第五章　图形基础

图形装置介面（GDI：Graphics Device Interface）是 Windows 的子系统，它负责在视讯显示器和印表机上显示图形。正如您所认为的那样，GDI 是 Windows 非常重要的部分。不只您为 Windows 编写的应用系统在显示视觉资讯时使用 GDI，就连 Windows 本身也使用 GDI 来显示使用者介面物件，诸如功能表、卷动列、图示和滑鼠游标。

不幸的是，如果要对 GDI 进行全面的讲述，将需要一整本书——当然不是这本书。在本章中，我只是想向您提供画线和填入区域的基本知识，这对於理解下面几章的 GDI 已经足够了。在後面几章中会讲述 GDI 支援的点阵图、metafile 以及格式化文字。

GDI 的结构

从程式写作者的观点来看，GDI 由几百个函式呼叫和一些相关的资料型态、巨集和结构组成。但是在开始讲述这些函式的细节之前，让我们先从巨观上了解一下 GDI 的整体结构。

GDI 原理

Windows 98 和 Microsoft Windows NT 中的图形主要由 GDI32.DLL 动态连结程式库输出的函式来处理。在 Windows 98 中，这个 GDI32.DLL 实际是利用 16 位元 GDI.EXE 动态连结程式库来执行许多函式。在 Windows NT 中，GDI.EXE 只用於 16 位元的程式。

这些动态连结程式库呼叫您安装的视讯显示器和任何印表机呼叫驱动程式中的常式。视讯驱动程式存取视讯显示器的硬体，印表机驱动程式将 GDI 命令转换为各种印表机能够理解的代码或者命令。显然，不同的视讯显示卡和印表机要求不同的装置驱动程式。

因为 PC 相容机种上可以连接许多种不同的视讯设备，所以，GDI 的主要目的之一是支援与装置无关的图形。Windows 程式应该能够毫无困难地在 Windows 支援的任意一种图形输出设备上执行，GDI 通过将您的程式和不同输出设备的特性隔离开来的方法来达到这一目的。

图形输出设备分为两大类：位元映射设备和向量设备。大多数 PC 的输出设备是位元映射设备，这意味著它们以图点构成的阵列来表示图像，这类设备包括视讯显示卡、点阵印表机和雷射印表机。向量设备使用线来绘制图像，通常

局限於绘图机。

许多传统的电脑图形程式设计方式都是完全以向量为主的，这意味著使用向量图形系统的程式与硬体有著一定层次的隔离。输出设备用图素表示图形，但是程式与程式介面之间并不是用图素进行沟通的。您当然可以使用 Windows GDI 作为一个高阶的向量绘制系统，同时也可以将它用於比较低阶的图素操作。

从这方面来看，Windows GDI 和传统的图形介面语言之间的关系，就如同 C 和其他程式设计语言之间的关系一样。C 以它在不同作业系统和环境之间的高度可携性而闻名，然而 C 也以允许程式写作者进行低阶系统呼叫而闻名，这些呼叫在其他高阶语言中通常是不可能的。正如 C 有时被认为是一种「高级组合语言」一样，您可以认为 GDI 是图形设备硬体之间的一种高阶介面。

您已经看到，Windows 内定使用图素座标系统。大多数传统的图形语言使用「虚拟」座标系，其水平和垂直轴的范围在 0 到 32,767 之间。虽然有些图形语言不让您使用图素座标，但是 Windows GDI 允许您使用两种座标系统之一（甚至依据实际度量衡的座标系）。您可以使用虚拟座标系以便让程式独立於硬体之外，或者也可以使用设备座标系而完全迎合硬体设备提供的环境。

某些程式写作者认为一旦开始使用操作图素的程式设计方式，就放弃了装置无关性。我们在上一章看到，这不完全是正确的，其中的诀窍是在与装置无关的方式中使用图素。这要求图形介面语言为程式提供一些方法来确定设备的硬体特徵，并进行适当的调节。例如，在 SYSMETS 程式中，我们根据标准系统字体字元的图素大小来确定萤幕上的文字间距，这种方法允许程式针对解析度、文字大小和方向比例各不相同的显示卡进行相应的调节。您将在本章看到一些用於确定显示尺寸的其他方法。

早期，许多使用者在单色显示器上执行 Windows。即使是几年前，笔记本电脑也还只有灰阶显示。为此，GDI 的设计保证了您可以在编写一个程式时不必太担心色彩问题——也就是说，Windows 可以将色彩转换为灰阶显示。甚至在今天，Windows 98 使用的视讯显示已经具有了不同的色彩能力（16 色、256 色、「high-Color」以及「true-color」）。虽然，彩色喷墨印表机的成本已经很低了，但是大多数使用者仍然坚持使用黑白印表机。盲目地使用这些设备是可以的，但是您的程式也应该能决定在某种显示设备上有多少色彩可以使用，从而最佳利用硬体功能。

当然，就如同您编写 C 程式时，为了使它在其他电脑上执行而遇到一些微妙的移植性问题一样，您也可能不小心让装置依赖性溜进您的 Windows 程式，这就是不与硬体完全隔离的代价。您还应该知道 Windows GDI 的局限。虽然可以在显示器上到处移动图形物件，但 GDI 通常是一个静态的显示系统，只有有

限的动画支援。如果需要为游戏编写复杂的动画，就应该研究一下 Microsoft DirectX，它提供了您需要的支援。

GDI 函式呼叫

组成 GDI 的几百个函式呼叫可以分为几大类：

取得（或者建立）和释放（或者清除）装置内容的函式 我们在前面的章节中已经看到过，您在绘图时需要装置内容代号。GetDC 和 RealseDC 函式让您在非 WM_PAINT 的讯息处理期间来做到这一点，而 BeginPaint 和 EndPaint 函式（虽然在技术上它们是 USER 模组而不是 GDI 模组的一部分）在进行绘图的 WM_PAINT 讯息处理期间使用。我们马上还会介绍有关装置内容的其他一些函式。

取得有关装置内容资讯的函式 再以第四章中 SYSMETS 程式为例，我们使用 GetTextMetrics 函式来取得有关装置内容中目前所选字体的尺寸资讯。在本章後面，我们将看到一个取得非常广泛的装置内容资讯的>DEVCAPS1 程式。

绘图函式 显然，在所有前提条件都得以满足之後，这些函式是真正重要的部分。在上一章中，我们使用 TextOut 函式在视窗的显示区域显示一些文字。我们将看到，其他 GDI 函式还可以让您画线、填入区域。在第十四章和第十五章还会看到如何建立点阵图图像。

设定和取得装置内容参数的函式 装置内容的「属性」决定有关绘图函式如何工作的细节。例如，用 SetTextColor 来指定 TextOut（或者其他文字输出函式）所绘制的文字色彩。在第四章中 SYSMETS 程式中，我们使用 SetTextAlign 来告诉 GDI：TextOut 函式中的字串的开始位置应该在字串的右边而不是内定的左边。装置内容的所有属性都有预设值，取得装置内容时这些预设值就设定好了。对於所有的 Set 函式，都有相应的 Get 函式，以允许您取得目前装置内容属性。

使用 GDI 物件的函式 GDI 在这里变得有点混乱。首先举一个例子：内定时使用 GDI 绘制的所有直线都是实线并具有一个标准的宽度。您可能希望绘制更细的直线，或者是由一系列的点或短划线组成的直线。这种线的宽度和这种线的画笔样式不是装置内容的属性，而是一个「逻辑画笔」的特徵。您可以通过在 CreatePen、 CreatePenIndirect 或 ExtCreatePen 函式中指定这些特徵来建立一个逻辑画笔，这些函式传回一个逻辑画笔的代号（虽然这些函式被认为是 GDI 的一部分，但是和大多数 GDI 函式呼叫不一样，它们不要求装置内容的代号）。要使用这个画笔，就要将画笔代号选进装置内容。我们认为，装置内容中目前选中的画笔就是装置内容的一个属性。这样，您画任何线都使用这个画笔，然後，您可以取消装置内容中的画笔选择，并清除画笔物件。清除画笔物件是必

要的，因为画笔定义占用了分配的记忆体空间。除了画笔以外，GDI 物件还用於建立填入封闭区域的画刷、字体、点阵图以及 GDI 的其他一些方面。

GDI 基本图形

您在萤幕或印表机上显示的图形型态本身可以被分为几类，通常被称为「基本图形」，它们是：

直线和曲线 线条是所有向量图形绘制系统的基础。GDI 支援直线、矩形、椭圆（包括椭圆的子集，也就是我们所说的「圆」）、椭圆圆周上的部分曲线即所谓的「弧」以及贝塞尔曲线（Bezier spline），我们将在本章中分别对它们进行介绍。所有更复杂的曲线可由折线（polyline）代替，折线通过一组非常短的直线来定义一条曲线。线条用装置内容中选中的目前画笔绘制。

填入区域 当一系列直线或者曲线封闭了一个区域时，该区域可以使用目前 GDI 画刷物件进行填图。这个画刷可以是实心色彩、图案（可以是一系列的水平、垂直或者对角标记）或者是在区域内垂直或者水平重复的点阵图图像。

点阵图 点阵图是位元的矩形阵列，这些位元对应於显示设备上的图素，它们是位元映射图形的基础工具。点阵图通常用於在视讯显示器或者印表机上显示复杂（一般都是真实的）图像。点阵图还可以用於显示必须快速绘制的小图像（诸如图示、滑鼠游标以及在应用工具条中出现的按钮等）。GDI 支援两种型态的点阵图——旧式的（虽然还非常有用）「装置相关」点阵图，是 GDI 物件；和新的（如 Windows 3.0 的）「装置无关」点阵图（或者 DIB），可以储存在磁片档案中。第十四章和第十五章讨论点阵图。

文字 文字的数学味道不像电脑图形的其他方面那样浓。文字和几百年的传统印刷术有关，它被许多印刷工人看作为一门艺术。因此，文字通常不仅是所有的电脑图形系统中最复杂的部分，而且（如果识字还是社会基本要求的话）也是最重要的部分。用於定义 GDI 字体物件和取得字体资讯的资料结构是 Windows 中最庞大的部分之一。从 Windows 3.1 开始，GDI 开始支援 TrueType 字体，该字体是在填入轮廓线基础上建立的，这样的填入轮廓线可由其他 GDI 函式处理。依据相容性和储存大小的考虑，Windows 98 继续支援旧式的点阵字体。我会在第十七章讨论字体。

其他部分

GDI 的其他部分无法这么容易地分类，它们是：

映射模式和变换 虽然内定以图素为单位进行绘图，但是您并非局限於此。GDI 映射模式允许您以英寸（或者甚至以几分之一英寸）、毫米或者任何您想使

用的单位来绘图（Windows NT 还支援传统的以三乘三矩阵表示的「座标变换」，这允许倾斜和旋转图形物件。不幸的是，在 Windows 98 中不支援座标变换）。

Metafile Metafile 是以二进位形式储存的 GDI 命令集合。Metafile 主要用於通过剪贴板传输向量图形,第十八章会讨论 metafile。

绘图区域 绘图区域是形状任意的复杂区域,通常定义为较简单的绘图区域组合。在 GDI 内部，绘图区域除了储存为最初用来定义绘图区域的线条组合以外，还以一系列扫描线的形式储存。您可以将绘图区域用於绘制轮廓、填入图形和剪裁。

路径 路径是 GDI 内部储存的直线和曲线的集合。路径可以用於绘图、填入图形和剪裁，还可以转换为绘图区域。

剪裁 绘图可以限制在显示区域的某一部分中，这就是所谓的剪裁。剪裁区域是不是矩形都可以，剪裁通常是通过区域或者路径来定义的。

调色盘 自订调色盘通常限於显示 256 色的显示器。Windows 仅保留这些色彩之中的 20 种以供系统使用，您可以改变其他 236 种色彩，以准确显示按点阵图形式储存的真实图像。第十六章会讨论调色盘。

列印 虽然本章限於讨论视讯显示，但是您在本章中所学到的全部知识都适用於列印。第十三章会讨论列印。

装置内容

在开始绘图之前，让我们比第四章更精确地讨论一下装置内容。

当您想在一个图形输出设备（诸如萤幕或者印表机）上绘图时，您首先必须获得一个装置内容（或者 DC）的代号。将代号传回给程式时，Windows 就给了您使用设备的许可权。然後您在 GDI 函式中将这个代号作为一个参数，向 Windows 标识您想在其上进行绘图的设备。

装置内容中包含许多确定 GDI 函式如何在设备上工作的目前「属性」，这些属性允许传递给 GDI 函式的参数只包含起始座标或者尺寸资讯，而不必包含 Windows 在设备上显示物件时需要的所有其他资讯。例如，呼叫 TextOut 时，您只需要在函式中给出装置内容代号、起始座标、文字和文字的长度。您不必指定字体、文字颜色、文字後面的背景色彩以及字元间距，因为这些属性都是装置内容的一部分。当您想改变这些属性之一时，您呼叫一个可以改变装置内容中属性的函式，以後针对该装置内容的 TextOut 呼叫来使用改变後的属性。

取得装置内容代号

Windows 提供了几种取得装置内容代号的方法。如果在处理一个讯息时取得

了装置内容代号，应该在退出视窗函式之前释放它（或者删除它）。一旦释放了代号，它就不再有效了。对於印表机装置内容代号，规则就没有这么严格。在第十三章会讨论列印。

最常用的取得并释放装置内容代号的方法是，在处理 WM_PAINT 讯息时，使用 BeginPaint 和 EndPaint 呼叫：

```
hdc = BeginPaint (hwnd, &ps) ;
```

其他行程式

```
EndPaint (hwnd, &ps) ;
```

变数 ps 是型态为 PAINTSTRUCT 的结构，该结构的 hdc 栏位是 BeginPaint 传回的装置内容代号。 PAINTSTRUCT 结构又包含一个名为 rcPaint 的 RECT（矩形）结构，rcPaint 定义一个包围视窗显示区域无效范围的矩形。使用从 BeginPaint 获得的装置内容代号，只能在这个区域内绘图。BeginPaint 呼叫使该区域有效。

Windows 程式还可以在处理非 WM_PAINT 讯息时取得装置内容代号：

```
hdc = GetDC (hwnd) ;
```

其他行程式

```
ReleaseDC (hwnd, hdc) ;
```

这个装置内容适用於视窗代号为 hwnd 的显示区域。这些呼叫与 BeginPaint 和 EndPaint 的组合之间的基本区别是，利用从 GetDC 传回的代号可以在整个显示区域上绘图。当然， GetDC 和 ReleaseDC 不使显示区域中任何可能的无效区域变成有效。

Windows 程式还可以取得适用於整个视窗（而不仅限於视窗的显示区域）的装置内容代号：

```
hdc = GetWindowDC (hwnd) ;
```

其他行程式

```
ReleaseDC (hwnd, hdc) ;
```

这个装置内容除了显示区域之外，还包括视窗的标题列、功能表、卷动列和框架（frame）。GetWindowDC 函式很少使用，如果想尝试用一用它，则必须拦截处理 WM_NCPAINT 讯息，Windows 使用该讯息在视窗的非显示区域上绘图。

BeginPaint、GetDC 和 GetWindowDC 获得的装置内容都与视讯显示器上的某个特定视窗相关。取得装置内容代号的另一个更通用的函式是 CreateDC：

```
hdc = CreateDC (pszDriver, pszDevice, pszOutput, pData) ;
```

其他行程式

```
DeleteDC (hdc) ;
```

例如，您可以通过下面的呼叫来取得整个萤幕的装置内容代号：

```
hdc = CreateDC (TEXT ("DISPLAY"), NULL, NULL, NULL) ;
```

在视窗之外写入画面一般是不恰当的，但对於一些不同寻常的应用程式来

说，这样做很方便（您还可通过在呼叫 GetDC 时使用一个 NULL 参数，从而取得整个萤幕的装置内容代号，不过这在文件中已经提到了）。在第十三章中，我们将使用 CreateDC 函式来取得一个印表机装置内容代号。

有时您只是需要取得关於某装置内容的一些资讯而并不进行任何绘画，在这种情况下，您可以使用 CreateIC 来取得一个「资讯内容」的代号，其参数与 CreateDC 函式相同，例如：

```
hdc = CreateIC (TEXT ("DISPLAY"), NULL, NULL, NULL) ;
```

您不能用这个资讯内容代号往设备上写东西。

使用点阵图时，取得一个「记忆体装置内容」有时是有用的：

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

其他行程式

```
DeleteDC (hdcMem) ;
```

您可以将点阵图选进记忆体装置内容，然後使用 GDI 函式在点阵图上绘画。我将在第十四章讨论这些技术。

前面已经提到过，metafile 是一些 GDI 呼叫的集合，以二进位形式编码。您可以通过取得 metafile 装置内容来建立 metafile：

```
hdcMeta = CreateMetaFile (pszFilename) ;
```

其他行程式

```
hmf = CloseMetaFile (hdcMeta) ;
```

在 metafile 装置内容有效期间，任何用 hdcMeta 所做的 GDI 呼叫都变成 metafile 的一部分而不会显示。在呼叫 CloseMetaFile 之後，装置内容代号变为无效，函式传回一个指向 metafile（hmf）的代号。我会在第十八章讨论 metafile。

取得装置内容资讯

一个装置内容通常是指一个实际显示设备，如视讯显示器和印表机。通常，您需要取得有关该设备的资讯，包括显示器的大小（单位为图素或者实际长度单位）和色彩显示能力。您可以通过呼叫 GetDeviceCaps（「取得设备功能」）函式来取得这些资讯：

```
iValue = GetDeviceCaps (hdc, iIndex) ;
```

其中，参数 iIndex 取值为 WINGDI.H 表头档案中定义的 29 个识别字之一。例如，iIndex 为 HORZRES 时将使 GetDeviceCaps 传回设备的宽度（单位为图素）；iIndex 为 VERTRES 时将让 GetDeviceCaps 传回设备的高度（单位为图素）。如果 hdc 是印表机装置内容的代号，则 GetDeviceCaps 传回印表机显示区域的高度和宽度，它们也是以图素为单位的。

还可以使用 GetDeviceCaps 来确定设备处理不同型态图形的能力，这对於

视讯显示器并不很重要，但是对於列印设备却是非常重要。例如，大多数绘图机不能画点阵图图像，GetDeviceCaps 就可以将这一情况告诉您。

DEVCAPS1 程式

程式 5-1 所示的 DEVCAPS1 程式显示了以一个视讯显示器的装置内容为参数时，可以从 GetDeviceCaps 函式中获得的部分资讯（该程式的另一个扩充版本 DEVCAPS2 将在第十三章给出，用於取得印表机资讯）。

程式 5-1　DEVCAPS1

```
DEVCAPS1.C
/*------------------------------------------------------------------------
   DEVCAPS1.C -- Device Capabilities Display Program No. 1
              (c) Charles Petzold, 1998
 ------------------------------------------------------------------------*/
#include <windows.h>
#define NUMLINES ((int) (sizeof devcaps / sizeof devcaps [0]))
struct
{
      int   iIndex ;
      TCHAR *     szLabel ;
      TCHAR *     szDesc ;
}
devcaps [] =
{
      HORZSIZE,   TEXT ("HORZSIZE"),TEXT ("Width in millimeters:"),
      VERTSIZE,   TEXT ("VERTSIZE"),TEXT ("Height in millimeters:"),
      HORZRES,    TEXT ("HORZRES"),TEXT ("Width in pixels:"),
      VERTRES,    TEXT ("VERTRES"),TEXT ("Height in raster lines:"),
      BITSPIXEL,  TEXT ("BITSPIXEL"),TEXT ("Color bits per pixel:"),
      PLANES,     TEXT ("PLANES"),   TEXT ("Number of color planes:"),
      NUMBRUSHES, TEXT ("NUMBRUSHES"),   TEXT ("Number of device brushes:"),
      NUMPENS,    TEXT ("NUMPENS"),      TEXT ("Number of device pens:"),
      NUMMARKERS, TEXT ("NUMMARKERS"),   TEXT ("Number of device markers:"),
      NUMFONTS,   TEXT ("NUMFONTS"),     TEXT ("Number of device fonts:"),
      NUMCOLORS,  TEXT ("NUMCOLORS"),    TEXT ("Number of device colors:"),
      PDEVICESIZE,     TEXT ("PDEVICESIZE"),  TEXT    ("Size    of    device
structure:"),
      ASPECTX,    TEXT ("ASPECTX"),      TEXT ("Relative width of pixel:"),
      ASPECTY,    TEXT ("ASPECTY"),TEXT ("Relative height of pixel:"),
      ASPECTXY,   TEXT ("ASPECTXY"),     TEXT ("Relative diagonal of pixel:"),
      LOGPIXELSX, TEXT ("LOGPIXELSX"),   TEXT ("Horizontal dots per inch:"),
      LOGPIXELSY, TEXT ("LOGPIXELSY"),   TEXT ("Vertical dots per inch:"),
      SIZEPALETTE,     TEXT ("SIZEPALETTE"),  TEXT    ("Number    of    palette
entries:"),
      NUMRESERVED,     TEXT ("NUMRESERVED"),  TEXT     ("Reserved     palette
entries:"),
```

```
      COLORRES,   TEXT ("COLORRES"),     TEXT ("Actual color resolution:")
} ;

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
          PSTR szCmdLine, int iCmdShow)
{
      static TCHAR szAppName[] = TEXT ("DevCaps1") ;
      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 ("Device Capabilities"),
                                   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  cxChar, cxCaps, cyChar ;
     TCHAR            szBuffer[10] ;
     HDC         hdc ;
     int         i ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;

     switch (message)
    {
     case WM_CREATE:
          hdc = GetDC (hwnd) ;
          GetTextMetrics (hdc, &tm) ;
          cxChar     = tm.tmAveCharWidth ;
          cxCaps     = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
          cyChar     = tm.tmHeight + tm.tmExternalLeading ;

          ReleaseDC (hwnd, hdc) ;
          return 0 ;

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;
          for (i = 0 ; i < NUMLINES ; i++)
          {
               TextOut (  hdc, 0, cyChar * i,
                    devcaps[i].szLabel,
                    lstrlen (devcaps[i].szLabel)) ;

               TextOut (  hdc, 14 * cxCaps, cyChar * i,
                    devcaps[i].szDesc,
                    lstrlen (devcaps[i].szDesc)) ;

               SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
               TextOut (hdc, 14*cxCaps+35*cxChar, cyChar*i, szBuffer,
                    wsprintf (szBuffer, TEXT ("%5d"),
                    GetDeviceCaps (hdc, devcaps[i].iIndex))) ;

          SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
          }

          EndPaint (hwnd, &ps) ;
          return 0 ;

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

```
}
```

可以看到，这个程式非常类似第四章的 SYSMETS1。为了保持程式码的短小，我没有使用卷动列，因为我知道资讯可以在一个画面上显示出来。在 256 色，640 480 的 VGA 上显示的结果如图 5-1 所示。

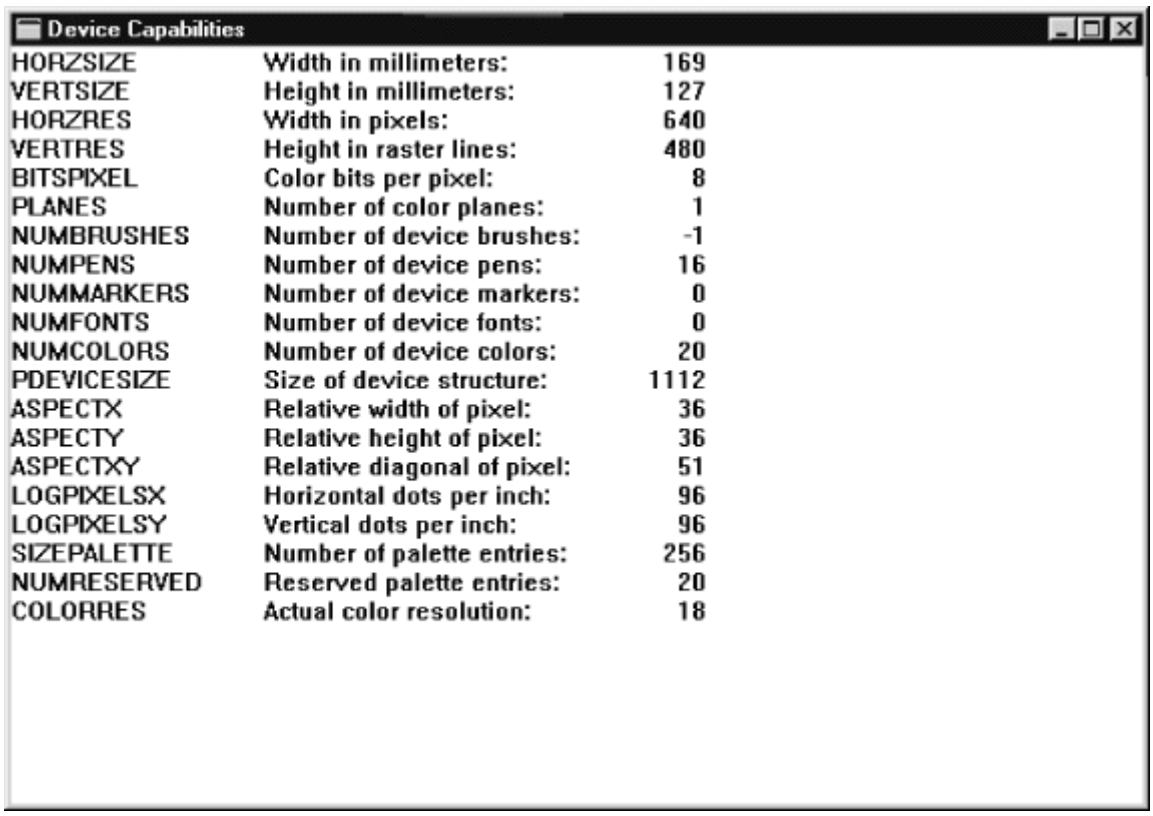


图 5-1 256 色，640×480VGA 上的 DEVCAPS1 显示

装置的大小

假定要绘制边长为 1 英寸的正方形，您（程式写作者）或 Windows（作业系统）需要知道视讯显示上 1 英寸对应多少图素。使用 GetDeviceCaps 函式能取得有关如视讯显示器和印表机之类输出设备的实际显示大小资讯。

视讯显示器和印表机是两个不同的设备。但也许最不明显的区别是「解析度」与装置联系起来的方式。对於印表机，我们经常用「每英寸的点数（dpi）」表示解析度。例如，大多数雷射印表机有 300 或 600dpi 的解析度。然而，视讯显示器的解析度是以水平和垂直的总图素数来表示的，例如，1024 768。大多数人不会告诉您他的印表机在一张纸上水平和垂直列印多少图素或他们的视讯显示器上每英寸有多少图素。

在本书中，我用「解析度」来严格定义每度量单位（一般为英寸）内的图素数。我使用「图素大小」或「图素尺寸」表示设备水平或垂直显示的总图素数。「度量大小」或「度量尺寸」是以英寸或毫米为单位的设备显示区域的大小。（对於印表机页面，它不是整个页面，只是可列印的区域。）图素大小除

以度量大小就得到解析度。

现在 Windows 使用的大多数视讯显示器的萤幕都是宽比高多 33%。这就表示纵横比为 1.33:1 或（一般写法）4:3。历史上，该比例可追溯到 Thomas Edison 制作电影的年代。它一直作为电影的标准纵横比，直到 1953 年出现各种型态的宽银幕投影机。电视机萤幕的纵横比也是 4:3。

然而，Windows 应用程式不应假设视讯显示器具有 4:3 的纵横比。人们进行文字处理时希望视讯显示器与一张纸的长和宽类似。最普通的选择是把 4:3 变为 3:4 显示，把标准显示翻转一下。

如果设备的水平解析度与垂直解析度相等，就称设备具有「正方形图素」。现在，Windows 普遍使用的视讯显示器都具有正方形图素，但也有例外。（应用程式也不应假设视讯显示器总是具有正方形图素。）Windows 第一次发表时，标准显示卡卡是 IBM Color Graphics Adapter（CGA），它有 640 200 的图素大小；Enhanced Graphics Adapter（EGA）有 640 350 的图素大小；Hercules Graphics Card 有 720 348 的图素大小。所有这些显示卡都使用 4:3 纵横比的显示器，但是水平和垂直图素数的比值都不是 4:3。

执行 Windows 的使用者很容易确定视讯显示器的图素大小。在「控制台」中执行「显示器」，并选择「设定」页面标签。在标有「桌面区域」的栏位中，可以看到这些图素尺寸之一：

- 640×480 图素
- 800×600 图素
- 1024×768 图素
- 1280×1024 图素
- 1600×1200 图素

所有这些都是 4:3。（除了 1280 1024 图素大小。这不但有些不好，还有些令人反感。所有这些图素尺寸都认为在 4:3 的显示器上会产生正方形的图素。）

Windows 应用程式可以使用 SM_CXSCREEN 和 SM_CYSCREEN 参数从 GetSystemMetrics 得到图素尺寸。从 DEVCAPS1 程式中您会注意到，程式可以用 HORZRES（水平解析度）和 VERTRES 参数从 GetDeviceCaps 中得到同样的值。这里「解析度」指的是图素大小而不是每度量单位的图素数。

这些是设备大小的简单部分，现在开始复杂的部分。

前两个设备能力，HORZSIZE 和 VERTSIZE，文件中称为「以毫米计的实际萤幕的宽度」及「以毫米计的实际萤幕的高度」（在/Platform SDK/Graphics 和 Multimedia Services/GDI/Device Contexts/Device Context Reference/Device Context Functions/GetDeviceCaps 中）。这些看起来更像

直接的定义。例如，给出视讯显示卡和显示器的介面特性，Windows 如何真正知道显示器的大小呢？如果您有台膝上型电脑（它的视讯驱动程式能知道准确的萤幕大小）并且连接了外部显示器，又是哪种情况呢？如果把视讯投影机连接到电脑上呢？

在 Windows 的 16 位元版本中（及在 Windows NT 中），Windows 为 HORZSIZE 和 VERTSIZE 使用「标准」的显示大小。然而，从 Windows 95 开始，HORZSIZE 和 VERTSIZE 值是从 HORZRES、VERTRES、LOGPIXELSX 和 LOGPIXELSY 值中衍生出来的。这是它的工作方式。

当您在「控制台」中使用「显示器」程式选择显示的图素大小时，也可以选择系统字体的大小。这个选项的原因是用於 640 480 显示的字体在提升到 1024 768 或更大时字太小，而您可能想要更大的系统字体。这些系统字体大小指「显示器」程式的「设定」页面标签中的「小字体」和「大字体」。

在传统的排版中，字体的字母大小由「点」表示。1 点大约 1/72 英寸，在电脑排版中 1 点正好为 1/72 英寸。

理论上，字体的点值是从字体中最高的字元顶部到例如 j、p、q 和 y 等字母下部的字元底部的距离，其中不包括重音符号。例如，在 10 点的字体中此距离是 10/72 英寸。根据 TEXTMETRIC 结构，字体的点值等於 tmHeight 栏位减去 tmInternalLeading 栏位，如图 5-2 所示（该图与上一章的图 4-3 一样）。

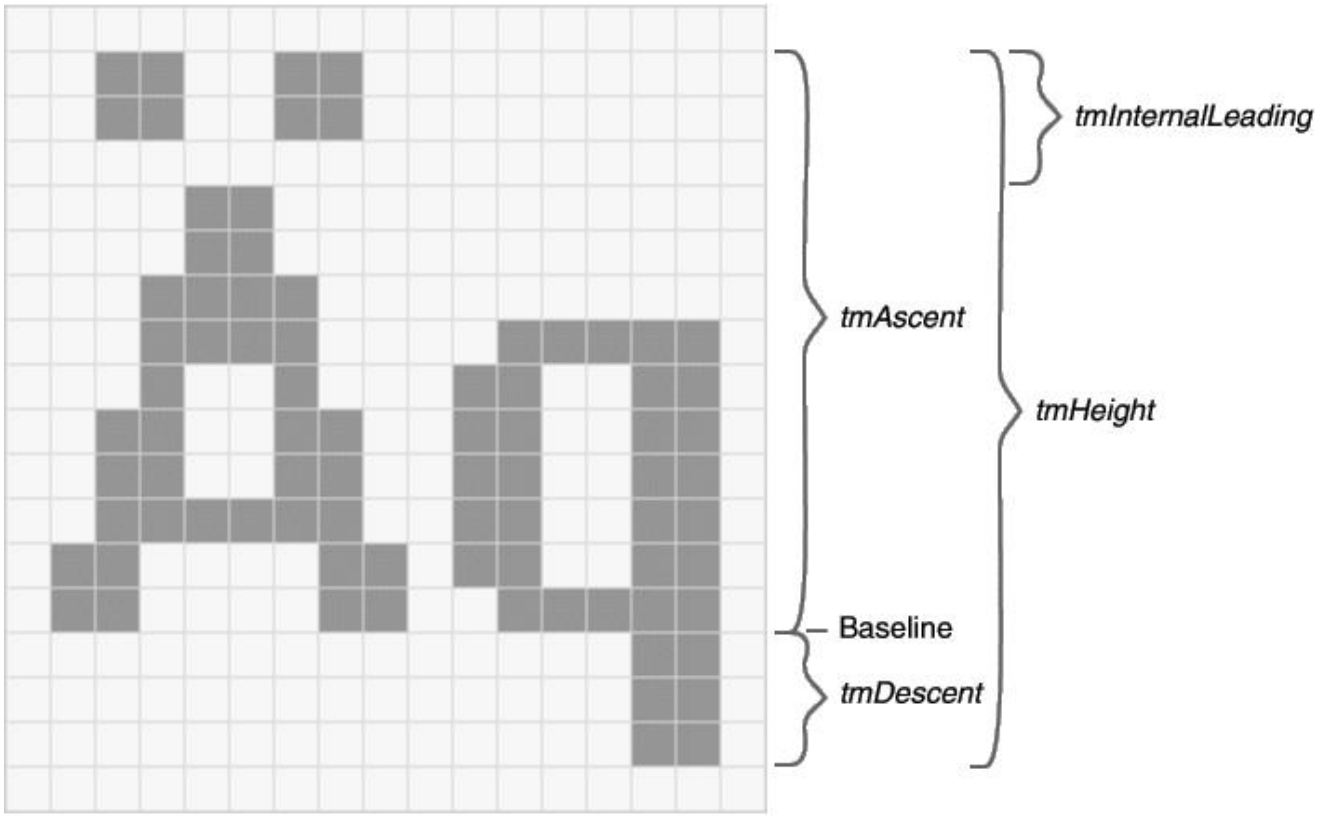


图 5-2 小字体和 TEXTMETRIC 栏位。

在真正的排版中，字体的点值与字体字母的实际大小并不正好相等。字体

的设计者做出的实际字元比点值指示的要大一些或小一些。毕竟，字体设计是一种艺术而不是科学。

TEXTMETRIC 结构的 tmHeight 栏位指出文字的连续行在萤幕或印表机上间隔的方式。这也可以用点来测量。例如，12 点的行距指出文字连续行的基准线应该间隔 12/72（或 1/6）英寸。不应该为 10 点字体使用 10 点行距，因为文字的连续行会碰到一起。

10 点字体读起来很舒服。小於 10 点的字体不益於长时间阅读。

Windows 系统字体——不考虑是大字体还是小字体，也不考虑所选择的视频图素大小——固定假设为 10 点字体和 12 点行距。这听起来很奇怪，如果字体都是 10 点，为什么还把它们称为大字体和小字体呢？

解答是： **当您在「控制台」的「显示」程式上选择小字体或大字体时，实际上是选择了一个假定的视讯显示解析度，单位是每英寸的点数** 。当选择小字体时，即要 Windows 假定视讯显示解析度为每英寸 96 点。当选择大字体时，即要 Windows 假定视讯显示解析度为每英寸 120 点。

再看看图 5-2。那是小字体，它依据的显示解析度为每英寸 96 点。我说过它是 10 点字体。10 点即是 10/72 英寸，如果乘以 96 点，每英寸大概就为 13 图素。这即是 tmHeight 减去 tmInternalLeading 的值。行距是 12 点，或 12/72 英寸，它乘以 96 点，每英寸就为 16 图素。这即是 tmHeight 的值。

图 5-3 显示大字体。这是依据每英寸 120 点的解析度。同样，它是 10 点字体，10/72 乘以 120 点，每英寸等於 16 图素，即是 tmHeight 减 tmInternalLeading 的值。12 点行距等於 20 图素，即是 tmHeight 的值。（像第四章一样，再次强调所显示的是实际的度量大小，因此您可以理解它工作的方式。不要在您的程式中对此写作程式。）

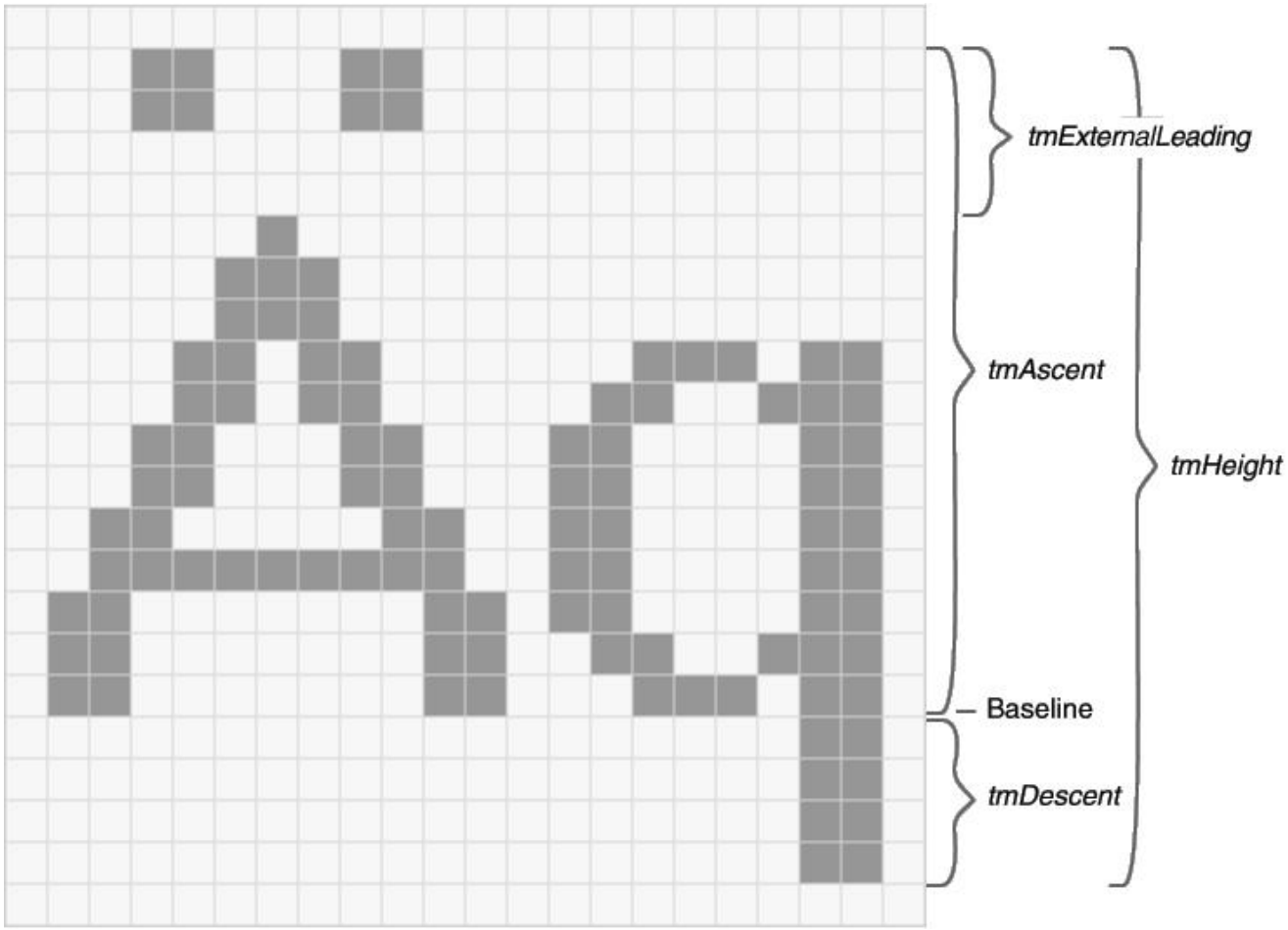


图 5-3 大字体和 FONTMETRIC 栏位

在 Windows 程式中，您可以使用 GetDeviceCaps 函式取得使用者在「控制台」的「显示器」程式中选择的以每英寸的点数为单位的假定解析度。要得到这些值（如果视讯显示器不具有正方形图素，在理论上这些值是不同的），可以使用索引 LOGPIXELSX 和 LOGPIXELSY。LOGPIXELS 指逻辑图素，它的基本意思是「以每英寸的图素数为单位的非实际解析度」。

用 HORZSIZE 和 VERTSIZE 索引从 GetDeviceCaps 得到的设备能力，在文件上称为「实际萤幕的宽度，单位毫米」及「实际萤幕的高度，单位毫米」。因为这些值是从 HORZRES、VERTRES、LOGPIXELSX 和 LOGPIXELSY 值中衍生出来的，所以它们应该称为「逻辑宽度」和「逻辑高度」。公式是：

$$\text{水平大小(mm)} = 25.4 \times \frac{\text{水平解析度(圖素)}}{\text{邏輯圖素}X\text{(每英寸的點數)}}$$

$$\text{垂直大小(mm)} = 25.4 \times \frac{\text{垂直解析度(圖素)}}{\text{邏輯圖素}Y\text{(每英寸的點數)}}$$

常数 25.4 用於把英寸转变为毫米。

这看起来是种不合逻辑的退步。毕竟，视讯显示器是可以用尺以毫米为单位的大小（至少是近似的）衡量的。但是 Windows 98 并不关心这个大小。相反，

它以使用者选择的显示图素大小和系统字体大小为基础计算以毫米为单位的显示大小。更改显示的图素大小并根据 GetDeviceCaps 更改度量大小。这有什么意义呢？

这非常有意义。假定有一个 17 英寸的显示器。实际的显示大小大约是 12 英寸乘 9 英寸。假定在最小要求的 640 480 图素大小下执行 Windows。这意味著实际的解析度是每英寸 53 点。10 点字体（在纸上便於阅读）在萤幕上从 A 的顶部到 q 的底部只有 7 个图素。这样的字体很难看而且不易读。（可问问那些在旧的 Color Graphics Adapter 上执行 Windows 的人们。）

现在，把您的电脑接上视讯投影机。投影的视讯显示器是 4 英尺宽，3 英尺高。同样的 640 480 图素大小现在是大约每英寸 13 点的解析度。在这种条件下试图显示 10 点的字体是很可笑的。

10 点字体在视讯显示器上应是可读的，因为它在列印时是肯定可读的。所以 10 点字体就成为一个重要的参照。当 Windows 应用程式确保 10 点萤幕字体为平均大小时，就能够使用 8 点字体显示较小的文字（仍可读），或用大於 10 点的字体显示较大的文字。因而，视频解析度（以每英寸的点数为单位）由 10 点字体的图素大小来确定是很有意义的。

然而，在 Windows NT 中，用老的方法定义 HORZSIZE 和 VERTSIZE 值。这种方法与 Windows 的 16 位元版本一致。HORZRES 和 VERTRES 值仍然表示水平和垂直图素的数值，LOGPIXELSX 和 LOGPIXELSY 仍然与在「控制台」的「显示器」程式中选择的字体有关。在 Windows 98 中，LOGPIXELSX 和 LOGPIXELSY 的典型值是 96 和 120 dpi，这取决於您选择的是小字体还是大字体。

在 Windows NT 中的区别是 HORZSIZE 和 VERTSIZE 值固定表示标准显示器大小。对於普通的显示卡，取得的 HORZSIZE 和 VERTSIZE 值分别是 320 和 240 毫米。这些值是相同的，与选择的图素大小无关。因此，这些值与用 HORZRES、VERTRES、LOGPIXELSX 和 LOGPIXELSY 索引从 GetDeviceCaps 中得到的值不同。然而，可以用前面的公式计算在 Windows 98 下的 HORZSIZE 和 VERTSIZE 值。

如果程式需要实际的视讯显示大小该怎么办？也许最好的解决方法是用对话方块让使用者输入它们。

最後，来自 GetDeviceCaps 的另三个值与视讯大小有关。ASPECTX、ASPECTY 和 ASPECTXY 值是每一个图素的相对宽度、高度和对角线大小，四舍五入到整数。对於正方形图素，ASPECTX 和 ASPECTY 值相同。无论如何，ASPECTXY 值应等於 ASPECTX 与 ASPECTY 平方和的平方根，就像直角三角形一样。

关於色彩

如果视讯显示卡仅显示黑色图素和白色图素，则每个图素只需要记忆体中的一位元。彩色显示器中每个图素需要多个位元。位元数越多，色彩越多，或者更具体地说，可以同时显示的不同色彩的数目等於 2 的位元数次方。

「Full-Color」视讯显示器的解析度是每个图素 24 位元——8 位元红色、8 位元绿色以及 8 位元蓝色。红、绿、蓝即「色光三原色」。混合这三种基本颜色可以生成许多其他的颜色，您通过放大镜看显示幕，就可以看出来。

「High-Color」显示解析度是每个图素 16 位元——5 位元红色、6 位元绿色以及 5 位元蓝色。绿色多一位元是因为人眼对绿色更敏感一些。

显示 256 种颜色的显示卡每个图素需要 8 位元。然而，这些 8 位元的值一般由定义实际颜色的调色盘组织的。我会在第十六章详细地讨论它们。

最後，显示 16 种颜色的显示卡每个图素需要 4 位元。这 16 种颜色一般固定分为暗的或亮的红、黑、蓝、青、紫、黄、两种灰色。这 16 种颜色要回溯到老式的 IBM CGA。

只有在某些怪异的程式中才需要知道视讯显示卡上的记忆体是如何组织的，但是 GetDeviceCaps 使程式写作者可以知道显示卡的储存组织以及它能够表示的色彩数目，下面的呼叫传回色彩平面的数目：

```
iPlanes = GetDeviceCaps (hdc, PLANES) ;
```

下面的呼叫传回每个图素的色彩位元数：

```
iBitsPixel = GetDeviceCaps (hdc, BITSPIXEL) ;
```

大多数彩色图形显示设备使用多个色彩平面或每图素有多个色彩位元的设计，但是不能同时一齐使用这两种方式；换句话说，这两个呼叫必有一个传回 1。显示卡能够表示的色彩数可以用如下公式来计算：

```
iColors = 1 << (iPlanes * iBitsPixel) ;
```

这个值与用 NUMCOLORS 参数得到的色彩数值可能一样，也可能不一样：

```
iColors = GetDeviceCaps (hdc, NUMCOLORS) ;
```

我提到过，256 色的显示卡使用色彩调色盘。在那种情况下，以 NUMCOLORS 为参数时，GetDeviceCaps 传回由 Windows 保留的色彩数，值为 20，剩余的 236 种颜色可以由 Windows 程式用调色盘管理器设定。对於 High-Color 和 True-Color 显示解析度，带有 NUMCOLORS 参数的 GetDeviceCaps 通常传回-1，这样就无法得到需要的资讯，因此应该使用前面所示的带有 PLANES 和 BITSPIXEL 值的 iColors 公式。

在大多数 GDI 函式呼叫中，使用 COLORREF 值（只是一个 32 位元的无正负号长整数）来表示一种色彩。COLORREF 值按照红、绿和蓝色的亮度指定了一种颜色，通常叫做「RGB 色彩」 。32 位元的 COLORREF 值的设定如图 5-4 所示。

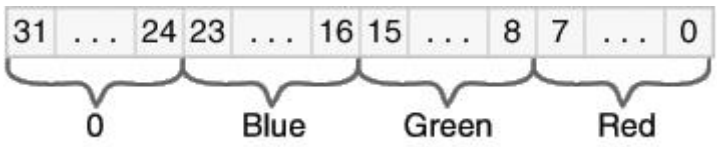


图 5-4 32 位 COLORREF 值

注意最前面是标为 0 的 8 个位元，并且每种原色都指定为一个 8 位元的值。理论上，COLORREF 可以指定二的二十四次方种或一千六百万种色彩。

这个无正负号长整数常常称为一个「RGB 色彩」。Windows 表头档案 WINGDI.H 提供了几种使用 RGB 色彩值的巨集。RGB 巨集要求三个参数分别代表红、绿和蓝值，然後将它们组合为一个无正负号长整数：

```
#define RGB(r,g,b) ((COLORREF)(((BYTE)(r) | \
                   ((WORD)((BYTE)(g)) << 8)) | \
                   (((DWORD)(BYTE)(b)) << 16)))
```

注意三个参数的顺序是红、绿和蓝。因此，值：

```
RGB (255, 255, 0)
```

是 0x0000FFFF，或黄色（红色和绿色的合成）。当所有三个参数设定为 0 时，色彩为黑色；当所有参数设定为 255 时，色彩为白色。GetRValue、GetGValue 和 GetBValue 巨集从 COLORREF 值中抽取出原色值。当您在使用传回 RGB 色彩值的 Windows 函式时，这些巨集有时会很方便。

在 16 色或 256 色显示卡上，Windows 可以使用「混色」来类比设备能够显示的颜色之外的色彩。混色利用了由多种色彩的图素组成的图素图案。可以呼叫 GetNearestColor 来决定与某一色彩最接近的纯色：

```
crPureColor = GetNearestColor (hdc, crColor) ;
```

装置内容属性

前面已经提到过，Windows 使用装置内容来保存控制 GDI 函式在显示器上如何操作的「属性」。例如，在用 TextOut 函式显示文字时，程式写作者不必指定文字的色彩和字体，Windows 从装置内容取得这个资讯。

程式取得一个装置内容的代号时，Windows 用预设值设定所有的属性（在下一节会看到如何取代这种设定）。表 5-1 列出了 Windows 98 支援的装置内容属性，程式可以改变或者取得任何一种属性。

表 5-1

装置内容属性	预设值	修改该值的函式	取得该值的函式
Mapping Mode	MM_TEXT	SetMapMode	GetMapMode
Window Origin	(0, 0)	SetWindowOrgEx OffsetWindowOrgEx	GetWindowOrgEx

Viewport Origin	(0, 0)	SetViewportOrgEx OffsetViewportOrgEx	GetViewportOrgEx
Window Extents	(1, 1)	SetWindowExtEx SetMapMode ScaleWindowExtEx	GetWindowExtEx
Viewport Extents	(1, 1)	SetViewportExtEx SetMapMode ScaleViewportExtEx	GetViewportExtEx
Pen	BLACK_PEN	SelectObject	SelectObject
Brush	WHITE_BRUSH	SelectObject	SelectObject
Font	SYSTEM_FONT	SelectObject	SelectObject
Bitmap	None	SelectObject	SelectObject
Current Position	(0, 0)	MoveToEx LineTo PolylineTo PolyBezierTo	GetCurrentPositionEx
Background Mode	OPAQUE	SetBkMode	GetBkMode
Background Color	White	SetBkColor	GetBkColor
Text Color	Black	SetTextColor	GetTextColor
Drawing Mode	R2_COPYPEN	SetROP2	GetROP2
Stretching Mode	BLACKONWHITE	SetStretchBltMode	GetStretchBltMode
Polygon Fill Mode	ALTERNATE	SetPolyFillMode	GetPolyFillMode
Intercharacter Spacing	0	SetTextCharacterExtra	GetTextCharacterExtra
Brush Origin	(0, 0)	SetBrushOrgEx	GetBrushOrgEx
Clipping Region	None	SelectObject SelectClipRgn IntersectClipRgn OffsetClipRgn ExcludeClipRect SelectClipPath	GetClipBox

保存装置内容

通常，在您呼叫 GetDC 或 BeginPaint 时，Windows 用预设值建立一个新的装置内容，您对属性所做的一切改变在装置内容用 ReleaseDC 或 EndPaint 呼叫释放时，都会丢失。如果您的程式需要使用非内定的装置内容属性，则您必须在每次取得装置内容代号时初始化装置内容：

```
case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;
    装置内容属性
    绘制视窗显示区域
    EndPaint (hwnd, &ps) ;
    return 0 ;
```

虽然在通常情况下这种方法已经很令人满意了，但是您还可能想要在释放装置内容之後，仍然保存程式中对装置内容属性所做的改变，以便在下一次呼叫 GetDC 和 BeginPaint 时它们仍然能够起作用。为此，可在登录视窗类别时，将 CS_OWNDC 旗标纳入视窗类别的一部分：

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

现在，依据这个视窗类别所建立的每个视窗都将拥有自己的装置内容，它一直存在，直到视窗被删除。如果使用了 CS_OWNDC 风格，就只需初始化装置内容一次，可以在处理 WM_CREATE 讯息处理期间完成这一操作：

```
case  WM_CREATE:
        hdc = GetDC (hwnd) ;
    初始化装置内容属性
        ReleaseDC (hwnd, hdc) ;
```

这些属性在改变之前一直有效。

CS_OWNDC 风格只影响 GetDC 和 BeginPaint 获得的装置内容，不影响其他函式（如 GetWindowDC）获得的装置内容。以前不提倡使用 CS_OWNDC 风格，因为它需要记忆体；现在，在处理大量图形的 Windows NT 应用程式中，它可以提高性能。即使用了 CS_OWNDC，您仍然应该在退出视窗讯息处理程式之前释放装置内容。

某些情况下，您可能想改变某些装置内容属性，用改变後的属性进行绘图，然後恢复原来的装置内容。要简化这一过程，可以通过如下呼叫来保存装置内容的状态：

```
idSaved = SaveDC (hdc) ;
```

现在，可以改变一些属性，在想要回到呼叫 SaveDC 前存在的装置内容时，呼叫：

```
RestoreDC (hdc, idSaved) ;
```

您可以在呼叫 RestoreDC 之前呼叫 SaveDC 数次。

大多数程式写作者以不同的方式使用 SaveDC 和 RestoreDC。然而，更像组合语言中的 PUSH 和 POP 指令，当您呼叫 SaveDC 时，不需要保存传回值：

```
SaveDC (hdc) ;
```

然後，您可以更改某些属性并再次呼叫 SaveDC。要将装置内容恢复到一个已经保存的状态，呼叫：

```
RestoreDC (hdc, -1) ;
```

这就将装置内容恢复到最近由 SaveDC 函式保存的状态中。

画点和线

在第一章，我们谈论过 Windows 图形装置介面将图形输出设备的装置驱动程式与电脑连在一起的方式。在理论上，只要提供 SetPixel 和 GetPixel 函式，就可以使用图形装置驱动程式绘制一切东西了。其余的一切都可以使用 GDI 模组中实作的更高阶的常式来处理。例如，画线时，只需 GDI 呼叫 SetPixel 数次，并适当地调整 x 和 y 座标。

在实际情况中，也的确可以仅使用 SetPixel 和 GetPixel 函式进行您需要的任何绘制。您也可以在这些函式的基础上设计出简洁和构造良好的图形编程系统。唯一的问题是启能。如果一个函式通过几次呼叫才能到达 SetPixel 函式，那么它执行起来会非常慢。如果一个图形系统画线和进行其他复杂的图形操作是在装置驱动程式的层次上，它就会更有效得多，因为装置驱动程式对完成这些操作的程式码进行了最佳化。此外，一些显示卡包含了图形辅助运算器，它允许视讯硬体自己绘制图形。

设定图素

即使 Windows GDI 包含了 SetPixel 和 GetPixel 函式，但很少使用它们。在本书，仅在第七章的 CONNECT 程式中使用了 SetPixel 函式，仅在第八章的 WHATCLR 程式中使用了 GetPixel 函式。尽管如此，由它们开始来研究图形仍是非常方便。

SetPixel 函式在指定的 x 和 y 座标以特定的颜色设定图素：

```
SetPixel (hdc, x, y, crColor) ;
```

如同在任何绘图函式中一样，第一个参数是装置内容的代号。第二个和第三个参数指明了座标位置。通常要获得视窗显示区域的装置内容，并且 x 和 y 相对于该显示区域的左上角。最後一个参数是 COLORREF 型态指定了颜色。如果在函式中指定的颜色视讯显示器不支援，则函式将图素设定为最接近的纯色并从函式传回该值。

GetPixel 函式传回指定座标处的图素颜色：

```
crColor = GetPixel (hdc, x, y) ;
```

直线

Windows 可以画直线、椭圆线（椭圆圆周上的曲线）和贝塞尔曲线。Windows 98 支援的 7 个画线函式是：

- LineTo 画直线。
- Polyline 和 PolylineTo 画一系列相连的直线。
- PolyPolyline 画多组相连的线。
- Arc 画椭圆线。
- PolyBezier 和 PolyBezierTo 画贝塞尔曲线。

另外，Windows NT 还支援 3 种画线函式：

- ArcTo 和 AngleArc 画椭圆线。
- PolyDraw 画一系列相连的线以及贝塞尔曲线。

这三个函式 Windows 98 不支援。

在本章的後面我将介绍一些既画线也填入所画图形的封闭区域的函式，这些函式是：

- Rectangle 画矩形。
- Ellipse 画椭圆。
- RoundRect 画带圆角的矩形。
- Pie 画椭圆的一部分，使其看起来像一个扇形。
- Chord 画椭圆的一部分，以呈弓形。

装置内容的五个属性影响著用这些函式所画线的外观：目前画笔的位置（仅用於 LineTo、PolylineTo、PolyBezierTo 和 ArcTo ）、画笔、背景方式、背景色和绘图模式。

画一条直线，必须呼叫两个函式。第一个函式指定了线的开始点，第二个函式指定了线的终点：

```
MoveToEx (hdc, xBeg, yBeg, NULL) ;
LineTo (hdc, xEnd, yEnd) ;
```

MoveToEx 实际上不会画线，它只是设定了装置内容的「目前位置」属性。然後 LineTo 函式从目前的位置到它所指定的点画一条直线。目前位置只是用於其他几个 GDI 函式的开始点。在内定的装置内容中，目前位置最初设定在点（0,0）。如果在呼叫 LineTo 之前没有设定目前位置，那么它将从显示区域的左上角开始画线。

小历史:

Windows 的 16 位元版本中，用来改变目前位置的函式是 MoveTo。该函式只调整三个参数——装置内容代号、x 和 y 座标。函式通过两个 16 位元数拼成的 32 位元无正负号长整数传回先前的目前位置。然而，在 Windows 的 32 位元版本中，座标是 32 位元的数值，而 C 的 32 位元版本中又没有定义 64 位元的整数资料型态，因此这种改变意味著 MoveTo 在其传回值中不再指出先前的目前位置。在实际的程式写作中，由 MoveTo 传回的值几乎从来不用，因此就需要一个新函式，这就是 MoveToEx。

MoveToEx 的最後一个参数是指向 POINT 结构的指标。从该函式传回後，POINT 结构的 x 和 y 栏位指出了先前的目前位置。如果您不需要这种资讯（通常如此），可以简单地如上面的例子所示的那样将最後一个参数设定为 NULL。

警告：

尽管 Windows 98 中的座标值看起来是 32 位元的，实际上却只用到了低 16 位元，座标值实际上被限制在-32,768 到 32,767 之间。在 Windows NT 中，使用完整的 32 位元值。

如果您需要目前位置，就可以通过以下呼叫获得：

GetCurrentPositionEx (hdc, &pt) ;

其中，pt 是 POINT 结构的。

下面的程式码从视窗的左上角开始，在显示区域中画一个网格，线与线之间相隔 100 个图素，其中 hwnd 是视窗代号，hdc 是装置内容代号，而 x 和 y 是整数：

```
GetClientRect (hwnd, &rect) ;
for ( x = 0 ; x < rect.right ; x+= 100)
{
      MoveToEx (hdc, x, 0, NULL) ;
      LineTo (hdc, x, rect.bottom) ;
}
for (y = 0 ; y < rect.bottom ; y += 100)
{
      MoveToEx (hdc, 0, y, NULL) ;
      LineTo (hdc, rect.right, y) ;
}
```

虽然用两个函式来画一条直线显得有些麻烦，但是在希望画一组相连的直线时，目前画笔位置属性又会变得很有用。例如，您可能想定义一个包含 5 个点（10 个值）的阵列，来画一个矩形的边界框：

```
POINT apt[5] = { 100, 100, 200, 100, 200, 200, 100, 200, 100, 100 } ;
```

注意，最後一个点与第一个点相同。现在，只需要使用 MoveToEx 移到第一个点，并对後面的点使用 LineTo：

```
MoveToEx (hdc, apt[0].x, apt[0].y, NULL) ;
for ( i = 1 ; i < 5 ; i++)
      LineTo (hdc, apt[i].x, apt[i].y) ;
```

由於 LineTo 从目前位置画到（但不包括）LineTo 函式中给出的点，所以这段程式码没有在任何座标处画两次。虽然在显示器上多输出几次不存在问题，但是在绘图机上或者在其他绘图方式（下面马上会讲到）下，视觉效果就不太好了。

当您要将阵列中的点连接成线时，使用 Polyline 函式要简单得多。下面这条叙述画出与上面一段程式码相同的矩形：

```
Polyline (hdc, apt, 5) ;
```

最後一个参数是点的数目。我们还可以使用(sizeof (apt) / sizeof (POINT))来表示这个值。Polyline 与一个 MoveToEx 函式後面加几个 LineTo 函式的效果相同，但是，Polyline 既不使用也不改变目前位置。PolylineTo 有些不同，这个函式使用目前位置作为开始点，并将目前位置设定为最後一根线的终点。下面的程式码画出与上面所示一样的矩形：

```
MoveToEx (hdc, apt[0].x, apt[0].y, NULL) ;
PolylineTo (hdc, apt + 1, 4) ;
```

您可以对几条线使用 Polyline 和 PolylineTo，这些函式在绘制复杂曲线最有用了。您使用由几百甚至几千条线组成的极短线段，把它们连在一起就像一条曲线一样。例如，画正弦波就是这样的，程式 5-2 所示的 SINEWAVE 程式显示了如何做到这一点。

程式 5-2　SINEWAVE

SINEWAVE.C

```
/*--------------------------------------------------------------------------
   SINEWAVE.C -- Sine Wave Using Polyline
          (c) Charles Petzold, 1998
--------------------------------------------------------------------------*/
#include <windows.h>
#include <math.h>

#define NUM 1000
#define TWOPI    (2 * 3.14159)
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

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

```
     MSG         msg ;
     WNDCLASS    wndclass ;

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

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

     hwnd = CreateWindow (  szAppName, TEXT ("Sine Wave Using Polyline"),
                     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 ;
     HDC        hdc ;
     int        i ;
     PAINTSTRUCT ps ;
     POINT      apt [NUM] ;

     switch (message)
     {
     case  WM_SIZE:
```

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

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

            MoveToEx (hdc, 0,      cyClient / 2, NULL) ;
            LineTo   (hdc, cxClient, cyClient / 2) ;

            for (i = 0 ; i < NUM ; i++)
        {
                  apt[i].x = i * cxClient / NUM ;
                  apt[i].y = (int) (cyClient / 2 * (1 - sin (TWOPI * i / NUM))) ;
        }

            Polyline (hdc, apt, NUM) ;
            return 0 ;

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

这个程式有一个含有 1000 个 POINT 结构的阵列。随著 for 回圈从 0 增加到 999，结构的 x 成员设定为从 0 递增到数值 cxClient。结构的 y 成员设定为一个周期的正弦曲线值，并被放大以填满显示区域。整个曲线的绘制仅仅使用了一个 Polyline 呼叫。因为 Polyline 函式是在装置驱动程式层次上实作的，因此它要比呼叫 1000 次 LineTo 快得多，结果如图 5-5 所示。

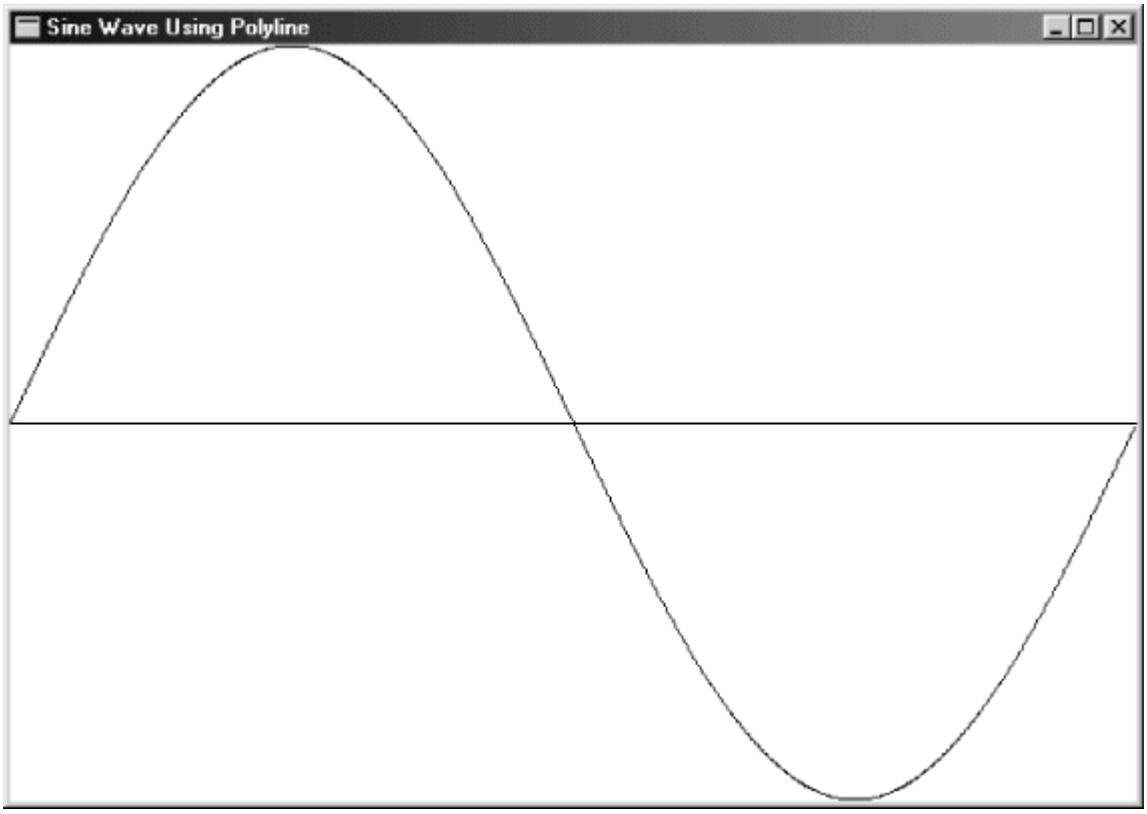


图 5-5 SINEWAVE 显示

边界框函式

下面我想讨论的是 Arc 函式，它绘制椭圆曲线。然而，如果不先讨论一下 Ellipse 函式，那么 Arc 函式将难以理解；而如果不先讨论 Rectangle 函式，那么 Ellipse 函式又将难以理解；而如果讨论 Ellipse 和 Rectangle 函式，那么我又会讨论 RoundRect、Chord 和 Pie 函式。

问题在於，Rectangle、Ellipse、RoundRect、Chord 和 Pie 函式严格来说不是画线函式。没错，这些函式是在画线，但它们同时又填入画刷填入一个封闭区域。这个画刷内定为白色，因此当您第一次使用这些函式时，您可能不会注意到它们不只是画线。严格地说，这些函式属於後面「填入区域」的小节，不过，我还是在这里讨论它们。

上面提到的函式有一个共同特性，即它们都是依据一个矩形边界框来绘图的。您定义一个包含该物件的框，即「边界框(bounding box)」；Windows 就在这个框内画出该物件。

这些函式中最简单的就是画一个矩形：

```
Rectangle (hdc, xLeft, yTop, xRight, yBottom) ;
```

点(xLeft, yTop)是矩形的左上角，(xRight, yBottom)是矩形的右下角。用函式 Rectangle 画出的图形如图 5-6 所示，矩形的边总是平行于显示器的水平和垂直边。

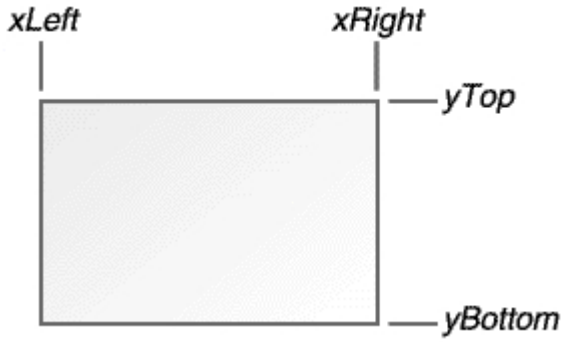


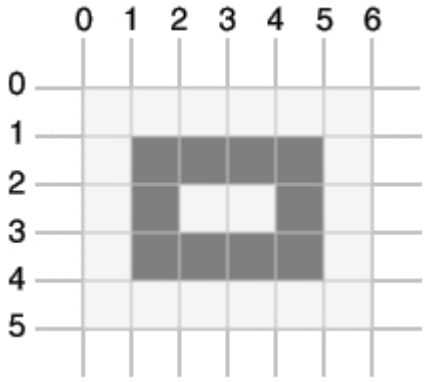
图 5-6 使用 Rectangle 函式画出的图形

以前写过图形程式的程式写作者熟悉图素偏差的问题。有些图形系统画出的图形包含右座标和底座标，而有些则只画到（而不包含）右座标和底座标。Windows 采用後一种方法，不过有一种更简单的方法来思考这个问题。

考虑下面的函式呼叫：

```
Rectangle (hdc, 1, 1, 5, 4) ;
```

上面我们提到，Windows 在边界框内画图。可以将显示器想像成一个网格，其中，每个图素都在一个网格单元内。边界框画在网格上，然後在边界框内画矩形，下面说明了图形画出来时的样子：



将矩形和显示区域左上角分开的区域有 1 个图素宽。

我以前提到过，Rectangle 严格地说不是画线函式，GDI 也填入封闭区域。然而，因为内定用白色填入区域，因此 GDI 填入区域并不明显。

您知道了如何画矩形，也就知道了如何画椭圆，因为它们使用的参数都是相同的：

```
Ellipse (hdc, xLeft, yTop, xRight, yBottom) ;
```

用 Ellipse 函式画出的图形如图 5-7 所示（加上了虚线构成的边界框）。

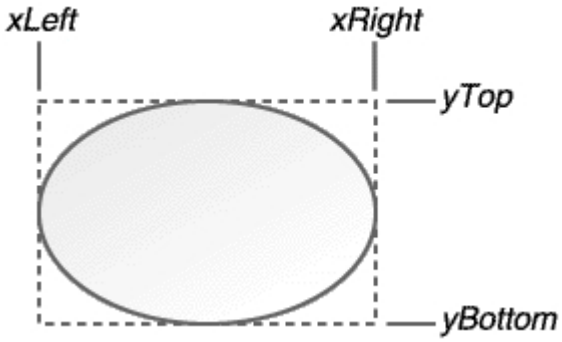


图 5-7 用 Ellipse 函式画出的图形

画圆角矩形的函式使用与函式 Rectangle 及 Ellipse 函式相同的边界框，还包含另外两个参数：

```
RoundRect (hdc, xLeft, yTop, xRight, yBottom,
           xCornerEllipse, yCornerEllipse) ;
```

用这个函式画出的图形如 5-8 所示。

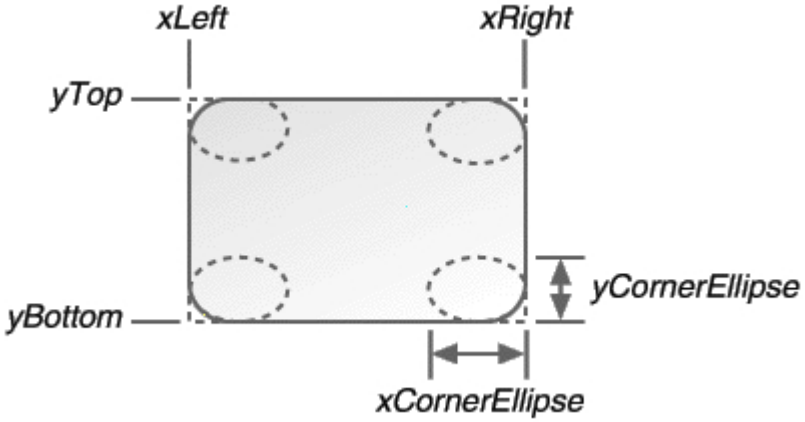


图 5-8 用 RoundRect 函式画出的图形

Windows 使用一个小椭圆来画圆角，这个椭圆的宽为 xCornerEllipse，高为 yCornerEllipse。可以想像这个小椭圆分为了四个部分，一个象限一个，每个刚好用在矩形的一个角上。 xCornerEllipse 和 yCornerEllipse 的值越大，角就越明显。如果 xCornerEllipse 等於 xLeft 与 xRight 的差，且 yCornerEllipse 等於 yTop 与 yBottom 的差，那么 RoundRect 函式将画出一个椭圆。

在绘制图 5-8 所示的圆角矩形时，用了下面的公式来计算角上椭圆的尺寸。

```
xCornerEllipse = (xRight - xLeft) / 4 ;
yCornerEllipse = (yBottom- yTop) / 4 ;
```

这是一种简单的方法，但是结果看起来有点不对劲，因为角的弯曲部分在矩形长的一边要大些。要矫正这一问题，您可以让 xCornerEllipse 与 yCornerEllipse 的值相等。

Arc、Chord 和 Pie 函式都只要相同的参数：

```
Arc   (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd) ;
Chord (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd) ;
Pie   (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd) ;
```

用 Arc 函式画出的线如图 5-9 所示；用 Chord 和 Pie 函式画出的线分别如图 5-10 和 5-11 所示。Windows 用一条假想的线将(xStart, yStart)与椭圆的中心连接，从该线与边界框的交点开始， Windows 按反时针方向，沿著椭圆画一条弧。Windows 还用另一条假想的线将(xEnd，yEnd)与椭圆的中心连接，在该线与边界框的交点处，Windows 停止画弧。

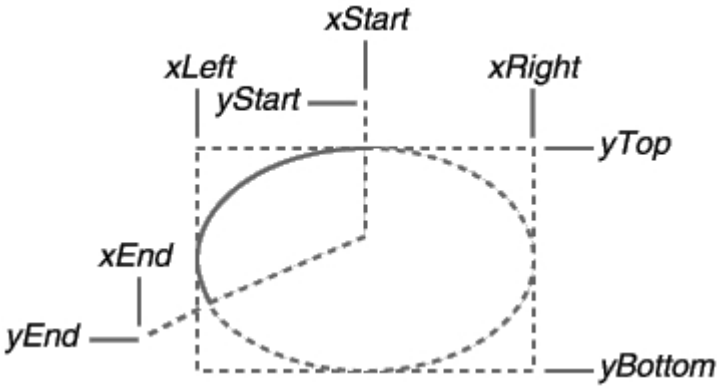


图 5-9 Arc 函式画出的线

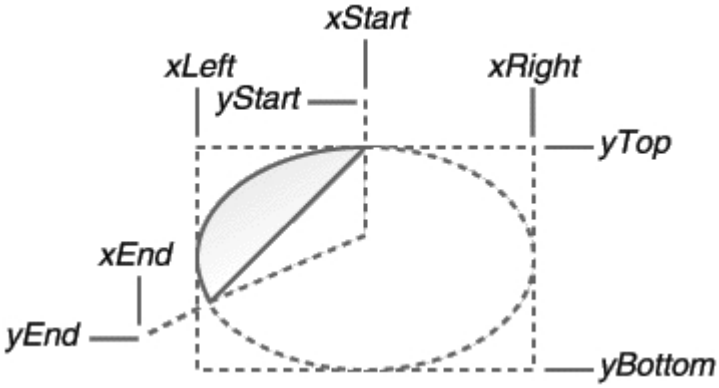


图 5-10 Chord 函式画出的线

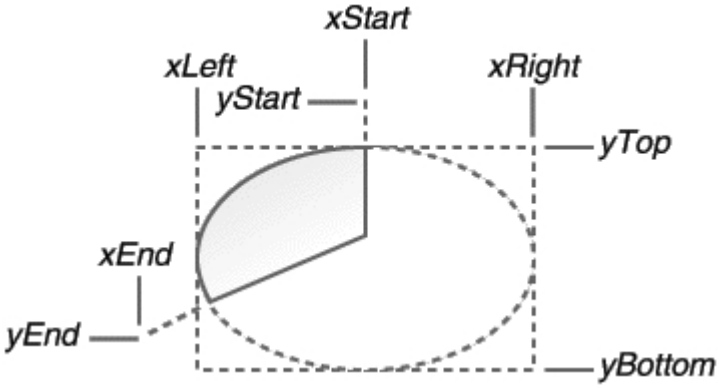


图 5-11 Pie 函式画出的线

对於 Arc 函式，这样就结束了。因为弧只是一条椭圆形的线而已，而不是一个填入区域。对於 Chord 函式，Windows 连接弧线的端点。而对於 Pie 函式，Windows 将弧的两个端点与椭圆的中心相连接。弦与扇形图的内部以目前画刷填入。

您可能不太明白在 Arc、Chord 和 Pie 函式中开始和结束位置的用法，为什

么不简单地在椭圆的周线上指定开始和结束点呢？是的，您可以这么做，但是您将不得不算出这些点。Windows 的方法在不要求这种精确性的条件下，却完成了相同的工作。

程式 5-3 LINEDEMO 画一个矩形、一个椭圆、一个圆角矩形和两条线段，不过不是按这一顺序。程式表明了定义封闭区域的函式实际上对这些区域进行了填入，因为在椭圆後面的线被遮住了，结果如图 5-12 中所示。

程式 5-3 LINEDEMO

```
LINEDEMO.C
/*------------------------------------------------------------
   LINEDEMO.C -- Line-Drawing Demonstration Program
                         (c) Charles Petzold, 1998
------------------------------------------------------------*/

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

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

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

      hwnd = CreateWindow (  szAppName, TEXT ("Line 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 message, WPARAM wParam, LPARAM lParam)
{
     static int  cxClient, cyClient ;
     HDC         hdc ;
     PAINTSTRUCT ps ;

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

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

           Rectangle (hdc,    cxClient / 8,    cyClient / 8,
                   7 * cxClient / 8, 7 * cyClient / 8) ;

           MoveToEx  (hdc,       0,       0, NULL) ;
           LineTo    (hdc, cxClient, cyClient) ;

           MoveToEx  (hdc,       0, cyClient, NULL) ;
           LineTo    (hdc, cxClient,       0) ;

           Ellipse   (hdc,    cxClient / 8,    cyClient / 8,
                   7 * cxClient / 8, 7 * cyClient / 8) ;

           RoundRect (hdc,    cxClient / 4,    cyClient / 4,
                   3 *      cxClient / 4, 3 * cyClient / 4,
                      cxClient / 4,    cyClient / 4) ;

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

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

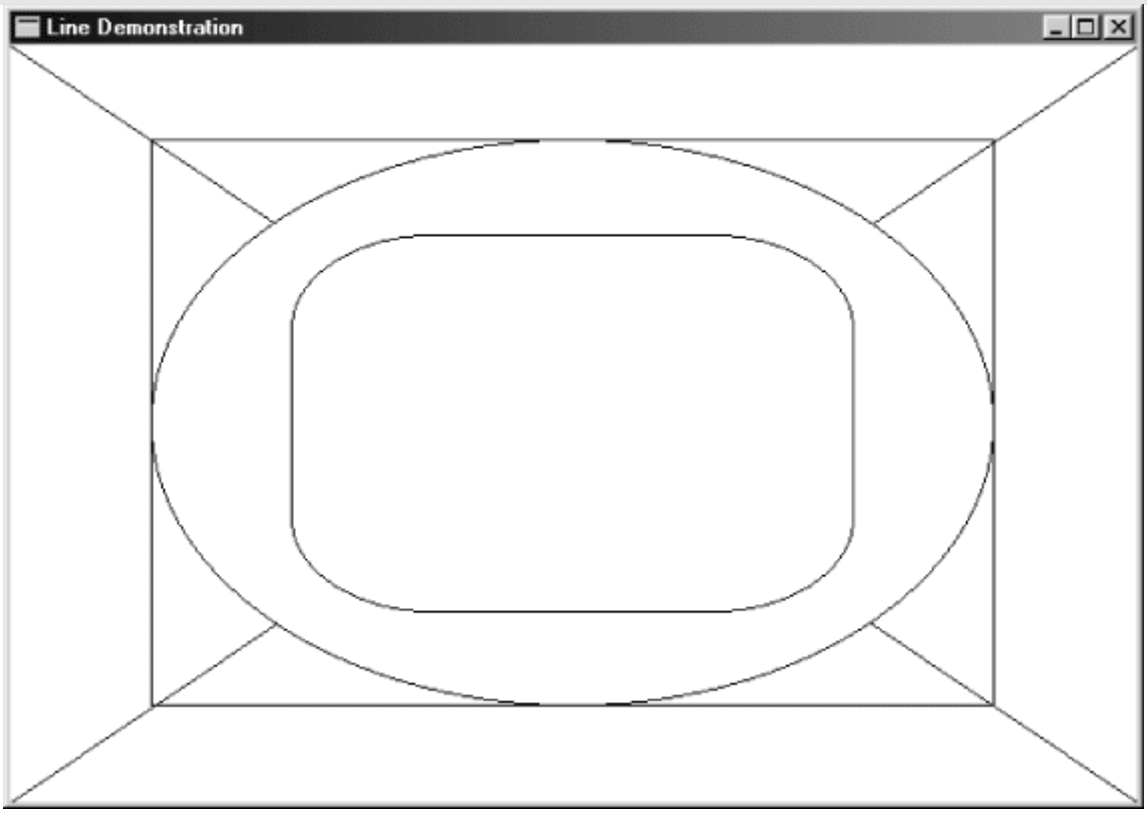


图 5-12 LINEDEMO 显示

贝塞尔曲线

「曲尺」这个词从前指的是一片木头、橡皮或者金属，用来在纸上画曲线。比如说，如果您有一些不同图点，您想要在它们之间画一条曲线（内插或者外插），您首先将这些点描在绘图纸上，然後，将曲尺定在这些点上，并用铅笔沿著曲尺绕著这些点弯曲的方向画曲线。

当然，时至今日，曲尺已经数学公式化了。有很多种不同的曲尺公式，它们各有千秋。贝塞尔曲线是电脑程式设计中用得最广的曲尺公式之一，它是直到最近才加到作业系统层次的图形支援中的。在六十年代 Renault 汽车公司进行了由手工设计车体（要用到粘土）到电脑辅助设计的转变。他们需要一些数学工具，而 Pierm Bezier 找到了一套公式，最後显示出这套公式应付这样的工作非常有用。

此後，二维的贝塞尔曲线成了电脑图学中最有用的曲线（在直线和椭圆之後）。在 PostScript 中，所有曲线都用贝塞尔曲线表示——椭圆线用贝塞尔曲

线来逼近。贝塞尔曲线也用於定义 PostScript 字体的字元轮廓（TrueType 使用一种更简单更快速的曲尺公式）。

一条二维的贝塞尔曲线由四个点定义——两个端点和两个控制点。曲线的端点在两个端点上，控制点就好像「磁石」一样把曲线从两个端点间的直线处拉走。这一点可以由底下的 BEZIER 互动交谈程式做出最好的展示，如程式 5-4 所示。

程式 5-4 BEZIER

```
BEZIER.C
/*------------------------------------------------------------------------
      BEZIER.C -- Bezier Splines Demo
                        (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 ("Bezier") ;
      HWND         hwnd ;
      MSG          msg ;
      WNDCLASS     wndclass ;

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

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

      hwnd = CreateWindow (  szAppName, TEXT ("Bezier Splines"),
                                    WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
```

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

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

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

void DrawBezier (HDC hdc, POINT apt[])
{
      PolyBezier (hdc, apt, 4) ;
      MoveToEx (hdc, apt[0].x, apt[0].y, NULL) ;
      LineTo   (hdc, apt[1].x, apt[1].y) ;

      MoveToEx (hdc, apt[2].x, apt[2].y, NULL) ;
      LineTo   (hdc, apt[3].x, apt[3].y) ;
}
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
      static POINT apt[4] ;
      HDC         hdc ;
      int         cxClient, cyClient ;
      PAINTSTRUCT  ps ;

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

                  apt[0].x = cxClient / 4 ;
                  apt[0].y = cyClient / 2 ;

                  apt[1].x = cxClient / 2 ;
                  apt[1].y = cyClient / 4 ;

                  apt[2].x = cxClient / 2 ;
                  apt[2].y = 3 * cyClient / 4 ;

                  apt[3].x = 3 * cxClient / 4 ;
                  apt[3].y = cyClient / 2 ;

                  return 0 ;
```

```
     case WM_LBUTTONDOWN:
     case WM_RBUTTONDOWN:
     case WM_MOUSEMOVE:
          if (wParam & MK_LBUTTON || wParam & MK_RBUTTON)
          {
               hdc = GetDC (hwnd) ;
               SelectObject (hdc, GetStockObject (WHITE_PEN)) ;
               DrawBezier (hdc, apt) ;

               if (wParam & MK_LBUTTON)
               {
                    apt[1].x = LOWORD (lParam) ;
                    apt[1].y = HIWORD (lParam) ;
               }

           if (wParam & MK_RBUTTON)
               {
                         apt[2].x = LOWORD (lParam) ;
                         apt[2].y = HIWORD (lParam) ;
               }

               SelectObject (hdc, GetStockObject (BLACK_PEN)) ;
               DrawBezier (hdc, apt) ;
               ReleaseDC (hwnd, hdc) ;
          }
               return 0 ;
     case  WM_PAINT:
          InvalidateRect (hwnd, NULL, TRUE) ;

          hdc = BeginPaint (hwnd, &ps) ;

          DrawBezier (hdc, apt) ;

          EndPaint (hwnd, &ps) ;
          return 0 ;

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

由於这个程式要用到一些在第七章才讲的滑鼠处理方式，所以我不在这里讨论它的内部运作（不过，这也是简单的），而是用这个程式来实验性地操纵贝塞尔曲线。在这个程式中，两个顶点设定在显示区域的上下居中、左右位於

1/4 和 3/4 处的位置；两个控制点可以改变，按住滑鼠左键或右键并拖动滑鼠可以分别改动两个控制点之一。图 5-13 是一个典型的例子。

除了贝塞尔曲线本身，程式还从第一个控制点向左边的第一个端点（也叫做开始点）画一条直线，并从第二个控制点向右边的端点画一条直线。

由於下面几个特点，贝塞尔曲线在电脑辅助设计中非常有用。首先，经过少量练习，就可以把曲线调整到与想要的形状非常接近。

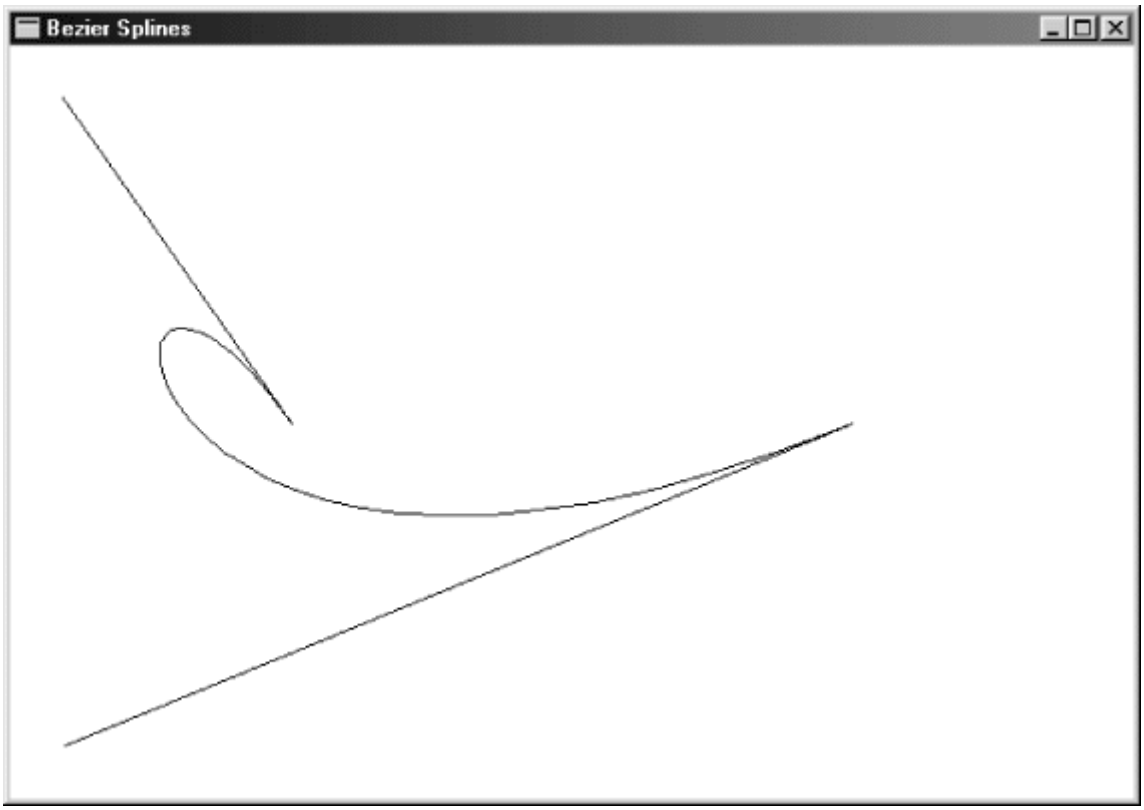


图 5-13 BEZIER 程式的显示

其次，贝塞尔曲线非常好控制。对於有的曲尺种类来说，曲线不经过任何一个定义该曲线的点。贝塞尔曲线总是由其两个端点开始和结束的（这是在推导贝塞尔公式时所做的假设之一）。另外，有些形式的曲尺公式有奇异点，在这些点处曲线趋向无穷远，这在电脑辅助设计中通常是很不合适的。事实上，贝塞尔曲线总是受限於一个四边形（叫做「凸包」），这个四边形由端点和控制点连接而成。

第三个特点涉及端点和控制点之间的关系。曲线总是与第一个控制点到起点的直线相切，并保持同一方向；同时，也与第二个控制点到终点的直线相切，并保持同一方向。这是用於推导贝塞尔公式时所做的另外两个假设。

第四，贝塞尔曲线通常比较具有美感。我知道这是一个主观评价的问题，不过，并非只有我才这样想。

在 32 位元的 Windows 版本之前，您必须利用 Polyline 来自己建立贝塞尔曲线，并且还需要知道下面的贝塞尔曲线的参数方程。起点是(x0,y0)，终点

是(x3,y3)，两个控制点是(x1,y1)和(x2,y2)，随著 t 的值从 0 到 1 的变化，就可以画出曲线：

```
x(t) = (1 - t)3 x0 + 3t (1 - t)2 x1 + 3t2 (1 - t) x2 + t3 x3
y(t) = (1 - t)3 y0 + 3t (1 - t)2 y1 + 3t2 (1 - t) y2 + t3 y3
```

在 Windows 98 中，您不需要知道这些公式。要画一条或多条连接的贝塞尔曲线，只需呼叫：

```
PolyBezier (hdc, apt, iCount) ;
```

或

```
PolyBezierTo (hdc, apt, iCount) ;
```

两种情况下，apt 都是 POINT 结构的阵列。对 PolyBezier，前四个点（按照顺序）给出贝塞尔曲线的起点、第一个控制点、第二个控制点和终点。此後的每一条贝塞尔曲线只需给出三个点，因为後一条贝塞尔曲线的起点就是前一条贝塞尔曲线的终点，如此类推。iCount 参数等於 1 加上您所绘制的这些首尾相接曲线条数的三倍。

PolyBezierTo 函式使用目前点作为第一个起点，第一条以及後续的贝塞尔曲线都只需要给出三个点。当函式传回时，目前点设定为最後一个终点。

一点提示：在画一系列相连的贝塞尔曲线时，只有当第一条贝塞尔曲线的第二个控制点、第一条贝塞尔曲线的终点（也就是第二条曲线的起点）和第二条贝塞尔曲线的第一个控制点线性相关时，也就是说这三个点在同一条直线上时，曲线在连接点处才是光滑的。

使用现有画笔（Stock Pens）

当您呼叫这一节中讨论的任何画线函式时，Windows 使用装置内容中目前选中的「画笔」来画线。画笔决定线的色彩、宽度和画笔样式，画笔样式可以是实线、点划线或者虚线，内定装置内容中画笔为 BLACK_PEN。不管映射方式是什么，这种画笔都画出一个图素宽的黑色实线来。BLACK_PEN 是 Windows 提供的三种现有画笔之一，其他两种是 WHITE_PEN 和 NULL_PEN，NULL_PEN 什么都不画。您也可以自己自订画笔。

Windows 程式以代号来使用画笔。 Windows 表头档案 WINDEF.H 中包含一个叫做 HPEN 的型态定义，即画笔的代号，可以定义这个型态的变数（例如 hPen）：

```
HPEN hPen ;
```

呼叫 GetStockObject，可以获得现有画笔的代号。例如，假设您想使用名为 WHITE_PEN 的现有画笔，可以如下取得画笔的代号：

```
hPen = GetStockObject (WHITE_PEN) ;
```

现在必须将画笔选进装置内容：

```
SelectObject (hdc, hPen) ;
```

目前的画笔是白色。在这个呼叫後，您画的线将使用 WHITE_PEN，直到您将另外一个画笔选进装置内容或者释放装置内容代号为止。

您也可以不定义 hPen 变数，而将 GetStockObject 和 SelectObject 呼叫合并成一个叙述：

```
SelectObject (hdc, GetStockObject (WHITE_PEN)) ;
```

如果想恢复到使用 BLACK_PEN 的状态，可以用一个叙述取得这种画笔的代号，并将其选进装置内容：

```
SelectObject (hdc, GetStockObject (BLACK_PEN)) ;
```

SelectObject 的传回值是此呼叫前装置内容中的画笔代号。如果启动一个新的装置内容并呼叫

```
hPen = SelectObject (hdc, GetStockobject (WHITE_PEN)) ;
```

则装置内容中的目前画笔将为 WHITE_PEN，变数 hPen 将会是 BLACK_PEN 的代号。以後通过呼叫

```
SelectObject (hdc, hPen) ;
```

就能够将 BLACK_PEN 选进装置内容。

画笔的建立、选择和删除

尽管使用现有画笔非常方便，但却受限於实心的黑画笔、实心的白画笔或者没有画笔这三种情况。如果想得到更丰富多彩的效果，就必须建立自己的画笔。

这一过程通常是：使用函式 CreatePen 或 CreatePenIndirect 建立一个「逻辑画笔」，这仅仅是对画笔的描述。这些函式传回逻辑画笔的代号；然後，呼叫 SelectObject 将画笔选进装置内容。现在，就可以使用新的画笔来画线了。在任何时候，都只能有一种画笔选进装置内容。在释放装置内容（或者在选择了另一种画笔到装置内容中）之後，就可以呼叫 DeleteObject 来删除所建立的逻辑画笔了。在删除後，该画笔的代号就不再有效了。

逻辑画笔是一种「GDI 物件」，它是您可以建立的六种 GDI 物件之一，其他五种是画刷、点阵图、区域、字体和调色盘。除了调色盘之外，这些物件都是通过 SelectObject 选进装置内容的。

在使用画笔等 GDI 物件时，应该遵守以下三条规则：

- 最後要删除自己建立的所有 GDI 物件。
- 当 GDI 物件正在一个有效的装置内容中使用时，不要删除它。
- 不要删除现有物件。

这些规则当然是有道理的，而且有时这道理还挺微妙的。下面我们将举些例子来帮助理解这些规则。

CreatePen 函式的语法形如：

```
hPen = CreatePen (iPenStyle, iWidth, crColor) ;
```

其中，iPenStyle 参数确定画笔是实线、点线还是虚线，该参数可以是 WINGDI.H 表头档案中定义的以下识别字，图 5-14 显示了每种画笔产生的画笔样式。

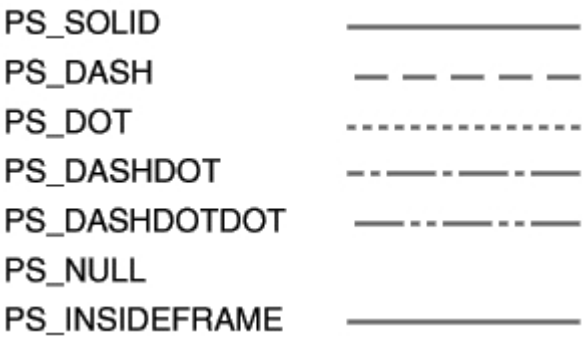


图 5-14 七种画笔样式

对於 PS_SOLID、PS_NULL 和 PS_INSIDEFRAME 画笔样式，iWidth 参数是画笔的宽度。iWidth 值为 0 则意味著画笔宽度为一个图素。现有画笔是一个图素宽。如果指定的是点划线或者虚线式画笔样式，同时又指定一个大於 1 的实际宽度，那么 Windows 将使用实线画笔来代替。

CreatePen 的 crColor 参数是一个 COLORREF 值，它指定画笔的颜色。对於除了 PS_INSIDEFRAME 之外的画笔样式，如果将画笔选入装置内容中，Windows 会将颜色转换为设备所能表示的最相近的纯色。PS_INSIDEFRAME 是唯一一种可以使用混色的画笔样式，并且只有在宽度大於 1 的情况下才如此。

在与定义一个填入区域的函式一起使用时，PS_INSIDEFRAME 画笔样式还有另外一个奇特之处：对於除了 PS_INSIDEFRAME 以外的所有画笔样式来说，如果用来画边界框的画笔宽度大於 1 个图素，那么画笔将居中对齐在边界框线上，这样边界框线的一部分将位於边界框之外；而对於 PS_INSIDEFRAME 画笔样式来说，整条边界框线都画在边界框之内。

您也可以通过建立一个型态为 LOGPEN（「逻辑画笔」）的结构，并呼叫 CreatePenIndirect 来建立画笔。如果您的程式使用许多能在原始码中初始化的画笔，那么使用这种方法将有效得多。

要使用 CreatePenIndirect，首先定义一个 LOGPEN 型态的结构：

```
LOGPEN logpen ;
```

此结构有三个成员：lopnStyle（无正负号整数或 UINT）是画笔样式，lopnWidth（POINT 结构）是按逻辑单位度量的画笔宽度，lopnColor (COLORREF) 是画笔颜色。Windows 只使用 lopnWidth 结构的 x 值作为画笔宽度，而忽略 y 值。

将结构的位址传递给 CreatePenIndirect 结构就可以建立画笔了：

```
hPen = CreatePenIndirect (&logpen) ;
```

注意，CreatePen 和 CreatePenIndirect 函式不需要装置内容代号作为参数。这些函式建立与装置内容没有联系的逻辑画笔。直到呼叫 SelectObject 之後，画笔才与装置内容发生联系。因此，可以对不同的设备（如萤幕和印表机）使用相同的逻辑画笔。

下面是建立、选择和删除画笔的一种方法。假设您的程式使用三种画笔——一种宽度为 1 的黑画笔、一种宽度为 3 的红画笔和一种黑色点式画笔，您可以先定义三个变数来存放这些画笔的代号：

```
static HPEN hPen1, hPen2, hPen3 ;
```

在处理 WM_CREATE 期间，您可以建立这三种画笔：

```
hPen1 = CreatePen (PS_SOLID, 1, 0) ;
hPen2 = CreatePen (PS_SOLID, 3, RGB (255, 0, 0)) ;
hPen3 = CreatePen (PS_DOT, 0, 0) ;
```

在处理 WM_PAINT 期间，或者是在拥有一个装置内容有效代号的任何时间里，您都可以将这三个画笔之一选进装置内容并用它来画线：

```
SelectObject (hdc, hPen2) ;
```

画线函式

```
SelectObject (hdc, hPen1) ;
```

其他画线函式

在处理 WM_DESTROY 期间，您可以删除您建立的三种画笔：

```
DeleteObject (hPen1) ;
DeleteObject (hPen2) ;
DeleteObject (hPen3) ;
```

这是建立、选择和删除画笔最直接的方法。但是您的程式必须知道执行期间需要哪些逻辑画笔，为此，您可能想要在每个 WM_PAINT 讯息处理期间建立画笔，并在呼叫 EndPaint 之後删除它们（您可以在呼叫 EndPaint 之前删除它们，但是要小心，不要删除装置内容中目前选择的画笔）。

您可能还希望随时建立画笔，并将 CreatePen 和 SelectObject 呼叫组合到同一个叙述中：

```
SelectObject (hdc, CreatePen (PS_DASH, 0, RGB (255, 0, 0))) ;
```

现在再开始画线，您将使用一个红色虚线画笔。在画完红色虚线之後，可以删除画笔。糟了！由於没有保存画笔代号，怎么才能删除这些画笔呢？不要紧，请记住，SelectObject 将传回装置内容中上一次选择的画笔代号。所以，您可以通过呼叫 SelectObject 将 BLACK_PEN 选进装置内容，并删除从 SelectObject 传回的值：

```
DeleteObject (SelectObject (hdc, GetStockObject (BLACK_PEN))) ;
```

下面是另一种方法，在将新建立的画笔选进装置内容时，保存 SelectObject 传回的画笔代号：

```
hPen = SelectObject (hdc, CreatePen (PS_DASH, 0, RGB (255, 0, 0))) ;
```

现在 hPen 是什么呢？如果这是在取得装置内容之後第一次呼叫 SelectObject，则 hPen 是 BLACK_PEN 物件的代号。现在，可以将 hPen 选进装置内容，并删除所建立的画笔（第二次 SelectObject 呼叫传回的代号），只要一道叙述即可：

```
DeleteObject (SelectObject (hdc, hPen)) ;
```

如果有一个画笔的代号，就可以通过呼叫 GetObject 取得 LOGPEN 结构各个成员的值：

```
GetObject (hPen, sizeof (LOGPEN), (LPVOID) &logpen) ;
```

如果需要目前选进装置内容的画笔代号，可以呼叫：

```
hPen = GetCurrentObject (hdc, OBJ_PEN) ;
```

在第十七章将讨论另一个建立画笔的函式 ExtCreatePen。

填入空隙

使用点式画笔和虚线画笔会产生一个有趣的问题：点和虚线之间的空隙会怎样呢？您所需要的是什么？

空隙的著色取决於装置内容的两个属性——背景模式和背景颜色。内定背景模式为 OPAQUE，在这种方式下，Windows 使用背景色来填入空隙，内定的背景色为白色。这与许多程式在视窗类别中用 WHITE_BRUSH 来擦除视窗背景的做法是一致的。

您可以通过如下呼叫来改变 Windows 用来填入空隙的背景色：

```
SetBkColor (hdc, crColor) ;
```

与画笔色彩所使用的 crColor 参数一样，Windows 将这里的背景色转换为纯色。可以通过用 GetBkColor 来取得装置内容中定义的目前背景色。

通过将背景模式转换为 TRANSPARENT，可以阻止 Windows 填入空隙：

```
SetBkMode (hdc, TRANSPARENT) ;
```

此後，Windows 将忽略背景色，并且不填入空隙，可以通过呼叫 GetBkMode 来取得目前背景模式（TRANSPARENT 或者 OPAQUE）。

绘图方式

装置内容中定义的绘图方式也影响显示器上所画线的外观。设想画这样一条直线，它的色彩由画笔色彩和画线区域原来的色彩共同决定。设想用同一种画笔在白色表面上画出黑线而在黑色表面上画出白线，而且不用知道表面是什么色彩。这样的功能对您有用吗？通过绘图方式的设定，这些都可以实作。

当 Windows 使用画笔来画线时，它实际上执行画笔图素与目标位置处原来图素之间的某种位元布林运算。图素间的位元布林运算叫做「位元映射运算」，

简称为「ROP」。由於画一条直线只涉及两种图素（画笔和目标），因此这种布林运算又称为「二元位元映射运算」，简记为「ROP2」。Windows 定义了 16 种 ROP2 代码，表示 Windows 组合画笔图素和目标图素的方式。在内定装置内容中，绘图方式定义为 R2_COPYPEN，这意味著 Windows 只是将画笔图素复制到目标图素，这也是我们通常所熟知的。此外，还有 15 种 ROP2 码。

16 种不同的 ROP2 码是怎样得来的呢？为了示范的需要，我们假设使用单色系统，目标色（视窗显示区域的色彩）为黑色（用 0 来表示）或者白色（用 1 来表示），画笔也可以为黑色或者白色。用黑色或者白色画笔在黑色或者白色目标上画图有四种组合：白笔与白目标、白笔与黑目标、黑笔与白目标、黑笔与黑目标。

画笔在目标上绘制後会得到什么呢？一种可能是不管画笔和目标的色彩，画出的线总是黑色的，这种绘图方式由 ROP2 代码 R2_BLACK 表示。另一种可能是只有当画笔与目标都为黑色时，画出的结果才是白色，其他情况下画出的都是黑色。尽管这似乎有些奇怪，Windows 还是为这种方式起了一个名字，叫做 R2_NOTMERGEPEN。Windows 执行目标图素与画笔图素的位元「或」运算，然後翻转所得色彩。

表 5-2 显示了所有 16 种 ROP2 绘图方式，表中指示了画笔色彩(P)与目标色彩(D)是如何组合而成结果色彩的。在标有「布林操作」的那一栏中，用 C 语言的表示法给出了目标图素与画笔图素的组合方式。

表 5-2

画笔 (P): 目标 (D):	1 1	1 0	0 1	0 0	布林 操作	绘图模式
结果:	0	0	0	0	0	R2_BLACK
	0	0	0	1	~(P | D)	R2_NOTMERGEPEN
	0	0	1	0	~P & D	R2_MASKNOTPEN
	0	0	1	1	~P	R2_NOTCOPYPEN
	0	1	0	0	P & ~D	R2_MASKPENNOT
	0	1	0	1	~D	R2_NOT
	0	1	1	0	P ^ D	R2_XORPEN
	0	1	1	1	~(P & D)	R2_NOTMASKPEN
	1	0	0	0	P & D	R2_MASKPEN
	1	0	0	1	~(P ^ D)	R2_NOTXORPEN
	1	0	1	0	D	R2_NOP
	1	0	1	1	~P | D	R2_MERGENOTPEN
	1	1	0	0	P	R2_COPYPEN（内定）

	1	1	0	1	P | ~D	R2_MERGEPENNOT
	1	1	1	0	P | D	R2_MERGEPEN
	1	1	1	1	1	R2_WHITE

可以通过以下呼叫在装置内容中设定新的绘图模式：

```
SetROP2 (hdc, iDrawMode) ;
```

iDrawMode 参数是表中「绘图模式」一栏中给出的值之一。您可以用函式：

```
iDrawMode = GetROP2 (hdc) ;
```

来取得目前绘图方式。装置内容中的内定设定为 R2_COPYPEN，它用画笔色彩替代目标色彩。在 R2_NOTCOPYPEN 方式下，若画笔为黑色，则画成白色；若画笔为白色，则画成黑色。R2_BLACK 方式下，不管画笔和背景色为何种色彩，总是画成黑色。与此相反，R2_WHITE 方式下总是画成白色。R2_NOP 方式就是「不操作」，让目标保持不变。

现在，我们已经讨论了单色系统。然而，大多数系统是彩色的。在彩色系统中，Windows 为画笔和目标图素的每个颜色位元执行绘图方式的位元运算，并再次使用上表描述的 16 种 ROP2 代码。R2_NOT 绘图方式总是翻转目标色彩来决定线的颜色，而不管画笔的色彩是什么。例如，在青色目标上的线会变成紫色。R2_NOT 方式总是产生可见的画笔，除非画笔在中等灰度的背景上绘图。我将在第七章的 BLOKOUT 程式中展示 R2_NOT 绘图方式的使用。

绘制填入区域

现在再更进一步，从画线到画图形。Windows 中七个用来画带边缘的填入图形的函式列於表 5-3 中。

表 5-3

函式	图形
Rectangle	直角矩形
Ellipse	椭圆
RoundRect	圆角矩形
Chord	椭圆周上的弧，两端以弦连接
Pie	椭圆上的圆形图
Polygon	多边形
PolyPolygon	多个多边形

Windows 用装置内容中选择的目前画笔来画图形的边界框，边界框还使用目前背景方式、背景色彩和绘图方式，这跟 Windows 画线时一样。关於直线的一切也适用於这些图形的边界框。

图形以目前装置内容中选择的画刷来填入。内定情况下，使用现有物件，这意味著图形内部将画为白色。Windows 定义六种现有画刷：WHITE_BRUSH、LTGRAY_BRUSH、GRAY_BRUSH、DKGRAY_BRUSH、BLACK_BRUSH 和 NULL_BRUSH （也叫 HOLLOW_BRUSH）。您可以将任何一种现有画刷选入您的装置内容中，就和您选择一种画笔一样。Windbws 将 HBRUSH 定义为画刷的代号，所以可以先定义一个画刷代号变数：

```
HBRUSH hBrush ;
```

您可以通过呼叫 GetStockObject 来取得 GRAY_BRUSH 的代号：

```
hBrush = GetStockObject (GRAY_BRUSH) ;
```

您可以呼叫 SelectObject 将它选进装置内容：

```
SelectObject (hdc, hBrush) ;
```

现在，如果您要画上表中的任一个图形，则其内部将为灰色。

如果您想画一个没有边界框的图形，可以将 NULL_PEN 选进装置内容：

```
SelectObject (hdc, GetStockObject (NULL_PEN)) ;
```

如果您想画出图形的边界框，但不填入内部，则将 NULL_BRUSH 选进装置内容：

```
SelectObject (hdc, GetStockobject (NULL_BRUSH) ;
```

您也可以自订画刷，就如同您自订画笔一样。我们将马上谈到这个问题。

Polygon 函式和多边形填入方式

我已经讨论过了前五个区域填入函式，Polygon 是第六个画带边界框的填入图形的函式，该函式的呼叫与 Polyline 函式相似：

```
Polygon (hdc, apt, iCount) ;
```

其中，apt 参数是 POINT 结构的一个阵列，iCount 是点的数目。如果该阵列中的最後一个点与第一个点不同，则 Windows 将会再加一条线，将最後一个点与第一个点连起来（在 Polyline 函式中，Windows 不会这么做）。PolyPolygon 函式如下所示：

```
PolyPolygon (hdc, apt, aiCounts, iPolyCount) ;
```

该函式绘制多个多边形。最後一个参数给出了所画的多边形的个数。对於每个多边形，aiCounts 阵列给出了多边形的端点数。apt 阵列具有全部多边形的所有点。除传回值以外，PolyPolygon 在功能上与下面的代码相同：

```
for (i = 0, iAccum = 0 ; i < iPolyCount ; i++)
{
      Polygon (hdc, apt + iAccum, aiCounts[i]) ;
      iAccum += aiCounts[i] ;
}
```

对於 Polygon 和 PolyPolygon 函式，Windows 使用定义在装置内容中的目前

画刷来填入这个带边界的区域。至於填入内部的方式，则取决於多边形填入方式，您可以用 SetPolyFillMode 函式来设定：

```
SetPolyFillMode (hdc, iMode) ;
```

内定情况下，多边形填入方式是 ALTERNATE，但是您可以将它设定为 WINDING。两种方式的区别参见图 5-15 所示。

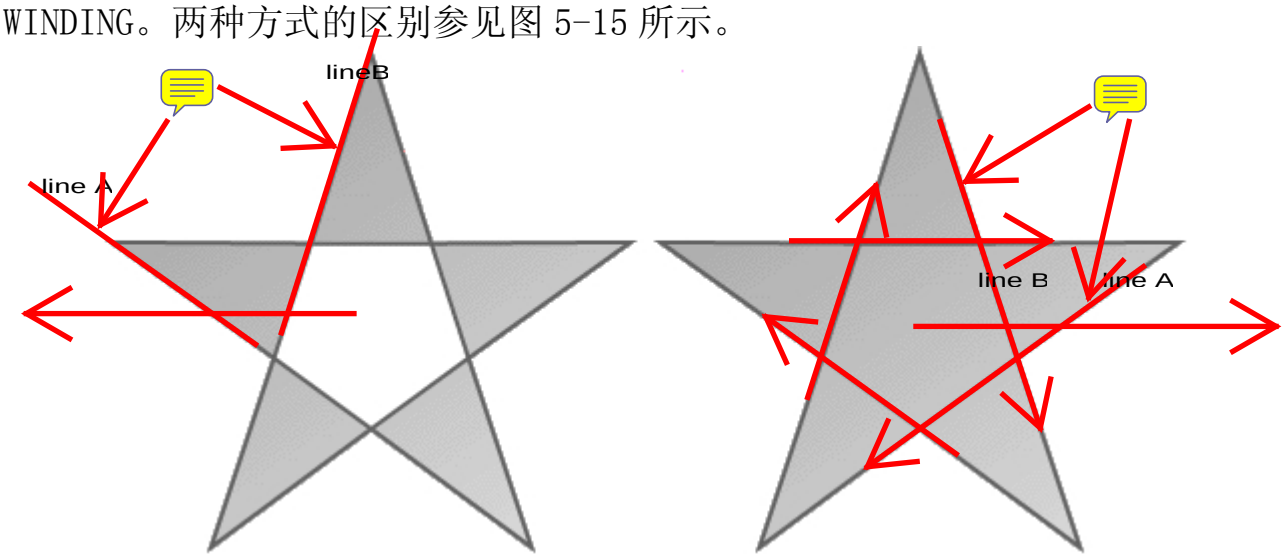


图 5-15 用两种多边形填入方式画出的图：ALTERNATE（左）和 WINDING（右）

首先，ALTERNATE 和 WINDING 方式之间的区别很容易察觉。对於 ALTERNATE 方式，您可以设想从一个无穷大的封闭区域内部的点画线，只有假想的线穿过了奇数条边界线时，才填入封闭区域。这就是填入了星的角而中心没被填入的原因。

五角星的例子使得 WINDING 方式看起来比实际上更简单一些。在绘制单个的多边形时， 大多数情况下，WINDING 方式会填入所有封闭的区域。但是也有例外。

在 WINDING 方式下要确定一个封闭区域是否被填入，您仍旧可以设想从那个无穷大的区域画线。如果假想的线穿过了奇数条边界线，区域就被填入，这和 ALTERNATE 方式一样。如果假想的线穿过了偶数条边界线，则区域可能被填入也可能不被填入。如果一个方向（相对於假想线）的边界线数与另一个方向的边界线数不相等，就填入区域。

例如，考虑图 5-16 中的物体。线上的箭头指出了画线的方向。两种方式都会填入三个封闭的 L 形区域，号码从 1 到 3。号码为 4 和 5 的两个小内部区域，在 ALTERNATE 方式下不会被填入。但是，在 WINDING 方式下，号码为 5 的区域会被填入，因为从区域内必须穿过两条相同方向的线才能到达图形外部。号码为 4 的区域不会被填入，因为必须穿过两条方向相反的线。

如果您怀疑 Windows 没有这么聪明，那么程式 5-5 ALTWIND 会展示给您看。

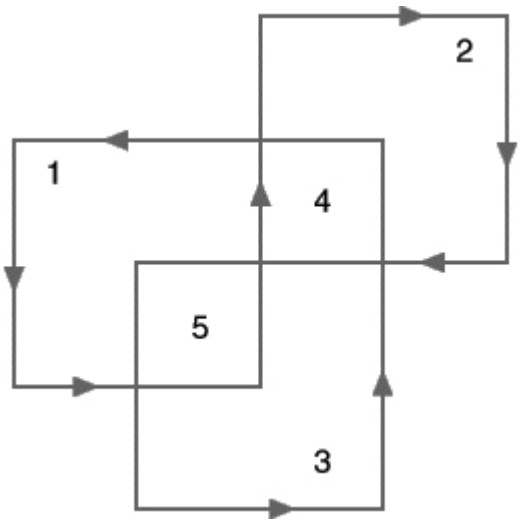


图 5-16 WINDING 方式不能填入所有内部区域的图形

程式 5-5 ALTWIND

```
ALTWIND.C
/*------------------------------------------------------------------------
      ALTWIND.C --      Alternate and Winding Fill Modes
                        (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 ("AltWind") ;
      HWND        hwnd ;
      MSG         msg ;
      WNDCLASS    wndclass ;
      wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc= WndProc ;
      wndclass.cbClsExtra    = 0 ;
      wndclass.cbWndExtra    = 0 ;
      wndclass.hInstance     = hInstance ;
      wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
      wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
      wndclass.lpszMenuName  = NULL ;
      wndclass.lpszClassName = szAppName ;

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

```
     hwnd = CreateWindow (szAppName, TEXT ("Alternate and Winding Fill Modes"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

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

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

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static POINT aptFigure [10] = {10,70, 50,70, 50,10, 90,10, 90,50,
                                    30,50, 30,90, 70,90, 70,30, 10,30 };
     static int  cxClient, cyClient ;
     HDC         hdc ;
     int         i ;
     PAINTSTRUCT ps ;
     POINT       apt[10] ;

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

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

          SelectObject (hdc, GetStockObject (GRAY_BRUSH)) ;

          for (i = 0 ; i < 10 ; i++)
          {
               apt[i].x = cxClient * aptFigure[i].x / 200 ;
               apt[i].y = cyClient * aptFigure[i].y / 100 ;
          }

          SetPolyFillMode (hdc, ALTERNATE) ;
          Polygon (hdc, apt, 10) ;
```

```
            for (i = 0 ; i < 10 ; i++)
            {
                  apt[i].x += cxClient / 2 ;
            }

            SetPolyFillMode (hdc, WINDING) ;
            Polygon (hdc, apt, 10) ;

            EndPaint (hwnd, &ps) ;
            return 0 ;

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

图形的座标（划分为 100 100 个单位）储存在 aptFigure 阵列中。这些座标是依据显示区域的宽度和高度划分的。程式显示图形两次，一次使用 ALTERNATE 填入方式，另一次使用 WINDING 方式。结果见图 5-17。

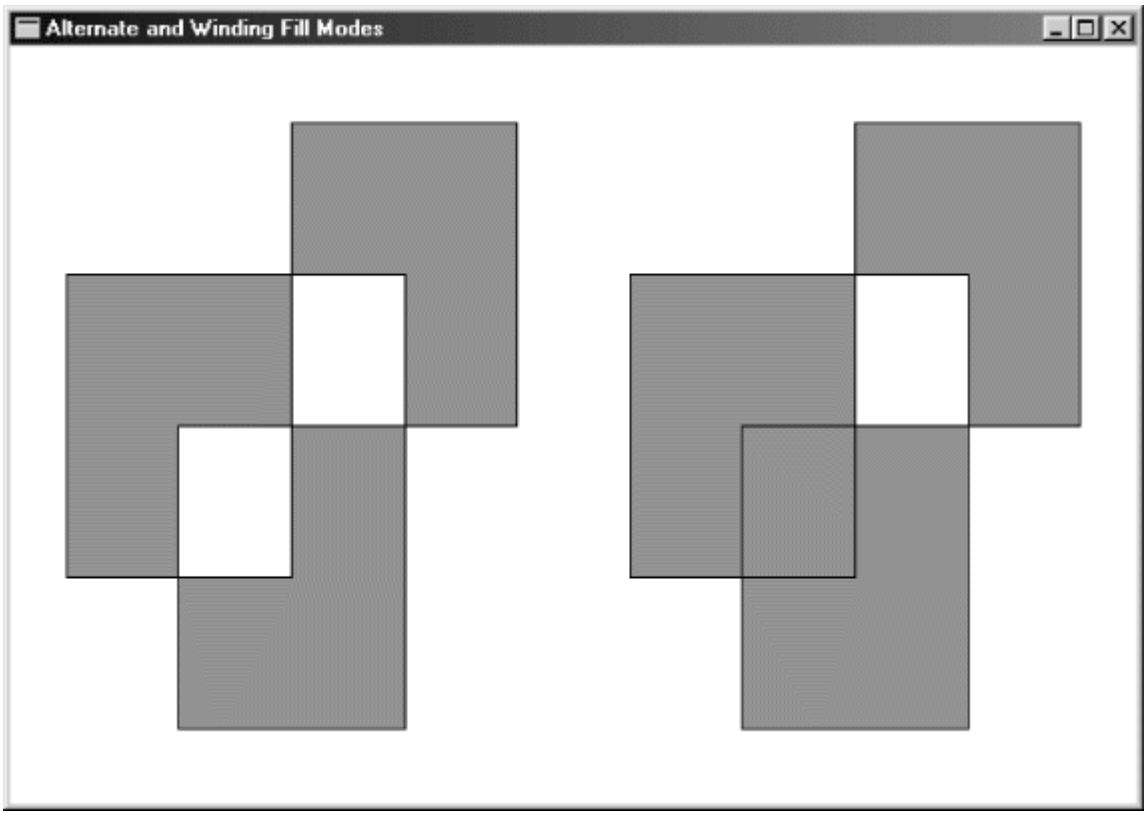


图 5-17 ALTWIND 的显示

用画刷填入内部

Rectangle、RoundRect、Ellipse、Chord、Pie、Polygon 和 PolyPolygon 图形的内部是用选进装置内容的目前画刷（也称为「图样」）来填入的。画刷

是一个 8 8 的点阵图，它水平和垂直地重复使用来填入内部区域。

当 Windows 用混色的方法来显示多於可从显示器上得到的色彩时，实际上是将画刷用於色彩。在单色系统上，Windows 能够使用黑色和白色图素的混色建立 64 种不同的灰色，更精确地说，Windows 能够建立 64 种不同的单色画刷。对於纯黑色，8 8 点阵图中的所有位元均为 0。第一种灰色有一位元为 1，第二种灰色有两位元为 1，以此类推，直到 8 8 点阵图中所有位元均为 1，这就是白色。在 16 色或 256 色显示系统上，混色也是点阵图，并且可以得到更多的色彩。

Windows 还有五个函式，可以让您建立逻辑画刷，然後就可使用 SelectObject 将画刷选进装置内容。与逻辑画笔一样，逻辑画刷也是 GDI 物件。您建立的所有画刷都必须被删除，但是当它还在装置内容中时不能将其删除。

下面是建立逻辑画刷的第一个函式：

```
hBrush = CreateSolidBrush (crColor) ;
```

函式中的 Solid 并不是指画刷为纯色。在将画刷选入装置内容中时，Windows 建立一个混色色的点阵图，并为画刷使用该点阵图。

您还可以使用由水平、垂直或者倾斜的线组成的「影线标记(hatch marks)」来建立画刷，这种风格的画刷对著色条形图的内部和在绘图机上进行绘图最有用。建立影线画刷的函式为：

```
hBrush = CreateHatchBrush (iHatchStyle, crColor) ;
```

iHatchStyle 参数描述影线标记的外观。图 5-18 显示了六种可用的影线标记风格。

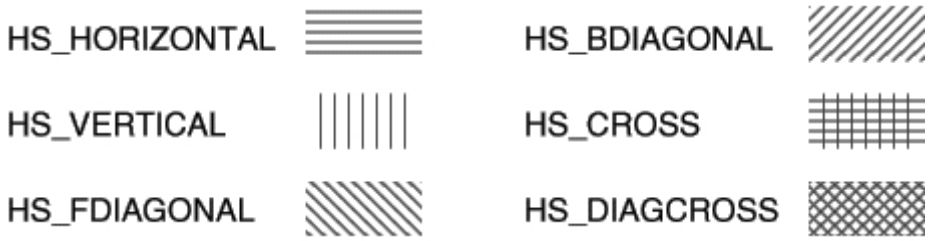


图 5-18 六种影线画刷风格

CreateHatchBrush 中的 crColor 参数是影线的色彩。在将画刷选进装置内容时，Windows 将这种色彩转换为与之最相近的纯色。影线之间的区域根据装置内容中定义的背景方式和背景色来著色。如果背景方式为 OPAQUE，则用背景色（它也被转换为纯色）来填入线之间的空间。在这种情况下，影线和填入色都不能是混色而成的颜色。如果背景方式为 TRANSPARENT，则 Windows 只画出影线，不填入它们之间的区域。

您也可以使用 CreatePatternBrush 和 CreateDIBPatternBrushPt 建立自己的点阵图画刷。

建立逻辑画刷的第五个函式包含其他四个函式：

```
hBrush = CreateBrushIndirect (&logbrush) ;
```

变数 logbrush 是一个型态为 LOGBRUSH（「逻辑画刷」）的结构，该结构的三个栏位如表 5-4 所示，lbStyle 栏位的值确定了 Windows 如何解释其他两个栏位的值：

表 5-4

lbStyle (UINT)	lbColor (COLORREF)	lbHatch (LONG)
BS_SOLID	画刷的色彩	忽略
BS_HOLLOW	忽略	忽略
BS_HATCHED	影线的色彩	影线画刷风格
BS_PATTERN	忽略	点阵图的代号
BS_DIBPATTERNPT	忽略	指向 DIB 的指标

前面我们用 SelectObject 将逻辑画笔选进装置内容，用 DeleteObject 删除画笔，用 GetObject 来取得逻辑画笔的资讯。对於画刷，同样能使用这三个函式。一旦您取得到了画刷代号，就可以使用 SelectObject 将该画刷选进装置内容：

```
SelectObject (hdc, hBrush) ;
```

然後，您可以使用 DeleteObject 函式删除所建立的画刷：

```
DeleteObject (hBrush) ;
```

但是，不要删除目前选进装置内容的画刷。

如果您需要取得画刷的资讯，可以呼叫 GetObject：

```
GetObject (hBrush, sizeof (LOGBRUSH), (LPVOID) &logbrush) ;
```

其中，logbrush 是一个型态为 LOGBRUSH 的结构。

GDI 映射方式

到目前为止，所有的程式都是相对於显示区域的左上角，以图素为单位绘图的。这是内定情况，但不是唯一选择。事实上，「映射方式」是一种几乎影响任何显示区域绘图的装置内容属性。另外有四种装置内容属性——视窗原点、视埠原点、视窗范围和视埠范围——与映射方式密切相关。

大多数 GDI 绘图函式需要座标值或大小。例如，下面是 TextOut 函式：

```
TextOut (hdc, x, y, psText, iLength) ;
```

参数 x 和 y 分别表示文字的开始位置。参数 x 是在水平轴上的位置，参数 y 是在垂直轴上的位置，通常用(x,y)来表示这个点。

在 TextOut 中，以及在几乎所有 GDI 函式中，这些座标值使用的都是一种「逻辑单位」。Windows 必须将逻辑单位转换为「装置单位」，即图素。这种转换是由映射方式、视窗和视埠的原点以及视窗和视埠的范围所控制的。映射方式还指示著 x 轴和 y 轴的方向(orientation)；也就是说，它确定了当您在向显

示器的左或者右移动时 x 的值是增大还是减小，以及在上下移动时 y 的值是增大还是减小。

Windows 定义了 8 种映射方式，它们在 WINGDI.H 中相应的识别字和含义如表 5-5 所示。

表 5-5

映射方式	逻辑单位	增加值	
		x 值	y 值
MM_TEXT	图素	右	下
MM_LOMETRIC	0.1 mm	右	上
MM_HIMETRIC	0.01 mm	右	上
MM_LOENGLISH	0.01 in.	右	上
MM_HIENGLISH	0.001 in.	右	上
MM_TWIPS	1/1440 in.	右	上
MM_ISOTROPIC	任意(x = y)	可选	可选
MM_ANISOTROPIC	任意(x != y)	可选	可选

METRIC 和 ENGLISH 指一般通行的度量衡系统，点是印刷的测量单位，约等於 1/72 英寸，但在图形程式设计中假定为正好 1/72 英寸。「Twip」等於 1/20 点，也就是 1/1440 英寸。「Isotropic」和「anisotropic」是真正的单字，意思是「等方性」（同方向）和「异方性」（不同方向）。

您可以使用下面的叙述来设定映射方式：

```
SetMapMode (hdc, iMapMode) ;
```

其中，iMapMode 是 8 个映射方式识别字之一。您可以通过以下呼叫取得目前的映射方式：

```
iMapMode = GetMapMode (hdc) ;
```

内定映射方式为 MM_TEXT。在这种映射方式下，逻辑单位与实际单位相同，这样我们可以直接以图素为单位进行操作。在 TextOut 呼叫中，它看起来像这样：

```
TextOut (hdc, 8, 16, TEXT ("Hello"), 5) ;
```

文字从距离显示区域左端 8 图素、上端 16 图素的位置处开始。

如果映射方式设定为 MM_LOENGLISH：

```
SetMapMode (hdc, MM_LOENGLISH) ;
```

则逻辑单位是百分之一。现在，TextOut 呼叫如下：

```
TextOut (hdc, 50, -100, TEXT ("Hello"), 5) ;
```

文字从距离显示区域左端 0.5 英寸、上端 1 英寸的位置处开始。至於 y 座标前面的负号，随著我们对映射方式更详细的讨论，将逐渐清楚。其他映射方

式允许程式按照毫米、印表机的点大小或者任意单位的座标轴来指定座标。

如果您认为使用图素进行工作很合适，那么就不要使用内定的 MM_TEXT 方式外的任何映射方式。如果需要以英寸或者毫米尺寸显示图像，那么可以从 GetDeviceCaps 中取得所需要的资讯，自己再进行缩放。其他映射方式都是避免您自己进行缩放的一个方便途径而已。

虽然您在 GDI 函式中指定的座标是 32 位元的值，但是仅有 Windows NT 能够处理全 32 位元。在 Windows 98 中，座标被限制为 16 位元，范围从-32,768 到 32,767。一些使用座标表示矩形的开始点和结束点的 Windows 函式也要求矩形的宽和高小於或者等於 32,767。

装置座标和逻辑座标

您也许会问：如果使用 MM_LOENGLISH 映射方式，是不是将会得到以百分之一英寸为单位的 WM_SIZE 讯息呢？绝对不会。Windows 对所有讯息（如 WM_MOVE、WM_SIZE 和 WM_MOUSEMOVE），对所有非 GDI 函式，甚至对一些 GDI 函式，永远使用装置座标。可以这样来考虑：由於映射方式是一种装置内容属性，所以，只有对需要装置内容代号作参数的 GDI 函式，映射方式才会起作用。GetSystemMetrics 不是 GDI 函式，所以它总是以装置单位（即图素）为量度来传回大小的。尽管 GetDeviceCaps 是 GDI 函式，需要一个装置内容代号作为参数，但是 Windows 仍然对 HORZRES 和 VERTRES 以装置单位作为传回值，因为该函式的目的之一就是给程式提供以图素为单位的设备大小。

不过，从 GetTextMetrics 呼叫中传回的 TEXTMETRIC 结构的值是使用逻辑单位的。如果在进行此呼叫时映射方式为 MM_LOENGLISH，则 GetTextMetrics 将以百分之一英寸为单位提供字元的宽度和高度。在呼叫 GetTextMetrics 以取得关於字元的宽度和高度资讯时，映射方式必须设定成根据这些资讯输出文字时所使用的映射方式，这样就可以简化工作。

装置座标系

Windows 将 GDI 函式中指定的逻辑座标映射为装置座标。在讨论以各种不同的映射方式使用逻辑座标系之前，我们先来看一下 Windows 为视讯显示器区域定义的不同的装置座标系。尽管我们大多数时间在视窗的显示区域内工作，但 Windows 在不同的时间使用另外两种装置座标区域。所有装置座标系都以图素为单位，水平轴（即 x 轴）上的值从左到右递增，垂直轴（即 y 轴）上的值从上到下递增。

当我们使用整个萤幕时，就根据「萤幕座标」进行操作。萤幕的左上角为

(0,0)点，萤幕座标用在 WM_MOVE 讯息（对於非子视窗）以及下列 Windows 函式中：CreateWindow 和 MoveWindow（都是对於非子视窗）、GetMessagePos、GetCursorPos、SetCursorPos、GetWindowRect 以及 WindowFromPoint（这不是全部函式的列表）。它们或者是与视窗无关的函式（如两个游标函式），或者是必须相对於某个萤幕点来移动（或者寻找）视窗的函式。如果以 DISPLAY 为参数呼叫 CreateDC，以取得整个萤幕的装置内容，则内定情况下 GDI 呼叫中指定的逻辑座标将被映射为萤幕座标。

「全视窗座标」 以程式的整个视窗为基准，如标题列、功能表、卷动列和视窗框都包括在内。而对於普通视窗，点（0,0）是缩放边框的左上角。全视窗座标在 Windows 中极少使用，但是如果用 GetWindowDC 取得装置内容，GDI 函式中的逻辑座标就会转换为显示区域座标。

第三种座标系是我们最常使用的「显示区域座标系」。点（0,0）是显示区域的左上角。当使用 GetDC 或 BeginPaint 取得装置内容时，GDI 函式中的逻辑座标就会内定转换为显示区域座标。

用函式 ClientToScreen 和 ScreenToClient 可以将显示区域座标转换为萤幕座标，或者反过来，将萤幕座标转换为显示区域座标。也可以使用 GetWindowRect 函式取得萤幕座标下的整个视窗的位置和大小。这三个函式为一种装置座标转换为另一种提供了足够的资讯。

视埠和视窗

映射方式定义了 Windows 如何将 GDI 函式中指定的逻辑座标映射为装置座标，这里的装置座标系取决於您用哪个函式来取得装置内容。要继续讨论映射方式，我们需要一些术语：映射方式用於定义从「视窗」（逻辑座标）到「视埠」（装置座标）的映射。

「视窗」和「视埠」这两个词用得并不恰当。在其他图形介面语言中，视埠通常包含有剪裁区域的意思，并且，我们已经用视窗来指程式在萤幕上占据的区域。在这里的讨论中，我们必须把关於这些词的先入之见丢到一边。

「视埠」是依据装置座标（图素）的。通常，视埠和显示区域相同，但是，如果您已经用 GetWindowDC 或 CreateDC 取得了一个装置内容，则视埠也可以是指整视窗座标或者萤幕座标。点(0,0)是显示区域（或者整个视窗或萤幕）的左上角，x 的值向右增加，y 的值向下增加。

「视窗」是依据逻辑座标的，逻辑座标可以是图素、毫米、英寸或者您想要的任何其他单位。您在 GDI 绘图函式中指定逻辑视窗座标。

但是在真正的意义上，视埠和视窗仅是数学上的概念。对於所有的映射方

式，Windows 都用下面两个公式来将视窗（逻辑）座标转化为视埠（设备）座标：

$$xViewport = (xWindow - xWinOrg) \times \frac{xViewExt}{xWinExt} + xViewOrg$$

$$yViewport = (yWindow - yWinOrg) \times \frac{yViewExt}{yWinExt} + yViewOrg$$

其中，(xWindow,yWindow)是待转换的逻辑点，(xViewport,yViewport)是转换後的装置座标点，一般情形下差不多就是显示区域座标了。

这两个公式使用了分别指定视窗和视埠「原点」的点：(xWinOrg，yWinOrg)是逻辑座标的视窗原点；(xViewOrg,yViewOrg)是装置座标的视埠原点。在内定的装置内容中，这两个点均被设定为(0,0)，但是它们可以改变。此公式意味著，逻辑点(xWinOrg,yWinOrg)总被映射为装置点(xViewOrg,yViewOrg)。如果视窗和视埠的原点是预设值(0,0)，则公式简化为：

$$xViewport = xWindow \times \frac{xViewExt}{xWinExt}$$

$$yViewport = yWindow \times \frac{yViewExt}{yWinExt}$$

此公式还使用了两点来指定「范围」：(xWinExt,yWinExt)是逻辑座标的视窗范围；(xViewExt,yViewExt)是装置座标的视窗范围。在多数映射方式中，范围是映射方式所隐含的，不能够改变。每个范围自身没有什么意义，但是视埠范围与视窗范围的比例是逻辑单位转换为装置单位的换算因数。

例如，当您设定 MM_LOENGLISH 映射方式时，Windows 将 xViewExt 设定为某个图素数而将 xWinExt 设定为 xViewExt 图素占据的一英寸内有几百图素的长度。比值给出了一英寸内有几百个图素的数值。为了提高转换效能，换算因数表示为整数比而不是浮点数。

范围可以为负，也就是说，逻辑 x 轴上的值不一定非得在向右时增加；逻辑 y 轴上的值不一定非得在向下时增加。

Windows 也能将视埠（设备）座标转换为视窗（逻辑）座标：

$$xWindow = (xViewport - xViewOrg) \times \frac{xWinExt}{xViewExt} + xWinOrg$$

$$yWindow = (yViewport - yViewOrg) \times \frac{yWinExt}{yViewExt} + yWinOrg$$

Windows 提供了两个函式来让您将装置点转换为逻辑点以及将逻辑点转换为装置点。下面的函式将装置点转换为逻辑点：

```
DPtoLP (hdc, pPoints, iNumber) ;
```

其中，pPoints 是一个指向 POINT 结构阵列的指标，而 iNumber 是要转换的点的个数。您会发现这个函式对於将 GetClientRect（它总是使用装置单位）取得的显示区域大小转换为逻辑座标很有用：

```
GetClientRect (hwnd, &rect) ;
DPtoLP (hdc, (PPOINT) &rect, 2) ;
```

下面的函式将逻辑点转换为装置点：

```
LPtoDP (hdc, pPoints, iNumber) ;
```

处理 MM_TEXT

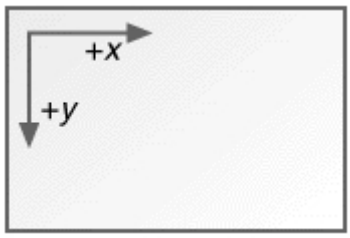
对於 MM_TEXT 映射方式，内定的原点和范围如下所示：

- 视窗原点： (0, 0) 可以改变
- 视埠原点： (0, 0) 可以改变
- 视窗范围： (1, 1) 不可改变
- 视埠范围： (1, 1) 不可改变

视埠范围与视窗范围的比例为 1，所以不用在逻辑座标与装置座标之间进行缩放。上面所给出的公式可以简化为：

$$xViewport = xWindow - xWinOrg + xViewOrg$$
$$yViewport = yWindow - yWinOrg + yViewOrg$$

这种映射方式称为「文字」映射方式，不是因为它对於文字最适合，而是由於轴的方向。我们读文字是从左至右，从上至下的，而 MM_TEXT 以同样的方向定义轴上值的增长方向：



Windows 提供了函式 SetViewportOrgEx 和 SetWindowOrgEx，用来改变视埠和视窗的原点，这些函式都具有改变轴的效果，以致(0,0)不再指左上角。一般来说，您会使用 SetViewportOrgEx 或 SetWindowOrgEx 之一，但不会同时使用二者。

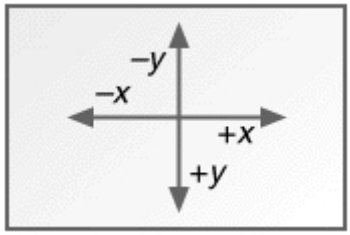
我们来看一看这些函式有何效果：如果将视埠原点改变为

(xViewOrg,yViewOrg)，则逻辑点(0,0)就会映射为装置点(xViewOrg,yViewOrg)。如果将视窗原点改变为(xWinOrg,yWinOrg)，则逻辑点(xWinOrg,yWinOrg)将会映射为装置点(0,0)，即左上角。不管对视窗和视埠原点作什么改变，装置点(0,0)始终是显示区域的左上角。

例如，假设显示区域为 cxClient 个图素宽和 cyClient 个图素高。如果想将逻辑点(0，0)定义为显示区域的中心，可进行如下呼叫：

```
SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
```

SetViewportOrgEx 的参数总是使用装置单位。现在，逻辑点(0,0)将映射为装置点(cxClient/2,cyClient/2)，而显示区域的座标系变成如下形状：



逻辑 x 轴的范围从-cxClient/2 到+cxClient/2，逻辑 y 轴的范围从-cyClient/2 到 +cyClient/2，显示区域的右下角为逻辑点(cxClient/2,cyClient/2)。如果您想从显示区域的左上角开始显示文字。则需要使用负座标：

```
TextOut (hdc, -cxClient / 2, -cyClient / 2, "Hello", 5) ;
```

用下面的 SetWindowOrgEx 叙述可以获得与上面使用 SetViewportOrgEx 同样的效果：

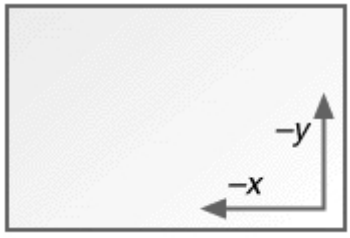
```
SetWindowOrgEx (hdc, -cxClient / 2, -cyClient / 2, NULL) ;
```

SetWindowOrgEx 的参数总是使用逻辑单位。在这个呼叫之後，逻辑点(-cxClient / 2,-cyClient / 2)映射为装置点(0,0)，即显示区域的左上角。

您不会将这两个函式一起用，除非您知道这么做的结果：

```
SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
SetWindowOrgEx (hdc, -cxClient / 2, -cyClient / 2, NULL) ;
```

这意味著逻辑点(-cxClient/2,-cyClient/2)将映射为装置点(cxClient/2,cyClient/2)，结果是如下所示的座标系：



您可以使用下面两个函式取得目前视埠和视窗的原点：

```
GetViewportOrgEx (hdc, &pt) ;
GetWindowOrgEx (hdc, &pt) ;
```

其中 pt 是 POINT 结构。由 GetViewportOrgEx 传回的值是装置座标，而由

GetWindowOrgEx 传回的值是逻辑座标。

您可能想改变视埠或者视窗的原点，以改变视窗显示区域内的显示输出——例如，回应使用者在卷动列内的输入。但是，改变视埠和视窗原点并不能立即改变显示输出，而必须在改变原点之後更新输出。例如，在第四章的 SYSMETS2 程式中，我们使用了 iVscrollPos 值（垂直卷动列的目前位置）来调整显示输出的 y 座标：

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

      for (i = 0 ; i < NUMLINES ; i++)
      {
            y = cyChar * (i - iVscrollPos) ;
            // 显示文字
      }
      EndPaint (hwnd, &ps) ;
      return 0 ;
```

我们可以使用 SetWindowOrgEx 获得同样的效果：

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

      SetWindowOrgEx (hdc, 0, cyChar * iVscrollPos) ;

      for (i = 0 ; i < NUMLINES ; i++)
      {
            y = cyChar * i ;
            // 显示文字
      }
      EndPaint (hwnd, &ps) ;
      return 0 ;
```

现在，TextOut 函式的 y 座标的计算不需要 iVscrollPos 的值。这意味著您可以将文字输出函式放到一个常式中，不用将 iVscrollPos 值传给该常式，因为我们是通过改变视窗原点来调整文字显示的。

如果您有使用直角座标系（即笛卡尔座标系）的经验，那么将逻辑点（0,0）移到显示区域的中央（像我们上面所说的那样）的确值得考虑。但是，对於 MM_TEXT 映射方式来说，还存在著一个小小的问题：笛卡尔座标系中，y 值是随著上移而增加的，而 MM_TEXT 定义为下移时 y 值增加。从这一点来看，MM_TEXT 有点古怪，而下面这五种映射方式都使用通常的增值方法。

「度量」映射方式

Windows 包含五种以实际尺寸来表示逻辑座标的映射方式。由於 x 轴和 y 轴

的逻辑座标映射为相同的实际单位，这些映射方式能使您画出不变形的圆和矩形。

这五种「度量」映射方式在表 5-6 中列出，按照从低精度到高精度的顺序排列。右边的两列分别给出了以英寸和毫米为单位时逻辑单位的大小，以便比较。

表 5-6

映射方式	逻辑单位	英寸	毫米
MM_LOENGLISH	0.01 in.	0.01	0.254
MM_LOMETRIC	0.1 mm.	0.00394	0.1
MM_HIENGLISH	0.001 in.	0.001	0.0254
MM_TWIPS	1/1400 in.	0.000694	0.0176
MM_HIMETRIC	0.01 mm.	0.000394	0.01

内定视窗及视埠的原点和范围如下所示：

- 视窗原点： (0, 0) 可以改变
- 视埠原点： (0, 0) 可以改变
- 视窗范围： (1, 1) 不可改变
- 视埠范围： (1, 1) 不可改变

问号表示视窗和视埠的范围依赖於映射方式和设备的解析度。前面已经提到过，这些范围本身并不重要，但是表示比例时就必须知道。下面是视窗座标到视埠座标的转换公式：

$$xViewport = (xWindow - xWinOrg) \times \frac{xViewExt}{xWinExt} + xViewOrg$$

$$yViewport = (yWindow - yWinOrg) \times \frac{yViewExt}{yWinExt} + yViewOrg$$

例如，对於 MM_LOENGLISH，Windows 计算的范围如下：

$$\frac{xViewExt}{xWinExt} = 0.01\ in\ 中的水平圖素數$$

$$\frac{-yViewExt}{yWinExt} = 0.01\ in\ 中的垂直圖素數$$

Windows 使用这些来自 GetDeviceCaps 的有用资讯设定范围。只是在 Windows 98 和 Windows NT 之间有一点差别。

首先，来看看 Windows 98 是如何做的：假设您使用「控制台」的「显示」

程式选择了 96 dpi 的系统字体。GetDeviceCaps 对於 LOGPIXELSX 和 LOGPIXELSY 索引都将传回值 96。Windows 为视埠范围使用这些值并以表 5-7 的方式设定视埠和视窗的范围。

表 5-7

映射方式	视埠范围 (x,y)	视窗范围 (x,y)
MM_LOMETRIC	(96, 96)	(254, -254)
MM_HIMETRIC	(96, 96)	(2540, -2540)
MM_LOENGLISH	(96, 96)	(100, -100)
MM_HIENGLISH	(96, 96)	(1000, -1000)
MM_TWIPS	(96, 96)	(1440, -1440)

这样，对 MM_LOENGLISH 来说，96 除以 100 的比值是 0.01 英寸中的图素数。对 MM_LOMETRIC 来说，96 除以 254 的比值是 0.1 毫米中的图素数。

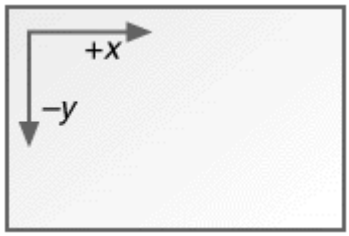
Windows NT 使用不同的方法设定视埠和视窗的范围（与早期 16 位元版本的 Windows 一致的方法）。视埠范围依据萤幕的图素尺寸。可以使用 HORZRES 和 VERTRES 索引从 GetDeviceCaps 取得这种资讯。视窗范围依据假定的显示大小，它是您使用 HORZSIZE 和 VERTSIZE 索引时由 GetDeviceCaps 传回的。我在前面提到过，这些值一般是 320 和 240 毫米。如果您将显示器的图素尺寸设定为 1024 768，则表 5-8 就是 Windows NT 报告的视埠和视窗范围的值。

表 5-8

映射方式	视埠范围 (x,y)	视窗范围 (x,y)
MM_LOMETRIC	(1024, -768)	(3,200, 2,400)
MM_HIMETRIC	(1024, -768)	(32,000, 24,000)
MM_LOENGLISH	(1024, -768)	(1,260, 945)
MM_HIENGLISH	(1024, -768)	(12,598, 9,449)
MM_TWIPS	(1024, -768)	(18,142, 13,606)

这些视窗范围表示包含显示器全部宽度和高度的逻辑单位元数值。320 毫米宽的萤幕也为 1260 MM_LOENGLISH 单位或 12.6 英寸(320 除以 25.4 毫米/英寸)。

范围中，y 前面的负号表示改变了轴的方向。对於这五种映射方式，y 值随上升而增加，然而注意内定的视窗和视埠原点均为(0,0)。这个事实有一个有趣的结果。当一开始改变为五种映射方式之一时，座标系如下：



要想在显示区域显示任何东西，必须使用负的 y 值。例如下面的程式码：

```
SetMapMode (hdc, MM_LOENGLISH) ;
TextOut (hdc, 100, -100, "Hello", 5) ;
```

将把文字显示在距离显示区域左边和上边各一英寸的地方。

为了使自己保持头脑清醒，您可能想避免这样做。一种解决办法是将逻辑的(0,0)点设为显示区域的左下角，您可以通过呼叫 SetViewportOrgEx 来完成（假设 cyClient 是以图素为单位的显示区域的高度）：

```
SetViewportOrgEx (hdc, 0, cyClient, NULL) ;
```

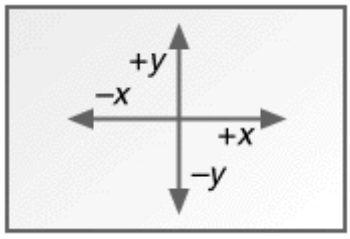
此时的座标系如下：

这是直角座标系的右上象限。

另一种方法是将逻辑(0,0)点设为显示区域的中心：

```
SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
```

此时的座标系如下所示：



现在，我们有了一个真正的 4 象限笛卡尔座标系，在 x 轴和 y 轴上有相等的按英寸、毫米或 twip 计算的逻辑单位。

您还可以使用 SetWindowOrgEx 函式来改变逻辑(0,0)点，但是这稍微困难一些，因为 SetWindowOrgEx 的参数必须使用逻辑单位，先要将(cxClient,cyClient)用 DPtoLP 函式转换为逻辑座标。假设变数 pt 是型态为 POINT 的结构，下面的代码将逻辑(0,0)点改变到显示区域的中央：

```
pt.x = cxClient ;
pt.y = cyClient ;
DptoLP (hdc, &pt, 1) ;
SetWindowOrgEx (hdc, -pt.x / 2, -pt.y / 2, NULL) ;
```

「自行决定」的映射方式

剩下的两种映射方式为 MM_ISOTROPIC 和 MM_ANISOTROPIC。只有这两种映射方式可以让您改变视埠和视窗范围，也就是说可以改变 Windows 用来转换逻辑和装置座标的换算因数。「isotropic」的意思是「同方向性」；「anisotropic」的意思是「异方向性」。与上面所讨论的度量映射方式相似，MM_ISOTROPIC 使用相同的轴，x 轴上的逻辑单位与 y 轴上的逻辑单位的实际尺寸相等。这对您建立纵横比与显示比无关的图像是有帮助的。

MM_ISOTROPIC 与度量映射方式之间的区别是，使用 MM_ISOTROPIC，您可以控制逻辑单位的实际尺寸。如果愿意，您可以根据显示区域的大小来调整逻辑单位的实际尺寸，从而使所画的图像总是包含在显示区域内，并相应地放大或缩小。例如，第八章的两个时钟程式就是方向同性的例子。在您改变视窗大小时，时钟也相应地调整。

Windows 程式完全可以通过调整视窗和视埠范围来处理图像大小的变化。因此，不管视窗尺寸怎样变，程式都可以在绘图函式中使用相同的逻辑单位。

有时候 MM_TEXT 和度量映射方式称为「完全局限性」映射方式，这就是说，您不能改变视窗和视埠的范围以及 Windows 将逻辑座标换算为装置座标的方法。MM_ISOTROPIC 是一种「半局限性」的映射方式，Windows 允许您改变视窗和视埠范围，但只是调整它们，以便 x 和 y 逻辑单位代表同样的实际尺寸。MM_ANISOTROPIC 映射方式是「非局限性」的，您可以改变视窗和视埠范围，但是 Windows 不调整这些值。

MM_ISOTROPIC 映射方式

如果想要在使用任意的轴时都保证两个轴上的逻辑单位相同，则 MM_ISOTROPIC 映射方式就是理想的映射方式。这时，具有相同逻辑宽度和高度的矩形显示为正方形，具有相同逻辑宽度和高度的椭圆显示为圆。

当您刚开始将映射方式设定为 MM_ISOTROPIC 时，Windows 使用与 MM_LOMETRIC 同样的视窗和视埠范围（但是，不要对此有所依赖）。区别在於，您现在可以呼叫 SetWindowExtEx 和 SetViewportExtEx 来根据自己的偏好改变范围了，然後，Windows 将调整范围的值，以便两条轴上的逻辑单位有相同的实际距离。

一般说来，您可以用所期望的逻辑视窗的逻辑尺寸作为 SetWindowExtEx 的参数，用显示区域的实际宽和高作为 SetViewportExtEx 的参数。Windows 在调整这些范围时，必须让逻辑视窗适应实际视窗，这就有可能导致显示区域的一

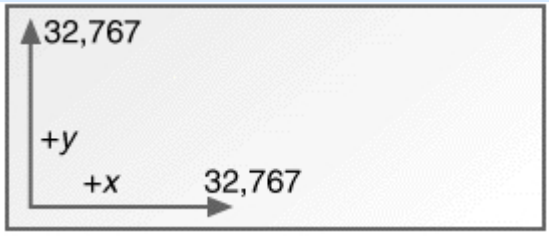
段落到了逻辑视窗的外面。必须在呼叫 SetViewportExtEx 之前呼叫 SetWindowExtEx，以便最有效地使用显示区域中的空间。

例如，假设您想要一个「传统的」单象限虚拟座标系，其中(0,0)在显示区域的左下角，宽度和高度的范围都是从 0 到 32,767，并且希望 x 和 y 轴的单位具有同样的实际尺寸。以下就是所需的程式：

```
SetMapMode (hdc, MM_ISOTROPIC) ;
SetWindowExtEx (hdc, 32767, 32767, NULL) ;
SetViewportExtEx (hdc, cxClient, -cyClient, NULL) ;
SetViewportOrgEx (hdc, 0, cyClient, NULL) ;
```

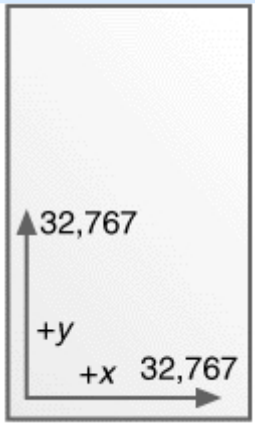
如果其後用 GetWindowExtEx 和 GetViewportExtEx 函式获得了视窗和视埠的范围，可以发现，它们并不是先前指定的值。Windows 将根据显示设备的纵横比来调整范围，以便两条轴上的逻辑单位表示相同的实际尺寸。

如果显示区域的宽度大於高度（以实际尺寸为准），Windows 将调整 x 的范围，以便逻辑视窗比显示区域视埠窄。这样，逻辑视窗将放置在显示区域的左边：



Windows 98 不允许在显示区域的右边超越 x 轴的范围之外显示任何东西，因为这需要一个大於 16 位元所能表示的座标。Windows NT 使用全 32 位元座标，您可以在超出右边显示一些东西。

如果显示区域的高度大於宽度（以实际尺寸为准），那么 Windows 将调整 y 的范围。这样，逻辑视窗将放置在显示区域的下边：



Windows 98 不允许在显示区域的顶部显示任何东西。

如果您希望逻辑视窗总是放在显示区域的左上部，那么将前面给出的程式码改为：

```
SetMapMode (MM_ISOTROPIC) ;
```

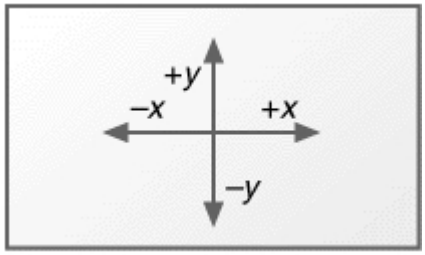
```
SetWindowExtEx (hdc, 32767, 32767, NULL) ;
SetViewportExtEx (hdc, cxClient, -cyClient, NULL) ;
SetWindowOrgEx (hdc, 0, 32767, NULL) ;
```

在呼叫 SetWindowOrgEx 中，我们要求将逻辑点(0, 32767)映射为装置点(0,0)。现在，如果显示区域的高大於宽，则座标系将安排为：

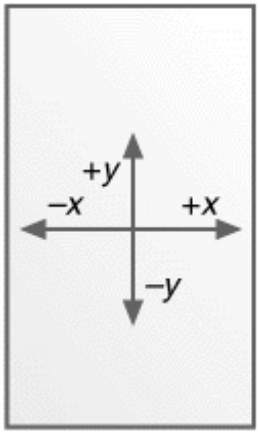
对於时钟程式，您也许想要使用一个四象限的笛卡尔座标系，四个方向的座标尺度可以任意指定，(0,0) 必须居於显示区域的中央。如果您想要每条轴的范围从 0 到 1000，则可以使用以下程式码：

```
SetMapMode (hdc, MM_ISOTROPIC) ;
SetWindowExtEx (hdc, 1000, 1000, NULL) ;
SetViewportExtEx (hdc, cxClient / 2, -cyClient / 2, NULL) ;
SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
```

如果显示区域的宽度大於高度，则逻辑座标系形如：



如果显示区域的高度大於宽度，那么逻辑座标也会居中：



记住，视窗或者视埠范围并不意味著要进行剪裁。在呼叫 GDI 函式时，您仍然对以随便地使用小於-1000 和大於 1000 的 x 和 y 值。根据显示区域的外形，这些点可能看得见，也可能看不见。

在 MM_ISOTROPIC 映射方式下，可以使逻辑单位大於图素。例如，假设您想要一种映射方式，使点(0,0)显示在萤幕的左上角，y 的值向下增长（和 MM_TEXT 相似），但是逻辑座标单位为 1/16 英寸。以下是一种方法：

```
SetMapMode (hdc, MM_ISOTROPIC) ;
SetWindowExtEx (hdc, 16, 16, NULL) ;
SetViewportExtEx (hdc, GetDeviceCaps (hdc, LOGPIXELSX),
                       GetDeviceCaps (hdc, LOGPIXELSY), NULL) ;
```

SetWindowExtEx 函式的参数指出了每一英寸中逻辑单位数。

SetViewportExtEx 函式的参数指出了每一英寸中实际单位数（图素）。

然而，这种方法与 Windows NT 中的度量映射方式不一致。这些映射方式使用显示器的图素大小和公制大小。要与度量映射方式保持一致，可以这样做：

```
SetMapMode (hdc, MM_ISOTROPIC) ;
SetWindowExtEx (hdc, 160 * GetDeviceCaps (hdc, HORZSIZE) / 254,
         160 * GetDeviceCaps (hdc, VERTSIZE) / 254, NULL) ;
SetViewportExtEx (hdc, GetDeviceCaps (hdc, HORZRES),
            GetDeviceCaps (hdc, VERTRES), NULL) ;
```

在这个程式码中，视埠范围设定为按图素计算的整个萤幕的大小，视窗范围则必须设定为以 1/16 英寸为单位的整个萤幕的大小。GetDeviceCaps 以 HORZRES 和 VERTRES 为参数，传回以毫米为单位的装置尺寸。如果我们使用浮点数，将把毫米数除以 25.4，转换为英寸，然後，再乘以 16 以转换为 1/16 英寸。但是，由於我们使用的是整数，所以先乘以 160，再除以 254。

当然，这种座标系会使逻辑单位大於实际单位。在设备上输出的所有东西都将映射为按 1/16 英寸增量的座标值。当然，这样就不能画两条间隔 1/32 英寸的水平直线，因为这样将需要小数逻辑座标。

MM_ANISOTROPIC：根据需要放缩图像

在 MM_ISOTROPIC 映射方式下设定视窗和视埠范围时，Windows 会调整范围，以便两条轴上的逻辑单位具有相同的实际尺度。在 MM_ANISOTROPIC 映射方式下，Windows 不对您所设定的值进行调整，这就是说，MM_ANISOTROPIC 不需要维持正确的纵横比。

使用 MM_ANISOTROPIC 的一种方法是对显示区域使用任意座标，就像我们对 MM_ISOTROPIC 所做的一样。下面的程式码将点(0,0)设定为显示区域的左下角，x 轴和 y 轴都从 0 到 32,767：

```
SetMapMode (hdc, MM_ANISOTROPIC) ;
SetWindowExtEx (hdc, 32767, 32767, NULL) ;
SetViewportExtEx (hdc, cxClient, -cyClient, NULL) ;
SetViewportOrgEx (hdc, 0, cyClient, NULL) ;
```

在 MM_ISOTROPIC 方式下，相似的程式码导致显示区域的一部分在轴的范围之外。但是对於 MM_ANISOTROPIC，不论其尺度多大，显示区域的右上角总是(32767, 32767)。如果显示区域不是正方形的，则逻辑 x 和 y 的单位具有不同的实际尺度。

前一节在 MM_ISOTROPIC 映射方式下，我们讨论了在显示区域中画一个类似时钟的图像，x 和 y 轴的范围都是从-1000 到+1000。对於 MM_ANISOTROPIC，也可以写出类似的程式：

```
SetMapMode (hdc, MM_ANISOTROPIC) ;
```

```
SetWindowExtEx (hdc, 1000, 1000, NULL) ;
SetViewportExtEx (hdc, cxClient / 2, -cyClient / 2, NULL) ;
SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
```

与 MM_ANISOTROPIC 方式不同的是，这个时钟一般是椭圆形的，而不是圆形的。

另一种使用 MM_ANISOTROPIC 的方法是将 x 和 y 轴的单位固定，但其值不相等。例如，如果有一个只显示文字的程式，您可能想根据单个字元的高度和宽度设定一种粗刻度的座标：

```
SetMapMode (hdc, MM_ANISOTROPIC) ;
SetWindowExtEx (hdc, 1, 1, NULL) ;
SetViewportExtEx (hdc, cxChar, cyChar, NULL) ;
```

当然，这里假设 cxChar 和 cyChar 分别是那种字体的字元宽度和高度。现在，您可以按字元行和列指定座标。下面的叙述在距离显示区域左边三个字元，上边二个字元处显示文字：

```
TextOut (hdc, 3, 2, TEXT ("Hello"), 5) ;
```

如果您使用固定大小的字体时会更加方便，就像下面的 WHATSIZE 程式所示的那样。

当您第一次设定 MM_ANISOTROPIC 映射方式时，它总是继承前面所设定的映射方式的范围，这会很方便。可以认为 MM_ANISOTROPIC 不「锁定」范围；也就是说，它允许您任意改变视窗范围。例如，假设您想用 MM_LOENGLISH 映射方式，因为希望逻辑单位为 0.01 英寸，但您不希望 y 轴的值向上增加，喜欢如 MM_TEXT 那样的方向，即 y 轴的值向下增加，可以使用如下的代码：

```
SIZE size ;
```

其他行程式

```
SetMapMode (hdc, MM_LOENGLISH) ;
SetMapMode (hdc, MM_ANISOTROPIC) ;
GetViewportExtEx (hdc, &size) ;
SetViewportExtEx (hdc, size.cx, -size.cy, NULL) ;
```

我们首先将映射方式设定为 MM_LOENGLISH，然後，通过将映射方式设定为 MM_ANISOTROPIC 让范围可以自由改变。GetViewportExtEx 取得视埠范围并放到一个 SIZE 结构中，然後，我们使用范围来呼叫 SetViewportExtEx，只是要将 y 范围取反。

WHATSIZE 程式

Windows 的小历史：第一篇如何写作 Windows 程式的介绍文章出现在《Microsoft Systems Journal》1986 年 12 月号上。在那篇文章中，范例程式叫做 WSZ（「what size：什么尺寸」），它以图素、英寸和毫米为单位显示了

显示区域的大小。那个程式的更简易版本是 WHATSIZE，如程式 5-6 所示。程式显示了以五种度量映射方式显示的视窗显示区域的大小。

程式 5-6 WHATSIZE

```
WHATSIZE.C
/*------------------------------------------------------------
   WHATSIZE.C -- What Size is the Window?
          (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 ("WhatSize") ;
      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 ("What Size is the Window?"),
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL) ;

      ShowWindow (hwnd, iCmdShow) ;
      UpdateWindow (hwnd) ;
      while (GetMessage (&msg, NULL, 0, 0))
      {
            TranslateMessage (&msg) ;
```

```
            DispatchMessage (&msg) ;
      }
      return msg.wParam ;
}
void Show (HWND hwnd, HDC hdc, int xText, int yText, int iMapMode,
            TCHAR * szMapMode)
{
      TCHAR szBuffer [60] ;
      RECT  rect ;

      SaveDC (hdc) ;
      SetMapMode (hdc, iMapMode) ;
      GetClientRect (hwnd, &rect) ;
      DPtoLP (hdc, (PPOINT) &rect, 2) ;

      RestoreDC (hdc, -1) ;
      TextOut (   hdc, xText, yText, szBuffer,
                  wsprintf (szBuffer, TEXT ("%-20s %7d %7d %7d %7d"), szMapMode,
                  rect.left, rect.right, rect.top, rect.bottom)) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
      static TCHAR szHeading [] =
            TEXT ("Mapping Mode    Left  Right Top  Bottom") ;
      static TCHAR szUndLine [] =
            TEXT ("------------   ----   -----  ---  ------") ;
      static int   cxChar, cyChar ;
      HDC         hdc ;
      PAINTSTRUCT ps ;
      TEXTMETRIC  tm ;

      switch (message)
      {
      case  WM_CREATE:
            hdc = GetDC (hwnd) ;
            SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;

            GetTextMetrics (hdc, &tm) ;
            cxChar = tm.tmAveCharWidth ;
            cyChar = tm.tmHeight + tm.tmExternalLeading ;

            ReleaseDC (hwnd, hdc) ;
            return 0 ;

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

```
            SetMapMode (hdc, MM_ANISOTROPIC) ;
            SetWindowExtEx (hdc, 1, 1, NULL) ;
            SetViewportExtEx (hdc, cxChar, cyChar, NULL) ;

            TextOut (hdc, 1, 1, szHeading, lstrlen (szHeading)) ;
            TextOut (hdc, 1, 2, szUndLine, lstrlen (szUndLine)) ;

            Show (hwnd, hdc, 1, 3, MM_TEXT,    TEXT ("TEXT (pixels)")) ;
            Show (hwnd, hdc, 1, 4, MM_LOMETRIC,      TEXT ("LOMETRIC (.1 mm)")) ;
            Show (hwnd, hdc, 1, 5, MM_HIMETRIC,      TEXT    ("HIMETRIC    (.01
mm)")) ;
            Show (hwnd, hdc, 1, 6, MM_LOENGLISH, TEXT ("LOENGLISH (.01 in)")) ;
            Show (hwnd, hdc, 1, 7, MM_HIENGLISH,TEXT ("HIENGLISH (.001 in)")) ;
            Show (hwnd, hdc, 1, 8, MM_TWIPS,  EXT ("TWIPS (1/1440 in)")) ;

            EndPaint (hwnd, &ps) ;
            return 0 ;

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

为了便於用 TextOut 函式显示资讯，WHATSIZE 使用了一种固定间距的字体。下面一条简单的叙述就可以切换为固定间距的字体（在 Windows 3.0 中它是优先使用的）：

```
SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
```

有两个同样的函式用於选取画笔和画刷。像前面提到的，WHATSIZE 也使用 MM_ANISTROPIC 映射方式将逻辑单位设定为字元大小。

当 WHATSIZE 需要取得六种映射方式之一的显示区域的大小时，它保存目前的装置内容，设定一种新的映射方式，取得显示区域座标，将它们转换为逻辑座标，然後在显示资讯之前，恢复原映射方式。底下这些程式码在 WHATSIZE 的 Show 函式里：

```
SaveDC (hdc) ;
SetMapMode (hdc, iMapMode) ;
GetClientRect (hwnd, &rect) ;
DptoLP (hdc, (PPOINT) &rect, 2) ;
RestoreDC (hdc, -1) ;
```

图 5-19 显示了 WHATSIZE 的典型输出。

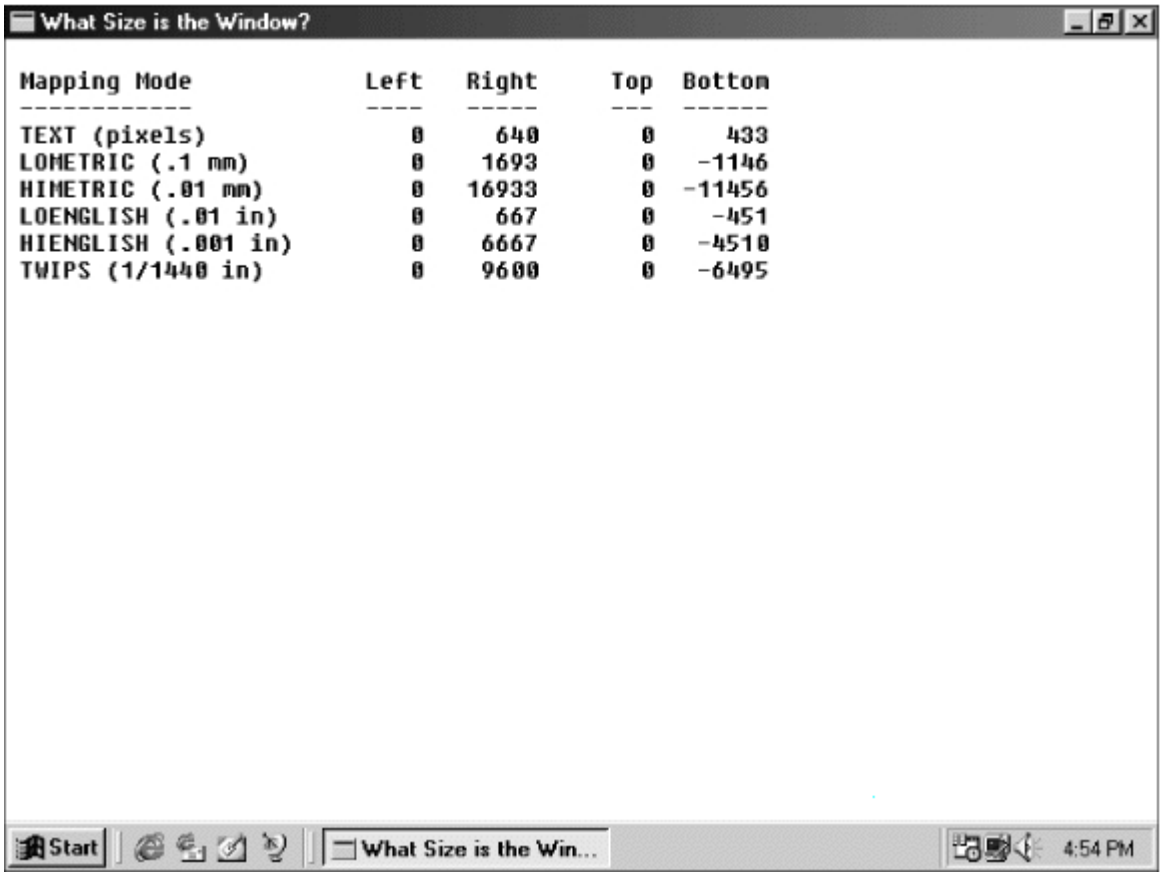


图 5-19 典型的 WHATSIZE 显示

矩形、区域和剪裁

Windows 包含了几种使用 RECT（矩形）结构和「区域」的绘图函式。区域就是萤幕上的一块地方，它是矩形、多边形和椭圆的组合。

矩形函式

下面三个绘图函式需要一个指向矩形结构的指标：

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

在这些函式中，rect 参数是一个 RECT 型态的结构，它包含有 4 个栏位：left、top、right 和 bottom。这个结构中的座标被当作逻辑座标。

FillRect 用指定画刷来填入矩形（直到但不包含 right 和 bottom 座标），该函式不需要先将画刷选进装置内容。

FrameRect 使用画刷画矩形框，但是不填入矩形。使用画刷画矩形看起来有点奇怪，因为对於我们所介绍过的函式（如 Rectangle），其边线都是用目前画笔绘制的。FrameRect 允许使用者画一个不一定为纯色的矩形框。该边界框为一个逻辑单位元宽。如果逻辑单位大於装置单位，则边界框将会为 2 个图素宽或

者更宽。

InvertRect 将矩形中所有图素翻转，1 转换成 0，0 转换为 1，该函式将白色区域转变成黑色，黑色区域转变为白色，绿色区域转变成洋红色。

Windows 还提供了 9 个函式，使您可以更容易、更清楚地操作 RECT 结构。例如，要将 RECT 结构的四个栏位设定为特定值，通常使用如下的程式段：

```
rect.left         = xLeft ;
rect.top          = xTop ;
rect.right        = xRight ;
rect.bottom       = xBottom ;
```

但是，通过呼叫 SetRect 函式，只需要一道叙述就可以得到同样的结果：

```
SetRect (&rect, xLeft, yTop, xRight, yBottom) ;
```

在您想要做以下事情之一时，可以很方便地选用其他 8 个函式：

将矩形沿 x 轴和 y 轴移动几个单元	OffsetRect (&rect, x, y) ;
增减矩形的尺寸	InflateRect (&rect, x, y) ;
矩形各栏位设定为 0	SetRectEmpty (&rect) ;
将矩形复制给另一个矩形	CopyRect (&DestRect, &SrcRect) ;
取得两个矩形的交集	IntersectRect (&DestRect, &SrcRect1, &SrcRect2) ;
取得两个矩形的联集	UnionRect (&DestRect, &SrcRect1, &SrcRect2) ;
确定矩形是否为空	bEmpty = IsRectEmpty (&rect) ;
确定点是否在矩形内	bInRect = PtInRect (&rect, point) ;

大多数情况下，与这些函式相同作用的程式码很简单。例如，您可以用下列叙述来替代 CopyRect 函式呼叫：

```
DestRect = SrcRect ;
```

随机矩形

在图形系统中，有这么一个「永远」有人执行的有趣程式，它简单地使用随机的大小和色彩绘制一系列矩形。您可以在 Windows 中建立一个这样的程式，但是它并不像乍看起来那样容易编写。我希望您能认识到，您不能简单地在 WM_PAINT 讯息中使用一个 while(TRUE)回圈。当然，它能够执行，但是程式将停止对其他讯息的处理，同时，这个程式不能中止或者最小化。

一种可以接受的方法是设定一个 Windows 计时器，给视窗程序发送 WM_TIMER 讯息（我将在第八章中讨论计时器）。对於每条 WM_TIMER 讯息，您使用 GetDC 取得一个装置内容，画一个随机的矩形，然後用 ReleaseDC 释放装置内容。但是这样又降低了程式的趣味性，因为程式不能尽可能快地画随机矩形，它必须等待 WM_TIMER 讯息，而这又依赖於系统时钟的解析度。

在 Windows 中一定有很多「闲置时间」，在这个时间内，所有讯息伫列为

空，Windows 只停在一个小回圈中等待键盘或者滑鼠输入。我们能否在闲置时间内获得控制，绘制矩形，并且只在有讯息加入程式的讯息伫列之後才释放控制呢？这就是 PeekMessage 函式的目的之一。下面是 PeekMessage 呼叫的一个例子：

```
PeekMessage (&msg, NULL, 0, 0, PM_REMOVE) ;
```

前面的四个参数（一个指向 MSG 结构的指标、一个视窗代号、两个值指示讯息范围）与 GetMessage 的参数相同。将第二、三、四个参数设定为 NULL 或 0 时，表明我们想让 PeekMessage 传回程式中所有视窗的所有讯息。如果要将讯息从讯息伫列中删除，则将 PeekMessage 的最後一个参数设定为 PM_REMOVE。如果您不希望删除讯息，那么您可以将这个参数设定为 PM_NOREMOVE。这就是为什么 Peek_Message 是「偷看」而不是「取得」的原因，它使得程式可以检查程式的伫列中的下一个讯息，而不实际删除它。

GetMessage 不将控制传回给程式，直到从程式的讯息伫列中取得讯息，但是 PeekMessage 总是立刻传回，而不论一个讯息是否出现。当讯息伫列中有一个讯息时，PeekMessage 的传回值为 TRUE（非 0），并且将按通常方式处理讯息。当伫列中没有讯息时，PeekMessage 传回 FALSE（0）。

这使得我们可以改写普通的讯息回圈。我们可以将如下所示的回圈：

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

替换为下面的回圈：

```
while (TRUE)
{
      if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
      {
            if (msg.message == WM_QUIT)
                  break ;
            TranslateMessage (&msg) ;
            DispatchMessage (&msg) ;
      }
      else
      {
            // 完成某些工作的其他行程式
      }
}
return msg.wParam ;
```

注意，WM_QUIT 讯息被另外挑出来检查。在普通的讯息回圈中您不必这么作，

因为如果 GetMessage 接收到一个 WM_QUIT 讯息，它将传回 0，但是 PeekMessage 用它的传回值来指示是否得到一个讯息，所以需要对 WM_QUIT 进行检查。

如果 PeekMessage 的传回值为 TRUE，则讯息按通常方式进行处理。如果传回值为 FALSE，则在将控制传回给 Windows 之前，还可以作一点工作（如显示另一个随机矩形）。

（尽管 Windows 文件上说，您不能用 PeekMessage 从讯息伫列中删除 WM_PAINT 讯息，但是这并不是什么大不了的问题。毕竟，GetMessage 并不从讯息伫列中删除 WM_PAINT 讯息。从伫列中删除 WM_PAINT 讯息的唯一方法是令视窗显示区域的失效区域变得有效，这可以用 ValidateRect 和 ValidateRgn 或者 BeginPaint 和 EndPaint 对来完成。如果您在使用 PeekMessage 从伫列中取出 WM_PAINT 讯息後，同平常一样处理它，那么就不会有问题了。所不能作的是使用如下所示的程式码来清除讯息伫列中的所有讯息：

```
while (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE)) ;
```

这行叙述从讯息伫列中删除 WM_PAINT 之外的所有讯息。如果伫列中有一个 WM_PAINT 讯息，程式就会永远地陷在 while 回圈中。）

PeekMessage 在 Windows 的早期版本中比在 Windows 98 中要重要得多。这是因为 Windows 的 16 位元版本使用的是非优先权式的多工（我将在第二十章中讨论这一点）。Windows 的 Terminal 程式在从通讯埠接收输入後，使用一个 PeekMessage 回圈。列印管理器程式使用这个技术来进行列印，其他的 Windows 列印应用程式通常都会使用一个 PeekMessage 回圈。在 Windows 98 优先权式的多工环境下，程式可以建立多个执行绪，我们将第二十章看到这一点。

不管怎样，有了 PeekMessage 函式，我们就可以编写一个不停地显示随机矩形的程式。这个 RANDRECT 如程式 5-7 中所示。

程式 5-7 RANDRECT

```
RANDRECT.C
/*------------------------------------------------------------------------
   RANDRECT.C --  Displays Random Rectangles
                  (c) Charles Petzold, 1998
------------------------------------------------------------------------*/

#include <windows.h>
#include <stdlib.h>    // for the rand function

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

int cxClient, cyClient ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
```

```
{
     static TCHAR szAppName[] = TEXT ("RandRect") ;
     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 ("Random Rectangles"),
                     WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     NULL, NULL, hInstance, NULL) ;

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

     while (TRUE)
     {
          if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
          {
               if (msg.message == WM_QUIT)
                  break ;
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
        }
          else
               DrawRectangle (hwnd) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
```

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

      case  WM_DESTROY:
            PostQuitMessage (0) ;
            return 0 ;
      }
      return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
void DrawRectangle (HWND hwnd)
{
      HBRUSH      hBrush ;
      HDC         hdc ;
      RECT        rect ;

      if (cxClient == 0 || cyClient == 0)
            return ;
      SetRect (&rect, rand () % cxClient, rand () % cyClient,
                        rand () % cxClient, rand () % cyClient) ;
      hBrush = CreateSolidBrush (
                  RGB (rand () % 256, rand () % 256, rand () % 256)) ;
      hdc = GetDC (hwnd) ;
      FillRect (hdc, &rect, hBrush) ;
      ReleaseDC (hwnd, hdc) ;
      DeleteObject (hBrush) ;
}
```

这个程式在现在的电脑上执行得非常快，看起来都不像是一系列随机矩形了。程式使用我在上面讨论过的 SetRect 和 FillRect 函式，根据由 C 的 rand 函式得到的乱数决定矩形座标和实心画刷的色彩。我将在第二十章中提供这个程式的多执行绪版本。

建立和绘制剪裁区域

剪裁区域是对显示器上一个范围的描述，这个范围是矩形、多边形和椭圆的组合。剪裁区域可以用於绘制和剪裁，通过将剪裁区域选进装置内容，就可以用剪裁区域来进行剪裁（就是说，将可以绘图的范围限制为显示区域的一部分）。与画笔、画刷和点阵图一样，剪裁区域是GDI物件，您应该呼叫DeleteObject来删除您所建立的剪裁区域。

当您建立一个剪裁区域时，Windows 传回一个该剪裁区域的代号，型态为

HRGN。最简单的剪裁区域是矩形，有两种建立矩形的方法：

```
hRgn = CreateRectRgn (xLeft, yTop, xRight, yBottom) ;
```

或者

```
hRgn = CreateRectRgnIndirect (&rect) ;
```

您也可以建立椭圆剪裁区域：

```
hRgn = CreateEllipticRgn (xLeft, yTop, xRight, yBottom) ;
```

或者

```
hRgn = CreateEllipticRgnIndirect (&rect) ;
```

CreateRoundRectRgn 建立圆角的矩形剪裁区域。

建立多边形剪裁区域的函式类似於 Polygon 函式：

```
hRgn = CreatePolygonRgn (&point, iCount, iPolyFillMode) ;
```

point 参数是一个 POINT 型态的结构阵列，iCount 是点的数目，iPolyFillMode 是 ALTERNATE 或者 WINDING。您还可以用 CreatePolyPolygonRgn 来建立多个多边形剪裁区域。

那么，您会问，剪裁区域究竟有什么特别之处？下面这个函式才真正显示出了剪裁区域的作用：

```
iRgnType = CombineRgn (hDestRgn, hSrcRgn1, hSrcRgn2, iCombine) ;
```

这一函式将两个剪裁区域（hSrcRgn1 和 hSrcRgn2）组合起来并用代号 hDestRgn 指向组合成的剪裁区域。这三个剪裁区域代号都必须是有效的，但是 hDestRgn 原来所指向的剪裁区域被破坏掉了（当您使用这个函式时，您可能要让 hDestRgn 在初始时指向一个小的矩形剪裁区域）。

iCombine 参数说明 hSrcRgn1 和 hSrcRgn2 如何组合，见表 5-9。

表 5-9

iCombine 值	新剪裁区域
RGN_AND	两个剪裁区域的公共部分
RGN_OR	两个剪裁区域的全部
RGN_XOR	两个剪裁区域的全部除去公共部分
RGN_DIFF	hSrcRgn1 不在 hSrcRgn2 中的部分
RGN_COPY	hSrcRgn1 的全部（忽略 hSrcRgn2）

从 CombineRgn 传回的 iRgnType 值是下列之一：NULLREGION，表示得到一个空剪裁区域；SIMPLEREGION，表示得到一个简单的矩形、椭圆或者多边形；COMPLEXREGION，表示多个矩形、椭圆或多边形的组合；ERROR，表示出错了。

剪裁区域的代号可以用於四个绘图函式：

```
FillRgn (hdc, hRgn, hBrush) ;
FrameRgn (hdc, hRgn, hBrush, xFrame, yFrame) ;
InvertRgn (hdc, hRgn) ;
PaintRgn (hdc, hRgn) ;
```

FillRgn、FrameRgn 和 InvertRgn 类似於 FillRect、FrameRect 和 InvertRect。FrameRgn 的 xFrame 和 yFrame 参数是画在区域周围的边框的宽度和高度。PaintRgn 函式用装置内容中目前画刷填入所指定的区域。所有这些函式都假定区域是用逻辑座标定义的。

在您用完一个区域後，可以像删除其他 GDI 物件那样删除它：

```
DeleteObject (hRgn) ;
```

矩形与区域的剪裁

区域也在剪裁中扮演了一个角色。InvalidateRect 函式使显示的一个矩形区域失效，并产生一个 WM_PAINT 讯息。例如，您可以使用 InvalidateRect 函式来清除显示区域并产生一个 WM_PAINT 讯息：

```
InvalidateRect (hwnd, NULL, TRUE) ;
```

您可以通过呼叫 GetUpdateRect 来取得失效矩形的座标，并且可以使用 ValidateRect 函式使显示区域的矩形有效。当您接收到一个 WM_PAINT 讯息时，无效矩形的座标可以从 PAINTSTRUCT 结构中得到，该结构是用 BeginPaint 函式填入的。这个无效矩形还定义了一个「剪裁区域」，您不能在剪裁区域外绘图。

Windows 有两个作用於剪裁区域而不是矩形的函式，它们类似於 InvalidateRect 和 ValidateRect：

```
InvalidateRgn (hwnd, hRgn, bErase) ;
```

和

```
ValidateRgn (hwnd, hRgn) ;
```

当您接收到一个由无效区域引起的 WM_PAINT 讯息时，剪裁区域不一定是矩形。

您可以使用以下两个函式之一：

```
SelectObject (hdc, hRgn) ;
```

或

```
SelectClipRgn (hdc, hRgn) ;
```

通过将一个剪裁区域选进装置内容来建立自己的剪裁区域，这个剪裁区域使用装置座标。

GDI 为剪裁区域建立一份副本，所以在将它选进装置内容之後，使用者可以删除它。Windows 还提供了几个对剪裁区域进行操作的函式，如 ExcludeClipRect 用於将一个矩形从剪裁区域里排除掉，IntersectClipRect 用於建立一个新的剪裁区域，它是前一个剪裁区域与一个矩形的交，OffsetClipRgn 用於将剪裁区域移动到显示区域的另一部分。

CLOVER 程式

CLOVER 程式用四个椭圆组成一个剪裁区域，将这个剪裁区域选进装置内容中，然後画出从视窗显示区域的中心出发的一系列直线，这些直线只出现在剪裁区域所限定的范围，结果显示如图 5-20 所示。

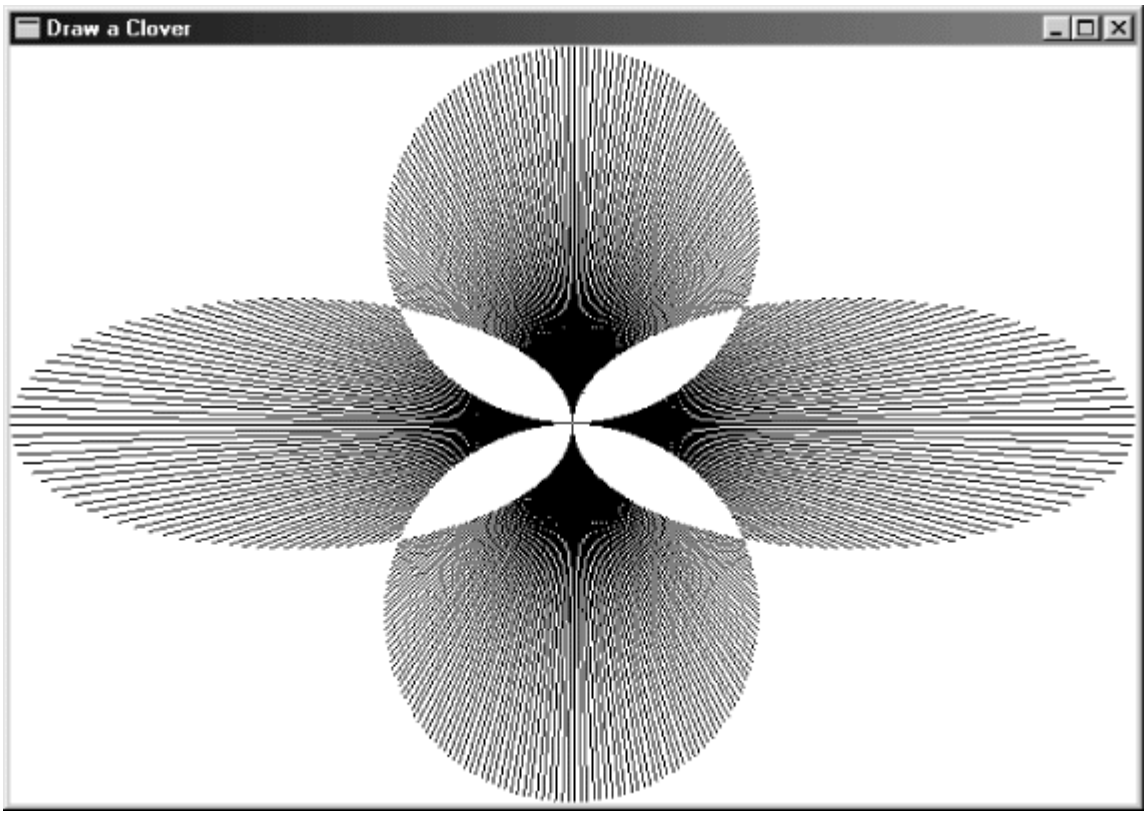


图 5-20 CLOVER 利用复杂的剪裁区域画出的图像

要用常规的方法画出这个图形，就必须根据椭圆的边线公式计算出每条直线的端点。利用复杂的剪裁区域，可以直接画出这些线条，而让 Windows 确定其端点。CLOVER 如程式 5-8 所示。

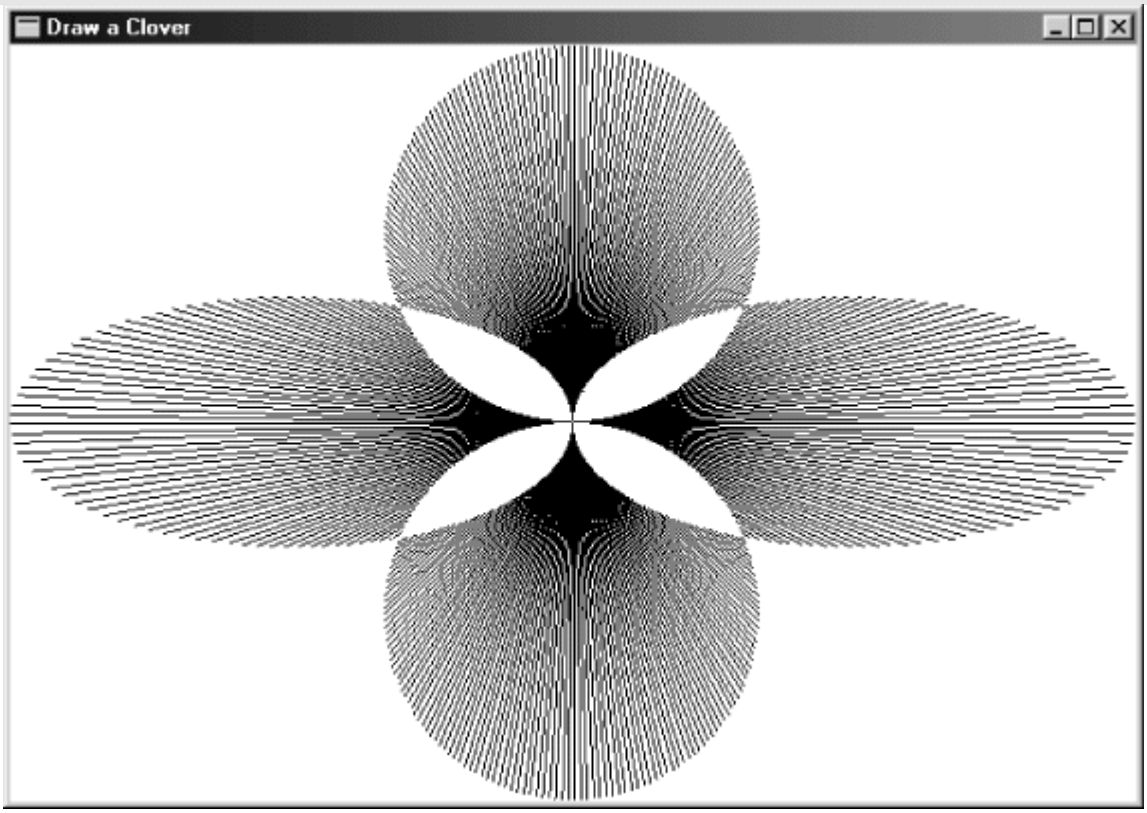
程式 5-8 CLOVER

```
CLOVER.C
/*------------------------------------------------------------------------
      CLOVER.C -- Clover Drawing Program Using Regions
                  (c) Charles Petzold, 1998
------------------------------------------------------------------------*/
#include <windows.h>
#include <math.h>

#define TWO_PI (2.0 * 3.14159)
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
      static TCHAR szAppName[] = TEXT ("Clover") ;
      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 ("Draw a Clover"),
                     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 HRGN hRgnClip ;
     static int  cxClient, cyClient ;
     double      fAngle, fRadius ;
     HCURSOR     hCursor ;
     HDC         hdc ;
     HRGN        hRgnTemp[6] ;
     int         i ;
     PAINTSTRUCT ps ;
     switch (iMsg)
```

```
     {
     case  WM_SIZE:
           cxClient    = LOWORD (lParam) ;
           cyClient    = HIWORD (lParam) ;
           hCursor     = SetCursor (LoadCursor (NULL, IDC_WAIT)) ;
           ShowCursor (TRUE) ;

           if (hRgnClip)
                 DeleteObject (hRgnClip) ;

           hRgnTemp[0] = CreateEllipticRgn (0, cyClient / 3,
                                 cxClient / 2, 2 * cyClient / 3) ;
           hRgnTemp[1] = CreateEllipticRgn (cxClient / 2, cyClient / 3,
                                 cxClient, 2 * cyClient / 3) ;
           hRgnTemp[2] = CreateEllipticRgn (cxClient / 3, 0,
                                 2 * cxClient / 3, cyClient / 2) ;
           hRgnTemp[3] = CreateEllipticRgn (cxClient / 3, cyClient / 2,
2 * cxClient / 3, cyClient) ;
           hRgnTemp[4] = CreateRectRgn (0, 0, 1, 1) ;
           hRgnTemp[5] = CreateRectRgn (0, 0, 1, 1) ;
           hRgnClip    = CreateRectRgn (0, 0, 1, 1) ;

           CombineRgn (hRgnTemp[4], hRgnTemp[0], hRgnTemp[1], RGN_OR) ;
           CombineRgn (hRgnTemp[5], hRgnTemp[2], hRgnTemp[3], RGN_OR) ;
           CombineRgn (hRgnClip,    hRgnTemp[4], hRgnTemp[5], RGN_XOR) ;

           for (i = 0 ; i < 6 ; i++)
                 DeleteObject (hRgnTemp[i]) ;

           SetCursor (hCursor) ;
           ShowCursor (FALSE) ;
           return 0 ;

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

           SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
        SelectClipRgn (hdc, hRgnClip) ;
           fRadius = _hypot (cxClient / 2.0, cyClient / 2.0) ;
           for (fAngle = 0.0 ; fAngle < TWO_PI ; fAngle += TWO_PI / 360)
           {
                 MoveToEx (hdc, 0, 0, NULL) ;
                 LineTo (hdc,     (int) ( fRadius * cos (fAngle) + 0.5),
                       (int) (-fRadius * sin (fAngle) + 0.5)) ;
           }
           EndPaint (hwnd, &ps) ;
           return 0 ;
     case WM_DESTROY:
```

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



由於剪裁区域总是使用装置座标，CLOVER 程式必须在每次接收到 WM_SIZE 讯息时重新建立剪裁区域。几年前，这可能需要几秒钟。现在的快速机器在一瞬间就可以画出来。

CLOVER 从建立四个椭圆剪裁区域开始，这四个椭圆存放在 hRgnTemp 阵列的头四个元素中，然後建立三个「空」剪裁区域：

```
hRgnTemp [4]     = CreateRectRgn (0, 0, 1, 1) ;
hRgnTemp [5]     = CreateRectRgn (0, 0, 1, 1) ;
hRgnClip       = CreateRectRgn (0, 0, 1, 1) ;
```

显示区域左右的两个椭圆区域组合起来：

```
CombineRgn (hRgnTemp [4], hRgnTemp [0], hRgnTemp [1], RGN_OR) ;
```

同样，显示区域上下两个椭圆区域组合起来：

```
CombineRgn (hRgnTemp [5], hRgnTemp [2], hRgnTemp [3], RGN_OR) ;
```

最後，两个组合後的区域再组合到 hRgnClip 中：

```
CombineRgn (hRgnClip, hRgnTemp [4], hRgnTemp [5], RGN_XOR) ;
```

RGN_XOR 识别字用於从结果区域中排除重叠部分。最後，删除 6 个临时区域：

```
for (i = 0 ; i < 6 ; i++)
     DeleteObject (hRgnTemp [i]) ;
```

与画出的图形比起来，WM_PAINT 的处理很简单。视埠原点设定为显示区域

的中心（使画直线更容易一些），在 WM_SIZE 讯息处理期间建立的区域选择为装置内容的剪裁区域：

```
SetViewportOrg (hdc, xClient / 2, yClient / 2) ;
SelectClipRgn (hdc, hRgnClip) ;
```

现在，剩下的就是画直线了，共 360 条，每隔一度画一条。每条线的长度为变数 fRadius，这是从中心到显示区域的角落的距离：

```
fRadius = hypot (xClient / 2.0, yClient / 2.0) ;
for (fAngle = 0.0 ; fAngle < TWO_PI ; fAngle += TWO_PI / 360)
{
      MoveToEx (hdc, 0, 0, NULL) ;
      LineTo (hdc,      (int) ( fRadius * cos (fAngle) + 0.5),
                  (int) (-fRadius * sin (fAngle) + 0.5)) ;
}
```

在处理 WM_DESTROY 讯息时，删除该剪裁区域：

```
DeleteObject (hRgnClip) ;
```

这不是本书关於图形程式设计的最後内容第十三章讨论列印，第十四章和十五章讨论点阵图,第十七章讨论文字和字体，第十八章讨论 metafile。