

2016

Frankie

[GDI+ MADE FLAT – PART 1]

[A guide to program GDI+ with flat API's and plain 'C' language]

Contents

1	GDI+ Abstract	3
1.1	Disclaimer	3
1.2	Purpose	3
1.3	Where applicable	3
1.4	Developer audience	3
1.5	Run-time requirements	3
2	GDI+ Flat: Using C++ Methods under plain C	5
2.1	Overloading and polymorphism under Flat API's	6
3	Security Considerations: GDI+	8
3.1	Verifying the Success of Constructors	8
3.2	Allocating Buffers	8
3.3	Error Checking	9
3.4	Thread Synchronization	11
4	About GDI+	12
4.1	Overview of GDI+	12
4.2	The Three Parts of GDI+	12
4.2.1	2-D vector graphics	12
4.2.2	Imaging	13
4.2.3	Typography	13
4.3	The Structure of the Class-Based Interface	13
5	What's New In GDI+?	14
5.1	New Features	14
5.1.1	Gradient Brushes	14
5.1.2	Cardinal Splines	14
5.1.3	Independent Path Objects	15
5.1.4	Transformations and the Matrix Object	15
5.1.5	Scalable Regions	15
5.1.6	Alpha Blending	15
5.1.7	Support for Multiple Image Formats	15
5.2	Changes in the Programming Model	16
5.2.1	Device Contexts, Handles, and Graphics Objects	16

5.2.2	Two Ways to Draw a Line	17
5.2.3	Drawing a line with GDI	17
5.2.4	Drawing a line with GDI+ and the flat API's	17
5.2.5	Pens, Brushes, Paths, Images, and Fonts as Parameters	17
5.2.6	Method Overloading in flat API's	18
5.2.7	No More Current Position	18
5.2.8	Separate Methods for Draw and Fill	18
5.2.9	Constructing Regions	19
6	Lines, Curves, and Shapes	20
6.1	Overview of Vector Graphics	20
6.2	Pens, Lines, and Rectangles	21
6.3	Ellipses and Arcs	22
6.4	Polygons	23
6.5	Cardinal Splines	23
6.6	Bezier Splines	24
6.7	Paths	25
6.8	Brushes and Filled Shapes	26
6.8.1	Solid Brushes	26
6.8.2	Hatch Brushes	27
6.8.3	Texture Brushes	27
6.8.4	Gradient Brushes	27
6.9	Open and Closed Curves	28
6.10	Regions	29
6.11	Clipping	30
6.12	Flattening Paths	30
6.13	Antialiasing with Lines and Curves	31
7	Images, Bitmaps, and Metafiles	32
7.1	Types of Bitmaps	32
7.2	Graphics File Formats	33
7.3	Metafiles	35
7.4	Drawing, Positioning, and Cloning Images	36
7.5	Cropping and Scaling Images	37

1 GDI+ ABSTRACT

1.1 Disclaimer

This is not an original work; almost all materials come from Microsoft documentation on GDI+ classes.

The aim of this paper is to provide documents as close to standard docs as possible, but adapted for the use of GDI+ flat API's and plain 'C' programming.

This choice comes from knowledge that MS doesn't support directly flat API's and that this paper cannot be in any case exhaustive. For this reasons having a paper close to the formal documentation permits a direct compare with the C++ version (or the C# one that is becoming even more supported), that should train the user on how to convert, by himself, any code or function that is not described here.

In hope that this could be of any help, enjoy!

1.2 Purpose

Windows GDI+ is a class-based API for C/C++ programmers. It enables applications to use graphics and formatted text on both the video display and the printer. Applications based on the Microsoft Win32 API do not access graphics hardware directly. Instead, GDI+ interacts with device drivers on behalf of applications. GDI+ is also supported by Microsoft Win64.

GDI+ can be used directly under plain C through the so called GDI+ flat API's. They are a collection of 600+ functions that acts as wrappers around the C++ core. Unfortunately even the flat API's are not directly compatible with plain 'C' because many required definitions are in C++ format or mixed in C++ headers that cannot be directly included in plain code.

This is the reason for the existence of "***fGdiPlusFlat.h***" header.

This paper, that follows the standard GDI+ documentation, reproduces the same examples using flat C and the header "***fGdiPlusFlat.h***".

1.3 Where applicable

GDI+ functions and classes are not supported for use within a Windows service. Attempting to use these functions and classes from a Windows service may produce unexpected problems, such as diminished service performance and run-time exceptions or errors.

Note When you use the GDI+ API, you must never allow your application to download arbitrary fonts from untrusted sources. The operating system requires elevated privileges to assure that all installed fonts are trusted.

1.4 Developer audience

The GDI+ flat API's "***fGdiPlusFlat.h***" interface is designed for use by C programmers. Familiarity with the Windows graphical user interface and message-driven architecture is required.

1.5 Run-time requirements

Gdiplus.dll is included in MS operating systems since Windows XP and Windows Server 2003. For information about which operating systems are required to use a particular class or method, see the More Information

section of the documentation for the class or method. GDI+ is available as a redistributable for Microsoft Windows NT 4.0 SP6, Windows 2000, Windows 98, and Windows Millennium Edition (Windows Me). To download the latest redistributable, see <http://go.microsoft.com/fwlink/?LinkId=20993>.

Note If you are redistributing GDI+ to a down-level platform or a platform that does not ship with that version of GDI+ natively, install Gdiplus.dll in your application directory. This puts it in your address space, but you should use the linker's /BASE option to rebase the Gdiplus.dll to prevent address space conflict. For more information, see [/BASE \(Base Address\)](#).

2 GDI+ FLAT: USING C++ METHODS UNDER PLAIN C

To convert the GDI+ objects, constructors, destructors and methods to the flat format has been followed the rules below that resembles that used for the object automation:

- GDI+ objects are represented as opaque incomplete structures.
The only property available is the address of the object that is the pointer to the incomplete structure.
No operations that require the size of the object, as instantiation, can be performed. Objects are create and destroyed exclusively using the constructors and destructors.
- Constructors accept a set of properties and a pointer to the object pointer. They return a GpStatus to check for function success/failure. The values are enumerated in the Status enum.
I.e. to use a Path object we declare a pointer to an object of type Path, then we use the constructor to return an address (a reference to) an instantiate object of type Path:

```
GpPath *path;    //This will hold the pointer to the object
GpStatus sts = GdipCreatePath(FillModeAlternate, &path);    //Instanziate
if (sts != Ok)
    goto error;    //Error instanziating object
```

If the function is successful the variable path holds a reference to a Path object.

In some cases the constructor name is not so direct as in the case of Graphics::CreateFromHDC that become **GpStatus WINGDIPAPI GdipCreateFromHDC(HDC hdc, GpGraphics **graphics);**

- When calling methods in flat API's the first parameter is always the object reference:

```
sts = GdipAddPathLine(path, 0, 0, 10, 10);    //Add a line to path
if (sts != Ok)
    goto error;    //Error
```

- Always Dispose/delete objects after use to relief the memory.

```
sts = GdipDeletePath (path);    //destroy object
if (sts != Ok)
    goto error;    //Error
```

- Always initialize and shutdown the GDI+ environment using the functions:

```
Status WINAPI GdiplusStartup(ULONG_PTR *token,
                             const GdiplusStartupInput *input,
                             GdiplusStartupOutput *output);
VOID WINAPI GdiplusShutdown(ULONG_PTR token);
```

fGdiPlusFlat header includes a default GdiplusStartupInput structure, named **gdiplusStartupInputDef**. This is standard use:

```
ULONG_PTR gdiplusToken;
GdiplusStartup(&gdiplusToken, &gdiplusStartupInputDef, NULL); //GDI+ Startup
... //GDI+ code
GdiplusShutdown(gdiplusToken);    //GDI+Shutdown
```

- The table 1 below reports the main GDI+ objects types as denominated for use with original C++ classes and their correspondent Flat object names.

Please note that you can still use the original names from C++ classes, they have been redefined in **fGdiPlusFlat.h** as pointers to respective objects, and this means that when you use them there is no need to specify that the variable is a pointer to the object type.

I.e. the following declarations are equivalent:

```
GpGraphics *myGraphic;
```

Or

```
Graphics MyGraphic;
```

C++ Classes	Flat 'C' objects
Graphics	GpGraphics
Pen	GpPen
Brush	GpBrush
Matrix	GpMatrix
Bitmap	GpBitmap
Metafile	GpMetafile
GraphicsPath	GpPath
PathIterator	GpPathIterator
Region	GpRegion
Image	GpImage
TextureBrush	GpTexture
HatchBrush	GpHatch
SolidBrush	GpSolidFill
LinearGradientBrush	GpLineGradient
PathGradientBrush	GpPathGradient
Font	GpFont
FontFamily	GpFontFamily
FontCollection	GpFontCollection
InstalledFontCollection	GpInstalledFontCollection
PrivateFontCollection	GpPrivateFontCollection
ImageAttributes	GpImageAttributes
CachedBitmap	GpCachedBitmap

TABLE 1

2.1 Overloading and polymorphism under Flat API's

Under flat API's the overloading is, of course, not available. Anyway many methods, and structures, are present in two versions a **REAL** version and an **INT** one. The first accepts parameter in floating point format and the other in fixed integer format. The two versions are differentiated with a postfix of an uppercase 'I' to the method

```
GpStatus WINGDIPAPI GdiDrawLine(GpGraphics *graphics, GpPen *pen, REAL x1, REAL y1, REAL x2, REAL y2);
GpStatus WINGDIPAPI GdiDrawLineI(GpGraphics *graphics, GpPen *pen, INT x1, INT y1, INT x2, INT y2);
```

The same happen with structs, but the method is reversed using a postfix of an 'F' to the structure name for the floating point one:

```
GpRect myRect = { 0, 0, 10, 20}; //Integer rectangle
GpRectF myRectf = {0.0f, 0.0f, 10.0f, 20.0f} //Float Rectangle
```

Are also present methods to replace multiple executions as in ***GdipDrawLines***:

```
GpStatus WINGDIPAPI GdipDrawLines(GpGraphics *graphics, GpPen *pen, GDIPCONST GpPointF *points, INT count);
GpStatus WINGDIPAPI GdipDrawLinesI(GpGraphics *graphics, GpPen *pen, GDIPCONST GpPoint *points, INT count);
```

Using all those different functions is normally possible to translate 99% of existing C++ programs handling manually the more complex applications of overloading and polymorphism. Anyway is not a straight job, it requires knowledge of C++ and C programming to correctly translate between the two. The most common error happens always to be the forgetting of constructors and, more subtle, to forget to dispose objects.

3 SECURITY CONSIDERATIONS: GDI+

This topic provides information about security considerations related to programming with Windows GDI+. This topic doesn't provide all you need to know about security issues—instead, use it as a starting point and reference for this technology area.

- [Verifying the Success of Constructors](#)
- [Allocating Buffers](#)
- [Error Checking](#)
- [Thread Synchronization](#)
- [Related topics](#)

3.1 Verifying the Success of Constructors

All the GDI+ flat functions returns a status which can be tested to determine whether methods invoked on an object are successful.

The following example shows how to construct a [GpImage](#) object and determine whether the constructor was successful. The values **Ok** and **InvalidParameter** are elements of the [GpStatus](#) enumeration.

```
GpImage *myImage;
GpStatus st = GdipLoadImageFromFile(L"Climber.jpg", & myImage);
if(Ok == st)
    // The constructor was successful. Use myImage.
else if(InvalidParameter == st)
    // The constructor failed because of an invalid parameter.
else
    // Compare st to other elements of the Status
    // enumeration or do general error processing.
```

3.2 Allocating Buffers

Several GDI+ methods return numeric or character data in a buffer that is allocated by the caller. For each of those methods, there is a companion method that gives the size of the required buffer. For example, the [GdipGetPathPoints](#) method returns an array of [GpPoint](#) or [GpPointF](#) objects. Before you call **GdipGetPathPoints**, you must allocate a buffer large enough to hold that array. You can determine the size of the required buffer by calling the [GdipGetPointCount](#) method of a [GraphicsPath](#) object.

The following example shows how to determine the number of points in a [GraphicsPath](#) object, allocate a buffer large enough to hold that many points, and then call [GdipGetPointCount](#) to fill the buffer. Before the code calls [GdipGetPointCount](#), it verifies that the buffer allocation was successful by making sure that the buffer pointer is not **NULL**.

```
GpPath *path;
if (Ok != GdipCreatePath(FillModeAlternate, &path))
    goto error;    //handle error

GdipAddPathEllipseI(path, 10, 10, 200, 100);

INT count;
```

```

if (Ok != GdipGetPointCount(path, &count)) // get the size
    goto error;    //handle error

// allocate the buffer
GpPointF * pointArray = GdipAlloc(sizeof(REAL) * count);
if(pointArray) // Check for successful allocation.
{
    GdipGetPathPoints(path, pointArray, count); // get the data
    ... // use pointArray
    GdipFree(pointArray); // release the buffer
    pointArray = NULL;
}

```

The previous example uses the `GdipAlloc` operator to allocate a buffer. The `GdipAlloc` operator was convenient because the buffer was filled with a known number of `GpPoint` or `GpPointF` objects. In some cases, GDI+ writes more into buffer than an array of GDI+ objects. Sometimes a buffer is filled with an array of GDI+ objects along with additional data that is pointed to by members of those objects. For example, the `GdipGetAllPropertyItems` method returns an array of `PropertyItem` objects, one for each property item (piece of metadata) stored in the image. But `GdipGetAllPropertyItems` returns more than just the array of `PropertyItem` objects; it appends the array with additional data.

Before you call `GdipGetAllPropertyItems`, you must allocate a buffer large enough to hold the array of `PropertyItem` objects along with the additional data. You can call the `GdipGetPropertySize` method of a `GpImage` object to determine the total size of the required buffer.

The following example shows how to create a `GpImage` object and later call the `GdipGetAllPropertyItems` method of that `GpImage` object to retrieve all the property items (metadata) stored in the image. The code allocates a buffer based on a size value returned by the `GdipGetPropertySize` method. `GdipGetPropertySize` also returns a count value that gives the number of property items in the image. Notice that the code does not calculate the buffer size as `count*sizeof(PropertyItem)`. A buffer calculated that way would be too small.

```

UINT count = 0;
UINT size = 0;

GpImage *myImage;
GdipLoadImageFromFile(L"FakePhoto.jpg", &myImage);

GdipGetPropertySize(myImage, &size, &count);

// GetAllPropertyItems returns an array of PropertyItem objects
// along with additional data. Allocate a buffer large enough to
// receive the array and the additional data.
PropertyItem* propBuffer =(PropertyItem*)malloc(size);

if(propBuffer)
{
    GdipGetAllPropertyItems(myImage, size, count, propBuffer);
    ...
    free(propBuffer);
    propBuffer = NULL;
}

```

3.3 Error Checking

Most of the code examples in the GDI+ documentation do not show error checking. Complete error checking makes a code example much longer and can obscure the point being illustrated by the example. You should not paste examples from the documentation directly into production code; rather, you should enhance the examples by adding your own error checking.

The following example shows one way of implementing error checking with GDI+. Each time a GDI+ object is constructed, the code checks to see whether the constructor was successful. That check is especially important for the **GpImage** constructor, which relies on reading a file. If all four of the GDI+ objects (**GpGraphics**, **GpPath**, **GpImage**, and **GpTexture**) are constructed successfully, the code calls methods on those objects. Each method call is checked for success, and in the event of failure, the remaining method calls are skipped.

```
GpStatus GdipExample1(HDC hdc)
{
    GpStatus status = GenericError;
    INT count = 0;
    GpPoint* points = NULL;

    GpGraphics *graphics;
    status = GdipCreateFromHDC(hdc, &graphics);
    if(Ok != status)
        return status;

    GpPath *path;
    status = GdipCreatePath(FillModeAlternate, &path);
    if(Ok != status)
        return status;

    GpImage *image;
    status = GdipLoadImageFromFile(L"MyTexture.jpg", &image);
    if(Ok != status)
        return status;

    GpTexture *brush;
    status = GdipCreateTexture(image, FillModeAlternate, &brush);
    if(Ok != status)
        return status;

    status = GdipAddPathEllipseI(path, 10, 10, 200, 100);

    if(Ok == status)
    {
        status = GdipAddPathBezierI (path, 40, 130, 200, 130, 200, 200, 60, 200);
    }

    if(Ok == status)
    {
        status = GdipGetPointCount(path, &count);
    }

    if(Ok == status)
    {
        points = GdipAlloc(sizeof(GpPoint) * count);
        if(NULL == points)
            status = OutOfMemory;
    }

    if(Ok == status)
    {

```

```

        status = GdipGetPathPointsI(path, points, count);
    }

    if(Ok == status)
    {
        status = GdipFillPath(graphics, (GpBrush *)brush, path);
    }

    if(Ok == status)
    {
        for(int j = 0; j < count; ++j)
        {
            status = GdipFillEllipseI(graphics,
                                      (GpBrush *)brush, points[j].X - 5, points[j].Y - 5, 10, 10);
        }
    }

    if(points)
    {
        GdipFree(points);
        points = NULL;
    }

    return status;
}

```

3.4 Thread Synchronization

It is possible for more than one thread to have access to a single GDI+ object. However, GDI+ does not provide any automatic synchronization mechanism. So if two threads in your application have a pointer to the same GDI+ object, it is your responsibility to synchronize access to that object.

Some GDI+ methods return **ObjectBusy** if a thread attempts to call a method while another thread is executing a method on the same object. Do not try to synchronize access to an object based on the **ObjectBusy** return value. Instead, each time you access a member or call a method of the object, place the call inside a critical section, or use some other standard synchronization technique.

4 ABOUT GDI+

Windows GDI+ is the graphic support that, since Windows XP operating system or Windows Server 2003 operating system, provides two-dimensional vector graphics, imaging, and typography. GDI+ improves on Windows Graphics Device Interface (GDI) (the graphics device interface included with earlier versions of Windows) by adding new features and by optimizing existing features.

The following topics provide information about the GDI+ flat API's with the C programming language.

- [Introduction to GDI+](#)
- [What's New In GDI+?](#)
- [Lines, Curves, and Shapes](#)
- [Images, Bitmaps, and Metafiles](#)
- [Coordinate Systems and Transformations](#)
- [Graphics Containers](#)

4.1 Overview of GDI+

Windows GDI+ is the graphic subsystem that, since Windows XP operating system or Windows Server 2003, is responsible for displaying information on screens and printers. GDI+ is an API that is exposed through a set of C++ classes, but also comes with a flat API, composed of 600+ functions and structs and enums, that can be interfaced with plain C.

As its name suggests, GDI+ is the successor to Windows Graphics Device Interface (GDI), the graphics device interface included with earlier versions of Windows. Windows XP or Windows Server 2003 supports GDI for compatibility with existing applications, but programmers of new applications should use GDI+ for all their graphics needs because GDI+ optimizes many of the capabilities of GDI and also provides additional features.

A graphics device interface, such as GDI+, allows application programmers to display information on a screen or printer without having to be concerned about the details of a particular display device. The application programmer makes calls to methods provided by GDI+ classes and those methods in turn make the appropriate calls to specific device drivers. GDI+ insulates the application from the graphics hardware, and it is this insulation that allows developers to create device-independent applications.

4.2 The Three Parts of GDI+

The services of Windows GDI+ fall into the following three broad categories:

- [2-D vector graphics](#)
- [Imaging](#)
- [Typography](#)

4.2.1 2-D VECTOR GRAPHICS

Vector graphics involves drawing primitives (such as lines, curves, and figures) that are specified by sets of points on a coordinate system. For example, a straight line can be specified by its two endpoints, and a rectangle can be specified by a point giving the location of its upper-left corner and a pair of numbers giving its width and height. A simple path can be specified by an array of points to be connected by straight lines. A Bézier spline is a sophisticated curve specified by four control points.

GDI+ provides classes that store information about the primitives themselves, classes that store information about how the primitives are to be drawn, and classes that actually do the drawing. For example, the **GpRect** class stores the location and size of a rectangle; the **GpPen** class stores information about line color, line width, and line style; and the **GpGraphics** class has methods for drawing lines, rectangles, paths, and other figures. There are also several **GpBrush** classes that store information about how closed figures and paths are to be filled with colors or patterns.

4.2.2 IMAGING

Certain kinds of pictures are difficult or impossible to display with the techniques of vector graphics. For example, the pictures on toolbar buttons and the pictures that appear as icons would be difficult to specify as collections of lines and curves. A high-resolution digital photograph of a crowded baseball stadium would be even more difficult to create with vector techniques. Images of this type are stored as bitmaps, arrays of numbers that represent the colors of individual dots on the screen. Data structures that store information about bitmaps tend to be more complex than those required for vector graphics, so there are several classes in GDI+ devoted to this purpose. An example of such a class is **GpCachedBitmap**, which is used to store a bitmap in memory for fast access and display.

4.2.3 TYPOGRAPHY

Typography is concerned with the display of text in a variety of fonts, sizes, and styles. GDI+ provides an impressive amount of support for this complex task. One of the new features in GDI+ is subpixel antialiasing, which gives text rendered on an LCD screen a smoother appearance.

4.3 The Structure of the Class-Based Interface

The C++ interface to Windows GDI+ contains about 40 classes, 50 enumerations, and 6 structures. There are also a few functions that are not members of any class.

The **GpGraphics** class is the core of the GDI+ interface; it is the class that actually draws lines, curves, figures, images, and text.

Many classes work together with the **GpGraphics** class. For example, the **GdiplusDrawLine** method receives a pointer to a **GpPen** object, which holds attributes (color, width, dash style, and the like) of the line to be drawn. The **GdiplusFillRectangle** method can receive a pointer to a **GpLineGradient** object, which works with the **GpGraphics** object to fill a rectangle with a gradually changing color. **GpFont** and **GpStringFormat** objects influence the way a **GpGraphics** object draws text. A **GpMatrix** object stores and manipulates the world transformation of a **GpGraphics** object, which is used to rotate, scale, and flip images.

Certain classes serve primarily as structured data types. Some of those classes (for example, **GpRect**, **GpPoint**, and **Size**) are for general purposes. Others are for specialized purposes and are considered helper classes. For example, the **BitmapData** class is a helper for the **GpBitmap** class, and the **PathData** class is a helper for the **GraphicsPath** class. GDI+ also defines a few structures that are used for organizing data. For example, the **ColorMap** structure holds a pair of **Color** objects that form one entry in a color conversion table.

GDI+ defines several enumerations, which are collections of related constants. For example, the **LineJoin** enumeration contains the elements **LineJoinBevel**, **LineJoinMiter**, and **LineJoinRound**, which specify styles that can be used to join two lines.

GDI+ provides a few functions that are not part of any class. Two of those functions are **GdiplusStartup** and **GdiplusShutdown**. You must call **GdiplusStartup** before you make any other GDI+ calls, and you must call **GdiplusShutdown** when you have finished using GDI+.

5 WHAT'S NEW IN GDI+?

Windows GDI+ is different from Windows Graphics Device Interface (GDI) in a couple of ways. First, GDI+ expands on the features of GDI by providing new capabilities, such as gradient brushes and alpha blending. Second, the programming model has been revised to make graphics programming easier and more flexible.

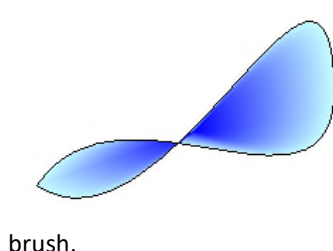
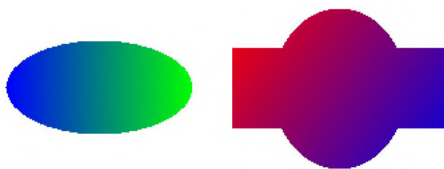
5.1 New Features

The following sections describe several of the new features in Windows GDI+.

- [Gradient Brushes](#)
- [Cardinal Splines](#)
- [Independent Path Objects](#)
- [Transformations and the Matrix Object](#)
- [Scalable Regions](#)
- [Alpha Blending](#)
- [Support for Multiple Image Formats](#)

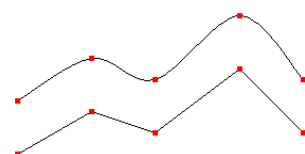
5.1.1 GRADIENT BRUSHES

GDI+ expands on Windows Graphics Device Interface (GDI) by providing linear gradient and path gradient brushes for filling shapes, paths, and regions. Gradient brushes can also be used to draw lines, curves, and paths. When you fill a shape with a linear gradient brush, the color gradually changes as you move across the shape. For example, suppose you create a horizontal gradient brush by specifying blue at the left edge of a shape and green at the right edge. When you fill that shape with the horizontal gradient brush, it will gradually change from blue to green as you move from its left edge to its right edge. Similarly, a shape filled with a vertical gradient brush will change color as you move from top to bottom. The following illustration shows an ellipse filled with a horizontal gradient brush and a region filled with a diagonal gradient brush.



When you fill a shape with a path gradient brush, you have a variety of options for specifying how the colors change as you move from one portion of the shape to another. One option is to have a center color and a boundary color so that the pixels change gradually from one color to the other as you move from the middle of the shape towards the outer edges. The following illustration shows a path (created from a pair of Bézier splines) filled with a path gradient brush.

5.1.2 CARDINAL SPLINES



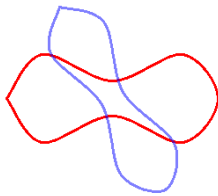
GDI+ supports cardinal splines, which are not supported in GDI. A cardinal spline is a sequence of individual curves joined to form a larger curve. The spline is specified by an array of points and passes through each point in that array. A

cardinal spline passes smoothly (no sharp corners) through each point in the array and thus is more refined than a path created by connecting straight lines. The illustration shows two paths, one created by connecting straight lines and one created as a cardinal spline.

5.1.3 INDEPENDENT PATH OBJECTS

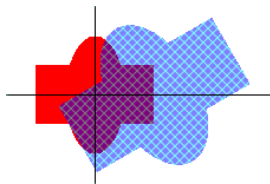
In GDI, a path belongs to a device context, and the path is destroyed as it is drawn. With GDI+, drawing is performed by a **GpGraphics** object, and you can create and maintain several **GraphicsPath** objects that are separate from the **GpGraphics** object. A **GraphicsPath** object is not destroyed by the drawing action, so you can use the same **GraphicsPath** object to draw a path several times.

5.1.4 TRANSFORMATIONS AND THE MATRIX OBJECT



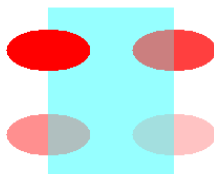
GDI+ provides the **Matrix** object, a powerful tool that makes transformations (rotations, translations, and so on) easy and flexible. A matrix object works in conjunction with the objects that are transformed. For example, a **GpPath** object has a **GdipTransformPath** method that receives the address of a **GpMatrix** object as an argument. A single 3×3 matrix can store one transformation or a sequence of transformations. The illustration shows a path before and after a sequence of two transformations (first scale, then rotate).

5.1.5 SCALABLE REGIONS



GDI+ expands greatly on GDI with its support for regions. In GDI, regions are stored in device coordinates, and the only transformation that can be applied to a region is a translation. GDI+ stores regions in world coordinates and allows a region to undergo any transformation (scaling, for example) that can be stored in a transformation matrix. The following illustration shows a region before and after a sequence of three transformations: scale, rotate, and translate.

5.1.6 ALPHA BLENDING



Note that in the previous figure, you can see the untransformed region (filled with red) through the transformed region (filled with a hatch brush). This is made possible by alpha blending, which is supported by GDI+. With alpha blending, you can specify the transparency of a fill color. A transparent color is blended with the background color — the more transparent you make a fill color, the more the background shows through. The illustration shows four ellipses that are filled with the same color (red) at different transparency levels.

5.1.7 SUPPORT FOR MULTIPLE IMAGE FORMATS

GDI+ provides the **GpImage**, **GpBitmap**, and **GpMetafile** classes, which allow you to load, save and manipulate images in a variety of formats. The following formats are supported:

- BMP
- Graphics Interchange Format (GIF)
- JPEG
- Exif
- PNG

- TIFF
- ICON
- WMF
- EMF

5.2 Changes in the Programming Model

The following sections describe several ways that programming with Windows GDI+ is different from programming with Windows Graphics Device Interface (GDI).

- [Device Contexts, Handles, and Graphics Objects](#)
- [Two Ways to Draw a Line](#)
 - [Drawing a line with GDI](#)
 - [Drawing a line with GDI+ and the C++ class interface](#)
- [Pens, Brushes, Paths, Images, and Fonts as Parameters](#)
- [Method Overloading](#)
- [No More Current Position](#)
- [Separate Methods for Draw and Fill](#)
- [Constructing Regions](#)

5.2.1 DEVICE CONTEXTS, HANDLES, AND GRAPHICS OBJECTS

If you have written programs using GDI (the graphics device interface included in previous versions of Windows), you are familiar with the idea of a device context (DC). A device context is a structure used by Windows to store information about the capabilities of a particular display device and attributes that specify how items will be drawn on that device. A device context for a video display is also associated with a particular window on the display. First you obtain a handle to a device context (HDC), and then you pass that handle as an argument to GDI functions that actually do the drawing. You also pass the handle as an argument to GDI functions that obtain or set the attributes of the device context.

When you use GDI+, you don't have to be as concerned with handles and device contexts as you do when you use GDI. You simply create a **GpGraphics** object and then invoke its methods using the flat API's wrapper functions — `GpStatus WINGDIPAPI GdipDrawLine(GpGraphics *graphics, GpPen *pen, REAL x1, REAL y1, REAL x2, REAL y2);`. The **GpGraphics** object is at the core of GDI+ just as the device context is at the core of GDI. The device context and the **GpGraphics** object play similar roles, but there are some fundamental differences between the handle-based programming model used with device contexts (GDI) and the object-oriented model used with **GpGraphics** objects (GDI+).

The **GpGraphics** object, like the device context, is associated with a particular window on the screen and contains attributes (for example, smoothing mode and text rendering hint) that specify how items are to be drawn. However, the **GpGraphics** object is not tied to a pen, brush, path, image, or font as a device context is. For example, in GDI, before you can use a device context to draw a line, you must call [SelectObject](#) to associate a pen object with the device context. This is referred to as selecting the pen into the device context. All lines drawn in the device context will use that pen until you select a different pen. With GDI+, you pass a **GpPen** object as an argument to the [GdipDrawLine](#) wrapper method of the **GpGraphics** class. You can use a different **GpPen** object in each of a series of DrawLine calls without having to associate a given **GpPen** object with a **GpGraphics** object.

5.2.2 TWO WAYS TO DRAW A LINE

The following two examples each draw a red line of width 3 from location (20, 10) to location (200,100). The first example calls GDI, and the second calls GDI+ through the C++ class interface.

- [Drawing a line with GDI](#)
- [Drawing a line with GDI+ and the C++ class interface](#)

5.2.3 DRAWING A LINE WITH GDI

To draw a line with GDI, you need two objects: a device context and a pen. If you don't have it you can obtain a handle to a device context by calling [BeginPaint](#), and a handle to a pen by calling [CreatePen](#). Next, you call [SelectObject](#) to select the pen into the device context. You set the pen position to (20, 10) by calling [MoveToEx](#) and then draw a line from that pen position to (200, 100) by calling [LineTo](#). Note that [MoveToEx](#) and [LineTo](#) both receive **hdc** as an argument.

```
HDC          hdc;
PAINTSTRUCT  ps;
HPEN         hPen;
HPEN         hPenOld;
hdc          = BeginPaint(hWnd, &ps);
hPen         = CreatePen(PS_SOLID, 3, RGB(255, 0, 0));
hPenOld      = (HPEN)SelectObject(hdc, hPen);
MoveToEx(hdc, 20, 10, NULL);
LineTo(hdc, 200, 100);
SelectObject(hdc, hPenOld);
DeleteObject(hPen);
EndPaint(hWnd, &ps);
```

5.2.4 DRAWING A LINE WITH GDI+ AND THE FLAT API'S

To draw a line with GDI+ and the C++ class interface, you need a [GpGraphics](#) object and a [GpPen](#) object. Note that you don't ask Windows for handles to these objects. Instead, you use constructors to create an instance of the [GpGraphics](#) class (a [GpGraphics](#) object) and an instance of the [GpPen](#) class (a [GpPen](#) object). Drawing a line involves calling the [GdipDrawLine](#) method of the [GpGraphics](#) class. The first parameter of the [GdipDrawLine](#) method is a pointer to your [GpPen](#) object. This is a simpler and more flexible scheme than selecting a pen into a device context as shown in the preceding GDI example.

```
HDC          hdc;
PAINTSTRUCT  ps;
GpPen*       myPen;
GpGraphics*  myGraphics;
hdc          = BeginPaint(hWnd, &ps);
GdipCreatePen1(MakeARGB(255, 255, 0, 0), 3.0f, UnitWorld, &myPen);
GdipCreateFromHDC(hdc, &myGraphics);
GdipDrawLineI(myGraphics, myPen, 20, 10, 200, 100);
GdipDeleteGraphics(myGraphics);
GdipDeletePen(myPen);
EndPaint(hWnd, &ps);
```

5.2.5 PENS, BRUSHES, PATHS, IMAGES, AND FONTS AS PARAMETERS

The preceding examples show that GpPen objects can be created and maintained separately from the GpGraphics object, which supplies the drawing methods. GpBrush, GraphicsPath, GpImage, and GpFont objects

can also be created and maintained separately from the `GpGraphics` object. Many of the drawing methods provided by the `GpGraphics` class receive a `GpBrush`, `GraphicsPath`, `GpImage`, or `GpFont` object as an argument. For example, the address of a `GpBrush` object is passed as an argument to the `GdipFillRectangle` method, and the address of a `GraphicsPath` object is passed as an argument to the `GdipDrawPath` method. Similarly, addresses of `GpImage` and `GpFont` objects are passed to the `GdipDrawImage` and `GdipDrawString` methods. This is in contrast to GDI where you select a brush, path, image, or font into the device context and then pass a handle to the device context as an argument to a drawing function.

5.2.6 METHOD OVERLOADING IN FLAT API'S

Many of the GDI+ methods are overloaded; that is, several methods share the same name but have different parameter lists. For example, the `DrawLine` method of the `GpGraphics` class comes in the following forms:

```
GpStatus WINGDIPAPI GdipDrawLine(GpGraphics *graphics,
                                   GpPen *pen,
                                   REAL x1, REAL y1,
                                   REAL x2, REAL y2);

GpStatus WINGDIPAPI GdipDrawLines(GpGraphics *graphics,
                                   GpPen *pen,
                                   GDIPCONST GpPointF *points,
                                   INT count);

GpStatus WINGDIPAPI GdipDrawLineI(GpGraphics *graphics,
                                   GpPen *pen,
                                   INT x1, INT y1,
                                   INT x2, INT y2);

GpStatus WINGDIPAPI GdipDrawLinesI(GpGraphics *graphics,
                                   GpPen *pen,
                                   GDIPCONST GpPoint *points,
                                   INT count);
```

All four of the `DrawLine` variations above receive a pointer to a `GpPen` object, the coordinates of the starting point, and the coordinates of the ending point. The first two variations receive the coordinates as floating point numbers, and the last two variations receive the coordinates as integers. The first and third variations receive the coordinates as a list of four separate numbers, while the second and fourth variations receive the coordinates as a pair of `GpPoint` (or `GpPointF`) objects.

5.2.7 NO MORE CURRENT POSITION

Note that in the `DrawLine` methods, shown previously, both the starting point and the ending point of the line are received as arguments. This is a departure from the GDI scheme where you call to set the current pen position followed by to draw a line starting at (**x1, y1**) and ending at (**x2, y2**). GDI+ as a whole has abandoned the notion of current position.

5.2.8 SEPARATE METHODS FOR DRAW AND FILL

GDI+ is more flexible than GDI when it comes to drawing the outlines and filling the interiors of shapes like rectangles. GDI has a `Rectangle` function that draws the outline and fills the interior of a rectangle all in one step. The outline is drawn with the currently selected pen, and the interior is filled with the currently selected brush.

```
hBrush = CreateHatchBrush(HS_CROSS, RGB(0, 0, 255));
hPen = CreatePen(PS_SOLID, 3, RGB(255, 0, 0));
SelectObject(hdc, hBrush);
SelectObject(hdc, hPen);
```

```
Rectangle(hdc, 100, 50, 200, 80);
```

GDI+ has separate methods for drawing the outline and filling the interior of a rectangle. The **GdipDrawRectangle** method of the **GpGraphics** class has the address of a **GpPen** object as one of its parameters, and the **GdipFillRectangle** method has the address of a **GpBrush** object as one of its parameters.

```
GpHatch* myHatchBrush;
GdipCreateHatchBrush(HatchStyleCross, MakeARGB(255, 0, 255, 0), MakeARGB(255, 0, 0, 255), &myHatchBrush);
GpPen* myPen;
GdipCreatePen1(MakeARGB(255, 255, 0, 0), 3.0f, UnitWorld, &myPen);
GdipDrawRectangleI(myGraphics, myPen, 100, 50, 100, 30);
GdipFillRectangleI(myGraphics, (GpBrush *)myHatchBrush, 100, 50, 100, 30);
```

Note that the **GdipFillRectangle** and **GdipDrawRectangle** methods in GDI+ receive arguments that specify the rectangle's left edge, top, width, and height. This is in contrast to the GDI **Rectangle** function, which takes arguments that specify the rectangle's left edge, right edge, top, and bottom. Also note that the constructor for the **Color** class in GDI+, **MakeARGB**, has four parameters. The last three parameters are the usual red, green, and blue values; the first parameter is the alpha value, which specifies the extent to which the color being drawn is blended with the background color.

5.2.9 CONSTRUCTING REGIONS

GDI provides several functions for creating regions: **CreateRectRgn**, **CreateEllpticRgn**, **CreateRoundRectRgn**, **CreatePolygonRgn**, and **CreatePolyPolygonRgn**. You might expect the **GpRegion** class in GDI+ to have analogous constructors that take rectangles, ellipses, rounded rectangles, and polygons as arguments, but that is not the case. The **GpRegion** class in GDI+ provides a constructor that receives a **GpRect** object reference and another constructor that receives the address of a **GraphicsPath** object. If you want to construct a region based on an ellipse, rounded rectangle, or polygon, you can easily do so by creating a **GraphicsPath** object (that contains an ellipse, for example) and then passing the address of that **GraphicsPath** object to a **GpRegion** constructor.

GDI+ makes it easy to form complex regions by combining shapes and paths. The **GpRegion** class has **Union**, **Intersect**, **Replace**, **Xor**, **Exclude** and **Complement** methods, defined in the **CombineMode** enum, that you can use to augment an existing region with a path or another region using wrapper functions:

```
GpStatus WINGDIPAPI GdipCombineRegionRect(GpRegion *region, GDIPCONST GpRectF *rect,
                                           CombineMode combineMode);
GpStatus WINGDIPAPI GdipCombineRegionRectI(GpRegion *region, GDIPCONST GpRect *rect,
                                           CombineMode combineMode);
GpStatus WINGDIPAPI GdipCombineRegionPath(GpRegion *region, GpPath *path,
                                           CombineMode combineMode);
```

One nice feature of the GDI+ scheme is that a **GraphicsPath** object is not destroyed when it is passed as an argument to a **GpRegion** constructor. In GDI, you can convert a path to a region with the **PathToRegion** function, but the path is destroyed in the process. Also, a **GraphicsPath** object is not destroyed when its address is passed as an argument to a **Union** or **Intersect** method, so you can use a given path as a building block for several separate regions. This is shown in the following example. Assume that **onePath** is a pointer to a **GraphicsPath** object (simple or complex) that has already been initialized.

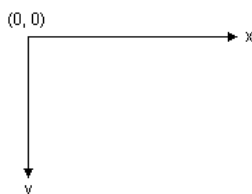
```
GpRegion region1(rect1);
GpRegion region2(rect2);
region1.Union(onePath);
region2.Intersect(onePath);
```

6 LINES, CURVES, AND SHAPES

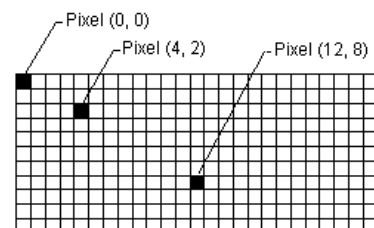
The vector graphics portion of GDI+ is used to draw lines, to draw curves, and to draw and fill shapes.

- [Overview of Vector Graphics](#)
- [Pens, Lines, and Rectangles](#)
- [Ellipses and Arcs](#)
- [Polygons](#)
- [Cardinal Splines](#)
- [Bézier Splines](#)
- [Paths](#)
- [Brushes and Filled Shapes](#)
- [Open and Closed Curves](#)
- [Regions](#)
- [Clipping](#)
- [Flattening Paths](#)
- [Antialiasing with Lines and Curves](#)

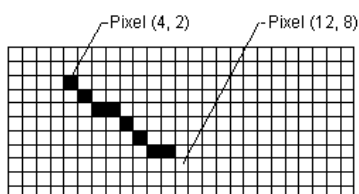
6.1 Overview of Vector Graphics



Windows GDI+ draws lines, rectangles, and other figures on a coordinate system. You can choose from a variety of coordinate systems, but the default coordinate system has the origin in the upper left corner with the x-axis pointing to the right and the y-axis pointing down. The unit of measure in the default coordinate system is the pixel.



A computer monitor creates its display on a rectangular array of dots called picture elements or pixels. The number of pixels appearing on the screen varies from one monitor to the next, and the number of pixels appearing on an individual monitor can usually be configured to some extent by the user.



point (4, 2) to the point (12, 8).

When you use GDI+ to draw a line, rectangle, or curve, you provide certain key information about the item to be drawn. For example, you can specify a line by providing two points, and you can specify a rectangle by providing a point, a height, and a width. GDI+ works in conjunction with the display driver software to determine which pixels must be turned on to show the line, rectangle, or curve. The following illustration shows the pixels that are turned on to display a line from the

Over time, certain basic building blocks have proven to be the most useful for creating two-dimensional pictures. These building blocks, which are all supported by GDI+, are given in the following list:

- Lines
- Rectangles
- Ellipses
- Arcs
- Polygons
- Cardinal splines
- Bézier splines

The **GpGraphics** class in GDI+ provides the following methods for drawing the items in the previous list: **GdipDrawLine**, **GdipDrawRectangle**, **GdipDrawEllipse**, **GdipDrawPolygon**, **GdipDrawArc**, **GdipDrawCurve** (for cardinal splines), and **GdipDrawBezier**. Each of these methods is overloaded; that is, each method comes in several variations with different parameter lists. For example, one variation of the DrawLine method receives the address of a **GpPen** object and four integers, while another variation of the DrawLine method receives the address of a **GpPen** object and two **GpPoint** object references.

The methods for drawing lines, rectangles, and Bézier splines have plural companion methods that draw several items in a single call: **GdipDrawLines**, **GdipDrawRectangles**, and **GdipDrawBeziers**. Also, the **GdipDrawCurve** method has a companion method, **GdipDrawClosedCurve**, which closes a curve by connecting the ending point of the curve to the starting point.

All the drawing methods of the **GpGraphics** class work in conjunction with a **GpPen** object. Thus, in order to draw anything, you must create at least two objects: a **GpGraphics** object and a **GpPen** object. The **GpPen** object stores attributes of the item to be drawn, such as line width and color. The address of the **GpPen** object is passed as one of the arguments to the drawing method. For example, one variation of the **GdipDrawRectangle** method receives the address of a **GpPen** object and four integers as shown in the following code, which draws a rectangle with a width of 100, a height of 50 and an upper-left corner of (20, 10).

```
GdipDrawRectangle(myGraphics, myPen, 20, 10, 100, 50);
```

6.2 Pens, Lines, and Rectangles

To draw lines with Windows GDI+ you need to create a **GpGraphics** object and a **GpPen** object. The **GpGraphics** object provides the methods that actually do the drawing, and the **GpPen** object stores attributes of the line, such as color, width, and style. Drawing a line is simply a matter of calling the **GdipDrawLine** method of the **GpGraphics** object. The address of the **GpPen** object is passed as one of the arguments to the DrawLine method. The following example draws a line from the point (4, 2) to the point (12, 6).

```
GdipDrawLine(myGraphics, myPen, 4.0, 2.0, 12.0, 6.0);
```

GdipDrawLines is an overloaded method of the **GpGraphics** class that accepts an array of points **GpPoint** objects and pass references to the **GpPoint** objects as arguments to the DrawLine method.

```
GpPoint myPoint[2] = {{4.0, 2.0}, {12.0, 6.0}};
GdipDrawLines(myGraphics, myPen, myPoint, 2);
```

You can specify certain attributes when you construct a **GpPen** object. For example, one **GpPen** constructor allows you to specify color and width. The following example draws a blue line of width 2 from (0, 0) to (60, 30).

```
GpPen *myPen;
GdipCreatePen1(MakeARGB(255, 0, 0, 255), 2.0, UnitPixel, myPen);
```

```
GdipDrawLineI(myGraphics, myPen, 0, 0, 60, 30);
```

The **GpPen** object also has attributes, such as dash style, that you can use to specify features of the line. For example, the following example draws a dashed line from (100, 50) to (300, 80).

```
GdipSetPenDashStyle(myPen, DashStyleDash);
GdipDrawLineI(myGraphics, myPen, 100, 50, 300, 80);
```



You can use various methods of the **GpPen** object to set many more attributes of the line. The **GdipSetPenStartCap** and **GdipSetPenEndCap** methods specify the appearance of the ends of the line; the ends can be flat, square, rounded, triangular, or a custom shape. The **GdipSetPenLineJoin** method lets you specify whether connected lines are mitered (joined with sharp corners), beveled, rounded, or clipped. The illustration shows lines with various cap and join styles.

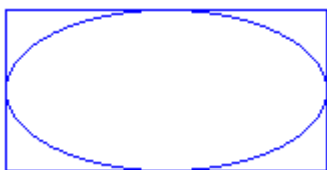
Drawing rectangles with GDI+ is similar to drawing lines. To draw a rectangle, you need a **GpGraphics** object and a **GpPen** object. The **GpGraphics** object provides a **GdipDrawRectangle** method, and the **GpPen** object stores attributes, such as line width and color. The address of the **GpPen** object is passed as one of the arguments to the **DrawRectangle** method. The following example draws a rectangle with its upper-left corner at (100, 50), a width of 80, and a height of 40.

```
GdipDrawRectangleI(myGraphics, myPen, 100, 50, 80, 40);
```

Under C++ **DrawRectangle** is an overloaded method of the **GpGraphics** class, so there are several ways you can supply it with arguments. For example, you can construct a **GpRect** object and pass a reference to the **GpRect** object as an argument. In flat API's this can be done using **GdipDrawRectangleI** method as shown below.

```
GpRect myRect = {100, 50, 80, 40};
GdipDrawRectangleI (myGraphics, myPen, &myRect);
```

6.3 Ellipses and Arcs



An ellipse is specified by its bounding rectangle. The following illustration shows an ellipse along with its bounding rectangle.

To draw an ellipse, you need a **GpGraphics** object and a **GpPen** object. The **GpGraphics** object provides the **GdipDrawEllipse** method, and the **GpPen** object stores attributes of the ellipse, such as line width and color. The address of the **GpPen** object is passed as one of the arguments to the **DrawEllipse** method. The remaining arguments passed to the **DrawEllipse** method specify the bounding rectangle for the ellipse. The following example draws an ellipse; the bounding rectangle has a width of 160, a height of 80, and an upper-left corner of (100, 50).

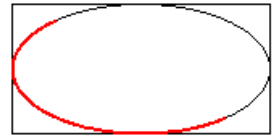
```
GdipDrawEllipseI(myGraphics, myPen, 100, 50, 160, 80);
```

DrawEllipse is an overloaded method of the **GpGraphics** class, so there are several ways you can supply it with arguments. For example, you can construct a **GpRect** object and pass a reference to the **GpRect** object as an argument to the **DrawEllipse** method.

```
GpRect myRect = {100, 50, 160, 80};
GdipDrawEllipseI(myGraphics, myPen, myRect);
```

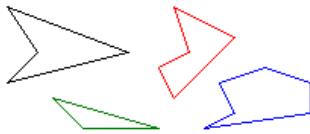
An arc is a portion of an ellipse. To draw an arc, you call the `GdipDrawArc` method of the `GpGraphics` class. The parameters of the `DrawArc` method are the same as the parameters of the `GdipDrawEllipse` method, except that `DrawArc` requires a starting angle and sweep angle. The following example draws an arc with a starting angle of 30 degrees and a sweep angle of 180 degrees.

```
GdipDrawArcI(myGraphics, myPen, 100, 50, 160, 80, 30, 180);
```



The illustration shows the arc, the ellipse, and the bounding rectangle.

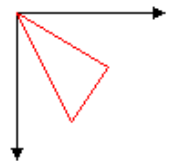
6.4 Polygons



A polygon is a closed figure with three or more straight sides. For example, a triangle is a polygon with three sides, a rectangle is a polygon with four sides, and a pentagon is a polygon with five sides.

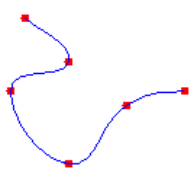
To draw a polygon, you need a `GpGraphics` object, a `GpPen` object, and an array of `GpPoint` (or `GpPointF`) objects. The `GpGraphics` object provides the `GdipDrawPolygon` method. The `GpPen` object stores attributes of the polygon, such as line width and color, and the array of `GpPoint` objects stores the points to be connected by straight lines. The addresses of the `GpPen` object and the array of `GpPoint` objects are passed as arguments to the `DrawPolygon` method. The following example draws a three-sided polygon. Note that there are only three points in `myPointArray`: (0, 0), (50, 30), and (30, 60). The `DrawPolygon` method automatically closes the polygon by drawing a line from (30, 60) back to the starting point (0, 0);

```
GpPoint myPointArray[] = {{0, 0}, {50, 30}, {30, 60}};  
GdipDrawPolygonI(myGraphics, myPen, myPointArray, 3);
```



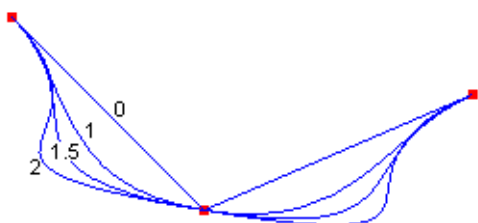
The illustration shows the polygon.

6.5 Cardinal Splines



A cardinal spline is a sequence of individual curves joined to form a larger curve. The spline is specified by an array of points and a tension parameter. A cardinal spline passes smoothly through each point in the array; there are no sharp corners and no abrupt changes in the tightness of the curve. The following illustration shows a set of points and a cardinal spline that passes through each point in the set.

A physical spline is a thin piece of wood or other flexible material. Before the advent of mathematical splines, designers used physical splines to draw curves. A designer would place the spline on a piece of paper and anchor it to a given set of points. The designer could then create a curve by drawing along the spline with a pencil. A given set of points could yield a variety of curves, depending on the properties of the physical spline. For example, a spline with a high resistance to bending would produce a different curve than an extremely flexible spline.



The formulas for mathematical splines are based on the properties of flexible rods, so the curves produced by mathematical splines are similar to the curves that were once produced by physical splines. Just as physical splines of different

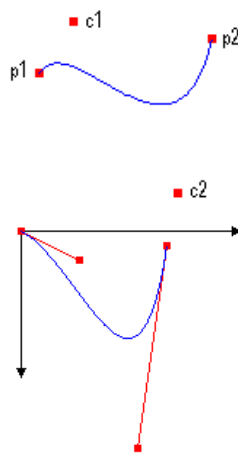
tension will produce different curves through a given set of points, mathematical splines with different values for the tension parameter will produce different curves through a given set of points. The following illustration shows four cardinal splines passing through the same set of points. The tension is shown for each spline. Note that a tension of 0 corresponds to infinite physical tension, forcing the curve to take the shortest way (straight lines) between points. A tension of 1 corresponds to no physical tension, allowing the spline to take the path of least total bend. With tension values greater than 1, the curve behaves like a compressed spring, pushed to take a longer path.

Note that the four splines in the preceding figure share the same tangent line at the starting point. The tangent is the line drawn from the starting point to the next point along the curve. Likewise, the shared tangent at the ending point is the line drawn from the ending point to the previous point on the curve.

To draw a cardinal spline, you need a **GpGraphics** object, a **GpPen** object, and an array of **GpPoint** objects. The **GpGraphics** object provides the **GdipDrawCurve2** method, which draws the spline, and the **GpPen** object stores attributes of the spline, such as line width and color. The array of **GpPoint** objects stores the points that the curve will pass through. The following example draws a cardinal spline that passes through the points in **myPointArray**. The third parameter is the tension.

```
GpPoint myPointArray[] = {{0.0f, 0.0f}, {100.0f, 100.0f}, {300.0f, 20.0f}};
GdipDrawCurve2(myGraphics, myPen, myPointArray, 3, 1.5f);
```

6.6 Bézier Splines



A Bézier spline is a curve specified by four points: two end points (p1 and p2) and two control points (c1 and c2). The curve begins at p1 and ends at p2. The curve doesn't pass through the control points, but the control points act as magnets, pulling the curve in certain directions and influencing the way the curve bends. The following illustration shows a Bézier curve along with its endpoints and control points.

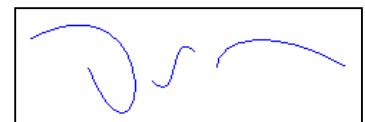
Note that the curve starts at p1 and moves toward the control point c1. The tangent line to the curve at p1 is the line drawn from p1 to c1. Also note that the tangent line at the endpoint p2 is the line drawn from c2 to p2.

To draw a Bézier spline, you need a **GpGraphics** object and a **GpPen** object. The **GpGraphics** object provides the **GdipDrawBezier** method, and the **GpPen** object stores attributes of the curve, such as line width and color. The address of the **GpPen** object is passed as one of the arguments to the **DrawBezier** method. The remaining arguments passed to the **DrawBezier** method are the endpoints and the control points. The following example draws a Bézier spline with starting point (0, 0), control points (40, 20) and (80, 150), and ending point (100, 10).

```
GdipDrawBezierI(myGraphics, myPen, 0, 0, 40, 20, 80, 150, 100, 10);
```

The illustration shows the curve, the control points, and two tangent lines.

Bézier splines were originally developed by Pierre Bézier for design in the automotive industry. They have since proven to be very useful in many types of computer-aided design and are also used to define the outlines of fonts. Bézier splines can yield a wide variety of shapes, some of which are shown in

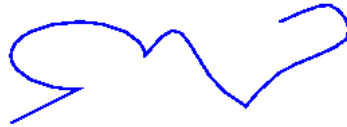


the right illustration.

6.7 Paths

Paths are formed by combining lines, rectangles, and simple curves. Recall from the [Overview of Vector Graphics](#) that the following basic building blocks have proven to be the most useful for drawing pictures.

- Lines
- Rectangles
- Ellipses
- Arcs
- Polygons
- Cardinal splines
- Bézier splines



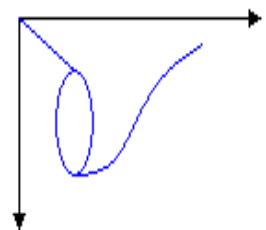
In Windows GDI+, the **GraphicsPath** object allows you to collect a sequence of these building blocks into a single unit. The entire sequence of lines, rectangles, polygons, and curves can then be drawn with one call to the **GdipDrawPath** method of the **GpGraphics** class. The following illustration shows a path created by combining a line, an arc, a Bézier spline, and a cardinal spline.

The **GraphicsPath** class provides the following methods for creating a sequence of items to be drawn: **GdipAddPathLine**, **GdipAddPathRectangle**, **GdipAddPathEllipse**, **GdipAddPathArc**, **GdipAddPathPolygon**, **GdipAddPathCurve** (for cardinal splines), and **GdipAddPathBezier**. Each of these methods is overloaded; that is, each method comes in several variations with different parameter lists.

The methods for adding lines, rectangles, and Bézier splines to a path have plural companion methods that add several items to the path in a single call: **GdipAddPathLine2**, **GdipAddPathRectangles**, and **GdipAddPathBeziers**. Also, the **GdipAddPathCurve** method has a companion method, **GdipAddPathClosedCurve**, which adds a closed curve to the path.

To draw a path, you need a **GpGraphics** object, a **GpPen** object, and a **GraphicsPath** object. The **GpGraphics** object provides the **GdipDrawPath** method, and the **GpPen** object stores attributes of the path, such as line width and color. The **GraphicsPath** object stores the sequence of lines, rectangles, and curves that make up the path. The addresses of the **GpPen** object and the **GraphicsPath** object are passed as arguments to the **GdipDrawPath** method. The following example draws a path that consists of a line, an ellipse, and a Bézier spline.

```
GraphicsPath myGraphicsPath;  
GdipCreatePath(FillModeAlternate, &myGraphicsPath);  
GdipAddPathLineI(myGraphicsPath, 0, 0, 30, 20);  
GdipAddPathEllipseI(myGraphicsPath, 20, 20, 20, 40);  
GdipAddPathBezierI(myGraphicsPath, 30, 60, 70, 60, 50, 30, 100, 10);  
GdipDrawPath(myGraphics, myPen, myGraphicsPath);  
GdipDeletePath(myGraphicsPath);
```



In addition to adding lines, rectangles, and curves to a path, you can add paths to a path. This allows you to combine existing paths to form large, complex paths. The following code adds **graphicsPath1** and **graphicsPath2** to **myGraphicsPath**. The second parameter of the **GraphicsPath::AddPath** method specifies whether the added path is connected to the existing path.

```
GdipAddPathPath (myGraphicsPath, graphicsPath1, FALSE);
```

```
GdipAddPathPath (myGraphicsPath, graphicsPath2, TRUE);
```

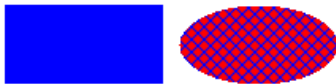
There are two other items you can add to a path: strings and pies. A pie is a portion of the interior of an ellipse. The following example creates a path from an arc, a cardinal spline, a string, and a pie.

```
GdipAddPathArcI(myGraphicsPath, 0, 0, 30, 20, -90, 180);
GdipStartPathFigure(myGraphicsPath);
GpPointF myPointArray[] = {{5.0f, 30.0f}, {20.0f, 40.0f}, {50.0f, 30.0f}};
GdipAddPathCurve(myGraphicsPath, myPointArray, 3);
GpFontFamily *myFontFamily;
GdipCreateFontFamilyFromName(L"Times New Roman", NULL, &myFontFamily);
GpFont *myFont;
GdipCreateFont(myFontFamily, 20.0, 0, UnitPixel, &myFont);
GpStringFormat *myStringFormat;
GdipStringFormatGetGenericDefault(&myStringFormat);
GdipSetStringFormatAlign(myStringFormat, StringAlignmentCenter);
GpRect myRect = {40, 20, 200, 20};
GdipAddPathStringI(myGraphicsPath, L"a string in a path", 18, myFontFamily, 0, 24, &myRect, myStringFormat);
GdipAddPathPieI(myGraphicsPath, 230, 10, 40, 40, 40, 110);
GdipDrawPath(myGraphics, myPen, myGraphicsPath);
GdipDeleteFontFamily(myFontFamily);
GdipDeleteFont(myFont);
GdipDeleteStringFormat(myStringFormat);
```



The right illustration shows the path. Note that a path does not have to be connected; the arc, cardinal spline, string, and pie are separated. The separation between drawing of the arc and the curve is obtained using the function [GdipStartPathFigure](#).

6.8 Brushes and Filled Shapes



A closed figure such as a rectangle or an ellipse consists of an outline and an interior. The outline is drawn with a [GpPen](#) object and the interior is filled with a [GpBrush](#) object. Windows GDI+ provides several brush classes for filling the interiors of closed figures: [GpSolidFill](#), [GpHatch](#), [GpTexture](#), [GpLineGradient](#), and [GpPathGradient](#). All these classes inherit from the [GpBrush](#) class. The following illustration shows a rectangle filled with a solid brush and an ellipse filled with a hatch brush.

- [Solid Brushes](#)
- [Hatch Brushes](#)
- [Texture Brushes](#)
- [Gradient Brushes](#)

6.8.1 SOLID BRUSHES

To fill a closed shape, you need a [GpGraphics](#) object and a [GpBrush](#) object. The [GpGraphics](#) object provides methods, such as [FillRectangle](#) and [FillEllipse](#), and the [GpBrush](#) object stores attributes of the fill, such as color and pattern. The address of the [GpBrush](#) object is passed as one of the arguments to the fill method. The following example fills an ellipse with a solid red color.

```
GpSolidFill *mySolidBrush;
GdipCreateSolidFill(MakeARGB(255, 255, 0, 0), &mySolidBrush);
GdipFillEllipse(myGraphics, mySolidBrush, 0, 0, 60, 40);
```

Note that in the preceding example, the brush is of type **GpSolidFill**, which inherits from **GpBrush**.

6.8.2 HATCH BRUSHES



When you fill a shape with a hatch brush, you specify a foreground color, a background color, and a hatch style. The foreground color is the color of the hatching.

```
GpHatch *myHatchBrush;
GdipCreateHatchBrush(HatchStyleHorizontal, MakeARGB(255, 0, 0, 255), MakeARGB(255, 128, 255, 255),
&myHatchBrush);
GdipFillEllipse(myGraphics, myHatchBrush, 105, 0, 100, 60);
GdipDeleteBrush(myHatchBrush);

GdipCreateHatchBrush(HatchStyleForwardDiagonal, MakeARGB(255, 0, 0, 255), MakeARGB(255, 128, 255, 255),
&myHatchBrush);
GdipFillEllipse(myGraphics, myHatchBrush, 210, 0, 100, 60);
GdipDeleteBrush(myHatchBrush);

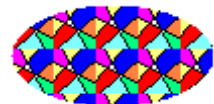
GdipCreateHatchBrush(HatchStyleCross, MakeARGB(255, 0, 0, 255), MakeARGB(255, 128, 255, 255), &myHatchBrush);
GdipFillEllipse(myGraphics, myHatchBrush, 315, 0, 100, 60);
GdipDeleteBrush(myHatchBrush);
```

GDI+ provides more than 50 hatch styles, specified in **HatchStyle**. The three styles shown in the illustration are Horizontal, ForwardDiagonal, and Cross.

6.8.3 TEXTURE BRUSHES



With a texture brush, you can fill a shape with a pattern stored in a bitmap. For example, suppose the following picture is stored in a disk file named MyTexture.bmp.



The following example fills an ellipse by repeating the picture stored in MyTexture.bmp, the left illustration shows the filled ellipse.

```
GpImage *myImage;
GdipLoadImageFromFile(L"MyTexture.bmp", &myImage);
GpTexture *myTextureBrush;
GdipCreateTexture(myImage, WrapModeTile, &myTextureBrush);
GdipFillEllipse(myGraphics, myTextureBrush, 0, 0, 100, 50);
```

6.8.4 GRADIENT BRUSHES



You can use a gradient brush to fill a shape with a color that changes gradually from one part of the shape to another. For example, a horizontal gradient brush will change color as you move from the left side of a figure to the right side. The following example fills an ellipse with a horizontal gradient brush that changes from blue to green as you move from the left side of the ellipse to the right side.

```
GpRect myRect = {0, 60, 200, 100};
GpLineGradient *myLinearGradientBrush;
GdipCreateLineBrushFromRectI(&myRect, MakeARGB(255, 0, 0, 255), MakeARGB(255, 0, 255, 0),
LinearGradientModeHorizontal, WrapModeTile, &myLinearGradientBrush);
GdipFillEllipseI(myGraphics, myLinearGradientBrush, myRect.X, myRect.Y, myRect.Width, myRect.Height);
```

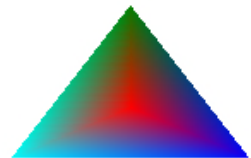
```
GdipDeleteBrush(myLinearGradientBrush);
```



A path gradient brush can be configured to change color as you move from the center of a figure toward the boundary.

```
GpPath *myPath;
GdipCreatePath(FillModeAlternate, &myPath);
GdipAddPathEllipseI(myPath, myRect.X, myRect.Y, myRect.Width, myRect.Height);
GpPathGradient *myPathBrush;
GdipCreatePathGradientFromPath(myPath, &myPathBrush);
GdipSetPathGradientCenterColor(myPathBrush, MakeARGB(255, 0, 0, 255));
int count = 1;
ARGB BorderColor = MakeARGB(255, 128, 255, 255);
GdipSetPathGradientSurroundColorsWithCount(myPathBrush, &BorderColor, &count);
GdipFillEllipseI(myGraphics, myPathBrush, myRect.X, myRect.Y, myRect.Width, myRect.Height);
GdipDeleteBrush(myPathBrush);
GdipDeletePath(myPath);
```

Path gradient brushes are quite flexible. The gradient brush used to fill the triangle in the illustration changes gradually from red at the center to each of three different colors at the vertices.



```
GpPoint myPoints[] = {{360, 100}, {510, 400}, {210, 400}};
GdipCreatePath(FillModeAlternate, &myPath);
GdipAddPathPolygonI(myPath, myPoints, 3);
ARGB BorderColors[] = {MakeARGB(255, 0, 255, 0), MakeARGB(255, 0, 64, 128), MakeARGB(255, 0, 255, 255)};
GdipCreatePathGradientI(myPoints, 3, WrapModeTile, &myPathBrush);
GdipSetPathGradientCenterColor(myPathBrush, MakeARGB(255, 255, 128, 64));
count = 3;
GdipSetPathGradientSurroundColorsWithCount(myPathBrush, BorderColors, &count);
GdipFillPolygonI(myGraphics, myPathBrush, myPoints, 3, FillModeAlternate);
GdipDeleteBrush(myPathBrush);
GdipDeletePath(myPath);
```

6.9 Open and Closed Curves



The illustration shows two curves: one open and one closed.

Closed curves have an interior and therefore can be filled with a brush. The **GpGraphics** class in Windows GDI+ provides the following methods for filling closed figures and curves: **GdipFillRectangle**, **GdipFillEllipse**, **GdipFillPie**, **GdipFillPolygon**, **GdipFillClosedCurve**, **GdipFillPath**, and **GdipFillRegion**. Whenever you call one of these methods, you must pass the address of one of the specific brush types (**GpSolidFill**, **GpHatch**, **GpTexture**, **GpLineGradient**, or **GpPathGradient**) as an argument.

The **GdipFillPie** method is a companion to the **GdipDrawArc** method. Just as the **DrawArc** method draws a portion of the outline of an ellipse, the **FillPie** method fills a portion of the interior of an ellipse. The following example draws an arc and fills the corresponding portion of the interior of the ellipse.

```
GpSolidFill *mySolidBrush;
GdipCreateSolidFill(MakeARGB(255, 128, 255, 255), &mySolidBrush);
GdipFillPie(myGraphics, mySolidBrush, 0, 0, 140, 70, 0, 120);
GdipDeleteBrush(mySolidBrush);
GpPen* myPen;
GdipCreatePen1(MakeARGB(255, 0, 0, 255), 1.0f, UnitPixel, &myPen);
GdipDrawArc(myGraphics, myPen, 0, 0, 140, 70, 0, 120);
```

```
GdiDeletePen(myPen);
```

The illustration shows the arc and the filled pie.



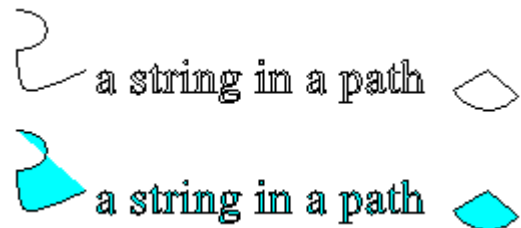
The [FillClosedCurve](#) method is a companion to the [DrawClosedCurve](#) method. Both methods automatically close the curve by connecting the ending point to the starting point. The following example draws a curve that passes through (0, 0), (60, 20), and (40, 50). Then, the curve is automatically closed by connecting (40, 50) to the starting point (0, 0), and the interior is filled with a solid color.

```
Point myPointArray[] = {{10, 10}, {60, 20}, {40, 50}};  
GdiDrawClosedCurveI(myGraphics, myPen, myPointArray, 3);  
GdiFillClosedCurveI(myGraphics, mySolidBrush, myPointArray, 3);
```

A path can consist of several figures (subpaths). The [GdiFillPath](#) method fills the interior of each figure. If a figure is not closed, the [GdiFillPath](#) method fills the area that would be enclosed if the figure were closed. The following example draws and fills a path that consists of an arc, a cardinal spline, a string, and a pie.

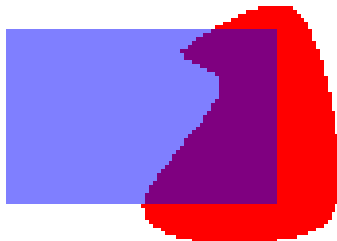
```
GpPath *myGraphicsPath;  
GdiCreatePath(FillModeAlternate, &myGraphicsPath);  
GdiFillPath(myGraphics, mySolidBrush, myGraphicsPath);  
GdiDrawPath(myGraphics, myPen, myGraphicsPath);
```

The illustration shows the path before and after it is filled with a solid brush. Note that the text in the string is outlined, but not filled, by the [GdiDrawPath](#) method. It is the [GdiFillPath](#) method that paints the interiors of the characters in the string.



6.10 Regions

A region is a portion of the display surface. Regions can be simple (a single rectangle) or complex (a combination of polygons and closed curves). The following illustration shows two regions: one constructed from a rectangle, and the other constructed from a path.



Regions are often used for clipping and hit testing. Clipping involves restricting drawing to a certain region of the screen, usually the portion of the screen that needs to be updated. Hit testing involves checking to see whether the cursor is in a certain region of the screen when a mouse button is pressed.

You can construct a region from a rectangle or from a path. You can also create complex regions by combining existing regions. The [GpRegion](#) class provides the following methods for combining regions: [Intersect](#), [Union](#), [Xor](#), [Exclude](#), and [Complement](#).



Intersection



Union

The intersection of two regions is the set of all points belonging to both regions. The union is the set of all points belonging to one or the other or both regions. The complement of a region is the set of all points that are not in the region. The following illustration shows the intersection and union of the two regions in the previous figure.



Xor



The curved region excluded from the rectangular region

The [Xor](#) method, applied to a pair of regions, produces a region that contains all points that belong to one region or the other, but not both. The [Exclude](#) method, applied to a pair of regions, produces a region that contains all points in the first region that are not in the second region. The following illustration shows the regions that result from applying the Xor and Exclude methods to

the two regions shown at the beginning of this topic.

To fill a region, you need a [GpGraphics](#) object, a [GpBrush](#) object, and a [GpRegion](#) object. The [GpGraphics](#) object provides the [GdipFillRegion](#) method, and the [GpBrush](#) object stores attributes of the fill, such as color or pattern. The following example fills a region with a solid color.

```
GdipFillRegion(myGraphics, mySolidBrush, myRegion);
```

6.11 Clipping



Clipping involves restricting drawing to a certain region. The following illustration shows the string "Hello" clipped to a heart-shaped region.

Regions can be constructed from paths, and paths can contain the outlines of strings, so you can use outlined text for clipping.

The following illustration shows a set of concentric ellipses clipped to the interior of a string of text.



To draw with clipping, create a [GpGraphics](#) object, call its [GdipSetClip](#) method, and then call the drawing methods of that same [GpGraphics](#) object. The following example draws a line that is clipped to a rectangular region.

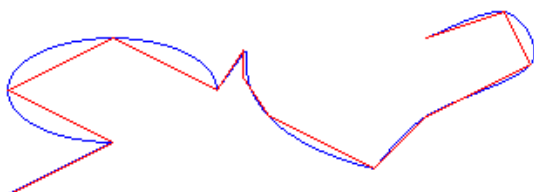
```
GpRegion *myRegion;
GpRect myRegionRect = {170, 30, 100, 50};
GdipCreateRegionRectI(&myRegionRect, &myRegion);
GdipDrawRectangleI(myGraphics, myPen, 170, 30, 100, 50);
GdipSetClipRegion(myGraphics, myRegion, CombineModeReplace);
GdipDrawLineI(myGraphics, myPen, 150, 0, 550, 400);
```

The following illustration shows the rectangular region along with the clipped line.



6.12 Flattening Paths

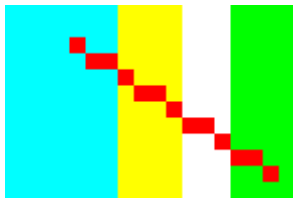
A [GraphicsPath](#) object stores a sequence of lines and Bézier splines. You can add several types of curves (ellipses, arcs, cardinal splines) to a path, but each curve is converted to a Bézier spline before it is stored in the path. Flattening a path consists of converting each Bézier spline in the path to a sequence of straight lines.



To flatten a path, call the [GraphicsPath::Flatten](#) method of a [GraphicsPath](#) object. The [GraphicsPath::Flatten](#) method receives a flatness argument that specifies the maximum distance between the flattened path and the original path. The following illustration shows a path before and after flattening.

6.13 Antialiasing with Lines and Curves

When you use Windows GDI+ to draw a line, you provide the starting point and ending point of the line, but you don't have to provide any information about the individual pixels on the line. GDI+ works in conjunction



with the display driver software to determine which pixels will be turned on to show the line on a particular display device.

Consider a straight red line that goes from the point (4, 2) to the point (16, 10). Assume the coordinate system has its origin in the upper-left corner and that the unit of measure is the pixel. Also assume that the x-axis points to the right and

the y-axis points down. The following illustration shows an enlarged view of the red line drawn on a multicolored background.

Note that the red pixels used to render the line are opaque. There are no partially transparent pixels involved in displaying the line. This type of line rendering gives the line a jagged appearance, and the line looks a bit like a staircase. This technique of representing a line with a staircase is called aliasing; the staircase is an alias for the theoretical line.



A more sophisticated technique for rendering a line involves using partially transparent pixels along with pure red pixels. Pixels are set to pure red or to some blend of red and the background color depending on how close they are to the line. This type of rendering is called antialiasing and results in a line that the human eye perceives as more smooth. The following illustration shows how certain pixels are blended with the background to produce an antialiased line.



Antialiasing (smoothing) can also be applied to curves. The illustration shows an enlarged view of a smoothed ellipse.



Without antialiasing

With antialiasing

The right illustration shows the same ellipse in its actual size, once without antialiasing and once with antialiasing.

To draw lines and curves that use antialiasing, create a **GpGraphics** object and pass **SmoothingModeAntiAlias** to its **GdipSetSmoothingMode** method. Then call one of the drawing methods of that same **GpGraphics** object.

```
GdipSetSmoothingMode(myGraphics, SmoothingModeAntiAlias);  
GdipDrawLineI(myGraphics, myPen, 0, 0, 12, 8);
```

SmoothingModeAntiAlias is an element of the **SmoothingMode** enumeration.

7 IMAGES, BITMAPS, AND METAFILES

Windows GDI+ provides the **GpImage** class for working with raster images (bitmaps) and vector images (metafiles). The **GpBitmap** class and the **GpMetafile** class both inherit from the **GpImage** class. The **GpBitmap** class expands on the capabilities of the **GpImage** class by providing additional methods for loading, saving, and manipulating raster images. The **GpMetafile** class expands on the capabilities of the **GpImage** class by providing additional methods for recording and examining vector images.

- [Types of Bitmaps](#)
- [Metafiles](#)
- [Drawing, Positioning, and Cloning Images](#)
- [Cropping and Scaling Images](#)

7.1 Types of Bitmaps

A bitmap is an array of bits that specifies the color of each pixel in a rectangular array of pixels. The number of bits devoted to an individual pixel determines the number of colors that can be assigned to that pixel. For example, if each pixel is represented by 4 bits, then a given pixel can be assigned one of 16 different colors ($2^4 = 16$). The following table shows a few examples of the number of colors that can be assigned to a pixel represented by a given number of bits.

Bits per pixel	Number of colors that can be assigned to a pixel
1	$2^1 = 2$
2	$2^2 = 4$
4	$2^4 = 16$
8	$2^8 = 256$
16	$2^{16} = 65,536$
24	$2^{24} = 16,777,216$

3 3 3 3 3 3 3 3
0 1 4 1 4 1 4 0
0 4 1 4 1 4 1 0
0 5 5 5 5 5 5 0
0 5 5 5 5 5 5 0
0 1 4 1 4 1 4 0
0 4 1 4 1 4 1 0
2 2 2 2 2 2 2 2



0	000000	Black
1	FF0000	Red
2	00FF00	Green
3	0000FF	Blue
4	FFFFFF	White
5	FFFF00	Yellow
6	FF00FF	Magenta
7	00FFFF	Cyan
8	FF0080	Dark Red
9	FF8040	Dark Orange
A	804000	Dark Green
B	008080	Dark Blue
C	800000	Dark Red
D	800080	Dark Purple
E	8080FF	Dark Blue


Disk files that store bitmaps usually contain one or more information blocks that store information such as number of bits per pixel, number of pixels in each row, and number of rows in the array. Such a file might also contain a color table (sometimes called a color palette). A color table maps numbers in the bitmap to specific colors. The following illustration shows an enlarged image along with its bitmap and color table. Each pixel is represented by a 4-bit number, so there are $2^4 = 16$ colors in the color table. Each color in the table is represented by a 24-bit number: 8 bits for red, 8 bits for green, and 8 bits for blue. The numbers are shown in hexadecimal

(base 16) form: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15.

Look at the pixel in row 3, column 5 of the image. The corresponding number in the bitmap is 1. The color table tells us that 1 represents the color red, so the pixel is red. All the entries in the top row of the bitmap are 3. The color table tells us that 3 represents blue, so all the pixels in the top row of the image are blue.

Note Some bitmaps are stored in bottom-up format; the numbers in the first row of the bitmap correspond to the pixels in the bottom row of the image.

A bitmap that stores indexes into a color table is called a **palette-indexed** bitmap. Some bitmaps have no need for a color table. For example, if a bitmap uses 24 bits per pixel, that bitmap can store the colors themselves rather than indexes into a color table. The following illustration shows a bitmap that stores colors directly (24 bits per pixel) rather than using a color table. The illustration also shows an enlarged view of the corresponding image. In the bitmap, FFFFFFFF represents white, FF0000 represents red, 00FF00 represents green, and 0000FF represents blue.

0000FF	0000FF	0000FF	0000FF	0000FF	0000FF	0000FF	0000FF	
00FF00	FF0000	FFFFFF	FF0000	FFFFFF	FF0000	FFFFFF	00FF00	
00FF00	FFFFFF	FF0000	FFFFFF	FF0000	FFFFFF	FF0000	00FF00	
00FF00	FF0000	FFFFFF	FF0000	FFFFFF	FF0000	FFFFFF	00FF00	
00FF00	FFFFFF	FF0000	FFFFFF	FF0000	FFFFFF	FF0000	00FF00	
00FF00	FFFFFF	FF0000	FFFFFF	FF0000	FFFFFF	FF0000	00FF00	
00FF00	FF0000	FFFFFF	FF0000	FFFFFF	FF0000	FFFFFF	00FF00	
00FF00	FFFFFF	FF0000	FFFFFF	FF0000	FFFFFF	FF0000	00FF00	
0000FF	0000FF	0000FF	0000FF	0000FF	0000FF	0000FF	0000FF	

7.2 Graphics File Formats

There are many standard formats for saving bitmaps in files. Windows GDI+ supports the graphics file formats described in the following paragraphs.

Bitmap (BMP)

BMP is a standard format used by Windows to store device-independent and application-independent images. The number of bits per pixel (1, 4, 8, 15, 24, 32, or 64) for a given BMP file is specified in a file header. BMP files with 24 bits per pixel are common.

Graphics Interchange Format (GIF)

GIF is a common format for images that appear on Web pages. GIFs work well for line drawings, pictures with blocks of solid color, and pictures with sharp boundaries between colors. GIFs are compressed, but no information is lost in the compression process; a decompressed image is exactly the same as the original. One color in a GIF can be designated as transparent, so that the image will have the background color of any Web page that displays it. A sequence of GIF images can be stored in a single file to form an animated GIF. GIFs store at most 8 bits per pixel, so they are limited to 256 colors.

Joint Photographic Experts Group (JPEG)

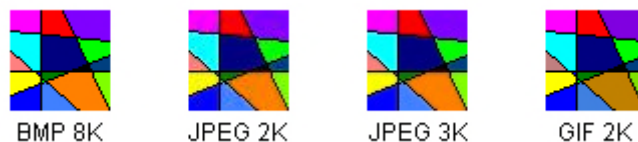
JPEG is a compression scheme that works well for natural scenes, such as scanned photographs. Some information is lost in the compression process, but often the loss is imperceptible to the human eye. Color JPEG images store 24 bits per pixel, so they are capable of displaying more than 16 million colors. There is also a grayscale JPEG format that stores 8 bits per pixel. JPEGs do not support transparency or animation.

The level of compression in JPEG images is configurable, but higher compression levels (smaller files) result in more loss of information. A 20:1 compression ratio often produces an image that the human eye finds difficult to distinguish from the original. The following illustration shows a BMP image and two JPEG images that were

compressed from that BMP image. The first JPEG has a compression ratio of 4:1 and the second JPEG has a compression ratio of about 8:1.



JPEG compression does not work well for line drawings, blocks of solid color, and sharp boundaries. The following illustration shows a BMP along with two JPEGs and a GIF. The JPEGs and the GIF were compressed from the BMP. The compression ratio is 4:1 for the GIF, 4:1 for the smaller JPEG, and 8:3 for the larger JPEG. Note that the GIF maintains the sharp boundaries along the lines, but the JPEGs tends to blur the boundaries.



JPEG is a compression scheme, not a file format. JPEG File Interchange Format (JFIF) is a file format commonly used for storing and transferring images that have been compressed according to the JPEG scheme. JFIF files displayed by Web browsers use the .jpg extension.

Exchangeable Image File (Exif)

Exif is a file format used for photographs captured by digital cameras. An Exif file contains an image that is compressed according to the JPEG specification. An Exif file also contains information about the photograph (date taken, shutter speed, exposure time, and so on) and information about the camera (manufacturer, model, and so on).

Portable Network Graphics (PNG)

The PNG format retains many of the advantages of the GIF format but also provides capabilities beyond those of GIF. Like GIF files, PNG files are compressed with no loss of information. PNG files can store colors with 8, 24, or 48 bits per pixel and gray scales with 1, 2, 4, 8, or 16 bits per pixel. In contrast, GIF files can use only 1, 2, 4, or 8 bits per pixel. A PNG file can also store an alpha value for each pixel, which specifies the degree to which the color of that pixel is blended with the background color.

PNG improves on GIF in its ability to progressively display an image; that is, to display better and better approximations of the image as it arrives over a network connection. PNG files can contain gamma correction and color correction information so that the images can be accurately rendered on a variety of display devices.

Tag Image File Format (TIFF)

TIFF is a flexible and extendable format that is supported by a wide variety of platforms and image-processing applications. TIFF files can store images with an arbitrary number of bits per pixel and can employ a variety of compression algorithms. Several images can be stored in a single, multiple-page TIFF file. Information related to the image (scanner make, host computer, type of compression, orientation, samples per pixel, and so on)

can be stored in the file and arranged through the use of tags. The TIFF format can be extended as needed by the approval and addition of new tags.

7.3 Metafiles

Windows GDI+ provides the **GpMetafile** class so that you can record and display metafiles. A metafile, also called a vector image, is an image that is stored as a sequence of drawing commands and settings. The commands and settings recorded in a **GpMetafile** object can be stored in memory or saved to a file or stream.

GDI+ can display metafiles that have been stored in the following formats:

- Windows Metafile Format (WMF)
- Enhanced Metafile (EMF)
- EMF+

GDI+ can record metafiles in the EMF and EMF+ formats, but not in the WMF format.

EMF+ is an extension to EMF that allows GDI+ records to be stored. There are two variations on the EMF+ format: EMF+ Only and EMF+ Dual. EMF+ Only metafiles contain only GDI+ records. Such metafiles can be displayed by GDI+ but not by Windows Graphics Device Interface (GDI). EMF+ Dual metafiles contain GDI+ and GDI records. Each GDI+ record in an EMF+ Dual metafile is paired with an alternate GDI record. Such metafiles can be displayed by GDI+ or by GDI.

The following example records one setting and one drawing command in a disk file. Note that the example creates a **GpGraphics** object and that the constructor for the **GpGraphics** object receives the address of a **GpMetafile** object as an argument.

```
GpMetafile *myMetafile;
RECT rc;
GetClientRect(g_hwnd, &rc);
GpRect myRect = { rc.left, rc.top, rc.right - rc.left, rc.bottom - rc.top };
GdipRecordMetafileFileNameI(L"MyDiskFile.emf", hdc, EmfTypeEmfOnly, &myRect,
                           MetafileFrameUnitPixel, L"Test File", &myMetafile);
status = GdipGetImageGraphicsContext(myMetafile, &myGraphics);
if (Ok != status)
    return status;
GpPen *myPen;
GdipCreatePen1(MakeARGB(255, 0, 0, 200), 1.0, UnitPixel, &myPen);
GdipSetSmoothingMode(myGraphics, SmoothingModeAntiAlias);
GdipDrawLineI(myGraphics, myPen, 0, 0, 60, 40);
GdipDeleteGraphics(myGraphics);
GdipDeletePen(myPen);
GdipDisposeImage(myMetafile);
```

As the preceding example shows, the **GpGraphics** class is the key to recording instructions and settings in a **GpMetafile** object. Any call made to a method of a **GpGraphics** object can be recorded in a **GpMetafile** object. Likewise, you can set any property of a **GpGraphics** object and record that setting in a **GpMetafile** object. The recording ends when the **GpGraphics** object is deleted or goes out of scope.

The following example displays the metafile created in the preceding example. The metafile is displayed with its upper-left corner at (100, 100).

```
status = GdipCreateFromHDC(hdc, &myGraphics);
if (Ok != status)
    return status;
GpImage *myImage;
GdipLoadImageFromFile(L"MyDiskFile.emf", &myImage);
GdipDrawImageI(myGraphics, myImage, 100, 100);
GdipDeleteGraphics(myGraphics);
```

The following example records several property settings (clipping region, world transformation, and smoothing mode) in a **GpMetafile** object. Then the code records several drawing instructions. The instructions and settings are saved in a disk file.

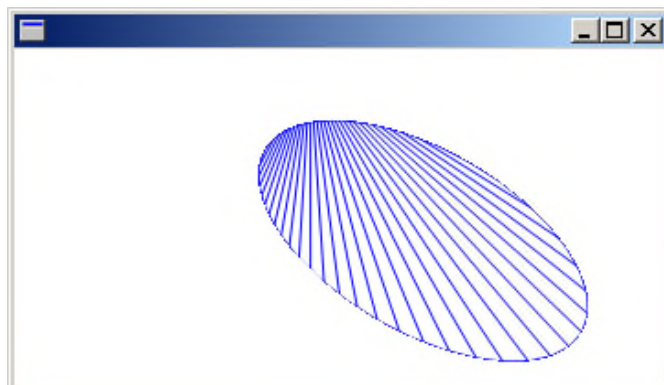
```
GdipRecordMetafileFileNameI(L"MyDiskFile2.emf", hdc, EmfTypeEmfOnly, &myRect, MetafileFrameUnitPixel,
                                                                    L"Test File", &myMetafile);

status = GdipGetImageGraphicsContext(myMetafile, &myGraphics);
if (Ok != status)
    return status;
GdipSetSmoothingMode(myGraphics, SmoothingModeAntiAlias);
GdipRotateWorldTransform(myGraphics, 30, MatrixOrderPrepend);
// Create an elliptical clipping region.
GpPath *myPath;
GdipCreatePath(FillModeAlternate, &myPath);
GdipAddPathEllipseI(myPath, 0, 0, 200, 100);
GpRegion *myRegion;
GdipCreateRegionPath(myPath, &myRegion);
GdipSetClipRegion(myGraphics, myRegion, CombineModeReplace);
GdipCreatePen1(MakeARGB(255, 0, 0, 255), 2.0, UnitPixel, &myPen);
GdipDrawPath(myGraphics, myPen, myPath);
for (INT j = 0; j <= 300; j += 10)
{
    GdipDrawLineI(myGraphics, myPen, 0, 0, 300 - j, j);
}
GdipDeleteGraphics(myGraphics);
GdipDeletePen(myPen);
GdipDisposeImage(myMetafile);
GdipDeleteRegion(myRegion);
GdipDeletePath(myPath);
```

The following example displays the metafile image created in the preceding example.

```
status = GdipCreateFromHDC(hdc, &myGraphics);
if (Ok != status)
    return status;
GdipLoadImageFromFile(L"MyDiskFile2.emf", &myImage);
GdipDrawImageI(myGraphics, myImage, 200, 100);
GdipDeleteGraphics(myGraphics);
```

The following illustration shows the output of the preceding code. Note the antialiasing, the elliptical clipping region, and the 30-degree rotation.



7.4 Drawing, Positioning, and Cloning Images

You can use the **GpImage** class to load and display raster images (bitmaps) and vector images (metafiles). To display an image, you need a **GpGraphics** object and an **GpImage** object. The **GpGraphics** object provides the **GdipDrawImageI** method, which receives the address of the **GpImage** object as an argument.

The following example constructs an **GpImage** object from the file Climber.jpg and then displays the image. The destination point for the upper-left corner of the image, (10, 10), is specified in the second and third parameters of the **GdipDrawImageI** method.

```
GpImage *myImage;  
GdipLoadImageFromFile(L"Climber.jpg", &myImage);  
GdipDrawImageI(myGraphics, myImage, 10, 10);  
GdipDisposeImage(myImage);
```

The preceding code, along with a particular file, Climber.jpg, produced the following output.



You can construct **GpImage** objects from a variety of graphics file formats: BMP, GIF, JPEG, Exif, PNG, TIFF, WMF, EMF, and ICON.

The **GpImage** class provides a **GdipCloneImage** and **GdipCloneBitmapArea** method that you can use to make a full or partial copy of an existing **GpImage**, **GpMetafile**, or **GpBitmap** object.

The following example creates a **GpBitmap** object by cloning the top half of an existing **GpBitmap** object. Then both images are displayed.

```
GpBitmap* originalBitmap;  
GdipCreateBitmapFromFile(L"Spiral.png", &originalBitmap);  
UINT width;  
UINT height;  
GdipGetImageWidth(originalBitmap, &width);  
GdipGetImageHeight(originalBitmap, &height);  
height /= 2; //half image  
GpBitmap* secondBitmap;  
GdipCloneBitmapAreaI(0, 0, width, height, PixelFormatDontCare, originalBitmap, &secondBitmap);  
GdipDrawImageI(myGraphics, originalBitmap, 200, 10);  
GdipDrawImageI(myGraphics, secondBitmap, 200+width, 10);  
GdipDisposeImage(originalBitmap);  
GdipDisposeImage(secondBitmap);
```



The preceding code, along with a particular file, Spiral.png, produced the output on the left.

7.5 Cropping and Scaling Images

You can use the [GdiDrawImage](#) method of the [GpGraphics](#) class to draw and position images. DrawImage is an overloaded method, so there are several ways you can supply it with arguments. One variation of the [GdiDrawImage](#) method receives the address of an [GpImage](#) object and a reference to a [GpRect](#) object. The rectangle specifies the destination for the drawing operation; that is, it specifies the rectangle in which the image will be drawn. If the size of the destination rectangle is different from the size of the original image, the image is scaled to fit the destination rectangle. The following example draws the same image three times: once with no scaling, once with an expansion, and once with a compression.

```
GpBitmap *myBitmap;
GdiLoadImageFromFile(L"Spiral.png", &myBitmap);
GdiGetImageWidth(myBitmap, &width);
GdiGetImageHeight(myBitmap, &height);
GdiDrawImage(myGraphics, myBitmap, 10, 200);
GpImageAttributes *myImageAttributes;
GdiCreateImageAttributes(&myImageAttributes);
GdiDrawImageRectRectI(myGraphics, myBitmap, width+10, 200, width*2, height,
                      0, 0, width, height, UnitPixel, myImageAttributes, NULL, NULL);
GdiDrawImageRectRectI(myGraphics, myBitmap, 3*width+10, 200, width/2, height/2,
                      0, 0, width, height, UnitPixel, myImageAttributes, NULL, NULL);
GdiDisposeImageAttributes(myImageAttributes);
GdiDisposeImage(myBitmap);
```

The preceding code, along with a particular file, Spiral.png, produced the following output.



The [GdiDrawImageRectRectI](#) method have a source-rectangle parameter as well as a destination-rectangle parameter. The source rectangle specifies the portion of the original image that will be drawn. The destination rectangle specifies where that portion of the image will be drawn. If the size of the destination rectangle is different from the size of the source rectangle, the image is scaled to fit the destination rectangle.

The following example constructs a [GpBitmap](#) object from the file Runner.jpg. The entire image is drawn with no scaling at (0, 0). Then a small portion of the image is drawn twice: once with a compression and once with an expansion.

```
GdiLoadImageFromFile(L"Runner.jpg", &myBitmap);
// Draw the original image.
GdiDrawImage(myGraphics, myBitmap, 500, 10);
// The rectangle (in myBitmap) with upper-left corner (80, 70),
// width 80, and height 45, encloses one of the runner's hands.
GdiCreateImageAttributes(&myImageAttributes);
GdiDrawImageRectRectI(myGraphics, myBitmap, 700, 10, 20, 16,
                      80, 70, 80, 45, UnitPixel, myImageAttributes, NULL, NULL);
GdiDrawImageRectRectI(myGraphics, myBitmap, 700, 50, 200, 160,
                      80, 70, 80, 45, UnitPixel, myImageAttributes, NULL, NULL);
GdiDisposeImageAttributes(myImageAttributes);
GdiDisposeImage(myBitmap);
```

The following illustration shows the unscaled image, and the compressed and expanded image portions.

