第十四章　点阵图和 Bitblt

点阵图是一个二维的位元阵列，它与图像的图素一一对应。当现实世界的图像被扫描成点阵图以後，图像被分割成网格，并以图素作为取样单位。在点阵图中的每个图素值指明了一个单位网格内图像的平均颜色。单色点阵图每个图素只需要一位元，灰色或彩色点阵图中每个图素需要多个位元。

点阵图代表了 Windows 程式内储存图像资讯的两种方法之一。储存图像资讯的另一种形式是 metafile，我将在第十八章讨论。Metafile 储存的就是对图像如何生成的描述，而不是将图像以数位化的图示代表。

以後我将更详细地讨论，Microsoft Windows 3.0 定义了一种称为装置无关点阵图（DIB：device-independent bitmap）。我将在下一章讨论 DIB。本章主要讨论 GDI 点阵图物件，这是一种在 Windows 中比 DIB 更早支援的点阵图形资料。如同本章大量的范例程式所说明的，这种比 DIB 点阵图更早被 Windows 支援的图形格式仍然有其利用价值。

点阵图入门

点阵图和 metafile 在电脑图形处理世界中都占有一席之地。点阵图经常用来表示来自真实世界的复杂图像，例如数位化的照片或者视讯图像。Metafile 更适合於描述由人或者机器产生的图像，比如建筑蓝图。点阵图和 metafile 都能存於记忆体或作为档案存於磁片上，并且都能通过剪贴簿在 Windows 应用程式之间传输。

点阵图和 metafile 的区别在於位元映射图像和向量图像之间的差别。位元映射图像用离散的图素来处理输出设备；而向量图像用笛卡尔座标系统来处理输出设备，其线条和填充物件能被个别拖移。现在大多数的图像输出设备是位元映射设备，这包括视讯显示、点阵印表机、雷射印表机和喷墨印表机。而笔式绘图机则是向量输出设备。

点阵图有两个主要的缺点。第一个问题是容易受装置依赖性的影响。最明显的就是对颜色的依赖性，在单色设备上显示彩色点阵图的效果总是不能令人满意的。另一个问题是点阵图经常暗示了特定的显示解析度和图像纵横比。尽管点阵图能被拉伸和缩小，但是这样的处理通常包括复制或删除图素的某些行和列，这样会破坏图像的大小。而 metafile 在放大缩小後仍然能保持图形样貌不受破坏。

点阵图的第二个缺点是需要很大的储存空间。例如，描述完整的 640×480

图素，16 色的视频图形阵列（VGA：Video Graphics Array）萤幕的一幅点阵图需要大於 150 KB 的空间；一幅 1024×768，并且每个图素为 24 位元颜色的图像则需要大於 2 MB 的空间。Metafile 需要通常比点阵图来得少的空间。点阵图的储存空间由图像的大小及其包含的颜色决定，而 metafile 的储存空间则由图像的复杂程度和它所包含的 GDI 指令数决定。

然而，点阵图优於 metafile 之处在於速度。将点阵图复制给视讯显示器通常比复制基本图形档案的速度要快。最近几年，压缩技术允许压缩点阵图的档案大小，以使它能有效地通过电话线传输并广泛地用於 Internet 的网页上。

点阵图的来源

点阵图可以手工建立，例如，使用 Windows 98 附带的「小画家」程式。一些人宁愿使用位元映射绘图软体也不使用向量绘图软体。他们假定：图形最後一定会复杂到不能用线条跟填充区域来表达。

点阵图图像也能由电脑程式计算生成。尽管大多数计算生成的图像能按向量图形 metafile 储存，但是高清晰度的画面或碎形图样通常还是需要点阵图。

现在，点阵图通常用於描述真实世界的图像，并且有许多硬体设备能让您把现实世界的图像输入到电脑。这类硬体通常使用 **电荷耦合装置**（CCD：charge-coupled device），这种装置接触到光就释放电荷。有时这些 CCD 单元能排列成一组，一个图素对应一个 CCD；为节约开支，只用一行 CCD 扫描图像。

在这些电脑 CCD 设备中，**扫描器** 是最古老的。它用一行 CCD 沿著纸上图像（例如照片）的表面扫描。CCD 根据光的强度产生电荷。类比数位转换器（ADC：Analog-to-digital converters）把电荷转换为数位讯号，然後排列成点阵图。

携带型摄像机也利用 CCD 单元组来捕捉影像。通常，这些影像是记录到录影带上。不过，这些视讯输出也能直接进入 **影像捕捉器**（frame grabber），该装置能把类比视讯信号转换为一组图素值。这些影像捕捉器与任何相容的视讯信号来源都能同时使用，例如 VCR、光碟、DVD 播放机或有线电视解码器。

最近，数位照相机的价位对於家庭使用者来说开始变得负担得起了。它看起来很像普通照相机。但是数位照相机不使用底片，而用一组 CCD 来拦截图像，并且在 ADC 内部把数位图像直接储存在照相机内的记忆体中。通常，数位照相机与电脑的介面要通过序列埠。

点阵图尺寸

点阵图呈矩形，并有空间尺寸，图像的高度和宽度都以图素为单位。例如，此网格可描述一个很小的点阵图：宽度为 9 图素，高度为 6 图素，或者更简单

地计为 9×6：

习惯上，点阵图的速记尺寸是先给出宽度。点阵图总数为 9×6 或者 54 图素。我将经常使用符号 cx 和 cy 来表示点阵图的宽度和高度。c 表示计数，因此 cx 和 cy 是沿著 x 轴（水平）和 y 轴（垂直）的图素数。

我们能根据 x 和 y 座标来描述点阵图上具体的图素。一般（并不都是这样），在网格内计算图素时，点阵图开始於图像的左上角。这样，在此点阵图右下角的图素座标就是(8, 5)。因为从 0 开始计数，所以此值比图像的宽度和高度小 1。

点阵图的空间尺寸通常也指定了解析度，但这是一个有争议的词。我们说我们的视讯显示有 640×480 的解析度，但是雷射印表机的解析度只有每英寸 300 点。我喜欢用後一种情况中解析度的意思作为每单位图素的数量。点阵图在这种意义上的解析度指的是点阵图在特定测量单位中的图素数。不管怎样，当我使用解析度这个词语时，其定义的内容应该是明确的。

点阵图是矩形的，但是电脑记忆体空间是线性的。通常（但并不都是这样）点阵图按列储存在记忆体中，且从顶列图素开始到底列结束。（DIB 是此规则的一个主要例外）。每一列，图素都从最左边的图素开始依次向右储存。这就好像储存几列文字中的各个字元。

颜色和点阵图

除空间尺寸以外，点阵图还有颜色尺寸。这里指的是每个图素所需要的位元数，有时也称为点阵图的 颜色深度 （color depth）、 位元数 （bit-count）或 位元/图素 （bpp：bits per pixel）数。点阵图中的每个图素都有相同数量的颜色位元。

每图素 1 位元的点阵图称为 二阶 （bilevel）、 二色 （bicolor）或者 单色 （monochrome）点阵图。每图素可以是 0 或 1，0 表示黑色，1 可以表示白色，但并不总是这样。对於其他颜色，一个图素就需要有多个位元。可能的颜色值等於 2 位元数值。用 2 位元可以得到 4 种颜色，用 4 位元可以得 16 种颜色，8 位元可得到 256 种颜色，16 位元可得到 65,536 种颜色，而 24 位元可得到 16,777,216 种颜色。

如何将颜色位元的组合与人们所熟悉的颜色相对应是目前处理点阵图时经常碰到（而且常常是灾难）的问题。

实际的设备

点阵图可按其颜色位元数来分类；在 Windows 的发展过程中，不同的点阵图颜色格式取决於常用视讯显示卡的功能。实际上，我们可把视讯显示记忆体看作是一幅巨大的点阵图——我们从显示器上就可以看见。

Windows 1.0 多数采用的显示卡是 IBM 的彩色图像适配器（CGA：Color Graphics Adapter）和单色图形卡（HGC：Hercules Graphics Card）。HGC 是单色设备，而 CGA 也只能在 Windows 以单色图形模式使用。单色点阵图现在还很常用（例如，滑鼠的游标一般为单色），而且单色点阵图除显示图像以外还有其他用途。

随著增强型图形显示卡(EGA:Enhanced Graphics Adapter)的出现,Windows 使用者开始接触 16 色的图形。每个图素需要 4 个颜色位元。（实际上，EGA 比这里所讲的更复杂，它还包括一个 64 种颜色的调色盘，应用程式可以从中选择任意的 16 种颜色，但 Windows 只按较简单的方法使用 EGA）。在 EGA 中使用的 16 种颜色是黑、白、两种灰色、高低亮度的红色、绿和蓝（三原色）、青色（蓝和绿组合的颜色）。现在认为这 16 种颜色是 Windows 的最低颜色标准。同样，其他 16 色点阵图也可以在 Windows 中显示。大多数的图示都是 16 色的点阵图。通常，简单的卡通图像也可以用这 16 种颜色制作。

在 16 色点阵图中的颜色编码有时称为 IRGB（高亮红绿蓝：Intensity-Red-Green-Blue），并且实际上是源自 IBM CGA 文字模式下最初使用的十六种颜色。每个图素所用的 4 个 IRGB 颜色位元都映射为表 14-1 所示的 Windows 十六进位 RGB 颜色。

表 14-1

IRGB	RGB 颜色	颜色名称
0000	00-00-00	黑
0001	00-00-80	暗蓝
0010	00-80-00	暗绿
0011	00-80-80	暗青
0100	80-00-00	暗红
0101	80-00-80	暗洋红
0110	80-80-00	暗黄
0111	C0-C0-C0	亮灰

1000	80-80-80	暗灰
1001	00-00-FF	蓝
1010	00-FF-00	绿
1011	00-FF-FF	青
1100	FF-00-00	红
1101	FF-00-FF	洋红
1110	FF-FF-00	黄
1111	FF-FF-FF	白

EGA 的记忆体组成了四个「颜色面」，也就是说，定义每个图素颜色的四位元在记忆体中是不连续的。然而，这样组织显示记忆体便於使所有的亮度位元都排列在一起、所有的红色位元都排在一起，等等。这样听起来就好像一种设备依赖特性，即 Windows 程式写作者不需要了解所有细节，但这时应或多或少地知道一些。不过，这些颜色面会出现在一些 API 呼叫中，例如 GetDeviceCaps 和 CreateBitmap。

Windows 98 和 Microsoft Windows NT 需要 VGA 或解析度更高的图形卡。这是目前公认的显示卡的最低标准。

1987 年，IBM 最早发表视讯图像阵列（Video Graphics Array：VGA）以及 PS/2 系列的个人电脑。它提供了许多不同的显示模式，但最好的图像模式（Windows 也使用其中之一）是水平显示 640 个图素，垂直显示 480 个图素，带有 16 种颜色。要显示 256 种颜色，最初的 VGA 必须切换到 320×240 的图形模式，这种图素数不适合 Windows 的正常工作。

一般人们已经忘记了最初 VGA 卡的颜色限制，因为其他硬体制造商很快就开发了「Super-VGA」（SVGA）显示卡，它包括更多的视讯记忆体，可显示 256 种颜色并有多於 640×480 的模式。这是现在的标准，而且也是一件好事，因为对於现实世界中的图像来说，16 种颜色过於简单，有些不适合。

显示 256 种颜色的显示卡模式采用每图素 8 位元。不过，这些 8 位元值都不必与实际的颜色相符。事实上，显示卡提供了「调色盘对照表（palette lookup table）」，该表允许软体指定这 8 位元的颜色值，以便与实际颜色相符合。在 Windows 中，应用程式不能直接存取调色盘对照表。实际上，Windows 储存了 256 种颜色中的 20 种，而应用程式可以通过「Windows 调色盘管理器」来自订其余的 236 种颜色。关於这些内容，我将在第十六章详细介绍。调色盘管理器允许应用程式在 256 色显示器上显示实际点阵图。Windows 所储存的 20 种颜色如表 14-2 所示。

表 14-2

IRGB	RGB 颜色	颜色名称
00000000	00-00-00	黑
00000001	80-00-00	暗红
00000010	00-80-00	暗绿
00000011	80-80-00	暗黄
00000100	00-00-80	暗蓝
00000101	80-00-80	暗洋红
00000110	00-80-80	暗青
00000111	C0-C0-C0	亮灰
00001000	C0-DC-C0	美元绿
00001001	A6-CA-F0	天蓝
11110110	FF-FB-F0	乳白
11110111	A0-A0-A4	中性灰
11111000	80-80-80	暗灰
11111001	FF-00-00	红
11111010	00-FF-00	绿
11111011	FF-FF-00	黄
11111100	00-00-FF	蓝
11111101	FF-00-FF	洋红
11111110	00-FF-FF	青
11111111	FF-FF-FF	白

最近几年，True-Color 显示卡很普遍，它们在每图素使用 16 位元或 24 位元。有时每图素虽然用了 16 位元，其中有 1 位元不用，而其他 15 位元主要近似於红、绿和蓝。这样红、绿和蓝每种都有 32 色阶，组合起来就可以达到 32,768 种颜色。更普遍的是，6 位元用於绿色（人类对此颜色最敏感），这样就可得到 65,536 种颜色。对於非技术性的 PC 使用者来说，他们并不喜欢看到诸如 32,768 或 65,536 之类的数字，因此通常将这种视讯显示卡称为 Hi-Color 显示卡，它能提供数以千计的颜色。

到了每个图素 24 位元时，我们总共有了 16,777,216 种颜色（或者 True Color、数百万的颜色），每个图素使用 3 位元组。这与今後的标准很相似，因为它大致代表了人类感官的极限而且也很方便。

在呼叫 GetDeviceCaps 时（参见第五章的 DEVCAPS 程式），您能利用 BITSPIXEL 和 PLANES 常数来获得显示卡的颜色单位，这些值显示如表 14-3 所示

表 14-3

BITSPIXEL	PLANES	颜色数
1	1	2
1	4	16
8	1	256
15 或 16	1	32,768 或 65 536
24 或 32	1	16 777 216

最近，您应该不会再碰到单色显示器了，但即便碰到了，您的应用程式也应该不会发生问题。

GDI 支援的点阵图

Windows 图形装置介面（GDI：Graphics Device Interface）从 1.0 版开始支援点阵图。不过，一直到 Windows 3.0 以前，Windows 下唯一支援 GDI 物件的只有点阵图，以点阵图代号来使用。这些 GDI 点阵图物件是单色的，或者与实际的图像输出设备（例如视讯显示器）有相同的颜色单位。例如，与 16 色 VGA 相容的点阵图有四个颜色面。问题是这些颜色点阵图不能储存，也不能用於颜色单位不同的图像输出设备（如每图素占 8 位元就可以产生 256 种颜色的设备）上。

从 Windows 3.0 开始，定义了一种新的点阵图格式，我们称之为装置无关点阵图（device-independent bitmap），或者 DIB。DIB 包括了自己的调色盘，其中显示了与 RGB 颜色相对应的图素位元。DIB 能显示在任何位元映射输出设备上。这里唯一的问题是 DIB 的颜色通常一定会转换成设备实际表现出来的颜色。

与 DIB 同时，Windows 3.0 还介绍了「Windows 调色盘管理器」，它让程式能够从显示的 256 种颜色中自订颜色。就像我们在第十六章所看到的那样，应用程式通常在显示 DIB 时使用「调色盘管理器」。

Microsoft 在 Windows 95（和 Windows NT 4.0）中扩展了 DIB 的定义，并且在 Windows 98（和 Windows NT 5.0）中再次扩展。这些扩展增加了所谓的「图像颜色管理器（ICM：Image Color Management），并允许 DIB 更精确地指定图像所需要的颜色。我将在第十五章简要讨论 ICM。

不论 DIB 多么重要，在处理点阵图时，早期的 GDI 点阵图物件依然扮演了重要的角色。掌握点阵图使用方式的最好方法是按各种用法在演进发展的时间顺序来学习，先从 GDI 点阵图物件和位元块传输的概念开始。

位元块传输

我前面提到过，您可以把整个视讯显示器看作是一幅大点阵图。您在萤幕上见到的图素由储存在视讯显示卡上记忆体中的位元来描述。任何视讯显示的矩形区域也都是一个点阵图，其大小是它所包含的行列数。

让我们从将图像从视讯显示的一个区域复制到另一个区域，开始我们在点阵图世界的旅行吧！这个是强大的 BitBlt 函式的工作。

Bitblt（读作「bit blit」）代表「位元块传输（bit-block transfer）」。BLT 起源於一条组合语言指令，该指令在 DEC PDP-10 上用来传输记忆体块。术语「bitblt」第一次用在图像上与 Xerox Palo Alto Research Center（PARC）设计的 SmallTalk 系统有关。在 SmallTalk 中，所有的图形输出操作都使用 bitblt。程式写作者有时将 blt 用作动词，例如：「Then I wrote some code to blt the happy face to the screen and play a wave file.」

BitBlt 函式移动的是图素，或者（更明确地）是一个位元映射图块。您将看到，术语「传输（transfer）」与 BitBlt 函式不尽相同。此函式实际上对图素执行了一次位元操作，而且可以产生一些有趣的结果。

简单的 BitBlt

程式 14-1 所示的 BITBLT 程式用 BitBlt 函式将程式系统的功能表图示（位於程式 Windows 的左上角）复制到它的显示区域。

程式 14-1　BITBLT

```
BITBLT.C
/*------------------------------------------------------------------------
      BITBLT.C --      BitBlt Demonstration
                                                (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 ("BitBlt") ;
      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_INFORMATION) ;
      wndclass.hCursor                   = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground       = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
      wndclass.lpszMenuName        = NULL ;
      wndclass.lpszClassName       = szAppName ;

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

      hwnd = CreateWindow (  szAppName, TEXT ("BitBlt 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 int             cxClient, cyClient, cxSource, cySource ;
      HDC              hdcClient, hdcWindow ;
      int              x, y ;
      PAINTSTRUCT            ps ;

      switch (message)
      {
      case WM_CREATE:
                        cxSource = GetSystemMetrics (SM_CXSIZEFRAME) +
                  GetSystemMetrics (SM_CXSIZE) ;
                        cySource = GetSystemMetrics (SM_CYSIZEFRAME) +
                  GetSystemMetrics (SM_CYCAPTION) ;
                        return 0 ;
```

```
     case  WM_SIZE:
                    cxClient = LOWORD (lParam) ;
                    cyClient = HIWORD (lParam) ;
                    return 0 ;

     case  WM_PAINT:
                    hdcClient = BeginPaint (hwnd, &ps) ;
                    hdcWindow = GetWindowDC (hwnd) ;

                    for (y = 0 ; y < cyClient ; y += cySource)
                    for (x = 0 ; x < cxClient ; x += cxSource)
                    {
                          BitBlt (hdcClient, x, y, cxSource, cySource,
                                                    hdcWindow, 0, 0, SRCCOPY) ;
                    }

                    ReleaseDC (hwnd, hdcWindow) ;
                    EndPaint (hwnd, &ps) ;
                    return 0 ;

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

但为什么只用了一个 BitBlt 呢？实际上，那个 BITBLT 用系统功能表图示的多个副本来填满显示区域（在此情况下是资讯方块中普遍使用的 IDI_INFORMATION 图示），如图 14-1 所示。

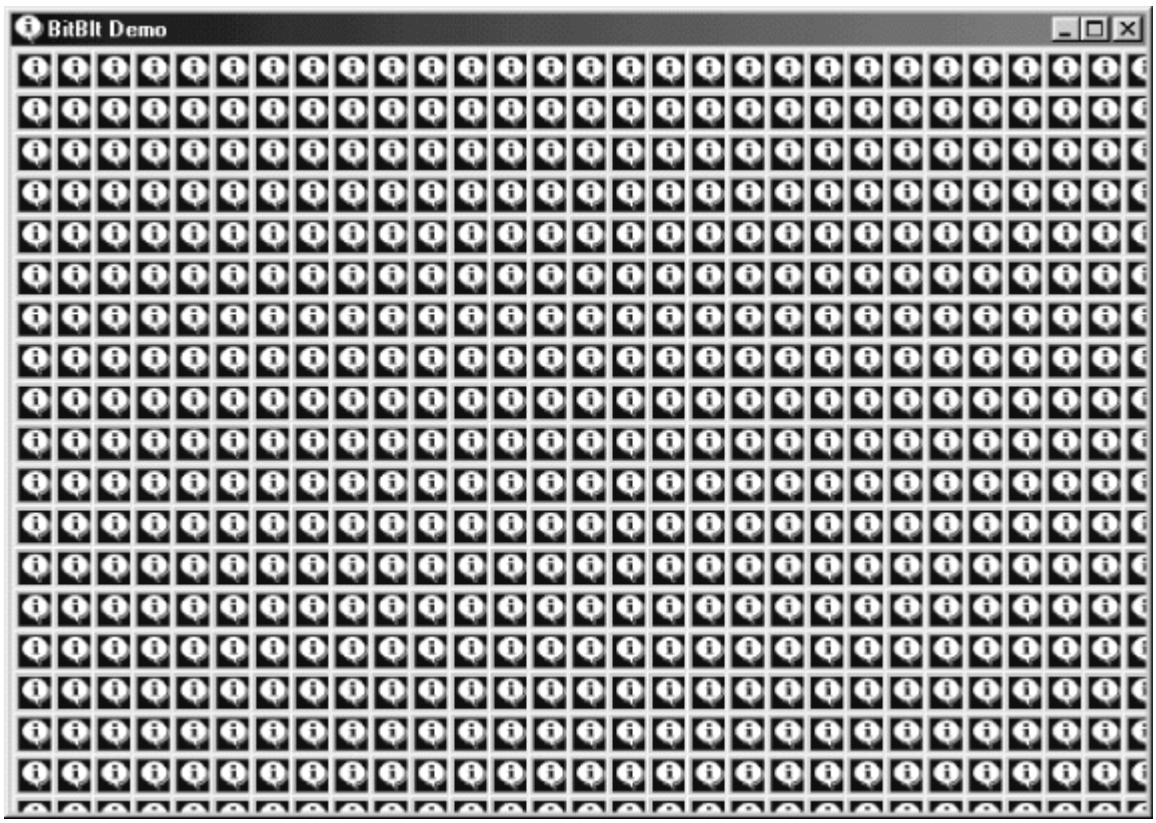


图 14-1 BITBLT 的萤幕显示

BitBlt 函式从称为「来源」的装置内容中将一个矩形区的图素传输到称为「目的(destination)」的另一个装置内容中相同大小的矩形区。此函式的语法如下：

```
BitBlt (hdcDst, xDst, yDst, cx, cy, hdcSrc, xSrc, ySrc, dwROP) ;
```

来源和目的装置内容可以相同。

在 BITBLT 程式中，目的装置内容是视窗的显示区域，装置内容代号从 BeginPaint 函式获得。来源装置内容是应用程式的整个视窗，此装置内容代号从 GetWindowDC 获得的。很明显地，这两个装置内容指的是同一个实际设备（视讯显示器）。不过，这两个装置内容的座标原点不同。

xSrc 和 ySrc 参数指明了来源图像左上角的座标位置。在 BITBLT 中，这两个参数设为 0，表示图像从来源装置内容（也就是整个视窗）的左上角开始，cx 和 cy 参数是图像的宽度和高度。BITBLT 根据从 GetSytemMetrics 函式获得的资讯来计算这些值。

xDst 和 yDst 参数表示了复制图像位置左上角的座标位置。在 BITBLT 中，这两个参数设定为不同的值以便多次复制图像。对於第一次 BitBlt 呼叫，这两个参数设制为 0，将图像复制到显示区域的左上角位置。

BitBlt 的最後一个参数是位元映射操作型态。我将简短地讨论一下这个值。

请注意，BitBlt 是从实际视讯显示记忆体传输图素，而不是从系统功能表图示的其他图像传输。如果您移动 BITBLT 视窗以使部分系统功能表图示移出萤

幕，然後调整 BITBLT 视窗的尺寸使其重画，这时您将发现 BITBLT 显示区域中显示的是功能表图示的一部分。BitBlt 函式不再存取整个图像。

在 BitBlt 函式中，来源和目的装置内容可以相同。您可以重新编写 BITBLT 以使 WM_PAINT 处理执行以下内容：

```
BitBlt (hdcClient, 0, 0, cxSource, cySource,
                  hdcWindow, 0, 0, SRCCOPY) ;
for (y = 0 ; y < cyClient ; y += cySource)
for (x = 0 ; x < cxClient ; x += cxSource)
{
      if (x > 0 || y > 0)
                        BitBlt (hdcClient, x, y, cxSource, cySource,
                  hdcClient, 0, 0, SRCCOPY) ;
}
```

这将与前面显示的 BITBLT 一样产生相同的效果，只是显示区域左上角比较模糊。

在 BitBlt 内的最大限制是两个装置内容必须是相容的。这意味著或者其中之一必须是单色的，或者两者的每个图素都相同的位元数。总而言之，您不能用此方法将萤幕上的某些图形复制到印表机。

拉伸点阵图

在 BitBlt 函式中，目的图像与来源图像的尺寸是相同的，因为函式只有两个参数来说明宽度和高度。如果您想在复制时拉伸或者压缩图像尺寸，可以使用 StretchBlt 函式。StretchBlt 函式的语法如下：

```
StretchBlt (     hdcDst, xDst, yDst, cxDst, cyDst,
                              hdcSrc, xSrc, ySrc, cxSrc, cySrc, dwROP) ;
```

此函式添加了两个参数。现在的函式就分别包含了目的和来源各自的宽度和高度。STRETCH 程式展示了 StretchBlt 函式，如程式 14-2 所示。

程式 14-2　STRETCH

```
STRETCH.C
/*------------------------------------------------------------------------
      STRETCH.C --      StretchBlt Demonstration
                                                (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 ("Stretch") ;
```

```
     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_INFORMATION) ;
     wndclass.hCursor                 = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground      = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName       = NULL ;
     wndclass.lpszClassName      = szAppName ;

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

    hwnd = CreateWindow (szAppName, TEXT ("StretchBlt 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 int             cxClient, cyClient, cxSource, cySource ;
     HDC                    hdcClient, hdcWindow ;
     PAINTSTRUCT            ps ;
```

```
     switch (message)
     {
     case  WM_CREATE:
                    cxSource = GetSystemMetrics (SM_CXSIZEFRAME) +
               GetSystemMetrics (SM_CXSIZE) ;

                    cySource = GetSystemMetrics (SM_CYSIZEFRAME) +
               GetSystemMetrics (SM_CYCAPTION) ;
                    return 0 ;

     case  WM_SIZE:
                    cxClient = LOWORD (lParam) ;
                    cyClient = HIWORD (lParam) ;
                    return 0 ;

     case  WM_PAINT:
                    hdcClient = BeginPaint (hwnd, &ps) ;
                    hdcWindow = GetWindowDC (hwnd) ;

                    StretchBlt (hdcClient, 0, 0, cxClient, cyClient,
               hdcWindow, 0, 0, cxSource, cySource, MERGECOPY) ;

                    ReleaseDC (hwnd, hdcWindow) ;
                    EndPaint (hwnd, &ps) ;
                    return 0 ;

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

此程式只有呼叫了 StretchBlt 函式一次，但是利用此函式以系统功能表图示填充了整个显示区域，如图 14-2 所示。

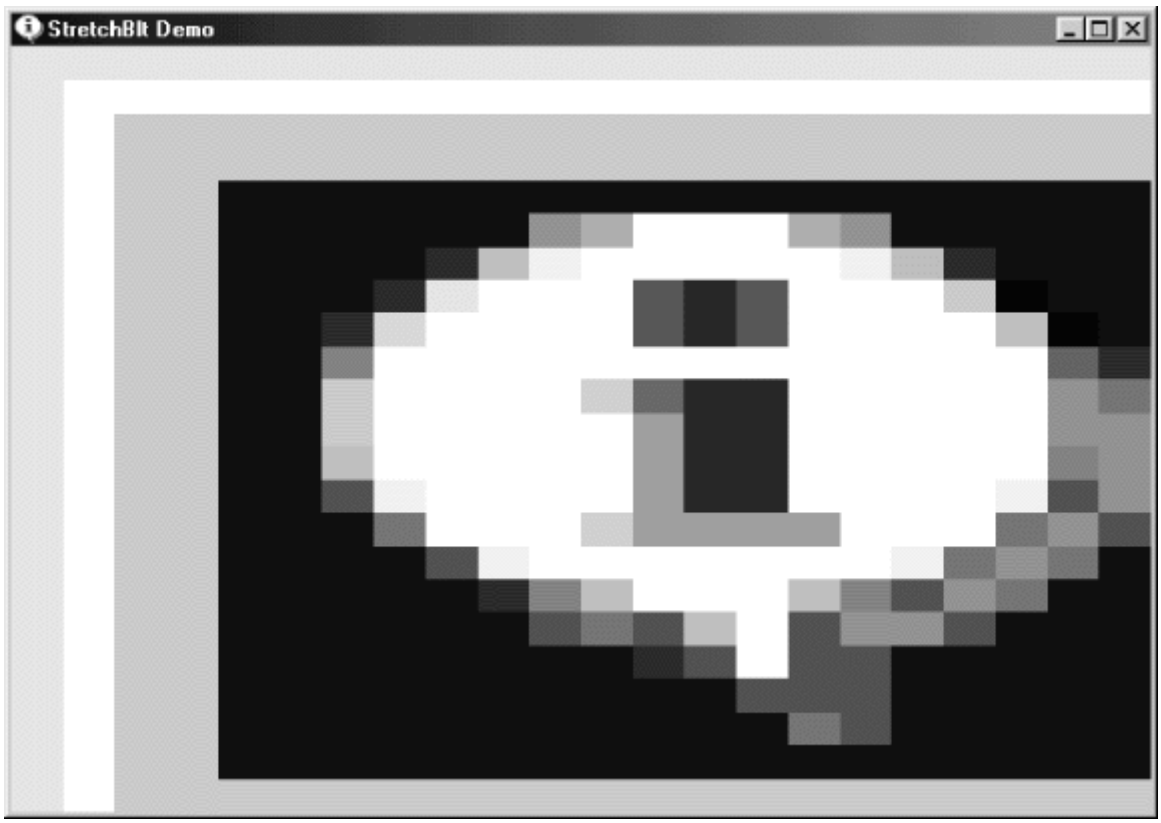


图 14-2 STRETCH 的萤幕显示

BitBlt 和 StretchBlt 函式中所有的座标与大小都是依据逻辑单位的。但是当您在 BitBlt 函式中定义了两个不同的装置内容，而这两个装置内容虽然参考同一个实际设备，却各自有著不同的映射模式，这时将发生什么结果呢？如果出现这种情况，呼叫 BitBlt 产生的结果就显得不明确了：cx 和 cy 参数都是逻辑单位，而它们同样应用於来源装置内容和目的装置内容中的矩形区。所有的座标和尺寸必须在实际的位元传输之前转换为装置座标。因为 cx 和 cy 值同时用於来源和目的装置内容，所以此值必须转换为装置内容自己的单位。

当来源和目的装置内容相同，或者两个装置内容都使用 MM_TEXT 图像模式时，装置单位下的矩形尺寸在两个装置内容中会是相同的，然後才由 Windows 进行图素对图素的转换。不过，如果装置单位下的矩形尺寸在两个装置内容中不同时，则 Windows 就把此工作转交给更通用的 StretchBlt 函式。

StretchBlt 也允许水平或垂直旋转图像。如果 cxSrc 和 cxDst 标记（转换成装置单位以後)不同,那么 StretchBlt 就建立一个镜像:左右旋转。在 STRETCH 程式中，通过将 xDst 参数改为 cxClient 并将 cxDst 参数改成-cxClient，您就可以做到这一点。如果 cySrc 和 cyDst 不同，则 StretchBlt 会上下旋转图像。要在 STRETCH 程式中测试这一点，可将 yDst 参数改为 cyClient 并将 cyDst 参数改成-cyClient。

StretchBlt 模式

使用 StretchBlt 会碰到一些与点阵图大小缩放相关的一些根本问题。在扩展一个点阵图时，StretchBlt 必须复制图素行或列。如果放大倍数不是原图的整数倍，那么此操作会造成产生的图像有些失真。

如果目的矩形比来源矩形小，那么 StretchBlt 在缩小图像时就必须把两行（或列）或者多行（或列）的图素合并到一行（或列）。完成此操作有四种方法，它根据装置内容伸展模式属性来选择其中一种方法。您可使用 SetStretchBltMode 函式来修改这个属性。

```
SetStretchBltMode (hdc, iMode) ;
```

iMode 可取下列值：

- BLACKONWHITE 或者 STRETCH_ANDSCANS（内定） 如果两个或多个图素得合并成一个图素，那么 StretchBlt 会对图素执行一个逻辑 AND 运算。这样的结果是只有全部的原始图素是白色时该图素才为白色，其实际意义是黑色图素控制了白色图素。这适用於白背景中主要是黑色的单色点阵图。
- WHITEONBLACK 或 STRETCH_ORSCANS 如果两个或多个图素得合并成一个图素，那么 StretchBlt 执行逻辑 OR 运算。这样的结果是只有全部的原-始图素都是黑色时才是黑色，也就是说由白色图素决定颜色。这适用於黑色背景中主要是白色的单色点阵图。
- COLORONCOLOR 或 STRETCH_DELETESCANS StretchBlt 简单地消除图素行或列，而没有任何逻辑组合。这是通常是处理彩色点阵图的最佳方法。
- HALFTONE 或 STRETCH_HALFTONE Windows 根据组合起来的来源颜色来计算目的的平均颜色。这将与半调调色盘联合使用，第十六章将展示这一程序。

Windows 还包括用於取得目前伸展模式的 GetStretchBltMode 函式。

位元映射操作

BITBLT 和 STRETCH 程式简单地将来源点阵图复制给了目的点阵图，在过程中也可能进行了缩放。这是把 SRCCOPY 作为 BitBlt 和 StretchBlt 函式最後一个参数的结果。SRCCOPY 只是您能在这些函式中使用的 256 个位元映射操作中的一个。让我们先在 STRETCH 程式中做一个别的实验，然後再系统地研究位元映射操作。

尽量用 NOTSRCCOPY 来代替 SRCCOPY。与它们名称一样，位元映射操作在复制点阵图时转换其颜色。在显示区域视窗，所有的颜色转换：黑色变成白色、

白色变成黑色，蓝色变成黄色。现在试一下 SRCINVERT，您将得到同样效果。如果试一下 BLACKNESS，正如其名称一样，整个显示区域都将变成黑色，而 WHITENESS 则使其变成白色。

现在试一试用下列三条叙述来代替 StretchBlt 呼叫：

```
SelectObject (hdcClient, CreateHatchBrush (HS_DIAGCROSS, RGB (0, 0, 0)));
StretchBlt (     hdcClient, 0, 0, cxClient, cyClient,
                           hdcWindow, 0, 0, cxSource, cySource, MERGECOPY) ;

DeleteObject (hdcClient, GetStockObject (WHITE_BRUSH)) ;
```

这次，您将在图像上看到一个菱形的画刷，这是什么？

我在前面说过，BitBlt 和 StretchBlt 函式不是简单的位元块传输。此函式实际在下面三种图像间执行位元操作。

Source 来源点阵图，拉伸或压缩（如果有必要）到目的矩形的尺寸。

Destination 在 BitBlt 或 StretchBlt 呼叫之前的目的矩形。

Pattern 在目的装置内容中选择的目前画刷，水平或垂直地复制到目的矩形范围内。

结果是复制到了目的矩形中。

位元映射操作与我们在第五章遇到的绘图模式在概念上相似。绘图模式采用图像物件的控制项方式，例如一条线就组合成一个目的。我们知道有 16 种绘图模式——也就是说，物件中的 0 和 1 画出时，唯一结果就是目的中 0 和 1 的组合。

使用 BitBlt 和 StretchBlt 的位元映射操作包含了三个物件的组合，这将产生 256 种位元映射操作。有 256 种方法来组合来源点阵图、目的点阵图和图案。有 15 种位元映射操作已经命名——其中一些名称其实还不能够清楚清楚说明其意义——它们定义在 WINGDI.H 里头，其余的都有数值，列在/Platform SDK/Graphics and Multimedia Services/GDI/Raster Operation Codes/Ternary Raster Operations 之中。

有名称的 15 种 ROP 代码见表 14-4。

表 14-4

图案（P）：1 1 1 1 0 0 0 0 来源（s）：1 1 0 0 1 1 0 0 目的（D）：1 0 1 0 1 0 1 0		**布林操作**	**ROP 代码**	**名称**
结果：	0 0 0 0 0 0 0 0	0	0x000042	BLACKNESS
	0 0 0 1 0 0 0 1	~(S|D)	0x1100A6	NOTSRCERASE
	0 0 1 1 0 0 1 1	~S	0x330008	NOTSRCCOPY

	0 1 0 0 0 1 0 0	S & ~D	0x440328	SRCERASE
	0 1 0 1 0 1 0 1	~D	0x550009	DSTINVERT
	0 1 0 1 1 0 1 0	P ^ D	0x5A0049	PATINVERT
	0 1 1 0 0 1 1 0	S ^ D	0x660046	SRCINVERT
	1 0 0 0 1 0 0 0	S & D	0x8800C6	SRCAND
	1 0 1 1 1 0 1 1	~S|D	0xBB0226	MERGEPAINT
	1 1 0 0 0 0 0 0	P & S	0xC000CA	MERGECOPY
	1 1 0 0 1 1 0 0	S	0xCC0020	SRCCOPY
	1 1 1 0 1 1 1 0	S|D	0xEE0086	SRCPAINT
	1 1 1 1 0 0 0 0	P	0xF00021	PATCOPY
	1 1 1 1 1 0 1 1	P| ~S|D	0xFB0A09	PATPAINT
	1 1 1 1 1 1 1 1	1	0xFF0062	WHITENESS

此表格对於理解和使用位元映射操作非常重要，因此我们应花点时间来研究。

在这个表格中，「ROP 代码」行的值将传递给 BitBlt 或 StretchBlt 的最後一个参数；在「名称」行中的值在 WINGDI.H 定义。ROP 代码的低字组协助装置驱动程式传输位元映射操作。高字组是 0 到 255 之间的数值。此数值与第 2 列的图案的位元相同，这是在图案、来源和显示在顶部的目的之间进行位元操作的结果。「布林运算」列按 C 语法显示图案、来源和目的的组合方式。

要开始了解此表，最简单的办法是假定您正处理一个单色系统（每图素 1 位元）其中 0 代表黑色，1 代表白色。BLACKNESS 操作的结果是不管是来源、目的和图案是什么，全部为零，因此目的将显示黑色。类似地，WHITENESS 总导致目的呈白色。

现在假定您使用位元映射操作 PATCOPY。这导致结果位元与图案位元相同，而忽略了来源和目的点阵图。换句话说，PATCOPY 简单地将目前图案复制给了目的矩形。

PATPAINT 位元映射操作包含一个更复杂的操作。其结果相同於在图案、目的和反转的来源之间进行位元或操作。当来源点阵图是黑色（0）时，其结果总是白色（1）；当来源是白色（1）时，只要图案或目的为白色，则结果就是白色。换句话说，只有来源为白色而图案和目的都是黑色时，结果才是黑色。

彩色显示时每个图素都使用了多个位元。BitBlt 和 StretchBlt 函式对每个颜色位元都分别提供了位元操作。例如，如果目的是红色而来源为蓝色，SRCPAINT 位元映射操作把目的变成洋红色。注意，操作实际是按显示卡内储存

的位元执行的。这些位元所对应的颜色取决於显示卡的调色盘的设定。Windows 完成了此操作，以便位元映射操作能达到您预计的结果。不过，如果您修改了调色盘（我将在第十六章讨论），位元映射操作将产生无法预料的结果。

如要得到位元映射操作较好的应用程式，请参见本章後面的「非矩形点阵图图像」一节。

图案 Blt

除了 BitBlt 和 StretchBlt 以外，Windows 还包括一个称为 PatBlt（「pattern block transfer：图案块传输」）的函式。这是三个「blt」函式中最简单的。与 BitBlt 和 StretchBlt 不同,它只使用一个目的装置内容。PatBlt 语法是：

```
PatBlt (hdc, x, y, cx, cy, dwROP) ;
```

x、y、cx 和 cy 参数位於逻辑单位。逻辑点（x,y）指定了矩形的左上角。矩形宽为 cx 单位，高为 cy 单位。这是 PatBlt 修改的矩形区域。PatBlt 在画刷与目的装置内容上执行的逻辑操作由 dwROP 参数决定，此参数是 ROP 代码的子集——也就是说，您可以只使用那些不包括来源目的装置内容的 ROP 代码。下表列出了 PatBlt 支援的 16 个位元映射操作：

表 14-5

图案（P）：1 1 0 0 目的（D）：1 0 1 0		布林操作	ROP 代码	名称
结果：	0 0 0 0	0	0x000042	BLACKNESS
	0 0 0 1	~(P | D)	0x0500A9	
	0 0 1 0	~P & D	0x0A0329	
	0 0 1 1	~P	0x0F0001	
	0 1 0 0	P & ~D	0x500325	
	0 1 0 1	~D	0x550009	DSTINVERT
	0 1 1 0	P ^ D	0x5A0049	PATINVERT
	0 1 1 1	~(P & D)	0x5F00E9	
	1 0 0 0	P & D	0xA000C9	
	1 0 0 1	~(P ^ D)	0xA50065	
	1 0 1 0	D	0xAA0029	
	1 0 1 1	~P | D	0xAF0229	

	1 1 0 0	P	0xF00021	PATCOPY
	1 1 0 1	P | ~D	0xF50225	
	1 1 1 0	P | D	0xFA0089	
	1 1 1 1	1	0xFF0062	WHITENESS

下面列出了 PatBlt 一些更常见用途。如果想画一个黑色矩形，您可呼叫

```
PatBlt (hdc, x, y, cx, cy, BLACKNESS) ;
```

要画一个白色矩形，请用

```
PatBlt (hdc, x, y, cx, cy, WHITENESS) ;
```

函式

```
PatBlt (hdc, x, y, cx, cy, DSTINVERT) ;
```

用於改变矩形的颜色。如果目前装置内容中选择了 WHITE_BRUSH，那么函式

```
PatBlt (hdc, x, y, cx, cy, PATINVERT) ;
```

也改变矩形。

您可以再次呼叫 FillRect 函式来用画笔充满一个矩形区域：

```
FillRect (hdc, &rect, hBrush) ;
```

FillRect 函式相同於下列代码：

```
hBrush = SelectObject (hdc, hBrush) ;
PatBlt (hdc,     rect.left, rect.top,
                            rect.right - rect.left,
                            rect.bottom - rect.top, PATCOPY) ;
SelectObject (hdc, hBrush) ;
```

实际上，此程式码是 Windows 用於执行 FillRect 函式的动作。如果您呼叫

```
InvertRect (hdc, &rect) ;
```

Windows 将其转换成函式：

```
PatBlt (hdc,     rect.left, rect.top,
                            rect.right - rect.left,
                            rect.bottom - rect.top, DSTINVERT) ;
```

在介绍 PatBlt 函式的语法时，我说过点（x,y）指出了矩形的左上角，而且此矩形宽度为 cx 单位，高度为 cy 单位。此叙述并不完全正确。BitBlt、PatBlt 和 StretchBlt 是最合适的 GDI 画图函式，它们根据从一个角测得的逻辑宽度和高度来指定逻辑直角座标。矩形边框用到的其他所有 GDI 画图函式都要求根据左上角和右下角的座标来指定座标。对於 MM_TEXT 映射模式，上面讲述的 PatBlt 参数就是正确的。然而对於公制的映射模式来说，就不正确。如果您使用一的 cx 和 cy 值，那么点（x,y）将是矩形的左下角。如果希望点（x,y）是矩形的左上角，那么 cy 参数必须设为矩形的负高度。

如果想更精确，用 PatBlt 修改颜色的矩形将通过 cx 的绝对值获得逻辑宽度，通过 cy 的绝对值获得逻辑高度。这两个参数可以是负值。由逻辑点（x, y）

和（x + cx, y + cy）给定的两个角定义了矩形。矩形的左上角通常属於 PatBlt 修改的区域。右上角则超出了矩形的范围。根据映射模式和 cx、cy 参数的符号，矩形左上角的点应为（x, y）、（x, y + cy）、（x + cx, y）或者（x + cx, y + cy）。

如果给 MM_LOENGLISH 设定了映射模式，并且您想在显示区域左上角的一小块正方形上使用 PatBlt，您可以使用

```
PatBlt (hdc, 0, 0, 100, -100, dwROP) ;
```

或

```
PatBlt (hdc, 0, -100, 100, 100, dwROP) ;
```

或

```
PatBlt (hdc, 100, 0, -100, -100, dwROP) ;
```

或

```
PatBlt (hdc, 100, -100, -100, 100, dwROP) ;
```

给 PatBlt 设定正确参数最容易的方法是将 x 和 y 设为矩形左上角。如果映射模式定义 y 座标随著向上卷动显示而增加，那么请使用负的 cy 参数。如果映射模式定义 x 座标向左增加（很少有人用），则需要使用负的 cx 参数。

GDI 点阵图物件

我在本章前面已提到过 Windows 从 1.0 开始就支援 GDI 点阵图物件。因为在 Windows 3.0 发表了装置无关点阵图，GDI 点阵图物件有时也称为装置相关点阵图，或者 DDB。我尽量不全部引用 device-dependent bitmap 的全文，因为它看上去与 device-independent bitmap 类似。缩写 DDB 会好一些，因为我们很容易把它与 DIB 区别开来。

对程式写作者来说，现存的两种不同型态的点阵图从 Windows 3.0 开始就更为混乱。许多有经验的 Windows 程式写作者都不能准确地理解 DIB 和 DDB 之间的关系。（恐怕本书的 Windows 3.0 版本不能澄清这个问题）。诚然，DIB 和 DDB 在许多方面是相关的：DIB 与 DDB 能相互转换（尽管转换程序中会丢失一些资讯）。然而 DIB 和 DDB 是不可以相互替换的，并且不能简单地选择一种方法来表示同一个可视资料。

如果我们能假设说 DIB 一定会替代 DDB，那以後就会很方便了。但现实并不是如此，DDB 还在 Windows 中扮演著很重要角色，尤其是您在乎程式执行表现好坏时。

建立 DDB

DDB 是 Windows 图形装置介面的图形物件之一（其中还包括绘图笔、画刷、

字体、metafile 和调色盘）。这些图形物件储存在 GDI 模组内部，由应用程式软体以代号数字的方式引用。您可以将 DDB 代号储存在一个 HBITMAP（「handle to a bitmap：点阵图代号」）型态的变数中，例如：

```
HBITMAP hBitmap ;
```

然後通过呼叫 DDB 建立的一个函式来获得代号，例如：CreateBitmap。这些函式配置并初始化 GDI 记忆体中的一些记忆体来储存关於点阵图的资讯，以及实际点阵图位元的资讯。应用程式不能直接存取这段记忆体。点阵图与装置内容无关。当程式使用完点阵图以後，就要清除这段记忆体：

```
DeleteObject (hBitmap) ;
```

如果程式执行时您使用了 DDB，那么程式终止时，您可以完成上面的操作。

CreateBitmap 函式用法如下：

```
hBitmap = CreateBitmap (cx, cy, cPlanes, cBitsPixel, bits) ;
```

前两个参数是点阵图的宽度和高度（以图素为单位），第三个参数是颜色面的数目，第四个参数是每图素的位元数，第五个参数是指向一个以特定颜色格式存放的位元阵列的指标，阵列内存放有用来初始化该 DDB 的图像。如果您不想用一张现有的图像来初始化 DDB，可以将最後一个参数设为 NULL。以後您还是可以设定该 DDB 内图素的内容。

使用此函式时，Windows 也允许建立您喜欢的特定型态 GDI 点阵图物件。例如，假设您希望点阵图宽 7 个图素、高 9 个图素、5 个?色位元面，并且每个图素占 3 位元，您只需要执行下面的操作：

```
hBitmap = CreateBitmap (7, 9, 5, 3, NULL) ;
```

这时 Windows 会好好给您一个有效的点阵图代号。

在此函式呼叫期间，Windows 将储存您传递给函式的资讯，并为图素位元配置记忆体。粗略的计算是此点阵图需要 7×9×5×3，即 945 位元，这需要比 118 个位元组还多几个位元。

然而，Windows 为点阵图配置好记忆体以後，每行图素都占用许多连贯的位元组，这样

```
iWidthBytes = 2 * ((cx * cBitsPixel + 15) / 16) ;
```

或者 C 程式写作者更倾向於写成：

```
iWidthBytes = (cx * cBitsPixel + 15) & ~15) >> 3 ;
```

因此，为 DDB 配置的记忆体就是：

```
iBitmapBytes = cy * cPlanes * iWidthBytes ;
```

本例中，iWidthBytes 占 4 位元组，iBitmapBytes 占 180 位元组。

现在，知道一张点阵图有 5 个颜色位元面，每图素占 3 个颜色位有什么意义吗？真是见鬼了，这甚至不能把它称作一个习题作业。虽然您让 GDI 内部配置了些记忆体，并且让这些记忆体有一定结构的内容，但是您这张点阵图完全

作不出任何有用的事情来。

实际上，您将用两种型态的参数来呼叫 CreateBitmap。

cPlanes 和 cBitsPixel 都等於 1（表示单色点阵图）；或者

cPlanes 和 cBitsPixel 都等於某个特定装置内容的值，您可以使用 PLANES 和 BITSPIXEL 索引来从 GetDeviceCaps 函式获得。

更现实的情况下，您只会在第一种情况下呼叫 CreateBitmap。对於第二种情况，您可以用 CreateCompatibleBitmap 来简化问题：

```
hBitmap = CreateCompatibleBitmap (hdc, cx, cy) ;
```

此函式建立了一个与设备相容的点阵图，此设备的装置内容代号由第一个参数给出。CreateCompatibleBitmap 用装置内容代号来获得 GetDeviceCaps 资讯，然後将此资讯传递给 CreateBitmap。除了与实际的装置内容有相同的记忆体组织之外，DDB 与装置内容没有其他联系。

CreateDiscardableBitmap 函式与 CreateCompatibleBitmap 的参数相同，并且功能上相同。在早期的 Windows 版本中，CreateDiscardableBitmap 建立的点阵图可以在记忆体减少时由 Windows 将其从记忆体中清除，然後程式再重建点阵图资料。

第三个点阵图建立函式是 CreateBitmapIndirect：

```
hBitmap CreateBitmapIndirect (&bitmap) ;
```

其中 bitmap 是 BITMAP 型态的结构。BITMAP 结构定义如下：

```
typedef struct _tagBITMAP
{
      LONG              bmType ;              // set to 0
      LONG              bmWidth ;             // width in pixels
      LONG              bmHeight ;            // height in pixels
      LONG              bmWidthBytes ;        // width of row in bytes
      WORD              bmPlanes ;            // number of color planes
      WORD              bmBitsPixel ;         // number of bits per pixel
      LPVOID      bmBits ;                    // pointer to pixel bits
}
BITMAP, * PBITMAP ;
```

在呼叫 CreateBitmapIndirect 函式时，您不需要设定 bmWidthBytes 栏位。Windows 将为您计算，您也可以将 bmBits 栏位设定为 NULL，或者设定为初始化点阵图时用的图素位元位址。

GetObject 函式内也使用 BITMAP 结构，首先定义一个 BITMAP 型态的结构。

```
BITMAP bitmap ;
```

并呼叫函式如下：

```
GetObject (hBitmap, sizeof (BITMAP), &bitmap) ;
```

Windows 将用点阵图资讯填充 BITMAP 结构的栏位，不过，bmBits 栏位等於

NULL。

您最後应呼叫 DeleteObject 来清除程式内建立的所有点阵图。

点阵图位元

用 CreateBitmap 或 CreateBitmapIndirect 来建立设备相关 GDI 点阵图物件时，您可以给点阵图图素位元指定一个指标。或者您也可以让点阵图维持未初始化的状态。在建立点阵图以後，Windows 还提供两个函式来获得并设定图素位元。

要设定图素位元，请呼叫：

```
SetBitmapBits (hBitmap, cBytes, &bits) ;
```

GetBitmapBits 函式有相同的语法：

```
GetBitmapBits (hBitmap, cBytes, &bits) ;
```

在这两个函式中，cBytes 指明要复制的位元组数，bits 是最少 cBytes 大小的缓冲区。

DDB 中的图素位元从顶列开始排列。我在前面说过，每列的位元组数都是偶数。除此之外，没什么好说明的了。如果点阵图是单色的，也就是说它有 1 个位元面并且每个图素占 1 位元，则每个图素不是 1 就是 0。每列最左边的图素是本列第一个位元组最高位元的位元。我们在本章的後面讲完如何显示单色 DDB 之後，将做一个单色的 DDB。

对於非单色点阵图，应避免出现您需要知道图素位元含义的状况。例如，假定在 8 位颜色的 VGA 上执行 Windows，您可以呼叫 CreateCompatibleBitmap。通过 GetDeviceCaps，您能够确定您正处理一个有 1 个颜色位元面和每图素 8 位元的设备。一个位元组储存一个图素。但是图素值 0x37 是什么意思呢？很明显是某种颜色，但到底是什么颜色呢？

图素实际上并不涉及任何固定的颜色，它只是一个值。DDB 没有颜色表。问题的关键在於：当 DDB 显示在萤幕上时，图素的颜色是什么。它肯定是某种颜色，但具体是什么颜色呢？显示的图素将与在显示卡上的调色盘查看表里的 0x37 索引值代表的 RGB 颜色有关。这就是您现在碰到的装置依赖性。

不过，不要只因为我们不知道图素值的含义，就假定非单色 DDB 没用。我们将简要看一下它们的用途。下一章，我们将看到 SetBitmapBits 和 GetBitmapBits 函式是如何被更有用的 SetDIBits 和 GetDIBits 函式所取代的。

因此，基本的规则是这样的：不要用 CreateBitmap、CreateBitmapIndirect 或 SetBitmapBits 来设定彩色 DDB 的位元，您只能安全地使用这些函式来设定单色 DDB 的位元。（如果您在呼叫 GetBitmapBits 期间，从其他相同格式的 DDB 中获得位元，那么这些规则例外。）

在继续之前，让我再讨论一下 SetBitmapDimensionEx 和 GetBitmapDimensionEx 函式。这些函式让您设定（和获得）点阵图的测量尺寸（以 0.1 毫米为单位）。这些资讯与点阵图解析度一起储存在 GDI 中，但不用於任何操作。它只是您与 DDB 联系的一个测量尺寸标识。

记忆体装置内容

我们必须解决的下一个概念是记忆体装置内容。您需要用记忆体装置内容来处理 GDI 点阵图物件。

通常，装置内容指的是特殊的图形输出设备（例如视讯显示器或者印表机）及其装置驱动程式。记忆体装置内容只位於记忆体中，它不是真正的图形输出设备，但可以说与指定的真正设备「相容」。

要建立一个记忆体装置内容，您必须首先有实际设备的装置内容代号。如果是 hdc，那么您可以像下面那样建立记忆体装置内容：

```
hdcMem = CreateCompatibleDC (hdc) ;
```

通常，函式的呼叫比这更简单。如果您将参数设为 NULL，那么 Windows 将建立一个与视讯显示器相相容的记忆体装置内容。应用程式建立的任何记忆体装置内容最终都通过呼叫 DeleteDC 来清除。

记忆体装置内容有一个与实际位元映射设备相同的显示平面。不过，最初此显示平面非常小——单色、1 图素宽、1 图素高。显示平面就是单独 1 位元。

当然，用 1 位元的显示平面，您不能做更多的工作，因此下一步就是扩大显示平面。您可以通过将一个 GDI 点阵图物件选进记忆体装置内容来完成这项工作，例如：

```
SelectObject (hdcMem, hBitmap) ;
```

此函式与您将画笔、画刷、字体、区域和调色盘选进装置内容的函式相同。然而，记忆体装置内容是您可以选进点阵图的唯一一种装置内容型态。（如果需要，您也可以将其他 GDI 物件选进记忆体装置内容。）

只有选进记忆体装置内容的点阵图是单色的，或者与记忆体装置内容相容设备有相同的色彩组织时，SelectObject 才会起作用。这也是建立特殊的 DDB（例如有 5 个位元面，且每图素 3 位元）没有用的原因。

现在情况是这样：SelectObject 呼叫以後，DDB 就是记忆体装置内容的显示平面。处理实际装置内容的每项操作，您几乎都可以用於记忆体装置内容。例如，如果用 GDI 画图函式在记忆体装置内容中画图，那么图像将画在点阵图上。这是非常有用的。还可以将记忆体装置内容作为来源，把视讯装置内容作为目的来呼叫 BitBlt。这就是在显示器上绘制点阵图的方法。如果把视讯装置内容作为来源，把记忆体装置内容作为目的，那么呼叫 BitBlt 可将萤幕上的一

些内容复制给点阵图。我们将看到这些都是可能的。

载入点阵图资源

除了各种各样的点阵图建立函式以外，获得 GDI 点阵图物件代号的另一个方法就是呼叫 LoadBitmap 函式。使用此函式，您不必担心点阵图格式。在程式中，您只需简单地按资源来建立点阵图，这与建立图示或者滑鼠游标的方法类似。LoadBitmap 函式的语法与 LoadIcon 和 LoadCursor 相同：

```
hBitmap = LoadBitmap (hInstance, szBitmapName) ;
```

如果想载入系统点阵图，那么将第一个参数设为 NULL。这些不同的点阵图是 Windows 视觉介面（例如关闭方块和勾选标记）的一小部分，它们的识别字以字母 OBM 开始。如果点阵图与整数识别字而不是与名称有联系，那么第二个参数就可以使用 MAKEINTRESOURCE 巨集。由 LoadBitmap 载入的所有点阵图最终应用 DeleteObject 清除。

如果点阵图资源是单色的，那么从 LoadBitmap 传回的代号将指向一个单色的点阵图物件。如果点阵图资源不是单色，那么从 LoadBitmap 传回的代号将指向一个 GDI 点阵图物件，该物件与执行程式的视讯显示器有相同的色彩组织。因此，点阵图始终与视讯显示器相容，并且总是选进与视讯显示器相容的记忆体装置内容中。采用 LoadBitmap 呼叫後，就不用担心任何色彩转换的问题了。在下一章中，我们就知道 LoadBitmap 的具体运作方式了。

程式 14-3 所示的 BRICKS1 程式示范了载入一小张单色点阵图资源的方法。此点阵图本身不像砖块，但当它水平和垂直重复时，就与砖墙相似了。

程式 14-3 BRICKS1

```
BRICKS1.C
/*------------------------------------------------------------------------
      BRICKS1.C -- LoadBitmap Demonstration
                                                (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 ("Bricks1") ;
      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 ("This program requires Windows NT!"),
                                          szAppName, MB_ICONERROR) ;
               return 0 ;
    }

     hwnd = CreateWindow (  szAppName, TEXT ("LoadBitmap 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 HBITMAP hBitmap ;
     static int                  cxClient, cyClient, cxSource, cySource ;
     BITMAP                      bitmap ;
     HDC                         hdc, hdcMem ;
     HINSTANCE                   hInstance ;
     int                         x, y ;
     PAINTSTRUCT                 ps ;

     switch (message)
     {
```

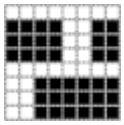
```
     case  WM_CREATE:
                        hInstance = ((LPCREATESTRUCT) lParam)->hInstance ;

                        hBitmap = LoadBitmap (hInstance, TEXT ("Bricks")) ;

                        GetObject (hBitmap, sizeof (BITMAP), &bitmap) ;

                        cxSource = bitmap.bmWidth ;
                        cySource = bitmap.bmHeight ;

                        return 0 ;

     case  WM_SIZE:
                        cxClient = LOWORD (lParam) ;
                        cyClient = HIWORD (lParam) ;
                        return 0 ;

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

                        hdcMem = CreateCompatibleDC (hdc) ;
                        SelectObject (hdcMem, hBitmap) ;

                        for (y = 0 ; y < cyClient ; y += cySource)
                        for (x = 0 ; x < cxClient ; x += cxSource)
                        {
                        BitBlt (hdc, x, y, cxSource, cySource, hdcMem, 0, 0,
SRCCOPY) ;
        }

                        DeleteDC (hdcMem) ;
                        EndPaint (hwnd, &ps) ;
                        return 0 ;

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

BRICKS1.RC （摘录）

```
//Microsoft Developer Studio generated resource script.
#include "resource.h"
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
/
```

```
// Bitmap
BRICKS                                  BITMAP  DISCARDABLE     "Bricks.bmp"
```

BRICKS.BMP



在 Visual C++ Developer Studio 中建立点阵图时，应指明点阵图的高度和宽度都是 8 个图素，是单色，名称是「Bricks」。BRICKS1 程式在 WM_CREATE 讯息处理期间载入了点阵图并用 GetObject 来确定点阵图的图素尺寸（以便当点阵图不是 8 图素见方时程式仍能继续工作）。以後，BRICKS1 将在 WM_DESTROY 讯息中删除此点阵图。

在 WM_PAINT 讯息处理期间，BRICKS1 建立了一个与显示器相容的记忆体装置内容，并且选进了点阵图。然後是从记忆体装置内容到显示区域装置内容一系列的 BitBlt 函式呼叫，再删除记忆体装置内容。图 14-3 显示了程式的执行结果。

顺便说一下，Developer Studio 建立的 BRICKS.BMP 档案是一个装置无关点阵图。您可能想在 Developer Studio 内建立一个彩色的 BRICKS.BMP 档案（您可自己选定颜色），并且保证一切工作正常。

我们看到 DIB 能转换成与视讯显示器相容的 GDI 点阵图物件。我们将在下一章看到这是如何操作的。

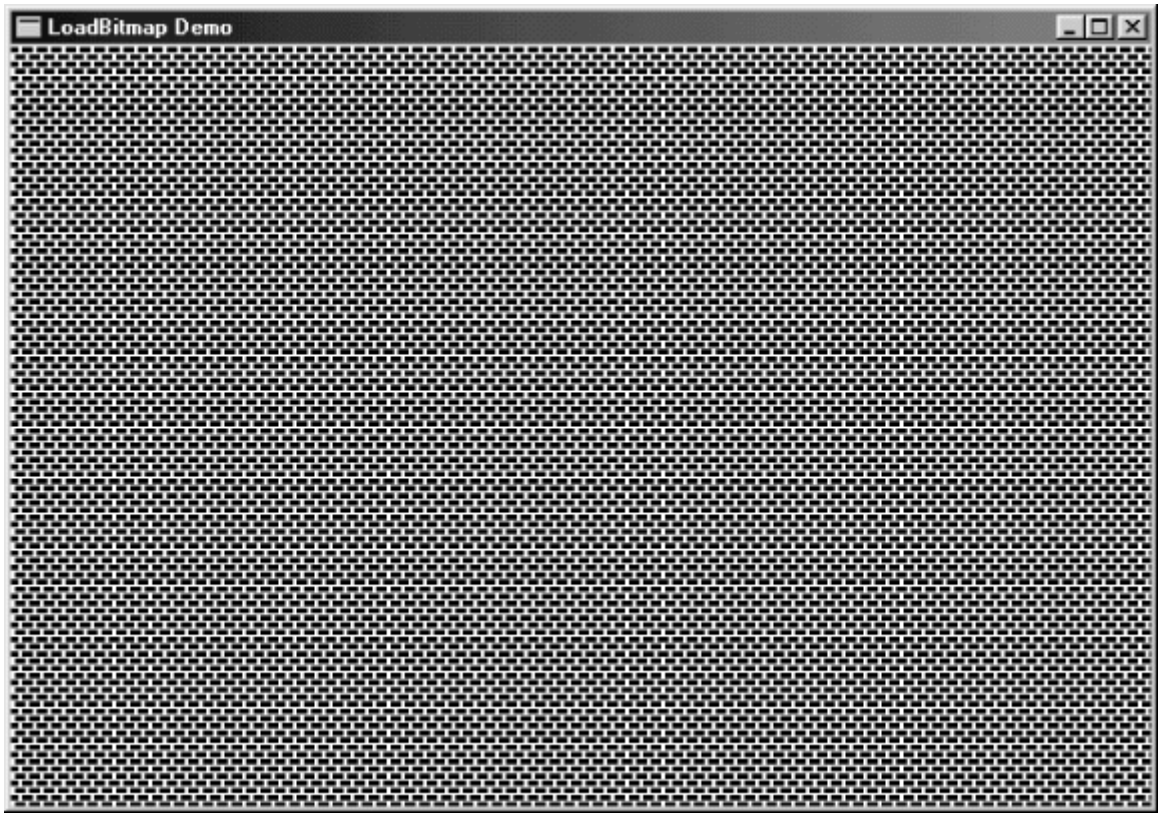
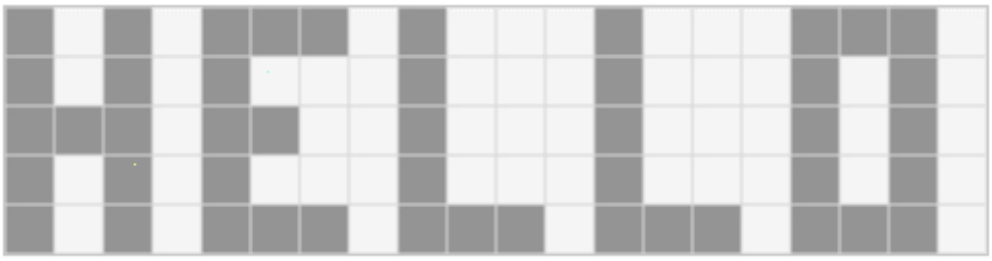


图 14-3 BRICKS1 的萤幕显示

单色点阵图格式

如果您在处理小块单色图像，那么您不必把它们当成资源来建立。与彩色点阵图物件不同，单色位元的格式相对简单一些，而且几乎能全部从您要建立的图像中分离出来。例如，假定您要建立下图所示的点阵图：



您能写下一系列的位元（0 代表黑色，1 代表白色），这些位元直接对应於网格。从左到右读这些位元，您能给每 8 位元组配置一个十六进位元的位元组值。如果点阵图的宽度不是 16 的倍数，在位元组的右边用零填充，以得到偶数个位元组：

```
0 1 0 1 0 0 0 1 0 1 1 1 0 1 1 1 0 0 0 1 = 51 77 10 00
0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 = 57 77 50 00
0 0 0 1 0 0 1 1 0 1 1 1 0 1 1 1 0 1 0 1 = 13 77 50 00
0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 = 57 77 50 00
0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 = 51 11 10 00
```

图素宽为 20，扫描线高为 5，位元组宽为 4。您可以用下面的叙述来设定此点阵图的 BITMAP 结构：

```
static BITMAP bitmap               = { 0, 20, 5, 4, 1, 1 } ;
```

并且可以将位元储存在 BYTE 阵列中：

```
static BYTE  bits []    = {   0x51, 0x77, 0x10, 0x00,
                              0x57, 0x77, 0x50, 0x00,
                                    0x13, 0x77, 0x50, 0x00,
                            0x57, 0x77, 0x50, 0x00,
                             0x51, 0x11, 0x10, 0x00 } ;
```

用 CreateBitmapIndirect 来建立点阵图需要下面两条叙述：

```
bitmap.bmBits = (PSTR) bits ;
hBitmap = CreateBitmapIndirect (&bitmap) ;
```

另一种方法是：

```
hBitmap = CreateBitmapIndirect (&bitmap) ;
SetBitmapBits (hBitmap, sizeof bits, bits) ;
```

您也可以用一道叙述来建立点阵图：

```
hBitmap = CreateBitmap (20, 5, 1, 1, bits) ;
```

在程式 14-4 显示的 BRICKS2 程式利用此技术直接建立了砖块点阵图，而没有使用资源。

程式 14-4　BRICKS2

```
BRICKS2.C
```

```
/*------------------------------------------------------------------------
      BRICKS2.C --     CreateBitmap Demonstration
                                          (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 ("Bricks2") ;
      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 ("This program requires Windows
NT!"),
                           szAppName, MB_ICONERROR) ;
                return 0 ;
      }

            hwnd = CreateWindow (szAppName, TEXT ("CreateBitmap 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 BITMA           Pbitmap = {      0, 8, 8, 2, 1, 1 } ;
     static BYTE            bits [8][2]={    0xFF, 0, 0x0C, 0, 0x0C, 0, 0x0C,
0,
                                        0xFF, 0, 0xC0, 0, 0xC0, 0, 0xC0, 0 } ;
     static HBITMAP hBitmap ;
     static int                  cxClient, cyClient, cxSource, cySource ;
     HDC                         hdc, hdcMem ;
     int                         x, y ;
     PAINTSTRUCT                 ps ;

     switch (message)
     {
     case  WM_CREATE:
                    bitmap.bmBits = bits ;
                    hBitmap      = CreateBitmapIndirect (&bitmap) ;
                    cxSource     = bitmap.bmWidth ;
                    cySource     = bitmap.bmHeight ;
                    return 0 ;

     case  WM_SIZE:
                    cxClient = LOWORD (lParam) ;
                    cyClient = HIWORD (lParam) ;
                    return 0 ;

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

                    hdcMem = CreateCompatibleDC (hdc) ;
                    SelectObject (hdcMem, hBitmap) ;

                    for (y = 0 ; y < cyClient ; y += cySource)
                    for (x = 0 ; x < cxClient ; x += cxSource)
                    {
                              BitBlt (hdc, x, y, cxSource, cySource, hdcMem,
0, 0, SRCCOPY) ;
                    }

                    DeleteDC (hdcMem) ;
                    EndPaint (hwnd, &ps) ;
                    return 0 ;
```

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

您可以尝试一下与彩色点阵图相似的物件。例如，如果您的视讯显示器执行在 256 色模式下，那么您可以根据表 14-2 来定义彩色砖的每个图素。不过，当程式执行在其他显示模式下时，此程式码不起作用。以装置无关方式处理彩色点阵图需要使用下章讨论的 DIB。

点阵图中的画刷

BRICKS 系列的最後一个专案是 BRICKS3，如程式 14-5 所示。乍看此程式，您可能会有这种感觉：程式码哪里去了呢？

程式 14-5　BRICKS3

```
BRICKS3.C
/*------------------------------------------------------------------------
      BRICKS3.C -- CreatePatternBrush Demonstration
                                                (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 ("Bricks3") ;
      HBITMAP                hBitmap ;
      HBRUSH                 hBrush ;
      HWND                   hwnd ;
      MSG              msg ;
      WNDCLASS               wndclass ;

      hBitmap = LoadBitmap (hInstance, TEXT ("Bricks")) ;
      hBrush = CreatePatternBrush (hBitmap) ;
      DeleteObject (hBitmap) ;

      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 ;
      wndclass.lpszMenuName        = NULL ;
      wndclass.lpszClassName       = szAppName ;

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

      hwnd = CreateWindow (szAppName, TEXT ("CreatePatternBrush 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) ;
    }

      DeleteObject (hBrush) ;
      return msg.wParam ;
}

LRESULT CALLBACK WndProc (   HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
      switch (message)
      {
      case  WM_DESTROY:
                       PostQuitMessage (0) ;
                       return 0 ;
    }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

BRICKS3.RC （摘录）

```
//Microsoft Developer Studio generated resource script.
#include "resource.h"
#include "afxres.h"
```

```
//////////////////////////////////////////////////////////////////////////
/
// Bitmap
BRICKS                                  BITMAP  DISCARDABLE     "Bricks.bmp"
```

此程式与 BRICKS1 使用同一个 BRICKS.BMP 档案，而且视窗看上去也相同。

正如您看到的一样，视窗讯息处理程式没有更多的内容。BRICKS3 实际上使用砖块图案作为视窗类别背景画刷，它在 WNDCLASS 结构的 hbrBackground 栏位中定义。

您现在可能猜想 GDI 画刷是很小的点阵图，通常是 8 个图素见方。如果将 LOGBRUSH 结构的 lbStyle 栏位设定为 BS_PATTERN，然後呼叫 CreatePatternBrush 或 CreateBrushIndirect，您就可以在点阵图外面来建立画刷了。此点阵图至少是宽高各 8 个图素。如果再大，Windows 98 将只使用点阵图的左上角作为画刷。而 Windows NT 不受此限制，它会使用整个点阵图。

请记住，画刷和点阵图都是 GDI 物件，而且您应该在程式终止前删除您在程式中建立画刷和点阵图。如果您依据点阵图建立画刷，那么在用画刷画图时，Windows 将复制点阵图位元到画刷所绘制的区域内。呼叫 CreatePatternBrush（或者 CreateBrushIndirect)之後，您可以立即删除点阵图而不会影响到画笔。类似地，您也可以删除画刷而不会影响到您选进的原始点阵图。注意，BRICKS3 在建立画刷後删除了点阵图，并在程式终止前删除了画刷。

绘制点阵图

在视窗中绘图时，我们已经将点阵图当成绘图来源使用过了。这要求先将点阵图选进记忆体装置内容，并呼叫 BitBlt 或者 StretchBlt。您也可以用记忆体装置内容代号作为所有实际呼叫的 GDI 函式中的第一参数。记忆体装置内容的动作与实际的装置内容相同，除非显示平面是点阵图。

程式 14-6 所示的 HELLOBIT 程式展示了此项技术。程式在一个小点阵图上显示了字串「Hello, world!」，然後从点阵图到程式显示区域执行 BitBlt 或 StretchBlt（依照选择的功能表选项而定）。

程式 14-6 HELLOBIT

```
HELLOBIT.C
/*------------------------------------------------------------------------
      HELLOBIT.C --          Bitmap Demonstration
                                                (c) Charles Petzold, 1998
------------------------------------------------------------------------*/

#include <windows.h>
#include "resource.h"
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                PSTR szCmdLine, int iCmdShow)
{
      static TCHAR           szAppName [] = TEXT ("HelloBit") ;
      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        = szAppName ;
      wndclass.lpszClassName       = szAppName ;

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

      hwnd = CreateWindow (szAppName, TEXT ("HelloBit"),
                      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     HBITMAP    hBitmap ;
     static     HDC        hdcMem ;
     static           int    cxBitmap, cyBitmap, cxClient, cyClient, iSize =
IDM_BIG ;
     static TCHAR *   szText = TEXT (" Hello, world! ") ;
     HDC                 hdc ;
     HMENU               hMenu ;
     int                 x, y ;
     PAINTSTRUCT         ps ;
     SIZE                size ;

     switch (message)
     {
     case  WM_CREATE:
                         hdc = GetDC (hwnd) ;
                         hdcMem  = CreateCompatibleDC (hdc) ;

                         GetTextExtentPoint32 (hdc, szText, lstrlen (szText),
&size) ;
                         cxBitmap = size.cx ;
                         cyBitmap = size.cy ;
                         hBitmap  =  CreateCompatibleBitmap  (hdc,  cxBitmap,
cyBitmap) ;

                         ReleaseDC (hwnd, hdc) ;

                         SelectObject (hdcMem, hBitmap) ;
                         TextOut (hdcMem, 0, 0, szText, lstrlen (szText)) ;
                         return 0 ;

     case  WM_SIZE:
                         cxClient = LOWORD (lParam) ;
                         cyClient = HIWORD (lParam) ;
                         return 0 ;

     case  WM_COMMAND:
                         hMenu = GetMenu (hwnd) ;

                         switch (LOWORD (wParam))
                         {
                         case  IDM_BIG:
                         case  IDM_SMALL:
                                        CheckMenuItem     (hMenu,     iSize,
MF_UNCHECKED) ;
                                        iSize = LOWORD (wParam) ;
                                        CheckMenuItem     (hMenu,     iSize,
MF_CHECKED) ;
                                        InvalidateRect (hwnd, NULL, TRUE) ;
```

```
                                        break ;
                    }
                    return 0 ;

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

                    switch (iSize)
                    {
               case  IDM_BIG:
                                        StretchBlt  (hdc,  0,  0,  cxClient,
cyClient,
                         hdcMem, 0, 0, cxBitmap, cyBitmap, SRCCOPY) ;
                                                    break ;

                    case  IDM_SMALL:
                                        for (y = 0 ; y < cyClient ; y += cyBitmap)
                                        for (x = 0 ; x < cxClient ; x += cxBitmap)
                                        {
                         BitBlt (hdc, x, y, cxBitmap, cyBitmap,
                         hdcMem, 0, 0, SRCCOPY) ;
                                             }
                                             break ;
                    }

                    EndPaint (hwnd, &ps) ;
                    return 0 ;

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

HELLOBIT.RC （摘录）

```
//Microsoft Developer Studio generated resource script.
#include "resource.h"
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
/
// Menu
HELLOBIT MENU DISCARDABLE
BEGIN
     POPUP "&Size"
     BEGIN
```

```
                        MENUITEM "&Big",              IDM_BIG, CHECKED
                        MENUITEM "&Small",            IDM_SMALL
      END
END
```

RESOURCE.H （摘录）

```
// Microsoft Developer Studio generated include file.
// Used by HelloBit.rc

#define IDM_BIG           40001
#define IDM_SMALL         40002
```

程式从呼叫 GetTextExtentPoint32 确定字串的图素尺寸开始。这些尺寸将成为与视讯显示相容的点阵图尺寸。当此点阵图被选进记忆体装置内容（也与视讯显示相容）後，再呼叫 TextOut 将文字显示在点阵图上。记忆体装置内容在程式执行期间保留。在处理 WM_DESTROY 资讯期间，HELLOBIT 删除了点阵图和记忆体装置内容。

HELLOBIT 中的一条功能表选项允许您显示点阵图尺寸，此尺寸或者是显示区域中水平和垂直方向平铺的实际尺寸，或者是缩放成显示区域大小的尺寸，如图 14-4 所示。正与您所见到的一样，这不是显示大尺寸字元的好方法！它只是小字体的放大版，并带有放大时产生的锯齿线。

图 14-4 HELLOBIT 的萤幕显示

您可能想知道一个程式，例如 HELLOBIT，是否需要处理 WM_DISPLAYCHANGE 讯息。只要使用者（或者其他应用程式）修改了视讯显示大小或者颜色深度，应用程式就接收到此讯息。其中颜色深度的改变会导致记忆体装置内容和视讯

装置内容不相容。但这并不会发生，因为当显示模式修改後，Windows 自动修改了记忆体装置内容的颜色解析度。选进记忆体装置内容的点阵图仍然保持原样，但不会造成任何问题。

阴影点阵图

在记忆体装置内容绘图（也就是点阵图）的技术是执行「阴影点阵图（shadow bitmap）」的关键。此点阵图包含视窗显示区域中显示的所有内容。这样，对 WM_PAINT 讯息的处理就简化到简单的 BitBlt。

阴影点阵图在绘画程式中最有用。程式 14-7 所示的 SKETCH 程式并不是一个最完美的绘画程式，但它是一个开始。

程式 14-7　SKETCH

```
SKETCH.C
/*------------------------------------------------------------------------
      SKETCH.C -- Shadow Bitmap Demonstration
                                                (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 ("Sketch") ;
      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 ("This program requires Windows
```

```
NT!"),
              szAppName, MB_ICONERROR) ;
                  return 0 ;
      }

      hwnd = CreateWindow (szAppName, TEXT ("Sketch"),
                           WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           NULL, NULL, hInstance, NULL) ;

      if (hwnd == NULL)
      {
                  MessageBox (     NULL, TEXT ("Not enough memory to create
bitmap!"),
                              szAppName, MB_ICONERROR) ;
                  return 0 ;
    }

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

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

void GetLargestDisplayMode (int * pcxBitmap, int * pcyBitmap)
{
      DEVMODE     devmode ;
      int         iModeNum = 0 ;

      * pcxBitmap = * pcyBitmap = 0 ;

      ZeroMemory (&devmode, sizeof (DEVMODE)) ;
      devmode.dmSize = sizeof (DEVMODE) ;

      while (EnumDisplaySettings (NULL, iModeNum++, &devmode))
      {
                  * pcxBitmap = max (* pcxBitmap, (int) devmode.dmPelsWidth) ;
                  * pcyBitmap = max (* pcyBitmap, (int) devmode.dmPelsHeight) ;
      }
}
LRESULT CALLBACK WndProc (   HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
```

```
{
      static BOOL      fLeftButtonDown, fRightButtonDown ;
      static HBITMAP hBitmap ;
      static HDC             hdcMem ;
      static int             cxBitmap, cyBitmap, cxClient, cyClient, xMouse,
yMouse ;
      HDC                              hdc ;
      PAINTSTRUCT            ps ;

      switch (message)
      {
      case  WM_CREATE:
                        GetLargestDisplayMode (&cxBitmap, &cyBitmap) ;

                        hdc = GetDC (hwnd) ;
                        hBitmap   =   CreateCompatibleBitmap   (hdc,   cxBitmap,
cyBitmap) ;
                        hdcMem  = CreateCompatibleDC (hdc) ;
                        ReleaseDC (hwnd, hdc) ;

                  if (!hBitmap)                           //  no  memory  for
bitmap
                  {
                              DeleteDC (hdcMem) ;
                              return -1 ;
                  }

                        SelectObject (hdcMem, hBitmap) ;
                        PatBlt (hdcMem, 0, 0, cxBitmap, cyBitmap, WHITENESS) ;
                        return 0 ;

      case  WM_SIZE:
                        cxClient = LOWORD (lParam) ;
                        cyClient = HIWORD (lParam) ;
                        return 0 ;

      case  WM_LBUTTONDOWN:
                        if (!fRightButtonDown)
                                          SetCapture (hwnd) ;

                        xMouse = LOWORD (lParam) ;
                        yMouse = HIWORD (lParam) ;
                        fLeftButtonDown = TRUE ;
                        return 0 ;

      case  WM_LBUTTONUP:
                        if (fLeftButtonDown)
                                          SetCapture (NULL) ;
```

```
                    fLeftButtonDown = FALSE ;
                    return 0 ;

     case  WM_RBUTTONDOWN:
                    if (!fLeftButtonDown)
                                   SetCapture (hwnd) ;

                    xMouse = LOWORD (lParam) ;
                    yMouse = HIWORD (lParam) ;
                    fRightButtonDown = TRUE ;
                    return 0 ;

     case  WM_RBUTTONUP:
                    if (fRightButtonDown)
                          SetCapture (NULL) ;

                    fRightButtonDown = FALSE ;
                    return 0 ;

     case  WM_MOUSEMOVE:
                    if (!fLeftButtonDown && !fRightButtonDown)
                          return 0 ;

                    hdc = GetDC (hwnd) ;

                    SelectObject (hdc,
                          GetStockObject  (fLeftButtonDown  ?  BLACK_PEN  :
WHITE_PEN)) ;

                    SelectObject (hdcMem,
                          GetStockObject  (fLeftButtonDown  ?  BLACK_PEN  :
WHITE_PEN)) ;

                    MoveToEx (hdc,    xMouse, yMouse, NULL) ;
                    MoveToEx (hdcMem, xMouse, yMouse, NULL) ;

                    xMouse = (short) LOWORD (lParam) ;
                    yMouse = (short) HIWORD (lParam) ;

                    LineTo (hdc,    xMouse, yMouse) ;
                    LineTo (hdcMem, xMouse, yMouse) ;

                    ReleaseDC (hwnd, hdc) ;
                    return 0 ;

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

```
                        BitBlt (hdc, 0, 0, cxClient, cyClient, hdcMem, 0, 0,
SRCCOPY) ;

                        EndPaint (hwnd, &ps) ;
                        return 0 ;

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

要想在 SKETCH 中画线，请按下滑鼠左键并拖动滑鼠。要擦掉画过的东西（更确切地说，是画白线），请按下滑鼠右键并拖动滑鼠。要清空整个视窗，请¡结束程式，然後重新载入，一切从头再来。图 14-5 中显示的 SKETCH 程式图样表达了对苹果公司的麦金塔电脑早期广告的敬意。

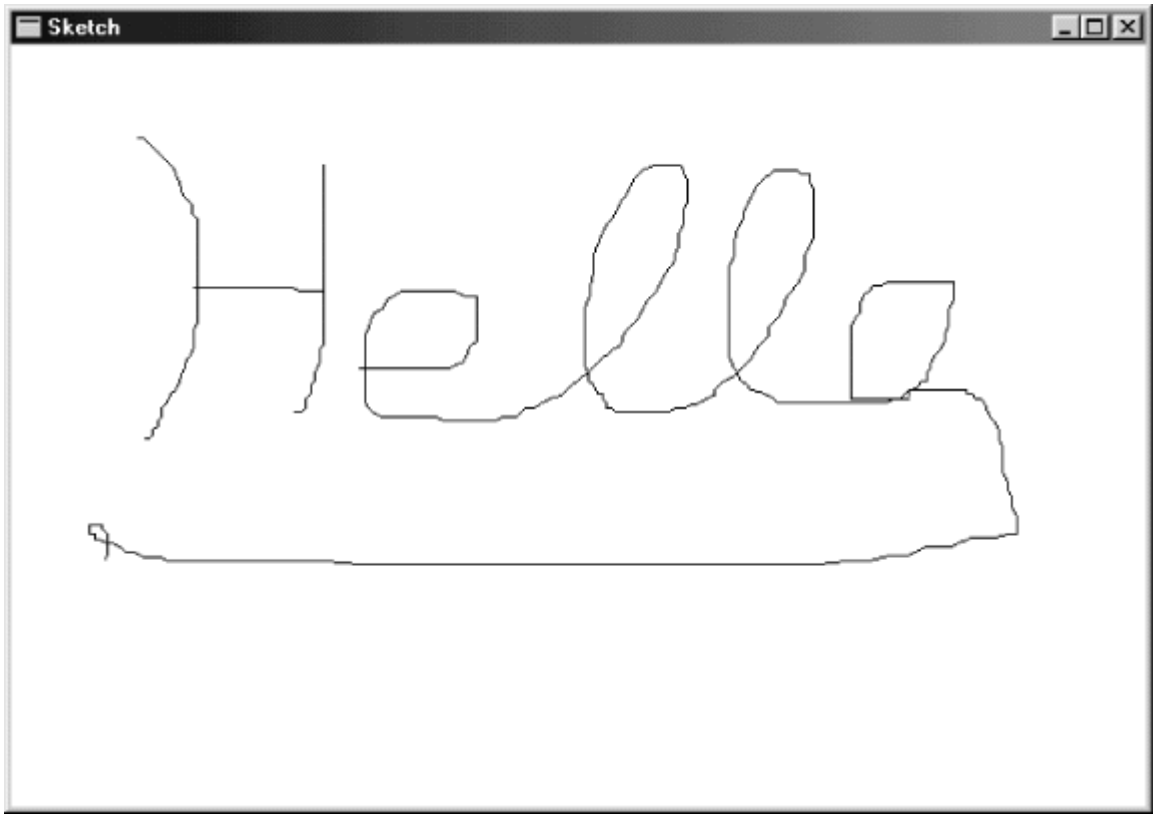


图 14-5 SKETCH 的萤幕显示

此阴影点阵图应多大？在本程式中，它应该大到能包含最大化视窗的整个显示区域。这一问题很容易根据 GetSystemMetrics 资讯计算得出，但如果使用者修改了显示设定後再显示，进而扩大了最大化时视窗的尺寸，这时将发生什么呢？SKETCH 程式在 EnumDisplaySettings 函式的帮助下解决了此问题。此函式使用 DEVMODE 结构来传回全部有效视讯显示模式的资讯。第一次呼叫此函式

时，应将 EnumDisplaySettings 的第二参数设为 0，以後每次呼叫此值都增加。EnumDisplaySettings 传回 FALSE 时完成。

与此同时，SKETCH 将建立一个阴影点阵图，它比目前视讯显示模式的表面还多四倍，而且需要几百万位元组的记忆体。由於如此，SKETCH 将检查点阵图是否建立成功了，如果没有建立，就从 WM_CREATE 传回-1，以表示错误。

在 WM_MOUSEMOVE 讯息处理期间，按下滑鼠左键或者右键，并在记忆体装置内容和显示区域装置内容中画线时，SKETCH 拦截滑鼠。如果画线方式更复杂的话，您可能想在一个函式中实作，程式将呼叫此函式两次——一次画在视讯装置内容上，一次画在记忆体装置内容上。

下面是一个有趣的实验：使 SKETCH 视窗小於全画面尺寸。随著滑鼠左键的按下，将滑鼠拖出视窗的右下角。因为 SKETCH 拦截滑鼠，所以它继续接收并处理 WM_MOUSEMOVE 讯息。现在扩大视窗，您将看到阴影点阵图包含您在 SKETCH 视窗外所画的内容。

在功能表中使用点阵图

您也可以用点阵图在功能表上显示选项。如果您联想起功能表中档案夹、剪贴簿和资源回收筒的图片，那么不要再想那些图片了。您应该考虑一下，功能表上显示点阵图对画图程式用途有多大，想像一下在功能表中使用不同字体和字体大小、线宽、阴影图案以及颜色。

GRAFMENU 是展示图形功能表选项的范例程式。此程式顶层功能表如图 14-6 所示。放大的字母来自於 40×16 图素的单色点阵图档案，该档案在 Visual C++ Developer Studio 建立。从功能表上选择「FONT」将弹出三个选择项——「Courier New」、「 Arial」和「Times New Roman」。它们是标准的 Windows TrueType 字体，并且每一个都按其相关的字体显示，如图 14-7 所示。这些点阵图在程式中用记忆体装置内容建立。

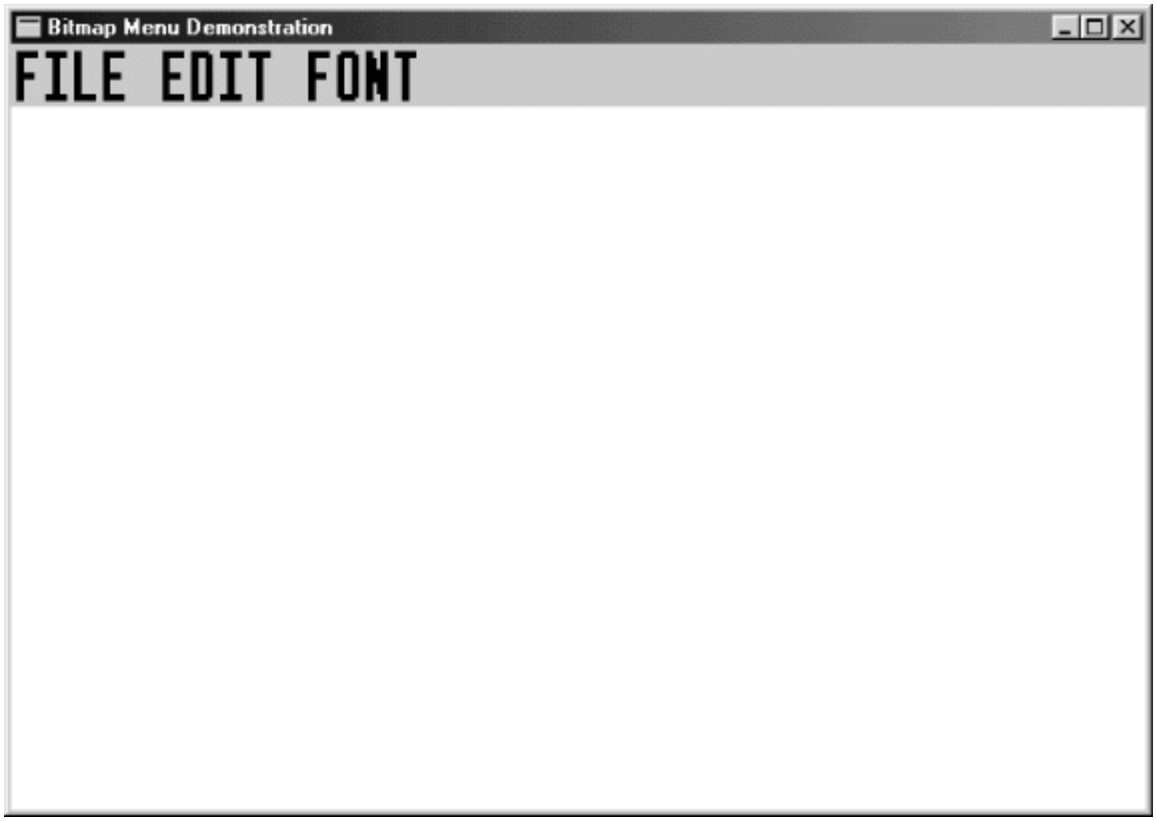


图 14-6 GRAFMENU 程式的顶层功能表

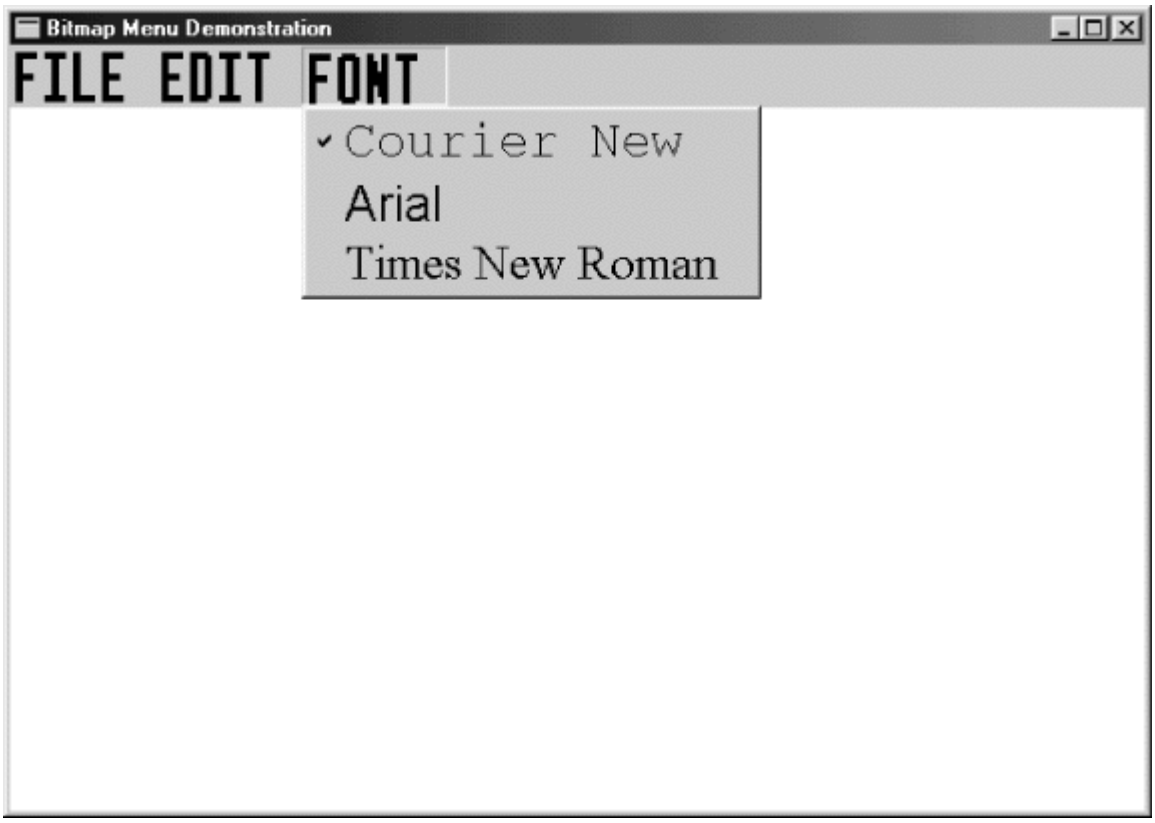


图 14-7 GRAFMENU 程式弹出的「FONT」功能表

最後，在拉下系统功能表时，您将获得一些「辅助」资讯，用「HELP」表示了新使用者的线上求助项目（参见图 14-8）。此 64×64 图素的单色点阵图是在 Developer Studio 中建立的。

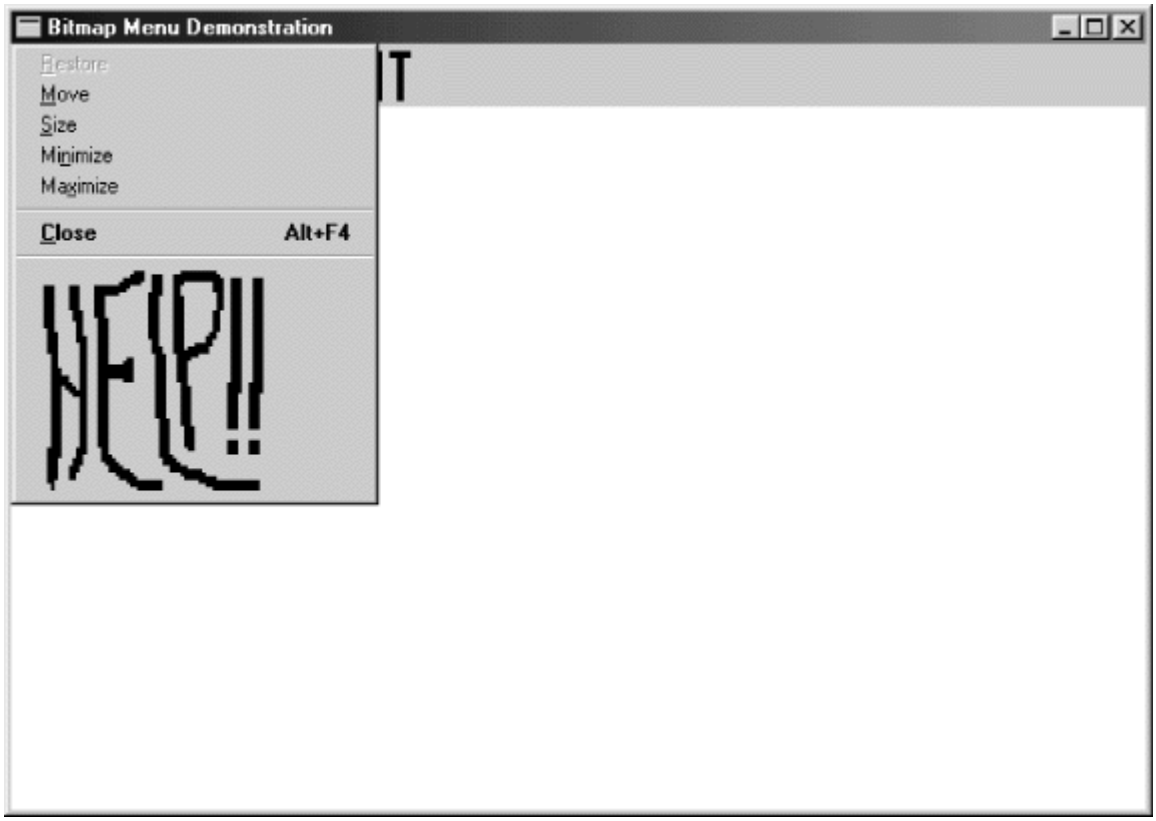


图 14-8 GRAFMENU 程式系统功能表

GRAFMENU 程式，包括四个 Developer Studio 中建立的点阵图，如程式 14-8 所示。

程式 14-8 GRAFMENU

```
GRAFMENU.C
/*------------------------------------------------------------------------
--
      GRAFMENU.C --          Demonstrates Bitmap Menu Items
                                                (c) Charles Petzold, 1998
------------------------------------------------------------------------
-*/

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

LRESULT           CALLBACK WndProc       (HWND, UINT, WPARAM, LPARAM) ;
void              AddHelpToSys           (HINSTANCE, HWND) ;
HMENU             CreateMyMenu           (HINSTANCE) ;
HBITMAP           StretchBitmap        (HBITMAP) ;
HBITMAP           GetBitmapFont          (int) ;
void              DeleteAllBitmaps     (HWND) ;
TCHAR szAppName[] = TEXT ("GrafMenu") ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
      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 ("This program requires Windows
NT!"),
                           szAppName, MB_ICONERROR) ;
               return 0 ;
    }

     hwnd = CreateWindow (szAppName,TEXT ("Bitmap Menu Demonstration"),
                     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 iMsg, WPARAM wParam,LPARAM lParam)
{
     HMENU                         hMenu ;
     static int       iCurrentFont = IDM_FONT_COUR ;

     switch (iMsg)
     {
     case WM_CREATE:
                     AddHelpToSys   (((LPCREATESTRUCT)   lParam)->hInstance,
```

```
hwnd) ;
                    hMenu      =      CreateMyMenu      (((LPCREATESTRUCT)
lParam)->hInstance) ;
                    SetMenu (hwnd, hMenu) ;
                    CheckMenuItem (hMenu, iCurrentFont, MF_CHECKED) ;
                    return 0 ;

     case  WM_SYSCOMMAND:
                    switch (LOWORD (wParam))
                    {
                    case  IDM_HELP:
                                   MessageBox (hwnd, TEXT ("Help not yet
implemented!"),
                         szAppName, MB_OK | MB_ICONEXCLAMATION) ;
                                   return 0 ;
                    }
                    break ;

     case  WM_COMMAND:
                    switch (LOWORD (wParam))
                    {
                    case  IDM_FILE_NEW:
                    case  IDM_FILE_OPEN:
                    case  IDM_FILE_SAVE:
                    case  IDM_FILE_SAVE_AS:
                    case  IDM_EDIT_UNDO:
                    case  IDM_EDIT_CUT:
                    case  IDM_EDIT_COPY:
                    case  IDM_EDIT_PASTE:
                    case  IDM_EDIT_CLEAR:
                                   MessageBeep (0) ;
                                   return 0 ;

                    case  IDM_FONT_COUR:
                    case  IDM_FONT_ARIAL:
                    case  IDM_FONT_TIMES:
                                hMenu = GetMenu (hwnd) ;
                    CheckMenuItem (hMenu, iCurrentFont, MF_UNCHECKED) ;
                                   iCurrentFont = LOWORD (wParam) ;
                                   CheckMenuItem  (hMenu,  iCurrentFont,
MF_CHECKED) ;
                                   return 0 ;
                    }
                    break ;

          case  WM_DESTROY:
                         DeleteAllBitmaps (hwnd) ;
                         PostQuitMessage (0) ;
```

```
                              return 0 ;
      }
      return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}

/*------------------------------------------------------------------------
      AddHelpToSys: Adds bitmap Help item to system menu
  ------------------------------------------------------------------------*/

void AddHelpToSys (HINSTANCE hInstance, HWND hwnd)
{
      HBITMAP           hBitmap ;
      HMENU             hMenu ;

      hMenu = GetSystemMenu (hwnd, FALSE);
      hBitmap = StretchBitmap (LoadBitmap (hInstance, TEXT ("BitmapHelp"))) ;
      AppendMenu (hMenu, MF_SEPARATOR, 0, NULL) ;
      AppendMenu (hMenu, MF_BITMAP, IDM_HELP, (PTSTR) (LONG) hBitmap) ;
}

/*------------------------------------------------------------------------
   CreateMyMenu: Assembles menu from components
------------------------------------------------------------------------*/

HMENU CreateMyMenu (HINSTANCE hInstance)
{
      HBITMAP           hBitmap ;
      HMENU             hMenu, hMenuPopup ;
      int               i ;

      hMenu = CreateMenu () ;
      hMenuPopup = LoadMenu (hInstance, TEXT ("MenuFile")) ;
      hBitmap = StretchBitmap (LoadBitmap (hInstance, TEXT ("BitmapFile"))) ;
      AppendMenu (hMenu, MF_BITMAP | MF_POPUP, (int) hMenuPopup,
                                                  (PTSTR) (LONG) hBitmap) ;
      hMenuPopup = LoadMenu (hInstance, TEXT ("MenuEdit")) ;
      hBitmap = StretchBitmap (LoadBitmap (hInstance, TEXT ("BitmapEdit"))) ;
      AppendMenu (hMenu, MF_BITMAP | MF_POPUP, (int) hMenuPopup,
                      (PTSTR) (LONG) hBitmap) ;
      hMenuPopup = CreateMenu () ;
      for (i = 0 ; i < 3 ; i++)
     {
                  hBitmap = GetBitmapFont (i) ;
                  AppendMenu (hMenuPopup, MF_BITMAP, IDM_FONT_COUR + i,
                      (PTSTR) (LONG) hBitmap) ;
      }

      hBitmap = StretchBitmap (LoadBitmap (hInstance, TEXT ("BitmapFont"))) ;
```

```
     AppendMenu (hMenu, MF_BITMAP | MF_POPUP, (int) hMenuPopup,
                    (PTSTR) (LONG) hBitmap) ;
     return hMenu ;
}

/*------------------------------------------------------------------------
   StretchBitmap: Scales bitmap to display resolution
------------------------------------------------------------------------*
/

HBITMAP StretchBitmap (HBITMAP hBitmap1)
{
     BITMAP                bm1, bm2 ;
     HBITMAP               hBitmap2 ;
     HDC                   hdc, hdcMem1, hdcMem2 ;
     int                   cxChar, cyChar ;

                // Get the width and height of a system font character

     cxChar = LOWORD (GetDialogBaseUnits ()) ;
     cyChar = HIWORD (GetDialogBaseUnits ()) ;

                // Create 2 memory DCs compatible with the display
     hdc = CreateIC (TEXT ("DISPLAY"), NULL, NULL, NULL) ;
     hdcMem1 = CreateCompatibleDC (hdc) ;
     hdcMem2 = CreateCompatibleDC (hdc) ;
     DeleteDC (hdc) ;

                // Get the dimensions of the bitmap to be stretched
     GetObject (hBitmap1, sizeof (BITMAP), (PTSTR) &bm1) ;
                // Scale these dimensions based on the system font size
     bm2 = bm1 ;
     bm2.bmWidth           = (cxChar * bm2.bmWidth)  / 4 ;
     bm2.bmHeight          = (cyChar * bm2.bmHeight) / 8 ;
     bm2.bmWidthBytes      = ((bm2.bmWidth + 15) / 16) * 2 ;

                // Create a new bitmap of larger size

     hBitmap2 = CreateBitmapIndirect (&bm2) ;
                // Select the bitmaps in the memory DCs and do a StretchBlt
     SelectObject (hdcMem1, hBitmap1) ;
     SelectObject (hdcMem2, hBitmap2) ;
     StretchBlt (hdcMem2, 0, 0, bm2.bmWidth, bm2.bmHeight,
                hdcMem1, 0, 0, bm1.bmWidth, bm1.bmHeight, SRCCOPY) ;
                      // Clean up
     DeleteDC (hdcMem1) ;
     DeleteDC (hdcMem2) ;
     DeleteObject (hBitmap1) ;
```

```
     return hBitmap2 ;
}

/*-------------------------------------------------------------------------
-
   GetBitmapFont: Creates bitmaps with font names
-------------------------------------------------------------------------
-*/

HBITMAP GetBitmapFont (int i)
{
     static TCHAR  * szFaceName[3]= {   TEXT ("Courier New"), TEXT ("Arial"),
      TEXT ("Times New Roman") } ;
     HBITMAP                     hBitmap ;
     HDC                         hdc, hdcMem ;
     HFONT                       hFont ;
     SIZE                        size ;
     TEXTMETRIC                  tm ;

     hdc = CreateIC (TEXT ("DISPLAY"), NULL, NULL, NULL) ;
     GetTextMetrics (hdc, &tm) ;

     hdcMem      = CreateCompatibleDC (hdc) ;
     hFont       = CreateFont (2 * tm.tmHeight, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0,
                         szFaceName[i]) ;

     hFont = (HFONT) SelectObject (hdcMem, hFont) ;
     GetTextExtentPoint32 (hdcMem, szFaceName[i],
      lstrlen (szFaceName[i]), &size);
     hBitmap = CreateBitmap (size.cx, size.cy, 1, 1, NULL) ;
     SelectObject (hdcMem, hBitmap) ;

     TextOut (hdcMem, 0, 0, szFaceName[i], lstrlen (szFaceName[i])) ;
     DeleteObject (SelectObject (hdcMem, hFont)) ;
     DeleteDC (hdcMem) ;
     DeleteDC (hdc) ;

     return hBitmap ;
}

/*-------------------------------------------------------------------------
-
   DeleteAllBitmaps: Deletes all the bitmaps in the menu

-------------------------------------------------------------------------*/
```

```
void DeleteAllBitmaps (HWND hwnd)
{
      HMENU                         hMenu ;
      int                     i ;
      MENUITEMINFO mii = { sizeof (MENUITEMINFO), MIIM_SUBMENU | MIIM_TYPE } ;
                        // Delete Help bitmap on system menu
      hMenu = GetSystemMenu (hwnd, FALSE);
      GetMenuItemInfo (hMenu, IDM_HELP, FALSE, &mii) ;
      DeleteObject ((HBITMAP) mii.dwTypeData) ;

                        // Delete top-level menu bitmaps
      hMenu = GetMenu (hwnd) ;
      for (i = 0 ; i < 3 ; i++)
      {
                        GetMenuItemInfo (hMenu, i, TRUE, &mii) ;
                        DeleteObject ((HBITMAP) mii.dwTypeData) ;
      }

                        // Delete bitmap items on Font menu
      hMenu = mii.hSubMenu ;;
      for (i = 0 ; i < 3 ; i++)
      {
                  GetMenuItemInfo (hMenu, i, TRUE, &mii) ;
                  DeleteObject ((HBITMAP) mii.dwTypeData) ;
      }
}
```

GRAFMENU.RC （摘录）

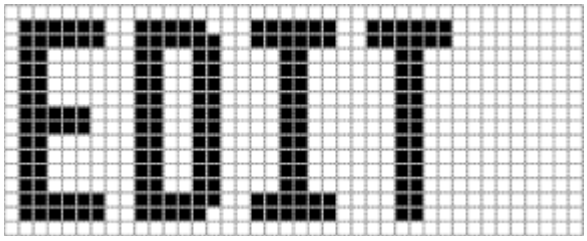
```
//Microsoft Developer Studio generated resource script.
#include "resource.h"
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
/
// Menu
MENUFILE MENU DISCARDABLE
BEGIN
            MENUITEM "&New",                    IDM_FILE_NEW
            MENUITEM "&Open...",                IDM_FILE_OPEN
            MENUITEM "&Save",                   IDM_FILE_SAVE
            MENUITEM "Save &As...",             IDM_FILE_SAVE_AS
END
MENUEDIT MENU DISCARDABLE
BEGIN
            MENUITEM "&Undo",                   IDM_EDIT_UNDO
            MENUITEM SEPARATOR
            MENUITEM "Cu&t",                    IDM_EDIT_CUT
            MENUITEM "&Copy",                   IDM_EDIT_COPY
            MENUITEM "&Paste",                  IDM_EDIT_PASTE
```

```
            MENUITEM "De&lete",          IDM_EDIT_CLEAR
END

//////////////////////////////////////////////////////////////////////////////
/
// Bitmap
BITMAPFONT     BITMAP    DISCARDABLE    "Fontlabl.bmp"
BITMAPHELP     BITMAP    DISCARDABLE    "Bighelp.bmp"
BITMAPEDIT     BITMAP    DISCARDABLE    "Editlabl.bmp"
BITMAPFILE     BITMAP    DISCARDABLE    "Filelabl.bmp"
```

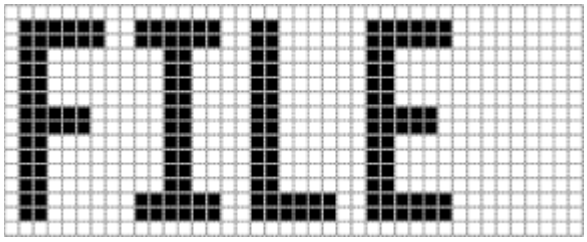
RESOURCE.H （摘录）

```
// Microsoft Developer Studio generated include file.
// Used by GrafMenu.rc

#define IDM_FONT_COUR    101
#define IDM_FONT_ARIAL   102
#define IDM_FONT_TIMES   103
#define IDM_HELP        104
#define IDM_EDIT_UNDO    40005
#define IDM_EDIT_CUT     40006
#define IDM_EDIT_COPY    40007
#define IDM_EDIT_PASTE   40008
#define IDM_EDIT_CLEAR   40009
#define IDM_FILE_NEW     40010
#define IDM_FILE_OPEN    40011
#define IDM_FILE_SAVE    40012
#define IDM_FILE_SAVE_AS 40013
```

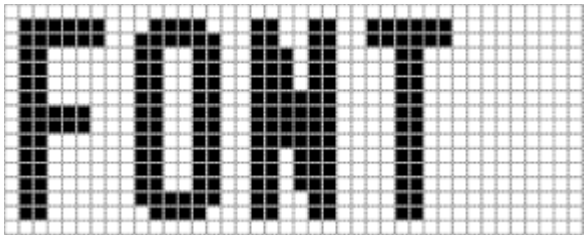
EDITLABL.BMP



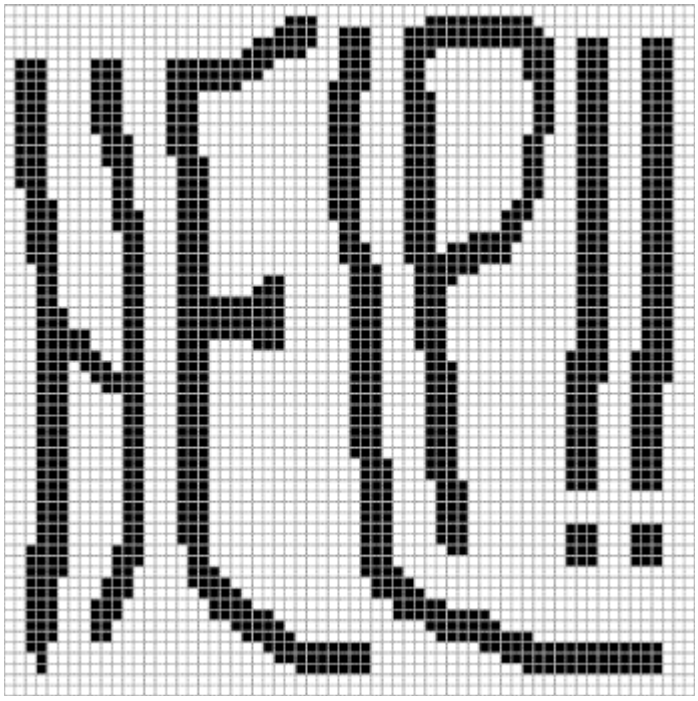
FILELABL.BMP



FONTLABL.BMP



BIGHELP.BMP



要将点阵图插入功能表，可以利用 AppendMenu 或 InsertMenu。点阵图有两个来源：可以在 Visual C++ Developer Studio 建立点阵图，包括资源脚本中的点阵图档案，并在程式使用 LoadBitmap 时将点阵图资源载入到记忆体，然後呼叫 AppendMenu 或 InsertMenu 将点阵图附加到功能表上。但是用这种方法会有一些问题：点阵图不适於所有显示模式的解析度和纵横比；有时您需要缩放载入的点阵图以解决此问题。另一种方法是：在程式内部建立点阵图，并将它选进记忆体装置内容，画出来，然後再附加到功能表中。

GRAFMENU 中的 GetBitmapFont 函式的参数为 0、1 或 2，传回一个点阵图代号。此点阵图包含字串「Courier New」、「Arial」或「Times New Roman」，而且字体是各自对应的字体，大小是正常系统字体的两倍。让我们看看 GetBitmapFont 是怎么做的。（下面的程式码与 GRAFMENU.C 档案中的有些不同。为了清楚起见，我用「Arial」字体相应的值代替了引用 szFaceName 阵列。）

第一步是用 TEXTMETRIC 结构来确定目前系统字体的大小，并建立一个与目前萤幕相容的记忆体装置内容：

```
hdc = CreateIC (TEXT ("DISPLAY"), NULL, NULL, NULL) ;
GetTextMetrics (hdc, &tm) ;
hdcMem = CreateCompatibleDC (hdc) ;
```

CreateFont 函式建立了一种逻辑字体，该字体高是系统字体的两倍，而且逻辑名称为「Arial」：

```
hFont = CreateFont (2 * tm.tmHeight, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                          TEXT ("Arial")) ;
```

从记忆体装置内容中选择该字体，然後储存内定字体代号：

```
hFont = (HFONT) SelectObject (hdcMem, hFont) ;
```

现在，当我们向记忆体装置内容写一些文字时，Windows 就会使用选进装置内容的 TrueType Arial 字体了。

但这个记忆体装置内容最初只有一个单图素单色设备平面。我们必须建立一个足够大的点阵图以容纳我们所要显示的文字。通过 GetTextExtentPoint32 函式，可以取得文字的大小，而用 CreateBitmap 可以根据这些尺寸来建立点阵图：

```
GetTextExtentPoint32 (hdcMem, TEXT ("Arial"), 5, &size) ;
hBitmap = CreateBitmap (size.cx, size.cy, 1, 1, NULL) ;
SelectObject (hdcMem, hBitmap) ;
```

现在这个装置内容是一个单色的显示平面，大小也是严格的文字尺寸。我们现在要做的就是书写文字：

```
TextOut (hdcMem, 0, 0, TEXT ("Arial"), 5) ;
```

除了清除，所有的工作都完成了。要清除，我们可以用 SelectObject 将系统字体（带有代号 hFont）重新选进装置内容，然後删除 SelectObject 传回的前一个字体代号，也就是 Arial 字体代号：

```
DeleteObject (SelectObject (hdcMem, hFont)) ;
```

现在可以删除两个装置内容：

```
DeleteDC (hdcMem) ;
DeleteDC (hdc) ;
```

这样，我们就获得了一个点阵图，该点阵图上有 Arial 字体的字串「Arial」。

当我们需要缩放字体以适应不同显示解析度或纵横比时，记忆体装置内容也能解决问题。在 GRAFMENU 程式中，我建立了四个点阵图，这些点阵图只适用於系统字体高 8 图素、宽 4 图素的显示。对於其他尺寸的系统字体，只能缩放点阵图。GRAFMENU 中的 StretchBitmap 函式完成此功能。

第一步是获得显示的装置内容，然後取得系统字体的文字规格，接下来建立两个记忆体装置内容：

```
hdc = CreateIC (TEXT ("DISPLAY"), NULL, NULL, NULL) ;
GetTextMetrics (hdc, &tm) ;
hdcMem1 = CreateCompatibleDC (hdc) ;
hdcMem2 = CreateCompatibleDC (hdc) ;
DeleteDC (hdc) ;
```

传递给函式的点阵图代号是 hBitmap1。程式能用 GetObject 获得点阵图的

大小：

```
GetObject (hBitmap1, sizeof (BITMAP), (PSTR) &bm1) ;
```

此操作将尺寸复制到 BITMAP 型态的结构 bm1 中。结构 bm2 等於结构 bm1，然後根据系统字体大小来修改某些栏位：

```
bm2 = bm1 ;
bm2.bmWidth            = (tm.tmAveCharWidth * bm2.bmWidth)  / 4 ;
bm2.bmHeight           = (tm.tmHeight      * bm2.bmHeight) / 8 ;
bm2.bmWidthBytes       = ((bm2.bmWidth + 15) / 16) * 2 ;
```

下一个点阵图带有代号 hBitmap2，可以根据动态的尺寸建立：

```
hBitmap2 = CreateBitmapIndirect (&bm2) ;
```

然後将这两个点阵图选进两个记忆体装置内容中：

```
SelectObject (hdcMem1, hBitmap1) ;
SelectObject (hdcMem2, hBitmap2) ;
```

我们想把第一个点阵图复制给第二个点阵图，并在此程序中进行拉伸。这包括 StretchBlt 呼叫：

```
StretchBlt (     hdcMem2, 0, 0, bm2.bmWidth, bm2.bmHeight,
                           hdcMem1, 0, 0, bm1.bmWidth, bm1.bmHeight, SRCCOPY) ;
```

现在第二幅图适当地缩放了，我们可将其用到功能表中。剩下的清除工作很简单：

```
DeleteDC (hdcMem1) ;
DeleteDC (hdcMem2) ;
DeleteObject (hBitmap1) ;
```

在建造功能表时，GRAFMENU 中的 CreateMyMenu 函式呼叫了 StretchBitmap 和 GetBitmapFont 函式。GRAFMENU 在资源档案中定义了两个功能表，在选择「File」和「Edit」选项时会弹出这两个功能表。函式开始先取得一个空功能表的代号：

```
hMenu = CreateMenu () ;
```

从资源档案载入「File」的突现式功能表（包括四个选项：「New」、「Open」、「Save」和「Save as」）：

```
hMenuPopup = LoadMenu (hInstance, TEXT ("MenuFile")) ;
```

从资源档案还载入了包含「FILE」的点阵图，并用 StretchBitmap 进行了拉伸：

```
hBitmapFile = StretchBitmap (LoadBitmap (hInstance, TEXT ("BitmapFile"))) ;
```

点阵图代号和突现式功能表代号都是 AppendMenu 呼叫的参数：

```
AppendMenu (hMenu, MF_BITMAP | MF_POPUP, hMenuPopup, (PTSTR) (LONG)
hBitmapFile) ;
```

「Edit」功能表类似程序如下：

```
hMenuPopup  = LoadMenu (hInstance, TEXT ("MenuEdit")) ;
hBitmapEdit       = StretchBitmap (LoadBitmap (hInstance, TEXT ("BitmapEdit"))) ;
AppendMenu (hMenu, MF_BITMAP | MF_POPUP, hMenuPopup, (PTSTR) (LONG) hBitmapEdit) ;
```

呼叫 GetBitmapFont 函式可以构造这三种不同字体的突现式功能表：

```
hMenuPopup = CreateMenu () ;
for (i = 0 ; i < 3 ; i++)
{
      hBitmapPopFont [i] = GetBitmapFont (i) ;
      AppendMenu (hMenuPopup, MF_BITMAP, IDM_FONT_COUR + i,
                  (PTSTR) (LONG) hMenuPopupFont [i]) ;
}
```

然後将突现式功能表添加到功能表中：

```
hBitmapFont = StretchBitmap (LoadBitmap (hInstance, "BitmapFont")) ;
AppendMenu (     hMenu, MF_BITMAP | MF_POPUP, hMenuPopup, (PTSTR) (LONG)
                           hBitmapFont) ;
```

WndProc 通过呼叫 SetMenu，完成了视窗功能表的建立工作。

GRAFMENU 还改变了 AddHelpToSys 函式中的系统功能表。此函式首先获得一个系统功能表代号：

```
hMenu = GetSystemMenu (hwnd, FALSE) ;
```

这将载入「HELP」点阵图，并将其拉伸到适当尺寸：

```
hBitmapHelp = StretchBitmap (LoadBitmap (hInstance, TEXT ("BitmapHelp"))) ;
```

这将给系统功能表添加一条分隔线和拉伸的点阵图：

```
AppendMenu (hMenu, MF_SEPARATOR, 0, NULL) ;
AppendMenu (hMenu, MF_BITMAP, IDM_HELP, (PTSTR)(LONG) hBitmapHelp) ;
```

GRAFMENU 在退出之前呼叫一个函式来清除并删除所有点阵图。

下面是在功能表中使用点阵图的一些注意事项。

在顶层功能表中，Windows 调整功能表列的高度以适应最高的点阵图。其他点阵图(或字串)是根据功能表列的顶端对齐的。如果在顶层功能表中使用了点阵图，那么从使用常数 SM_CYMENU 的 GetSystemMetrics 得到的功能表列大小将不再有效。

执行 GRAFMENU 期间可以看到：在突现式功能表中，您可使用带有点阵图功能表项的勾选标记，但勾选标记是正常尺寸。如果不满意，您可以建立一个自订的勾选标记，并使用 SetMenuItemBitmaps。

在功能表中使用非文字（或者使用非系统字体的文字）的另一种方法是「拥有者绘制」功能表。

功能表的键盘介面是另一个问题。当功能表含有文字时，Windows 会自动添加键盘介面。要选择一个功能表项，可以使用 Alt 与字串中的一个字母的组合键。而一旦在功能表中放置了点阵图，就删除了键盘介面。即使点阵图表达了一定的含义，但 Windows 并不知道。

目前我们可以使用 WM_MENUCHAR 讯息。当您按下 Alt 和与功能表项不相符的一个字元键的组合键时，Windows 将向您的视窗讯息处理程式发送一个

WM_MENUCHAR 讯息。GRAFMENU 需要截取 WM_MENUCHAR 讯息并检查 wParam 的值(即按键的 ASCII 码）。如果这个值对应一个功能表项，那么向 Windows 传回双字组：其中高字组为 2，低字组是与该键相关的功能表项索引值。然後由 Windows 处理余下的事。

非矩形点阵图图像

点阵图都是矩形，但不需要都显示成矩形。例如，假定您有一个矩形点阵图图像，但您却想将它显示成椭圆形。

首先，这听起来很简单。您只需将图像载入 Visual C++ Developer Studio 或者 Windows 的「画图」程式（或者更昂贵的应用程式），然後用白色的画笔将图像四周画上白色。这时将获得一幅椭圆形的图像，而椭圆的外面就成了白色。只有当背景色为白色时此点阵图才能正确显示，如果在其他背景色上显示，您就会发现椭圆形的图像和背景之间有一个白色的矩形。这种效果不好。

有一种非常通用的技术可解决此类问题。这种技术包括「遮罩（mask）」点阵图和一些位元映射操作。遮罩是一种单色点阵图，它与您要显示的矩形点阵图图像尺寸相同。每个遮罩的图素都对应点阵图图像的一个图素。遮罩图素是 1(白色)，对应著点阵图图素显示；是 0(黑色)，则显示背景色。（或者遮罩点阵图与此相反，这根据您使用的位元映射操作而有一些相对应的变化。）

让我们看看 BITMASK 程式是如何实作这一技术的。如程式 14-9 所示。

程式 14-9　BITMASK

```
BITMASK.C
/*-------------------------------------------------------------------------
-
      BITMASK.C --      Bitmap Masking Demonstration
                        (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 ("BitMask") ;
      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 (LTGRAY_BRUSH) ;
     wndclass.lpszMenuName       = NULL ;
     wndclass.lpszClassName      = szAppName ;

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

     hwnd = CreateWindow (szAppName, TEXT ("Bitmap Masking 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 HBITMAP         hBitmapImag, hBitmapMask ;
     static HINSTANCE       hInstance ;
     static int             cxClient, cyClient, cxBitmap, cyBitmap ;
     BITMAP                 bitmap ;
     HDC                    hdc, hdcMemImag, hdcMemMask ;
     int                    x, y ;
     PAINTSTRUCT            ps ;

     switch (message)
    {
     case WM_CREATE:
                          hInstance = ((LPCREATESTRUCT) lParam)->hInstance ;
                                    // Load the original image and get its size
```

```
                    hBitmapImag = LoadBitmap (hInstance, TEXT ("Matthew")) ;
                    GetObject (hBitmapImag, sizeof (BITMAP), &bitmap) ;
                    cxBitmap = bitmap.bmWidth ;
                    cyBitmap = bitmap.bmHeight ;

                              // Select the original image into a memory DC
                    hdcMemImag  = CreateCompatibleDC (NULL) ;
                    SelectObject (hdcMemImag, hBitmapImag) ;
                              // Create the monochrome mask bitmap and memory
DC
                    hBitmapMask = CreateBitmap (cxBitmap, cyBitmap, 1, 1,
NULL) ;
                    hdcMemMask = CreateCompatibleDC (NULL) ;
                    SelectObject (hdcMemMask, hBitmapMask) ;

                              // Color the mask bitmap black with a white
ellipse
                    SelectObject        (hdcMemMask,        GetStockObject
(BLACK_BRUSH)) ;
                    Rectangle (hdcMemMask, 0, 0, cxBitmap, cyBitmap) ;
                    SelectObject        (hdcMemMask,        GetStockObject
(WHITE_BRUSH)) ;
                    Ellipse (hdcMemMask, 0, 0, cxBitmap, cyBitmap) ;

                              // Mask the original image
                    BitBlt (hdcMemImag, 0, 0, cxBitmap, cyBitmap,
                         hdcMemMask, 0, 0, SRCAND) ;
                    DeleteDC (hdcMemImag) ;
                    DeleteDC (hdcMemMask) ;
                    return 0 ;

     case  WM_SIZE:
                    cxClient = LOWORD (lParam) ;
                    cyClient = HIWORD (lParam) ;
                    return 0 ;

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

                              // Select bitmaps into memory DCs

                    hdcMemImag = CreateCompatibleDC (hdc) ;
                    SelectObject (hdcMemImag, hBitmapImag) ;

                    hdcMemMask = CreateCompatibleDC (hdc) ;
                    SelectObject (hdcMemMask, hBitmapMask) ;

                              // Center image
```

```
                        x = (cxClient - cxBitmap) / 2 ;
                        y = (cyClient - cyBitmap) / 2 ;

                                    // Do the bitblts

                        BitBlt (hdc, x, y, cxBitmap, cyBitmap, hdcMemMask, 0, 0,
0x220326) ;
                        BitBlt (hdc, x, y, cxBitmap, cyBitmap, hdcMemImag, 0, 0,
SRCPAINT) ;

                        DeleteDC (hdcMemImag) ;
                        DeleteDC (hdcMemMask) ;
                        EndPaint (hwnd, &ps) ;
                        return 0 ;

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

BITMASK.RC

```
// Microsoft Developer Studio generated resource script.
#include "resource.h"
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////
/
// Bitmap
MATTHEW                 BITMAP        DISCARDABLE     "matthew.bmp"
```

资源档案中的 MATTHEW.BMP 档案是我侄子的一幅黑白数位照片，宽 200 图素，高 320 图素，每图素 8 位元。不过，另外制作个 BITMASK 只是因为此档案的内容是任何东西都可以。

注意，BITMASK 将视窗背景设为亮灰色。这样就确保我们能正确地遮罩点阵图，而不只是将其涂成白色。

下面让我们看一下 WM_CREATE 的处理程序：BITMASK 用 LoadBitmap 函式获得 hBitmapImag 变数中原始图像的代号。用 GetObject 函式可取得点阵图的宽度高度。然後将点阵图代号选进代号为 hdcMemImag 的记忆体装置内容中。

程式建立的下一个单色点阵图与原来的图大小相同，其代号储存在 hBitmapMask，并选进代号为 hdcMemMask 的记忆体装置内容中。在记忆体装置内容中，使用 GDI 函式，遮罩点阵图就涂成了黑色背景和一个白色的椭圆：

```
SelectObject (hdcMemMask, GetStockObject (BLACK_BRUSH)) ;
Rectangle (hdcMemMask, 0, 0, cxBitmap, cyBitmap) ;
SelectObject (hdcMemMask, GetStockObject (WHITE_BRUSH)) ;
Ellipse (hdcMemMask, 0, 0, cxBitmap, cyBitmap) ;
```

因为这是一个单色的点阵图，所以黑色区域的位元是 0，而白色区域的位元是 1。

然後 BitBlt 呼叫就按此遮罩修改了原图像：

```
BitBlt (hdcMemImag, 0, 0, cxBitmap, cyBitmap,
                        hdcMemMask, 0, 0, SRCAND) ;
```

SRCAND 位元映射操作在来源位元（遮罩点阵图）和目的位元（原图像）之间执行了位元 AND 操作。只要遮罩点阵图是白色，就显示目的；只要遮罩是黑色，则目的就也是黑色。现在原图像中就形成了一个黑色包围的椭圆区域。

现在让我们看一下 WM_PAINT 处理程序。此程序同时改变了选进记忆体装置内容中的图像点阵图和遮罩点阵图。两次 BitBlt 呼叫完成了这个魔术，第一次在视窗上执行遮罩点阵图的 BitBlt：

```
BitBlt (hdc, x, y, cxBitmap, cyBitmap, hdcMemMask, 0, 0, 0x220326) ;
```

这里使用了一个没有名称的位元映射操作。逻辑运算子是 D & ~S。回忆来源——即遮罩点阵图——是黑色（位元值 0）包围的一个白色（位元值 1）椭圆。位元映射操作首先将来源反色，也就是改成白色包围的黑色椭圆。然後位元操作在这个已转换的来源和目的（即视窗上）之间执行位元 AND 操作。当目的和位元值 1「AND」时保持不变；与位元值 0「AND」时，目的将变黑。因此，BitBlt 操作将在视窗上画一个黑色的椭圆。

第二次的 BitBlt 呼叫则在视窗中绘制图像点阵图：

```
BitBlt (hdc, x, y, cxBitmap, cyBitmap, hdcMemImag, 0, 0, SRCPAINT) ;
```

位元映射操作在来源和目的之间执行位元「OR」操作。由於来源点阵图的外面是黑色，因此保持目的不变；而在椭圆区域内，目的是黑色，因此图像就原封不动地复制了过来。执行结果如图 14-9 所示。

注意事项：

有时您需要一个很复杂的遮罩——例如，抹去原始图像的整个背景。您将需要在画图程式中手工建立然後将其储存到成档案。

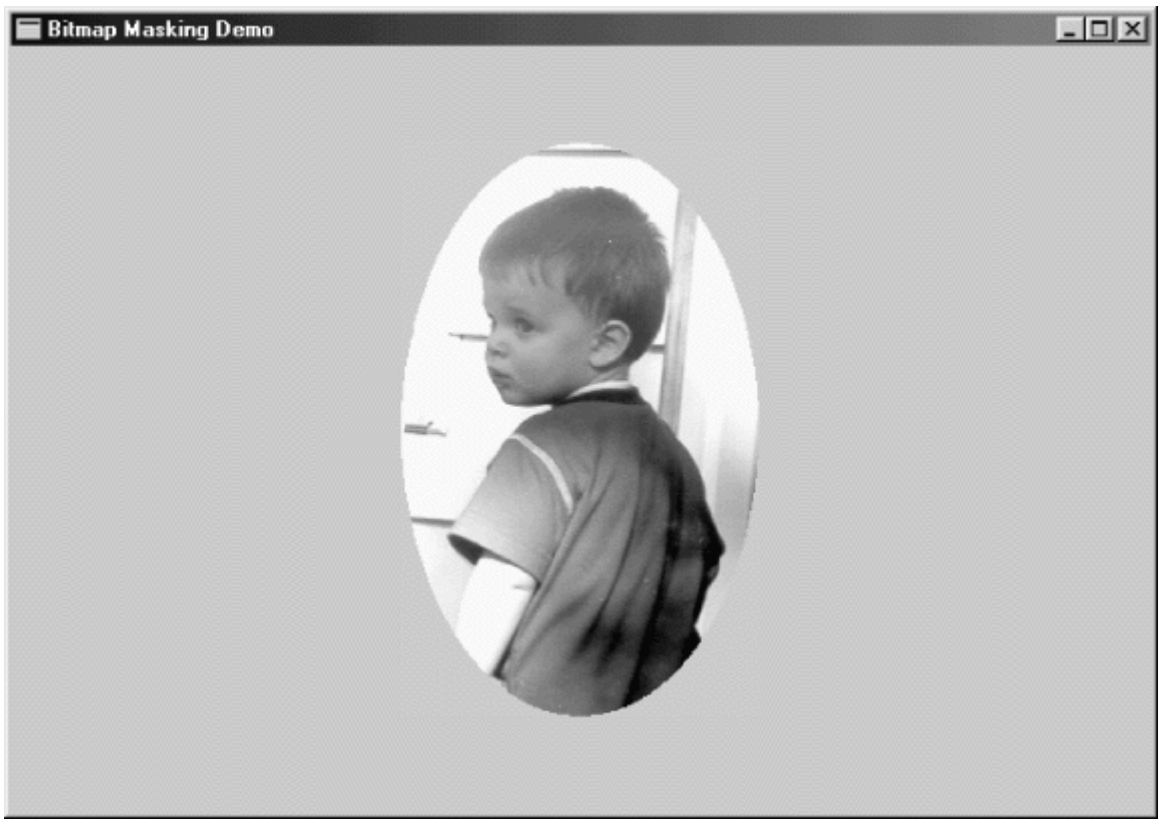


图 14-9 BITMASK 的萤幕显示

如果正在为 Windows NT 编写类似的应用程式，那么您可以使用与 MASKBIT 程式类似的 MaskBlt 函式，而只需要更少的函式呼叫。Windows NT 还包括另一个类似 BitBlt 的函式，Windows 98 不支援该函式。此函式是 PlgBlt（「平行四边形位元块移动：parallelogram blt」）。这个函式可以对图像进行旋转或者倾斜点阵图图像。

最後，如果在您的机器上执行 BITMASK 程式，您就只会看见黑色、白色和两个灰色的阴影，这是因为您执行的显示模式是 16 色或 256 色。对於 16 色模式，显示效果无法改进，但在 256 色模式下可以改变调色盘以显示灰阶。您将在第十六章学会如何设定调色盘。

简单的动画

小张的点阵图显示起来非常快，因此可以将点阵图和 Windows 计时器联合使用，来完成一些基本的动画。

现在开始这个弹球程式。

BOUNCE 程式，如程式 14-10 所示，产生了一个在视窗显示区域弹来弹去的小球。该程式利用计时器来控制小球的行进速度。小球本身是一幅点阵图，程式首先通过建立点阵图来建立小球，将其选进记忆体装置内容，然後呼叫一些简单的 GDI 函式。程式用 BitBlt 从一个记忆体装置内容将这个点阵图小球画到显示器上。

程式 14-10 BOUNCE

```
BOUNCE.C
/*------------------------------------------------------------------------
-
      BOUNCE.C --      Bouncing Ball Program
                                              (c) Charles Petzold, 1998
------------------------------------------------------------------------
*/

#include <windows.h>
#define ID_TIMER    1

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                                    PSTR    szCmdLine,    int
iCmdShow)
{
      static TCHAR szAppName[] = TEXT ("Bounce") ;
      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 ("This program requires Windows
NT!"),
                                                          szAppName,
MB_ICONERROR) ;
                  return 0 ;
      }

      hwnd = CreateWindow (  szAppName, TEXT ("Bouncing Ball"),
                             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 iMsg, WPARAM wParam, LPARAM lParam)
{
     static HBITMAP   hBitmap ;
     static int             cxClient, cyClient, xCenter, yCenter, cxTotal,
cyTotal,
               cxRadius,             cyRadius,  cxMove,  cyMove,  xPixel,
yPixel ;
     HBRUSH                         hBrush ;
     HDC                            hdc, hdcMem ;
     int                            iScale ;

     switch (iMsg)
     {
     case  WM_CREATE:
                    hdc = GetDC (hwnd) ;
                    xPixel = GetDeviceCaps (hdc, ASPECTX) ;
                    yPixel = GetDeviceCaps (hdc, ASPECTY) ;
                    ReleaseDC (hwnd, hdc) ;

                    SetTimer (hwnd, ID_TIMER, 50, NULL) ;
                    return 0 ;

     case  WM_SIZE:
                    xCenter = (cxClient = LOWORD (lParam)) / 2 ;
                    yCenter = (cyClient = HIWORD (lParam)) / 2 ;

                    iScale = min (cxClient * xPixel, cyClient * yPixel) / 16 ;

                    cxRadius = iScale / xPixel ;
                    cyRadius = iScale / yPixel ;

                    cxMove = max (1, cxRadius / 2) ;
                    cyMove = max (1, cyRadius / 2) ;

                    cxTotal = 2 * (cxRadius + cxMove) ;
                    cyTotal = 2 * (cyRadius + cyMove) ;

                    if (hBitmap)
```

```
                                DeleteObject (hBitmap) ;
                        hdc = GetDC (hwnd) ;
                        hdcMem = CreateCompatibleDC (hdc) ;
                        hBitmap = CreateCompatibleBitmap (hdc, cxTotal, cyTotal) ;
                        ReleaseDC (hwnd, hdc) ;

                        SelectObject (hdcMem, hBitmap) ;
                        Rectangle (hdcMem, -1, -1, cxTotal + 1, cyTotal + 1) ;

                        hBrush = CreateHatchBrush (HS_DIAGCROSS, 0L) ;
                        SelectObject (hdcMem, hBrush) ;
                        SetBkColor (hdcMem, RGB (255, 0, 255)) ;
                        Ellipse (hdcMem, cxMove, cyMove, cxTotal - cxMove, cyTotal
- cyMove) ;
                        DeleteDC (hdcMem) ;
                        DeleteObject (hBrush) ;
                        return 0 ;

      case  WM_TIMER:
                        if (!hBitmap)
                                DeleteObject (hBitmap) ;

                        hdc = GetDC (hwnd) ;
                        hdcMem = CreateCompatibleDC (hdc) ;
                        SelectObject (hdcMem, hBitmap) ;

                        BitBlt (hdc, xCenter - cxTotal / 2,
                                yCenter - cyTotal / 2, cxTotal,
cyTotal,
                                hdcMem, 0, 0, SRCCOPY) ;

                        ReleaseDC (hwnd, hdc) ;
                        DeleteDC (hdcMem) ;

                        xCenter += cxMove ;
                        yCenter += cyMove ;

                        if ((xCenter + cxRadius >= cxClient) || (xCenter - cxRadius
<= 0))
                                cxMove = -cxMove ;

                        if ((yCenter + cyRadius >= cyClient) || (yCenter - cyRadius
<= 0))
                                cyMove = -cyMove ;

                        return 0 ;

      case  WM_DESTROY:
```

```
                        if (hBitmap)
                                        DeleteObject (hBitmap) ;

                        KillTimer (hwnd, ID_TIMER) ;
                        PostQuitMessage (0) ;
                        return 0 ;
    }
      return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

BOUNCE 每次收到一个 WM_SIZE 讯息时都重画小球。这就需要与视讯显示器相容的记忆体装置内容：

```
hdcMem = CreateCompatibleDC (hdc) ;
```

小球的直径设为视窗显示区域高度或宽度中较短者的十六分之一。不过，程式构造的点阵图却比小球大：从点阵图中心到点阵图四个边的距离是小球半径的 1.5 倍：

```
hBitmap = CreateCompatibleBitmap (hdc, cxTotal, cyTotal) ;
```

将点阵图选进记忆体装置内容後，整个点阵图背景设成白色：

```
Rectangle (hdcMem, -1, -1, xTotal + 1, yTotal + 1) ;
```

那些不固定的座标使矩形边框在点阵图之外著色。一个对角线开口的画刷选进记忆体装置内容，并将小球画在点阵图的中央：

```
Ellipse (hdcMem, xMove, yMove, xTotal - xMove, yTotal - yMove) ;
```

当小球移动时，小球边界的空白会有效地删除前一时刻的小球图像。在另一个位置重画小球只需在 BitBlt 呼叫中使用 SRCCOPY 的 ROP 代码：

```
BitBlt (hdc, xCenter - cxTotal / 2, yCenter - cyTotal / 2, cxTotal, cyTotal,
                hdcMem, 0, 0, SRCCOPY) ;
```

BOUNCE 程式只是展示了在显示器上移动图像的最简单的方法。在一般情况下，这种方法并不能令人满意。如果您对动画感兴趣，那么除了在来源和目的之间执行或操作以外，您还应该研究其他的 ROP 代码（例如 SRCINVERT）。其他动画技术包括 Windows 调色盘(以及 AnimatePalette 函式)和 CreateDIBSection 函式。对於更高级的动画您只好放弃 GDI 而使用 DirectX 介面了。

视窗外的点阵图

SCRAMBLE 程式，如程式 14-11 所示，编写非常粗糙，我本来不应该展示这个程式，但它示范了一些有趣的技术，而且在交换两个显示矩形内容的 BitBlt 操作的程序中，用记忆体装置内容作为临时储存空间。

程式 14-11　SCRAMBLE

```
SCRAMBLE.C
/*------------------------------------------------------------------------
-
```

```
   SCRAMBLE.C --  Scramble (and Unscramble) Screen
                                                (c) Charles Petzold, 1998
------------------------------------------------------------------------
-*/

#include <windows.h>

#define NUM 300

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

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                                    PSTR    szCmdLine,    int
iCmdShow)
{
      static int       iKeep [NUM][4] ;
      HDC                        hdcScr, hdcMem ;
      int                        cx, cy ;
      HBITMAP                hBitmap ;
      HWND                       hwnd ;
      int                        i, j, x1, y1, x2, y2 ;

      if (LockWindowUpdate (hwnd = GetDesktopWindow ()))
      {
                  hdcScr      =    GetDCEx    (hwnd,    NULL,    DCX_CACHE    |
DCX_LOCKWINDOWUPDATE) ;
                  hdcMem      = CreateCompatibleDC (hdcScr) ;
                  cx     = GetSystemMetrics (SM_CXSCREEN) / 10 ;
                  cy     = GetSystemMetrics (SM_CYSCREEN) / 10 ;
                  hBitmap = CreateCompatibleBitmap (hdcScr, cx, cy) ;

                  SelectObject (hdcMem, hBitmap) ;
            srand ((int) GetCurrentTime ()) ;

                  for (i = 0 ; i < 2   ; i++)
                  for (j = 0 ; j < NUM ; j++)
                  {
                              if (i == 0)
                              {
                              iKeep [j] [0] = x1 = cx * (rand () % 10) ;
                              iKeep [j] [1] = y1 = cy * (rand () % 10) ;
                              iKeep [j] [2] = x2 = cx * (rand () % 10) ;
                              iKeep [j] [3] = y2 = cy * (rand () % 10) ;
                              }
                              else
                        {
                                    x1 = iKeep [NUM - 1 - j] [0] ;
                                    y1 = iKeep [NUM - 1 - j] [1] ;
```

```
                                    x2 = iKeep [NUM - 1 - j] [2] ;
                                    y2 = iKeep [NUM - 1 - j] [3] ;
                               }
                               BitBlt (hdcMem,  0,  0, cx, cy, hdcScr, x1, y1,
SRCCOPY) ;
                               BitBlt (hdcScr, x1, y1, cx, cy, hdcScr, x2, y2,
SRCCOPY) ;
                               BitBlt (hdcScr, x2, y2, cx, cy, hdcMem,  0,  0,
SRCCOPY) ;

                          Sleep (10) ;
                    }

                    DeleteDC (hdcMem) ;
                    ReleaseDC (hwnd, hdcScr) ;
                    DeleteObject (hBitmap) ;

                    LockWindowUpdate (NULL) ;
          }
          return FALSE ;
}
```

SCRAMBLE 没有视窗讯息处理程式。在 WinMain 中，它首先呼叫带有桌面视窗代号的 LockWindowUpdate。此函式暂时防止其他程式更新萤幕。然後 SCRAMBLE 通过呼叫带有参数 DCX_LOCKWINDOWUPDATE 的 GetDCEx 来获得整个萤幕的装置内容。这样就只有 SCRAMBLE 可以更新萤幕了。

然後 SCRAMBLE 确定全萤幕的尺寸，并将长宽分别除以 10。程式用这个尺寸（名称是 cx 和 cy）来建立一个点阵图，并将该点阵图选进记忆体装置内容。

使用 C 语言的 rand 函式，SCRAMBLE 计算出四个随机值（两个座标点）作为 cx 和 cy 的倍数。程式透过三次呼叫 BitBlt 函式来交换两个矩形块中显示的内容。第一次将从第一个座标点开始的矩形复制到记忆体装置内容。第二次 BitBlt 将从第二座标点开始的矩形复制到第一点开始的位置。第三次将记忆体装置内容中的矩形复制到第二个座标点开始的区域。

此程序将有效地交换显示器上两个矩形中的内容。SCRAMBLE 执行 300 次交换，这时的萤幕显示肯定是一团糟。但不用担心，因为 SCRAMBLE 记得是怎么把显示弄得这样一团糟的，接著在退出前它会按相反的次序恢复原来的桌面显示（锁定萤幕前的画面）！

您也可以用记忆体装置内容将一个点阵图复制给另一个点阵图。例如，假定您要建立一个点阵图，该点阵图只包含另一个点阵图左上角的图形。如果原来的图像代号为 hBitmap，那么您可以将其尺寸复制到一个 BITMAP 型态的结构中：

```
GetObject (hBitmap, sizeof (BITMAP), &bm) ;
```

然後建立一个未初始化的新点阵图，该点阵图的尺寸是原来图的 1/4：

```
hBitmap2 = CreateBitmap (    bm.bmWidth / 2, bm.bmHeight / 2,
                         bm.bmPlanes, bm.bmBitsPixel, NULL) ;
```

现在建立两个记忆体装置内容，并将原来点阵图和新点阵图选分别进这两个记忆体装置内容：

```
hdcMem1 = CreateCompatibleDC (hdc) ;
hdcMem2 = CreateCompatibleDC (hdc) ;

SelectObject (hdcMem1, hBitmap) ;
SelectObject (hdcMem2, hBitmap2) ;
```

最後，将第一个点阵图的左上角复制给第二个：

```
BitBlt (    hdcMem2, 0, 0, bm.bmWidth / 2, bm.bmHeight / 2,
                         hdcMem1, 0, 0, SRCCOPY) ;
```

剩下的只是清除工作：

```
DeleteDC (hdcMem1) ;
DeleteDC (hdcMem2) ;
DeleteObject (hBitmap) ;
```

BLOWUP.C 程式，如图 14-21 所示，也用视窗更新锁定来在程式视窗之外显示一个捕捉的矩形。此程式允许使用者用滑鼠圈选萤幕上的矩形区域，然後 BLOWUP 将该区域的内容复制到点阵图。在 WM_PAINT 讯息处理期间，点阵图复制到程式的显示区域，必要时将拉伸或压缩。（参见程式 14-12。）

程式 14-12　BLOWUP

```
BLOWUP.C
/*-----------------------------------------------------------------------------
      BLOWUP.C --      Video Magnifier Program
                                              (c) Charles Petzold, 1998
-----------------------------------------------------------------------------
*/

#include <windows.h>
#include <stdlib.h>                                   // for abs definition
#include "resource.h"
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                                      PSTR    szCmdLine,    int
iCmdShow)
{
      static TCHAR           szAppName [] = TEXT ("Blowup") ;
      HACCEL                              hAccel ;
      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       = szAppName ;
     wndclass.lpszClassName      = szAppName ;

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

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

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

     hAccel = LoadAccelerators (hInstance, szAppName) ;
     while (GetMessage (&msg, NULL, 0, 0))
     {
               if (!TranslateAccelerator (hwnd, hAccel, &msg))
               {
                              TranslateMessage (&msg) ;
                              DispatchMessage (&msg) ;
               }
     }
     return msg.wParam ;
}

void InvertBlock (HWND hwndScr, HWND hwnd, POINT ptBeg, POINT ptEnd)
{
     HDC hdc ;
     hdc = GetDCEx (hwndScr, NULL, DCX_CACHE | DCX_LOCKWINDOWUPDATE) ;
     ClientToScreen (hwnd, &ptBeg) ;
     ClientToScreen (hwnd, &ptEnd) ;
     PatBlt (hdc, ptBeg.x, ptBeg.y, ptEnd.x - ptBeg.x, ptEnd.y - ptBeg.y,
```

```
                                    DSTINVERT) ;
      ReleaseDC (hwndScr, hdc) ;
}

HBITMAP CopyBitmap (HBITMAP hBitmapSrc)
{
      BITMAP           bitmap ;
      HBITMAP          hBitmapDst ;
      HDC                    hdcSrc, hdcDst ;

      GetObject (hBitmapSrc, sizeof (BITMAP), &bitmap) ;
      hBitmapDst = CreateBitmapIndirect (&bitmap) ;

      hdcSrc = CreateCompatibleDC (NULL) ;
      hdcDst = CreateCompatibleDC (NULL) ;

      SelectObject (hdcSrc, hBitmapSrc) ;
      SelectObject (hdcDst, hBitmapDst) ;

      BitBlt (hdcDst, 0, 0, bitmap.bmWidth, bitmap.bmHeight,
                            hdcSrc, 0, 0, SRCCOPY) ;
      DeleteDC (hdcSrc) ;
      DeleteDC (hdcDst) ;

      return hBitmapDst ;
}

LRESULT CALLBACK WndProc (   HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
      static BOOL            bCapturing, bBlocking ;
      static HBITMAP       hBitmap ;
      static HWND            hwndScr ;
      static POINT           ptBeg, ptEnd ;
      BITMAP                 bm ;
      HBITMAP                hBitmapClip ;
      HDC                    hdc, hdcMem ;
      int                    iEnable ;
      PAINTSTRUCT            ps ;
      RECT                   rect ;

      switch (message)
      {
      case  WM_LBUTTONDOWN:
                        if (!bCapturing)
                        {
                                    if     (LockWindowUpdate     (hwndScr     =
GetDesktopWindow ()))
```

```
                                    {
                                          bCapturing = TRUE ;
                                          SetCapture (hwnd) ;
                                          SetCursor     (LoadCursor     (NULL,
IDC_CROSS)) ;
                                    }
                                    else
                                          MessageBeep (0) ;
                              }
                              return 0 ;

      case  WM_RBUTTONDOWN:
                        if (bCapturing)
                        {
                                    bBlocking = TRUE ;
                                    ptBeg.x = LOWORD (lParam) ;
                                    ptBeg.y = HIWORD (lParam) ;
                                    ptEnd = ptBeg ;
                                    InvertBlock (hwndScr, hwnd, ptBeg, ptEnd) ;
                        }
                        return 0 ;

      case  WM_MOUSEMOVE:
                        if (bBlocking)
                        {
                                    InvertBlock (hwndScr, hwnd, ptBeg, ptEnd) ;
                                    ptEnd.x = LOWORD (lParam) ;
                                    ptEnd.y = HIWORD (lParam) ;
                                    InvertBlock (hwndScr, hwnd, ptBeg, ptEnd) ;
                        }
                        return 0 ;

      case  WM_LBUTTONUP:
      case  WM_RBUTTONUP:
                        if (bBlocking)
                        {
                                    InvertBlock  (hwndScr,  hwnd,  ptBeg,
ptEnd) ;
                                    ptEnd.x = LOWORD (lParam) ;
                                    ptEnd.y = HIWORD (lParam) ;

                                    if (hBitmap)
                                    {
                                          DeleteObject (hBitmap) ;
                                          hBitmap = NULL ;
                                    }

                                    hdc = GetDC (hwnd) ;
```

```
                                        hdcMem      =      CreateCompatibleDC
(hdc) ;
                                        hBitmap     =  CreateCompatibleBitmap
(hdc,
                           abs (ptEnd.x - ptBeg.x),
                           abs (ptEnd.y - ptBeg.y)) ;

                      SelectObject (hdcMem, hBitmap) ;

                      StretchBlt (hdcMem, 0, 0,          abs    (ptEnd.x    -
ptBeg.x),
                         abs (ptEnd.y - ptBeg.y),
                hdc, ptBeg.x,     ptBeg.y, ptEnd.x - ptBeg.x,
                             ptEnd.y - ptBeg.y, SRCCOPY) ;
                                        DeleteDC (hdcMem) ;
                                        ReleaseDC (hwnd, hdc) ;
                                        InvalidateRect (hwnd, NULL, TRUE) ;
                      }
                      if (bBlocking || bCapturing)
                      {
                                 bBlocking = bCapturing = FALSE ;
                                 SetCursor (LoadCursor (NULL, IDC_ARROW)) ;
                                 ReleaseCapture () ;
                                 LockWindowUpdate (NULL) ;
                      }
                      return 0 ;

      case  WM_INITMENUPOPUP:
                      iEnable =  IsClipboardFormatAvailable (CF_BITMAP) ?
                MF_ENABLED : MF_GRAYED ;

                      EnableMenuItem   ((HMENU)   wParam,   IDM_EDIT_PASTE,
iEnable) ;

                      iEnable = hBitmap ? MF_ENABLED : MF_GRAYED ;

                      EnableMenuItem    ((HMENU)    wParam,    IDM_EDIT_CUT,
iEnable) ;
                      EnableMenuItem    ((HMENU)    wParam,    IDM_EDIT_COPY,
iEnable) ;
                      EnableMenuItem   ((HMENU)   wParam,   IDM_EDIT_DELETE,
iEnable) ;
                      return 0 ;

      case  WM_COMMAND:
                      switch (LOWORD (wParam))
                      {
                 case IDM_EDIT_CUT:
```

```
                    case IDM_EDIT_COPY:
                                    if (hBitmap)
                                    {
                              hBitmapClip = CopyBitmap (hBitmap) ;
                              OpenClipboard (hwnd) ;
                              EmptyClipboard () ;
                    SetClipboardData (CF_BITMAP, hBitmapClip) ;
                                    }
                                    if (LOWORD (wParam) == IDM_EDIT_COPY)
                                                return 0 ;
                    //fall through for IDM_EDIT_CUT
                    case  IDM_EDIT_DELETE:
                                    if (hBitmap)
                                    {
                                          DeleteObject (hBitmap) ;
                                          hBitmap = NULL ;
                                    }
                                          InvalidateRect   (hwnd,   NULL,
TRUE) ;
                                          return 0 ;

                    case  IDM_EDIT_PASTE:
                                          if (hBitmap)
                                          {
                                    DeleteObject (hBitmap) ;
                                    hBitmap = NULL ;
                              }
                              OpenClipboard (hwnd) ;
                              hBitmapClip = GetClipboardData (CF_BITMAP) ;

                              if (hBitmapClip)
                                    hBitmap = CopyBitmap (hBitmapClip) ;

                              CloseClipboard () ;
                              InvalidateRect (hwnd, NULL, TRUE) ;
                              return 0 ;
               }
               break ;
     case  WM_PAINT:
                    hdc = BeginPaint (hwnd, &ps) ;

                    if (hBitmap)
                    {
                              GetClientRect (hwnd, &rect) ;

                              hdcMem = CreateCompatibleDC (hdc) ;
                              SelectObject (hdcMem, hBitmap) ;
                              GetObject (hBitmap, sizeof (BITMAP), (PSTR)
```

```
&bm) ;
                                    SetStretchBltMode (hdc, COLORONCOLOR) ;

                                    StretchBlt  (hdc,       0,  0,  rect.right,
rect.bottom,
                  hdcMem, 0, 0, bm.bmWidth, bm.bmHeight, SRCCOPY) ;

                                    DeleteDC (hdcMem) ;
                         }
                         EndPaint (hwnd, &ps) ;
                         return 0 ;

      case  WM_DESTROY:
                         if (hBitmap)
                                         DeleteObject (hBitmap) ;

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

BLOWUP.RC （摘录）

```
//Microsoft Developer Studio generated resource script.
#include "resource.h"
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////
/
// Menu
BLOWUP MENU DISCARDABLE
BEGIN
      POPUP "&Edit"
      BEGIN
       MENUITEM "Cu&t\tCtrl+X",          IDM_EDIT_CUT
       MENUITEM "&Copy\tCtrl+C",         IDM_EDIT_COPY
       MENUITEM "&Paste\tCtrl+V",              IDM_EDIT_PASTE
       MENUITEM "De&lete\tDelete",             IDM_EDIT_DELETE
      END
END

////////////////////////////////////////////////////////////////////////////
/
// Accelerator
BLOWUP ACCELERATORS DISCARDABLE
BEGIN
      "C",      IDM_EDIT_COPY,    VIRTKEY, CONTROL, NOINVERT
      "V",      IDM_EDIT_PASTE,   VIRTKEY, CONTROL, NOINVERT
      VK_DELETE,           IDM_EDIT_DELETE,   VIRTKEY, NOINVERT
```

```
    "X",     IDM_EDIT_CUT,     VIRTKEY, CONTROL, NOINVERT
END
```

RESOURCE.H （摘录）

```
// Microsoft Developer Studio generated include file.
// Used by Blowup.rc

#define IDM_EDIT_CUT        40001
#define IDM_EDIT_COPY       40002
#define IDM_EDIT_PASTE      40003
#define IDM_EDIT_DELETE     40004
```

图 14-10 BLOWUP 显示的一个范例

由於滑鼠拦截的限制，所以开始使用 BLOWUP 时会有些困难，需要逐渐适应。下面是使用本程式的方法：

在 BLOWUP 显示区域按下滑鼠左键不放，滑鼠指标会变成「+」字型。

继续按住左键，将滑鼠移到萤幕上的任何其他位置。滑鼠游标的位置就是您要圈选的矩形区域的左上角。

继续按住左键，按下滑鼠右键，然後拖动滑鼠到您要圈选的矩形区域的右下角。释放滑鼠左键和右键。（释放滑鼠左、右键次序无关紧要。）

滑鼠游标恢复成箭头状，这时您圈选的矩形区域已复制到了 BLOWUP 的显示区域，并作了适当的压缩或拉伸变化。

如果您从右上角到左下角选取的话，BLOWUP 将显示矩形区域的镜像。如果从左下到右上角选取，BLOWUP 将显示颠倒的图像。如果从右上角至左上角选取，程式将综合两种效果。

BLOWUP 还包含将点阵图复制到剪贴簿，以及将剪贴簿中的点阵图复制到程式的处理功能。BLOWUP 处理 WM_INITMENUPOPUP 讯息来启用或禁用「Edit」功能表中的不同选项，并通过 WM_COMMAND 讯息来处理这些功能表项。您应该对这些程式码的结构比较熟悉，因为它们与第十二章中的复制和粘贴文字项目的处理方式在本质上是一样的。

不过，对於点阵图，剪贴簿物件不是整体代号而是点阵图代号。当您使用 CF_BITMAP 时， GetClipboardData 函式传回一个 HBITMAP 物件，而且 SetClipboardData 函式接收一个 HBITMAP 物件。如果您想将点阵图传送给剪贴簿又想保留副本以供程式本身使用，那么您必须复制点阵图。同样，如果您从剪贴簿上粘贴了一幅点阵图，也应该做一个副本。BLOWUP 中的 CopyBitmap 函式是通过取得现存点阵图的 BITMAP 结构，并在 CreateBitmapIndirect 函式中用这个结构建立一个新点阵图来完成此项操作的。(变数名的尾码 Src 和 Dst 分别代表「来源」和「目的」。）两个点阵图都被选进记忆体装置内容，而且通过呼叫 BitBlt 来复制点阵图内容。（另一种复制位元的方法，可以先按点阵图大小配置一块记忆体，然後为来源点阵图呼叫 GetBitmapBits，为目的点阵图呼叫 SetBitmapBits。）

我发现 BLOWUP 对於检查 Windows 及其应用程式中大量分散的小点阵图和图片非常有用。