent the on and off state bitmaps. These look the same as they do in the check box behavior:

```
property pOnMember, pOffMember, pState, pGroupList

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pOnMember, [#comment: "On Member",\
    #format: #member, #default: ""]
  addProp list, #pOffMember, [#comment: "Off Member",\
    #format: #member, #default: ""]
  addProp list, #pState, [#comment: "Initial State",\
    #format: #boolean, #default: FALSE]
  addProp list, #pGroupList, [#comment: "Group List",\
    #format: #list, #default: []]
  return list
end
```

Then, the "on turnMeOn" and "on turnMeOff" handlers need to be modified to set the member of the sprite, rather than the *bilite* property of the member:

```
on turnMeOn me
  pState = TRUE
  sprite(me.spriteNum).member = pOnMember
  repeat with i in pGroupList
    if i <> me.spriteNum then
      sendSprite(sprite i,#turnMeOff)
    end if
  end repeat
end

on turnMeOff me
  pState = FALSE
  sprite(me.spriteNum).member = pOffMember
end
```

Nintendo Exhibit 1029
Page 00204

The rest of the behavior can stay the same. Figure 15.5 shows what this screen might look like. The off members and the on member differ only in that the on member includes an arrow to the right of the picture to signify that it is selected.

→ For more examples of using radio buttons, **see** "Creating Standardized Tests," **p. 542**

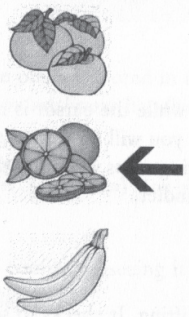


Figure 15.5
A set of custom-built radio buttons. A behavior controls which member is used depending on which item is selected.

DRAGGING SPRITES

Although dragging sprites is possible without Lingo (by setting the *movable* property of a sprite in the Score), Lingo enables you to add all sorts of functionality to dragging. Learning how to drag a sprite with Lingo is the first step, and then you will be ready to try some dragging applications.

Dragging is a very useful interface action that all computer users are familiar with. The operating systems on modern computers require you to drag and drop files and folders all the time. Using them in Director movies is a good way to allow the users to manipulate items on the screen.

SIMPLE DRAG SCRIPT

A drag script can be as simple as setting the sprite to constantly follow the cursor. However, a real drag behavior waits until the user clicks the sprite, and then follows the cursor around until the user releases the mouse button.

To do this type of drag, you don't even need any parameters. However, you do need one property to tell the behavior when the dragging is taking place. This should be set to *FALSE* when the sprite begins.

```
property pPressed

on beginSprite me
  pPressed = FALSE
end
```

Nintendo Exhibit 1029
Page 00205

When the user clicks the sprite, the behavior needs to change this "pPressed" property to *TRUE*. When the user lifts up, it needs to be set to *FALSE*.

```
on mouseDown me
  pPressed = TRUE
end

on mouseUp me
  pPressed = FALSE
end
```

It's possible for the user to lift up the mouse button while the cursor is not over the sprite. The mouse location is updated in real time, whereas you will be setting the location of the sprite only every frame loop. To make sure that dragging stops whenever the mouse button is lifted, you need to also use the *on mouseUpOutside* handler.

```
on mouseUpOutside me
  pPressed = FALSE
end
```

Finally, the *on exitFrame* handler does all the heavy lifting. It checks to see whether the "pPressed" property is *TRUE*, and moves the sprite to the mouse location if it is

```
on exitFrame me
  if pPressed then
    sprite(me.spriteNum).loc = the mouseLoc
  end if
end
```

A BETTER DRAG BEHAVIOR

The main problem with the simple drag script is a cosmetic one. The sprite appears to snap to center itself on the cursor no matter where users click. So, if you click the upper-right side of the sprite, the sprite immediately shifts so that the center, or actually the registration point, is directly under the cursor.

Fixing this glitch is not a problem. When the initial click is made, the difference in location between the mouse and the center of the sprite can be recorded. This value can then be applied to every change in the sprite's location. The result is that the cursor and the sprite remain synchronized, no matter where the user grabs the sprite. If the user grabs the sprite by the upper-right corner, it then drags by the upper-right corner.

To make this change, first add the property "pClickDiff" to the property declaration. Then, alter the *on mouseDown* handler to record the difference between the click location and the sprite location, as follows:

```
on mouseDown me
  pPressed = TRUE
  pClickDiff = sprite(me.spriteNum).loc - the clickLoc
end
```

Nintendo Exhibit 1029
Page 00206

Note

The property *the clickLoc* is similar to *the mouseLoc*. They both return a point as their value. However, *the mouseLoc* might change if the mouse moves between the time the click was made and the time the line of Lingo code runs. *the clickLoc* gives the exact location of the last click in an *on mouseDown* or similar handler, whereas *the mouseLoc* is the current mouse position.

Now that the offset is stored in the "pClickDiff", it can be applied to the position of the sprite in the *on exitFrame* handler.

```
on exitFrame me
  if pPressed then
    sprite(me.spriteNum).loc = the mouseLoc + pClickDiff
  end if
end
```

If this is still confusing, seeing it in action will help; check the example on the CD-ROM called 15dragging.dir.

CLICK, DRAG, AND LOCK

You can apply what you've learned about dragging by using it to build a matching game or quiz. You have elements of one type on one side of the screen and elements of another type on another side. The game is essentially made up of matching pairs. It's up to the user to drag the sprites on the left over to the sprites on the right. Figure 15.6 shows what this might look like.

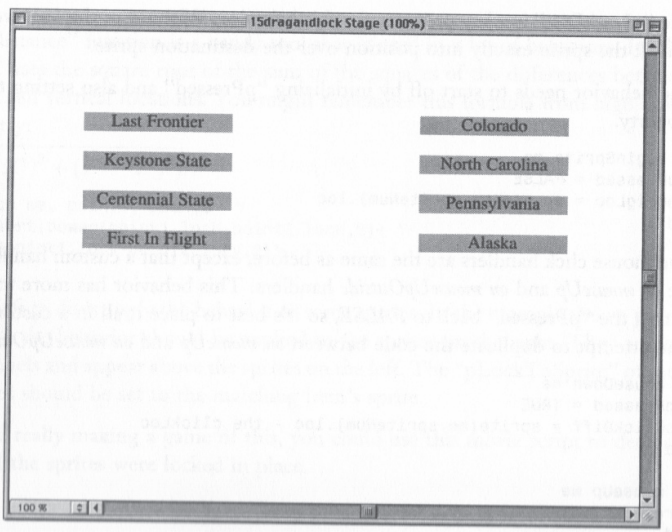


Figure 15.6 The screen shows four pairs of matching items. The user's task is to drag the items on the right onto the correct ones on the left.

Nintendo Exhibit 1029
Page 00207

Although this can be done with the drag scripts already described, these scripts have no capability to tell the user whether the match is correct. A better way to do it would be to have a script that locks a sprite into position if it is near the place it belongs.

This requires some parameters. The first is the number of the sprite to which that behavior's sprite should lock. The second should be the maximum distance that the dragged sprite can be from the destination sprite before it automatically locks.

Also, an option is needed to tell the behavior what to do in case the sprite is not close enough to its destination to lock. One option is for it to snap back to its original position. The other option is to leave it where it is.

In addition to these new properties, you still need the old properties of "pPressed" and "pClickDiff". You also need a property to store the original location of the sprite in cases in which it needs to snap back.

```
property pPressed, pClickDiff, pLockToSprite, pOrigLoc, pLockDist, pSnapBack

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pLockToSprite, [#comment: "Lock To Sprite",\
    #format: #integer, #default: 0]
  addProp list, #pLockDist, [#comment: "Maximum Lock Distance",\
    #format: #integer, #default: 25]
  addProp list, #pSnapBack, [#comment: "Snap Back If Not Locked",\
    #format: #boolean, #default: TRUE]
  return list
end
```

The "pLockDist" is a key property for this behavior. Without it, the user would be required to lock the sprite exactly into position over the destination sprite.

The behavior needs to start off by initializing "pPressed" and also setting the "pOrigLoc" property.

```
on beginSprite me
  pPressed = FALSE
  pOrigLoc = sprite(me.spriteNum).loc
end
```

The mouse click handlers are the same as before, except that a custom handler is called from the *on mouseUp* and *on mouseUpOutside* handlers. This behavior has more to do than just setting the "pPressed" back to *FALSE*, so it's best to place it all in a custom handler rather than attempt to duplicate the code between *on mouseUp* and *on mouseUpOutside*.

```
on mouseDown me
  pPressed = TRUE
  pClickDiff = sprite(me.spriteNum).loc - the clickLoc
end

on mouseUp me
  release(me)
end
```

Nintendo Exhibit 1029
Page 00208

```
on mouseUpOutside me
  release(me)
end
```

The *on exitFrame* script is exactly as before.

```
on exitFrame me
  if pPressed then
    sprite(me.spriteNum).loc = the mouseLoc + pClickDiff
  end if
end
```

The custom "on release" handler has to do several things. First, it sets the "pPressed" to *FALSE*. Then, it needs to determine whether the sprite is now close enough to its destination to lock into place. It calls yet another custom handler, "on distance", to do this. This function takes two points as parameters and returns the distance between them in pixels.

If the sprite is close enough, the new location of the sprite is set to exactly the location of the destination sprite. Otherwise, if the "pSnapBack" property is *TRUE*, the location snaps back to the original location. If not, the location remains where the user left it.

```
on release me
  pPressed = FALSE

  if distance(me,sprite(me.spriteNum).loc,sprite(pLockToSprite).loc) \
      < pLockDist then
    sprite(me.spriteNum).loc = sprite(pLockToSprite).loc
  else if pSnapBack then
    sprite(me.spriteNum).loc = pOrigLoc
  end if
end
```

The "on distance" function is a handy tool that is used in a lot of behaviors. It takes two points and uses the square root of the sum of the squares of the differences between horizontal and vertical locations. You might remember this formula from high school trigonometry:

$$\sqrt{((X^2 - X^1)^2 + (Y^2 - Y^1)^2)}$$

```
on distance me, point1, point2
  return sqrt(power(point1.locH-point2.locH,2)+ \
    power(point1.locV-point2.locV,2))
end
```

All that is left to do is apply the behavior to some sprites. In the example shown previously in Figure 15.6, the behavior should be applied to the sprites on the right. They are in higher Sprite channels and appear above the sprites on the left. The "pLockToSprite" of each one of these sprites should be set to the matching item's sprite.

If you were really making a game of this, you could use this movie script to determine whether all the sprites were locked in place.

Nintendo Exhibit 1029
Page 00209

```
on checkGameDone
  done = TRUE
  repeat with i = 5 to 8
    if (sprite i).loc <> sprite(sprite(i).pLockToSprite).loc then
      done = FALSE
    end if
  end repeat
  if done then beep()
end
```

This movie handler assumes the game is done by setting the local variable "done" to *TRUE*. It then looks at sprites 5 through 8, which is where the dragging sprites are located, and compares their locations to the locations of their designated destination sprites. If they all match, "done" is never set to *FALSE*. In this case, the result is a simple beep. However, you could also make it jump to another frame or play a sound.

The perfect place for a call to this handler is in the "on release" handler of the behavior. The movie needs to check this every time a sprite is locked, so place it after the "sprite(me.spriteNum).loc = sprite(pLockToSprite).loc" line.

→ For an example of click, drag, and lock behavior, **see** "Creating a Matching Game," **p. 526**

DRAG AND THROW

Another way to apply a drag behavior is to enable users to grab and throw a sprite. They click it and drag it around while the mouse is down, like the other drag behaviors, but then when they release it, the sprite keeps going with some amount of momentum.

This is much more complex than a simple drag behavior. First, you have more than just pressed and nonpressed states. A third state is the state of being thrown. So, the "pPressed" property should be replaced with a "pMode" property that is #normal, #pressed, or #throw.

In addition, you need to know how far and in which direction to throw the sprite when it is released. Measuring the distance between the position of the sprite upon release and the position of the sprite just before release provides the appropriate information. However, don't take the position of the sprite exactly one frame prior to release, because that can be too short a period of time: It is only one sixtieth of a second if the frame rate is 60fps. Instead, have a parameter that determines how many frames back to look to determine the momentum of the released sprite. A good default for this is five. So, if a sprite is clicked and dragged for 150 frame loops and then released, the position in frame 145 and the position in frame 150 are compared to set the throwing momentum. A list is needed to record the last five positions at any given time. Here is the property list *and on getPropertyDescriptionList* for this behavior:

```
property pThrowSpan, pMode, pCurrentLoc, pLocList, pMoveAmount

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pThrowSpan, [#comment: "Frame Span of Throw",\
    #format: #integer, #range: [#min: 1, #max: 20], \
    #default: 5]
  return list
end
```

Nintendo Exhibit 1029
Page 00210

The sprite starts by setting the "pMode" to #normal.

```
on beginSprite me
  pMode = #normal
end
```

When the user clicks, the mode must change to #pressed and the list used to store the positions must be initialized:

```
on mouseDown me
  pMode = #pressed
  pLocList = []
end
```

Whenever the mouse button lifts, a custom handler called "on throw" runs. This handler calculates the momentum by taking the current mouse location and the first item in the "pLocList". It needs to divide that by "pThrowSpan" to get a relative per-frame movement amount. This way, if "pThrowSpan" is set to 5, the "pMoveAmount" is set to the current location of the sprite, minus the location of the sprite 5 frame loops ago, divided by 5:

```
on mouseUp me
  throw(me)
end

on mouseUpOutside me
  throw(me)
end

on throw me
  pMoveAmount = (the mouseLoc - pLocList[1])/pThrowSpan
  pMode = #throw
end
```

The *on exitFrame* handler must move the sprite whether it is being dragged or thrown. If being dragged, it needs to set the location of the sprite to the current mouse location. It also needs to record this location in "pLocList". If "pLocList" has more items than specified by "pThrowSpan", the oldest item is removed.

If the mode is #throw, the "pMoveAmount" is used to move the sprite. In addition, this property is multiplied by .9, thereby decreasing it by approximately 10%. This imitates a sort of friction that one would expect the sprite to exhibit when thrown. If you want, the .9 can be made into a property, say "pFriction", and altered using the behavior parameter dialog box.

```
on exitFrame me
  if pMode = #pressed then
    pCurrentLoc = the mouseLoc
    sprite(me.spriteNum).loc = pCurrentLoc
    pLocList.add(pCurrentLoc)
    if pLocList.count > pThrowSpan then pLocList.deleteAt(1)
```

Nintendo Exhibit 1029
Page 00211

```
  else if pMode = #throw then
    pCurrentLoc = pCurrentLoc + pMoveAmount
    sprite(me.spriteNum).loc = pCurrentLoc
    pMoveAmount = pMoveAmount * .9
  end if
end
```

Think of the possibilities of a behavior like this. Do you want to add code like that used in the bouncing behavior so that the sprite bounces off the sides of the screen? How about code that adds gravity? How about code that locks the sprite to the position of another if it's close enough? Combine all three and you get something that resembles a basketball free-throw video game.

CREATING SLIDERS

A slider is a familiar user interface element used in all major software programs. Sliders offer the best way to enable users to input a number within a small range. They are even used in behavior Parameter dialog boxes for this purpose.

Creating a slider with several bitmap members and a behavior can be fairly complex. A lot of actions must be taken into account, and more than one sprite is needed to form the elements of a slider.

Figure 15.7 shows what a slider should look like. This is just a straight imitation of the sliders used in the Director Preferences dialog boxes.



Figure 15.7
A typical slider enables users to pick a number from a range.

A full slider element contains six parts: a marker, a shadow, a background graphic, two buttons, and a text field. However, you can start by creating the main three parts: the marker, the shadow, and the background graphic. The shadow is the dark coloring to the left side of the marker. It is the only nonbitmap element in this case; it uses a shape sprite instead.

You actually will need only one behavior script, attached to the marker sprite. Like the button and drag behaviors, this behavior needs a property to tell whether it is in the process of being pressed. It also needs to know how far to the left and right it can move, and what range of values it represents, such as 1 to 3, 0 to 100, or -500 to 500. A relationship with the shadow sprite is needed, so one property should hold that sprite number. Another property needs to record the current value of the slider, as follows:

```
property pPressed -- whether the sprite is being pressed
property pBounds -- the rect of the shadow sprite at start
property pMinimumValue, pMaximumValue -- used by the marker sprite only
property pShadowSprite -- the number of the shadow sprite
```

Nintendo Exhibit 1029
Page 00212

```
property pValue -- actual value of the slider
on getPropertyDescriptionList me
  list = [:]
  addProp list, #pShadowSprite, [#comment: "Shadow Sprite",\
    #format: #integer, #default: 0]
addProp list, #pMinimumValue, [#comment: "Minimum Value",\
    #format: #integer, #default: 0]
  addProp list, #pMaximumValue, [#comment: "Maximum Value",\
    #format: #integer, #default: 100]
  addProp list, #pValue, [#comment: "Start Value",\
    #format: #integer, #default: 50]
  return list
end
```

The *on getPropertyDescriptionList* needs to use only four of the properties as parameters: the minimum, maximum, and starting value of the slider, as well as the shadow sprite's number.

Although the minimum and maximum values are set by the behavior's Parameter dialog box, the behavior also needs to know the physical screen locations of the minimum and maximum values. To determine these locations, a trick is used. The shadow sprite is set to mark the exact bounds of the marker sprite. Because the shadow sprite will be reset by the behavior when it starts anyway, using it to show the boundaries of the slider does not affect its future appearance. In the case of the slider shown previously in Figure 15.7, the shadow sprite is stretched all the way across the inner part of the background graphic. Its *rect* is recorded by the *on beginSprite* handler of the behavior. Then, it is set to display properly, as shown earlier in Figure 15.7:

```
on beginSprite me
  pBounds = sprite(pShadowSprite).rect
  setMarker(me)
  setShadow(me)
end
```

The left and right physical limits of the slider could have been added to the *on getPropertyDescriptionList* handler, but this would mean that you would have had to determine the exact screen locations and type them in. Worse than that, if the slider were moved, even by one pixel, you would have to re-enter these numbers. Using the shadow sprite as a "template" of sorts saves you the trouble, and it is easily adjusted on the Stage.

The *on beginSprite* handler includes calls to the custom "on setMarker" and "on setShadow" handlers. These take the current value of the slider and set the position of these two sprites.

The "on setMarker" handler first figures out the value range of the slider. If the slider goes from 0 to 100, the range is 100 (not 101). It computes the value of the slider as a number between 0 and 1, regardless of the real range. It then takes this percentage and applies it to the physical screen range to get the location of the marker:

```
-- this sets the marker sprite
on setMarker me
  -- compute the value as a number between 0 and 1
  valueRange = pMaximumValue - pMinimumValue
  sliderPos = float(pValue)/float(valueRange)
```

Nintendo Exhibit 1029
Page 00213

```
  -- translate to a screen position
  sliderRange = pBounds.right-pBounds.left
  x = sliderPos*sliderRange + pBounds.left

  -- set marker
  sprite(me.spriteNum).locH = x
end
```

The "on setShadow" handler sets the shadow to its original rectangle, but with the right side adjusted to fall under the marker.

```
-- this handler lets the marker sprite set the shadow sprite
on setShadow me
  x = sprite(me.spriteNum).locH
  r = rect(pBounds.left, pBounds.top, x, pBounds.bottom)
  sprite(pShadowSprite).rect = r
end
```

These handlers accomplish the task of setting the slider to its starting position. Now you need some handlers to enable users to click and drag the marker:

```
on mouseDown me
  pPressed = TRUE
end

on mouseUp me
  pPressed = FALSE
end

on mouseUpOutside me
  pPressed = FALSE
end

on exitFrame me
  if pPressed then
    moveMarker(me)
    setMarker(me)
    setShadow(me)
  end if
end
```

The *on exitFrame* handler checks to make sure that "pPressed" is true, and then calls other handlers to handle the work. The "on setMarker" and "on setShadow" handlers are there, but called after an "on moveMarker" handler. The "on moveMarker" handler does the opposite of what the "on setMarker" handler does: It determines the value of the slider based on the mouse location.

In addition, the "on moveMarker" handler translates the value to an integer. If you want the slider to show floating point numbers instead, just remove that line:

```
-- this handler takes the mouse position and figures the
-- value of the slider
on moveMarker me
  -- compute the position as a number between 0 and 1
  x = the mouseH - pBounds.left
  sliderRange = pBounds.right-pBounds.left
```

Nintendo Exhibit 1029
Page 00214

```
  pos = float(x)/sliderRange

  -- translate to a value
  valueRange = pMaximumValue - pMinimumValue
  pValue = pos*valueRange + pMinimumValue
  pValue = integer(pValue)

  -- check to make sure it is within bounds
  if pValue > pMaximumValue then
    pValue = pMaximumValue
  else if pValue < pMinimumValue then
    pValue = pMinimumValue
  end if
end
```

The "on moveMarker" also makes sure that the new value of the slider falls within its range. Note that this handler's sole purpose is to set the "pValue" property. After it is called in the *on exitFrame* handler, the "on setMarker" and "on setShadow" handlers update the sprites.

Because the value of the slider is held in the "pValue" property, a handler that returns the value of the slider is quite simple. You can call this handler with *sendSprite* from other behaviors or movie scripts to get the current value of the slider, even while it is being dragged.

```
-- this handler returns the value of the slider
on getValue me
  return pValue
end
```

Several elements can be added to a slider to make it as complete as sliders found in other software. For example, a text field can show the current value of the slider and two buttons can enable users to move the slider one value at a time.

Figure 15.8 shows all these elements. There are a total of eight cast members if you include down states for the buttons.

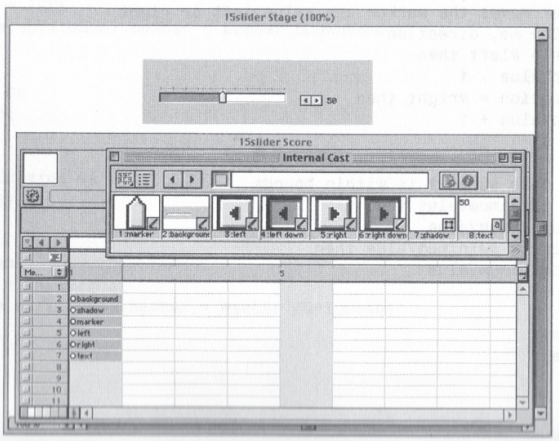


Figure 15.8 The Cast window shows the eight elements of the slider. The Score shows the sprite placement of each. The Stage shows the assembled slider.

Nintendo Exhibit 1029
Page 00215