CHAPTER 23

Movies in a Window

✦ ✦ ✦ ✦

In This Chapter

Understanding MIAWs (movies in a window)

Creating MIAWs

Controlling MIAWs

Extending Director by making MIAW Xtras

✦ ✦ ✦ ✦

Director's movie in a window (MIAW) feature enables you to play two or more movies simultaneously in separate windows. This is especially useful when you have a large movie that has changing elements. Suppose that you are designing an interactive tour of the facilities of a corporate entity that has numerous facilities sites. You can create the host movie, and then make separate movies featuring an individual site, and play each site's movie in a window with the host movie. The modular nature of MIAWs enables you to change elements easily without rewriting the entire movie.

Movies in a window are especially effective for use as dialog boxes, control panels, toolbars, menus, floating palettes, and movie-building utilities.

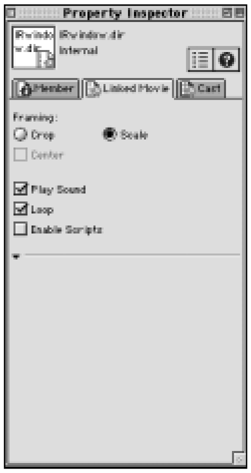
MIAWs and Linked Movies

When you create a movie in a window, you are running two or more Director movies at the same time that can interact. An MIAW is a separate and independent movie that your host movie (otherwise known as the Stage) can open. Because they exist in their own window, you don't add them to the cast of the Stage. MIAWs also have their own casts, Lingo scripts, and attributes, such as Stage size and location.

You can't use a movie in a window in a Shockwave movie. Web browsers don't enable Shockwave to open new windows.

Director imports movies as either film loops or linked movies, and places them in the cast. As a cast member an imported movie behaves like any other graphic cast member and is subject to the control of the Stage and the Score. A linked movie has an advantage over film loops in that its scripts still work. After the member is imported, you must select the Enable

Scripts option to activate that movie's scripts (see Figure 23-1). When you import a movie into another Director movie, like other cast members, it must be positioned within the physical boundaries of the movie.



Framing:
Crop [✓ Scale]
☐ Center

☑ Play Sound
☑ Loop
☐ Enable Scripts

Figure 23-1: The Enable Scripts option is not selected by default.

An advantage of linked movies is that you can use them in Shocked movies, but having a cast member that is actually a Director movie makes for some complex sprite behavior. The linked movie has all of its sprites, and all of those sprites have their behaviors attached. Following is output from a two-frame movie with a sprite that is a linked movie. The sprite's movie loops on the frame and the Stage plays from frame 1 to frame 2.

```
-- "prepareMovie"
-- "beginSprite"
-- "prepareMovie linked"
-- "beginSprite linked"
-- "prepareFrame linked"
-- "startMovie linked"
-- "enterFrame linked"
-- "prepareFrame"
-- "startMovie"
-- "enterFrame"
-- "exitFrame"
-- "exitFrame linked"
```

A linked movie receives events just as a normal Director movie. It advances a frame every time `updateStage` is called or the playback head moves. Sprites within the linked movie receive mouse events, unless the linked movie sprite itself has a behavior attached; in that case, any event the behavior has traps that event with some caveats. If you have a `mouseDown` or `mouseUp` in a behavior attached to a linked movie sprite, it traps both the `mouseDown` and the `mouseUp`; the sprites in the linked movie are not sent either, but they still get `mouseEnter` and `mouseLeave` events. If the behavior attached to the linked movie sprite has a `mouseEnter` or `mouseLeave` handler, then the sprites in the linked movie do not receive either.

The biggest hurdle in dealing with linked movies is communicating with them. There is no easy way to send one a message. For example, Flash movies have a `goToFrame`, `rewind`, `play`, and other commands to control playback; MIAWs can be sent messages with the `tell` statement. Linked movies can share globals variables, which is the best route to communicating. Through some planning, you can have a linked movie continually checking the state of a global (or globals) and respond accordingly.

In contrast, MIAWs don't appear in the Score or Cast window and communication is quite easy. The independent nature of an MIAW offers you several advantages over imported movies. You specify the size, orientation, and location of each MIAW. You aren't constrained by the boundaries of the Stage.

Each MIAW opens in a window as needed and has the life span that you specify using Lingo. When the movie has finished playing, it can either be retained in memory, if you will need it later, or you can close it and remove it from memory. Because you don't need to extend sprites within the Score, you can pace the loading and unloading of data into memory. Using memory only on demand enables you to create larger and more complex movies that operate faster and animate more smoothly.

You can't modify an MIAW when it's opened by another movie. If you need to edit the MIAW, you need to open it separately in Director. In fact, during development of movies with MIAWs, you can reasonably expect to go back and forth between the movies several times.

Creating a Movie with an MIAW

The first step in producing an MIAW is to plan and create the elements for the project. If you have a clear idea of what you want, the creation process should go more smoothly. Create your host movie (Stage) and the individual movies you want to play in a window. You can include any desired Lingo, interactive element, sound, and animation into an MIAW. Palette, transition, and tempo settings are ignored when your movie is played in a window. These elements are under the control of the Stage.

The Stage remaps your MIAW to match its active palette. It's a good idea to create MIAWs using the Stage's palette to ensure that remapping doesn't cause undesirable effects.

In many programming environments, you can create a window solely from code. You cannot create an MIAW from Lingo alone; you always need to start with an existing Director movie.

On the CD-ROM

You can find the movie mars scanner.dir in the EXERCISE:CH23 (EXERCISE\CH23) folder on the companion CD-ROM.

Open the movie mars scanner.dir on the CD-ROM. In the same folder, is a movie called infrared.dir, which appears in a window. You can create an MIAW in one line of code. Open the Message window and type the following:

```
open window "Infrared"
```

That's it! You did it. Now, on to Chapter 24. . . . Well, actually, there's more to it. Although this is a complex subject, it doesn't need to be difficult. You will want to choose where to open the window, what type of window, the title of the window, and many other aspects.

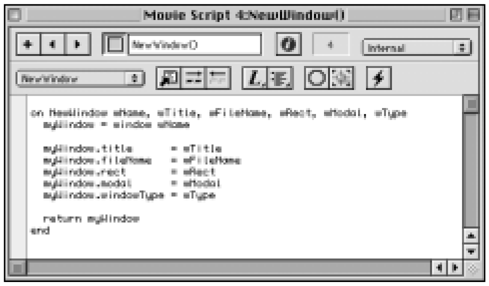
Writing scripts to create an MIAW

There are several window properties to consider when opening a window: `name`, `title`, `fileName`, `rect`, `modal`, and `windowType`.

A simple, general-purpose window-creation function is shown in Figure 23-2. Create a new movie script, as shown in the figure. To use it, you can type (there is no hard return, it is one long statement) the following:

```
myWindow = NewWindow("Infrared", "Infrared Scanner", the
moviePath & "Infrared", rect(20, 50, 340, 290), FALSE, 49)
```

When finished, save this movie as **mars scanner 2.dir**. Copy infrared.dir to the same folder as mars scanner 2.dir.



```
on NewWindow wName, wTitle, wFileName, wRect, wModal, wType
  myWindow = window wName

  myWindow.title      = wTitle
  myWindow.fileName   = wFileName
  myWindow.rect       = wRect
  myWindow.modal      = wModal
  myWindow.windowType = wType

  return myWindow
end
```

Figure 23-2: A simple window-creation function

The open window command

The `NewWindow()` function creates the window reference but it does not open it. The window does not open until you use either the `open` command or set the `visible` to TRUE:

```
open myWindow
```

This enables you to create the window variable but not open it until you need to use it. It may seem like you need to pass a good deal of information to this function when compared with the following statement:

```
open window "Infrared"
```

When you use the `open` command in this manner, on a window you've yet to create, Director assumes that you want the name of the window to be (in this case) Infrared, the title to be Infrared, and the `fileName` to be Infrared. You were directed to open the mars scanner.dir movie, so the default directory would call the one that contained the movie infrared.dir. If you'd just opened a new movie, the default directory would have been that of the Director application, and you would have gotten a dialog box asking you to find Infrared.

Your next question is probably "If Director uses Infrared as the name, how did it know to open infrared.dir?" Good question! When you tell Director to open a file, and you don't specify an extension, Director opens the file whether it has a .DIR, .DXR, or .DCR extension. Because source, protected, and Shockwave movies each have different extensions (.DIR, .DXR, and .DCR respectively), not having to create a protected movie or a Shockwave movie before testing it is convenient. After you do protect or "Shock" your movie, you need not change your code if you did not specify an extension.

The windowList

After you create a window, a reference appears in the `windowList`. The `windowList` is a system property. It is a global list of all the windows that you have created. If you create a window and do not put it into a variable, you can still access it through the `windowList`, as follows:

```
put the windowList
-- [(window "Infrared")]
```

To learn whether a window was already created, you can use the `window Present()` function, as shown here (remember that a window can be created and yet not be visible):

```
put windowPresent("Infrared")
-- 1
```

Window properties

The window properties mentioned earlier — `name`, `title`, `fileName`, `rect`, `modal`, and `windowType` — can all be changed while the MIAW is running. There are three additional properties — `drawRect`, `titleVisible`, and `sourceRect`. The `sourceRect` is the only one you cannot change.

The name of window

The `name of window` property is how you refer to the window within your code and in the `windowList`. You can change the name while the window is opening. Because you reference a window by its name, just like cast members, use caution whenever you change a name. Fortunately, Director is smart enough to change all references to the window as well, but that won't help you if you've hard-coded the wrong name in your program. Try the following in the Message window:

```
put the windowlist
-- [(window "Infrared")]
set myWindow to (the windowList)[1]
put myWindow
-- (window "Infrared")
set the name of window "Infrared" to "Bill"
put the windowList
-- [(window "Bill")]
put myWindow
-- (window "Bill")
```

The title of window

The `title of window` is what appears in the title bar of the window. You can change the title as often as you like:

```
open window "Infrared"
set the title of window "Infrared" to "Mars Scanner"
```

Changing the title of the window does not affect the `name` or the `fileName` of the window.

The fileName of window

The `fileName of window` is the path to the Director movie that is playing in the window. Think of the window as something separate from the movie that plays in it. You can specify any movie to play in a window, but you need a Director document to create it initially. After that, you can change the `fileName` property at will. Doing so changes the movie playing in the window. The `name` and `title` remain the same, but the contents are that of the new movie.

The rect of window

The `rect` of a window property has the coordinates of where the window is on the screen. You can get the screen size from the `desktopRectList`. This property is a list of rects, because you might have more than one monitor attached to your computer:

```
put the desktopRectList
-- [rect(0, 0, 1024, 768)]
```

The `rect` of a window is relative to the screen's coordinates. When you had the Infrared window appear at `rect(20, 50, 340, 290)`, it appeared 20 pixels from the left and 50 pixels from the top.

You can set the rect of the window to make it larger or smaller than the movie it contains. You can also use this property to move the window or to place it in a specific spot on the screen, or to make sure it is on the screen.

The drawRect of window

The drawRect is the rect of the movie within the window. Its coordinates are relative to the window, not the screen. The drawRect does not have to be the same size as the window. The infrared.dir movie is 320×240, as is the window you opened earlier. If you changed the drawRect to the following:

```
set the drawRect of myWindow to rect(40, 40, 280, 200)
-- or dot notation
myWindow.drawRect = rect(40, 40, 280, 200)
```

you would get the result shown in Figure 23-3.

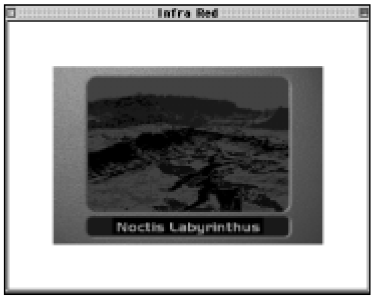


Figure 23-3: In this movie, the drawRect is smaller and not proportional to the source movie.

You can also scale the drawRect much larger than the original size; first restore it, and then increase the magnification to six times, as follows:

```
set the drawRect of myWindow to rect(0, 0, 320, 240)
set the drawRect of myWindow to the drawRect of myWindow * 6

-- or dot notation
myWindow.drawRect = rect(0, 0, 320, 240)
myWindow.drawRect = myWindow.drawRect * 6
```

Your window should now look like Figure 23-4.

The modal of window

The modal property is a Boolean value (1 or 0). It indicates whether the window prevents other movies from responding to events while it is open. A dialog box is a good example of a modal window: You can't do anything else until you click OK or Cancel.

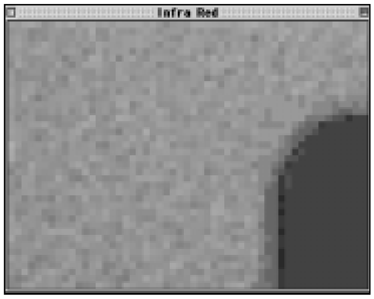


Figure 23-4: In this movie, the drawRect is six times its original size.

The windowType of window

The `windowType` is the style of window. For the most part, it does not affect functionality (unless it is missing a close box). You can change the `windowType` on the fly:

```
set the windowType of window "Infrared" to 4

-- or dot notation
window("Infrared").windowType = 4
```

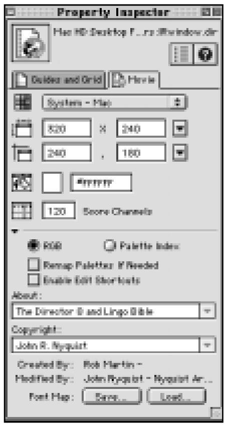
Table 23-1 lists the different possibilities. Although style 1 is for modal dialog boxes, the window itself is only modal if you set the `modal` property to TRUE (or 1).

Table 23-1
Window Types

Value	*Description*
0	Moveable, sizable window without zoom box
1	Alert box or modal dialog box
2	Plain box; no title bar
3	Plain box with shadow; no title bar
4	Moveable window without size box or zoom box
5	Moveable modal dialog box
8	Standard document window
12	Zoomable, nonresizeable window
16	Rounded-corner window
49	Floating palette during authoring (in Macintosh projectors, the value 49 specifies a stationary window)

The sourceRect of window

The `sourceRect` of a window is the position specified for the movie when you choose Modify ➪ Movie and then click the Movie tab (see Figure 23-5). You cannot set this value from Lingo — it is read-only. If you want to start the movie off screen, set the Stage Location to a very large number or a very small number (such as –1000). You can also set the `rect` to the size that you want the window to be initially. Perhaps you've created a tool that changes sizes based on choices the user makes. Similar to the way the Sprite Inspector has three different sizes, you can make your movie change its orientation.



Property Inspector
Mac HD:Desktop F...rs:Window.dir
Guides and Grid | Movie
System - Mac
320 x 240
240 , 180
#FFFFFF
120 Score Channels
[● RGB] ○ Palette Index
☐ Remap Palettes If Needed
☐ Enable Edit Shortcuts
About: The Director 8 and Lingo Bible
Copyright: John R. Nyquist
Created By: Rob Martin -
Modified By: John Nyquist - Nyquist Ar...
Font Map: Save... Load...

Figure 23-5: The Movie Properties tab of the Property Inspector

Controlling MIAWs

An essential part of creating a project that contains a movie in a window is creating Lingo scripts that pass information and instructions between the movies. For example, your MIAW can instruct the host movie to play another movie, go to a marker in a movie, or move to the front or back. You use the Lingo `tell` statement to send instructions between movies. You can place a `tell` statement in a cast member, frame, or sprite script.

The tell statement

If you don't have it open, open the mars scanner 2.dir. Make sure infrared.dir is in the same folder. Then open the MIAW Infrared by typing the following in the Message window (there is no hard return, it is one long statement):

```
myWindow = NewWindow("Infrared", "Infrared Scanner", the
moviePath & "Infrared", rect(20, 50, 340, 290), FALSE, 49)
```

After the MIAW is open, type the following in the Message window:

```
tell myWindow to go to marker(1)
```

The MIAW goes to the next marker (which happens to be in frame 2). You can confirm this by typing:

```
tell myWindow to put the frame
-- 2
```

The movie infrared.dir has a `go to the frame` script for each of the frames. Any Lingo statement can appear after the `tell...to` command. Using a variable containing a reference to a window, as with `myWindow`, is convenient because you don't have to remember to use the window keyword, as in the following:

```
tell window "Infrared" to put the lastFrame
-- 5
```

This isn't a one-way conversation. An MIAW can also tell the Stage what to do, as follows:

```
tell the Stage to put the movieName
-- "Mars Scanner.dir"
```

You can even tell a window to tell the Stage something:

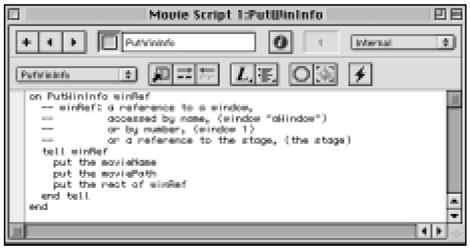
```
tell myWindow to tell the Stage to put the time
-- "6:08 PM"
```

This way, you can have one window talking to another, or talking to the Stage, or both.

Multiline tell statements

It is very useful to be able to send commands to a window by using `tell` statements. Multiline `tell` statements can be even more useful. A multiline `tell` forgoes the `to` and ends with `end tell`.

Create a new movie script, as shown in Figure 23-6. This handler has one parameter that is a reference to a window or the Stage.



```
on PutWinInfo winRef
  -- winRef: a reference to a window,
  --        accessed by name, (window "aWindow")
  --        or by number, (window 1)
  --        or a reference to the stage, (the stage)
  tell winRef
    put the movieName
    put the moviePath
    put the rect of winRef
  end tell
end
```

Figure 23-6: A multiline tell statement

Using the PutWinInfo handler in the Message window should produce results similar to the following:

```
PutWinInfo the stage
-- "Mars Scanner.dir"
-- "Mac HD:Desktop Folder:D8 Bible:Ch23:"
-- rect(192, 144, 832, 624)
PutWinInfo myWindow
-- "Infrared.dir"
-- "Mac HD:Desktop Folder:D8 Bible:Ch23:"
-- rect(20, 50, 340, 290)
```

Setting the visibility of a window

Setting the visibility is the same as opening or closing the window. Following are equivalent statements to open a window:

```
set the visibility of myWindow to TRUE
open myWindow

-- or dot notation
myWindow.visibility = TRUE
myWindow.open()
```

To close a window, you can use either of these:

```
set the visibility of myWindow to FALSE
close myWindow

-- or dot notation
myWindow.visibility = FALSE
myWindow.close()
```

Moving an MIAW

You can place an MIAW anywhere on the screen by setting its rect property. Figure 23-7 shows the moveme.dir. In the Message window, type this code:

```
open window "moveMe"
```

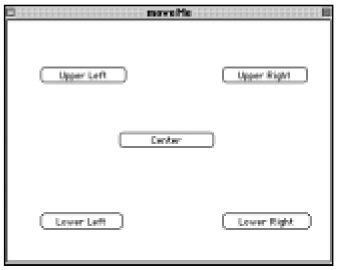


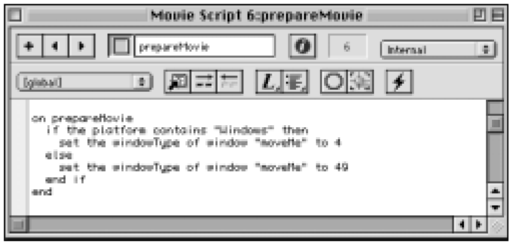
Figure 23-7: The moveme.dir movie

When it opens, you'll notice it change immediately to window type 4 for Windows machines and window type 49 on the Mac (see Figure 23-8). You can prevent the user from seeing this by opening it off screen and then moving it on screen:

```
myWindow = window "moveMe"
winRect = myWindow.rect
myWindow.rect = myWindow.rect – 1000
open myWindow
myWindow.rect = winRect
```

Another way to do the same thing is to have the original Stage coordinates set to a value off screen by choosing Modify ➪ Movie ➪ Properties. If you are creating MIAW Xtras, you can do this and then, in the `prepareMovie` script, place the window on screen, which prevents the unappealing sudden change of window types when the movie opens.

After it's open, click the different buttons. Note that the window jumps to the appropriate spots on the Stage.

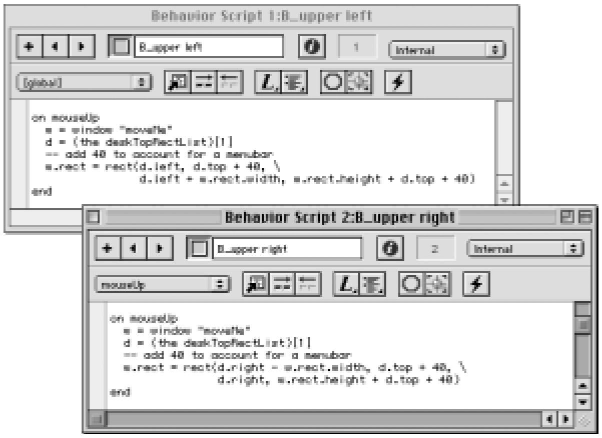


```
on prepareMovie
  if the platform contains "Windows" then
    set the windowType of window "moveMe" to 4
  else
    set the windowType of window "moveMe" to 49
  end if
end
```

Figure 23-8: Changing the windowType based on the platform

To move the window to the upper-left and the upper-right, you need to consider whether there is a menu bar. If there is, you should add the height of the menu bar plus a little extra. The behaviors for the upper-left and upper-right buttons are shown in Figure 23-9.

The behavior for the lower-left and lower-right buttons puts the window flush with the bottom of the screen, as shown in Figure 23-10. All four use the same temporary variables: *w* for the window reference and *d* for the first `rect` in `the desktop RectList`. Given the brevity of the code and the lengthy assignment statement for the window's `rect`, short variable names seemed acceptable for illustration purposes. The `desktopRectList` has a list of all the monitor's coordinates; this code assumes that you are using the first `rect`. The `rect` of the window is set relative to these coordinates. Thanks to the dot notation introduced with version 7 of Director, the code is much more concise.

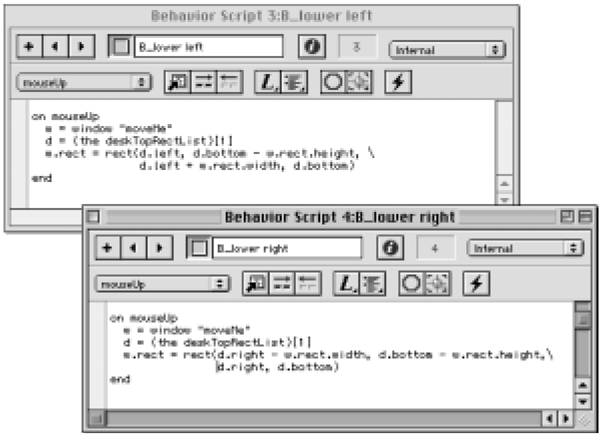

Behavior Script 1:B_upper left
B_upper left
1
Internal
[global]

```
on mouseUp
  w = window "moveMe"
  d = (the deskTopRectList)[1]
  -- add 40 to account for a menubar
  w.rect = rect(d.left, d.top + 40, \
             d.left + w.rect.width, w.rect.height + d.top + 40)
end
```

Behavior Script 2:B_upper right
B_upper right
2
Internal
mouseUp

```
on mouseUp
  w = window "moveMe"
  d = (the deskTopRectList)[1]
  -- add 40 to account for a menubar
  w.rect = rect(d.right - w.rect.width, d.top + 40, \
             d.right, w.rect.height + d.top + 40)
end
```

Figure 23-9: Accounting for a menu bar when moving t
he window to the upper positions


Behavior Script 3:B_lower left
B_lower left
3
Internal
mouseUp

```
on mouseUp
  w = window "moveMe"
  d = (the deskTopRectList)[1]
  w.rect = rect(d.left, d.bottom - w.rect.height, \
             d.left + w.rect.width, d.bottom)
end
```

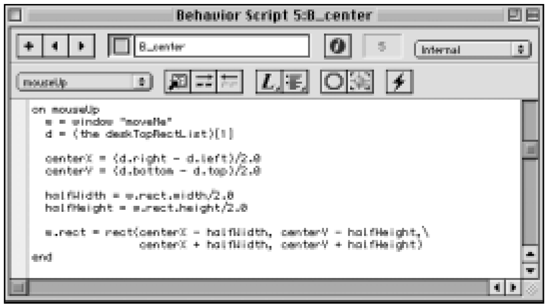
Behavior Script 4:B_lower right
B_lower right
4
Internal
mouseUp

```
on mouseUp
  w = window "moveMe"
  d = (the deskTopRectList)[1]
  w.rect = rect(d.right - w.rect.width, d.bottom - w.rect.height,\
             d.right, d.bottom)
end
```

Figure 23-10: The behaviors for moving the window to
the lower positions

The last button to consider is the Center button. The code for its behavior is slightly more involved, because it determines the center of the screen and then the centering of the window. The behavior is shown in Figure 23-11.

Instead of moving the MIAW relative to the `rect` from the `desktopRectList`, any `rect` could be substituted. The `rect` of the Stage would be one possibility, as would another window, or even a sprite on the Stage.



```
on mouseUp
  w = window "moveMe"
  d = (the deskTopRectList)[1]

  centerX = (d.right - d.left)/2.0
  centerY = (d.bottom - d.top)/2.0

  halfWidth = w.rect.width/2.0
  halfHeight = w.rect.height/2.0

  w.rect = rect(centerX - halfWidth, centerY - halfHeight,\
                centerX + halfWidth, centerY + halfHeight)
end
```

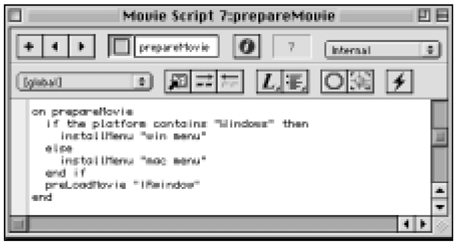
Figure 23-11: The behavior to center the window

Preloading MIAWs

By default, an MIAW isn't loaded until the playback head reaches the specified starting point in the Stage. This can be problematic if you have a large MIAW, or an MIAW that contains digital video or sound. It's a good idea to assign a script to a frame in the host movie that preloads the MIAW.

Preloading an MIAW

1. Open your host movie in Director if it's not already open.
2. Open the movie script member `prepareMovie`.
3. In the Script window, add the statement shown in Figure 23-12.



```
on prepareMovie
  if the platform contains "Windows" then
    installMenu "win menu"
  else
    installMenu "mac menu"
  end if
  preLoadMovie "iReminder"
end
```

Figure 23-12: Using preLoadMovie in the host movie

4. Close the Script window.
5. Rewind your movie. Click the Play button to test the preload.

You can place the preload script in any frame before the start of your MIAW. To keep memory usage to a minimum, keep the script as close to the start of the MIAW as possible while still ensuring a smooth start of your MIAW.

Moving windows to the front or back

Moving a window to the front or to the back is just a matter of using the `moveToFront` or `moveToBack` command. Assume `myWindow` holds a reference to a window:

```
moveToFront myWindow
moveToBack myWindow

-- or dot notation
myWindow.moveToFront()
myWindow.moveToBack()
```

There is no property to tell you where the window is in the order of things, such as `myWindow` is second from the front.

You can determine the frontmost window. Using the `frontWindow` command returns a reference to the window that has the focus of the application. If the front window is not a Director window, `VOID` is returned. Otherwise, a reference to the window or the Stage is returned.

Removing an MIAW

Clicking the close box of a window does not remove it from memory. Closing a window simply makes it invisible. To dispose of the window from memory, all references to it must be eliminated. Variables holding the reference should be set to VOID or 0. References within a list should be deleted.

Using the `forget` command is useful for removing references to windows. If you have a window in the global variable `myWindow`, and the same reference in another variable, calling `forget` sets `myWindow` to 0, the other variable to `<Void>`, and removes the reference in the `windowList`. Look at the following statements in the Message window:

```
set myWindow to (the windowList)[1]
put myWindow
-- (window "Window 1")
set otherWin to myWindow
put otherWin
-- (window "Window 1")
forget myWindow
put myWindow
-- 0
put otherWin
-- <Void>
put the windowList
-- []
```

It would seem that forget works pretty well. It is odd that it sets one variable to 0 and the rest to void, but it does achieve the desired results. It seems that way, but it is better not to rely on it. You should always make sure you remove the references. Look what happens in the following case:

```
set myWindow to (the windowList)[1]
set myWinList to [myWindow]
put myWinList
-- [(window "Window 1")]
forget myWindow
put the windowList
-- []
put myWinList
-- [(window "Window 1")]
```

The forget command does not remove the reference that exists in the list held by myWinList. This is also true when the reference is in the property of a child object or parent script:

```
set myWindow to (the windowList)[1]
set the pWin of script "winObj" to myWindow
forget myWindow
put the pWin of script "winObj"
-- (window "Window 1")
```

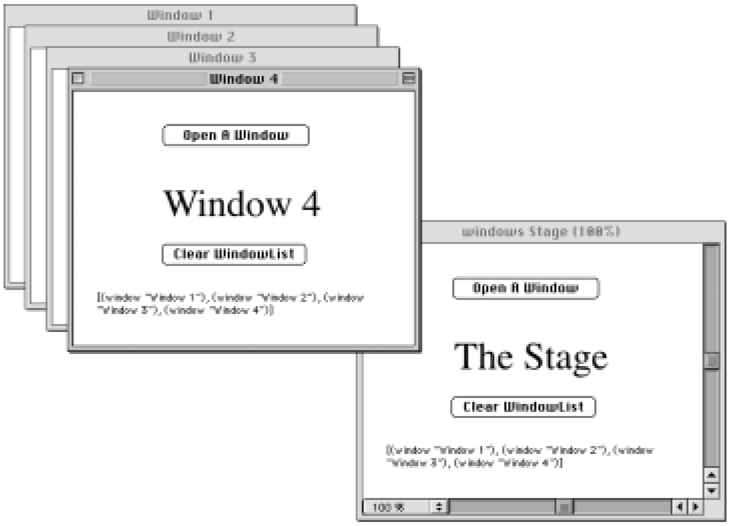
Note

In the preceding example, the property of a parent script is set to the window reference. We are not setting the property of a child object of script winObj, but the property of the parent script itself (see Chapter 13 for more information on this subject). The forget command does not affect properties of a parent script or child objects of a parent script, and this holds true for behaviors as well. In author mode, the parent script property retains its value even after the movie is stopped; to clear it out, you have to recompile the scripts.

Creating multiple MIAWs

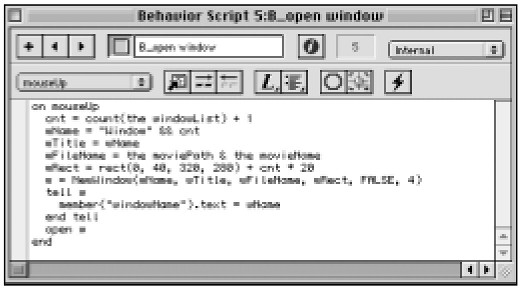
You can open more than one MIAW at a time. Two MIAWs can reference the same Director file. The windows created in Figure 23-13 are all created from the same movie.

Figure 23-14 shows the script for the Open a Window button. Each time it is clicked, it calls the NewWindow() function. This NewWindow() is the same one that you created earlier. The script passes in the path and name of the current movie and uses it to create a new MIAW. It then tells the new window to set the name of the text member windowName to wName.



```
[(window "Window 1"), (window "Window 2"), (window
"Window 3"), (window "Window 4")]
```

Figure 23-13: Multiple MIAWs made from the Stage



```
on mouseUp
  cnt = count(the windowList) + 1
  wName = "Window" && cnt
  wTitle = wName
  wFileName = the moviePath & the movieName
  wRect = rect(0, 40, 320, 280) + cnt * 20
  w = NewWindow(wName, wTitle, wFileName, wRect, FALSE, 4)
  tell w
    member("windowName").text = wName
  end tell
  open w
end
```

Figure 23-14: This behavior opens an MIAW whose source is the Stage.

The Clear WindowList button uses just a single line of code in its `mouseUp` handler:

```
set the windowList to []

-- or
windowList = []
```

Because none of the windows have references stored in variables, clearing the `windowList` gets rid of all the windows. If there were a variable in existence that held a reference to one of the windows, then that window would not be closed. It would no longer have a reference in the `windowList`, but it would still be in memory. An MIAW stays in memory until no more references of it exist or until the `forget` command is used on it.

Creating MIAWs Tools

You can send messages to and from windows and the Stage by using `tell` statements. This enables you to create tools to modify Director to facilitate authoring. A great example of a tool written in Director is the BehaviorWriter Xtra by Roy Pardi, which is included on the CD-ROM that accompanies this book.

After you've created a movie in Director, you can place it in the XTRAS folder (which is in the same folder where the Director application resides). The next time you run Director, your movie appears in the Xtras menu. To close out this chapter, you make a tool that you can use while you're authoring in Director.

ScoreSaver Xtra—unlimited, persistent undos

The ScoreSaver Xtra enables you to save the state of the Score. You can save as many states as you like and revert to them at any time. The score is saved in film loops in the source movie, not the extra. Every time you open your movie, you can open the ScoreSaver Xtra and revert to previous Score layouts.

Create a new movie, and specify the width as 200 and the height as 200. Add a field named Scores and a button, as shown in Figure 23-15.

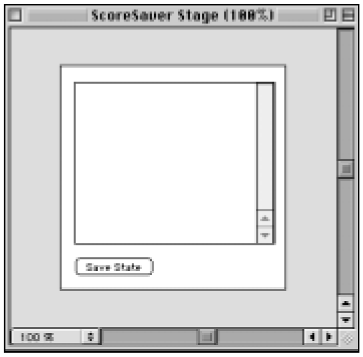
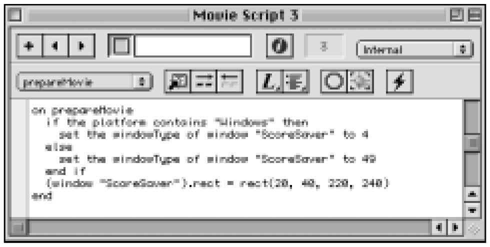


Figure 23-15: The ScoreSaver interface

Create a new movie script as shown in Figure 23-16. This handler changes the `windowType` based on the platform and then moves the window to the upper-left corner. After we're finished with the movie, you will set the movie preferences so that the Stage initially appears off screen.

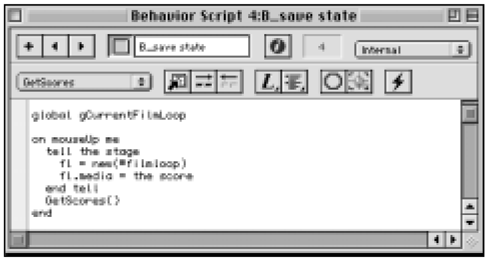
The Score information can be placed in a `filmloop`; we'll take advantage of that property for the ScoreSaver Xtra. Create a new behavior, as shown in Figure 23-17, and attach it to the Save State button.



Movie Script 3

```
on prepareMovie
  if the platform contains "Windows" then
    set the windowType of window "ScoreSaver" to 4
  else
    set the windowType of window "ScoreSaver" to 49
  end if
  (window "ScoreSaver").rect = rect(20, 40, 220, 240)
end
```

Figure 23-16: Changing the windowType based on the platform and placing the window in the upper-left corner



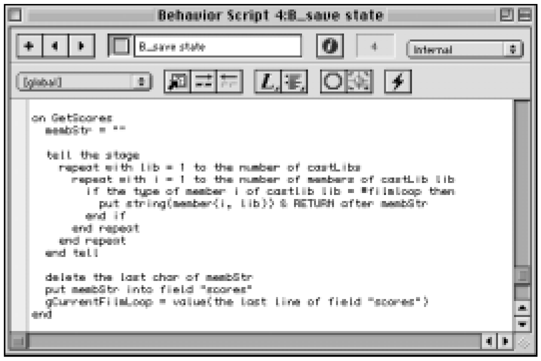
Behavior Script 4:B_save state

```
global gCurrentFilmLoop

on mouseUp me
  tell the stage
    fl = new(#filmloop)
    fl.media = the score
  end tell
  GetScores()
end
```

Figure 23-17: The behavior for the Save State button

The `mouseUp` in the B_save state button uses a multiline `tell` statement to tell the Stage to create a new `filmloop` and then to set the media of the new `filmloop` to the Score. After the `tell` statement, a user-defined handler — `GetScores` — is called. That handler should be added to the B_save state behavior (see Figure 23-18).



Behavior Script 4:B_save state

```
on GetScores
  membStr = ""

  tell the stage
    repeat with lib = 1 to the number of castLibs
      repeat with i = 1 to the number of members of castLib lib
        if the type of member i of castLib lib = #filmloop then
          put string(member(i, lib)) & RETURN after membStr
        end if
      end repeat
    end repeat
  end tell

  delete the last char of membStr
  put membStr into field "scores"
  gCurrentFilmLoop = value(the last line of field "scores")
end
```

Figure 23-18: The GetScores handler is added to the B_save state behavior.

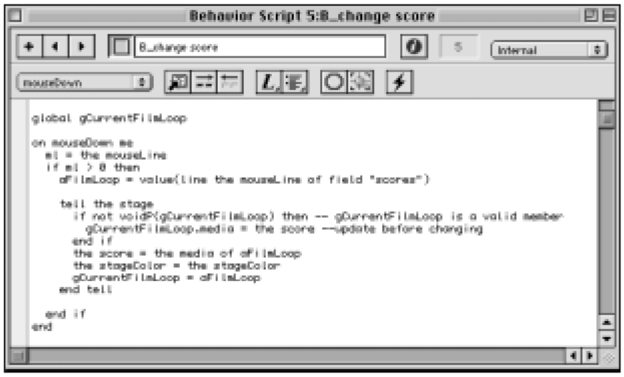
`GetScores` creates a string variable called `membStr`. It then tells the Stage to search through every member of every `castLib`. If it encounters a `filmloop`, it converts the member reference to a string and concatenates it to the `membStr`. When it has finished looping through all of the `castLibs`, it deletes the last `char` of `membStr` because it is a return. Then it puts that string into the field scores. The last thing the handler does is convert the last line of field scores into a member reference and place it into the global variable `gCurrentFilmLoop`.

Tip

An important aspect of `tell` statements is the capability to share variables with the code outside of it. `membStr` is created, and then the `tell` statement begins and all of the code in-between relates to the Stage; yet the variable `membStr` can be used. This works the other way as well: You can create a variable within a `tell` statement and assign it a value from another MIAW or the Stage and use the variable outside of the `tell` statement.

To recap, the B_save state behavior takes a snapshot of the Score and places it in a `filmloop`, and it then adds that member reference to the field.

The next behavior that you need to create is one for the field scores. It needs to let the user click a member reference, which then sets the Score to that member's medium. Create a new behavior, as shown in Figure 23-19, and attach it to the field scores sprite.



```
global gCurrentFilmLoop

on mouseDown me
  ml = the mouseLine
  if ml > 0 then
    aFilmLoop = value(line the mouseLine of field "scores")

    tell the stage
      if not voidP(gCurrentFilmLoop) then -- gCurrentFilmLoop is a valid member
        gCurrentFilmLoop.media = the score --update before changing
      end if
      the score = the media of aFilmLoop
      the stageColor = the stageColor
      gCurrentFilmLoop = aFilmLoop
    end tell

  end if
end
```

Figure 23-19: The behavior for the scores field sprite

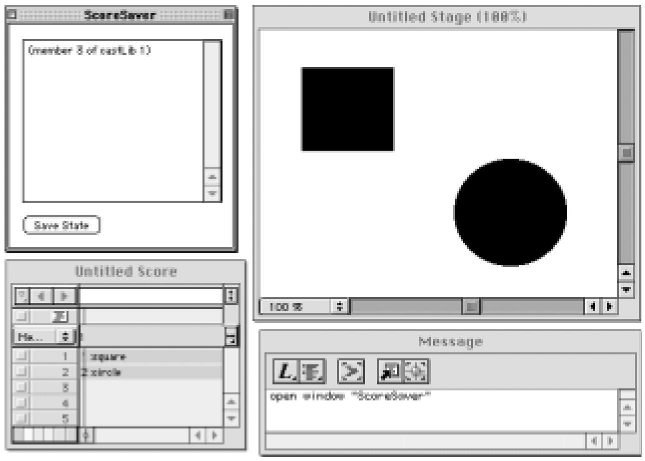
This behavior gets the `mouseLine` when the field is clicked. If the `mouseLine` is above 0, then it sets the variable `aFilmLoop` to the value of the line clicked. This should result in a member reference. The global `gCurrentFilmLoop` is checked to see whether it contains a member reference yet. If it does, it sets its value to the Stage. After that, the Score is given the value of the member reference chosen by the mouse click. Setting the `stageColor` to `the stageColor` refreshes the Stage. Then `gCurrentFilmLoop` is assigned the value of `aFilmLoop`.

The last step, then, before trying our Xtra is to remember to put a `go to the frame` in the `exitFrame` of the behavior channel (frame script channel).

Open a new movie and then open the scoresaver.dir from the Message window by typing the following:

```
open window "ScoreSaver"
```

It should appear in the upper-left corner. Now open the Score window and draw a few items on the Stage so that you can test the ScoreSaver Xtra. When you're happy with what you've drawn, click Save State, as shown in Figure 23-20. A member reference appears in your ScoreSaver list. This member corresponds to a film loop in your cast that was just created.



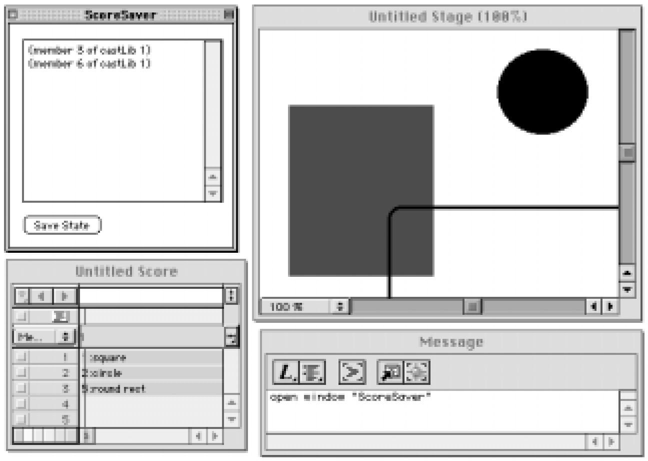
```
(member 3 of castLib 1)
```

```
open window "ScoreSaver"
```

Figure 23-20: The state of the Score has just been saved in member 3 of castLib 1.

Make some additional changes to the Score and click Save State again. Another member reference appears in the list and another `filmloop` is added to your movie (see Figure 23-21). Now click the first reference, and the Score switches back to when you had previously clicked Save State. Click the second reference, and the Score switches to the second time you clicked Save State. You can save as many variations of the Score as you like and then switch back to them, like turning pages of a book.

Note The ScoreSaver saves the state of the Score, with the exclusion of the Marker channel. It only saves the Score layout, *not* cast members. If you delete any cast members used by a previous Score state, it references whatever happens to be in that member position now.



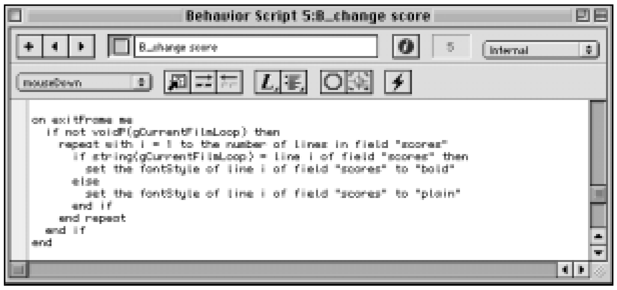
```
(member 5 of castLib 1)
(member 6 of castLib 1)
```

1	1:square
2	2:circle
3	3:round rect
4	
5	

```
open window "ScoreSaver"
```

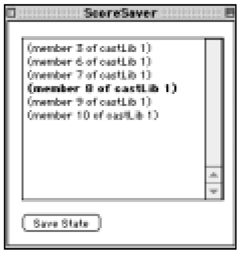
Figure 23-21: Another snapshot of the Score has been saved.

The interface needs to be improved a bit on the ScoreSaver. It would be nice to see which member is selected. Add the code shown in Figure 23-22 to the behavior B_change score. The `exitFrame` continuously checks to see which line of field scores correspond to the member reference in the global `gCurrentFilmLoop`. If it finds it, it makes that line bold. The result is shown in Figure 23-23.



```
on exitFrame me
  if not voidP(gCurrentFilmLoop) then
    repeat with i = 1 to the number of lines in field "scores"
      if string(gCurrentFilmLoop) = line i of field "scores" then
        set the fontStyle of line i of field "scores" to "bold"
      else
        set the fontStyle of line i of field "scores" to "plain"
      end if
    end repeat
  end if
end
```

Figure 23-22: Adding an `exitFrame` handler that keeps track of the currently selected Score state



```
(member 5 of castLib 1)
(member 6 of castLib 1)
(member 7 of castLib 1)
(member 8 of castLib 1)
(member 9 of castLib 1)
(member 10 of castLib 1)
```

Figure 23-23: The result of adding the code in Figure 23-22

Suggestions for ScoreSaver

At this point, we're sure that you can imagine additional features that would be useful, such as the following:

- ✦ A button to update the list
- ✦ A button to delete a saved state
- ✦ A naming convention for the `filmloops` so that the ScoreSaver doesn't use `filmloops` other than the ones that it has created
- ✦ A method for the user to give meaningful names to the saved states

We'll leave these as exercises for you. For now, you can move your ScoreSaver to your Director XTRAS folder if you want to use it in your daily authoring. The last modification that you might want to make is to set the Stage Location of the movie by choosing Modify ➪ Movie ➪ Properties and setting a value that is off screen, so that the user does not see the `windowType` change and then the window jump to the upper-left.

The ScoreSaver is a good example of the power of Director and a little Lingo. In a short period, you've created a tool that enables you to have an infinite amount of undos. On top of that, these undos are still available after you quit the movie and come back to it later. Another interesting aspect of this tool is that the capability to create it is not a new feature: We first wrote a version of this tool when Director 5 came out. Many experienced Lingo programmers are unaware of this functionality.

Summary

Movies in a window are effective for use as dialog boxes, control panels, toolbars, menus, floating palettes, and movie-building utilities. Take a moment to recall what you learned in this chapter:

- ✦ Linked movies provide a method of putting a movie within a movie.
- ✦ Linked movies have some limitations that make them difficult to work with.
- ✦ MIAWs are Director movies playing within a separate window.
- ✦ More than one MIAW can be opened at a time and the source movie can be the same for more than one MIAW.
- ✦ MIAWs can be "told" what to do from other MIAWs or the Stage.
- ✦ MIAWs can be changed and modified on the fly.
- ✦ MIAWs are a powerful way to extend the capabilities of Director.

Now we leave the world of MIAWs and move on to the world of Shockwave and NetLingo, where MIAWs do not exist.

✦ ✦ ✦