第15章　图形界面元素

无论是创建一个简单的演示或是一个复杂的软件，主要都是通过 Director影片和用户相互作用的用户界面元素来实现的。它们可以像按钮一样简单，也可以像滑动条一样复杂。本章介绍最常用的用户界面元素和用来创建它们的行为。

15.1　创建显示掠过

第14章“ 创建行为 ”中，包含了一个显示简单掠过行为的剧本。典型地说，掠过就是当光标位于其上时改变演员的角色。但是，还有另一种掠过形式，它并不改变被掠过的角色本身，而却改变另一个角色。

这种界面元素的典型用途是，首先显示一个项目清单，然后当用户掠过它时，在屏幕另一部分显示更多的关于这些项目的信息。图 15-1显示就是上述内容的显示。掠过左边的三个角色，从而在右边的角色中产生三个不同的文本演员。

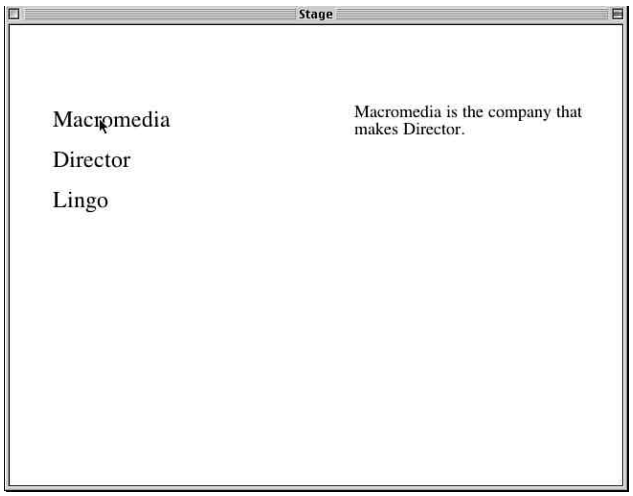

图15-1　屏幕上显示了左边的三个角色和右边的一个角色。右边角色的变化是由光标置于左边的哪个角色之上所决定的

完成这个任务的行为与第 14章中所讨论的掠过行为类似。但它不是改变被掠过的角色本身，而是改变另一个角色。它还要知道哪个演员被使用。最后一个参数应是一个缺省演员，当光标不在任何一个掠过角色上时，它才得以显示。下面是行为的开始部分：

```
property pRolloverMember, pRolloverSprite, pDefaultMember

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pRolloverMember, [#comment: "Rollover Member",¬
    #format: #member, #default: " "]
  addProp list, #pRolloverSprite, [#comment: "Rollover Sprite",¬
    #format: #integer, #default: 0]
```

```
  addProp list, #pDefaultMember, [#comment: "Default Member",¬
    #format: #member, #default: " "]
  return list
end
```

该缺省演员应该是剪辑室里掠过角色中的一个演员。我们可以用 on mouseEnter将角色变成想要的演员。然后，在光标离开角色时， on mouseLeave处理程序能够将它转回到缺省演员。

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

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

当光标在角色间移动时， mouseEnter和mouseLeave消息控制掠过角色显示哪个演员。在这时，我们要确保角色没有重叠，否则消息的处理将产生错误的效果。例如，如果角色 1和角色2重叠，光标在离开角色 1之前，就会进入角色 2。结果显示的却是缺省的演员而非由角色 2定义的演员。

因为一个角色能附带多个行为，所以，我们也能够对这些角色添加按钮行为。如果那样的话，掠过将由此处的掠过显示行为处理，而点击则由按钮行为处理。还可以制作显示预览信息的角色，如某个角色可以显示“ Click this button to go to the index ”，并对点击动作有所反应，能够跳到另一帧。

参见第14章里的14.2节“ 创建简单的行为 ”，以了解关于创建掠过行为的更多信息。

15.2 使用复选框

复选框是一个具有“ 开(On) ”和“ 关(Off) ”两种状态的按钮。Director有一个复选框演员类型，它是按钮演员的一个变种。我们可以使用这种内置演员作为复选框，或建立自制复选框，其中的每个选项包括两个演员。

为内置复选框演员制作行为几乎是不必要的。这些演员自身就可以响应鼠标点击动作，被点击时在框内放一个标志，然后在下一次点击时，除去它。图 15-2 显示了一组复选框演员。

行为能做的一件事情是使确定哪个复选框被选中的工作更加容易，我们可以将它作为演员的一个简单的 Lingo属性。例如， (member "apple check box") .hilite。hilite属性返回的是真或假。

图15-2 包含三个复选框演员的一组选项

行为能够使用同样的句法，但因为行为被分配给角色，而不是演员，我们可以根据角色的编号获得复选框的状态。下面是做这项工作的一个简单行为：

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

我们能够通过isChecked (sprite X)语句询问，以查看任一复选框的状态，其中“ X ”是角色的编号。或者用 sendSprite (sprite X， #isChecked)做同样的事情。用这两种语句都能得到

TRUE或FALSE值(TRUE值表示在复选框中有选中标识， FALSE值表示复选框里面是空的)。

因为内置复选框的外观很一般，所以用位图创建自定义的多状态按钮常常比较合适。图15-3显示了三个这样的按钮，其中中间的按钮被选中。

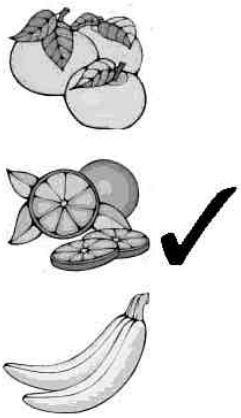

图15-3 三个角色都使用了复选框的位图形式。行为控制着这些角色

每个角色能够包含两个演员中的其中一个：“开(On)”状态和“关(Off)”状态。所以，图15-3共有六个演员。在这个例子中，除了挨着角色的“√”标志外，它们的“开”状态看起来与“关”状态相同。

处理这些状态的行为需要知道“开”和“关”状态的演员是什么。它也要知道角色起始状态是“开”还是“关”。下面的三个属性和参数正是解决这些问题的：

```
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
```

当角色开始时，它将自己调整到正确的状态，而不考虑在剪辑室中是什么状态。这个工作由on beginSprite处理程序完成。因为我们需要根据pState属性重新使用这些改变角色的演员的代码，因此创建一个自定义处理程序，以完成这项工作比较合适。 on beginSprite处理程序需要调用该自定义处理程序。

```
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
```

当用户点击角色时，它的状态要发生改变。如果它是“开”状态，则它要变成“关”，反之亦然。这个工作用使用not操作符的一行语句就可完成。

```
on mouseUp me
    pState = not pState
    setMember(me)
```

```
end
```

在状态改变后，on mouseUp 处理程序调用on setMember处理程序改变演员。

最后的一个处理程序可以是返回复选框状态的一个函数。它可以在影片剧本或其他行为中被其他的Lingo处理程序使用。

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

记住，我们也能够通过下面的这种方式直接访问这个属性： sprite(x).pState

参见第28章“商业软件”里的28.3节“制作调查问卷”，以了解更多关于使用复选框的信息。

15.3 使用单选按钮

介绍了复选框自然要提到单选按钮。它们十分类似。但是，复选框能够让用户从某一清单中选择一个或多个选项，而单选按钮只允许从一个清单中仅仅选择一项。

Diretor也有以按钮演员的变种形式存在的内置单选按钮演员。它的工作方式与复选框相同，它接收鼠标点击，从而将自己转成“开”或“关”状态。图15-4显示了一组单选按钮。

图15-4 包含三个单选按钮的一组选项

复选框几乎不需要任何 Lingo语言的参与，而单选按钮则确实要用到一些 Lingo语言。原因是到目前为止，单选按钮并不知道它们彼此的存在关系。在图 15-4的例子中，Apples按钮和Oranges按钮不以任何方式互相影响。这就是说用户能同时选中两个按钮，这与单选按钮的选择原则相违背。用户应该只能够选中 Apples、Oranges或Bananas里的一个。

我们可以用一个简单的行为来限制三个按钮，使它们共同协作。首先，该行为要知道这一组中还有什么其他的角色。它也要了解该角色的初始状态是否为“选中”，即TRUE。

```
property pState, pGroupList

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

pGroupList属性包含该组中角色的线性列表。例如，如果此例中的三个单选按钮是角色1~3，pGroupList将是[1,2,3]。

pState属性的设置也要格外小心。这一组中有一个并仅能有一个单选按钮被设置为真。然后，在on beginSprite处理程序中，如果某个按钮最初的 pState被设置为真，那么该角色要使用“开”演员。

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

自定义处理程序on turnMeOn用于设置该角色的演员。因为这些都是单选按钮，它还必须确保此组中的其他角色被关闭。

```
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
```

该处理程序中的repeat命令使用repeat with i in 的格式。repeat命令的这种特殊格式仅能随列表使用。它不是按变量i逐一计数，而是在列表的值间移动。如果列表是 [5,8,14]，循环运行3次。i值在3次循环中分别被设置为5、8和14。

sendSprite命令用于发送#turnMeOff消息给列表中的每一个角色。 if语句确保消息不被回送给当前角色。on rutnMeoff处理程序十分简单：

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

既然已有处理程序可以打开当前的单选按钮，那么处理鼠标点击就十分容易了。只要调用on turnMeOn处理程序，以打开当前角色和关闭所有其他的角色就可以了。

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

最后的一个处理程序能被用于确定此组中目前选中的是哪个角色。它使用像 on turnMeOn处理程序中一样的repeat循环，但在对角色的改变上有所不同，它只是找到一个“开”状态的角色并返回那个值。

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

该处理程序依赖这样一个事实，即有一个并仅有一个角色被打开。也就是说，从来不会出现没有角色或多于一个的角色为“开”状态的情况。它也有一种非凡能力，即无论用该组中的哪个角色调用它，它都能返回相同的答案。如果一个单选按钮组有角色 1~3，因为角色1和角色2有同样的pGroupList属性值，所以selected (sprite 1)和selected (sprite 2)将返回同样的答案。

虽然该行为处理了单选按钮组的复杂性，但它并不提供使用自定义位图作为单选按钮的机会，而只能使用单调的内置单选按钮演员。然而，要做到这些，只要做一些简单的修改就可以了。

第一步是添加两个新的属性，分别代表 on和off状态的位图。看上去与在复选框中的做法相同：

```
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
```

然后，要修改on turnMeOn和on turnMeoff处理程序以设置该角色的演员，而并非演员的hilite(突出显示)属性：

```
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
```

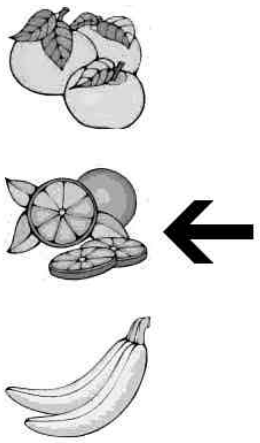

图15-5 一组自定义的单选按钮。行为根据选中的项目使用相应的演员

这个行为的余下部分与前面相同。图 15-5显示了屏幕上可以看见的效果。关演员和开演员的不同仅是在开演员包括一个指向图片右边的箭头，以标识它被选中。

参见第27章“教学软件”里的27.5节“制作标准化考试”，以了解更多使用单选按钮的例子。

15.4 拖动角色

虽然不用Lingo语言也可能实现拖动角色(通过设置剪辑室中角色的可移动属性)，但我们也能借助Lingo语言添加各种拖动功能。本节中，我们首先学会怎样使用 Lingo语言拖动角色，然后进行一些拖动应用。

15.4.1 简单的拖动剧本

拖动剧本可以简单到让角色一直跟着光标走。然而，一个真正的拖动行为要等到用户点击角色时才发生，然后在用户释放鼠标按钮前，一直跟着鼠标运动。

要实现这种类型的拖动，我们甚至不需要任何参数。然而，我们确实需要一个属性，以通知该行为何时发生。当角色开始时，此属性将被设置成假。

```
property pPressed

on beginSprite me
  pPressed = FALSE
end
```

在用户点击角色时，该行为要将pPressed属性变成TRUE 。而当用户释放按钮时，它又要被设置成FALSE。

```
on mouseDown me
  pPressed = TRUE
end

on mouseUp me
  pPressed = FALSE
end
```

当光标不在角色上时，用户可能要释放鼠标按钮。鼠标位置被实时更新，而我们只要在每个帧循环时设置角色的位置就可以了。无论什么时候鼠标按钮被释放，为了确保拖动停止，我们还要使用 on mouseUpOutside处理程序。

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

最后，所有的重要任务都由 on exitFrame处理器完成。它查看 pPressed属性是否为真，如果是则将角色移到鼠标位置。

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

15.4.2 更好的拖动行为

简单的拖动剧本中存在的最主要问题是外观修饰的问题。无论用户点击角色的哪个位置，角色都好象突然集中到光标上。所以，如果我们在角色的右上边点击，该角色立刻会发生偏移，以使它的中心，或实际套准点落在在光标的正下方。

要修改这个突然变化并不成问题。当最初的点击发生时，鼠标位置和角色的中心之间的距离将被记录下来。然后该值被应用到角色位置的每一次变化中。结果，无论用户将角色拖到哪里，光标和角色都保持同步状态。

为了产生这种变化，首先将 pClickDiff属性添加在属性声明部分。然后，改变 on mouseDown处理程序，从而能够像下面的程序一样记录点击位置和角色位置间的距离：

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

注释 clickLoc属性与mouseLoc类似。它们返回的值都是一个点。然而，如果鼠标在鼠标点击时和Lingo代码语句运行时之间发生移动，mouseLoc可能会改变。clickLoc返回在on mouseDown或类似处理程序中的最后一次点击的精确位置，而mouseLoc返回当前

鼠标位置。

既然该距离值存储在pClickDiff中，它能够在on exitFrame处理程序中被应用到角色的位置上。

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

如果，仍觉得不太清楚，看看有它的真实例子会有所帮助；请查看CD-ROM中的相关实例。

15.4.3 点击、拖动和锁定

拖动的一种应用是制作一个配对游戏或测验。图 15-6是一种配对游戏。在屏幕的一侧有一种类型的元素，而在另一侧有另一种类型的元素。这个游戏是由成对的元素构成的。需要由用户将左边的角色拖到右边的角色上面。

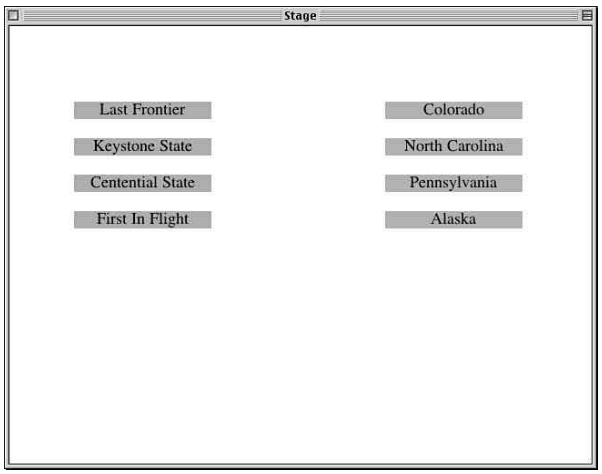

图15-6 屏幕显示了四对匹配项。用户的任务就是将右边的项拖到左边正确对应的那一项上

虽然前面已经说过，这个过程可以由拖动剧本帮助完成，但是这些剧本并不具备通知用户匹配是否正确的功能。做这件事情更好的方法是编写一个剧本，使得当某一角色贴近它所属于的那个位置时，它的位置将被锁定。

这需要一些参数。第一个参数是将要与该行为的角色锁定在一起的那个角色的编号。第二个是在被拖动角色与目标角色将要被自动锁定之前二者间的最大距离。

同时，在角色与目标位置不够近，以致无法达到锁定程度时，我们需要告诉行为该如何动作。一种做法是让角色跳回到起始位置，另一种做法是让角色留在当前的位置。

除了这些新的属性，还需要使用前面提到过的 pPressed和pClickDiff属性。还需要一个属性存储角色的原始位置，用于跳回起始位置的情况。

```
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
```

pLockDist是这个行为的一个关键属性。如果没有它，就要求用户精确地将角色锁定在目标角色上。

这个行为需要从初始化pPressed属性开始，同时对pOrigLoc属性进行设置。

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

除了要从on mouseUp和on mouseUpOutside处理程序中调用一个自定义处理程序外，鼠标点击处理程序与前述相同。这个行为要做的事情并不只将 pPressed置回FALSE状态，所以最好是将它全部放在自定义处理程序中，而不必在 on mouseUp和on mouseUpOutside间复制代码。

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

on mouseUp me
  release(me)
end

on mouseUpOutside me
  release(me)
end
```

on exitFrame剧本与前述一样。

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

自定义的on release处理程序必须做以下几件事情。首先将 pPressed设置成FALSE。然后，要判断是否角色现在足够靠近目标以锁住它的位置。它也要调用另一个自定义处理程序 on distance来做这项工作。这个函数中用两个点作参数，并以像素为单位返回两点间的距离。

如果角色与目标位置足够贴近，角色的新位置将被精确地设置到目标角色的位置。否则，如果pSnapBack属性为TRUE，则角色返回到起始位置。如果不是前两种情况，则角色停留在用户释放它的位置。

```
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
```

on distance函数是一个在许多行为中使用都很便利的工具。它取两个点为参数，并计算出横-纵坐标差的平方和的平方根。我们可能还记得中学学过的这个三角几何公式：

$$\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$$

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

余下的工作只是将这个行为赋予一些角色了。在图 15-6显示的例子中，这个行为将被赋予右边的角色。这些角色处于更高的角色通道上，因此出现在左边的角色的上方。每一个角色的pLockToSprite属性应当被设置成与之配对的角色。

如果我们确实制作了一个这样的游戏，可以使用下面这个影片剧本判断是否所有的角色都被锁定在正确的地方。

```
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
```

这个影片处理程序认为局部变量 done设置成TRUE后，游戏即告结束。于是它查看角色5~8 —— 即被拖动角色 —— 的位置，并将这些位置与指定的目标角色的位置进行比较。如果它们完全匹配， done不会被设置成FALSE。在这种情况下，结果就是一个简单的 beep(系统警告声)。然而，我们也能让它跳到另一帧或播放另一种声音。

对这个处理程序调用的最佳位置是在行为的 on release处理程序中。它仅在每一次角色被锁定时检测它，所以将它放在 sprite(me.spriteNum).loc = sprite(pLockToSprite).loc语句后面。

参见第27章里的27.1节“ 制作配对游戏 ”中关于点击、拖动和锁定行为的例子。

15.4.4 拖动和投掷

应用拖动行为的另一种方法是使用户能够抓住并释放某一角色。像其他的拖动行为一样，随着鼠标被按下，我们点击角色并拖着它运动，但是释放鼠标时，角色会因为带有一定的动量而继续保持运动。

这比一个简单的拖动行为要复杂的多。首先，除了有“ 按下 ”和“ 非按下 ”两种状态外，还需要其他状态。这第三种状态是投掷状态。所以， pPressed属性将被pMode属性代替，后者具有#normal、#pressed和#throw三个值。

另外，我们要知道当释放角色时，要向哪个方向、投掷多长的距离。测量角色被释放时与被释放前的一刻之间的距离，将会得到一个合适的信息。然而，取被释放那帧的前一帧的位置值是不太合适的，因为如果那样，时间间隔太短：如果帧速度是 60fps，这仅有1/60秒。更好一些的做法是用一个参数，让它来决定选向回数多少帧，用来计算被释放角色的动量。这个参数的合适的缺省值是 5。所以，如果一个角色被点击，并拖动了 150帧，然后被释放，

就将第145帧的位置和第150帧的位置相比较，以设置投掷动量。我们需要一个列表，记载在任意给定时间内的最后5帧的位置。下面是属性列表和这个行为的 on getPropertyDescriptionList处理程序：

```
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
```

该角色从将pMode设置成 #normal开始。

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

当用户点击时，此模式必须设置成 #pressed，并且必须对用于存储这些位置的列表进行初始化处理：

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

每当鼠标按钮被释放， on throw自定义处理程序就运行。它先计算当前鼠标位置和pLocList中的第一项的差，然后被 pThrowSpan除，以得到每帧的相对动量值。这样，如果pThrowSpan被设置成5，pMoveAmount将被设置成角色的当前位置减去角色在 5帧前的位置，再被5除，就得到了所要的计算结果。

```
on mouseUp me
  throw(me)
end

on mouseUpOutside me
  throw(me)
end

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

无论角色是被拖动还是被投掷， on exitFrame处理程序必须对角色进行移动。如果是拖动状态，它须将角色位置设置在当前鼠标位置。同样，它也要在 pLoclist中记录该位置。如果pLocList内的项目比pThrowSpan所指定的项目多，最前面的项目则被删除。

如果模式是#throw，pMoveAmout被用来移动角色。另外，这个属性与 0.9相乘，因此它被减少了大约10%。这是模仿一种摩擦力，当角色被投掷时，我们希望角色也能体现这种力量的影响。如果我们愿意， 0.9可以被设置成属性，叫做 pFriction，并根据影片作者的意愿加以改变。

```
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)

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

考虑一下这个行为实现的可能性。我们是否想添加一些代码，像在回弹行为中使用的那样，那么角色就会在屏幕的各边间回弹吗？添加重力怎么样？如果角色离另一个位置足够近，怎样编程才能够将角色锁在那个位置上呢？将这三个问题联系起来，我们可以获得模拟篮球自由投掷的视频游戏。

15.5 创建滑动条

滑动条是一种我们熟悉的用户界面元素，它用在所有大型的软件中。滑动条是用户在一个小范围内输入数值最好的办法。它们甚至也用在行为的 Parameter对话框中。

创建带有几个位图演员和一个行为的滑动条是相当复杂的，许多行动必须被考虑，并且构成滑动条的元素需要多个角色。

图15-7显示了一个滑动条的外观。它直接模仿在Director的参数设置对话框中使用的滑动条。

图15-7 典型的滑动条大约包括六个部分

一个完整的滑动条元素包括六个部分：滑标、阴影、背景图形、两个按钮和一个文本域。然而，我们可以从创建三个主要的部分开始：滑标、阴影和背景图形。阴影是滑标左边的较暗的颜色。在这个例子里，它是唯一的非位图元素，它使用一个图形角色。

实际上我们只需要一个赋予滑标角色的行为剧本。像按钮和拖动行为一样，该行为有要求有一个属性来说明是否它正处于被按下的过程。它还要知道从左到右它能够移动多远的距离，代表什么范围的值，如 1~3、0~100或-500~500等。还需要与阴影角色的关系，所以要有一个属性存储阴影角色的编号。另一个属性要像下面一样，用于记录滑标的当前值：

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

on getPropertyDescriptionList只要以4个属性作为参数使用：最小值、最大值、滑动条的起始值，还有阴影角色的编号。

虽然最小值和最大值由行为的 Parameters对话框设置，但是行为也要知道最小值和最大值在屏幕上的实际位置。为了确定这些位置，有一个小小的技巧。阴影角色要用来标识滑标角色的精确范围。因为阴影角色在开始时，总要被设为复位，使用它显示滑块的边界不影响它以后的显示。在图 15-7显示的滑动条中，阴影角色在背景图形的内部一路伸展。它的 rect由行为的on beginSprite处理程序记录。然后，它被设置成图 15-7中显示的那样，正确地显示出来：

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

滑标的左边和右边的限制可以被添加到 on getPropertyDescriptionList处理程序中，但是这意味着我们需要确定它们在屏幕上的精确位置，并把这些值输入进去。更坏的是，如果滑动条移动了，哪怕只移动了一个像素，都不得不重新输入这些数值。使用阴影角色作为某种“模版”，可以解决我们的这个麻烦，并且在舞台上很容易调节。

on beginSprite处理程序包括对自定义的on setMarker和on setShadow处理程序的调用。它们取得滑标的当前值，并设置这两个角色的位置。

on setMarker处理程序首先分析滑标的值的范围。如果滑标可以从 0~100滑移，那么移动范围是100(不是101)。它先不考虑真正的范围，而把滑标的值计算成 0~1之间的一个数值。然后，它把这个百分比数值应用于屏幕上的真实尺寸范围，以得到滑标的位置：

```
-- 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)

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

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

on setShadow处理程序把阴影设置为它的原始矩形，但调节它的右边，使之落在滑标下面。

```
-- 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
```

这些处理程序完成了将滑标设置到起始位置的任务。现在，我们需要用到一些处理程序，以使用户能够点击并拖动其中的滑标：

```
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
```

on exitFrame处理程序查看并确认pPressed为真，然后调用其他的处理程序以处理这项工作。on setMarker和on setShadow处理程序对这项工作起作用，但对它们的调用在 on moveMarker处理程序之后。这个处理程序所做的工作与 on setMarker相反，它根据鼠标的位置确定滑标值。

另外，它将这个值解释成整数。而如果我们希望此滑标显示浮点数，只要将下面相关语句去掉就可以了。

```
--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
  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
```

on moveMarker处理程序也用于确定滑标的新值要落在刻度范围里。注意，这个处理程序

唯一的目的就是设置 pValue属性。在它被 on exitFrame处理程序调用后， on setMarker和on setShadow处理程序对角色进行更新。

因为滑标的数值存放在 pValue属性中，所以返回滑标值的处理程序十分简单。我们可以从其他的行为或影片剧本中调用带有 sendSprite的处理程序，以得到滑标的当前值，甚至在它被拖动的时候也可以。

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

还可以为滑动条添加几个元素，以使它像其他软件中的滑动条一样完整。一个文本域可以显示滑标当前的值；两个按钮使用户能够一次让滑标移动一个值。

图15-8显示了所有的这些元素。如果包括按钮的按下状态，一共有八个演员。

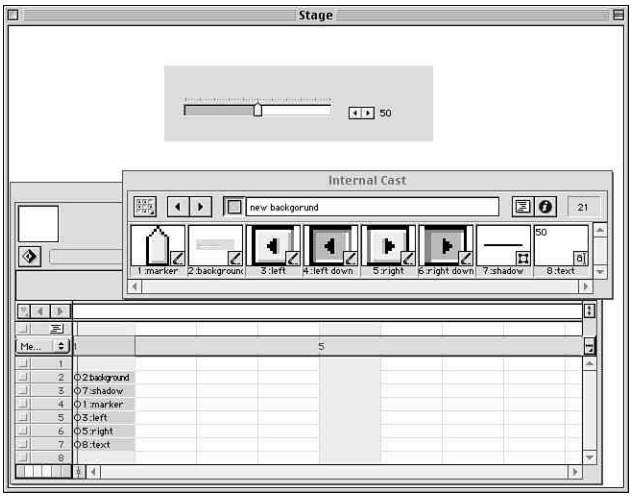

图15-8 演员表窗口显示了滑动条的8个元素，剪辑室显示每个角色的布置，舞台中显示了装配好的滑动条

我们想为这种简单的滑动条行为添加的第一种附加功能是一个文本域，它能显示滑标的值。做这项工作，要创建一个文本域，并将它放置在舞台上。我们要告诉滑标角色，文本角色位于哪里。另外一个参数将做这项工作。在行为的顶部属性声明部分中添加一个 pTextSprite属性。在on getPropertyDescriptionList处理程序中添加如下语句：

```
addProp list, #pTextSprite, [#comment: "Text Sprite",¬
   #format: #integer, #default: 0]
```

on setText处理程序将当前的滑标值放在文本域中。在没有文本角色被使用的情况下，该处理程序要证实pTextSprite属性不是0。如果是0，它就认为无文本角色，并不放置文本。

```
-- this handler sets the text of the text sprite
on setText me
  if pTextSprite <> 0 then -- is there a text sprite?
    sprite(pTextSprite).member.text = string(pValue)
  end if
end
```

对on setText的调用要在on beginSprite和on exitFrame处理程序中调用on setShadow后立即进行。

滑动条所需要的另一种元素是按钮，这些按钮使用户能够向左或向右把滑标移动一个单位值。这些按钮需要“正常”和“按下”两种状态，像图 15-8中显示的一样。

这些按钮需要有它们自己的行为，但是在我们创建这些行为之前，要在滑标的行为中要添加一个处理程序。这个处理程序使滑标能够在任一方向上移动一个值。它还要有一个参数，根据滑标向哪个方向移动来确定这个参数是 #left或#right。它也要进行与on moveMarketOne处理程序同样的边界检测，以确定滑标不会超出边界。

```
-- this handler moves the marker one value left or right
on moveMarkerOne me, direction
  if direction = #left then
    pValue = pValue - 1
  else if direction = #right then
    pValue = pValue + 1
  end if

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

  setMarker(me)
  setShadow(me)
  setText(me)
end
```

on moveMarkerOne处理程序以调用三个用来更新滑块的处理程序为结束。这样，调用这一个处理程序就可以用其他三个处理程序自动对滑标进行改变和更新。

负责这两个按钮的处理程序同任何简单的按钮处理程序一样。它通常只控制按钮角色的“正常”和“按下”状态。它也必须知道滑标角色在哪儿，以使它能将 moveMarkerOne消息传送给滑标角色。有一个关键的参数是 pArrowDirection,它根据哪个按钮被选中而被设置成 #left或#right。

```
property pDownMember, pOrigMember -- down and normal states
property pPressed -- whether the sprite is being pressed
property pMarkerSprite -- the number of the marker sprite
property pArrowDirection -- 1 or -1 to add to slider

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pMarkerSprite, [#comment: "Marker Sprite",¬
    #format: #integer, #default: 0]
  addProp list, #pDownMember, [#comment: "Arrow Button Down Member",¬
    #format: #bitmap, #default: " "]
  addProp list, #pArrowDirection, [#comment: "Arrow Direction",¬
    #format: #symbol, #range: [#left,#right], #default: #right]
  return list
end
```

滑动条按钮行为的剩余工作是处理鼠标点击，并在按钮按下时，在每一帧调用一次滑动条行为中的moveMarker One处理程序：

```
on beginSprite me
  pOrigMember = sprite(me.spriteNum).member
end

on mouseDown me
  pPressed = TRUE
  sprite(me.spriteNum).member = member pDownMember
end

on mouseUp me
  liftUp(me)
end

on mouseUpOutside me
  liftUp(me)
end

on liftUp me
  pPressed = FALSE
  sprite(me.spriteNum).member = member pOrigMember
end

on exitFrame me
  if pPressed then
    sendSprite(sprite pMarkerSprite, #moveMarkerOne, ¬
     pArrowDirection)
  end if
end
```

有了这八个演员和两个行为，滑动条就有了其他软件中所使用的滑动条的功能。更好的是，我们可以任意改变各个元素的外观，以使我们的滑动条更有风格，更适合我们的设计要求。

对于滑动条的外观和行为，我们有很多控制权。可以把程序稍加改变，在文本域中的数字后面放置一个附加的词。因此，除了写成“ 50 ”外，还可以写成“ 50% ”或“ 50个单位(widdgets) ”。如果阴影角色的表现与我们想要的结果不同，我们可以轻微移动滑动条屏幕的边界。我们甚至可以删除阴影角色。

我们也可以改变横向位置和边界的引用变量，以将它们转换成纵向位置和边界，而得到一个垂直滑动条。这表明与其他创作程序的拖－放界面元素相比， Lingo的行为给我们提供了更多的控制权。

参见第30章“ 声音软件 ”里的30.5节“ 调节音量控制 ”中的滑动条的例子。

15.6 创建进程条

使用一些与滑动条相同的技术可以创建进程条行为。进程条是一个矩形，此矩形随着处理过程的完成，逐渐扩大，最终填满一个空间。一个例子是在 Director里我们每次选用 File｜Save时所显示的进程条。

如果一个Lingo操作需要花一秒以上的时间完成，我们可能想要显示一个进程条，以使用户知道，计算机并不是死机，而只是在执行他们的要求。

图15-9显示了一个简单的进程条。它显示两个角色：一个空心的矩形和一个被填充的矩形。为使这个进程条看起来与其他软件的进程条一样，矩形的填充可被设置成图案，而非单一颜色。

图15-9 一个简单的进程条显示某任务大约已执行了1/3

为创建一个进程条，我们要用一个进程去测试它。下面有两个影片处理程序，它们共同协作，以计算出0~1000之间的所有素数。素数是指只能被1和它自身整除的数字。

```
on findPrimeNumbers
  list = []
  repeat with i = 1 to 1000
    if isPrime(i) then add list, i
  end repeat
  return list
end

on isPrime n
  repeat with i = 2 to integer(sqrt(n))
    div = float(n)/float(i)
    if div = integer(div) then return FALSE
  end repeat
  return TRUE
end
```

on isPrime函数用2和该数值的平方根间的每一数字进行偿试，以查看是否有整除的情况。它将除后的数值与它该数值转换成整数后的结果相比较，以查看是否有余数存在。

这个过程在225MHz的PoweMac 8500上运行，只须几秒钟的时间。执行一遍这样的运算要花大约1000步，它最适合于测试进程条。

在进程条涉及的两个角色中，仅有被填充的那个角色需要行为。另一个角色是固定不变的。

与滑动条的滑标的行为一样，这个角色要了解整个矩形的最终尺寸。为了便于确定这个尺寸，将角色放在舞台上时，已经是最终的尺寸。那样的话， on beginSprite处理程序只要在行为开始时，查看该角色矩形就可以得到整个矩形的大小了。

```
property pFullRect

on beginSprite me
  pFullRect = sprite(me.spriteNum).rect
end
```

目前唯一缺少是一个在需要时设置进程条的处理程序。它是一个自定义处理程序，在运行时由on findPrimeNumbers处理程序调用。

```
on setProgress me, currentVal, highestVal

  -- get amount filled as a value between 0 and 1
  percentFilled = float(currentVal)/float(highestVal)

  -- convert to a pixel width
  pixelRange = pFullRect.right-pFullRect.left
  x = percentFilled*pixelRange

  -- set the rect of the sprite
  r = rect(pFullRect.left, pFullRect.top, ¬
     pFullRect.left + x, pFullRect.bottom)
```

```
  sprite(me.spriteNum).rect = r
end
```

on setProgress处理程序包括两个参数。第一个是进程条的当前值，第二是进程条的最大值。它得到这两个数值并将它们相除以获得需要填充部分的比例。然后确定进程条区域的实际宽度，并确定角色应该延伸到的那个点。最后，它根据这个信息建立一个矩形并设置角色。

为了使用这个行为，我们需要从过程发生时就调用它。在这种时候，on findPrime Numbers处理程序将用到它。

```
on findPrimeNumbers
  list = []
  repeat with i = 1 to 1000
    sendSprite(sprite 2,#setProgress,i,1000)
    updateStage
    if isPrime(i) then add list, i
  end repeat
  return list
end
```

注意，因为这儿的帧不是循环，所以需要有 updateStage语句。当Director进行repeat循环的时候，如果我们不让它更新舞台，它就不会更新。

可以很容易地改变一个像这样的进程条的颜色和尺寸。如果我们要求一个更加风格化的进程条，我们可以把这个行为转换为使用三个元素的行为：左端点、右端点和可延伸的中间块。这样做要求用同一个行为控制其他角色，并很可能由参数指定。

15.7 创建图形化的弹出菜单

像复选框、单选按钮、滑动条和进程条一样，另一个可以用位图和行为创建的元素是弹出菜单。

与出现在屏幕或窗口顶部的主菜单不同，弹出菜单是出现在窗口和对话框中的小菜单。在Director的创作环境中，我们可以在剪辑室窗口和角色监察窗中看见这些例子。事实上，Director软件中的几乎每一个窗口都包括某种形式的弹出菜单。

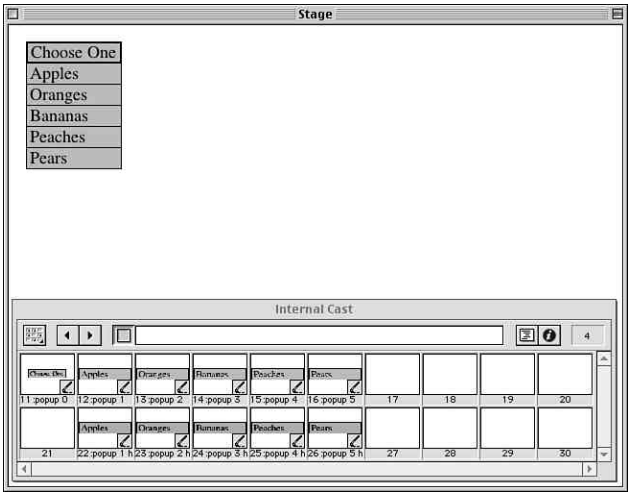

图15-10 该舞台显示了一个弹出菜单，它出现在影片创建过程中。演员表窗口显示了其中的演员和它们的突出显示状态

用行为创建影片中的弹出菜单是可能的。首先，在将要使用的演员表中创建一系列位图。图15-10显示了带有一些位图的演员表窗口。一个位图表示没被激活的弹出菜单。当点击它时，在它的下方将显示另外五个位图。这五个位图全都有一个对应的突出显示状态。

为在舞台上创建弹出菜单，需要按照在弹出菜单被按下并激活时我们想看到的效果放置角色。在用户首次进入此帧时，不要介意 Choose One图形下的清单并没有存在；行为将对这种情况进行处理。

对做这项工作的行为来说，要知道这些角色存放在哪儿。所以，此行为必须有一项属性是角色的列表。另一属性将是起初位于这些角色中的演员。这个行为本身将被赋予弹出菜单按钮，在这里是Choose One图形。

```
property pSpriteList, pMemberList, pPressed

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pSpriteList, [#comment: "Sprite List",¬
    #format: #list, #default: []]
  return list
end
```

开始时，此行为需要记录被角色使用的演员，每一个角色都是弹出菜单的一部分。这个例子并不包括弹出按钮。

```
on beginSprite me
  pMemberList = [:]
  repeat with i in pSpriteList
    addProp pMemberList, i, sprite(i).member.name
  end repeat
  hidePopup(me)
end
```

在用户看见它们前，调用 on hidePopUp处理程序将弹出的项目隐藏。达到这个目的最好的方法就是将这些角色的memberNum属性设置成0：

```
on hidePopup me
  repeat with i = 1 to pSpriteList.count
    sprite(pSpriteList[i]).memberNum = 0
  end repeat
end
```

鼠标点击处理程序看上去与前面的行为中使用的十分类似。 on mouseUp和on mouseUp Outside都调用一个自定义处理程序，此处理程序又调用 on hidePopUp和on select处理程序。

```
on mouseDown me
  pPressed = TRUE
end

on mouseUp me
  liftUp(me)
end

on mouseUpOutSide me
  liftUp(me)
end

on liftUp me
```

```
  pPressed = FALSE
  hidePopup(me)
  select(me)
end
```

在on exitFrame处理程序中，当弹出菜单被按下后，将执行许多任务。首先，它调用 on showPopup，该处理程序把角色返回到我们创建影片时它所使用的演员。然后，它用the rollover属性确定鼠标当前落在哪上角色之上，并使用pSpriteList属性的getOne属性确定是否它是一个弹出项。

注释 列表的getOne属性可以让我们找到列表中某项的位置。如果此项位置找到，它返回此项的编号，也可以被解释为TRUE。如果此项不在列表内，则返回0，即FALSE。

如果鼠标落在列表中某一角色之上，那么它就将这一角色的演员换成另外那个同名的、但名称后面带有“ hilite ”的演员(这只是行为用来判断哪个突出显示演员属于哪个普通演员的习惯用法)。我们也可以将所有突出显示演员放置在紧接着普通演员的下一个演员位置，并记录这些角色的演员编号，用a+1去找到突出显示演员，它恰是演员表中的紧接着的下一个演员。

```
on exitFrame me
  if pPressed then
    showPopup(me)
    s = the rollover
    if (pSpriteList.getOne(s)) then
      sprite(s).member = member (pMemberList.getProp(s)&& "hilite")
    end if
  end if
end
```

这个处理程序的结果是，如果 pPressed属性为真，则显示弹出项，并且光标当前所在的任一项将使用突出显示演员。图 15-11是正在进行的操作。

图15-11 弹出菜单已被选中并且光标位于第二项上

on showPopup处理程序与on hidePopup处理程序十分相似，除了在第一个处理程序中pMemberList被用于为每个角色分配正确的演员外。

```
on showPopup me
  repeat with i in pSpriteList
    sprite(i).member = member pMemberList.getProp(i)
  end repeat
end
```

最后，当用户释放鼠标按钮时， on select处理程序被调用。它不能假设光标在弹出列表的某一项上，所以必须采用与 on exitFrame处理程序相同的查看方法进行查看。

```
on select me
  s = the rollover
  if (pSpriteList.getOne(s)) then
    alert pMemberList.getProp(s)
  end if
end
```

在这种情况下，一个简单的 alert框出现，以表示已有了选项。然而，在我们的程序中，

还要设置一个全局变量，从而转到某一帧，或者执行一些其他的动作。

弹出菜单行为可以用许多种方式自定义。只要多做一点工作，我们甚至可以有一个代替Choose One的选项出现，或在使用弹出菜单时播放一些声音。

注意，剧本中没有任何代码规定只能让选项位于原始角色的正下方。为什么它们不排列在角色的右边呢？或者，出现在角色的上面呢？可能发生的情况有很多。

参见第16章“控制文本”里的16.4节“创建文本弹出菜单”，以了解制作弹出菜单的另一种方式。

15.8 图形界面元素的故障排除

■ 要理解clickLoc 和mouseLoc间的差异。第一个表示用户点击时鼠标的精确位置。第二个表示鼠标当前的位置，自从用户点击后，这个位置可能已经发生了变化。

■ 如果运算的结果是介于0～1之间的数字时(如在滑动条行为或进程条行为中的计算)，在作除法前，将参与运算的数字转换成浮点数是十分重要的。否则，作像 3/6这样的计算将返回0，而非0.5。

■ 角色的memberNum属性使我们能够仅用编号就可以设置角色的演员。它适合于引用下一个演员或类似的情况，但对其他情况就不适用了。在其他情况下，应使用 member属性，用名称设置角色的演员。

15.9 你知道吗

■ 在sendSprite命令中，“sprite”是可写可不写的。所以我们可以用 sendSprite(7, #myHandler)格式代替sendSprite(sprite 7, #myHandler)。

■ 关键词“rollover”有两种句法形式。如果光标在角色 x之上，函数 rollover(x)返回TRUE，而属性the rollover直接返回光标下的角色编码。

■ memberNum属性不允许我们将角色设置为另一个演员表库中的演员。然而，通过修改castLibNum，可以达到这个目的。

■ castNum属性已废弃，但仍然可以使用。它为第一个演员表库中的演员返回的值与memberNum的值相同，但是为其他演员表库中的演员返回的却是大得多的数值。例如，在演员表库2中的第一个演员的castNum值为131073。