第32章 游 戏

也许Director的主要用途是制作游戏。游戏的用途很广，可以娱乐，可以教学。我们甚至可以用它来阐明论点，或只是用来使信息更有趣。

Director作品的创作者们可能会非常喜欢某一类游戏，因为它们制作简单，而且能用于多种场合。配对游戏就是一个很好的例子。

有一些游戏用Director制作起来是非常困难的，但由于它们是非常流行的那类游戏，因此Director创作者们也学会了制作它们。纸牌游戏，如blackjack (二十一点)，就是这类游戏之一。本章将介绍如何制作各种类型的游戏。

32.1 制作配对游戏

这个配对游戏的基本目标是把屏幕上成对的项目正确地连接起来。每个项目都是隐藏起来的，也就是像扑克牌一样，面朝下扣着。如果成功了，这一对项目就被从屏幕上拿走。如果两个项目不匹配，它们又会被翻过去。这个游戏是用来锻炼记忆力的，在把每个项目翻开时应记住它们的位置。

要制作这样一个游戏，首先需要一系列位图，以表示这些项目。图 32-1是带有一系列位图的演员表库，其中含有18个元素。由于每个元素将要用在一对角色里，因此舞台上将有 36个角色，排成6×6的方格结构。

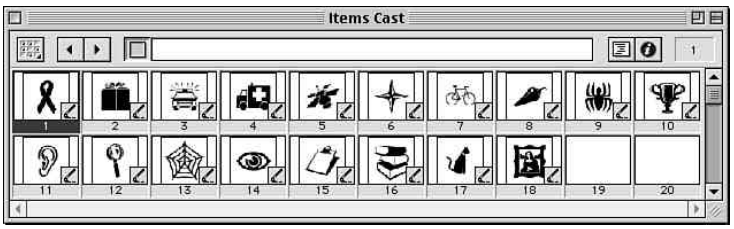

图32-1 演员表窗口显示了一个演员表库，其中有18个将要用于配对游戏的元素

有了这些位图之后，还需要一个位图，来表现角色面朝下的情形。这里使用一个尺寸相同，颜色为纯黑的位图，把它命名为Blank。

用一个影片剧本来控制整个游戏。需要两个全局变量。第一个全局变量包含一个列表，其中有用于角色的全部元素；另一个全局变量里含有面朝上的角色的编号。影片刚开始时，随机地编排元素在角色里的位置。

```
global gItemList, gSelected

on startMovie
  -- randomize items
  gItemList = createList()

  gSelected = 0
end
```

当然，元素要被放在舞台上的随机的角色里，否则每次游戏的内容就都相同了。首先，每个元素被先后两次添加到列表里，得到由双份元素组成的列表。然后，把该列表的顺序打乱，得到一个新列表。

```
on createList
  -- create ordered list
  templist = []
  repeat with i = 1 to 18
    add templist, i
    add templist, i
  end repeat

  -- shuffle list
  list = []
  repeat while templist.count > 0
    r = random(templist.count)
    add list, templist[r]
    deleteAt templist, r
  end repeat

  return list
end
```

舞台上的所有角色都应显示Blank演员，因此无需对角色进行任何设置。影片的剪辑室把它们都设为Blank演员，它们将一直保持这个状态，直至用户点击它们为止。

当用户点击某个图像时， on clickItem处理程序被调用。用户所点击的角色的编号被传给这个处理程序。影片把36个元素放在角色11至角色46里。要在gItemList全局变量里得到每个元素的位置，需要从这个数字里减去10，得到1至36之间的一个数字。

当某个角色被点击时， Blank演员就换成了与列表中的项目对应的那个演员。然后该处理程序检查这是否为第一个被选中的项目。如果是，它就把 gSelected全局变量设为该角色的编号；如果不是，则比较最近一次被点击的角色和当前角色的元素。如果二者相同，它们就被拿走；如果不同，前一个角色被翻回成Blank，当前角色被存储在gSelected全局变量里。

```
-- this handler is called by the item sprites when the user clicks
-- 10 is subtracted from the sprite number because the items
-- start with sprite 11
on clickItem sNum
  -- take control of sprite and turn over, use members from "items" cast
  puppetSprite sNum, TRUE
  sprite(sNum).member = member(gItemList[sNum-10], "Items")

  if gSelected = 0 then
    -- first item turned over
    gSelected = sNum

  else if gSelected = sNum then
    -- user clicked on selected item
    -- do nothing

  else if gItemList[sNum-10] = gItemList[gSelected-10] then
    -- user clicked on matching item
    -- make both go away
    sprite(sNum).memberNum = 0
```

```
    sprite(gSelected).memberNum = 0

    -- set items in list to 0
    gItemList[sNum-10] = 0
    gItemList[gSelected-10] = 0

    -- reset selection
    gSelected = 0

    -- check for end of game
    if checkForDone() then
      go to frame "done"
    end if

  else
    -- user clicked on wrong item
    -- turn last item back over
    sprite(gSelected).member = member("Blank")

    -- item clicked in now one selected
    gSelected = sNum
  end if
end
```

如果配成了一对，on clickItem处理程序还将调用on checkForDone处理程序。这个处理程序检查所有角色的演员编号，如果它们都已被设为0，游戏就该结束了。这时，影片将跳到另一帧。

```
on checkForDone
  repeat with i in gItemList
    -- found an item still there
    if gItemList <> 0 then return FALSE
  end repeat

  -- all were 0, so game is over
  return TRUE
end
```

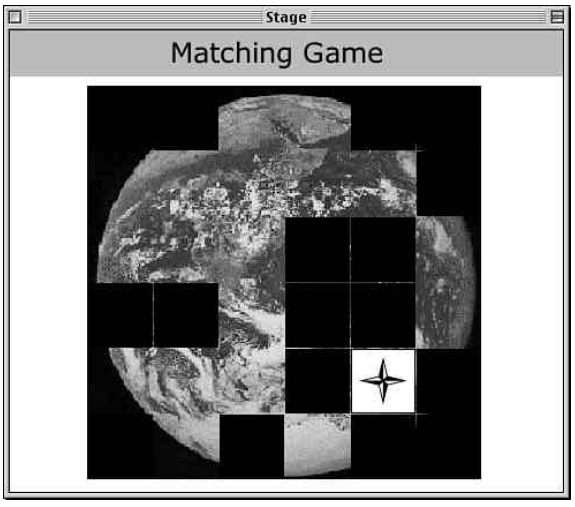

图32-2 正在进行的配对游戏。随着用户不断找到成对的元素，一幅图像便显露出来

图32-2是正在进行的游戏。通常，在游戏的背后放着一幅图像，这样，当用户挑出了全部元素后，这幅图像就显露出来了。由于用于配对的角色从第 11号开始，因此有足够的角色用来放置这类图像。

32.2 制作滑动的拼图游戏

另外一种在玩的过程中可以揭示出一幅图像的游戏是滑动拼图游戏，它也在屏幕上使用一些方块。这一次，所有元素都显示出来，但顺序却是乱的。此外，还缺少一个图块。用户可以把与空格毗邻的图块移到空格里。用户可以就这样移来移去，直至把图像的顺序排好。图32-3就是这样一个游戏。

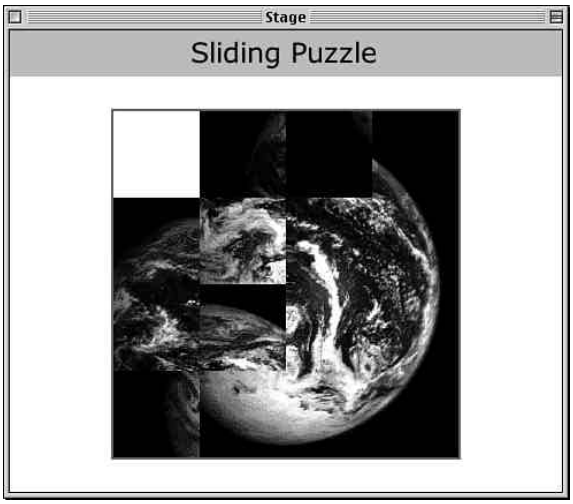

图32-3 已经部分完成的15块滑动拼图游戏

这个拼图游戏是由 15个不同的演员制作的，每个演员是一个图块。由于它们将在舞台上被随机摆放，因此每个演员都需要知道自己的正确位置在哪里。为了让它们能够知道这一点，每个演员的名称里都包含着这个图块在拼图游戏里的横、纵位置值。请参看图 32-4里的例子。

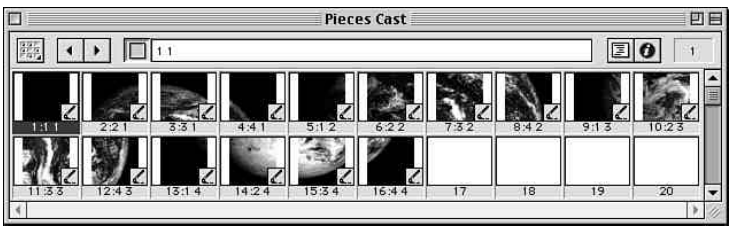

图32-4 带有16个拼图图块的演员表。每个演员的名字包含着它在正确答案里的横、纵坐标

同配对游戏一样，这个游戏的主要内容也是影片剧本。它只要监控一条信息：拼图游戏里的空白方块的位置。它把它存储为两个数字，即该方块横、纵位置。影片开始时，这个位置被设为1,1。它也使用了puppetSprite命令，对图块角色进行 Lingo控制。这使得影片剧本能够控制各个角色，而不是用行为逐个控制角色。

```
global gOpenSpotX, gOpenSpotY
```

```
on startMovie
  gOpenSpotX = 1
  gOpenSpotY = 1
  repeat with i = 11 to 25
    puppetSprite i, TRUE
  end repeat
  scramble
end
```

乱序的效果与配对游戏里的乱序效果使用的是相似的方法，但 on scramble处理程序采取的是一种看起来更有趣的方法。它随机选择两个角色，然后把它们相互交换。由于这是一种有趣的效果，于是它使用了updateStage命令，以便让用户看到互换的过程。

```
on scramble
  repeat with i = 1 to 100
    -- pick two sprites at random and switch them
    s1 = random(15)+10 -- sprites between 11 and 25
    s2 = random(15)+10
    loc = sprite(s1).loc
    sprite(s1).loc = sprite(s2).loc
    sprite(s2).loc = loc
    updateStage
  end repeat
end
```

当某个图块角色被点击时， on clickOnPiece处理程序被调用。该角色的编号就传给了它。角色的位置被75除，得到那个图块的位置。75(像素)是相邻图块的间距。还要先从横向和纵向位置里分别减去90和70，因为左上角的图块的位置是90,70。

得到该图块的位置后，要检查它的四周，看看有没有空白方块。如果有，则调用 on slide处理程序。

```
on clickOnPiece sNum
  h = sprite(sNum).locH
  v = sprite(sNum).locV
  x =  (h-90)/75+1
  y = (v-70)/75+1

  -- check all surrounding spots
  if (gOpenSpotX = x-1) and (gOpenSpotY = y) then
    slide(sNum,-1,0)
  else if (gOpenSpotX = x+1) and (gOpenSpotY = y) then
    slide(sNum,1,0)
  else if (gOpenSpotX = x) and (gOpenSpotY = y-1) then
    slide(sNum,0,-1)
  else if (gOpenSpotX = x) and (gOpenSpotY = y+1) then
    slide(sNum,0,1)
  end if
end
```

on slide处理程序把当前图块向上、下、左或右移动 75像素，移动的动作被分为20步。添加一些步数可以降低移动的速度，减少一些步数可以提高移动的速度。

```
on slide sNum, dx, dy
  step = 20
```

```
  x1 = sprite(sNum).locH
  y1 = sprite(sNum).locV
  x2 = x1+(dx*75)
  y2 = y1+(dy*75)
  repeat with i = 0 to step
    p = float(i)/step
    sprite(sNum).locH = (p*x2)+((1.0-p)*x1)
    sprite(sNum).locV = (p*y2)+((1.0-p)*y1)
    updateStage
  end repeat
  gOpenSpotX = gOpenSpotX-dx
  gOpenSpotY = gOpenSpotY-dy
  if checkDone() then alert "You got it! "
end
```

移动结束后，on checkDone处理程序把所有角色的位置与其各自演员的名称所表示的位置进行比较。只要发现一例不匹配，就表明拼图游戏还没有结束；如果全都匹配，表明用户已经完成了拼图游戏。

```
on checkDone
  repeat with i = 11 to 25
    x = sprite(i).locH
    y = sprite(i).locV
    x = (x-90)/75+1
    y = (y-70)/75+1
    name = sprite(i).member.name
    if (value(name.word[1]) <> x) or (value(name.word[2]) <> y) then
      return FALSE
    end if
  end repeat
  return TRUE
end
```

这种拼图游戏的图块可以更多，也可以更少。我们还可以改变图块的尺寸及其间距，只要在程序里把有关数字修改一下就可以了。另外一个任务或许就是把全部处理程序进行正规的处理，使它们不必依赖于75、90和70等硬代码，而是使用演员的宽度以及图块位置的最小值。

32.3 制作下落物体的游戏

Director作品的创作者们经常制作的另一种游戏是下落物体的游戏。用户可以控制舞台底部的某个物体，如一个棒球手套或一个卡通人物。不断有物体从舞台顶部落下来。用户必须抓住这些物体，又要避开其他物体。图32-5就是这样一个游戏。

这个游戏由几个行为和一个短的影片剧本构成。控制手套的运动的行为是很简单的，它使该角色的横向位置与鼠标的横向位置相同。

```
on exitFrame me
  -- move glove with mouse
  sprite(me.spriteNum).locH = the mouseH
end
```

下落物体的行为稍微复杂一点。它使用 pMode属性来决定角色是正在下落还是正在等待指令。它还用pSpeed属性指定物体具有不同的下落速度。

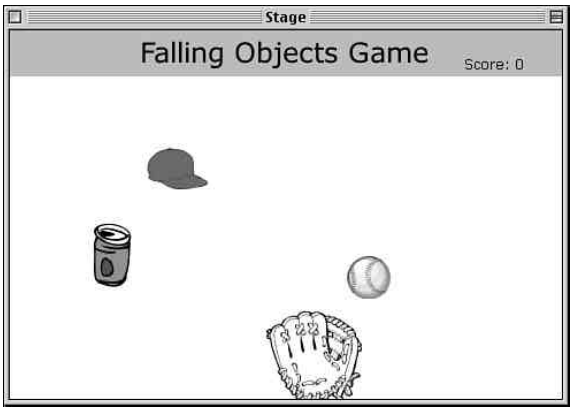

图32-5 带有棒球道具的下落物体游戏。用户可以左右移动手套，必须抓住棒球又要避开其他物体

```
property pMode, pSpeed

on beginSprite me
  pMode = #none
  pSpeed = 10
end
```

在on exitFrame处理程序里，要查看 pMode。如果它被设为#fall，对象的纵向位置就将发生变化。然后它检查该角色与手套 (即角色5)是否相交。

提示 Intersects是一个不同寻常的Lingo。它并不是一个函数，但却可以比较两个对象。其句法是这样的：if sprite a intersects b then...。sprite这个词出现在第一个角色编号之前，却不出现在第二个编号之前。如果这两个角色的矩形相交，就返回 TRUE，否则返回 FALSE。

如果两个角色相交，则进行另一个测试，即判断物体的中心是否靠近了手套的中心。所谓“中心”，就是套准点。手套的套准点被向下移了一点，代表手套的手心。

如果这个测试结果为真，则表明物体被接住。如果接住的是棒球就加分，如果不是，就减分。无论是哪种，角色都将被重新设置。

```
on exitFrame me
  if pMode = #fall then
    -- move object down
    sprite(me.spriteNum).locV = sprite(me.spriteNum).locV + pSpeed

    -- see if it intersects the glove
    if sprite me.SpriteNum intersects 5 then

      -- see if it is close to the center of the glove
      if distance(me,sprite(me.spriteNum).loc,sprite(5).loc) < 20 then

        -- add points
        if sprite(me.spriteNum).member.name = "Baseball" then
          addPoint
        else
          subtractPoint
```

```
      end if

      -- reset object sprite
      pMode = #none
      sprite(me.spriteNum).locV = -100
    end if

  else if sprite(me.spriteNum).locV > 400 then
    -- went past bottom, reset
    pMode = #none
    sprite(me.spriteNum).locV = -100
  end if
 end if
end

-- utility handler
on distance me, p1, p2
 return sqrt(power(p1.locH-p2.locH,2)+power(p1.locV-p2.locV,2))
end
```

由于所有角色的 pMode开始时都是 #none，因此需要采取某种方法把它变成 #fall。on startFall处理程序就完成这个任务。它被帧剧本调用。它指定一个速度值和一个演员。如果那个角色已经在下落，消息则被传给下一个角色。

```
on startFall me, speed, type
 if pMode <> #none then
   -- this sprite being used, go to next
   sendSprite(sprite(me.spriteNum+1), #startFall, speed, type)

 else
   -- set member, location, speed and mode
   sprite(me.spriteNum).member = member(type)
   sprite(me.spriteNum).loc = point(40+random(400),-20)
   pSpeed = speed
   pMode = #fall
 end if
end
```

在on startFall处理程序里，角色的横向位置是随机设置的。由于本例中的舞台的宽度是480像素，于是选择1~400间的一个随机数，再加上 40，得到41~440间的一个随机数，这样物体不会太靠边。

帧剧本负责随机地让物体下落。有 1/10的概率是剧本让第一个角色下落。如果该角色正忙，则让下一个角色下落，依此类推。

该剧本还随机地选择一个物体，该物体可以是棒球演员，也可是另外三种物体之一。它还选择一个5~16的数字为速度值，请看程序：

```
on exitFrame

 -- drop object on 10% chance
 if random(10) = 1 then

   -- decide what type of object
   r = random(4)
   if r = 4 then type = "Baseball"
```

```
    else type = "Object"&&r

    -- send message to sprite(s)
    sendSprite(sprite 8, #startFall, 5+random(10), type)
  end if

  go to the frame
end
```

影片剧本的作用只是记录得分。在影片开始时，它把得分复位为 0，然后管理得分的增加或减少。它还要保证得分不能低于 0，因为每当抓住的不是棒球，而是另外三种物体之一时，将要减去1分。

```
global gScore

on startMovie
  gScore = 0
  showScore
end

on addPoint
  gScore = gScore + 1
  showScore
end

on subtractPoint
  gScore = gScore - 1
  if gScore < 0 then gScore = 0
  showScore
end

on showScore
  member("Score").text = "Score: "&&gScore
end
```

还有一个问题是这个游戏还没有一个结尾。有很多种可能的结尾方法：可以当用户的得分到达一定值时结束；可以对下落物体计数，当下落了一定数量的物体后即结束；可以记录正确抓住棒球的次数和错误的次数，当错误的次数到达一定值时即结束；还可以设定一个计时器，让用户在特定的时间里尽可能多地抓住棒球。

32.4 模仿射击靶场

另一个很容易用Director制作的常见游戏是射击靶场。对象在舞台上出现和移动，而用户则试图射中它们。

图32-6是一个典型的设置。那些矩形表示箱子或其他障碍物。鸭子从里面冒出来，随后又落回去，隐藏起来。光标变成了十字线，用户可以点击鼠标来射击。如果射中了鸭子，将会记录得分。

这个影片主要由一个行为组成。该行为控制鸭子的行动。这些鸭子以角色的身份出现在舞台上，并可以隐藏在矩形角色背后。

该行为用pMode来确定鸭子是在上升、下降还是静止的。如果鸭子是静止的， pMode则被设为#down。然后，在30次里有一次，鸭子开始冒出来，其速度为 1~3间的某个值。

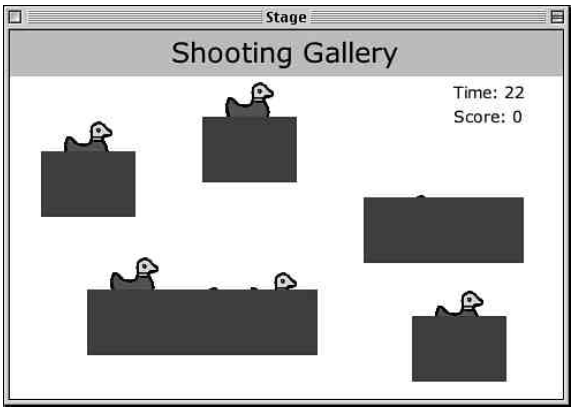

图32-6 射击游戏让用户射击木质的鸭子

如果pMode是#rise，鸭子则上升；如果pMode是#fall，鸭子则下降。当鸭子上升时，该行为检查它是否已上升了30像素。如果是，则把pMode改为#fall。

第四种模式是#hit，表示用户刚击中了鸭子。这时，它将复位至其初始位置，并停止运动。

```
property pMode, pOrigLocV, pSpeed

on beginSprite me
  pMode = #down
  pOrigLocV = sprite(me.spriteNum).locV
end

on exitFrame me
  if pMode = #down then
    -- see if it is time to pop up
    if random(30) = 1 then
      pMode = #rise
      pSpeed = random(3) -- random speed
    end if

  else if pMode = #rise then
    -- move duck up
    sprite(me.spriteNum).locV = sprite(me.spriteNum).locV - pSpeed

    -- see if at highest point
    if sprite(me.spriteNum).locV < pOrigLocV-30 then
      pMode = #drop
    end if

  else if pMode = #drop then
    -- move duck down
    sprite(me.spriteNum).locV = sprite(me.spriteNum).locV + pSpeed

    -- see if at the lowest point
    if sprite(me.spriteNum).locV >= pOrigLocV then
      pMode = #down
    end if
```

```
  else if pMode = #hit then
    -- if recently hit, reset locV and member
    sprite(me.spriteNum).locV = pOrigLocV
    sprite(me.spriteNum).member = member("Duck")
    pMode = #down
  end if
end
```

鸭子的行为里还有一个on mouseDown行为。当用户点击鸭子后，鸭子被视为“击中(hit)”。下面这个处理程序为鸭子换一个演员，以表示它已被击中。在本例中，鸭子变为红色。addScore影片处理程序被调用，pMode变为#hit，这样on exitFrame处理程序能够知道鸭子已被击中。

```
on mouseDown me
  -- if already hit, then ignore
  if pMode = #hit then exit

  -- use other member
  sprite(me.spriteNum).member = member("Duck Hit")
  updateStage

  -- add point
  addScore
  pMode = #hit
end
```

鸭子的行为构成游戏的大部分内容。余下的只是在舞台上把鸭子放到矩形的背后。矩形可以挡住用户的射击，这样用户不能透过它们去打鸭子。用一个简单的行为就可以“吃掉”这些点击鼠标的动作。

```
-- block mouseDowns with boxes
on mouseDown
  nothing
end
```

被鸭子的行为调用的on addScore处理程序是位于影片剧本里的。此外，还应当有一个计时器负责游戏的计时。这个计时器从30秒开始倒计时。在影片开始时，它先记录当前时间(单位为tick，即1/60秒)，然后把这个时间加上30秒就表示游戏结束的时间。然后从这个值中减去当前时间，就得到游戏的剩余时间。

如果有必要，在每帧都将显示剩余时间。这个处理程序还检测游戏是否结束，如果结束，则走到另一帧。

```
global gScore, gEndTime

on startMovie
  -- use sight cursor
  cursor([member "Sight",member "Sight"])

  -- reset score
  gScore = 0

  -- game ends 30 seconds from now
  gEndTime = the ticks + 30*60
```

```
  showScore
  showTime
end

on showScore
  member("Score").text = "Score: "&&gScore
end

on showTime
  -- convert ticks to seconds remaining
  timeLeft = (gEndTime - the ticks + 30)/60

  -- use this text
  text = "Time: "&&timeLeft

  -- if text is different than text displayed
  if member("Time").text <> text then
    member("Time").text = text
  end if

  -- time up?
  if timeLeft <= 0 then
    cursor(0)
    go to frame "Done"
  end if
end

on addScore
  gScore = gScore + 1
  showScore
end

on stopMovie
  cursor(0)
end
```

影片处理程序执行的另一个功能是使用 cursor命令把光标变成一个十字线。光标存储在演员sight里。在on startMovie里，程序打开该光标；当游戏结束时，关闭光标；或者当游戏被on stopMovie处理程序打断时，关闭光标。

帧剧本最后结束这个游戏。它只是在每帧调用 on showTime，查看当前的剩余时间，如果需要还可以显示它，当结束时间一到就结束游戏。

```
on exitFrame
  showTime
  go to the frame
end
```

游戏的玩法是由鸭子的行为所决定的。如果想让鸭子在游戏里横向由一边走到另一边，可以把行为修改一下，让鸭子横向移动，而不是上下移动。我们甚至还可以创建多个行为，让射击目标沿多个方向移动。还可以把这些行为结合在一个影片里，其中有各种不同类型的射击目标。

对该游戏的另一个改进方法可以是随着时间或用户得分的变化加快角色运动的速度。这使得游戏对于熟练的用户更有吸引力。

32.5 制作“宇宙入侵者”

所有游戏中最经典的游戏“宇宙入侵者”已有成百上千种翻版，其中有些就是用 Director 制作的。下面的例子只是非常简单的一个游戏。图 32-7是正在玩游戏时的情景。

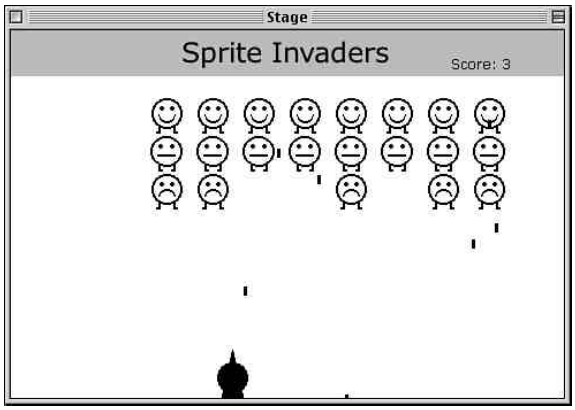

图32-7 在这个“入侵者”游戏里，入侵者是一些样子十分可笑的位图

这个入侵者游戏比前面几个游戏都复杂。它分为四个行为：一个用于入侵者的角色，一个用于战舰，一个用于入侵者所发射的子弹，一个用于战舰所发射的子弹。

32.5.1 制作入侵者角色

游戏主要由入侵者的运动所控制，它们集体从舞台的一侧横向移到另一侧。当它们碰到舞台某个侧边时，就向下掉，离舞台的底边更近一步。

控制这种运动只需要一个行为，再把这个行为赋予舞台上的每一个入侵者角色就可以了。不过，还要使用gHitWall和gHitBottom两个全局变量。当至少一个角色碰到舞台的侧边或底边时，相应的全局变量就被设为 TRUE，然后由帧剧本来处理这个情况。

该行为开始时，方向和速度都被设为 2，表示入侵者每帧向右移动两个像素。它还记录角色的演员编号。在演员表里，每个入侵者有两个演员，它们基本相同，只是脚的方向反过来，给人以“进军”的感觉。入侵者行为在各帧交替使用这两个演员。

```
global gHitWall, gHitBottom
property pDirection, pMemNum

on beginSprite me
  -- start moving 2 pixels to right
  pDirection = 2
  pMemNum = sprite(me.spriteNum).memberNum
end

on exitFrame me
  -- freeze if not on play frame
  if the frameLabel <> "Play" then exit

  if pDirection = 0 then
    -- no direction, must have been hit
    sprite(me.spriteNum).memberNum = 0 -- remove sprite
```

```
  else
    -- move
    sprite(me.spriteNum).locH = sprite(me.spriteNum).locH + pDirection

    -- hit a wall?
    if pDirection > 0 and sprite(me.spriteNum).locH > 460 then
      gHitWall = TRUE
    else if pDirection < 0 and sprite(me.spriteNum).locH < 20 then
      gHitWall = TRUE
    end if

    -- toggle to other member to create animation
    if sprite(me.spriteNum).memberNum = pMemNum then
      sprite(me.spriteNum).memberNum = pMemNum + 1
    else
      sprite(me.spriteNum).memberNum = pMemNum
    end if

    -- fire 1 out of 200 times
    if random(200) = 1 then
      sendSprite(sprite 55, #fire, sprite(me.spriteNum).loc)
    end if
  end if
end
```

on exitFrame处理程序还以1/200的机会引发入侵者发射子弹。它向角色55——即入侵者的子弹——发送一个#fire消息。

如果哪个角色与舞台侧边的距离太近， on exitFrame处理程序就把全局变量gHitWall设为TRUE。图32-7的例子里使用了24个入侵者。只要有一个入侵者距离墙(舞台的侧边)太近，就要把这个全局变量设置为TRUE。如果真的变为TRUE了，帧剧本就将处理它，即向所有角色发送一个#changeDirection消息。只有入侵者角色才使用这个消息。下面就是那个被调用的处理程序。它把入侵者向下移动，也改变运动方向。它还检查是否有角色已经太靠下了。

```
on changeDirection me
  -- got change direction message

  -- move down
  sprite(me.spriteNum).locV = sprite(me.spriteNum).locV + abs(pDirection)

  -- hit bottom?
  if sprite(me.spriteNum).locV > sprite(5).rect.top then gHitBottom = TRUE

  -- reverse direction
  pDirection = -pDirection
end
```

可以发送给入侵者的另一个消息是#hit。这时，演员变为一个爆炸情景的位图。舞台被更新，于是爆炸的情景立刻显示出来。然后， pDirection属性被设为0。这是为了让on exitFrame处理器知道这个入侵者已经死了，并通过在下一帧把它的演员编号设为0而删除这个角色。

```
on hit me
  -- got the message I was hit

  -- change to hit graphic
```

```
  sprite(me.spriteNum).member = member("Invader Hit")

  -- show me, since I will disappear next frame
  updateStage

  -- dead, so no direction
  pDirection = 0
end
```

与入侵者行为相连的是入侵者的子弹的行为。这个行为将赋予角色 55至角色75。如果想让角色每次发射更多子弹，可以把它赋予更多的角色；如果想让入侵者少发射些子弹，可以赋予较少的角色。

入侵者行为里有一个pMode属性，它告诉行为现在是否射击。 on fire处理程序导致发射子弹，或者当已经发射子弹时，把该消息传给下一个角色。 on exitFrame处理程序负责移动发射出来的子弹，并检测它是否击中了什么目标，或是落到了舞台的底部。

```
property pMode

on beginSprite me
  pMode = #none
end

on fire me, loc
  if pMode = #fire then
    -- busy, send to next sprite
    sendSprite(sprite(me.spriteNum+1),#fire,loc)
  else
    -- set loc, mode
    sprite(me.spriteNum).loc = loc
    pMode = #fire
  end if
end

on exitFrame me
  -- freeze unless on play frame
  if the frameLabel <> "Play" then exit

  if pMode = #fire then
    -- move down
    sprite(me.spriteNum).locV = sprite(me.spriteNum).locV + 8
    if sprite(me.spriteNum).locV > 320 then
      -- hit bottom
      pMode = #none
    else
      -- hit gun?
      didIHit(me)
    end if
  end if
end
```

on exitFrame处理程序调用on didIHit，以确定子弹是否击中了战舰。战舰(或枪)是角色5。

```
on didIHit me
  -- hit gun?
  if sprite 5 intersects me.spriteNum then
```

```
    -- gun explodes
    sprite(5).member = member("Invader Hit")
    updateStage

    -- game over
    go to frame "Done"
  end if
end
```

32.5.2 创建战舰

现在入侵者和入侵者的子弹都可以活动了，下面的任务是让战舰活动。用左、右箭头键让战舰左右移动是十分容易的。如果按空格键，则把 #fire消息发送给一组战舰子弹角色。

```
property pFiredLastFrame

on exitFrame me
  if the frameLabel <> "Play" then exit

  if keyPressed(123) then
    -- left arrow
    sprite(me.spriteNum).locH = sprite(me.spriteNum).locH - 5
  end if

  if keyPressed(124) then
    -- right arrow
    sprite(me.spriteNum).locH = sprite(me.spriteNum).locH + 5
  end if

  -- check spacebar, plus check to make sure did not fire last frame
  if keyPressed(SPACE) and not pFiredLastFrame then
    -- space, fire
    sendSprite(sprite 6, #fire, sprite(me.spriteNum).loc)
    pFiredLastFrame = TRUE
  else
    pFiredLastFrame = FALSE
  end if
end
```

战舰行为使用了一个名为 pFiredLastFrame的属性，它可以阻止用户按住空格键不放，从而发射连续的子弹流，导致入侵者的伤亡太重。相反，只能每隔一帧发射一颗子弹。要把发射子弹的时间间隔拉得更长，可以用该属性计算从上一次发射子弹到现在已经过了多少帧，并只让子弹每隔3或4帧才能发射一次。

战舰的子弹的行为与入侵者的子弹的行为十分相似。其区别在于战舰的子弹向上运动，并检测是否击中了任何入侵者角色。

由于每当击中入侵者就要加分，需要使用全局变量 gScore。当检测到击中时，就为用户加分。

```
global gScore
property pMode

on beginSprite me
  pMode = #none
```

```
end

on fire me, loc
  -- got signaled to fire
  if pMode = #fire then
    -- busy, send to next sprite
    sendSprite(sprite(me.spriteNum+1),#fire,loc)
  else
    -- fire
    sprite(me.spriteNum).loc = loc
    pMode = #fire
  end if
end

on exitFrame me
  -- freeze if not on play frame
  if the frameLabel <> "Play" then exit

  if pMode = #fire then
    -- move bullet up
    sprite(me.spriteNum).locV = sprite(me.spriteNum).locV - 16

    if sprite(me.spriteNum).locV < 0 then
      -- reached top of screen
      pMode = #none
    else
      -- check for hit
      didIHit(me)
    end if
  end if
end

on didIHit me
  -- loop through invader sprites
  repeat with i = 30 to 53
    -- see if it hit
    if sprite i intersects me.spriteNum then

      -- send hit message
      sendSprite(sprite i, #hit)

      -- get rid of bullet
      sprite(me.spriteNum).locV = -100
      pMode = #none

      -- add to score
      gScore = gScore + 1
      showScore
    end if
  end repeat
end
```

影片剧本只承担很少的任务，如在必要时重新设置用户的得分，并显示新的分数等。

```
global gScore
```

```
on startMovie
  gScore = 0
  showScore
  go to frame "Play"
end

on showScore
  member("Score").text = "Score: "&&gScore
end
```

32.5.3 创建帧剧本

最后，游戏还需要一个帧剧本。这个帧剧本比一般的帧剧本的任务要多一些。其任务之一是当发现gHitWall标志时，向所有角色发送 #changeDirection。当发现gHitBottom时，它就让游戏结束。这些任务都是在 on enterFrame处理程序里完成的，它保证这些任务将在角色的行为里的on exitFrame处理程序之前执行。

```
global gHitWall, gHitBottom

on enterFrame me
  if gHitWall then
    -- an invader hit the wall
    sendAllSprites(#changeDirection)

  else if gHitBottom then
    -- an invader hit the bottom
    go to frame "Done"

  end if

  -- reset wall hit flag for this frame
  gHitWall = FALSE
end

on exitFrame
  go to the frame
end
```

以上就是游戏所需要的剧本。其中，战舰、子弹和入侵者的速度的数值还都可以改变。可以根据需要，在屏幕上放置更多或更少的入侵者。也可以使用其他位图，但记住每个入侵者由两个演员组成，这两个演员频繁替换，以造成活动的效果。

把CD-ROM上的这个影片打开来，并试着改一改某些设置。可以把入侵者的阵形改变成其他形状，也可以添加一些“盾牌”，方法是添加一些角色和“盾牌”行为。可以采取多种方法改进这个游戏。

32.6 创建问答游戏

问答游戏随处可见。制作问答游戏最困难的部分是提出问题。能够提出问题并记录得分的Director影片的结构是很简洁的。

在这个例子里，影片剧本承担了大部分工作。最终得到的游戏与图 32-8相似。每次提出一个问题，用户必须点击四个按钮中的一个来回答。有一个计时器在倒计时，因此回答问题

所花的时间越长，得到的分数越少。如果答案是错误的，将从计时器里减去 100点。

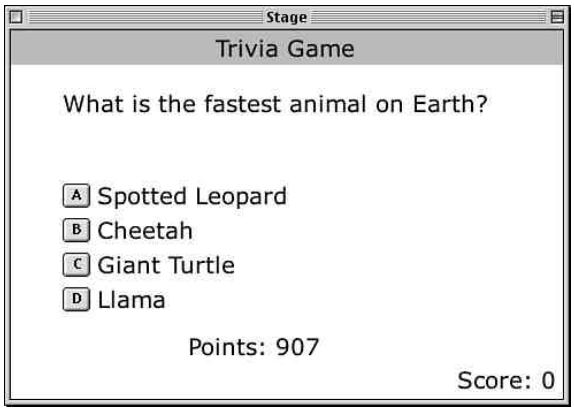

图32-8 一个简单的问答游戏，它每次提出一个问题，并提供四个选择答案

影片剧本需要用几个全局变量来记录几件事情。它需要知道用户正在回答的问题的编号、如果答案正确应当得多少分、当前的得分以及哪一个答案是当前问题的正确答案。影片开始时设置这些全局变量。

```
global gQuestionNum, gPossiblePoints, gScore, gCorrectAnswer

on startGame
  gQuestionNum = 1
  gScore = 0
  showScore
  askQuestion
  go to frame "Play"
end
```

问题存储在一个名为 Data的域里。这个域的每一行是一道题。每一行包括三个项目，其间由分号分隔。第一个项目是问题；第二个项目是四个选择答案；第三个项目是正确答案的编号。

on askQuestion处理程序负责从域里搜索这些信息，并把这些文字摆到屏幕上。

```
on askQuestion
  text = member("Data").text.line[gQuestionNum]

  the itemDelimiter = ";"
  question = text.item[1]
  answers = text.item[2]
  gCorrectAnswer = value(text.item[3])

  member("Question").text = question
  the itemDelimiter = ","
  repeat with i = 1 to 4
    member("Answer"&&i).text = answers.item[i]
  end repeat

  gPossiblePoints = 1000
  showPossiblePoints
end
```

on gameTimer处理程序在每一个帧循环里被调用一次。它从用户答对这道题后可能的得分里减去1分。它操作的速度与帧速度有关。它调用 on showPossiblePoints来改变舞台上的文本演员。

```
on gameTimer
  gPossiblePoints = gPossiblePoints - 1
  showPossiblePoints
end


on showPossiblePoints
  member("Possible Points").text = "Points: "&&gPossiblePoints
end
```

当用户回答完一道题后， on showScore处理程序负责修改舞台上的得分文本演员。

```
on showScore
  member("Score").text = "Score: "&&gScore
end
```

要回答问题，用户必须点击舞台上四个按钮中的一个。每一个按钮都附带有第 14章“创建行为”里所讨论的按钮行为。每一个行为都根据按钮的编号以 1、2、3或4为参数执行 on clickAnswer处理程序。

如果用户答对了，这个处理程序就将应得分数加到得分中，并再问下一个问题。但如果用户的答案是错的，它从应得分中减 100，然后用户可以再试一次。

```
on clickAnswer n
  if n = gCorrectAnswer then
    gScore = gScore + gPossiblePoints
    showScore
    nextQuestion
  else
    gPossiblePoints = gPossiblePoints - 100
    showPossiblePoints
  end if
end
```

on nextQuestion处理程序把全局变量 gQuestionNum加“1”。如果域里的所有问题都已回答完毕，它就把影片带到另一帧。

```
on nextQuestion
  gQuestionNum = gQuestionNum + 1
  if gQuestionNum > member("Data").text.line.count then
    go to frame "done"
  else
    askQuestion
  end if
end
```

这个影片当然也可以使用一些声音效果。当用户点击了正确答案时，应当发出一些悦耳的声音；如果点击了错误的答案，应当发出难听的声音。当影片等待用户回答问题时，甚至应当发出一些钟表的滴答声。

我们还可以试着处理游戏中出现的一些异常情况。例如，用户总是点击错误的答案应当怎么办？时间用完了应当怎么办？有没有负分？是否自动把用户带到下一个问题？

一些流行的问答游戏在问题之间有很多种动画。用 Director当然可以很容易地实现这一点。在提出每个问题的帧之间可以安插一些较长的动画影片。

32.7 创建Blackjack游戏

制作Blackjack(二十一点)等纸牌游戏比本章前面介绍的那些游戏要难得多。下面的这个程序虽然没有考虑“拆分(Splitting)”和“加倍(Doubling down)”等普通的Blackjack规则，程序量已经很大了。

从图32-9可以看出这个游戏的复杂程度。显示了两套牌：顶部的发牌人的牌和紧跟在它下面玩游戏者的牌。共使用了 6张牌。每个人手里可以有多张牌——如果玩游戏者或发牌人抽到的全是2，可以多达11张牌。在牌的下面有一些文字，显示玩游戏者手里的钱、赌注和手里的牌的分值，以及发牌人手里的牌的分值，还有一些附加信息。此外，还有三个按钮(Deal、Hit和Stay)以及一个可编辑的文本区域，使玩游戏者能够在此输入赌注。

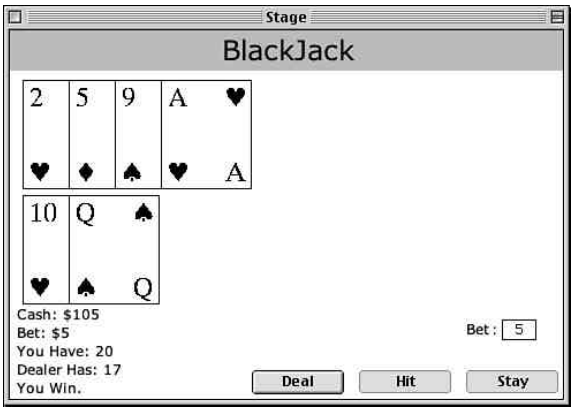

图32-9 即使是一个简单的Blackjack游戏也有很多内容

这个例子使用一个影片剧本来控制游戏。其中的全局变量有：一个包含游戏里的牌的列表、一个用于发牌人的牌的列表、一个用于玩游戏者的牌的列表、玩游戏者的钱以及当前的赌注。

这个影片使用两帧。第一帧有被激活的 Deal按钮，并允许用户输入赌注。点击这个 Deal按钮可以把用户带到第二帧，其中， Hit和Stay按钮是被激活的。

在影片的开始，由on shuffle处理程序来洗牌。这个处理程序创建一个 6副牌的列表，并打乱牌的顺序。每一张牌由一个短的字符串表示，例如 5c表示梅花5，Qh表示红桃皇后。

还可以注意到， the floatPrecision被设置为2。这样，游戏里的钱数可以用更恰当的形式表示，而不是在小数点后面用 4位数。

```
global gDeck, gNextCardSprite, gDealer, gPlayer, gCash, gBet

-- Shuffles Deck, resets cash
on startMovie
  the floatPrecision = 2
  shuffle
  gCash = 100
  member("bet field").text = "5"
```

```
  clearScreen
  go to frame "Bet"
end

-- Set up deck list
on shuffle
  gDeck = []
  tempList = []
  suit = ["c","s","h","d"]
  number = ["2","3","4","5","6","7","8","9","10","J","Q","K","A"]

  -- create an ordered deck
  repeat with i = 1 to 6 -- six decks of cards
    repeat with s = 1 to 4
      repeat with n = 1 to 13
        card = getAt(number,n)&getAt(suit,s)
        add tempList, card
      end repeat
    end repeat
  end repeat

-- pick cards out of tempList and place them in the deck
  repeat while tempList.count > 0
    r = random(tempList.count)
    add gDeck, getAt(tempList,r)
    deleteAt tempList,r
  end repeat
end
```

另一个被on startMovie调用的处理程序负责设置舞台上所有的文本演员。它还把角色 10～40清零，它们将要用于纸牌。

```
on clearScreen
  -- reset all text
  member("Cash").text = "Cash: $"&gCash
  member("Bet").text = " "
  member("Player Value").text = " "
  member("Dealer Value").text = " "
  member("Message").text = " "

  -- clear out sprites 10 to 40
  repeat with i = 10 to 40
    sprite(i).memberNum = 0
  end repeat
end
```

每发一张牌，都将发生几件事情。首先，文本演员被重设。其次，检查 gDeck列表，以确认该列表中还有至少一半的纸牌。如果列表中的纸牌已经不到一半了，就重新洗牌，得到新的6副牌。

接着，发牌人和玩游戏者的牌被复位为空的列表。然后，按照人们玩 Blackjack的常规发牌顺序为每个人发两张牌。 on dealCard处理程序被调用，负责把牌放到舞台上以及发牌人或玩游戏者的列表里。所发的第一张牌是面朝下扣着的，这由 on dealCard处理程序的第二个参数指定。

发完牌后，要从玩游戏者的钱里减去赌注。有几个文本演员被更新，以反映这一变化。

在Blackjack里，如果玩游戏者抽到21分，他就赢了，这一手牌也就结束了。因此该处理程序要检测是否出现这一局面。 on figureHand处理程序被调用，以计算每个人手里的牌的分值。它可以返回由一个或两个数字组成的列表。某一手牌里如果包含有两个“ A”，它将有两个可能的值，因为“ A”可以计为1分，也可以计为11分。

由于我们不希望一手牌超过21分，因此如果有手里有两个“ A”就可以少考虑一种计分方式了，因为若把两个“ A”都计为11分，就是22分了，谁也不想要这个分数。

如果玩游戏者一开始就得到21点，他的得分将增加，增加的量是赌注的2.5倍，然后这一手牌就结束了。程序将翻开发牌人的那张朝下扣着的牌，然后返回到下注画面里。

```
-- Clears screen and deals initial hand
on dealHand
  clearScreen

  -- if less than half a deck left, shuffle before deal
  if gDeck.count <= 26 then
    shuffle
  end if

  -- start with sprite 10
  gNextCardSprite = 10

  gDealer = []
  gPlayer = []

  -- initial deal
  dealCard #dealer,TRUE -- deal face down
  dealcard #player
  dealcard #dealer
  dealCard #player

  -- deduct bet from cash
  gBet = value(member("bet field").text)
  gCash = gCash-gBet

  -- set screen text
  member("Cash").text = "Cash: $"&gCash
  member("Bet").text = "Bet: $"&gBet
  playerVal = figureHand(gPlayer)
  member("Player Value").text = "You Have: "&&displayVal(playerVal)
  go to frame "Play"

  -- Check for inital Blackjack
  if getLast(figureHand(gPlayer)) = 21 then
    member("Message").text = "BlackJack! "
    gCash = 2.5*gBet+gCash
    member("Cash").text = "Cash: $"&gCash
    sprite(10).member = member(gDealer[1])
    go to frame "bet"
  end if
end
```

on dealCard处理程序取gDeck列表里的第一张牌，并使用它。它从gDeck里删除这个项目。

如果toWho的参数是#dealer，它就把这张牌添加到gDealer列表里，并向on drawCard发送消息，把牌放在纵向位置为90的地方；如果是#player，则添加到gPlayer列表里，并把牌放在纵向位置为190的地方。

```
-- Pick the next card off the deck
on dealCard toWho, faceDown
  c = gDeck[1]
  deleteAt gDeck, 1
  if toWho = #dealer then
    add gDealer, c
    drawCard(90,gDealer.count,c,faceDown)
  else
    add gPlayer, c
    drawCard(190,gPlayer.count,c,faceDown)
  end if
  updateStage
end
```

on drawCard处理程序负责创建表现每一张牌的角色。它使用 puppetSprite来控制由gNextCardSprite所指定的牌。然后根据参数 y设置角色的纵向位置，根据参数 n设置角色的横向位置。如果参数faceDown为TRUE，它将显示演员back，而不是纸牌演员。

要使这个处理程序能正常运行，我们需要准备 53个位图演员：每张牌一个演员，外加用于朝下扣着的back演员。牌的外观与图32-9里的相似，你也可以自己设计其他的花样。

```
-- Draw the card on the stage by assigning it to the next sprite
on drawCard y, n, card, faceDown
  -- control next sprite and set it to card location
  puppetSprite gNextCardSprite, TRUE
  sprite(gNextCardSprite).locV = y
  sprite(gNextCardSprite).locH = n*40+10

  -- Card up or down
  if not faceDown then
    sprite(gNextCardSprite).member = member(card)
  else
    sprite(gNextCardSprite).member = member("back")
  end if

  gNextCardSprite = gNextCardSprite+1

  -- Delay a bit
  startTimer
  repeat while the timer < 15
  end repeat
end
```

on figureHand处理程序取一个列表里的牌，并计算它的分值。它把“ 10 ”、“ J ”、“ Q ”和“ K ”算作10分，当这手牌里至少有1个“ A ”时，它会识别出来，并为它计算出两种分值，一种是“ A ”为1分，一种是“ A ”为11分。它把这些值放在一个短列表里返回。

```
-- Calculate the value of a hand and return a list
on figureHand list
  total = 0

  -- Loop through hand, and add up cards
  repeat with i = 1 to list.count
    card = list[i]
    if "1JQK" contains card.char[1]  then
      total = total + 10
    else if card.char[1]  = "A" then
      total = total + 1
      haveAce = TRUE
    else
      total = total + value(card.char[1])
    end if
  end repeat

  -- If an ace is present, then there are two values
  if haveAce then
    if total+10 > 21 then return [total]
    else return [total,total+10]
  else
    return [total]
  end if
end
```

尽管on figureHand计算出了一手牌的分值，并把它们放在一个列表里返回，但我们并不能把这个列表显示给用户。一个好的方法是把这个列表拆开，把一手牌的分值显示为一个数值或两个数值。

```
-- Take the returned value list and display it in English
on displayVal val
  if val.count = 1 then return string(val[1])
  else return val[1]&& "or"&&val[2]
end
```

舞台上的三个按钮使用的是第 14章所描述的复杂的按钮行为。 Deal按钮调用 on dealHand；而Hit按钮调用下面的on hitMe处理程序，它可以再发一张牌给玩游戏者。它要检查玩游戏者是否已“撑死”，如果没有“撑死”，则显示新的分值。

```
-- Give the player another card
on hitMe
  dealCard #player
  playerVal = figureHand(gPlayer)
  if playerVal[1] > 21 then
    member("Player Value").text = "Bust. "
    dealerWins
  else
    member("Player Value").text = "You Have: "&&displayVal(playerVal)
  end if
end
```

Stay按钮调用on doDealer处理程序。首先，它翻开那张面朝下的牌。然后，这个处理程序创建一个循环，向发牌人手里加牌。只有当发牌人手里的牌的分值小于或等于 16时才加牌。如果发牌人手里的牌的分值大于或等于 17，或已经“撑死”，就不加牌了。下一步调用 on

decideWhoWins处理程序。

```
-- Dealer hits until 17 or above
on doDealer
  -- Show face down card first
  sprite(10).member = member(gDealer[1])
  dealerVal = figureHand(gDealer)
  member("Dealer Value").text = "Dealer Has: "&&displayVal(dealerVal)
  updateStage

  -- Keep adding cards
  repeat while TRUE
    -- See if dealer is done
    if (dealerVal[1] > 16) or ((getLast(dealerVal) > 16) and¬
(getLast(dealerVal) < 22)) then
      decideWhoWins
      exit repeat
    else
      dealCard #dealer

      dealerVal = figureHand(gDealer)
      member("Dealer Value").text = "Dealer Has: "&&displayVal(dealerVal)
      updateStage

      -- wait a second
      startTimer
      repeat while the timer < 60
      end repeat
    end if

  end repeat
end
```

要查看谁赢了，首先查看发牌人是否已经“撑死”。如果没有，则比较二人的分值。在这个程序里，如果二人的分值相等，算发牌人赢。

```
-- Figure out who has highest valid hand
on decideWhoWins
  -- Get hand values
  dealerVal = figureHand(gDealer)
  playerVal = figureHand(gPlayer)

  -- Dealer busts
  if (dealerVal[1] > 21) then
    member("Message").text = "Dealer busts. You Win. "
    playerWins

    -- Dealer and player have valid hands
  else
    -- Decide highest possible value of each hand, given aces present
    if (dealerVal[dealerVal.count] < 22) then dval = dealerVal[dealerVal.count]
    else dval = dealerVal[1]
    if (playerVal[playerVal.count] < 22) then pval = playerVal[playerVal.count]
    else pval = playerVal[1]

    -- Who wins
```

```
    if pval > dval then
      member("Message").text =  "You Win. "
      playerWins
    else
      member("Message").text =  "You Lose. "
      dealerWins
    end if
  end if
end
```

on decideWhoWins调用另外两个处理程序来结束这一轮游戏。第一个把赌注的两倍返回给玩游戏者，第二个返回Bet画面。

```
-- Double money back
on playerWins
  gCash = 2*gBet+gCash
  member("Cash").text = "Cash: $"&gCash
  go to frame "bet"
end

-- No money back
on dealerWins
  go to frame "bet"
end
```

除了影片剧本外，还有一个附属于赌注域的小剧本。它防止用户输入比他手里的钱数多的赌注，并防止用户输入除数字以外的其他字符。

```
on keyDown
  if "0123456789" contains the key then
    pass
  end if
end

on keyUp
  global gCash
  if value(field "bet field") > gCash then
    put string(gCash) into field "bet field"
  end if
end
```

请看一看CD-ROM上的影片，看各部分是如何协作的。可以试着加一些新功能。也许你喜欢加上洗牌和发牌的声音，并在玩游戏者赢了时发出一声欢呼。

一个难度更大的任务是使用户能够使用“拆分”。使用了“拆分”后，在屏幕上，我们可以同时有两手牌，也可以每次只玩其中的一手。“加倍”是一种比较简单的规则，用户把赌注翻倍，但只能要一张额外的牌。我们可以从用户的钱里减去与赌注等量的钱，再发给他们一张牌。要使游戏能够完整，应当想办法处理用户把钱花光时的情形。

32.8 游戏的故障排除

■ 很多游戏都是既有行为，又有影片剧本。应当记得检查是否正确地设置了每类剧本，即不要把行为设置为影片剧本或把影片剧本设置为行为。

■ Director 7的新函数keyPressed()可以用于很多种游戏，因为它确切地告诉我们键盘上发

生了哪些操作，而on keyDown只能在给定的时间发出单一的信息。不过，值得注意的是用户有时会一次按下多个键。应当确保游戏能够处理用户同时按下左、右箭头键的情况。

■ 对于游戏来说，真正要担心的不是程序错误，而是游戏的可玩性和娱乐性，因为游戏的任务就是让人们喜欢它。如果设计的游戏没有娱乐性，它本身就是一个程序错误。

32.9 你知道吗

■ 很多游戏都用洗牌的方式把牌的顺序随机打乱。洗牌使用 random函数，从第一个列表中取出牌放到第二个列表里。如果想要每次洗牌的结果都相同，以用于测试等用途，试着用randomSeed属性指定一个数字，如1、2、3等等。这将保证每次都生成相同的随机数，对于同一盘游戏的洗牌结果保持稳定。