



基于 Visual C++.NET 的
GDI+开发教程

使用 CIMAGE 类

Visual C++的 CBitmap 类和静态图片控件的功能是比较弱的，它只能显示出在资源中的图标、位图、光标以及图元文件的内容，而不像 VB 中的 Image 控件可以显示出绝大多数的外部图像文件(BMP、GIF、JPEG 等)。因此，想要在对话框或其他窗口中显示外部图像文件则只能借助于第三方提供的控件或代码。现在，MFC 和 ATL 共享的新类 CImage 为图像处理提供了许多相应的方法，这使得 Visual C++在图像方面的缺憾一去不复返了。

[全文阅读](#)

相关专题



Visual C++.NET 编程基础讲座



Visual C++ .Net 编程实战

网友留言

本文章的留言内容:

最新推荐

基于 Visual C++.NET 的 GDI+开发教程

以前开发人员可以通过 GDI(Graphics Device Interface)在 Windows 窗口中绘制图形、文本和图像。但 GDI 的图像处理能力却非常欠缺，位图超过 256 色就会失真或不能显示，因此开发人员在焦急的等待新一代的图形处理工具。



在漫长的等待后，我们终于等来了 GDI+。GDI+ 是 GDI 的新版本，它在 GDI 的基础上添加了许多新特性，为开发人员提供了处理二维矢量图形、文本、图像以及图形数据矩阵的一系列 API 接口。

本文从实际应用出发，着重讨论了 GDI+和 CImage 类的一般使用方法以及在图像处理等方面的使用方法。

GDI+概述

在这一节中首先介绍一下 GDI+的新特性以及其编程方式的改变，然后介绍用 Visual C++.NET 在基于对话框和单文档/多文档等应用程序中使用 GDI+的一般方法。

[全文阅读](#)

GDI+绘图基础

GDI+提供从简单到复杂图形绘制的大量方法，并且我们可以通过对路径和区域的操作构造出更复杂的图形，这在 CAD 等场合极为有用。当然，在绘图之前我们有必要搞清一些基本内容，如坐标空间、画笔和画刷等。

[全文阅读](#)

字体和文本绘制

点击这里留言

字体是文字显示和打印的外观形式，它包括了文字的字样、风格和尺寸等多方面的属性。适当地选用不同的字体，可以大大地丰富文字的外在表现力。例如，把文字中某些重要的字句用较粗的字体显示，能够体现出突出、强调的意图。当然，文本输出时还可使用其格式化属性和显示质量来优化文本显示的效果。

[全文阅读](#)

◎ 图像处理

在以往的图像处理中，常常要根据不同图像文件的格式及其数据存储结构在不同格式中进行转换。某个图像文件的显示也是依靠对文件数据结构的剖析，然后读取相关图像数据而实现的。现在，GDI+提供了 Image 和 Bitmap 类使我们能轻松地处理图像。

[全文阅读](#)

GDI+接口是 Microsoft Whistler 操作系统中的一部分，它是 GDI 的一个新版本，不仅在 GDI 基础上添加许多新特性而且对原有的 GDI 功能进行优化。在为开发人员提供的二维矢量图形、文本、图像处理、区域、路径以及图形数据矩阵等方面构造了一系列相关的类，如 Bitmap(位图类)、Brush(画刷类)、Color(颜色类)、Font(字体类)、Graphics(图形类)、Image(图像类)、Pen(画笔类)和 Region(区域类)等。其中，图形类 Graphics 是 GDI+接口中的一个核心类，许多绘图操作都可用它来完成。

我们首先介绍一下 GDI+的新特性以及其编程方式的改变，然后介绍用 Visual C++.NET 在基于对话框和单文档/多文档等应用程序中使用 GDI+的一般方法。

GDI+新特性

GDI+与 GDI 相比，增加了下列新的特性：

1、渐变画刷

以往 GDI 实现颜色渐变区域的方法是通过使用不同颜色的线条来填充一个裁剪区域而达到的。现在 GDI+拓展了 GDI 功能，提供线型渐变和路径渐变画刷来填充一个图形、路径和区域，甚至也可用来绘制直线、曲线等。这里的路径可以视为由各种绘图函数产生的轨迹。

2、样条曲线

对于曲线而言,最具实际意义的莫过于样条曲线。样条曲线是在生产实践的基础上产生和发展起来的。模线间的设计人员在绘制模线时,先按给定的数据将型值点准确地"点"到图板上。然后,采用一种称为"样条"的工具(一根富有弹性的有机玻璃条或木条),用压铁强迫它通过这些型值点,再适当调整这些压铁,让样条的形态发生变化,直至取得合适的形状,才沿着样条画出所需的曲线。如果我们把样条看成弹性细梁,那么压铁就可看成作用在这梁上的某些点上的集中力。GDI+的 `Graphics:: DrawCurve` 函数中就有一个这样的参数用来调整集中力的大小。除了样条曲线外,GDI+还支持原来 GDI 中的 `Bezier` 曲线。

3、持久的路径对象

我们知道,在 GDI 中,路径是隶属于一个设备环境(上下文),也就是说一旦设备环境指针超过它的有效期,路径也会被删除。而 GDI+是使用 `Graphics` 对象来进行绘图操作,并将路径操作从 `Graphics` 对象分离出来,提供一个 `GraphicsPath` 类供用户使用。这就是说,我们不必担心路径对象会受到 `Graphics` 对象操作的影响,从而可以使用同一个路径对象进行多次的路径绘制操作。

4、矩阵和矩阵变换

在图形处理过程中常需要对其几何信息进行变换以便产生复杂的新图形,矩阵是这种图形几何变换最常用的方法。为了满足人们对图形变换的需求,GDI+提供了功能强大的 `Matrix` 类来实现矩阵的旋转、错切、平移、比例等变换操作,并且 GDI+还支持 `Graphics` 图形和区域(`Region`)的矩阵变换。

5、Alpha 混色

在图像处理中,Alpha 用来衡量一个像素或图像的透明度。在非压缩的 32 位 RGB 图像中,每个像素是由四个部分组成:一个 Alpha 通道和三个颜色分量(R、G 和 B)。当 Alpha 值为 0 时,该像素是完全透明的,而当 Alpha 值为 255 时,则该像素是完全不透明。

Alpha 混色是将源像素和背景像素的颜色进行混合,最终显示的颜色取决于其 RGB 颜色分量和 Alpha 值。它们之间的关系可用下列公式来表示:

显示颜色 = 源像素颜色 X alpha / 255 + 背景颜色 X (255 - alpha) / 255

GDI+的 `Color` 类定义了 `ARGB` 颜色数据类型,从而可以通过调整 Alpha 值来改变线条、图像等与背景色混合后的实际效果。

除了上述新特性外,GDI+还将支持重新着色、色彩修正、消除走样、元数据以及 `Graphics` 容器等特性。

GDI+编程模块的变化

为了简化 GDI+的编程开发过程,Microsoft 对 GDI+的编程模块作了一些调整,这主要体现在以下几个方面:

1、不再使用设备环境或句柄

我们知道，在使用 GDI 绘图时，必须要指定一个设备环境(DC)。MFC 为设备环境提供了许多由基类 CDC 派生的设备环境类，如 CPaintDC、CClientDC 和 CWindowDC 等，用来将某个窗口或设备与设备环境类的句柄指针关联起来，所有的绘图操作都与该句柄有关。而 GDI+不再使用这个设备环境或句柄，取而代之是使用 Graphics 对象。

与设备环境相类似，Graphics 对象也是将屏幕的某一个窗口与之相关联，并包含绘图操作所需要的相关属性。但是，只有这个 Graphics 对象与设备环境句柄还存在着联系，其余的如 Pen、Brush、Image 和 Font 等对象均不再使用设备环境。

2、绘图方式的变化

先来看看同样绘制一条从点(20, 10)到点(200, 100)直线的 GDI 和 GDI+代码，假设这些代码都是添加在 OnDraw 函数中。

GDI 绘制该直线的代码如下：

```
void CMyView::OnDraw(CDC* pDC)
{
    CMyDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CPen newPen( PS_SOLID, 3, RGB(255, 0, 0) );
    CPen* pOldPen = pDC->SelectObject( &newPen );
    pDC->MoveTo( 20, 10 );
    pDC->LineTo( 200, 100);
    pDC->SelectObject( pOldPen );
}
```

GDI+绘制该直线的代码如下：

```
void CMyView::OnDraw(CDC* pDC)
{
    CMyDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    using namespace Gdiplus; // 使用名称空间
    Graphics graphics( pDC->m_hDC );
    Pen newPen( Color( 255, 0, 0 ), 3 );
    graphics.DrawLine(&newPen, 20, 10, 200, 100);
}
```

从上面代码可以看出，GDI 先创建一个 CPen(画笔)对象，然后通过 SelectObject 将该画笔选入到设备环境(pDC)中。接下来调用相应的画线函数，最后恢复设备环境中原来的

GDI 对象。而 GDI+是先使用 Graphics 类创建一个与 pDC 设备环境相关联的 Graphics 对象，然后使用 Pen 类进行画笔的创建，最后调用相应的画线方法。由于 Pen 和设备环境是相互独立的，因而不需要像 GDI 那样恢复设备环境中原来的设置，而且 Pen 和 Graphics 对象的创建不存在先后次序。

2、Graphics 绘图方法直接将 Pen、Brush 等对象作为自己的参数

从上面的代码可以看出，Graphics 绘图方法直接将 Pen 对象作为自己的参数，从而避免了在 GDI 使用 SelectObject 进行繁琐的切换，类似的还有 Brush、Path、Image 和 Font 等。

3、不再使用"当前位置"

我们知道，GDI 绘图操作(如画线)中总存在一个被称为"当前位置"的特殊位置。每次画线都是以此当前位置为起始点，画线操作结束之后，直线的结束点位置又成为了当前位置。设置当前位置的理由是为了提高画线操作的效率，因为在一些场合下，总是一条直线连着另一条直线，首尾相接。有了当前位置的自动更新，就可避免每次画线时都要给出两点的坐标。尽管有其必要性，但是单独绘制一条直线的场合总是比较多的，因此 GDI+取消这个"当前位置"以避免当无法确定"当前位置"时所造成的绘图的差错，取而代之的是直接在 DrawLine 中指定直线起止点的坐标。

4、形状轮廓绘制和填充采用不同的方法

GDI 总是让形状轮廓绘制和填充使用同一个绘图函数，例如 Rectangle。我们知道，轮廓绘制需要一个画笔，而填充一个区域需要一个画刷。也就是说，不管我们是否需要填充所绘制的形状，我们都需要指定一个画刷，否则 GDI 采用默认的画刷进行填充。这种方式确实给我们带来了许多不便，现在 GDI+将形状轮廓绘制和填充操作分开而采用不同的方法，例如 DrawRectangle 和 FillRectangle 分别用来绘制和填充一个矩形。

5、简化区域的创建

我们知道，GDI 提供了许多区域创建函数，如 CreateRectRgn、CreateEllipticRgn、CreateRoundRectRgn、CreatePolygonRgn 和 CreatePolyPolygonRgn 等。诚然，这些函数给我们带来了许多方便。但在 GDI+中，由于为了便于将区域引入矩阵变换操作，GDI+简化一般区域创建的方法，而将更复杂的区域创建交由 Path 接管。由于 Path 对象是与设备环境分离开来的，因而可以直接在 Region 构造函数中加以指定。

6、用 Visual C++.NET 使用 GDI+的一般方法

在 Visual C++.NET 使用 GDI+一般遵循下列步骤：

(1) 在应用程序中添加 GDI+的包含文件 `gdiplus.h` 以及附加的类库 `gdiplus.lib`。通常 `gdiplus.h` 包含文件添加在应用程序的 `stdafx.h` 文件中，而 `gdiplus.lib` 可用两种进行添加：第一种是直接 `stdafx.h` 文件中添加下列语句：

```
#pragma comment( lib, "gdiplus.lib" )
```

另一种方法是：选择"项目"ø"属性"菜单命令，在弹出的对话框中选中左侧的"链接器"ø"输入"选项，在右侧的"附加依赖项"框中键入 `gdiplus.lib`，结果如图 1 所示。



图 7.: 在项目属性中添加 `gdiplus.lib`

图 1

(2) 在应用程序项目的应用类中，添加一个成员变量，如下列代码：

```
ULONG_PTR m_gdiplusToken;
```

其中，`ULONG_PTR` 是一个 `DWORD` 数据类型，该成员变量用来保存 `GDI+` 被初始化后在应用程序中的 `GDI+` 标识，以便能在应用程序退出后，引用该标识来调用 `Gdiplus::GdiplusShutdown` 来关闭 `GDI+`。

(3) 在应用类中添加 `ExitInstance` 的重载，并添加下列代码用来关闭 `GDI+`：

```
int CEx_GDIPlusApp::ExitInstance()  
{  
    Gdiplus::GdiplusShutdown(m_gdiplusToken);  
    return CWinApp::ExitInstance();  
}
```

(4) 在应用类的 `InitInstance` 函数中添加 `GDI+` 的初始化代码：

```
BOOL CEx_GDIPlusApp::InitInstance()  
{  
    CWinApp::InitInstance();  
    Gdiplus::GdiplusStartupInput gdiplusStartupInput;  
    Gdiplus::GdiplusStartup(&m_gdiplusToken, &gdiplusStartupInput, NULL);  
    ...  
}
```

```
}
```

(5) 在需要绘图的窗口或视图类中添加 GDI+的绘制代码。

下面分别就单文档和基于对话框应用程序为例，说明使用 GDI+的一般过程和方法。

1. 在单文档应用程序中使用 GDI+

在上面的过程中，我们就是以单文档应用程序 Ex_GDIPlus 作为示例的。下面列出第 5 步所涉及的代码：

```
void CEx_GDIPlusView::OnDraw(CDC* pDC)
{
    CEx_GDIPlusDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    using namespace Gdiplus;
    Graphics graphics( pDC->m_hDC );
    Pen newPen( Color( 255, 0, 0 ), 3 );
    HatchBrush newBrush( HatchStyleCross,
    Color(255, 0, 255, 0),
    Color(255, 0, 0, 255));
    // 创建一个填充画刷，前景色为绿色，背景色为蓝色

    graphics.DrawRectangle( &newPen, 50, 50, 100, 60);
    // 在(50,50)处绘制一个长为 100，高为 60 的矩形

    graphics.FillRectangle( &newBrush, 50, 50, 100, 60);
    // 在(50,50)处填充一个长为 100，高为 60 的矩形区域
}
```

编译并运行，结果如图 2 所示。

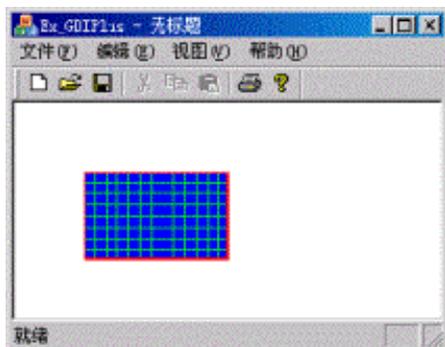


图 2 Ex_GDIPlus 运行结果

图 2

2. 在基于对话框应用程序中使用 GDI+

步骤如下:

(1) 创建一个默认的基于对话框的应用程序 Ex_GDIPlusDlg。

(2) 打开 `stdafx.h` 文件添加下列代码:

```
#include <gdiplus.h>
#pragma comment( lib, "gdiplus.lib" )
```

(3) 打开 `Ex_GDIPlusDlg.h` 文件, 添加下列代码:

```
class CEx_GDIPlusDlgApp : public CWinApp
{
    ...
public:
    virtual BOOL InitInstance();
    ULONG_PTR m_gdiplusToken;
    ...
};
```

(4) 在 `CEx_GDIPlusDlgApp` 类的属性窗口中, 单击"重写"工具按钮, 为该添加 `ExitInstance` 的重载:

```
int CEx_GDIPlusDlgApp::ExitInstance()
{
    Gdiplus::GdiplusShutdown(m_gdiplusToken);
    return CWinApp::ExitInstance();
}
```

(5) 定位到 `CEx_GDIPlusDlgApp::InitInstance` 函数处, 添加下列 GDI+初始化代码:

```
BOOL CEx_GDIPlusDlgApp::InitInstance()
{
    CWinApp::InitInstance();
    Gdiplus::GdiplusStartupInput gdiplusStartupInput;
    Gdiplus::GdiplusStartup(&m_gdiplusToken, &gdiplusStartupInput, NULL);
    ...
}
```

(6) 定位到 `CEx_GDIPlusDlgDlg::OnPaint` 函数处, 添加下列 GDI+代码:

```
void CEx_GDIPlusDlgDlg::OnPaint()
```

```

{
    if (IsIconic())
    {
        ...
    }
    else
    {
        CPaintDC dc(this); // 用于绘制的设备上下文
        using namespace Gdiplus;
        Graphics graphics( dc.m_hDC );
        Pen newPen( Color( 255, 0, 0 ), 3 );
        HatchBrush newBrush( HatchStyleCross,
        Color(255, 0, 255, 0),
        Color(255, 0, 0, 255));
        graphics.DrawRectangle( &newPen, 50, 50, 100, 60);
        graphics.FillRectangle( &newBrush, 50, 50, 100, 60);
        CDialog::OnPaint();
    }
}
}

```

(7) 编译并运行，结果如图 3 所示。

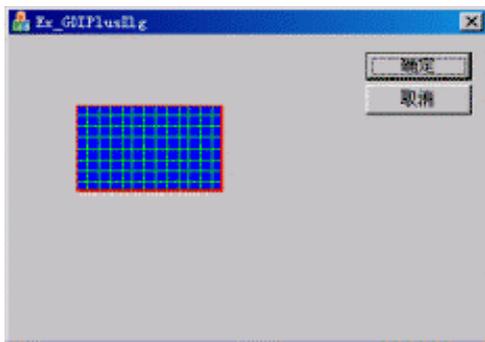


图 7.3 Ex_GDIPlusDlg 运行结果

图 3

从上述例子可以看出，只要能获得一个窗口的设备环境指针，就可构造一个 **Graphics** 对象，从而可以在其窗口中进行绘图，我们不必在像以往那样使用 **Invalidate/UpdateWindow** 来防止 **Windows** 对对话框窗口进行重绘。

GDI+ 提供从简单到复杂图形绘制的大量方法，并且我们可以通过对路径和区域的操作构造出更复杂的图形，这在 **CAD** 等场合极为有用。当然，在绘图之前我们有必要搞清一些基本内容，如坐标空间、画笔和画刷等。

坐标空间及其变换

在视图和窗口中绘图或定位总是在一个二维坐标系进行，依据作用方法的不同，坐标有

多种表示方法，并且各种不同坐标之间可以相互转换。

1. 世界坐标系、设备坐标系和页面坐标系

GDI+为我们提供了三种坐标空间：世界坐标系、页面坐标系和设备坐标系。

"世界坐标系"是应用程序用来进行图形输入输出所使用的一种与设备无关的笛卡尔坐标系。通常，我们可以根据自己的需要和方便定义一个自己的世界坐标系，这个坐标系称为用户坐标系。例如，前面"`DrawLine(&newPen, 20, 10, 200, 100);`"中的坐标都是以这个用户坐标系为基准的，默认时使用像素为单位。

"设备坐标系"是指显示设备或打印设备坐标系下的坐标，它的特点是以设备上的象素点为单位。对于窗口中的视图而言，设备坐标的原点在客户区的左上角，*x* 坐标从左向右递增，*y* 坐标自上而下递增。由于设备的分辨率不同，相同坐标值的物理位置可能不同。如对于边长为 100 的正方形，当显示器为 640 x 480 和 800 x 600 时的大小是不一样的。

"页面坐标系"是指某种映射模式下的一种坐标系。所谓映射是指将世界坐标系通过某种方式进行的变换。默认时，设备坐标和页面坐标是一致的。

2. 坐标映射和坐标原点的设置

为了能保证打印或显示的结果不受设备的影响，GDI+定义了一些映射方法和属性来决定设备坐标和页面坐标之间的关系。这些映射方法和属性有：

SetPageUnit 和 GetPageUnit

这两个属性函数是用来设置和获取每个单位所对应的实际度量单位。它通常可以有以下的值：

UnitDisplay -- 每个页面单位为 1/75 英寸；

UnitPixel -- 每个页面单位为 1 个像素，此时页面坐标与设备坐标相同；

UnitPoint -- 每个页面单位为 1/72 英寸；

UnitInch -- 每个页面单位为 1 英寸；

UnitDocument -- 每个页面单位为 1/300 英寸；

UnitMillimeter-- 每个页面单位为 1 毫米。

例如，或将 `Ex_GDIPlusDlg` 示例中的绘图代码修改成：

```
CPaintDC dc(this);
using namespace Gdiplus;
Graphics graphics( dc.m_hDC );
graphics.SetPageUnit(UnitMillimeter);
Pen newPen( Color( 255, 0, 0 ), 3 );
HatchBrush newBrush( HatchStyleCross,
```

```

Color(255, 0, 255, 0),
Color(255, 0, 0, 255));
graphics.DrawRectangle( &newPen, 50, 50, 100, 60);
graphics.FillRectangle( &newBrush, 50, 50, 100, 60);

```

则笔画宽度为 3，以及矩形的左上角顶点坐标和大小单位都为毫米，其结果如图所示。

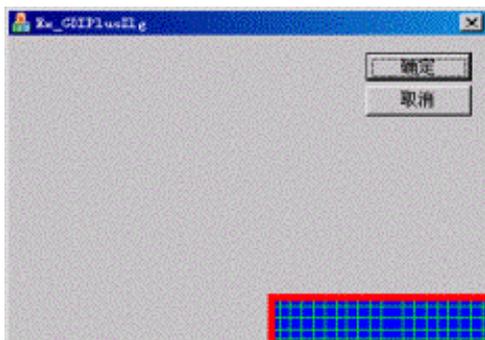


图 1.4 将页面单位设置为毫米后的结果

SetPageScale 和 GetPageScale

GDI+的这两个属性函数分别用来设置和获取页面的缩放比例。例如，当上面的绘图代码变成：

```

...
graphics.SetPageUnit(UnitMillimeter);
graphics.SetPageScale( (REAL)0.1);
Pen newPen( Color( 255, 0, 0 ), 3);
...

```

代码中，REAL 是一个浮点类型的定义。上述代码的结果如图 2 所示。

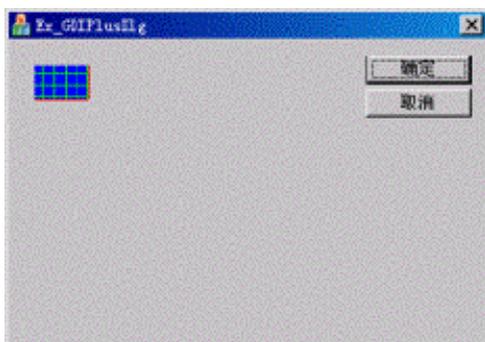


图 1.5 页面缩小的结果

图 2

TranslateTransform

GDI+ 的 `TranslateTransform` 方法用来改变坐标的原点位置，例如 `TranslateTransform(100, 50)` 是将坐标原点移到点(100,50)。

画笔

画笔是用来绘制各种直线和曲线的一种图形工具，GDI+的 `Pen` 类为画笔提供了丰富的方法。一般来说，我们可以通过其构造函数来指定画笔的颜色和宽度，其定义如下：

```
Pen( const Color& color, REAL width );
```

其中，`color` 是用来指定画笔颜色，`width` 用来指定画笔宽度。`REAL` 是一个 `float` 类型定义，而 `Color` 是 GDI+ 的一个颜色类，它既可以指定一个 `ARGB` 颜色类型，也可以使用 GDI+ 预定义的颜色值，甚至可以将 `COLORREF` 转换成 `Color` 类型的颜色。例如，下面的代码都是创建一个宽度为 3，颜色为蓝色的画笔：

```
Pen newPen( Color( 255, 0, 0, 255 ), 3 );
```

```
Pen newPen(Color( 0, 0, 255), 3);  
// 当 Color 只有三个实参时，颜色 Alpha 分量值为 255。
```

```
Pen newPen(Color::Blue, 3);
```

```
COLORREF crRef = RGB( 0, 0, 255);  
Color color;  
color.SetFromCOLORREF(crRef);  
Pen newPen(color, 3);
```

除了颜色外，GDI+ 的 `Pen` 类还提供 `SetDashStyle` 和 `SetDashPattern` 方法来设置画笔的预定义风格和自定义类型。其中，预定义风格可以有：`DashStyleSolid`(实线)、`DashStyleDash`(虚线)、`DashStyleDot`(点线)、`DashStyleDashDot`(点划线)、`DashStyleDashDotDot`(双点划线)和 `DashStyleCustom`(自定义类型)。例如下列代码，其结果如图 7.6 所示：

```
using namespace Gdiplus;  
Graphics graphics( pDC->m_hDC );  
Pen pen(Color(255, 0, 0, 255), 15);  
  
pen.SetDashStyle(DashStyleDash);  
graphics.DrawLine(&pen, 0, 50, 400, 150);  
  
pen.SetDashStyle(DashStyleDot);  
graphics.DrawLine(&pen, 0, 80, 400, 180);  
  
pen.SetDashStyle(DashStyleDashDot);
```

```
graphics.DrawLine(&pen, 0, 110, 400, 210);
```

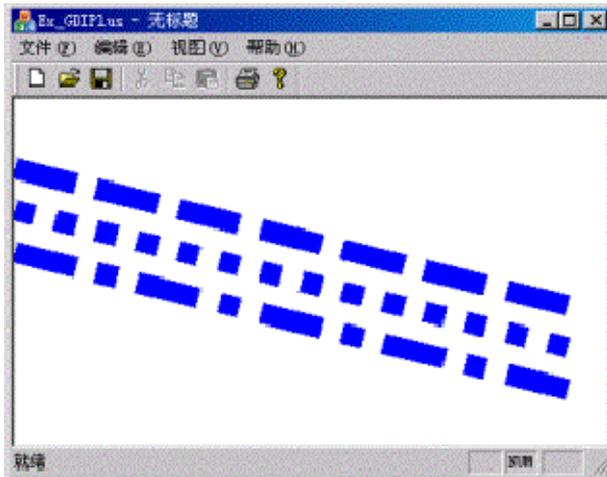


图 7.6 画出的网格

但是，在工程应用中，预定义风格的画笔有时不能满足需求。这时，可以使用 `SetDashPattern` 函数来实现。`SetDashPattern` 的原型如下：

```
Status SetDashPattern( const REAL* dashArray, INT count);
```

其中，`dashArray` 是一个包含短划和间隔长度的数组，`count` 表示数组的大小。注意，`dashArray` 中的短划长度和间隔长度是成对出现的，例如下列代码是使用自定义类型的画笔，其结果如图 7.7 所示。

```
REAL dashVals[4] = {  
2, // 短划长为 2  
2, // 间隔为 2  
15, // 短划长为 15  
2}; // 间隔为 2  
Pen pen(Color(255, 0, 0, 0), 5);  
pen.SetDashPattern(dashVals, 4);  
graphics.DrawLine(&pen, 5, 20, 405, 200);
```

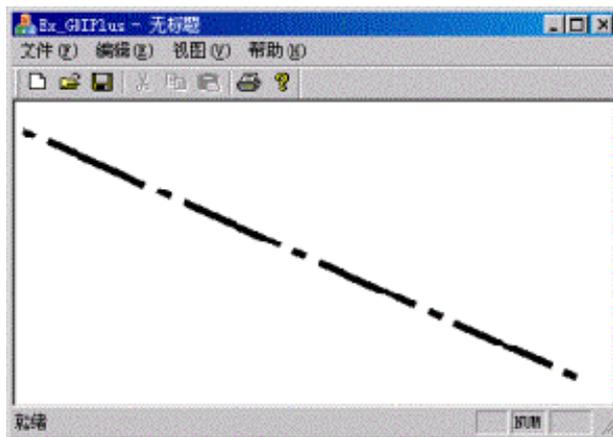


图 7.7 自定义画笔

需要说明的是，GDI+的 Pen 类还提供 `SetStartCap` 和 `SetEndCap` 方法来设置一条直线的起始端和终止端的样式。例如下面的代码，其结果如图 7.8 所示。

```
using namespace Gdiplus;
Graphics graphics( pDC->m_hDC );

Pen pen( Color( 255, 0, 0, 255 ), 15);

pen.SetStartCap(LineCapFlat);
pen.SetEndCap(LineCapSquare);
graphics.DrawLine(&pen, 50, 50, 250, 50);

pen.SetStartCap(LineCapRound );
pen.SetEndCap(LineCapRoundAnchor);
graphics.DrawLine(&pen, 50, 100, 250, 100);

pen.SetStartCap(LineCapDiamondAnchor);
pen.SetEndCap(LineCapArrowAnchor);
graphics.DrawLine(&pen, 50, 150, 250, 150);
```

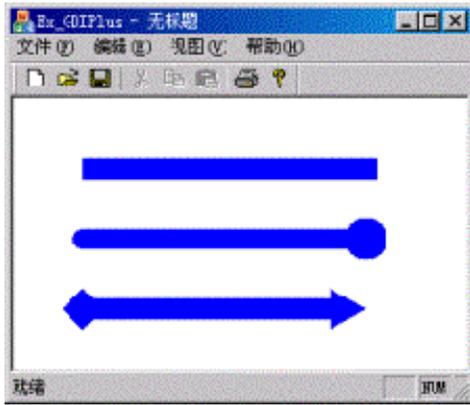


图 7.8 填充样式

画刷和渐变

画刷用于指定填充的特性，GDI+为填充色和阴影线画刷提供了 `SolidBrush` 和 `HatchBrush` 类。通过它们的构造函数直接可以创建一个画刷，其构造函数的原型如下：

```
SolidBrush( const Color& color);
```

```
HatchBrush( HatchStyle hatchStyle, const Color& foreColor,
const Color& backColor);
```

其中，`foreColor` 和 `backColor` 用来指定阴影线颜色和填充的背景颜色，背景色可以不指定。`hatchStyle` 用来指定阴影线的样式，它可以是这样的一些预定义样式：`HatchStyleHorizontal` (水平线)、`HatchStyleVertical`(垂直线)、`HatchStyleForwardDiagonal`(上斜线)、`HatchStyleBackwardDiagonal`(下斜线)、`HatchStyleCross`(十字线)以及 `HatchStyleDiagonalCross` (交叉线)等。当然，还有许多样式如 `HatchStyle30Percent`(30%填充)、`HatchStyleSolidDiamond` (实心菱形)等，这里不一一列举。

由于在前面的示例中，对这种简单的画刷的使用已介绍过，因而这里着重讨论渐变画刷的创建和使用。

GDI+提供了 `LinearGradientBrush` 和 `PathGradientBrush` 类分别用来创建一个直线渐变和路径渐变画刷。

直线渐变是指在一个矩形区域使用两种颜色进行过渡(渐变)，过渡方向可以是水平、垂直以及对角线方向。`LinearGradientBrush` 构造函数的原型如下：

```
LinearGradientBrush(Point & point1, Point & point2,
Color & color1, Color & color2);
```

```
LinearGradientBrush(Rect & rect, Color & color1, Color & color2,
REAL angle, BOOL isAngleScalable);
```

```
LinearGradientBrush(Rect & rect, Color & color1, Color & color2,
LinearGradientMode mode);
```

其中, **point1** 和 **point2** 分别用来指定矩形区域的左上角和右下角点坐标, **color1** 和 **color2** 分别用来指定渐变起始和终止的颜色。 **rect** 用来指定一个矩形区域的大小和位置, **angle** 用来指定渐变的方向角度, 正值为顺时针。 **isAngleScalable** 是一个即将废除的参数。 **mode** 用来指定渐变的方法, 它可以是 **LinearGradientModeHorizontal**(水平方向)、**LinearGradientModeVertical** (垂直方向)、**LinearGradientModeForwardDiagonal**(从左下到右上的对角线方向)和 **LinearGradientModeBackwardDiagonal**(从左上到右下的对角线方向)。

需要说明的是, **Point** 和 **Rect** 是 GDI+新的数据类型, 它们和 MFC 的 **CPoint** 和 **CRect** 类的功能基本一样, 但它们相互之间不能混用。

路径渐变画刷是用渐变颜色来填充一个封闭的路径。一个路径既可以由一系列的直线和曲线构成, 也可以由其它对象来构造。路径渐变是一种中心颜色渐变模式, 它从路径的中心点向四周进行颜色渐变。 **PathGradientBrush** 构造函数的原型如下:

```
PathGradientBrush(const GraphicsPath* path);  
PathGradientBrush(const Point * points, INT count, WrapMode wrapMode);
```

其中, **path** 用来指定一个路径指针, **points** 和 **count** 分别用来指定组成路径的一系列直线端点的数组及其大小, **wrapMode** 是一个可选项, 用来指定填充的包围模式。一个包围模式用来决定是否在区域内部、在区域外部以及所有区域都填充。默认时, 其值为 **WrapModeClamp**, 即在区域内部填充。

下面的代码说明了上述两种渐变画刷的使用方法:

```
Graphics graphics( pDC->m_hDC );  
  
GraphicsPath path; // 构造一个路径  
path.AddEllipse(50, 50, 200, 100);  
  
// 使用路径构造一个画刷  
PathGradientBrush pthGrBrush(&path);  
  
// 将路径中心颜色设为蓝色  
pthGrBrush.SetCenterColor(Color(255, 0, 0, 255));  
  
// 设置路径周围的颜色为蓝芭, 但 alpha 值为 0  
Color colors[] = {Color(0, 0, 0, 255)};  
INT count = 1;  
pthGrBrush.SetSurroundColors(colors, &count);  
  
graphics.FillRectangle(&pthGrBrush, 50, 50, 200, 100);
```

```
LinearGradientBrush linGrBrush(  
Point(300, 50),  
Point(500, 150),  
Color(255, 255, 0, 0), // 红色  
Color(255, 0, 0, 255)); // 蓝色  
  
graphics.FillRectangle(&linGrBrush, 300, 50, 200, 100);
```

结果如图 7.9 所示。

需要说明的是，画笔和画刷还可使用一个图片来创建。例如下列代码，其结果如图 7.10 所示。

```
Graphics graphics( pDC->m_hDC );  
  
Image image(L"image.jpg");  
TextureBrush tBrush(&image);  
Pen texturedPen(&tBrush, 10);  
  
graphics.DrawLine(&texturedPen, 25, 25, 325, 25);  
tBrush.SetWrapMode(WrapModeTileFlipXY);  
graphics.FillRectangle(&tBrush, 25, 100, 300, 200);
```

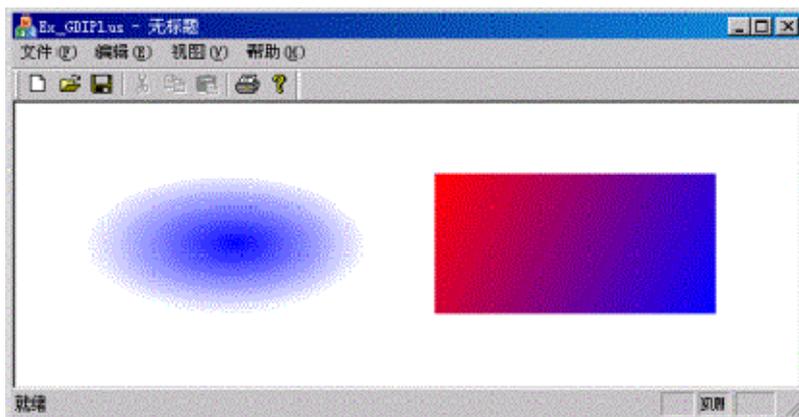


图 7.9 渐变图形

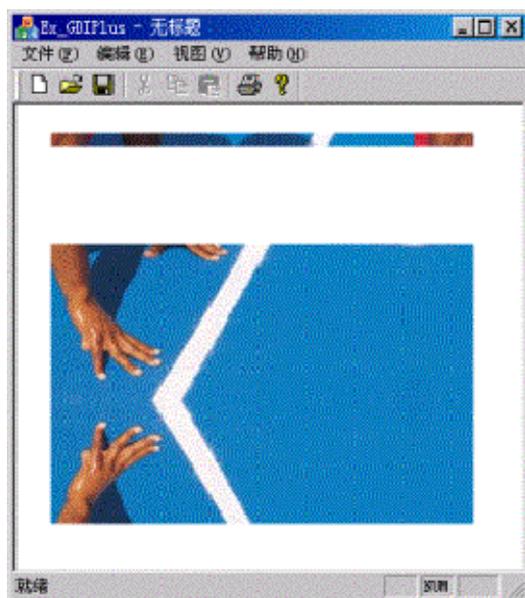


图 7.10 使用图片创建复杂图形

图形几何变换

图形变换一般是对图形的几何信息经过几何变换后产生新的图形。常见二维图形的变换有平移、比例、对称、旋转、错切等。图形几何变换最有效的手段是采用矩阵变换，GDI+就有这样的矩阵类 `Matrix`，它为我们提供了许多变换的方法，如 `Invert`(转置)、`Multiply`(矩阵相乘)、`Rotate`(旋转)等。例如下面的代码就是 `Matrix::Rotate` 一个例子，其结果如图 7.11 所示。

```
Graphics graphics( pDC->m_hDC );
Pen pen(Color(255, 0, 0, 255));

Matrix matrix;
matrix.Translate(40, 0); // 先平移
matrix.Rotate(30, MatrixOrderAppend); // 后旋转

graphics.SetTransform(&matrix);
graphics.DrawEllipse(&pen, 0, 0, 100, 50);
```

需要说明的是，代码中的 `MatrixOrderAppend` 用来指明第二个矩阵(若有)的操作次序是后置的，即 `matrix1 OP matrix2`，`OP` 表示某种操作；若为 `MatrixOrderPrepend` 则表示 `matrix2 OP matrix1`。而 `SetTransform` 则指定一个矩阵对点坐标进行变换，新的坐标点(x^*, y^*)结果可用下列公式来表示：

$$[x^* \ y^* \ 1] = [x \ y \ 1] \begin{pmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ dx & dy & 1 \end{pmatrix} = [m_{11}x+m_{21}y+dx \ m_{12}x+m_{22}y+dy \ 1]$$

式中，`dx` 和 `dy` 用来指定 `x` 和 `y` 方向的平移量，若 `dx = dy = 0`，则：

(1) 当 `m21 = m12 = 0`，`m11 = -1`，`m22 = 1` 时，有 $x^* = -x$ ， $y^* = y$ ，产生与 `y` 轴对称的反射图形；

(2) 当 `m21 = m12 = 0`，`m11 = 1`，`m22 = -1` 时，有 $x^* = x$ ， $y^* = -y$ ，产生与 `x` 轴对称的反射图形；

(3) 当 `m21 = m12 = 0`，`m11 = m22 = -1` 时，有 $x^* = -x$ ， $y^* = -y$ ，产生与原点对称的反射图形；

(4) 当 `m21 = m12 = 1`，`m11 = m22 = 0` 时，有 $x^* = y$ ， $y^* = x$ ，产生与直线 `y = x` 对称的反射图形；

(5) 当 `m21 = m12 = -1`，`m11 = m22 = 0` 时，有 $x^* = -y$ ， $y^* = -x$ ，产生与直线 `y = -x` 对称的反射图形；

(6) 而当 `m11 = m22 = cosq`，`m21 = -m12 = sinq` 时，则进行旋转变换。

例如下列代码，其结果如图 7.12 所示。

```
Graphics graphics( pDC->m_hDC );
Pen pen(Color::Blue,3);
graphics.DrawLine(&pen, 150,50,200,80);

pen.SetColor(Color::Gray);
Matrix matrix( -1,0,0,1, 150,50); // 使用第一种情况

graphics.SetTransform(&matrix);
graphics.DrawLine(&pen, 0,0,50,30);
```

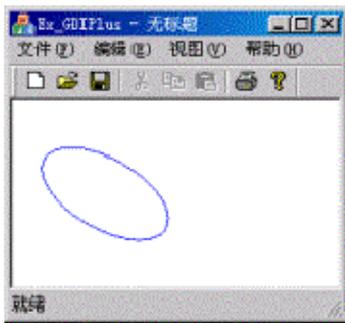


图 7.11 矩阵旋转变换

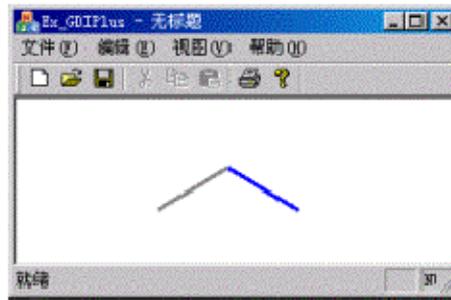


图 7.12 对称变换

其中，Matrix 的构造函数有如下定义：

Matrix(REAL m11, REAL m12, REAL m21, REAL m22, REAL dx, REAL dy);

需要说明的是，除了使用 Matrix 进行图形变换外，Graphics 本身提供相应的变换方法，如 RotateTransform(旋转变换)、ScaleTransform(比例变换)和 TranslateTransform(平移变换)等。

基本绘图函数

在前面许多示例中，我们已经用到如 DrawLine 等基本绘图函数。除此之外，还有许多这样的函数，并且每个绘图函数都有其重载形式，这给我们带来了许多方便。表 7.1 列出这些基本绘图函数。

表 1 GDI+常用基本绘图函数

绘图函数	功能描述
DrawArc	绘制一条圆弧曲线，范围由起止角大小决定，大小由矩形或长宽值指定
DrawBezier	绘制一条由一系列型值顶点决定的三次 Bezier 曲线
DrawBeziers	绘制一系列的三次 Bezier 曲线
DrawClosedCurve	绘制一条封闭的样条曲线
DrawCurve	绘制一条样条曲线
DrawEllipse	绘制一条椭圆轮廓线，大小由矩形或长宽值指定
DrawLine	绘制一条直线
DrawPath	绘制由 GraphicsPath 定义的路径轮廓线
DrawPie	绘制一条扇形(饼形)轮廓线
DrawPolygon	绘制一个多边形的轮廓线
DrawRectangle	绘制一个矩形
FillEllipse	填充一个椭圆区域
FillPath	填充一个由路径指定的区域
FillPie	填充一个扇形(饼形)区域

FillPolygon	填充一个多边形区域
FillRectangle	填充一个矩形区域
FillRectangles	用同一个画刷填充一系列矩形区域
FillRegion	填充一个区域(Region)的内部

下面的代码是通过路径用两条样条曲线构造一个复杂的区域，然后填充它，其结果如图 7.13 所示。

```

Graphics graphics( pDC->m_hDC );

Pen pen(Color::Blue, 3);
Point point1( 50, 200);
Point point2(100, 150);
Point point3(160, 180);
Point point4(200, 200);
Point point5(230, 150);
Point point6(220, 50);
Point point7(190, 70);
Point point8(130, 220);

Point curvePoints[8] = {point1, point2, point3, point4,
point5, point6, point7, point8};
Point* pcurvePoints = curvePoints;

GraphicsPath path;
path.AddClosedCurve(curvePoints, 8, 0.5);

PathGradientBrush pthGrBrush(&path);
pthGrBrush.SetCenterColor(Color(255, 0, 0, 255));
Color colors[] = {Color(0, 0, 0, 255)};
INT count = 1;
pthGrBrush.SetSurroundColors(colors, &count);

graphics.DrawClosedCurve(&pen, curvePoints, 8, 0.5);
graphics.FillPath(&pthGrBrush, &path);

```

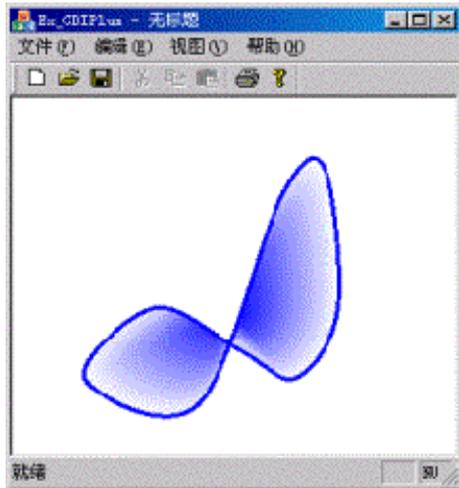


图 7.13 使用样条曲线绘制路径

字体是文字显示和打印的外观形式，它包括了文字的字样、风格和尺寸等多方面的属性。适当地选用不同的字体，可以大大地丰富文字的外在表现力。例如，把文字中某些重要的字句用较粗的字体显示，能够体现出突出、强调的意图。当然，文本输出时还可使用其格式化属性和显示质量来优化文本显示的效果。

字体属性和字体创建

字体的属性有很多，这里主要介绍字样、风格和尺寸三个主要属性。

字样是字符书写和显示时表现出的特定模式，例如，对于汉字，通常有宋体、楷体、仿宋、黑体、隶书以及幼圆等多种字样。GDI+是通过 `FontFamily` 类来定义字样的，例如下面的代码：

```
FontFamily fontFamily(L"幼圆"); // 定义"幼圆"字样
```

字体风格主要表现为字体的粗细和是否倾斜等特点。GDI+为用户提供了一些预定义的字体风格：`FontStyleRegular`(正常)、`FontStyleBold`(加粗)、`FontStyleItalic`(斜体)、`FontStyleBoldItalic`(粗斜体)、`FontStyleUnderline`(下划线)和 `FontStyleStrikeout`(删除线)。

字体尺寸是用来指定字符所占区域的大小，通常用字符高度来描述。字体尺寸可以取毫米或英寸作为单位，但为了直观起见，也常常采用一种称为点的单位，一点约折合为 $1/72$ 英寸。对于汉字，还常用号数来表示字体尺寸，初号字最大，以下依次为小初、一号、小一、二号、小二??，如此类推，字体尺寸越来越小。GDI+为用户提供了 `UnitDisplay`($1/75$ 英寸)、`UnitPixel`(像素)、`UnitPoint`(点)、`UnitInch`(英寸)、`UnitDocument`($1/300$ 英寸)、`UnitMillimeter`(毫米)等字体尺寸单位。

使用 GDI+中的 `Font` 类，可以直接通过构造函数创建一个字体对象，例如下列代码：

```
Font font(&fontFamily, 12, FontStyleRegular, UnitPoint);
```

构造函数的第一个参数是用来指定 **FontFamily** 类对象指针，第二参数是用来指定字体的尺寸，它的实际大小取决于第四个参数所指定的尺寸单位。第三个参数用来指定字体风格。

为了与原来的 **GDI** 字体相兼容，**Font** 的构造函数还有另外一种型式：

```
Font( HDC hdc, const LOGFONTW* logfont)
```

其中，**hdc** 是用来指定一个窗口的设备环境句柄，**logfon** 是指向 **LOGFONT**(逻辑字体) 数据结构的指针。

文本输出

文本的最终输出不仅依赖于文本的字体，而且还跟文本的颜色、对齐方式、字符间隔等有很大关系。**GDI+** 只有一个输出文本的函数 **DrawString**，它的原型如下：

```
DrawString( const WCHAR* string, INT length, const Font* font,  
const RectF& layoutRect, const StringFormat* stringFormat,  
const Brush* brush );
```

```
DrawString( const WCHAR* string, INT length, const Font* font,  
const PointF& origin, const Brush* brush );
```

```
DrawString( const WCHAR* string, INT length, const Font* font,  
const PointF& origin, const StringFormat* stringFormat,  
const Brush* brush);
```

其中，**string** 用来指定要输出的字符串，**length** 表示该字符串的长度，**font** 用来指定字体，**layoutRect** 用来指定一个字符串所输出的矩形区域，**stringFormat** 用来指定文本输出格式化属性，**origin** 用来指定字符串输出的起点。需要注意的是，**PointF** 和 **RectF** 类与 **Point** 和 **Rect** 类基本相同，所不同的是数据类型是浮点而后者是 **INT** 型。**brush** 用来指定一个画刷，这个画刷既可以是 **SolidBrush** 和 **HatchBrush**，也可以是 **TextureBrush**(纹理画刷)，甚至是渐变画刷。例如下面的代码，结果如图 7.14 所示。



图 7.14 使用画刷绘制文本

```
Graphics graphics( pDC->m_hDC );
```

```

FontFamily fontFamily(L"幼圆");
Font font(&fontFamily, 20, FontStyleRegular, UnitPoint);
PointF pointF(30, 10);
Image image(L"image.jpg");
TextureBrush tBrush(&image);

LinearGradientBrush linGrBrush(
Point(30, 50),
Point(100, 50),
Color(255, 255, 0, 0),
Color(255, 0, 0, 255));

WCHAR string[256];
wcscpy(string, L"欢迎使用 GDI+ ! ");

graphics.DrawString(string, (INT)wcslen(string), &font, pointF, &tBrush);
pointF.Y += 50;
graphics.DrawString(string, (INT)wcslen(string), &font, pointF,
&linGrBrush);

```

需要说明的是，在 GDI+中，我们可以通过 `SetTextRenderingHint` 来控制文本输出的质量。例如下面的代码，其结果如图 7.15 所示。

```

Graphics graphics( pDC->m_hDC );

FontFamily fontFamily(L"楷体_GB2312");
Font font(&fontFamily, 30, FontStyleRegular, UnitPixel);
SolidBrush solidBrush(Color(255, 0, 0, 255));
WCHAR string1[] = L"没有任何优化处理";
WCHAR string2[] = L"字体优化，但边不作平滑处理";
WCHAR string3[] = L"消除走样，且边作平滑处理";

graphics.SetTextRenderingHint(TextRenderingHintSingleBitPerPixel);
graphics.DrawString(
string1, (INT)wcslen(string1), &font, PointF(10, 10), &solidBrush);

graphics.SetTextRenderingHint(TextRenderingHintSingleBitPerPixelGridFit);
graphics.DrawString(
string2, (INT)wcslen(string2), &font, PointF(10, 50), &solidBrush);

graphics.SetTextRenderingHint(TextRenderingHintAntiAliasGridFit);
graphics.DrawString(
string3, (INT)wcslen(string3), &font, PointF(10, 90), &solidBrush);

```

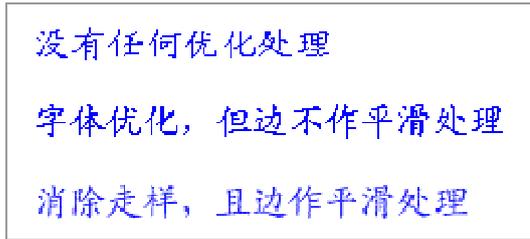


图 7.15 不同文本输出质量的比较

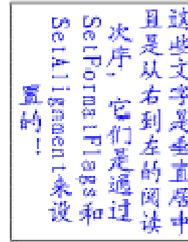


图 7.16 文本格式输出的结果

文本格式化属性

文本的格式属性通常包括对齐方式、字符间隔以及文本调整等。GDI+提供 `StringFormat` 类来控制这些格式属性，通常我们调用以下几个函数来进行相关属性设置。

```
Status SetAlignment( StringAlignment align);
Status SetLineAlignment( StringAlignment align);
```

该函数用来设置文本对齐方式，`align` 可以是 `StringAlignmentNear`(左对齐或右对齐，取决于书写方向是从左到右还是从右到左)、`StringAlignmentCenter`(居中)或 `StringAlignmentFar`(两端对齐)。

```
Status SetFormatFlags( INT flags );
```

该函数用来设置文本格式化标志，`flags` 可以是 `StringFormatFlagsDirectionRightToLeft`(水平阅读方向是从右向左)和 `StringFormatFlagsDirectionVertical`(垂直文本)等值。例如下面的代码，其结果如图 7.16 所示。

```
Graphics graphics( pDC->m_hDC );
SolidBrush solidBrush(Color::Blue);
FontFamily fontFamily(L"楷体_GB2312");
Font font(&fontFamily, 16, FontStyleRegular, UnitPoint);

StringFormat stringFormat;
stringFormat.SetFormatFlags( StringFormatFlagsDirectionRightToLeft |
StringFormatFlagsDirectionVertical |
StringFormatFlagsNoFitBlackBox);
stringFormat.SetAlignment(StringAlignmentCenter);
WCHAR string[] = L"这些文字是垂直居中且是从右到左的阅读次序, 它们是通过 \
SetFormatFlags 和 SetAlignment 来设置的! ";

graphics.DrawString( string, (INT)wcslen(string), &font,
RectF(30, 30, 150, 200), &stringFormat, &solidBrush);
```

我们知道，在以往的图像处理中，常常要根据不同图像文件的格式及其数据存储结构在不同格式中进行转换。某个图像文件的显示也是依靠对文件数据结构的剖析，然后读取相关图像数据而实现的。现在，GDI+提供了 Image 和 Bitmap 类使我们能轻松容易地处理图像。

概述

GDI+支持大多数流行的图像文件格式，如 BMP、GIF、JPEG、TIFF 和 PNG 等。下面先来介绍这些图像文件，然后再说明 Image 和 Bitmap 类支持的特性。

1. 图像文件格式简介

图像文件是描绘一幅图像的计算机磁盘文件，其文件格式不下数十种。这里仅介绍 BMP、GIF、JPEG、TIFF 和 PNG 等图像文件格式。

BMP 文件格式

BMP 图像文件格式是 Microsoft 为其 Windows 环境设置的标准图像格式。一个 Windows 的 BMP 位图实际上是一些和显示像素相对应的位阵列，它有两种类型：一种称之为 GDI 位图，另一种是 DIB 位图(Device-Independent Bitmap)。GDI 位图包含了一种和 Windows 的 GDI 模块有关的 Windows 数据结构，该数据结构是与设备有关的，故此位图又称为 DDB 位图(Device-Dependent Bitmap)。当用户的程序取得位图数据信息时，其位图显示方式视显示卡而定。由于 GDI 位图的这种设备依赖性，当位图通过网络传送到另一台 PC，很可能就会出现问題。

DIB 比 GDI 位图有很多编程优势，例如它自带颜色信息，从而使调色板管理更加容易。且任何运行 Windows 的机器都可以处理 DIB，并通常以后缀为.BMP 的文件形式被保存在磁盘中或作为资源存在于程序的 EXE 或 DLL 文件中。

GIF 文件格式

图形交换格式(GIF--Graphics Interchange Format)最早由 CompuServe 公司于 1987 年 6 月 15 日制定的标准，主要用于 CompuServe 网络图形数据的在线传输、存储。GIF 提供了足够的信息并很好地组织了这些信息，使得许多不同的输入输出设备能够方便地交换图像，它支持 24 位彩色，由一个最多 256 种颜色的调色板实现，图像的大小最多是 64K x 64K 个像点。GIF 的特点是 LZW 压缩、多图像和交错屏幕绘图。

JPEG 文件格式

国际标准化组织(ISO)和国际电报电话咨询委员会(CCITT)联合成立的"联合照片专家组"JPEG(Joint Photographic Experts Group)经过五年艰苦细致工作后，于 1991 年 3 月提出了 ISO CD 10918 号建议草案："多灰度静止图像的数字压缩编码"(通常简称为 JPEG 标准)。这是一个适用于彩色和单色多灰度或连续色调静止数字图像的压缩标准。它包括无损压缩和基于离散余弦变换和 Huffman 编码的有损压缩两个部分。前者不会产生失真，但压缩比很小；后一种算法进行图像压缩时，信息虽有损失但压缩比可以很大。例如压缩 20~40 倍时，

人眼基本上看不出失真。

JPEG 图像文件也是一种像素格式的文件格式，但它比 BMP 等图像文件要复杂许多。所幸的是，GDI+的 Image 提供了对 JPEG 文件格式的支持，使得我们不需要对 JPEG 格式有太多的了解就能处理该格式的图像。

TIFF 文件格式

TIFF(Tagged Image Format File, 标志图像文件格式)最早由 Aldus 公司于 1986 年推出的，它能对从单色到 24 位真彩的任何图像都有很好的支持，而且在不同的平台之间的修改和转换是十分容易的。与其它图像文件格式不同的是，TIFF 文件中有一个标记信息区用来定义文件存储的图像数据类型、颜色和压缩方法。

PNG 文件格式

PNG(Portable Network Graphic, 可移植的网络图像)文件格式是由 Thomas Boutell、Tom Lane 等人提出并设计的，它是为了适应网络数据传输而设计的一种图像文件格式，用于取代格式较为简单、专利限制严格的 GIF 图像文件格式。而且，这种图像文件格式在某种程度上甚至还可以取代格式比较复杂的 TIFF 图像文件格式。它的特点主要有：压缩效率通常比 GIF 要高、提供 Alpha 通道控制图像的透明度、支持 Gamma 校正机制用来调整图像的亮度等。

需要说明的是，PNG 文件格式支持三种主要的图像类型：真彩色图像、灰度级图像以及颜色索引数据图像。JPEG 只支持前两种图像类型，而 GIF 虽然可以利用灰度调色板补偿图像的灰度级别，但原则上它仅仅支持第三种图像类型。

Image 和 Bitmap 类概述

GDI+的 Image 类封装了对 BMP、GIF、JPEG、PNG、TIFF、WMF(Windows 元文件)和 EMF(增强 WMF)图像文件的调入、格式转换以及简单处理的功能。而 Bitmap 是从 Image 类继承的一个图像类，它封装了 Windows 位图操作的常用功能。例如，Bitmap::SetPixel 和 Bitmap::GetPixel 分别用来对位图进行读写像素操作，从而可以为图像的柔化和锐化处理提供一种可能。

3. DrawImage 方法

DrawImage 是 GDI+的 Graphics 类显示图像的核心方法，它的重载函数有许多个。常用的一般重载函数有：

```
Status DrawImage( Image* image, INT x, INT y);  
Status DrawImage( Image* image, const Rect& rect);  
Status DrawImage( Image* image, const Point* destPoints, INT count);  
Status DrawImage( Image* image, INT x, INT y, INT srcx, INT srcy,  
INT srcwidth, INT srcheight, Unit srcUnit);
```

其中，(x,y)用来指定图像 image 显示的位置，这个位置和 image 图像的左上角点相对应。rect 用来指定被图像填充的矩形区域， destPoints 和 count 分别用来指定一个多边形的顶点和顶点个数。若 count 为 3 时，则表示该多边形是一个平行四边形，另一个顶点由系统自动给出。此时， destPoints 中的数据依次对应于源图像的左上角、右上角和左下角的顶点坐标。srcx、srcy、srcwidth 和 srcheight 用来指定要显示的源图像的位置和大小，srcUnit 用来指定所使用的单位，默认时使用 PageUnitPixel，即用像素作为度量单位。

调用和显示图像文件

在 GDI+中调用和显示图像文件是非常容易的，一般先通过 Image 或 Bitmap 调入一个图像文件构造一个对象，然后调用 Graphics::DrawImage 方法在指定位置处显示全部或部分图像。例如下面的代码：

```
void CEx_GDIPlusView::OnDraw(CDC* pDC)
{
    CEx_GDIPlusDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    using namespace Gdiplus;
    Graphics graphics( pDC->m_hDC );

    Image image(L"sunflower.jpg");
    graphics.DrawImage(&image, 10,10);

    Rect rect(130, 10, image.GetWidth(), image.GetHeight());
    graphics.DrawImage(&image, rect);
}
```

结果如图 7.17 所示，从图中我们可以看出，两次 DrawImage 的结果是不同的，按理应该相同，这是怎么回事？原来，DrawImage 在不指定显示区域大小时会自动根据设备分辨率进行缩放，从而造成显示结果的不同。



图 7.17 sunflower.jpg 显示的结果

当然，也可以使用 `Bitmap` 类来调入图像文件来构造一个 `Bitmap` 对象，其结果也是一样的。例如，上述代码可改为：

```
Bitmap bmp(L"sunflower.jpg");
graphics.DrawImage(&bmp, 10,10);

Rect rect(130, 10, bmp.GetWidth(), bmp.GetHeight());
graphics.DrawImage(&bmp, rect);
```

需要说明的是，`Image` 还提供 `GetThumbnailImage` 的方法用来获得一个缩略图的指针，调用 `DrawImage` 后可将该缩略图显示，这在图像预览时极其有用。例如下面的代码：

```
Graphics graphics( pDC->m_hDC );
Image image(L"sunflower.jpg");
Image* pThumbnail = image.GetThumbnailImage(50, 50, NULL, NULL);

// 显示缩略图
graphics.DrawImage(pThumbnail, 20, 20);

// 使用后，不要忘记删除该缩略图指针
delete pThumbnail;
```

图像旋转和拉伸

图像的旋转和拉伸通常是通过在 `DrawImage` 中指定 `destPoints` 参数来实现，`destPoints` 包含对新的坐标系定义的点的数据。图 7.18 说明了坐标系定义的方法。



图 7.18 destPoints 定义坐标系的方法

从图中可以看出，`destPoints` 中的第一个点是用来定义坐标原点的，第二点用来定义 X 轴的方法和图像 X 方向的大小，第三个是用来定义 Y 轴的方法和图像 Y 方向的大小。若 `destPoints` 定义的新坐标系中两轴方向不垂直，就能达到图像拉伸的效果。

下面的代码就是图像旋转和拉伸的一个示例，其结果如图 7.19 所示。

```
Image image(L"sunflower.jpg");
graphics.DrawImage(&image, 10,10);

Point points[] = { Point(0, 0), Point(image.GetWidth(), 0),
Point(0, image.GetHeight())};
Matrix matrix(1,0,0,1,230,10); // 定义一个单位矩阵，坐标原点在(230,10)
matrix.Rotate(30); // 顺时针旋转 30 度

matrix.Scale(0.63,0.6); // X 和 Y 方向分别乘以 0.63 和 0.6 比例因子
matrix.TransformPoints(points, 3); // 用该矩阵转换 points
graphics.DrawImage(&image, points, 3);

Point newpoints[] = {Point(450, 10), Point(510, 60), Point(350, 80)};
graphics.DrawImage(&image, newpoints, 3);
```

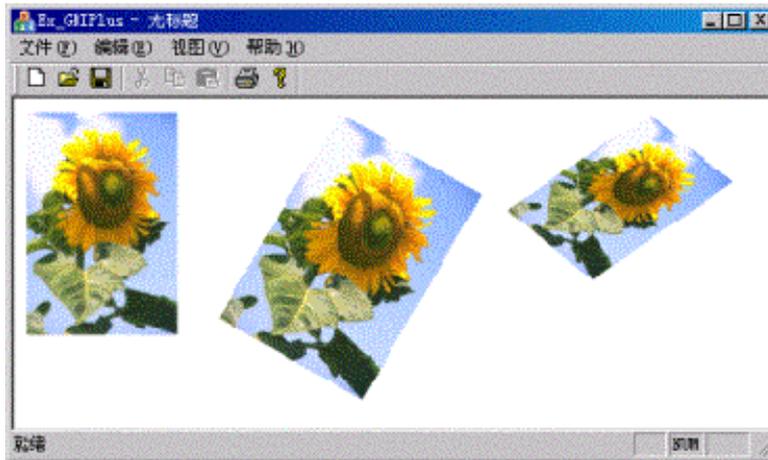


图 7.19 图像旋转和插补的结果

当然，对于图像旋转还可直接使用 `Graphics::RotateTransform` 来进行，例如下面的代码。但这样设置后，以后所有的绘图结果均会旋转，有时可能感觉不方便。

```
Image image(L"sunflower.jpg");
graphics.TranslateTransform(230,10); // 将原点移动到(230,10)
graphics.RotateTransform(30); // 顺时针旋转 30 度
graphics.DrawImage(&image, 0,0);
```

调整插补算法的质量

当图像进行缩放时，需要对图像像素进行插补，不同的插补算法其效果是不一样的。`Graphics::SetInterpolationMode` 可以让我们根据自己的需要使用不同质量效果的插补算法。当然，质量越高，其渲染时间越长。下面的代码就是使用不同质量效果的插补算法模式，其结果如图 7.20 所示。

```
Graphics graphics( pDC->m_hDC );
Image image(L"log.gif");
UINT width = image.GetWidth();
UINT height = image.GetHeight();

// 不进行缩放
graphics.DrawImage( &image,10,10);

// 使用低质量的插补算法
graphics.SetInterpolationMode(InterpolationModeNearestNeighbor);
graphics.DrawImage( &image,
Rect(170, 30, (INT)(0.6*width), (INT)(0.6*height)));

// 使用中等质量的插补算法
```

```
graphics.SetInterpolationMode(InterpolationModeHighQualityBilinear);
graphics.DrawImage( &image,
Rect(270, 30, (INT)(0.6*width), (INT)(0.6*height)));
```

// 使用高质量的插补算法

```
graphics.SetInterpolationMode(InterpolationModeHighQualityBicubic);
graphics.DrawImage( &image,
Rect(370, 30, (INT)(0.6*width), (INT)(0.6*height)));
```



图 7.20 插补的不同质量效果

事实上, Image 功能还不止这些, 例如还有不同格式文件之间的转换等。但这些功能和 MFC 的新类 CImage 功能基本一样, 但 CImage 更符合 MFC 程序员的编程习惯, 因此下一节中我们来重点介绍 CImage 的使用方法和技巧。

我们知道, Visual C++ 的 CBitmap 类和静态图片控件的功能是比较弱的, 它只能显示出在资源中的图标、位图、光标以及图元文件的内容, 而不像 VB 中的 Image 控件可以显示出绝大多数的外部图像文件(BMP、GIF、JPEG 等)。因此, 想要在对话框或其他窗口中显示外部图像文件则只能借助于第三方提供的控件或代码。衷冢浮 FC 和 ATL 共享的新类 CImage 为图像处理提供了许多相应的方法, 这使得 Visual C++ 在图像方面的缺憾一去不复返了。

CImage 类概述

CImage 是 MFC 和 ATL 共享的新类, 它能从外部磁盘中调入一个 JPEG、GIF、BMP 和 PNG 格式的图像文件加以显示, 而且这些文件格式可以相互转换。由于 CImage 在不同的 Windows 操作系统中其某些性能是不一样的, 因此在使用时要特别注意。例如, CImage::PlgBlit 和 CImage::MaskBlit 只能在 Windows NT 4.0 或更高版本中使用, 但不能运行在 Windows 95/98 应用程序中。CImage::AlphaBlend 和 CImage::TransparentBlit 也只能在 Windows 2000/98 或其更高版本中使用。即使在 Windows 2000 运行程序还必须将 stdafx.h 文件中的 WINVER 和 _WIN32_WINNT 的预定义修改成 0x0500 才能正常使用。

CImage 封装了 DIB(设备无关位图)的功能, 因而可以让我们能够处理每个位图像素。它具有下列最酷特性:

- 1、AlphaBlend 支持像素级的颜色混合, 从而实现透明和半透明的效果。

2、**PlgBlt** 能使一个矩形区域的位图映射到一个平行四边形区域中，而且还可能使用位屏蔽操作。

3、**TransparentBlt** 在目标区域中产生透明图像，**SetTransparentColor** 用来设置某种颜色是透明色。

4、**MaskBlt** 在目标区域中产生源位图与屏蔽位图合成的效果。

使用 **CImage** 的一般方法

使用 **CImage** 的一般方法是这样的过程：

(1) 打开应用程序的 **stdafx.h** 文件添加 **CImage** 类的包含文件：

```
#include <atlimage.h>
```

(2) 定义一个 **CImage** 类对象，然后调用 **CImage::Load** 方法装载一个外部图像文件。

(3) 调用 **CImage::Draw** 方法绘制图像。**Draw** 方法具有如下定义：

```
BOOL Draw( HDC hDestDC, int xDest, int yDest,  
int nDestWidth, int nDestHeight, int xSrc, int ySrc,  
int nSrcWidth, int nSrcHeight );  
BOOL Draw( HDC hDestDC, const RECT& rectDest, const RECT& rectSrc );  
BOOL Draw( HDC hDestDC, int xDest, int yDest );  
BOOL Draw( HDC hDestDC, const POINT& pointDest );  
BOOL Draw( HDC hDestDC, int xDest, int yDest,  
int nDestWidth, int nDestHeight );  
BOOL Draw( HDC hDestDC, const RECT& rectDest );
```

其中，**hDestDC** 用来指定绘制的目标设备环境句柄，(**xDest**, **yDest**)和 **pointDest** 用来指定图像显示的位置，这个位置和源图像的左上角点相对应。**nDestWidth** 和 **nDestHeight** 分别指定图像要显示的高度和宽度，**xSrc**、**ySrc**、**nSrcWidth** 和 **nSrcHeight** 用来指定要显示的源图像的某个部分所在的位置和大小。**rectDest** 和 **rectSrc** 分别用来指定目标设备环境和源图像所要显示的某个部分的位置和大小。

需要说明的是，**Draw** 方法综合了 **StretchBlt**、**TransparentBlt** 和 **AlphaBlend** 函数的功能。默认时，**Draw** 的功能和 **StretchBlt** 相同。但当图像含有透明色或 **Alpha** 通道时，它的功能又和 **TransparentBlt**、**AlphaBlend** 相同。因此，在一般情况下，我们都应该尽量调用 **CImage::Draw** 方法来绘制图像。

例如，下面的示例 **Ex_Image** 是实现这样的功能：当选择“文件”→“打开”菜单命令后，弹出一个文件打开对话框。当选定一个图像文件后，就会在窗口客户区中显示该图像文件内容。这个示例的具体步骤如下：

(1) 创建一个默认的单文档程序项 `x_Image`。

(2) 打开 `stdafx.h` 文件中添加 `CImage` 类的包含文件 `atlimage.h`。

(3) 在 `CEx_ImageView` 类添加 `ID_FILE_OPEN` 的 `COMMAND` 事件映射程序, 并添加下列代码:

```
void CEx_ImageView::OnFileOpen()
{
    CString strFilter;
    CSimpleArray<GUID> aguidFileTypes;
    HRESULT hResult;

    // 获取 CImage 支持的图像文件的过滤字符串
    hResult = m_Image.GetExporterFilterString(strFilter,aguidFileTypes,
    _T("All Image Files"));
    if (FAILED(hResult)) {
        MessageBox("GetExporterFilter 调用失败! ");
        return;
    }
    CFileDialog dlg(TRUE, NULL, NULL, OFN_FILEMUSTEXIST, strFilter);
    if(IDOK != dlg.DoModal())
        return;

    m_Image.Destroy();
    // 将外部图像文件装载到 CImage 对象中
    hResult = m_Image.Load(dlg.GetFileName());
    if (FAILED(hResult)) {
        MessageBox("调用图像文件失败! ");
        return;
    }

    // 设置主窗口标题栏内容
    CString str;
    str.LoadString(AFX_IDS_APP_TITLE);
    AfxGetMainWnd()->SetWindowText(str + " - " +dlg.GetFileName());

    Invalidate(); // 强制调用 OnDraw
}
```

(4) 定位到 `CEx_ImageView::OnDraw` 函数处, 添加下列代码:

```
void CEx_ImageView::OnDraw(CDC* pDC)
{
```

```

CEx_ImageDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
if (!m_Image.IsNull()) {
    m_Image.Draw(pDC->m_hDC,0,0);
}
}
}

```

(5) 打开 Ex_ImageView.h 文件，添加一个公共的成员数据 m_Image:

```

public:
CImage m_Image;

```

(6) 编译并运行。单击"打开"工具按钮，在弹出的对话框中指定一个图像文件后，单击"打开"按钮，其结果如图 7.21 所示。

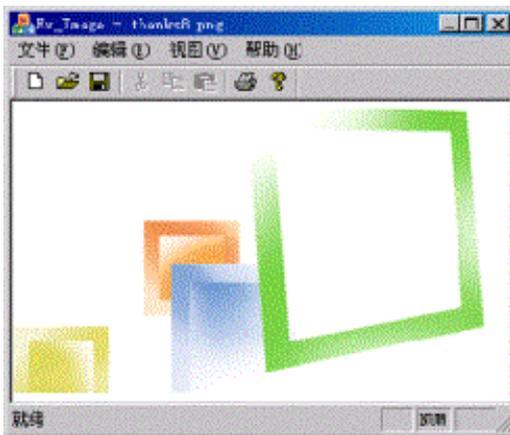


图 7.21 图像文件的显示

将图片用其它格式保存

CImage::Save 方法能将一个图像文件按另一种格式来保存，它的原型如下：

```

HRESULT Save( LPCTSTR pszFileName, REFGUID guidFileType= GUID_NULL);

```

其中，pszFileName 用来指定一个文件名，guidFileType 用来指定要保存的图像文件格式，当为 GUID_NULL 时，其文件格式由文件的扩展名来决定，这也是该函数的默认值。它还可以是 GUID_BMPFile(BMP 文件格式)、GUID_PNGFile(PNG 文件格式)、GUID_JPEGFile(JPEG 文件格式)和 GUID_GIFFile(GIF 文件格式)。

例如，下面的过程是在 Ex_Image 示例基础上进行的，我们在 CEx_ImageView 类添加 ID_FILE_SAVE_AS 的 COMMAND 事件映射程序，并添加下列代码：

```

void CEx_ImageView::OnFileSaveAs()
{
    if (m_Image.IsNull()) {

```

```

    MessageBox("你还没有打开一个要保存的图像文件！");
    return;
}

CString strFilter;
strFilter = "位图文件|.bmp|JPEG 图像文件|.jpg| \
GIF 图像文件|.gif|PNG 图像文件|.png|";
CFileDialog dlg(FALSE, NULL, NULL, NULL, strFilter);
if ( IDOK != dlg.DoModal())
    return;

// 如果用户没有指定文件扩展名, 则为其添加一个
CString strFileName;
CString strExtension;

strFileName = dlg.m_ofn.lpstrFile;
if (dlg.m_ofn.nFileExtension == 0)
{
    switch (dlg.m_ofn.nFilterIndex)
    {
        case 1:
            strExtension = "bmp"; break;
        case 2:
            strExtension = "jpg"; break;
        case 3:
            strExtension = "gif"; break;
        case 4:
            strExtension = "png"; break;
        default:
            break;
    }
    strFileName = strFileName + '.' + strExtension;
}

// 图像保存
HRESULT hResult = m_Image.Save(strFileName);
if (FAILED(hResult))
    MessageBox("保存图像文件失败！");
}

```

柔化和锐化处理

在图像处理中, 我们通常用一些数学手段, 对图像进行除去噪声、强调或抽取轮廓特征等图像空间的变换。所谓"图像空间的变换"是借助于一个称之为模板的局部像素域来完成

化和锐化模板进行处理，最后调用 `CImage::SetPixel` 函数将处理后的像素写回到 `CImage` 对象中。具体的代码如下：

```
void FilterImage(CImage* image, int nType)
{
    if (image->IsNull())
        return;
    int smoothGauss[9] = {1,2,1,2,4,2,1,2,1}; // 高斯模板
    int sharpLaplacian[9] = {-1,-1,-1,-1,9,-1,-1,-1,-1}; // 拉普拉斯模板

    int opTemp[9];
    float aver; // 系数
    if ( nType > 1) nType = 0;
    switch( nType ){
        case 0: // 高斯模板
            aver = (float)(1.0/16.0);
            memcpy( opTemp, smoothGauss, 9*sizeof(int));
            break;
        case 1: // 拉普拉斯模板
            aver = 1.0;
            memcpy( opTemp, sharpLaplacian, 9*sizeof(int));
            break;
    }

    int i,j;

    int nWidth = image->GetWidth();
    int nHeight = image->GetHeight();

    for (i = 1; i < nWidth-1; i++){
        for (j = 1; j < nHeight-1; j++){

            int rr = 0, gg = 0, bb = 0;
            int index = 0;

            for (int col = -1; col <= 1; col++){
                for (int row = -1; row <= 1; row++){
                    COLORREF clr = image->GetPixel( i+row, j+col);
                    rr += GetRValue(clr) * opTemp[index];
                    gg += GetGValue(clr) * opTemp[index];
                    bb += GetBValue(clr) * opTemp[index];
                    index++;
                }
            }
        }
    }
```

```

rr = (int)(rr*aver);
gg = (int)(gg*aver);
bb = (int)(bb*aver);

// 处理溢出点
if ( rr > 255 ) rr = 255;
else if ( rr < 0 ) rr = -rr;
if ( gg > 255 ) gg = 255;
else if ( gg < 0 ) gg = -gg;
if ( bb > 255 ) bb = 255;
else if ( bb < 0 ) bb = -bb;

// 错位重写以避免前一个像素被新的像素覆盖
image->SetPixel( i-1, j-1, RGB(rr,gg,bb));
}
}
}

```

图 7.22 是使用上述代码将某个图像处理后的结果。



图 7.22 图像柔化和锐化效果

变成黑白图片

由于许多图像文件使用颜色表来发挥显示设备的色彩显示能力,因而将一张彩色图片变成黑色图片时需要调用 `CImage::IsIndexed` 来判断是否使用颜色表,若是则修改颜色表,否则直接将像素进行颜色设置。例如下面的代码:

```

void CEx_ImageView::MakeBlackAndwhite(CImage* image)
{
    if (image->IsNull()) return;

    if (!image->IsIndexed()) {

```

```

// 直接修改像素颜色
COLORREF pixel;
int maxY = image->GetHeight(), maxX = image->GetWidth();
byte r,g,b,avg;
for (int x=0; x<maxX; x++) {
    for (int y=0; y<maxY; y++) {
        pixel = image->GetPixel(x,y);
        r = GetRValue(pixel);
        g = GetGValue(pixel);
        b = GetBValue(pixel);
        avg = (int)((r + g + b)/3);
        image->SetPixelRGB(x,y,avg,avg,avg);
    }
}
} else {
// 获取并修改颜色表
int MaxColors = image->GetMaxColorTableEntries();
RGBQUAD* ColorTable;
ColorTable = new RGBQUAD[MaxColors];
image->GetColorTable(0,MaxColors,ColorTable);
for (int i=0; i<MaxColors; i++)
{
    int avg = (ColorTable[i].rgbBlue + ColorTable[i].rgbGreen + ColorTable[i].rgbRed)/3;
    ColorTable[i].rgbBlue = avg;
    ColorTable[i].rgbGreen = avg;
    ColorTable[i].rgbRed = avg;
}
image->SetColorTable(0,MaxColors,ColorTable);
delete(ColorTable);
}
}

```

至此，我们介绍了 **GDI+**和 **CImage** 的一般使用方法和技巧。当然，它们本身还有许多更深入的方法，由于篇幅所限，这里不再一一讨论。