第八部分　用Director创建专业性的应用程序

第27章　教 学 软 件

无论是对于初学者还是对于水平较高的使用者，运用 Director的某种单一的命令、功能或属性都是比较容易的。但是，综合运用各种句法以制作一个实用的软件却比较困难。

本章以及接下去的几章介绍的是用 Director创建典型的程序的一些例子。首先将给出每个程序的概览，然后再给出构成该程序的 Lingo。所有这些程序代码以及完整的演示影片都包含在本书所附的CD-ROM内。若要充分理解这几章，可以查看 CD-ROM内的有关影片，以便研究Lingo程序代码、剪辑室、演员表以及舞台间的协作关系。

本章包含5个典型的应用程序。第一个是一个简单的配对游戏，用户必须点击右边的词条，并把它拖拽到左边。第二个是一个绘画程序，在这里，用户有机会创作自己的艺术作品。第三个程序是一个使用Lingo的简单实例，其功能是打开或关闭像课本中的透明覆盖片一样的覆盖层角色。第四个程序是一个地理知识测验，用户必须通过点击地图来回答问题。最后一个程序模拟的是标准化考试。

27.1　制作配对游戏

配对游戏在教学CD-ROM中很常见。用户将看到两列单词、名称或词组。如果在纸上做这个游戏，则需要画直线把两列中相互关联的条目匹配起来。

例如，要求用户把发明人与其发明成果匹配起来。两列词条是这样的：

Benjamin Franklin	Cotton Gin
Thomas Edison	Telegraph
Eli Whitney	Electricity
Alfred Nobel	Light Bulb
Samuel Morse	Rocket Engine
Robert Goddard	Dynamite

这两列词条当然不是横向直接配对的，否则就太简单了。

在计算机上做这个游戏时，就不需要画直线了，而是用鼠标把某个词条拖拽到另一个词条上面。在有些情况下，词条图形的形状也会吻合，像拼图游戏一样。

图27-1是一个配对游戏程序的屏幕显示，它有两栏，每栏包含 6个词条。它们的图形效果像被折断的木板。用户的任务是拖拽右边的木板，使之与左边的相应木板配对。

这类程序需要很多相似的演员。要制作这类程序，首先要为演员们确定其逻辑名称。在本例中，左栏内的所有演员都叫做“ left X ”，其中X代表一个数字；右栏内的所有演员都叫做“ right X ”。X值是这样规定的：左、右相互配对的那对词条的 X值相同。这样，“ Benjamin Franklin ”是“ left 1 ”，而“ Electricity ”是“ right 1 ”。

左栏是固定不动的，因此它们不必带有行为，但右栏的词条需要有行为。事实上，它们

的行为就是整个游戏的规则。

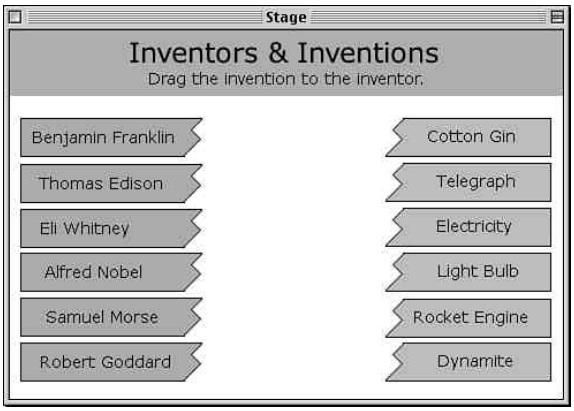

图27-1 配对游戏使用两栏词条，用户必须把右边的词条拖动到左边与之配对的词条上

行为开始于规定属性：

```
property pDrag -- is the sprite currently being dragged
property pOffset -- the cursor offset used in the drag
property pOrigLoc -- the starting location of the sprite
property pLocked -- is the sprite locked in place
```

行为开始后，它需要记录角色的初始位置，以及设定其他一些属性。原始位置是不可少的，因为如果用户拖拽了某个词条，但却没有成功地为之配对， 它还要跳回原始位置。

```
-- set all of the properties that need it
on beginSprite me
  pOrigLoc = sprite(me.spriteNum).loc
  pDrag = FALSE
  pLocked = FALSE
end
```

当用户用鼠标点击角色时，拖拽操作就开始了。但是， on mouseDown处理程序首先要检查角色的位置是否已被锁定。如果已锁定，就不允许再移动了。 on mouseDown处理程序还决定着鼠标与角色位置间的距离，使得鼠标落在角色上的任意位置都能够将其拖动，而不是只能落在角色的中心处。

```
on mouseDown me
  -- don't allow to drag if already in place
  if pLocked = TRUE then exit

  -- set to drag
  pDrag = TRUE

  -- record offset between cursor and sprite
  pOffset = sprite(me.spriteNum).loc - the mouseLoc
end
```

当用户释放鼠标键时，拖拽操作即告结束，同时也意味着行为必须检查该角色是否找到了与之匹配的角色。 on mouseUpOutside处理程序也被调用，以处理这样的情况：用户移动鼠标的速度过快，以至于当他释放鼠标键时 ，鼠标并没落在角色上。

```
-- turn off dragging and check for correct placement
```

```
on mouseUp me
  pDrag = FALSE
  checkLock(me)
end
on mouseUpOutside me
  pDrag = FALSE
  checkLock(me)
end
```

该行为在角色被拖动的时候，用 on prepare Frame处理程序为角色重新定位。这导致角色的位置与鼠标的位置尽可能地接近。但是， on exitFrame或on enterFrame处理程序实际上产生了相同的结果。

```
-- reposition sprite if being dragged
on prepareFrame me
  if pDrag then
    sprite(me.spriteNum).loc = the mouseLoc + pOffset
  end if
end
```

on checkLock处理程序带有演员的名称，并决定它应与哪个演员匹配。然后它在所有演员中寻找那个匹配的演员。当找到那个演员后，它要检查两个角色的距离是否足够近，以便可以锁定在一起。如果要求锁定，则把两个角色放在同一位置。然后调用 on checkForDone处理程序，来检查游戏是否该结束了。

由于相互匹配的角色将要被锁在一起，因此它们要有相同的坐标。在画这两幅位图时，它们的套准点(Registration point)决定了它们在舞台上显示时的相互位置关系。对于图 27-1中的情况来说，左边的图片的套准点在右边，右边的图片的套准点在左边，这样，当把它们放在舞台上的同一位置时，两个图片就像被并排锁在了一起。查看 CD-ROM中的影片，可以知道套准点的确切位置。

```
--check to see if there is a match
on checkLock me

  -- determine match member's name
  memName = sprite(me.spriteNum).member.name
  memNum = memName.word[2]
  matchName = "left"&&memNum

  -- check all the sprites
  repeat with i = 1 to the lastChannel

    -- is it the matching sprite?
    if sprite(i).member.name = matchName then

      -- is it close enough to lock in place?
      if closeEnough(me, sprite(i).loc,sprite(me.spriteNum).loc) then

        -- place sprite in exact location
        sprite(me.spriteNum).loc = sprite(i).loc

        -- set lock
        pLocked = TRUE
```

```
      -- change colors
      sprite(me.spriteNum).bgColor = rgb("#CCCCCC")
      sprite(i).bgColor = rgb("#CCCCCC")

      -- see if all are locked in place
      checkForDone(me)

      -- leave this handler
      exit
    end if
  end if
 end repeat

 -- never found a matching member close enough
 -- return to original position
 sprite(me.spriteNum).loc = pOrigLoc
end
```

下一个由on checkLock处理程序使用的处理程序决定两个图片的距离是否足够近，以便可以锁在一起。实际上它是检查二者的坐标值之差是否小于 10个像素：

```
-- determine if two locations are close
on closeEnough me, loc1, loc2
 maxdistance = 10 -- use 10 pixels as max distance
 if abs(loc1.locH - loc2.locH) < maxdistance then
  if abs(loc1.locV - loc2.locV) < maxdistance then
   -- close enough, return TRUE
   return TRUE
  end if
 end if
 return FALSE
end
```

如果两个图片已被锁在一起，这个将要被 on checkForDone处理程序调用的处理程序将返回一个TRUE：

```
-- simply report on condition of pLocked
on amIDone me
 return pLocked
end
```

amIDone消息将被送往每个角色，以检查是否所有图片都已被锁定。与该行为无关的角色将返回一个简单的VOID，其他的则返回TRUE或FALSE。若得到了FALSE，处理程序将停止工作，因为游戏还没有结束。如果从所有的角色中都没有得到 FALSE，影片则跳到“ payoff ”帧：

```
-- check all sprites to see if all are locked
on checkForDone me
 repeat with i = 1 to the lastChannel

  -- ask a sprite if it is done
  done = sendSprite(sprite i,#amIDone)

  -- sprite did not know about #amIDone
  if voidP(done) then next repeat
```

```
    -- sprite returned that it was not locked
    if done = FALSE then exit
  end repeat

  -- if they got here, then all are done
  -- go to another frame
  go to frame "payoff"
end
```

该程序所需要的唯一的其他Lingo剧本是一个简单的循环帧剧本。其余所有的工作都交给行为来完成。这意味着我们可以轻易地向游戏中添加词条。关键在于要创建一左一右两个匹配的演员，并正确地为它们命名。然后把它们放到舞台上，并把行为赋给右边的词条，我们不必关心这些演员饰演的是哪个角色。

提示 如果你想使用文本演员而不是位图演员，只需重新考虑一个问题。文本演员的套准点总是在它的左上角。要把它们配对，需要想出一个新的方法来检查它们是否已被“锁定”，并建立一个新的标准，用来把已锁定的词条定位。例如，也许你想要按角色的矩形框的右上角去锁定它，而不是它的左上角——角色的套准点的位置。

当游戏结束时，影片将移动到“payoff”帧。在这里，可以任意安排声音或动画。也可以让配好对的词条做些动作，因为我们此时已经知道如何放置它们就能让游戏结束。

另外还可以考虑声音。使用puppetSound，可以安排在词条被正确放置时发出一种声音，在词条被错误放置时发出另一种声音。

参见第15章“图形界面元素”里的15.4节“拖动角色”，以获得更多关于拖动角色的行为的信息。

27.2 实现绘画操作

为儿童制作的常用教学软件是绘画软件。这些软件大都是基于早先为70、80年代的计算机设计的简单的绘画软件。

最基本地，用户可以在屏幕上点击或拖动鼠标绘画。在本例中，用户还可以选择不同颜色和不同画刷。

用一个角色绘画实际上是非常简单的。所需要做的就是让角色跟随光标走，同时打开它的trails属性，这样当用户移动鼠标时，角色的印迹就一路留了下来。但是，这种方法有一个缺陷。如果用户移动鼠标的速度过快，角色的印迹将不能连续，导致在各个印迹间出现空隙。但我们用笔在纸上绘画时却不是这样的。因此，本程序必须很好地处理鼠标移动过快的情况，在角色的两个印迹间放置平滑的线条。

图27-2是这个影片实例。窗口的左侧有一些颜色块，当前选中色被用方框标示出来。它们的下方有一些画刷，当前画刷也用方框标出来。主要绘画区域开始是空白的，用户可以在其中绘画。

还有一个大型的行为驱动该影片，但它不是在角色上，而是在帧剧本通道内。这是有必要的，因为用户并不是通过在角色上点击来绘画的，而是在舞台上的窗口区域点击而绘画的。因此，帧剧本通道是放置该处理程序的合理位置。

不过绘画时仍需使用一个角色，因此绘画是否正在进行就是它的一个属性。有一个属性决定当前是否正在绘画，另一个属性记录上一个印迹的位置和用来绘画的演员，即画刷(brush)。

图27-2 在这个简单的绘图软件中，用户可以改变画刷及其颜色。用大一点的画刷和白颜色，可以起到橡皮擦的作用

```
property pDrawSprite -- which sprite to use
property pDraw -- drawing in progress
property pLastDrawLoc -- last spot drawn on
property pBrush -- member to use to paint

on beginSprite me
  pDrawSprite = 7 -- use sprite 7
  pLastDrawLoc = 0 -- no last drawing loc
  pBrush = sprite(pDrawSprite).member
end
```

当用户在舞台上点击时，绘画就开始了。所需要的只是把 pDraw属性设为TRUE。不过，向用户表明当前所用的画刷也是有用的。因为所有画刷都是小于 16×16的1-bit位图，所以用cursor命令可以把它们用作光标：

```
on mouseDown me
  pDraw = TRUE
  cursor([member pBrush, member pBrush])
end
```

当用户释放鼠标按钮时，绘画即告结束。不但 pDraw属性应复位，而且画刷也应当从屏幕上消失。由于用户再次绘画的位置可能与刚才完全不同， pLastDrawLoc也被复位，这样就不会发生把本次绘画的最后一笔与下次的第一笔用线连结起来的现象。光标也要复位。

```
on mouseUp me
  sprite(pDrawSprite).loc = point(-100,-100)
  pDraw = FALSE
  pLastDrawLoc = 0 -- forget last drawing loc
  cursor(0)
end
```

实际上绘画是由on exitFrame处理程序完成的。这意味着屏幕在每一帧都要被更新。由于我们希望该绘画软件的画面运行得尽可能流畅，因此应把影片的节奏调节到最大值： 999fps。

该处理程序也要检查鼠标的位置是否在特定的边界内。在此之后，它调用 on drawLine处理程序模块，把用户所画的上一点与当前点间的空隙填满：

```
on exitFrame me
  if pDraw then
```

```
  -- get current location
  curLoc = the mouseLoc

  -- restrict draw area
  if curLoc.locH < 60 then curLoc.locH = 60
  if curLoc.locH > 480 then curLoc.locH = 480
  if curLoc.locV < 35 then curLoc.locV = 35
  if curLoc.locV > 320 then curLoc.locV = 320

  -- if there is a last location
  if (pLastDrawLoc <> 0) then
    drawLine(me,pLastDrawLoc,curLoc)

    -- if not, then just draw a point
  else
    drawLine(me,curLoc,curLoc)
  end if

  -- new last location
  pLastDrawLoc = curLoc
 end if

 go to the frame
end
```

on drawLine处理程序很简单，只是包含大量的运算。它使用 distance函数去确定这两点相距有多少像素，然后按距离值除以 2得到的值进行循环，这样即每隔一点画一点。如果画刷只有1个像素大，这样画将会画出像虚线那样的线，但由于画刷都比 1个像素大，隔一点画一次得到的线与原本应画的线相比就看不出有什么不同了，但提高了绘画的速度。

在循环中每走完一步，角色的位置离鼠标的最新位置就更近一步，离鼠标的前一个位置就更远一步。updateStage命令将用来把角色的印迹放到舞台上：

```
-- this handle will draw a line of dots from one point to the next
on drawLine me, loc1, loc2
 sprite(pDrawSprite).trails = TRUE

 -- how many dots to draw
 numSteps = float(distance(me,loc1,loc2))/2+1

 -- repeat and place dots
 repeat with i = 1 to numSteps
  sprite(pDrawSprite).loc = loc1*(numSteps-float(i))/numStep¬
   + loc2* (float(i)/numSteps)
  updateStage
 end repeat
end
```

这个简单的distance函数根据两点的位置计算出其间的距离，以像素为单位。该值被用在on drawLine处理程序里，以确定在画刷的两个点之间有多少步：

```
-- calculate the distance between two pixels
on distance me, loc1, loc2
 return sqrt(power(loc1.locH-loc2.locH,2)+power(loc1.locV-loc2.locV,2))
end
```

由于这个帧剧本行为似乎控制了这部影片的功能，我们不妨把一些辅助操作也放在其中。下一个处理程序以一个颜色对象作为参数，并把绘画的角色的颜色设为那种颜色。由于画刷都是1-bit的演员，它们将采用角色的颜色，并从那一点起用这种颜色绘画。不过，需要由一个外部处理程序来调用该处理程序，以执行颜色的改变。这是附属于图 27-2中的色块的剧本。

```
-- accept a color and change to it
on changeColor me, color
  sprite(pDrawSprite).color = color
end
```

使用同样的思路，下一个处理程序从左边那些画刷的剧本中接受一个新画刷演员。除了改变该演员的编号，还需要设定pBrush属性。

```
-- accept a brush member and change to it
on changeBrush me, brush
  sprite(pDrawSprite).member = brush
  pBrush = brush
end
```

影片的主要行为就这么多。现在万事俱备，只是当用户想要改变颜色或画刷时，却没有任何反应。要实现这一点，需要给色块和画刷图标附带上很短的行为。下面是一个用于色块的行为。它把指定给那个角色的颜色传送给帧剧本中的主要行为，即 Sprite 0。它还向所有角色发出一个#change ColorSprit消息。该消息用来重新设置表示当前颜色的那个方框。

```
on mouseDown me
  color = sprite(me.spriteNum).color
  sendSprite(sprite 0, #changeColor, color)
  sendAllSprites(#changeColorSprite,me.spriteNum)
end
```

on changeColorSprite处理程序应该位于一个小型的行为里，该行为附属于包围着当前颜色的那个小方框。当 sentAllSprites消息被发出后，所有的角色都能收到它，但只有这一个使用它。这意味着该角色可以被放在任何地方，而不需要我们设置带有它的编号的全局变量。

该处理程序取得当前色块的矩形，再向四周各扩展 3个像素。色块外的方框随即被设置为这个新的矩形，使它能够包围着当前色块：

```
on changeColorSprite me , colorSprite
  sprite(me.spriteNum).rect = ¬
    sprite(colorSprite).rect + rect(-3, -3, 3, 3)
end
```

下面是一个为画刷编写的行为。它告诉主行为画刷已经改变了，而且向其他所有行为发出一个#changeBrushSprite消息，使得画刷外方框角色能识别它，并做出反应：

```
on mouseDown me
  brush = sprite(me.spriteNum).member
  sendSprite(sprite 0, #changeBrush, brush)
  sendAllSprites(#changeBrushSprite,me.spriteNum)
end
```

画刷外方框的行为与色块外边框角色的行为几乎相同：

```
on changeBrushSprite me , colorSprite
  sprite(me.spriteNum).rect = ¬
    sprite(colorSprite).rect + rect(-3, -3, 3, 3)
```

```
end
```

Clear(清除)按钮的剧本也很好写。它的任务是清除所有由画笔的笔迹产生的图画。这是用一种奇特的方法实现的：重设舞台的颜色。最好的方法是不做任何改动而设置一遍舞台的颜色。结果是所有笔迹都消失了。

```
on mouseDown
  the stageColor = the stageColor
end
```

最后，应该用一种方法阻止用户在绘画时点击标题条或工具条区域。由于帧剧本只在角色接收了mouseDown消息以后才能再接收该消息，我们所需要做的就是写一个简单的剧本，让它吃掉所有mouseDown消息，使得主体行为永远都接收不到这些消息：

```
on mouseDown
  -- no action
end
```

把这个行为赋给所有应该处于非激活状态的角色，如屏幕顶端和左侧的灰色的矩形区域。

看一看CD-ROM上的这个实例，以了解这些代码是如何结合在一起的。要做修改也是非常容易的。我们可以通过编辑画刷的位图来改变它们。改变颜色也很容易，因为我们可以改变舞台上的色块的颜色。也可以添加更多种颜色和画刷，方法是添加更多角色，并为它们附带上正确的行为。

对该程序的一个重大改进应该是添加允许用户存储他们的作品的方法，可以使用 (the stage).picture把画面放在一个演员内，并用 crop命令裁去标题部分和工具条部分。可以使用saveMovie命令把这些改动存储在影片文件里，或使用 savecastLib命令存储一个含有该图像的演员表库，并随后把它带回该程序。

参见第18章“控制位图”里的18.3节“处理位图演员”，可以获得更多有关影响位图演员的Lingo命令的信息。

参见第10章“角色和帧的属性”里的10.8.3节“轨迹”，可以了解有关笔迹背景知识。

27.3 制作透明覆盖层

还记得高中的生物课本吗？里面有很多人体的插图，其中，骨骼和各个器官都能看得见。有时我们可以掀起一些透明片，以“摘除”一些器官。可能是一个透明片上有骨骼，一个透明片上有循环系统，另一个透明片上有消化系统等等。用这种方法，我们可以了解各部分是在什么部位相联系的。

同样的透明覆盖片还可以用于其他科目。天文学的书应该有夜晚星空的插图，其中可以用一个透明片表示行星的名称，另一个画出了恒星星座等等。

要在Director中制作这种效果是非常容易的。当我们把各层分别做成不同的位图后，只需把它们按照所需要的顺序放到舞台上。然后就可以用剪辑室里位于左侧的“可见性”开关来测试它们了。

图27-3就是这样的一个画面。人体的轮廓用作背景，它上面另外有 4个角色，每个角色表示一个器官。

打开或关闭覆盖层上的角色，只需使用一个简单的行为。该行为不应附属于覆盖层，而应附属于能够激活覆盖层的按钮。在这里，按钮就是位于屏幕的左侧的文本演员。

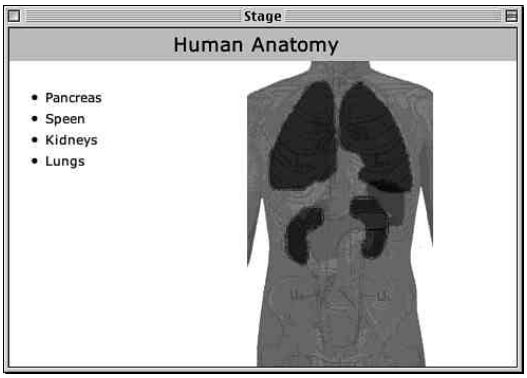

图27-3 一个简单的透明覆盖层应用程序，其中有一个背景角色、几个透明覆盖层和与每层对应的简单按钮

当用户点击某个文本演员按钮时，根据覆盖层的当前状态，覆盖层会出现或消失。按钮本身也应有所变化。在这里，每一行最左端的圆点的颜色将会变化。黑色表示本层为开，白色表示本层为关。

在行为开始时，它将试图决定圆点符号应该是黑色还是白色。它检查的是对应的覆盖层角色。因此我们可以在开始时设置角色的可见性，文本按钮也将反映出这些状态：

```
on beginSprite me
  setDot(me)
end

-- takes the first character of the text member and makes
-- it either black or white to reflect the state
on setDot me
  if sprite(me.spriteNum+1).visible = TRUE then
    sprite(me.spriteNum).member.char[1].color = rgb("#000000")
  else
    sprite(me.spriteNum).member.char[1].color = rgb("#FFFFFF")
  end if
end
```

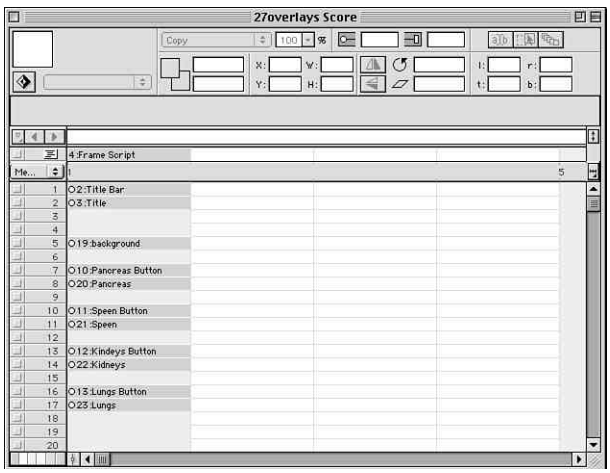

图27-4 从剪辑室可以看出，每个覆盖层角色位于与之对应的按钮角色的下一个通道里。这使得我们很容易搞清楚按钮影响着哪个角色

请注意，这个行为假设下一个角色 (me.spriteNum+1)是覆盖层角色。有了这个假设，就可以把剪辑室设置成如图 27-4所示的样子，这样我们可以避免使用一些行为参数手工地识别覆盖层角色。

行为的最后一部分内容决定角色是否可见。由于 on setDot处理程序已经能够根据覆盖层的可见性改变按钮的状态，因此在这里就不必写有关这方面的代码了。

```
-- will change the visible property of the next sprite
on mouseUp me
  sprite(me.spriteNum+1).visible = ¬
     not sprite(me.spriteNum+1).visible
  setDot(me)
end
```

这个行为由于使用了文本演员按钮而变得简单了。我们也可以用位图替换它们。在那种情况下，将需要代表开和关状态的不同演员。第 15章介绍的复选框行为可以经修改后用在这里。

另外一个要考虑的问题是覆盖层角色的油墨。用 Copy油墨是不行的，因为它将用白色覆盖住该角色的矩形下面的任何东西。很明显地，应选用 Background Transparent 或Matte油墨。它们会保持覆盖层角色的当前颜色，并使白色像素变为透明。在前面的例子里使用了Transparent油墨。它可以使边缘较光滑，但却会改变覆盖层的颜色。

提示 值得考虑的一种技巧是使用带 alpha通道的32-bit演员。那样我们就可以定义图像的透明程度了。

参见第15章的15.2节“ 使用复选框 ”，以了解复选框行为的大体情况。

27.4 制作地理知识测验

尽管学校的测验和考试的传统方法是用铅笔和纸，每道题占据着试卷的不同位置，但计算机测验更有交互性。实际上，地理知识测验可以显示一个交互式的地图，当学生用鼠标在上面掠过和点击时，它会有所反应。

图27-5就是这样一个测验。它是一幅美国地图 。实际上的影片是由 51个角色构成的。有一个大的轮廓地图，它覆盖了所有内容，并且永远可见。它下面是 50个独立的位图角色，每个角色代表一个州。

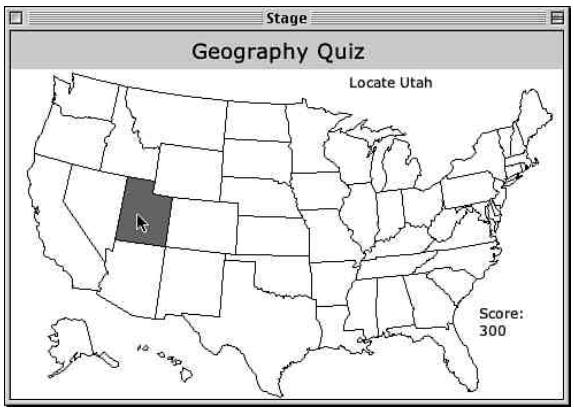

图27-5 一个地理知识测验。用户的光标掠过哪个州，哪个州就会亮起来，因此他就会知道他将要选择什么

当用户把光标在地图上移动时，光标下面的那个州会变亮。它实际上只是由白色变成了蓝色。完成这个任务的行为是非常简单的。它用 on mouseEnter处理器表示鼠标已进入了某个州，然后改变它的颜色。它也把该州的名称 —— 实际上是该角色的演员的名称 —— 放在文本演员里。它就是测验的答案，将用在随后的总结性屏幕里。实际上，体现着每个州的名称的文本演员并不出现在测验的帧里，只有当学生做完测验后才会显示出来。

```
property pState

-- get the state name
on beginSprite me
  pState = sprite(me.spriteNum).member.name
end

on mouseEnter me
  -- on rollover, change state color
  sprite(me.spriteNum).color = rgb("#336699")

  -- set the text member
  member("State Name").text = pState
end

on mouseLeave me
  -- change state color back to white
  sprite(me.spriteNum).color = rgb("#FFFFFF")

  -- reset text member
  member("State Name").text = " "
end

on mouseUp me
  -- see if answer is right
  answerQuestion(pState)
end
```

该行为需要赋给全部 50个州角色。当某个州角色被点击后，这个州角色的行为还要调用影片的on answerQuestion处理程序。要确保处于州角色上面的轮廓地图没有附带会吃掉这些mouseUp消息的行为，并永远不让这些行为接收它们。

测验的执行实际上是由影片剧本完成的。它使用几个全局变量，负责所有的考题、用户所需要回答的考题的数量和用户的得分。记分的方法是：答对一道题得 100分；如果某题答错了，用户虽然还有机会再回答，但只能得到一半的分。因此如果用户猜错了一次，但在第二次就答对了，就得 50分；第三次才答对，只得 25分；依此类推。全局变量 gPoints负责记录每个正确的答案值多少分。对于每个问题，它的初值都是 100分，每当用户答错一次，它就会被2除一次。

```
global gQuestionNum -- which question is being asked
global gQuestions -- list with all the questions
global gScore -- keep track of score
global gPoints -- keep track of potential points to be scored
```

在影片的开始，on initQuiz被调用，来确定要问哪些问题。然后，其他一些全局变量被复位。on showScore处理程序把得分放在一个文本演员里，初始分数为 0分。最后，on ask Question提出第一个问题，于是拉开了测验的帷幕。

```
on startMovie
  initQuiz
  gQuestionNum = 1
  gScore = 0
  showScore
  askQuestion
end
```

on initQuiz处理程序的任务是随机找 10个州。它先读取 States演员表(这里有所有州的位图)里的演员的名称，得到一个州名的列表。得到这个列表后，它随机选择其中的 10个州。每次选一个，并要检查这一个是否已被选作测验题。这个处理程序得到的结果是 gQuestions列表，其中有10个互不重复的州名。

```
-- this handler will come up with 10 random and unique states
-- to make up the quiz
on initQuiz

  -- get a list of all states from the cast library member names
  listOfStates = []
  repeat with i = 1 to the number of members of castLib "States"
    add listOfStates, member(i, "States").name
  end repeat

  -- add 10 random names to quiz
  gQuestions = []
  repeat with i = 1 to 10

    repeat while TRUE

      -- get a random state
      r = random(listOfStates.count)
      state = listOfStates[r]

      -- see if the state is already in quiz
      if getOne(gQuestions,state) then next repeat

      -- add state, go on to add next
      add gQuestions, state
      exit repeat

    end repeat
  end repeat
end
```

下一个处理程序把问题的文本放在舞台上的一个文本演员里。它也把 gPoints复位为100。

```
on askQuestion
  -- set text member
  member("Question").text = "Locate"&&gQuestions[gQuestionNum]

  -- set potential points
  gPoints = 100 -- potential points to earn
end
```

当用户点击某一个州时，下面的处理程序将被调用，以查看这个州是否正确。它将被点击的演员的名称与gQuestions列表里与这个问题的编号相对应的名称进行比较。如果二者不同，则显

示一个警告框，向用户发一条信息。你也许愿意用某种声音或其他方法来表明该答案是错误的。

如果二者相同，gScore将会变化，新的分数将显示出来，并发出一声系统警告声。这里，你可能想再次尝试用 puppetSound命令制作一种自定义的声音。问题编号被加 1。如果10个问题全都回答完毕，影片将跳到下一帧。否则将提问下一个问题。

```
-- this handler will check to see if a state name matches
-- the expected answer
on answerQuestion state
  -- make sure we are not done
  if gQuestionNum > gQuestions.count then exit

  if state <> gQuestions[gQuestionNum] then
    alert "Wrong. Try Again. "

    -- divide potential points in half when a wrong answer
    gPoints = gPoints/2

  else

    -- add to score
    gScore = gScore + gPoints
    showScore
    beep() -- replace with better sound

    -- next question
    gQuestionNum = gQuestionNum + 1

    -- are we done?
    if gQuestionNum > gQuestions.count then
      member("Question").text = "All done! Use the mouse to explore. "
      go to frame "Done"
    else
      askQuestion
    end if
  end if
end
```

最后一个处理程序把当前得分放在舞台上的一个文本演员里：

```
-- this handler places the current score in a member
on showScore
  member("Score").text = "Score: "&RETURN&gScore
end
```

当游戏结束时，影片跳到 Done帧。它的内容是任意的，例如可以是总结性的一帧。不过在本例中，使用的是测验里所使用的地图，只不过这一次用户可以随意探索这幅地图了。州的行为已经把该州的名称放到了一个文本演员里。该演员会出现在 Done帧里，并且附带有另一个行为，以便能跟随着光标的变化而变化。

```
on prepareFrame me
  -- determine sprite width
  textwidth = sprite(me.spriteNum).width

  -- set the location of the sprite to be just above the cursor
  sprite(me.spriteNum).loc = the mouseLoc+ ┐
```

```
    point(-textwidth/2,-16)
end
```

这个Done帧可以是一个总结，但也可以是一个独立的程序。它不仅仅在一个演员内显示州名，而且还在屏幕上的其他地方显示该州的相关信息。这些信息可以来自另一个演员表库里的文本演员或域。我们甚至可以在 on mouseUp处理程序里让影片走到关于某州的特定的帧里去，或使用gotoNetPage命令走到某一个Web网页。

参见第15章里的15.1节“ 创建显示掠过 ”，可以获得更多有关掠过行为的信息。

27.5 制作标准化考试

尽管计算机地理知识测验和配对游戏比传统的纸 -笔考试的交互性更强，但有时我们想让用户做的唯一一件事情就是回答问题。但纸 -笔考试的很多特点都难于用计算机再现出来。

一次考试通常有很多道题，一般无法在一张纸上印出所有这些题目。可以想象，试图用计算机的一屏画面显示出一次考试的全部试题也是不现实的，而每屏一道题却比较合适。当然，这带来了一个问题，即如何来回浏览各个问题。因为在纸 -笔考试里，应试者可以先跳过某些题，最后再返回来做这些题。

图27-6显示了这种标准化考试的一屏画面。它与 SAT、GRE或ACT考试很像。问题位于屏幕的中心，是整个画面的焦点。它的下面是几个选择答案。屏幕的顶部和左侧是为标题和一些浏览按钮而保留。

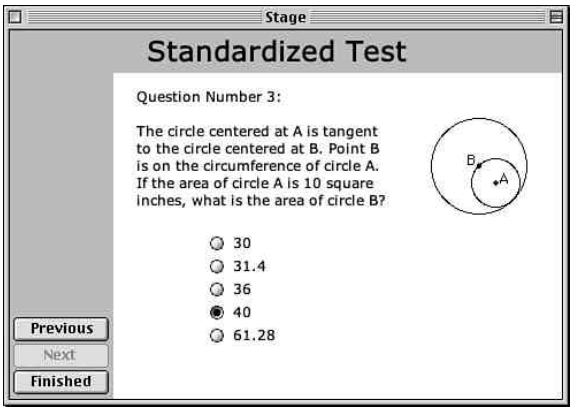

图27-6 标准化考试程序的一屏画面

要制作这样一个考试，应借助于前几章曾介绍过的两个行为。一个是第 15章的单选按钮行为。该行为用于选择答案左边的那排小单选按钮。浏览按钮使用的完全是第 14章“ 创建行为 ”里的按钮行为。

同地理知识测验一样，这个程序也需要一个影片剧本，其中含有所有重要的处理程序。该影片剧本只需要一个全局变量：用户已经选择的答案。影片在开始时先把这个全局变量复位，然后前进到“ question 1 ”帧：

```
global gAnswerList

on startTest
  gAnswerList = []
```

```
  go to frame "question 1"
end
```

影片依赖于帧标签来了解用户现在正在做哪一道题。因此“ question 1 ”帧是第一题，“ question 2 ”帧是第二题。

屏幕左侧的三个按钮Next、Previous和Finished都带有完整的按钮行为。CD-ROM里的影片实例里有按钮的“正常”状态和“按下”状态。这些行为让每个按钮分别执行on nextQuestion、on previousQuestion和on finishedTest影片处理程序。

on nextQuestion和on previousQuestion处理程序调用on recordAnswer处理程序，后者把当前被选中的按钮存储在一个全局变量里。然后，它使用 go next或go previous走到前一个或下一个问题，并且，如果用户已经回答过这个问题了，则使用 on setAnswer设置单选按钮。

```
on nextQuestion
  recordAnswer
  go next
  setAnswer
end

on previousQuestion
  recordAnswer
  go previous
  setAnswer
end
```

要记录答案，程序要从帧标签那里得到问题的编号。然后，它从单选按钮行为里的 on selected处理程序里得到单选按钮的状态。由于单选按钮在角色 11～15里，从其中减去10即得到1～5。接着，把它记录在全局变量gAnswerList里。

```
on recordAnswer
  -- get question number from frame label
  q = word 2 of the frameLabel
  q = value(q)

  -- get answer from radio buttons
  a = sendSprite(sprite 11, #selected)
  a = a - 10

  -- remember answer
  setAt gAnswerList, q, a
end
```

如果用户回答了一个问题后继续前进，后来又返回到已经回答过的题目里来，我们需要显示用户已经做出的选择。on setAnswer负责完成这个任务。它在gAnswerList里查找该问题的当前值，并相应地设置单选按钮。

```
on setAnswer
  -- get question number from frame label
  q = word 2 of the frameLabel
  q = value(q)

  -- get answer from radio buttons
  if gAnswerList.count >= q then
    a = gAnswerList[q]
    sendSprite(sprite (a+10), #turnMeOn)
```

```
  end if
end
```

当用户点击Finished按钮时，就被带到finished帧。在此之前，程序用on computeResults给用户算分。这个处理程序要依赖于一个名为 Correct Answers的域。在这个域里，一行接一行地按编号列着正确的答案。把其中的答案与 gAnswerList里的答案逐个进行比较，随后计算出总分。其结果放在一个在finished帧里能够看到的文本演员里。

```
on finishedTest
  recordAnswer
  computeResults
  go to frame "finished"
end

on computeResults

  -- get list of correct answers
  correct = member("Correct Answers").text

  -- find the number that matches up correctly
  numright = 0
  repeat with i = 1 to correct.line.count
    if value(correct.line[i] = gAnswerList[i]) then
      numright = numright + 1
    end if
  end repeat

  -- put results in Results member
  member("Results").text = "Test complete. "&RETURN&RETURN&¬
    "You got"&&numright&&¬
    "correct out of"&&correct.line.count&"."
end
```

余下的工作是设置单选按钮和浏览按钮的行为。图 27-7是一个带有3个问题的考试的剪辑室。在起始帧里有一个 Begin按钮，它执行on startText处理程序。随后是三个带有问题的帧。对于每一道题来说，单选按钮都在角色 11～15里。因此，每一个单选按钮的行为需要体现这一点。用行为监察窗来查看这些行为属性。

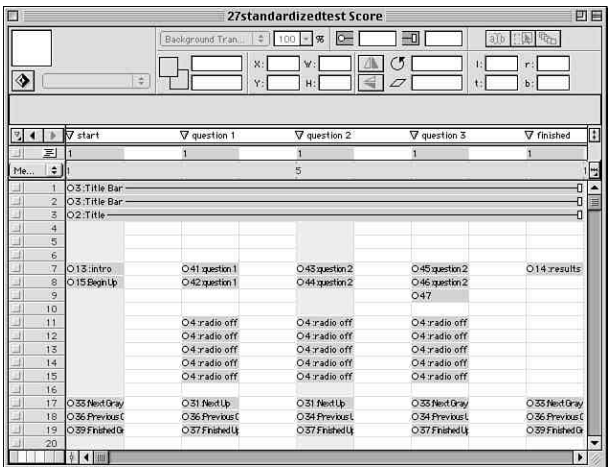

图27-7 剪辑室表示的是一个含有三道题的考试

浏览按钮还有一种淡化的状态，在这些按钮未被激活时要使用。例如， Previous按钮在第一个问题里未被激活， Next按钮在最后一道题里未被激活。这些非激活状态的按钮没有附带行为，而激活的按钮则调用 on nextQuestion、on previousQuestion和on finishedTest处理程序里的某一个。

图27-6表明屏幕里除了文本外，还可以有其他元素。有些问题只是文本，而有些问题可以带有位图、矢量图形甚至 Flash演员等附加素材。我们甚至可以在这些帧里放置影片片断，这在纸-笔考试里是不能实现的！

我们可以做的另一个改进是添加一种复杂的报告分数的方式。可以利用列表里的信息告诉应试者哪些题做对了，哪些题做错了。甚至可以让他再次浏览那些问题，并告诉他解释正确答案。

参见第14章里的14.5节“ 完整的按钮行为 ”，可以获得更多有关按钮行为的信息。

参见第15章里的15.3节“ 使用单选按钮 ”，可以获得更多有关单选按钮行为的信息。

27.6 教学软件的故障排除

■ 全局变量常用于在测验和考试里记录答案。要使软件能正确地运行，可以在消息窗口里使用clearGlobals，以保证有一个正确的开始。甚至应当把这个命令放到 on startMovie处理程序里。

■ 在地理知识测验的程序里，应当确认各个州角色使用的是 Matte油墨，且附带有行为。

■ 在前面提到的所有程序里，在每帧的 on exitFrame剧本里都应当放置一个简单的 go to the frame。我们经常容易忘记这一点。

■ 不要忘记要尽可能多地测试教学软件。学生们也许会点击屏幕上的其他区域，或进行其他我们未曾想到的操作。要使程序能够应付错误的操作。

27.7 你知道吗

■ 我们可以把所有这些教学软件组合成一个大型软件。这个软件里有配对游戏、地理知识测验、透明覆盖层游戏和用户能够在其中绘画的创作程序等。

■ 我们可以把一个考试的问题和答案做成独立的文本文件。在程序的开始，程序从磁盘或因特网上读这些文本文件。每个问题在文件里可以独占一行 ("What is the fastest animal?;Duck,Pig,Cheetah;3")。这样，不会编程序的人也可以制作考试软件了。

■ 可以把地理知识测验换成其他内容。可以使用元素周期表、人体的解剖示意图甚至是问题及其选择答案。不必把测验的内容局限在地理知识，而只要抓住这个程序的“ 感觉 ”就可以了。