第七章　滑鼠

滑鼠是有一个或多个键的定位设备。虽然也可以使用诸如触摸画面和光笔之类的输入设备，但是只有滑鼠以及常用在膝上型电脑上的轨迹球等才是渗透了 PC 市场的唯一输入设备。

情况并非总是如此。当然，Windows 的早期开发人员认为他们不应该要求使用者为了执行其产品而必须买只滑鼠。因此，他们将滑鼠作为一种选择性的附加设备，而为 Windows 中的所有操作以及 applet 提供一种键盘介面（例如，查看 Windows 小算盘程式的线上说明资讯，可以看到每个按钮都提供了一个同等功效的键盘操作方式）。第三方软体发展人员使用键盘介面来提供与滑鼠操作相同的功能，这本书以前的版本也是这么做的。

理论上来说，现在的 Windows 需要滑鼠。至少，一些讯息方块是这样讲的。当然，您也可以拔下滑鼠，而且 Windows 仍然可以执行良好（只有讯息方块会提示您没有连接滑鼠）。试图不用滑鼠来使用 Windows 就像用脚趾来弹钢琴一样（至少在最初的一段时间里是这样），但您依然可以这样做。正因为如此，我还是喜欢为滑鼠功能提供键盘操作。打字员尤其喜欢让他们的手保持在键盘上，并且我认为每个人都有在杂乱的桌上找不到滑鼠，或者滑鼠移动不灵敏的经验。使用键盘通常不需要花费更多的精力和努力，并且为喜欢使用键盘的人提供更多的功能。

我们通常认为，键盘便於输入和操作文字资料，而滑鼠则便於画图和操作图形物件。实际上，本章大多数的范例程式都画了一些图形，并且用到了我们在第五章所学到的知识。

滑鼠基础

Windows 98 能支援单键、双键或者三键滑鼠，也可以使用摇杆或者光笔来模拟单键滑鼠。早期，由於许多使用者都有单键滑鼠，所以 Windows 应用程式总是避免使用双键或三键滑鼠。不过，由於双键滑鼠已经成为事实上的标准，因此不使用第二个键的传统已经不再合理了。当然，第二个滑鼠按键是用於启动一个「快显功能表」，亦即出现在普通功能表列之外的视窗中功能表，或者用於特殊的拖曳操作（拖曳将在後面加以解释）。然而，程式不能依赖双键滑鼠。

理论上，您可以用我们的老朋友 GetSystemMetrics 函式来确认滑鼠是否存在：

```
fMouse = GetSystemMetrics (SM_MOUSEPRESENT) ;
```

如果已经安装了滑鼠，fMouse 将传回 TRUE（非 0）；如果没有安装，则传回 0。然而，在 Windows 98 中，不论滑鼠是否安装，此函式都将传回 TRUE 。在 Microsoft Windows NT 中，它可以正常工作。

要确定所安装滑鼠其上按键的个数，可使用

```
cButtons = GetSystemMetrics (SM_CMOUSEBUTTONS) ;
```

如果没有安装滑鼠，那么函式将传回 0。然而，在 Windows 98 下，如果没有安装滑鼠，此函式将传回 2。

习惯用左手的使用者可以使用 Windows 的「控制台」来切换滑鼠按键。虽然应用程式可以通过在 GetSystemMetrics 中使用 SM_SWAPBUTTON 参数来确定是否进行了这种切换，但通常没有这个必要。由食指触发的键被认为是左键，即使事实上是位於滑鼠的右边。不过，在一个教育训练程式中，您可能想在萤幕上画一个滑鼠，在这种情况下，您可能想知道滑鼠按键是否被切换过了。

您可以在「控制台」中设定滑鼠的其他参数，例如双击速度。从 Windows 应用程式，通过使用 SystemParametersInfo 函式可以设定或获得此项资讯。

一些简单的定义

当 Windows 使用者移动滑鼠时，Windows 在显示器上移动一个称为「滑鼠游标」的小点阵图。滑鼠游标有一个指向显示器上精确位置的单图素「热点」。当我提到滑鼠游标在萤幕上的位置时，指的是热点的位置。

Windows 支援几种预先定义的滑鼠游标，程式可以使用这些游标。最常见的是称为 IDC_ARROW 的斜箭头（在 WINUSER.H 中定义）。热点在箭头的顶端。IDC_CROSS 游标（在本章後面的 BLOKOUT 程式中有用到）的热点在十字交叉线的中心。 IDC_WAIT 游标是一个沙漏，通常用於指示程式正在执行。程式写作者也可以设计自己的游标。我们将在第十章学习设计方法。在定义视窗类别结构时指定特定视窗的内定游标，例如：

```
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
```

下面是一些描述滑鼠按键动作的术语：

- Clicking 按下并放开一个滑鼠按键。
- Double-clicking 快速按下并放开滑鼠按键两次。
- Dragging 按住滑鼠按键并移动滑鼠。

对三键滑鼠来说，三个键分别称为左键、中键、右键。在 Windows 表头档案中定义的与滑鼠有关的识别字使用缩写 LBUTTON、MBUTTON 和 RBUTTON。双键滑鼠只有左键与右键，单键滑鼠只有一个左键。

滑鼠(Mouse)的复数

现在，为了展现我的勇气，我将面对输入装置最难辩的争论话题：什么是「mouse」的复数。虽然每个人都知道多只啮齿动物称为 mice，似乎没有人对该如何称呼多个输入装置有最後的答案。不管「mice」或「mouse」听起来都不对劲。我惯常参考的《American Heritage Dictionary of the English Language》第三版则只字未提。

《Wired style: Principles of English Usage in the Digital Age》（HardWired, 1996）指出「mouse」比较好，以避免与啮齿动物搞混。在 1964 发明滑鼠的 Doug Engelbart 对此争议也帮不上忙。我曾经问过他 mouse 的复数是什么，他说我不知道。

最後，高权威的 Microsoft Manual of Style for Technical Publications 告诉我们「避免使用复数 mice。假如你必须提到多只 mouse，使用 mouse devices」。这听起来像是在逃避问题，但当一切听起来都不对劲时，它确实是个明智的忠告了。事实上，大部分需要 mouse 复数的句子都能重新修改来避开。例如，试著说"People use the almost as much as keyboard"，而不是"Pople use mice almost as much as keyboards"。

显示区域滑鼠讯息

在前一章中您已经看到，Windows 只把键盘讯息发送给拥有输入焦点的视窗。滑鼠讯息与此不同：只要滑鼠跨越视窗或者在某视窗中按下滑鼠按键，那么视窗讯息处理程式就会收到滑鼠讯息，而不管该视窗是否活动或者是否拥有输入焦点。Windows 为滑鼠定义了 21 种讯息，不过，其中有 11 个讯息和显示区域无关（下面称之为「非显示区域」讯息），Windows 程式经常忽略这些讯息。

当滑鼠移过视窗的显示区域时,视窗讯息处理程式收到 WM_MOUSEMOVE 讯息。当在视窗的显示区域中按下或者释放一个滑鼠按键时，视窗讯息处理程式会接收到下面这些讯息：

表 7-1

键	按下	释放	按下(双键)
左	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
中	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDBLCLK
右	WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDBLCLK

只有对三键滑鼠，视窗讯息处理程式才会收到 MBUTTON 讯息；只有对双键或者三键滑鼠,才会接收到 RBUTTON 讯息。只有当定义的视窗类别能接收 DBLCLK

（双击）讯息，视窗讯息处理程式才能接收到这些讯息（请参见本章中「双击滑鼠按键」一节）。

对於所有这些讯息来说，其 lParam 值均含有滑鼠的位置：低字组为 x 座标，高字组为 y 座标，这两个座标是相对於视窗显示区域左上角的位置。您可以用 LOWORD 和 HIWORD 巨集来提取这些值：

```
x = LOWORD (lParam) ;
y = HIWORD (lParam) ;
```

wParam 的值指示滑鼠按键以及 Shift 和 Ctrl 键的状态。您可以使用表头档案 WINUSER.H 中定义的位元遮罩来测试 wParam。MK 字首代表「滑鼠按键」。

MK_LBUTTON	按下左键
MK_MBUTTON	按下中键
MK_RBUTTON	按下右键
MK_SHIFT	按下 Shift 键
MK_CONTROL	按下 Ctrl 键

例如，如果收到了 WM_LBUTTONDOWN 讯息，而且值

```
wparam & MK_SHIFT
```

是 TRUE（非 0），您就知道当左键按下时也按下了 Shift 键。

当您把滑鼠移过视窗的显示区域时，Windows 并不为滑鼠的每个可能的图素位置都产生一个 WM_MOUSEMOVE 讯息。您的程式接收到 WM_MOUSEMOVE 讯息的次数，依赖於滑鼠硬体，以及您的视窗讯息处理程式在处理滑鼠移动讯息时的速度。换句话说，Windows 不能用未处理的 WM_MOUSEMOVE 讯息来填入讯息伫列。当您执行下面将描述的 CONNECT 程式时，您将会更了解 WM_MOUSEMOVE 讯息处理的速率。

如果您在非活动视窗的显示区域中按下滑鼠左键，Windows 将把活动视窗改为在其中按下滑鼠按键的视窗，然後把 WM_LBUTTONDOWN 讯息送到该视窗讯息处理程式。当视窗讯息处理程式得到 WM_LBUTTONDOWN 讯息时，您的程式就可以安全地假定该视窗是活动化的了。不过，您的视窗讯息处理程式可能在未接收到 WM_LBUTTONDOWN 讯息的情况下先接收到了 WM_LBUTTONUP 的讯息。如果在一个视窗中按下滑鼠按键，然後移动到使用者视窗释放它，就会出现这种情况。类似的情况，当滑鼠按键在另一个视窗中被释放时，视窗讯息处理程式只能接收到 WM_LBUTTONDOWN 讯息，而没有相应的 WM_LBUTTONUP 讯息。

这些规则有两个例外：

视窗讯息处理程式可以「拦截滑鼠」并且连续地接收滑鼠讯息，即使此时滑鼠在该视窗显示区域之外。您将在本章的後面学习如何拦截滑鼠。

如果正在显示一个系统模态讯息方块或者系统模态对话方块，那么其他程

式就不能接收滑鼠讯息。当系统模态讯息方块或者对话方块活动时，禁止切换到其他视窗或者程式。一个显示系统模态讯息方块的例子，是当您关闭 Windows 时。

简单的滑鼠处理：一个例子

程式 7-1 中所示的 CONNECT 程式能作一些简单的滑鼠处理，使您对 Windows 如何向您的程式发送滑鼠讯息有一些体会。

程式 7-1　CONNECT

```
CONNECT.C
/*------------------------------------------------------------------------
      CONNECT.C -- Connect-the-Dots Mouse Demo Program
                           (c) Charles Petzold, 1998
------------------------------------------------------------------------*
/

#include <windows.h>
#define MAXPOINTS 1000
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                  PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("Connect") ;
      HWND                  hwnd ;
      MSG                   msg ;
      WNDCLASS              wndclass ;

      wndclass.style              = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc        = WndProc ;
      wndclass.cbClsExtra         = 0 ;
      wndclass.cbWndExtra         = 0 ;
      wndclass.hInstance          = hInstance ;
      wndclass.hIcon              = LoadIcon (NULL, IDI_APPLICATION) ;
      wndclass.hCursor            = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground      = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
      wndclass.lpszMenuName       = NULL ;
      wndclass.lpszClassName      = szAppName ;
      if (!RegisterClass (&wndclass))
      {
            MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                                     szAppName, MB_ICONERROR) ;
            return 0 ;
      }

      hwnd = CreateWindow (  szAppName, TEXT ("Connect-the-Points Mouse Demo"),
```

```
                                                WS_OVERLAPPEDWINDOW,
                                                CW_USEDEFAULT, CW_USEDEFAULT,
                                                CW_USEDEFAULT, CW_USEDEFAULT,
                                                NULL, NULL, hInstance, NULL) ;

      ShowWindow (hwnd, iCmdShow) ;
      UpdateWindow (hwnd) ;

      while (GetMessage (&msg, NULL, 0, 0))
      {
            TranslateMessage (&msg) ;
            DispatchMessage (&msg) ;
      }

return msg.wParam ;
}

LRESULT CALLBACK WndProc (  HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
      static POINT pt[MAXPOINTS] ;
      static int       iCount ;
      HDC                    hdc ;
      in                     i, j ;
      PAINTSTRUCT  ps ;
      switch (message)
     {
      case  WM_LBUTTONDOWN:
            iCount = 0 ;
            InvalidateRect (hwnd, NULL, TRUE) ;
            return 0 ;

      case  WM_MOUSEMOVE:
            if (wParam & MK_LBUTTON && iCount < 1000)
            {
                        pt[iCount  ].x = LOWORD (lParam) ;
                        pt[iCount++].y = HIWORD (lParam) ;

                        hdc = GetDC (hwnd) ;
                        SetPixel (hdc, LOWORD (lParam), HIWORD (lParam), 0) ;
                        ReleaseDC (hwnd, hdc) ;
            }
                              return 0 ;

      case  WM_LBUTTONUP:
                  InvalidateRect (hwnd, NULL, FALSE) ;
                  return 0 ;
```

```
     case  WM_PAINT:
                hdc = BeginPaint (hwnd, &ps) ;

                SetCursor (LoadCursor (NULL, IDC_WAIT)) ;
                ShowCursor (TRUE) ;

                for (i = 0 ; i < iCount - 1 ; i++)
                                for (j = i + 1 ; j < iCount ; j++)
                                {
                           MoveToEx (hdc, pt[i].x, pt[i].y, NULL) ;
                           LineTo   (hdc, pt[j].x, pt[j].y) ;
           }

                ShowCursor (FALSE) ;
                SetCursor (LoadCursor (NULL, IDC_ARROW)) ;
                EndPaint (hwnd, &ps) ;
                return 0 ;

     case  WM_DESTROY:
                PostQuitMessage (0) ;
                return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

CONNECT 处理三个滑鼠讯息：

- WM_LBUTTONDOWN CONNECT 清除显示区域。
- WM_MOUSEMOVE 如果按下左键，那么 CONNECT 就在显示区域中的滑鼠位置处绘制一个黑点，并保存该座标。
- WM_LBUTTONUP CONNECT 把显示区域中绘制的点与其他每个点连接起来。有时会产生一个漂亮的图形，有时则会是黑鸦鸦的一团糟（见图 7-1）。

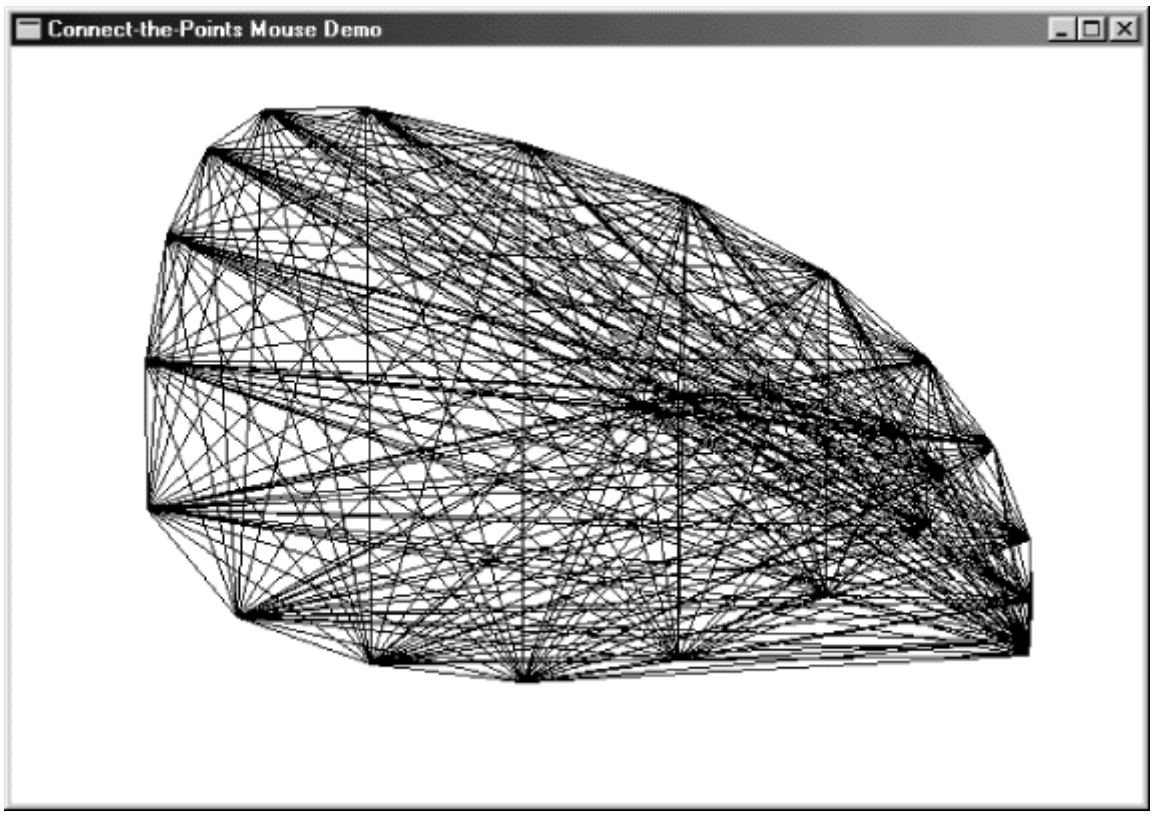

图 7-1 CONNECT 的萤幕显示

CONNECT 的使用方法：把滑鼠游标移动到显示区域中，按下左键，移动一下位置，释放左键。对几个构成曲线的点，CONNECT 能处理得很好，方法是按住左键，快速移动滑鼠，这样就可以绘制出该曲线图案。

CONNECT 使用了三个简单的图形装置介面(GDI)函式，我在第五章讨论过这些函式。当滑鼠左键按下时，SetPixel 为每个 WM_MOUSEMOVE 讯息绘制一个黑图素（对於高解析度的显示器，图素几乎看不见）。画直线需要 MoveToEx 和 LineTo 函式。

如果您在释放滑鼠按键之前把滑鼠游标移到显示区域之外，那么 CONNECT 就不会连接这些点，因为它没有收到 WM_LBUTTONUP 讯息。如果您把滑鼠移回显示区域内并按下左键，那么 CONNECT 将清除显示区域。如果想在显示区域外释放左键後还继续进行画图，那么可以在显示区域外按下滑鼠再移回显示区域中。

CONNECT 最多可以保存 1000 个点。设点数为 P，则 CONNECT 画的线数就等於 P × (P - 1) / 2。如果有 1000 个点，则要绘制 50 万条直线，大约需要几分钟才能画完（时间的长短取决於您的硬体设备）。由於 Windows 98 是一种优先权式多工环境，因此您可以在这一段时间切换到别的程式中。但是，当程式正在忙的时候，您将无法对 CONNECT 程式做任何事（诸如移动或者缩放等）。在第二十章中，我们将讨论解决这一问题的方法。

因为 CONNECT 可能会花一些时间来绘制直线，因此在处理 WM_PAINT 讯息时它将切换到沙漏游标，然後再恢复原状。这要求使用两个现有游标来呼叫

SetCursor。CONNECT 还呼叫两次 ShowCursor，一次用 TRUE 参数，另一次用 FALSE 参数。我将在本章的後面，「使用键盘模拟滑鼠」一节中更详细地讨论这些呼叫。

有时，我们使用「跟踪」这个词代表程式处理滑鼠移动的方法。但是，跟踪并不意味著，程式在视窗讯息处理程式中的某个回圈里，不断跟随滑鼠在显示器上的运动。实际上，视窗讯息处理程式处理每条滑鼠讯息，然後迅速退出。

处理 Shift 键

当 CONNECT 接收到一个 WM_MOUSEMOVE 讯息时，它把 wParam 和 MK_LBUTTON 进行位元与（AND）运算，来确定是否按下了左键。wParam 也可以用於确定 Shift 键的状态。例如，如果处理必须依赖於 Shift 和 Ctrl 键的状态，那么您可以使用如下所示的方法：

```
if (wParam & MK_SHIFT)
{
      if (wParam & MK_CONTROL)
      {
            //按下了 Shift 和 Ctrl 键
      }
      else
      {
                  //按下了 Shift 键
            }
      {
      else
      {
            if (wParam & MK_CONTROL)
            {
                  //按下了 Ctrl 键
            }
            else
            {
                  //Shift 和 Ctrl 键均未按下
            }
}
```

如果您想在程式中同时使用左右键，同时如果您还希望只有单键滑鼠的使用者也能使用您的程式，那么您可以这样来写作程式：Shift 与左键的组合使用等效於右键。在这种情况下，对滑鼠按键的处理可以采用如下所示的方法：

```
case  WM_LBUTTONDOWN:
      if (!(wParam & MK_SHIFT))
      {
                  //处理左键
                  return 0 ;
```

```
      }
                        // Fall through
case WM_RBUTTONDOWN:
      //处理右键
      return 0 ;
```

Windows 函式 GetKeyState（在第六章中介绍过）可以使用虚拟键码 VK_LBUTTON、VK_RBUTTON、VK_MBUTTON、VK_SHIFT 和 VK_CONTROL 来传回滑鼠按键与 Shift 键的状态。如果 GetKeyState 传回负值，则说明已按下了滑鼠按键或者 Shift 键。因为 GetKeyState 传回目前正在处理的滑鼠按键或者 Shift 键的状态，所以全部状态资讯与相应的讯息都是同步的。但是，正如不能把 GetKeyState 用於尚未按下的键一样，您也不能为尚未按下的滑鼠按键呼叫 GetKeyState。请不要这样做：

```
while (GetKeyState (VK_LBUTTON) >= 0) ;  // WRONG !!!
```

只有在您呼叫 GetKeyState 期间处理讯息时，而左键已经按下，才会报告键已经按下的讯息。

双击滑鼠按键

双击滑鼠按键是指在短时间内单击两次。要确定为双击，则这两次单击必须发生在其相距的实际位置十分接近的状况下（内定范围是一个平均系统字体字元的宽，半个字元的高），并且发生在指定的时间间隔（称为「双击速度」）内。您可以在「控制台」中改变时间间隔。

如果希望您的视窗讯息处理程式能够收到双按键的滑鼠讯息，那么在呼叫 RegisterClass 初始化视窗类别结构时，必须在视窗风格中包含 CS_DBLCLKS 识别字：

```
wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS ;
```

如果在视窗风格中未包含 CS_DBLCLKS，而使用者在短时间内双击了滑鼠按键，那么视窗讯息处理程式会接收到下面这些讯息：

- WM_LBUTTONDOWN
- WM_LBUTTONUP
- WM_LBUTTONDOWN
- WM_LBUTTONUP

视窗讯息处理程式可能在这些键的讯息之前还收到了其他讯息。如果您想实作自己的双击处理，那么您可以使用 Windows 函式 GetMessageTime 取得 WM_LBUTTONDOWN 讯息之间的相对时间。第八章将更详细地讨论这个函式。

如果您的视窗类别风格中包含了 CS_DBLCLKS，那么双击时视窗讯息处理程式将收到如下讯息：

- WM_LBUTTONDOWN
- WM_LBUTTONUP
- WM_LBUTTONDBLCLK
- WM_LBUTTONUP

WM_LBUTTONDBLCLK 讯息简单地替换了第二个 WM_LBUTTONDOWN 讯息。

如果双击中的第一次键操作完成单击的功能，那么双击这一讯息是很容易处理的。第二次按键（WM_LBUTTONDBLCLK 讯息）则完成第一次按键以外的事情。例如，看看 Windows Explorer 中是如何用滑鼠来操作档案列表的。按一次键将选中档案，Windows Explorer 用反白显示列指出被选择档案的位置。双击则实作两个功能：第一次是单击那个选中档案；第二次则指向 Windows Explorer 以打开该档案。执行方式相当简单，如果双击中的第一次按键不执行单击功能，那么滑鼠处理方式会变得非常复杂。

非显示区域滑鼠讯息

在视窗的显示区域内移动或按下滑鼠按键时，将产生 10 种讯息。如果滑鼠在视窗的显示区域之外但还在视窗内，Windows 就给视窗讯息处理程式发送一条「非显示区域」滑鼠讯息。视窗非显示区域包括标题列、功能表和视窗卷动列。

通常，您不需要处理非显示区域滑鼠讯息，而是将这些讯息传给 DefWindowProc，从而使 Windows 执行系统功能。就这方面来说，非显示区域滑鼠讯息类似於系统键盘讯息 WM_SYSKEYDOWN、WM_SYSKEYUP 和 WM_SYSCHAR。

非显示区域滑鼠讯息几乎完全与显示区域滑鼠讯息相对应。讯息中含有字母「NC」以表示是非显示区域讯息。如果滑鼠在视窗的非显示区域中移动，那么视窗讯息处理程式会接收到 WM_NCMOUSEMOVE 讯息。滑鼠按键产生如表 7-2 所示的讯息。

表 7-2

键	按下	释放	按下（双击）
左	WM_NCLBUTTONDOWN	WM_NCLBUTTONUP	WM_NCLBUTTONDBLCLK
中	WM_NCMBUTTONDOWN	WM_NCMBUTTONUP	WM_NCMBUTTONDBLCLK
右	WM_NCRBUTTONDOWN	WM_NCRBUTTONUP	WM_NCRBUTTONDBLCLK

对非显示区域滑鼠讯息，wParam 和 lParam 参数与显示区域滑鼠讯息的 wParam 和 lParam 参数不同。wParam 参数指明移动或者按滑鼠按键的非显示区域。它设定为 WINUSER.H 中定义的以 HT 开头的识别字之一(HT 表示「命中测试」)。

lParam 参数的低位元 word 为 x 座标，高位元 word 为 y 座标，但是，它们是萤幕座标，而不是像显示区域滑鼠讯息那样指的是显示区域座标。对萤幕座

标，显示器左上角的 x 和 y 的值为 0。当往右移时 x 的值增加，往下移时 y 的值增加（见图 7-2）。

您可以用两个 Windows 函式将萤幕座标转换为显示区域座标或者反之：

```
ScreenToClient (hwnd, &pt) ;
ClientToScreen (hwnd, &pt) ;
```

这里 pt 是 POINT 结构。这两个函式转换了保存在结构中的值，而且没有保留以前的值。注意，如果萤幕座标点在视窗显示区域的上面或者左边，显示区域座标 x 或 y 值就是负值。

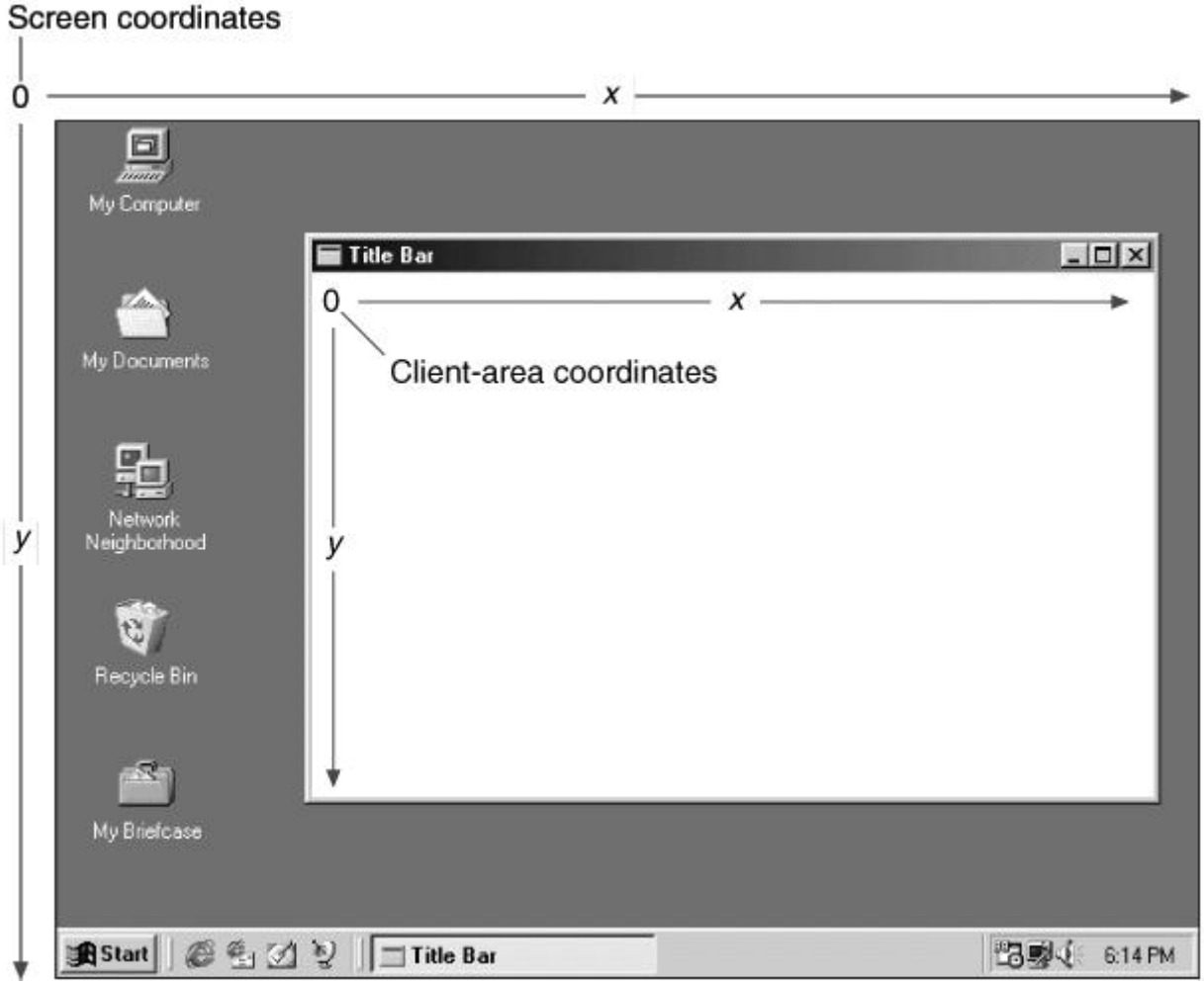

图 7-2 萤幕座标与客户显示区域座标

命中测试讯息

如果您数一下，就可以知道我们已经介绍了 21 个滑鼠讯息中的 20 个，最後一个讯息是 WM_NCHITTEST，它代表「非显示区域命中测试」。此讯息优先於所有其他的显示区域和非显示区域滑鼠讯息。lParam 参数含有滑鼠位置的 x 和 y 萤幕座标，wParam 参数没有用。

Windows 应用程式通常把这个讯息传送给 DefWindowProc，然後 Windows 用

WM_NCHITTEST 讯息产生与滑鼠位置相关的所有其他滑鼠讯息。对於非显示区域滑鼠讯息，在处理 WM_NCHITTEST 时，从 DefWindowProc 传回的值将成为滑鼠讯息中的 wParam 参数，这个值可以是任意非显示区域滑鼠讯息的 wParam 值再加上以下内容：

HTCLIENT	显示区域
HTNOWHERE	不在视窗中
HTTRANSPARENT	视窗由另一个视窗覆盖
HTERROR	使 DefWindowProc 产生警示用的哔声

如果 DefWindowProc 在其处理 WM_NCHITTEST 讯息後传回 HTCLIENT，那么 Windows 将把萤幕座标转换为显示区域座标并产生显示区域滑鼠讯息。

如果您还记得我们如何通过拦截 WM_SYSKEYDOWN 讯息来停用所有的系统键盘功能，那么您可能会想我们可否通过拦截滑鼠讯息完成类似的事情。完全可以！只要您在视窗讯息处理程式中包含以下几条叙述：

```
case WM_NCHITTEST:
      return (LRESULT) HTNOWHERE ;
```

就可以有效地禁用您视窗中的所有显示区域和非显示区域滑鼠讯息。这样一来，当滑鼠在您的视窗（包括系统功能表图示、缩放按钮以及关闭按钮）中时，滑鼠按键将会失效。

从讯息产生讯息

Windows 用 WM_NCHITTEST 讯息产生所有其他滑鼠讯息，这种由讯息引出其他讯息的想法在 Windows 中是很普遍的。让我们来举个例子。您知道，如果您在一个 Windows 程式的系统功能表图示上双击一下，那么程式将会终止。双击产生一系列的 WM_NCHITTEST 讯息。由於滑鼠定位在系统功能表图示上，因此 DefWindowProc 将传回 HTSYSMENU 的值，并且 Windows 把 wParam 等於 HTSYSMENU 的 WM_NCLBUTTONDBLCLK 讯息放在讯息伫列中。

视窗讯息处理程式通常把滑鼠讯息传递给 DefWindowProc，当 DefWindowProc 接收到 wParam 参数等於 HTSYSMENU 的 WM_NCLBUTTONDBLCLK 讯息时，它就把 wParam 参数等於 SC_CLOSE 的 WM_SYSCOMMAND 讯息放入讯息伫列中（这个 WM_SYSCOMMAND 讯息是在使用者从系统功能表中选择「Close」时产生的）。同样地，视窗讯息处理程式也把这个讯息传给 DefWindowProc。DefWindowProc 通过给视窗讯息处理程式发送 WM_CLOSE 讯息来处理该讯息。

如果一个程式在终止之前要求来自使用者的确认，那么视窗讯息处理程式就需要拦截 WM_CLOSE，否则，DefWindowProc 呼叫 DestroyWindow 函式来处理 WM_CLOSE。除了其他处理，DestroyWindow 还给视窗讯息处理程式发送一个

WM_DESTROY 讯息。视窗讯息处理程式通常用下列程式码来处理 WM_DESTROY 讯息：

```
case  WM_DESTROY:
      PostQuitMessage (0) ;
      return 0 ;
```

PostQuitMessage 使得 Windows 把 WM_QUIT 讯息放入讯息伫列中，此讯息永远不会到达视窗讯息处理程式，因为它使 GetMessage 传回 0，并终止讯息回圈，从而也终止了程式。

程式中的命中测试

我在前面讨论了 Windows Explorer 如何回应滑鼠的单击和双击。显然，程式（或者更精确的说，如同 Windows Explorer 般使用 list view control）必须确定使用者滑鼠所指向的是哪一个档案。

这叫做「命中测试」。正如 DefWindowProc 在处理 WM_NCHITTEST 讯息时做一些命中测试一样，视窗讯息处理程式经常必须在显示区域中进行一些命中测试。一般来说，命中测试中会使用 x 和 y 座标值，它们由传到视窗讯息处理程式的滑鼠讯息的 lParam 参数给出。

一个假想的例子

有这样一个例子。假设您的程式需要显示几列按字母排列的档案。通常，您可以使用 list view control，他会帮您由於要做全部的命中测试工作。但我们假设您由於某种原因而不能使用，这时就需要自己来做了。让我们假定档案名保存在称为 szFileNames 的已排序字串指标阵列中。

让我们也假定档案列表开始於显示区域的顶端，显示区域为 cxClient 图素宽， cyClient 图素高，每列为 cxColWidth 图素宽，每个字元高度为 cyChar 图素高。那么每栏可填入的档案数就是：

```
iNumInCol = cyClient / cyChar ;
```

接收到一个滑鼠单击讯息後，您就能从 lParam 获得 cxMouse 和 cyMouse 座标。然後可以用下面的公式来计算使用者所指的是哪一列的档案名：

```
iColumn = cxMouse / cxColWidth ;
```

相对於列顶端的档案名位置为：

```
iFromTop = cyMouse / cyChar ;
```

现在您就可以计算 szFileNames 阵列的下标：

```
iIndex = iColumn * iNumInCol + iFromTop ;
```

如果 iIndex 超过了阵列中的档案数，则表示使用者是在显示器的空白区域内按滑鼠按键。

在许多情况下，命中测试要比本例更加复杂。在显示一幅包含许多小图形

的图像时，您必须决定要显示的每个小图形的座标。在命中计算中，您必须从座标找到物件。但这将在使用不确定字体大小的字处理程式中变得非常凌乱，因为您必须找到字元在字串中的位置。

范例程式

程式 7-2 所示的 CHECKER1 程式展示了一些简单的命中测试，此程式把显示区域分为 5×5 的 25 个矩形。如果您在某个矩形中按下滑鼠按键，那么在该矩形中将出现一个「X」。如果您再按一次，那么「X」将被删除。

程式 7-2　CHECKER1

```
CHECKER1.C
/*------------------------------------------------------------------------
      CHECKER1.C --          Mouse Hit-Test Demo Program No. 1
                                          (c) Charles Petzold, 1998
------------------------------------------------------------------------*/

#include <windows.h>
#define DIVISIONS 5
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR  szCmdLine,
int iCmdShow)
{         static TCHAR  szAppName[] = TEXT ("Checker1") ;
      HWND             hwnd ;
      MSG         msg ;
      WNDCLASS         wndclass ;

      wndclass.style                          = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc              = WndProc ;
      wndclass.cbClsExtra               = 0 ;
      wndclass.cbWndExtra               = 0 ;
      wndclass.hInstance                = hInstance ;
      wndclass.hIcon                    = LoadIcon (NULL, IDI_APPLICATION) ;
      wndclass.hCursor                  = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground       = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
      wndclass.lpszMenuName        = NULL ;
      wndclass.lpszClassName       = szAppName ;

      if (!RegisterClass (&wndclass))
      {
                  MessageBox (     NULL, TEXT ("Program requires Windows NT!"),
                                          szAppName, MB_ICONERROR) ;
                  return 0 ;
      }
      hwnd = CreateWindow (  szAppName, TEXT ("Checker1 Mouse Hit-Test Demo"),
                    WS_OVERLAPPEDWINDOW,
```

```
                  CW_USEDEFAULT, CW_USEDEFAULT,
                  CW_USEDEFAULT, CW_USEDEFAULT,
                  NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
                  TranslateMessage (&msg) ;
                  DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (   HWND     hwnd,     UINT     message,     WPARAM
wParam,LPARAMlParam)
{
     static BOOL      fState[DIVISIONS][DIVISIONS] ;
     static int       cxBlock, cyBlock ;
     HDC              hdc ;
     int              x, y ;
     PAINTSTRUCT      ps ;
     RECT             rect ;

     switch (message)
    {
     case  WM_SIZE :
               cxBlock = LOWORD (lParam) / DIVISIONS ;
               cyBlock = HIWORD (lParam) / DIVISIONS ;
               return 0 ;

     case WM_LBUTTONDOWN :
               x = LOWORD (lParam) / cxBlock ;
               y = HIWORD (lParam) / cyBlock ;

               if (x < DIVISIONS && y < DIVISIONS)
               {
                              fState [x][y] ^= 1 ;
                              rect.left              = x * cxBlock ;
                              rect.top               = y * cyBlock ;
                              rect.right             = (x + 1) * cxBlock ;
                              rect.bottom            = (y + 1) * cyBlock ;

                              InvalidateRect (hwnd, &rect, FALSE) ;
               }
               else
                              MessageBeep (0) ;
```

```
                return 0 ;

     case  WM_PAINT :
                hdc = BeginPaint (hwnd, &ps) ;

                for (x = 0 ; x < DIVISIONS ; x++)
                for (y = 0 ; y < DIVISIONS ; y++)
                {
                           Rectangle (hdc, x * cxBlock, y * cyBlock,
                           (x + 1) * cxBlock, (y + 1) * cyBlock) ;

                     if (fState [x][y])
                     {
                     MoveToEx    (hdc, x * cxBlock,  y * cyBlock, NULL) ;
                     LineTo      (hdc, (x+1) * cxBlock, (y+1) * cyBlock) ;
                MoveToEx (hdc, x * cxBlock, (y+1) * cyBlock, NULL) ;
                LineTo      (hdc, (x+1) * cxBlock, y  * cyBlock) ;
                           }
                }
                EndPaint (hwnd,&ps);
          return 0 ;

     case  WM_DESTROY :
                PostQuitMessage (0) ;
                return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

图 7-3 是 CHECKER1 的显示。程式画的 25 个矩形的宽度和高度均相同。这些宽度和高度保存在 cxBlock 和 cyBlock 中，当显示区域大小发生改变时，将重新对这些值进行计算。WM_LBUTTONDOWN 处理过程使用滑鼠座标来确定在哪个矩形中按下了键，它在 fState 阵列中标志目前矩形的状态，并使该矩形区域失效，从而产生 WM_PAINT 讯息。

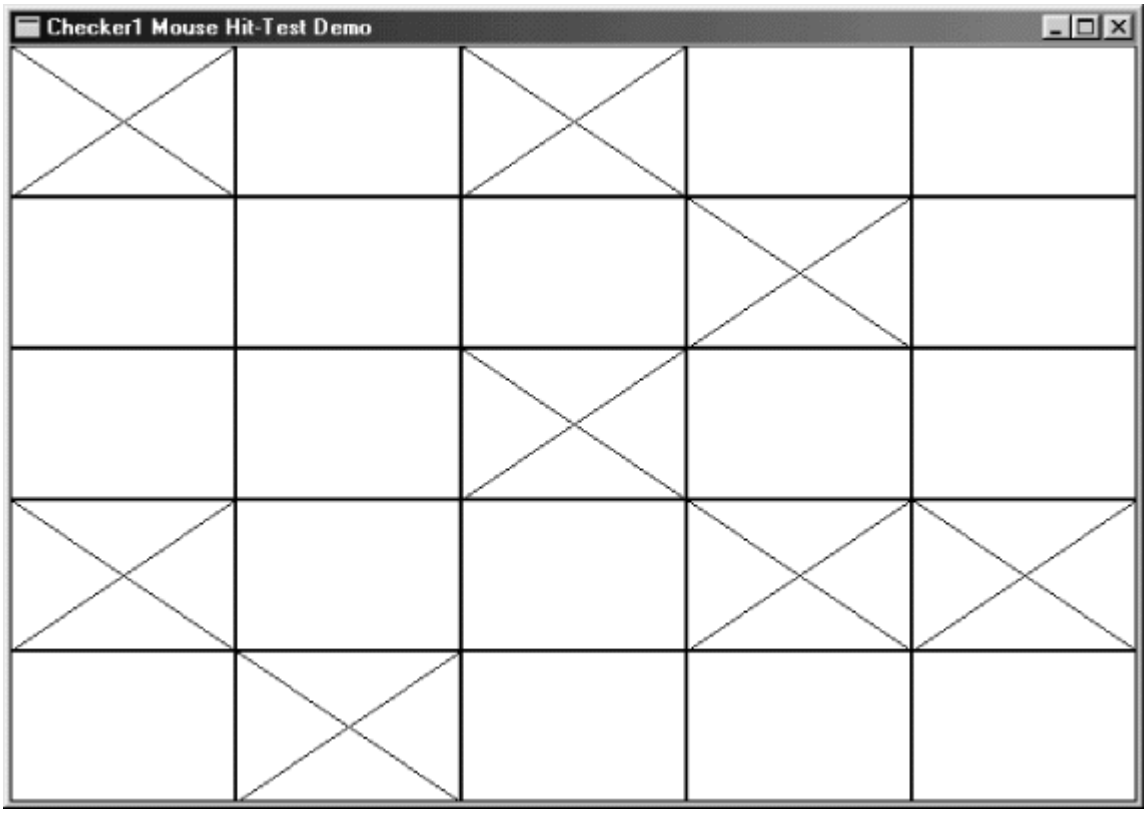

图 7-3 CHECKER1 的萤幕显示

如果显示区域的宽度和高度不能被 5 整除，那么在显示区域的左边和下边将有一小条区域不能被矩形所覆盖。对於错误情况，CHECKER1 通过呼叫 MessageBeep 回应此区域中的滑鼠按键操作。

当 CHECKER1 收到 WM_PAINT 讯息时，它通过 GDI 的 Rectangle 函式来重新绘制显示区域。如果设定了 fState 值，那么 CHECKER1 将使用 MoveToEx 和 LineTo 函式来绘制两条直线。在处理 WM_PAINT 期间，CHECKER1 在重新绘制之前并不检查每个矩形区域的有效性，尽管它可以这样做。检查有效性的一种方法是在回圈中为每个矩形块建立 RECT 结构（使用与 WM_LBUTTONDOWN 处理程式中相同的公式），并使用 IntersectRect 函式检查它是否与无效矩形（ps.rcPaint）相交。

使用键盘模拟滑鼠

CHECKER1 只能在装有滑鼠情况下才可执行。下面我们在程式中加入键盘介面，就如同第六章中对 SYSMETS 程式所做的那样。不过，即使在一个使用滑鼠游标作为指向用途的程式中加入键盘介面，我们还是必须处理滑鼠游标的移动和显示问题。

即使没有安装滑鼠，Windows 仍然可以显示一个滑鼠游标。Windows 为这个游标保存了一个「显示计数」。如果安装了滑鼠，显示计数会被初始化为 0；否则，显示计数会被初始化为-1。只有在显示计数非负时才显示滑鼠游标。要增

加显示计数，您可以呼叫：

```
ShowCursor (TRUE) ;
```

要减少显示计数，可以呼叫：

```
ShowCursor (FALSE) ;
```

您在使用 ShowCursor 之前，不需要确定是否安装了滑鼠。如果您想显示滑鼠游标，而不管滑鼠存在与否，那么只需呼叫 ShowCursor 来增加显示计数。增加一次显示计数之後，如果没有安装滑鼠则减少它以隐藏游标，如果安装了滑鼠，则保留其显示。

即使没有安装滑鼠，Windows 也保留了滑鼠目前的位置。如果没有安装滑鼠，而您又显示滑鼠游标，游标就可能出现在显示器的任意位置，直到您确实移动了它。要获得游标的位置，可以呼叫：

```
GetCursorPos (&pt) ;
```

其中 pt 是 POINT 结构。函式使用滑鼠的 x 和 y 座标来填入 POINT 栏位。要设定游标位置，可以使用：

```
SetCursorPos (x, y) ;
```

在这两种情况下，x 和 y 都是萤幕座标，而不是显示区域座标（这是很明显的，因为这些函式没有要求 hwnd 参数）。前面已经提到过，呼叫 ScreenToClient 和 ClientToScreen 就能做到萤幕座标与客户座标的相互转换。

如果您在处理滑鼠讯息并转换显示区域座标时呼叫 GetCursorPos ，这些座标可能与滑鼠讯息的 lParam 参数中的座标稍微有些不同。从 GetCursorPos 传回的座标表示滑鼠目前的位置。lParam 中的座标则是产生讯息时滑鼠的位置。

您或许想写一个键盘处理程式：使用键盘方向键来移动滑鼠游标，使用 Spacebar 和 Enter 键来模拟滑鼠按键。您肯定不希望每次按键只是将滑鼠游标移动一个图素，如果这样做，当要把滑鼠游标从显示器的一边移动到另一边时，会使用者在很长一段时间内都要按住同一个方向键。

如果您需要实作滑鼠游标的键盘介面，并保持游标的精确定位能力，那么您可以采用下面的方式来处理按键讯息：当按下方向键时，一开始滑鼠游标移动较慢，但随後会加快。您也许还记得 WM_KEYDOWN 讯息中的 lParam 参数标志著按键讯息是否是重复活动的结果，这就是此参数的一个重要应用。

在 CHECKER 中加入键盘介面

程式 7-3 所示的 CHECKER2 程式，除了包括键盘介面外，和 CHECKER1 是一样的，您可以使用左、右、上和下方向键在 25 个矩形之间移动游标。Home 键把游标移动到矩形的左上角， End 键把游标移动到矩形的右下角。Spacebar 和 Enter 键都能切换 X 标记。

程式 7-3 CHECKER2

CHECKER2.C

```
/*------------------------------------------------------------------------
--
   CHECKER2.C --  Mouse Hit-Test Demo Program No. 2
                                  (c) Charles Petzold, 1998
------------------------------------------------------------------------
*/

#include <windows.h>

#define DIVISIONS 5

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                  PSTR szCmdLine, int iCmdShow)
{
      static TCHAR szAppName[] = TEXT ("Checker2") ;
      HWND                              hwnd ;
      MSG                         msg ;
      WNDCLASS                    wndclass ;

      wndclass.style                          = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc                    = WndProc ;
      wndclass.cbClsExtra                     = 0 ;
      wndclass.cbWndExtra                     = 0 ;
      wndclass.hInstance                      = hInstance ;
      wndclass.hIcon                          =        LoadIcon        (NULL,
IDI_APPLICATION) ;
      wndclass.hCursor                              =    LoadCursor    (NULL,
IDC_ARROW) ;
      wndclass.hbrBackground            =       (HBRUSH)       GetStockObject
(WHITE_BRUSH) ;
      wndclass.lpszMenuName             = NULL ;
      wndclass.lpszClassName            = szAppName ;

      if (!RegisterClass (&wndclass))
      {
            MessageBox (     NULL, TEXT ("Program requires Windows NT!"),
                                    szAppName, MB_ICONERROR) ;
                  return 0 ;
      }
      hwnd = CreateWindow (  szAppName, TEXT ("Checker2 Mouse Hit-Test Demo"),
                  WS_OVERLAPPEDWINDOW,
                  CW_USEDEFAULT, CW_USEDEFAULT,
                  CW_USEDEFAULT, CW_USEDEFAULT,
                  NULL, NULL, hInstance, NULL) ;
```

```
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (    GetMessage (&msg, NULL, 0, 0))
     {
                    TranslateMessage (&msg) ;
                    DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static BOOL fState[DIVISIONS][DIVISIONS] ;
     static int       cxBlock, cyBlock ;
     HDC                         hdc ;
     int                         x, y ;
     PAINTSTRUCT                   ps ;
     POINT                       point ;
     RECT                        rect ;

     switch (message)
     {
     case  WM_SIZE :
               cxBlock = LOWORD (lParam) / DIVISIONS ;
               cyBlock = HIWORD (lParam) / DIVISIONS ;
               return 0 ;

     case  WM_SETFOCUS :
               ShowCursor (TRUE) ;
               return 0 ;

     case  WM_KILLFOCUS :
               ShowCursor (FALSE) ;
               return 0 ;

     case  WM_KEYDOWN :
               GetCursorPos (&point) ;
               ScreenToClient (hwnd, &point) ;
               x = max (0, min (DIVISIONS - 1, point.x / cxBlock)) ;
               y = max (0, min (DIVISIONS - 1, point.y / cyBlock)) ;

               switch (wParam)
               {
                    case VK_UP :
                                        y-- ;
                                        break ;
```

```
                    case  VK_DOWN :
                                        y++ ;
                                        break ;

                    case  VK_LEFT :
                                        x-- ;
                                        break ;

                    case  VK_RIGHT :
                                        x++ ;
                                        break ;

                    case  VK_HOME :
                                        x = y = 0 ;
                                        break ;

                    case  VK_END :
                                        x = y = DIVISIONS - 1 ;
                                        break ;

                    case  VK_RETURN :
                    case  VK_SPACE :
                                        SendMessage  (hwnd,  WM_LBUTTONDOWN,
MK_LBUTTON,
                              MAKELONG (x * cxBlock, y * cyBlock)) ;
                                        break ;
               }
               x = (x + DIVISIONS) % DIVISIONS ;
                    y = (y + DIVISIONS) % DIVISIONS ;

                    point.x = x * cxBlock + cxBlock / 2 ;
                    point.y = y * cyBlock + cyBlock / 2 ;

                    ClientToScreen (hwnd, &point) ;
                    SetCursorPos (point.x, point.y) ;
               return 0 ;
     case  WM_LBUTTONDOWN :
          x = LOWORD (lParam) / cxBlock ;
          y = HIWORD (lParam) / cyBlock ;

          if (x < DIVISIONS && y < DIVISIONS)
               {
                                        fState[x][y] ^= 1 ;

                                        rect.left    = x * cxBlock ;
                                        rect.top     = y * cyBlock ;
                                        rect.right   = (x + 1) * cxBlock ;
```

```
                                        rect.bottom   = (y + 1) * cyBlock ;

                                        InvalidateRect (hwnd, &rect, FALSE) ;
          }
                    else
                                        MessageBeep (0) ;
                    return 0 ;
       case  WM_PAINT :
             hdc = BeginPaint (hwnd, &ps) ;

                    for (x = 0 ; x < DIVISIONS ; x++)
                    for (y = 0 ; y < DIVISIONS ; y++)
                    {
                              Rectangle (hdc, x * cxBlock, y * cyBlock,
                              (x + 1) * cxBlock, (y + 1) * cyBlock) ;

                              if (fState [x][y])
                              {
                    MoveToEx (hdc,  x   *cxBlock,  y   *cyBlock, NULL) ;
                              LineTo   (hdc, (x+1)*cxBlock, (y+1)*cyBlock) ;
                              MoveToEx  (hdc,   x     *cxBlock,  (y+1)*cyBlock,
NULL) ;
                    LineTo   (hdc, (x+1)*cxBlock,  y   *cyBlock) ;
                              }
                    }
                    EndPaint (hwnd, &ps) ;
                    return 0 ;

       case  WM_DESTROY :
                    PostQuitMessage (0) ;
                    return 0 ;
       }
       return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

CHECKER2 中的 WM_KEYDOWN 的处理方式决定游标的位置(用 GetCursorPos)，把萤幕座标转换为显示区域座标（用 ScreenToClient），并用矩形方块的宽度和高度来除这个座标。这会产生指示矩形位置的 x 和 y 值（5×5 阵列）。当按下一个键时，滑鼠游标可能在或不在显示区域中，所以 x 和 y 必须经过 min 和 max 巨集处理以保证它们的范围是 0 到 4 之间。

对方向键,CHECKER2 近似地增加或减少 x 和 y。如果是 Enter 键或 Spacebar 键，那么 CHECKER2 使用 SendMessage 把 WM_LBUTTONDOWN 讯息发送给它自身。这种技术类似于在第六章 SYSMETS 程式中把键盘介面加到视窗卷动列时所使用的方法。WM_KEYDOWN 的处理方式是通过计算指向矩形中心的显示区域座标，再用 ClientToScreen 转换成萤幕座标，然後用 SetCursorPos 设定游标位置来实

作的。

将子视窗用於命中测试

有些程式（例如，Windows 的「画图」程式），把显示区域划分为几个小的逻辑区域。「画图」程式在其左边有一个由图示组成的工具功能表区，在底部有颜色功能表区。在这两个区做命中测试的时候，「画图」必须在使用者选中功能表项之前记住功能表的位置。

不过，也可能不需要这么做。实际上，画风经由使用子视窗简化了功能表的绘制和命中测试。子视窗把整个矩形区域划分为几个更小的矩形区，每个子视窗有自己的视窗代号、视窗讯息处理程式和显示区域，每个视窗讯息处理程式接收只适用於它的子视窗的滑鼠讯息。滑鼠讯息中的 lParam 参数含有相当於该子视窗显示区域左上角的座标，而不是其父视窗（那是「画图」的主应用程式视窗）显示区域左上角的座标。

以这种方式使用子视窗有助於程式的结构化和模组化。如果子视窗使用不同的视窗类别，那么每个子视窗都有它自己的视窗讯息处理程式。不同的视窗也可以定义不同的背景颜色和不同的内定游标。在第九章中，我将看到「子视窗控制项」——卷动列、按钮和编辑方块等预先定义的子视窗。现在，我们说明在 CHECKER 程式中是如何使用子视窗的。

CHECKER 中的子视窗

程式 7-4 所示的 CHECKER3 程式，这一版本建立了 25 个处理滑鼠单击的子视窗。它没有键盘介面，但是可以按本章後面的 CHECKER4 程式范例的方法添加。

程式 7-4　CHECKER3

```
CHECKER3.C
/*-------------------------------------------------------------------------
-
   CHECKER3.C --  Mouse Hit-Test Demo Program No. 3
                                        (c) Charles Petzold, 1998
-------------------------------------------------------------------------
*/

#include <windows.h>

#define DIVISIONS 5

LRESULT CALLBACK WndProc   (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK ChildWndProc (HWND, UINT, WPARAM, LPARAM) ;
TCHAR szChildClass[]                          = TEXT ("Checker3_Child") ;
```

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                          PSTR szCmdLine, int iCmdShow)
{
     static TCHAR     szAppName[] = TEXT ("Checker3") ;
     HWND                              hwnd ;
     MSG                          msg ;
     WNDCLASS                     wndclass ;

     wndclass.style               = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc         = WndProc ;
     wndclass.cbClsExtra          = 0 ;
     wndclass.cbWndExtra          = 0 ;
     wndclass.hInstance           = hInstance ;
     wndclass.hIcon               = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor             = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground       = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName        = NULL ;
     wndclass.lpszClassName       = szAppName ;
     if (!RegisterClass (&wndclass))
     {
          MessageBox (     NULL, TEXT ("Program requires Windows NT!"),
                                          szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     wndclass.lpfnWndProc              = ChildWndProc ;
     wndclass.cbWndExtra          = sizeof (long) ;
     wndclass.hIcon               = NULL ;
     wndclass.lpszClassName       = szChildClass ;

     RegisterClass (&wndclass) ;
     hwnd = CreateWindow (  szAppName, TEXT ("Checker3 Mouse Hit-Test Demo"),
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
                    TranslateMessage (&msg) ;
                    DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}
```

```
LRESULT CALLBACK WndProc (  HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
      static HWND      hwndChild[DIVISIONS][DIVISIONS] ;
      int                          cxBlock, cyBlock, x, y ;

      switch (message)
     {
      case  WM_CREATE :
                  for (x = 0 ; x < DIVISIONS ; x++)
                        for (y = 0 ; y < DIVISIONS ; y++)
                  hwndChild[x][y] = CreateWindow (szChildClass, NULL,
                  WS_CHILDWINDOW | WS_VISIBLE,
                  0, 0, 0, 0,
                   hwnd, (HMENU) (y << 8 | x),
                  (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                                                                    NULL) ;
                  return 0 ;

      case  WM_SIZE :
                  cxBlock = LOWORD (lParam) / DIVISIONS ;
                  cyBlock = HIWORD (lParam) / DIVISIONS ;
                  for (x = 0 ; x < DIVISIONS ; x++)
                                   for (y = 0 ; y < DIVISIONS ; y++)
                                             MoveWindow                    (
      hwndChild[x][y],
                                        x * cxBlock, y * cyBlock,
                                           cxBlock, cyBlock, TRUE) ;
                  return 0 ;

      case  WM_LBUTTONDOWN :
                  MessageBeep (0) ;
                  return 0 ;

      case  WM_DESTROY :
                  PostQuitMessage (0) ;
                  return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}

LRESULT CALLBACK ChildWndProc (HWND hwnd, UINT message,
              WPARAM wParam, LPARAM lParam)
{
      HDC                                hdc ;
      PAINTSTRUCT                         ps ;
      RECT                               rect ;
```

```
     switch (message)
     {
     case  WM_CREATE :
                SetWindowLong (hwnd, 0, 0) ;      // on/off flag
                return 0 ;

     case  WM_LBUTTONDOWN :
                SetWindowLong (hwnd, 0, 1 ^ GetWindowLong (hwnd, 0)) ;
                InvalidateRect (hwnd, NULL, FALSE) ;
                return 0 ;

     case  WM_PAINT :
                hdc = BeginPaint (hwnd, &ps) ;

                GetClientRect (hwnd, &rect) ;
                Rectangle (hdc, 0, 0, rect.right, rect.bottom) ;

                if (GetWindowLong (hwnd, 0))
                {
                           MoveToEx (hdc, 0,         0, NULL) ;
                           LineTo   (hdc, rect.right, rect.bottom) ;
                           MoveToEx (hdc, 0,         rect.bottom, NULL) ;
                           LineTo   (hdc, rect.right, 0) ;
          }

                EndPaint (hwnd, &ps) ;
                return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

CHECKER3 有两个视窗讯息处理程式 WndProc 和 ChildWndProc。WndProc 还是主（或父）视窗的视窗讯息处理程式。ChildWndProc 是针对 25 个子视窗的视窗讯息处理程式。这两个视窗讯息处理程式都必须定义为 CALLBACK 函式。

因为视窗讯息处理程式与特定的视窗类别结构相关联，该视窗类别结构由 Windows 呼叫 RegisterClass 函式来注册，CHECKER3 需要两个视窗类别。第一个视窗类别用於主视窗，名为「Checker3」。第二个视窗类别名为「Checker3_Child」。当然，您不必选择像这样有意义的名字。

CHECKER3 在 WinMain 函式中注册了这两个视窗类别。注册完常规的视窗类别之後，CHECKER3 只是简单地重新使用 wndclass 结构中的大多数的栏位来注册 Checker3_Child 类别。无论如何，有四个栏位根据子视窗类别而设定为不同的值：

- pfnWndProc 栏位设定为 ChildWndProc，子视窗类别的视窗讯息处理程式。

- cbWndExtra 栏位设定为 4 位元组，或者更确切地用 sizeof (long)。该栏位告诉 Windows 在其为依据此视窗类别的视窗保留的内部结构中，预留了 4 位元组额外的空间。您能使用此空间来保存每个视窗的可能有所不同的资讯。
- 因为像 CHECKER3 中的子视窗不需要图示,所以 hIcon 栏位设定为 NULL 。
- pszClassName 栏位设定为「Checker3_Child」，是类别的名称。

通常,在 WinMain 中,CreateWindow 呼叫建立依据 Checker3 类别的主视窗。然而，当 WndProc 收到 WM_CREATE 讯息後，它呼叫 CreateWindow 25 次以建立 25 个 Checker3_Child 类别的子视窗。表 7-3 是在 WinMain 中 CreateWindow 呼叫的参数，与在建立 25 个子视窗的 WndProc 中 CreateWindow 呼叫的参数间的比较。

表 7-3

参数	主视窗	子视窗
视窗类别	「Checker3」	「Checker3_Child」
视窗标题	「Checker3...」	NULL
视窗样式	WS_OVERLAPPEDWINDOW	WS_CHILDWINDOW | WS_VISIBLE
水平位置	CW_USEDEFAULT	0
垂直位置	CW_USEDEFAULT	0
宽度	CW_USEDEFAULT	0
高度	CW_USEDEFAULT	0
父视窗代号	NULL	hwnd
功能表代号/子 ID	NULL	(HMENU) (y << 8 | x)
执行实体代号	hInstance	(HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE)
额外参数	NULL	NULL

一般情况下，子视窗要求有关位置和大小的参数，但是在 CHECKER3 中的子视窗由 WndProc 确定位置和大小。对於主视窗，因为它本身就是父视窗，所以它的父视窗代号是 NULL。当使用 CreateWindow 呼叫来建立一个子视窗时，就需要父视窗代号了。

主视窗没有功能表，因此参数是 NULL。对於子视窗，相同位置的参数称为子 ID（或子视窗 ID）。这是唯一代表子视窗的数字。像我们在第十一章将看到的一样,在处理对话方块的子视窗控制项时,子 ID 显得更为重要。对於 CHECKER3 来说，我只是简单地将子 ID 设定为一个数值，该数值是每个子视窗在 5×5 的主视窗中的 x 和 y 位置的组合。

CreateWindow 函式需要一个执行实体代号。在 WinMain 中，执行实体代号可以很容易地取得，因为它是 WinMain 的一个参数。在建立子视窗时，CHECKER3 必须用 GetWindowLong 来从 Windows 为视窗保留的结构中取得 hInstance 值（相对於 GetWindowLong，我也能将 hInstance 的值保存到整体变数，并直接使用它）。

每一个子视窗都在 hwndChild 阵列中保存了不同的视窗代号。当 WndProc 接收到一个 WM_SIZE 讯息後，它将为这 25 个子视窗呼叫 MoveWindow。MoveWindow 的参数表示子视窗左上角相对於父视窗显示区域的座标、子视窗的宽度和高度以及子视窗是否需要重画。

现在让我们看一下 ChildWndProc。此视窗讯息处理程式为所有这 25 个子视窗处理讯息。ChildWndProc 的 hwnd 参数是子视窗接收讯息的代号。当 ChildWndProc 处理 WM_CREATE 讯息时（因为有 25 个子视窗，所以要发生 25 次），它用 SetWindowWord 在视窗结构保留的额外区域中储存一个 0 值（通过在定义视窗类别时使用的 cbWndExtra 来保留的空间）。ChildWndProc 用此值来恢复目前矩形的状态（有 X 或没有 X）。在子视窗中单击时，WM_LBUTTONDOWN 处理常式简单地修改这个整数值（从 0 到 1，或从 1 到 0），并使整个子视窗无效。此区域是被单击的矩形。WM_PAINT 的处理很简单，因为它所绘制的矩形与显示区域一样大。

因为 CHECKER3 的 C 原始码档案和.EXE 档案比 CHECKER1 的大（更不用说程式的说明了），我不会试著告诉你说 CHECKER3 比 CHECKER1 更简单。但请注意，我们没有做任何的滑鼠命中测试！我们所要的，就是知道 CHECKER3 中是否有个子视窗得到了命中视窗的 WM_LBUTTONDOWN 讯息。

子视窗和键盘

为 CHECKER3 添加键盘介面就像 CHECKER 系列构想中的最後一步。但在这样做的时候，可能有更适当的做法。在 CHECKER2 中，滑鼠游标的位置决定按下 Spacebar 键时哪个区域将获得标记符号。当我们处理子视窗时，我们能从对话方块功能中获得提示。在对话方块中，带有闪烁的插入符号或点划的矩形的子视窗表示它有输入焦点（当然也可以用键盘进行定位）。

我们不需要把 Windows 内部已有的对话方块处理方式重新写过，我只是要告诉您大致上应该如何在应用程式中模拟对话方块。研究过程中，您会发现这样一件事：父视窗和子视窗可能要共用同键盘讯息处理。按下 Spacebar 键和 Enter 键时，子视窗将锁定复选标记。按下方向键时，父视窗将在子视窗之间移动输入焦点。实际上，当您在子视窗上单击时，情况会有些复杂，这时是父视窗而不是子视窗获得输入焦点。

CHECKER4.C 如程式 7-5 所示。

程式 7-5 CHECKER4

```
CHECKER4.C
/*-----------------------------------------------------------------------
-
   CHECKER4.C --  Mouse Hit-Test Demo Program No. 4
                                  (c) Charles Petzold, 1998
-----------------------------------------------------------------------*
/

#include <windows.h>
#define DIVISIONS 5

LRESULT CALLBACK WndProc   (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK ChildWndProc (HWND, UINT, WPARAM, LPARAM) ;
int   idFocus = 0 ;
TCHAR szChildClass[] = TEXT ("Checker4_Child") ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                             PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("Checker4") ;
      HWND                                hwnd ;
      MSG                           msg ;
      WNDCLASS                         wndclass ;

      wndclass.style                           = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc                     = WndProc ;
      wndclass.cbClsExtra                      = 0 ;
      wndclass.cbWndExtra                      = 0 ;
      wndclass.hInstance                       = hInstance ;
      wndclass.hIcon                           =        LoadIcon        (NULL,
IDI_APPLICATION) ;
      wndclass.hCursor                               =    LoadCursor    (NULL,
IDC_ARROW) ;
      wndclass.hbrBackground             =       (HBRUSH)       GetStockObject
(WHITE_BRUSH) ;
      wndclass.lpszMenuName              = NULL ;
      wndclass.lpszClassName             = szAppName ;

      if (!RegisterClass (&wndclass))
      {
                  MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                                                         szAppName,
MB_ICONERROR) ;
                  return 0 ;
      }
```

```
     wndclass.lpfnWndProc                    = ChildWndProc ;
     wndclass.cbWndExtra                     = sizeof (long) ;
     wndclass.hIcon                          = NULL ;
     wndclass.lpszClassName            = szChildClass ;

     RegisterClass (&wndclass) ;
     hwnd = CreateWindow (  szAppName, TEXT ("Checker4 Mouse Hit-Test Demo"),
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    NULL, NULL, hInstance, NULL) ;
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
                    TranslateMessage (&msg) ;
                    DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (   HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
     static HWND      hwndChild[DIVISIONS][DIVISIONS] ;
     int                         cxBlock, cyBlock, x, y ;

     switch (message)
     {
     case WM_CREATE :
                for (x = 0 ; x < DIVISIONS ; x++)
                                for (y = 0 ; y < DIVISIONS ; y++)
           hwndChild[x][y] = CreateWindow (szChildClass, NULL,
                     WS_CHILDWINDOW | WS_VISIBLE,
                     0, 0, 0, 0,
                     hwnd, (HMENU) (y << 8 | x),
                     HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                                     NULL) ;
                return 0 ;
     case  WM_SIZE :
                cxBlock = LOWORD (lParam) / DIVISIONS ;
                cyBlock = HIWORD (lParam) / DIVISIONS ;

                for (x = 0 ; x < DIVISIONS ; x++)
                                for (y = 0 ; y < DIVISIONS ; y++)
                     MoveWindow (     hwndChild[x][y],
```

```
                    x * cxBlock, y * cyBlock,
                        cxBlock, cyBlock, TRUE) ;
               return 0 ;

     case  WM_LBUTTONDOWN :
               MessageBeep (0) ;
               return 0 ;

               // On set-focus message, set focus to child window
     case  WM_SETFOCUS:
               SetFocus (GetDlgItem (hwnd, idFocus)) ;
               return 0 ;

          // On key-down message, possibly change the focus window

     case  WM_KEYDOWN:
               x = idFocus & 0xFF ;
               y = idFocus >> 8 ;

               switch (wParam)
               {
      case VK_UP:           y-- ;
     break ;
          case VK_DOWN:               y++ ;
          break ;
          case VK_LEFT:               x-- ;
     break ;
      case VK_RIGHT:           x++ ;                        break ;
      case VK_HOME:              x = y = 0 ;                  break ;
      case VK_END:               x = y = DIVISIONS - 1 ;         break ;
               default:                                 return 0 ;
               }

               x = (x + DIVISIONS) % DIVISIONS ;
               y = (y + DIVISIONS) % DIVISIONS ;

               idFocus = y << 8 | x ;

               SetFocus (GetDlgItem (hwnd, idFocus)) ;
               return 0 ;

     case  WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

```
LRESULT CALLBACK ChildWndProc (HWND hwnd, UINT message,
                WPARAM wParam, LPARAM lParam)
{
      HDC                                hdc ;
      PAINTSTRUCT                         ps ;
      RECT                               rect ;

      switch (message)
     {
      case  WM_CREATE :
                  SetWindowLong (hwnd, 0, 0) ;      // on/off flag
                  return 0 ;

      case  WM_KEYDOWN:
                              // Send most key presses to the parent window

                  if (wParam != VK_RETURN && wParam != VK_SPACE)
                  {
                        SendMessage (GetParent (hwnd), message, wParam, lParam) ;
                        return 0 ;
                  }
                              // For Return and Space, fall through to toggle the
square

      case  WM_LBUTTONDOWN :
                  SetWindowLong (hwnd, 0, 1 ^ GetWindowLong (hwnd, 0)) ;
                  SetFocus (hwnd) ;
                  InvalidateRect (hwnd, NULL, FALSE) ;
                  return 0 ;

                  // For focus messages, invalidate the window for repaint

      case  WM_SETFOCUS:
                  idFocus = GetWindowLong (hwnd, GWL_ID) ;

                  // Fall through

      case  WM_KILLFOCUS:
                  InvalidateRect (hwnd, NULL, TRUE) ;
                  return 0 ;

         case     WM_PAINT :
                  hdc = BeginPaint (hwnd, &ps) ;

                  GetClientRect (hwnd, &rect) ;
                  Rectangle (hdc, 0, 0, rect.right, rect.bottom) ;

                              // Draw the "x" mark
```

```
                if (GetWindowLong (hwnd, 0))
                {
                        MoveToEx (hdc, 0,         0, NULL) ;
                        LineTo   (hdc, rect.right, rect.bottom) ;
                        MoveToEx (hdc, 0,         rect.bottom, NULL) ;
                        LineTo   (hdc, rect.right, 0) ;
                }
                        // Draw the "focus" rectangle

                if (hwnd == GetFocus ())
                {
                        rect.left   += rect.right / 10 ;
                        rect.right  -= rect.left ;
                        rect.top    += rect.bottom / 10 ;
                        rect.bottom -= rect.top ;

                        SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
                        SelectObject (hdc, CreatePen (PS_DASH, 0, 0)) ;
                        Rectangle (hdc, rect.left, rect.top, rect.right,
rect.bottom) ;
                        DeleteObject (SelectObject (hdc, GetStockObject
(BLACK_PEN))) ;
                }

                EndPaint (hwnd, &ps) ;
                return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

您应该能回忆起每一个子视窗有唯一的子视窗 ID，该 ID 在呼叫 CreateWindow 建立视窗时定义。在 CHECKER3 中，此 ID 是矩形的 x 和 y 位置的组合。一个程式可以通过下面的呼叫来获得一个特定子视窗的子视窗 ID：

```
idChild = GetWindowLong (hwndChild, GWL_ID) ;
```

下面的函式也有同样的功能：

```
idChild = GetDlgCtrlID (hwndChild) ;
```

正如函式名称所表示的，它主要用於对话方块和控制视窗。如果您知道父视窗的代号和子视窗 ID，此函式也可以获得子视窗的代号：

```
hwndChild = GetDlgItem (hwndParent, idChild) ;
```

在 CHECKER4 中，整体变数 idFocus 用於保存目前输入焦点视窗的子视窗 ID。我在前面说过，当您在子视窗上面单击滑鼠时，它们不会自动获得输入焦点。因此，CHECKER4 中的父视窗将通过呼叫下面的函式来处理 WM_SETFOCUS 讯息：

```
SetFocus (GetDlgItem (hwnd, idFocus)) ;
```

这样设定一个子视窗为输入焦点。

ChildWndProc 处理 WM_SETFOCUS 和 WM_KILLFOCUS 讯息。对於 WM_SETFOCUS，它将保存在整体变数 idFocus 中接收输入焦点的子视窗 ID。对於这两种讯息，视窗是无效的，并产生一个 WM_PAINT 讯息。如果 WM_PAINT 讯息画出了有输入焦点的子视窗，则它将用 PS_DASH 画笔的风格画一个矩形以表示此视窗有输入焦点。

ChildWndProc 也处理 WM_KEYDOWN 讯息。对於除了 Spacebar 和 Enter 键以外的其他讯息，WM_KEYDOWN 都将给父视窗发送讯息。另外，视窗讯息处理程式也处理类似 WM_LBUTTONDOWN 讯息的讯息。

处理方向移动键是父视窗的事情。在风格相似的 CHECKER2 中，此程式可获得有输入焦点的子视窗的 x 和 y 座标，并根据按下的特定方向键来改变它们。然後通过呼叫 SetFocus 将输入焦点设定给新的子视窗。

拦截滑鼠

一个视窗讯息处理程式通常只在滑鼠游标位於视窗的显示区域，或非显示区域上时才接收滑鼠讯息。一个程式也可能需要在滑鼠位於视窗外时接收滑鼠讯息。如果是这样，程式可以自行「拦截」滑鼠。别害怕，这么做没什么大不了的。

设计矩形

为了说明拦截滑鼠的必要性，请让我们看一下 BLOKOUT1 程式（如程式 7-6 所示）。此程式看起来达到了一定的功能，但它却有十分严重的缺陷。

程式 7-6　BLOKOUT1

```
BLOKOUT1.C
/*-------------------------------------------------------------------------
--
   BLOKOUT1.C --  Mouse Button Demo Program
                                    (c) Charles Petzold, 1998
-------------------------------------------------------------------------
*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                                PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("BlokOut1") ;
      HWND                                hwnd ;
```

```
     MSG                         msg ;
     WNDCLASS                    wndclass ;

     wndclass.style                          = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc                    = WndProc ;
     wndclass.cbClsExtra                     = 0 ;
     wndclass.cbWndExtra                     = 0 ;
     wndclass.hInstance                      = hInstance ;
     wndclass.hIcon                          =        LoadIcon        (NULL,
IDI_APPLICATION) ;
     wndclass.hCursor                        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground            =       (HBRUSH)       GetStockObject
(WHITE_BRUSH) ;
     wndclass.lpszMenuName             = NULL ;
     wndclass.lpszClassName            = szAppName ;

     if (!RegisterClass (&wndclass))
     {
               MessageBox (     NULL, TEXT ("Program requires Windows NT!"),
                                                      szAppName,
MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Mouse Button Demo"),
                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT, CW_USEDEFAULT,
                CW_USEDEFAULT, CW_USEDEFAULT,
                NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void DrawBoxOutline (HWND hwnd, POINT ptBeg, POINT ptEnd)
{
     HDC hdc ;
     hdc = GetDC (hwnd) ;
     SetROP2 (hdc, R2_NOT) ;
     SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
     Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y) ;
```

```
     ReleaseDC (hwnd, hdc) ;
}
LRESULT CALLBACK WndProc (  HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
     static BOOL      fBlocking, fValidBox ;
     static POINT     ptBeg, ptEnd, ptBoxBeg, ptBoxEnd ;
     HDC                         hdc ;
     PAINTSTRUCT  ps ;
     switch (message)
     {
     case  WM_LBUTTONDOWN :
               ptBeg.x = ptEnd.x = LOWORD (lParam) ;
               ptBeg.y = ptEnd.y = HIWORD (lParam) ;

               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

               SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

               fBlocking = TRUE ;
               return 0 ;

     case  WM_MOUSEMOVE :
               if (fBlocking)
               {
                         SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

                         DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

                         ptEnd.x = LOWORD (lParam) ;
                         ptEnd.y = HIWORD (lParam) ;

                         DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
               }
               return 0 ;

     case  WM_LBUTTONUP :
               if (fBlocking)
               {
                         DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

                         ptBoxBeg                    = ptBeg ;
                         ptBoxEnd.x                  = LOWORD (lParam) ;
                         ptBoxEnd.y                  = HIWORD (lParam) ;

                         SetCursor (LoadCursor (NULL, IDC_ARROW)) ;
                         fBlocking                   = FALSE ;
```

```
                              fValidBox                    = TRUE ;

                              InvalidateRect (hwnd, NULL, TRUE) ;
                    }
                    return 0 ;

      case  WM_CHAR :
                    if (fBlocking & wParam == '\x1B')      // i.e., Escape
                    {
                              DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

                              SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

                              fBlocking = FALSE ;
                    }
                    return 0 ;

      case  WM_PAINT :
                    hdc = BeginPaint (hwnd, &ps) ;

            if (fValidBox)
                    {
                              SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
                              Rectangle   (     hdc, ptBoxBeg.x, ptBoxBeg.y,
                                                  ptBoxEnd.x, ptBoxEnd.y) ;
                    }

            if (fBlocking)
                    {
                              SetROP2 (hdc, R2_NOT) ;
                              SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
                              Rectangle   (hdc,   ptBeg.x,   ptBeg.y,   ptEnd.x,
ptEnd.y) ;
                    }

                    EndPaint (hwnd, &ps) ;
                    return 0 ;

      case  WM_DESTROY :
                    PostQuitMessage (0) ;
                    return 0 ;
    }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

此程式展示了一些，它可以实作在 Windows 的「画图」程式中的东西。由按下滑鼠左键开始确定矩形的一角，然後拖动滑鼠。程式将画一个矩形的轮廓，其相对位置是滑鼠目前的位置。当您释放滑鼠後，程式将填入这个矩形。图 7-4

显示了一个已经画完的矩形和另一个正在画的矩形。

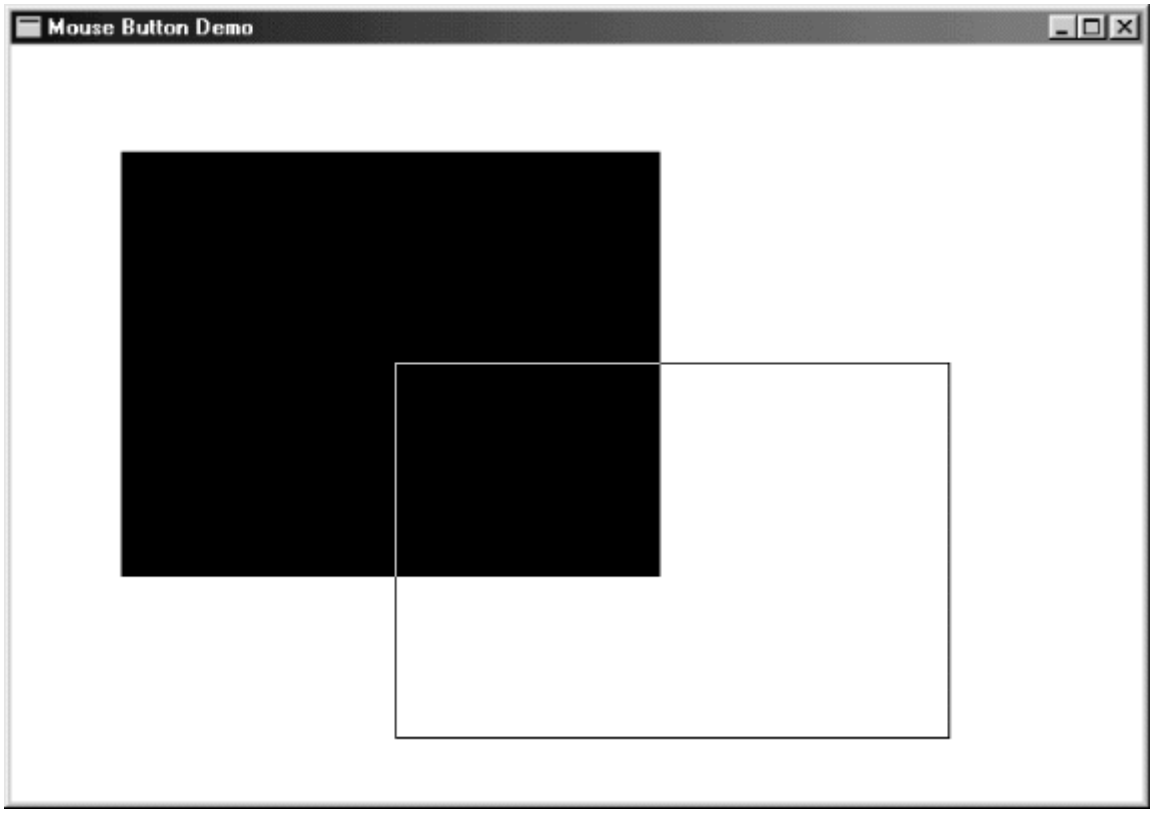

图 7-4 BLOKOUT1 的萤幕显示

那么，问题在哪里呢？

请试一试下面的操作：在 BLOKOUT1 的显示区域按下滑鼠的左键，然後将游标移出视窗。程式将停止接收 WM_MOUSEMOVE 讯息。现在释放按钮，BLOKOUT1 将不再获得 WM_BUTTONUP 讯息，因为游标在显示区域以外。然後将游标移回 BLOKOUT1 的显示区域，视窗讯息处理程式仍然认为按钮处於按下状态。

这样做并不好，因为程式不知道发生了什么事情。

拦截的解决方案

BLOKOUT1 显示了一些常见的程式功能，但它的程式码显然有缺陷。这种问题就是要使用滑鼠拦截来对付。如果使用者正在拖曳滑鼠，那么当滑鼠短时间内被拖出视窗时应该没有什么大问题，程式应该仍然控制著滑鼠。

拦截滑鼠要比放置一个老鼠夹子容易一些，您只要呼叫：

```
SetCapture (hwnd) ;
```

在这个函式呼叫之後，Windows 将所有滑鼠讯息发给视窗代号为 hwnd 的视窗讯息处理程式。之後收到滑鼠讯息都是以显示区域讯息的型态出现，即使滑鼠正在视窗的非显示区域。lParam 参数将指示滑鼠在显示区域座标中的位置。不过，当滑鼠位於显示区域的左边或者上方时，这些 x 和 y 座标可以是负的。当您想释放滑鼠时，呼叫：

```
ReleaseCapture () ;
```

从而使处理恢复正常。

在 32 位元的 Windows 中，滑鼠拦截要比在以前的 Windows 版本中有多一些限制。特别是，如果滑鼠被拦截，而滑鼠按键目前并未被按下，并且滑鼠游标移到了另一个视窗上，那么将不是由拦截滑鼠的那个视窗，而是由游标下面的视窗来接收滑鼠讯息。对於防止一个程式在拦截滑鼠之後不释放它而引起整个系统的混乱，这是必要的。

换句话说，只有当滑鼠按键在您的显示区域中被按下时才拦截滑鼠；当滑鼠按键被释放时，才释放滑鼠拦截。

BLOKOUT2 程式

展示滑鼠拦截的 BLOKOUT2 程式如程式 7-7 所示。

程式 7-7　BLOKOUT2

```
BLOKOUT2.C
/*------------------------------------------------------------------------
--
      BLOKOUT2.C --          Mouse Button & Capture Demo Program
                                   (c) Charles Petzold, 1998
------------------------------------------------------------------------
*/

#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                   PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("BlokOut2") ;
      HWND                              hwnd ;
      MSG                         msg ;
      WNDCLASS                    wndclass ;

      wndclass.style                    = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc              = WndProc ;
      wndclass.cbClsExtra               = 0 ;
      wndclass.cbWndExtra               = 0 ;
      wndclass.hInstance                = hInstance ;
      wndclass.hIcon                    = LoadIcon (NULL, IDI_APPLICATION) ;
      wndclass.hCursor                        = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground      = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
      wndclass.lpszMenuName  = NULL ;
      wndclass.lpszClassName      = szAppName ;

      if (!RegisterClass (&wndclass))
```

```
     {
          MessageBox (     NULL, TEXT ("Program requires Windows NT!"),
                                   szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (  szAppName, TEXT ("Mouse Button & Capture Demo"),
                            WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT, CW_USEDEFAULT,
                CW_USEDEFAULT, CW_USEDEFAULT,
                NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
                            TranslateMessage (&msg) ;
                            DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void DrawBoxOutline (HWND hwnd, POINT ptBeg, POINT ptEnd)
{
     HDC hdc ;
     hdc = GetDC (hwnd) ;
     SetROP2 (hdc, R2_NOT) ;
     SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
     Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y) ;

     ReleaseDC (hwnd, hdc) ;
}

LRESULT CALLBACK WndProc (   HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
     static BOOL  fBlocking, fValidBox ;
     static POINT ptBeg, ptEnd, ptBoxBeg, ptBoxEnd ;
     HDC         hdc ;
     PAINTSTRUCT  ps ;

     switch (message)
     {
     case  WM_LBUTTONDOWN :
                ptBeg.x = ptEnd.x = LOWORD (lParam) ;
                ptBeg.y = ptEnd.y = HIWORD (lParam) ;
```

```
                DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

                SetCapture (hwnd) ;
                SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

                fBlocking = TRUE ;
                return 0 ;

        case  WM_MOUSEMOVE :
                if (fBlocking)
                {
                        SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

                        DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

                        ptEnd.x = LOWORD (lParam) ;
                        ptEnd.y = HIWORD (lParam) ;

                        DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                }
                return 0 ;

        case  WM_LBUTTONUP :
                if (fBlocking)
                {
                        DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

                        ptBoxBeg               = ptBeg ;
                        ptBoxEnd.x             = LOWORD (lParam) ;
                        ptBoxEnd.y             = HIWORD (lParam) ;

                        ReleaseCapture () ;
                        SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

                        fBlocking              = FALSE ;
                        fValidBox              = TRUE ;

                        InvalidateRect (hwnd, NULL, TRUE) ;
                }
                return 0 ;

        case  WM_CHAR :
                if (fBlocking & wParam == '\x1B')      // i.e., Escape
                {
                        DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                        ReleaseCapture () ;
                        SetCursor (LoadCursor (NULL, IDC_ARROW)) ;
```

```
                              fBlocking = FALSE ;
                    }
                    return 0 ;

      case  WM_PAINT :
                    hdc = BeginPaint (hwnd, &ps) ;

                    if (fValidBox)
         {
                              SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
                              Rectangle (hdc, ptBoxBeg.x, ptBoxBeg.y,
                              ptBoxEnd.x, ptBoxEnd.y) ;
                    }

                         if (fBlocking)
                    {
                              SetROP2 (hdc, R2_NOT) ;
                              SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
                              Rectangle   (hdc,   ptBeg.x,   ptBeg.y,   ptEnd.x,
ptEnd.y) ;
                    }

                    EndPaint (hwnd, &ps) ;
                    return 0 ;

      case  WM_DESTROY :
                    PostQuitMessage (0) ;
                    return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

BLOKOUT2 程式和 BLOKOUT1 程式一样，只是多了三行新程式码：在 WM_LBUTTONDOWN 讯息处理期间呼叫 SetCapture，而在 WM_LBUTTONDOWN 和 WM_CHAR 讯息处理期间呼叫 ReleaseCapture。检查画出视窗：使视窗小於萤幕大小，开始在显示区域画出一块矩形，然後将滑鼠游标移出显示区域的右边或下边，最後释放滑鼠按键。程式将获得整个矩形的座标。但是需要扩大视窗才能看清楚它。

拦截滑鼠并非只适用於那些古怪的应用程式。如果您需要滑鼠按键在显示区域按下时都能够追踪 WM_MOUSEMOVE 讯息，并直到滑鼠按键被释放为止，那么您就应该拦截滑鼠。这样将简化您的程式，同时又符合使用者的期望。

滑鼠滑轮

与传统的滑鼠相比，Microsoft IntelliMouse 的特点是在两个键之间多了

一个小滑轮。您可以按下这个滑轮，这时它的功能相当於滑鼠按键的中键；或者您也可以用食指来转动它，这会产生一条特殊的讯息，叫做 WM_MOUSEWHEEL。使用滑鼠滑轮的程式通过滚动或放大文件来回应此讯息。它最初听起来像一个不必要的隐藏机关，但我必须承认，我很快就习惯於使用滑鼠滑轮来滚动 Microsoft Word 和 Microsoft Internet Explorer 了。

我不想讨论滑鼠滑轮的所有使用方法。实际上，我只是想告诉您如何在现有的程式（例如程式 SYSMETS4）中添加滑鼠滑轮处理程式，以便在显示区域中卷动资料。最终的 SYSMETS 程式如程式 7-8 所示。

程式 7-8 SYSMETS4

```
SYSMETS.C
/*-----------------------------------------------------------------------------
--
SYSMETS.C -- Final System Metrics Display Program
                              (c) Charles Petzold, 1998
-----------------------------------------------------------------------------
*/

#include <windows.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                                PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("SysMets") ;
      HWND                                hwnd ;
      MSG                           msg ;
      WNDCLASS                      wndclass ;
      wndclass.style                              = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc                        = WndProc ;
      wndclass.cbClsExtra                         = 0 ;
      wndclass.cbWndExtra                         = 0 ;
      wndclass.hInstance                          = hInstance ;
      wndclass.hIcon                              =        LoadIcon        (NULL,
IDI_APPLICATION) ;
      wndclass.hCursor                            = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground              =       (HBRUSH)       GetStockObject
(WHITE_BRUSH) ;
      wndclass.lpszMenuName               = NULL ;
      wndclass.lpszClassName              = szAppName ;

      if (!RegisterClass (&wndclass))
      {
            MessageBox (     NULL, TEXT ("Program requires Windows NT!"),
                                                      szAppName,
```

```
MB_ICONERROR) ;
                return 0 ;
     }

     hwnd = CreateWindow (  szAppName, TEXT ("Get System Metrics"),
                      WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                     NULL, NULL, hInstance, NULL) ;

          ShowWindow (hwnd, iCmdShow) ;
          UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
                TranslateMessage (&msg) ;
                DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (   HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
     static     int   cxChar, cxCaps, cyChar, cxClient, cyClient, iMaxWidth ;
     static           int         iDeltaPerLine, iAccumDelta ;            //
for mouse wheel logic
          HDC       hdc ;
          int       i, x, y, iVertPos, iHorzPos, iPaintBeg, iPaintEnd ;
          PAINTSTRUCT            ps ;
          SCROLLINFO             si ;
          TCHAR                        szBuffer[10] ;
          TEXTMETRIC       tm ;
          ULONG                        ulScrollLines ;        //  for  mouse
wheel logic
     switch (message)
     {
     case  WM_CREATE:
                hdc = GetDC (hwnd) ;

                GetTextMetrics (hdc, &tm) ;
                cxChar     = tm.tmAveCharWidth ;
                cxCaps     = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
                cyChar = tm.tmHeight + tm.tmExternalLeading ;

                ReleaseDC (hwnd, hdc) ;

                                     // Save the width of the three columns
```

```
                iMaxWidth = 40 * cxChar + 22 * cxCaps ;

                                        //  Fall  through  for  mouse  wheel
information

      case  WM_SETTINGCHANGE:
                SystemParametersInfo      (SPI_GETWHEELSCROLLLINES,      0,
&ulScrollLines, 0) ;

                // ulScrollLines usually equals 3 or 0 (for no scrolling)
                // WHEEL_DELTA equals 120, so iDeltaPerLine will be 40

                if (ulScrollLines)
                                iDeltaPerLine = WHEEL_DELTA / ulScrollLines ;
                else
                                iDeltaPerLine = 0 ;

           return 0 ;

      case  WM_SIZE:
                cxClient = LOWORD (lParam) ;
           cyClient = HIWORD (lParam) ;

                                        // Set vertical scroll bar range and page
size

                si.cbSize                  = sizeof (si) ;
                si.fMask                     = SIF_RANGE | SIF_PAGE ;
                si.nMin                      = 0 ;
                si.nMax                      = NUMLINES - 1 ;
                si.nPage                     = cyClient / cyChar ;
                SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;

                                        // Set horizontal scroll bar range and
page size

                si.cbSize                  = sizeof (si) ;
                si.fMask                     = SIF_RANGE | SIF_PAGE ;
                si.nMin                      = 0 ;
                si.nMax                      = 2 + iMaxWidth / cxChar ;
                si.nPage                     = cxClient / cxChar ;
           SetScrollInfo (hwnd, SB_HORZ, &si, TRUE) ;
                return 0 ;

      case WM_VSCROLL:
                                        //  Get  all  the  vertical  scroll  bar
information
```

```
                si.cbSize = sizeof (si) ;
                si.fMask  = SIF_ALL ;
                GetScrollInfo (hwnd, SB_VERT, &si) ;

                                        // Save the position for comparison
later on
                iVertPos = si.nPos ;
           switch (LOWORD (wParam))
                {
                case  SB_TOP:
                                si.nPos = si.nMin ;
                                break ;

                case  SB_BOTTOM:
                                si.nPos = si.nMax ;
                                break ;

                case SB_LINEUP:
                                si.nPos -= 1 ;
                                break ;

                case  SB_LINEDOWN:
                                si.nPos += 1 ;
                                break ;

                case SB_PAGEUP:
                                si.nPos -= si.nPage ;
                                break ;

                case  SB_PAGEDOWN:
                                si.nPos += si.nPage ;
                                break ;

                case  SB_THUMBTRACK:
                                si.nPos = si.nTrackPos ;
                                break ;
                default:
                                break ;
                }
                                // Set the position and then retrieve it.  Due
to adjustments
                      //   by Windows it may not be the same as the value set.

                si.fMask = SIF_POS ;
                SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;
                GetScrollInfo (hwnd, SB_VERT, &si) ;

                                // If the position has changed, scroll the
```

```
window and update it

                if (si.nPos != iVertPos)
                {
                           ScrollWindow (   hwnd, 0, cyChar * (iVertPos -
si.nPos),
                                      NULL, NULL) ;
                           UpdateWindow (hwnd) ;
                }
                return 0 ;

     case  WM_HSCROLL:
                                // Get all the vertical scroll bar information

                si.cbSize = sizeof (si) ;
                si.fMask  = SIF_ALL ;

                                // Save the position for comparison later on

                GetScrollInfo (hwnd, SB_HORZ, &si) ;
                iHorzPos = si.nPos ;

                switch (LOWORD (wParam))
                {
                case  SB_LINELEFT:
                                si.nPos -= 1 ;
                                break ;

                case  SB_LINERIGHT:
                                si.nPos += 1 ;
                                break ;

                case  SB_PAGELEFT:
                                si.nPos -= si.nPage ;
                                break ;

                case  SB_PAGERIGHT:
                                si.nPos += si.nPage ;
                                break ;

                case  SB_THUMBPOSITION:
                                si.nPos = si.nTrackPos ;
                                break ;

                default:
                           break ;
                }
                           // Set the position and then retrieve it.  Due to
```

```
adjustments
                //   by Windows it may not be the same as the value set.

            si.fMask = SIF_POS ;
                SetScrollInfo (hwnd, SB_HORZ, &si, TRUE) ;
                GetScrollInfo (hwnd, SB_HORZ, &si) ;

                            // If the position has changed, scroll the window

                if (si.nPos != iHorzPos)
                {
                                ScrollWindow (   hwnd, cxChar * (iHorzPos -
si.nPos), 0,
                                                NULL, NULL) ;
                }
                return 0 ;

      case  WM_KEYDOWN :
                switch (wParam)
                {
                case  VK_HOME :
                            SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0) ;
                            break ;

                case  VK_END :
                            SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0) ;
                            break ;

                case  VK_PRIOR :
                            SendMessage (hwnd, WM_VSCROLL, SB_PAGEUP, 0) ;
                            break ;

                case  VK_NEXT :
                            SendMessage (hwnd, WM_VSCROLL, SB_PAGEDOWN, 0) ;
                            break ;

                case  VK_UP :
                            SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0) ;
                            break ;

                case  VK_DOWN :
                            SendMessage (hwnd, WM_VSCROLL, SB_LINEDOWN, 0) ;
                            break ;

                case  VK_LEFT :
                            SendMessage (hwnd, WM_HSCROLL, SB_PAGEUP, 0) ;
                            break ;
```

```
                case  VK_RIGHT :
                            SendMessage (hwnd, WM_HSCROLL, SB_PAGEDOWN, 0) ;
                            break ;
                }
                return 0 ;

    case  WM_MOUSEWHEEL:
                if (iDeltaPerLine == 0)
                                break ;

                iAccumDelta += (short) HIWORD (wParam) ;     // 120 or -120

                while (iAccumDelta >= iDeltaPerLine)
                {
                            SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0) ;
                            iAccumDelta -= iDeltaPerLine ;
                }

                while (iAccumDelta <= -iDeltaPerLine)
                {
                            SendMessage (hwnd, WM_VSCROLL, SB_LINEDOWN, 0) ;
                            iAccumDelta += iDeltaPerLine ;
                }

                return 0 ;

    case  WM_PAINT :
                hdc = BeginPaint (hwnd, &ps) ;

                            // Get vertical scroll bar position

                      si.cbSize         = sizeof (si) ;
                      si.fMask                = SIF_POS ;
                      GetScrollInfo (hwnd, SB_VERT, &si) ;
                      iVertPos                = si.nPos ;

                            // Get horizontal scroll bar position

                      GetScrollInfo (hwnd, SB_HORZ, &si) ;
                      iHorzPos = si.nPos ;

                            // Find painting limits

                      iPaintBeg = max (0, iVertPos + ps.rcPaint.top / cyChar) ;
                      iPaintEnd = min (NUMLINES - 1,
                iVertPos + ps.rcPaint.bottom / cyChar) ;

                      for (i = iPaintBeg ; i <= iPaintEnd ; i++)
```

```
                        {
                                    x = cxChar * (1 - iHorzPos) ;
                                    y = cyChar * (i - iVertPos) ;

                                    TextOut (   hdc, x, y,
                                sysmetrics[i].szLabel,
                                 lstrlen (sysmetrics[i].szLabel)) ;

                                    TextOut (   hdc, x + 22 * cxCaps, y,
                                 sysmetrics[i].szDesc,
                                 lstrlen (sysmetrics[i].szDesc)) ;

                                    SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;

                    TextOut  (  hdc, x + 22 * cxCaps + 40 * cxChar, y, szBuffer,
                wsprintf (szBuffer, TEXT ("%5d"),
                GetSystemMetrics (sysmetrics[i].iIndex))) ;

                                    SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
                    }

                    EndPaint (hwnd, &ps) ;
                    return 0 ;

      case  WM_DESTROY :
                    PostQuitMessage (0) ;
                    return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

转动滑轮会导致 Windows 在有输入焦点的视窗（不是滑鼠游标下面的视窗）产生 WM_MOUSEWHEEL 讯息。与平常一样，lParam 将获得滑鼠的位置，当然座标是相对於萤幕左上角的，而不是显示区域的。另外，wParam 的低字组包含一系列的旗标，用於表示滑鼠按键、Shift 与 Ctrl 键的状态。

新的资讯保存在 wParam 的高字组。其中有一个「delta」值，该值目前可以是 120 或-120，这取决於滑轮的向前转动（也就是说，向滑鼠的前面，即带有按钮与电缆的一端）还是向後转动。值 120 或-120 表示文件将分别向上或向下卷动三行。这里的构想是，以後版本的滑鼠滑轮能有比现在的滑鼠产生更精确的移动速度资讯，并且用 delta 值，例如 40 和-40，来产生 WM_MOUSEWHEEL 讯息。这些值能使文件只向上或向下卷动一行。

为使程式能在一般化环境执行，SYSMETS 将在 WM_CREATE 和 WM_SETTINGCHANGE 讯息处理时，以 SPI_GETWHEELSCROLLLINES 作为参数来呼叫 SystemParametersInfo。此值说明 WHEEL_DELTA 的 delta 值将滚动多少行，

WHEEL_DELTA 在 WINUSER.H 中定义。WHEEL_DELTA 等於 120，并且，在内定情况下 SystemParametersInfo 传回 3，因此与卷动一行相联系的 delta 值就是 40。SYSMETS 将此值保存在 iDeltaPerLine。

在 WM_MOUSEWHEEL 讯息处理期间，SYSMETS 将 delta 值给静态变数 iAccumDelta。然後，如果 iAccumDelta 大於或等於 iDeltaPerLine（或者是小於或等於-iDeltaPerLin），SYSMETS 用 SB_LINEUP 或 SB_LINEDOWN 值产生 WM_VSCROLL 讯息。对於每一个 WM_VSCROLL 讯息，iAccumDelta 由 iDeltaPerLine 增加(或减少)。此代码允许 delta 值大於、小於或等於滚动一行所需要的 delta 值。

下面还有

还有一个引人注目的滑鼠问题：建立自订滑鼠游标。我将在第十章，与其他 Windows 资源一起讨论此问题。