Graphics and Drawing in Windows Forms

.NET Framework (current version)

The common language runtime uses an advanced implementation of the Windows Graphics Device Interface (GDI) called GDI+. With GDI+ you can create graphics, draw text, and manipulate graphical images as objects. GDI+ is designed to offer performance and ease of use. You can use GDI+ to render graphical images on Windows Forms and controls. Although you cannot use GDI+ directly on Web Forms, you can display graphical images through the Image Web Server control.

In this section, you will find topics that introduce the fundamentals of GDI+ programming. Although not intended to be a comprehensive reference, this section includes information about the Graphics, Pen, Brush, and Color objects, and explains how to perform such tasks as drawing shapes, drawing text, or displaying images. For more information, see "GDI+ Reference" in the MSDN library at http://msdn.microsoft.com/library.

In This Section

Graphics Overview (Windows Forms)
: Provides an introduction to the graphics-related managed classes.

About GDI+ Managed Code
: Provides information about the managed GDI+ classes.

Using Managed Graphics Classes
: Demonstrates how to complete a variety of tasks using the GDI+ managed classes.

Reference

System.Drawing
: Provides access to GDI+ basic graphics functionality.

System.Drawing.Drawing2D
: Provides advanced two-dimensional and vector graphics functionality.

System.Drawing.Imaging
: Provides advanced GDI+ imaging functionality.

System.Drawing.Text
: Provides advanced GDI+ typography functionality. The classes in this namespace can be used to create and use collections of fonts.

System.Drawing.Printing
: Provides printing functionality.

Related Sections

Custom Control Painting and Rendering

Details how to provide code for painting controls.

Overview of Graphics

.NET Framework (current version)

GDI+ is an application programming interface (API) that forms the subsystem of the Microsoft Windows operating system. GDI+ is responsible for displaying information on screens and printers. As its name suggests, GDI+ is the successor to GDI, the Graphics Device Interface included with earlier versions of Windows.

Managed Class Interface

The GDI+ API is exposed through a set of classes deployed as managed code. This set of classes is called the *managed class interface* to GDI+. The following namespaces make up the managed class interface:

- System.Drawing
- System.Drawing.Drawing2D
- System.Drawing.Imaging
- System.Drawing.Text
- System.Drawing.Printing

With a Graphics Device Interface, such as GDI+, you can display information on a screen or printer without having to be concerned about the details of a particular display device. The programmer makes calls to methods provided by GDI+ classes. Those methods, in turn, make the appropriate calls to specific device drivers. GDI+ insulates the application from the graphics hardware. It is this insulation that enables a programmer to create device-independent applications.

See Also

Graphics Overview (Windows Forms)

Three Categories of Graphics Services

.NET Framework (current version)

The graphics offerings in Windows Forms fall into the following three broad categories:

- Two-dimensional (2-D) vector graphics
- Imaging
- Typography

2-D Vector Graphics

Two-dimensional vector graphics are primitives; such as lines, curves, and figures; that are specified by sets of points on a coordinate system. For example, a straight line is specified by its two endpoints, and a rectangle is 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 is specified by an array of points that are connected by straight lines. A Bézier spline is a sophisticated curve specified by four control points.

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

You can record a vector image, which is a sequence of graphics commands, in a metafile. GDI+ provides the Metafile class for recording, displaying, and saving metafiles. With the MetafileHeader and MetaHeader classes, you can inspect the data stored in a metafile header.

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 are difficult to specify as collections of lines and curves. A high-resolution digital photograph of a crowded baseball stadium is even more difficult to create with vector techniques. Images of this type are stored as bitmaps, which are arrays of numbers that represent the colors of individual dots on the screen. GDI+ provides the Bitmap class for displaying, manipulating, and saving bitmaps.

Typography

Typography is the display of text in a variety of fonts, sizes, and styles. GDI+ provides extensive 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.

In addition, Windows Forms offers the option to draw text with GDI capabilities in its TextRenderer class.

See Also

Graphics Overview (Windows Forms)
About GDI+ Managed Code
Using Managed Graphics Classes

Structure of the Graphics Interface

.NET Framework (current version)

The managed class interface to GDI+ contains about 60 classes, 50 enumerations, and 8 structures. The Graphics class is at the core of GDI+ functionality; it is the class that actually draws lines, curves, figures, images, and text.

Important Classes

Many classes work together with the Graphics class. For example, the DrawLine method receives a Pen object, which holds attributes (color, width, dash style, and the like) of the line to be drawn. The FillRectangle method can receive a pointer to a LinearGradientBrush object, which works with the Graphics object to fill a rectangle with a gradually changing color. Font and StringFormat objects influence the way a Graphics object draws text. A Matrix object stores and manipulates the world transformation of a Graphics object, which is used to rotate, scale, and flip images.

GDI+ provides several structures (for example, Rectangle, Point, and Size) for organizing graphics data. Also, certain classes serve primarily as structured data types. For example, the BitmapData class is a helper for the Bitmap class, and the PathData class is a helper for the GraphicsPath class.

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

See Also

Graphics Overview (Windows Forms)
About GDI+ Managed Code
Using Managed Graphics Classes

About GDI+ Managed Code

.NET Framework (current version)

GDI+ is the portion of the Windows operating system that provides two-dimensional vector graphics, imaging, and typography. GDI+ improves on GDI (the Graphics Device Interface included with earlier versions of Windows) by adding new features and by optimizing existing features.

The GDI+ managed class interface (a set of wrappers) is part of the .NET Framework, an environment for building, deploying, and running XML Web services and other applications.

This section provides information about the GDI+ API for programmers using managed code.

In This Section

Lines, Curves, and Shapes
: Discusses vector graphics.

Images, Bitmaps, and Metafiles
: Discusses the type of images available and how to work with them.

Coordinate Systems and Transformations
: Discusses how to transform graphics with GDI+.

Reference

System.Drawing.Graphics
: Describes this class and has links to all its members.

System.Drawing.Image
: Describes this class and has links to all its members.

System.Drawing.Bitmap,
: Describes this class and has links to all its members.

System.Drawing.Imaging.Metafile,
: Describes this class and has links to all its members.

System.Drawing.Font,
: Describes this class and has links to all its members.

System.Drawing.Brush,
: Describes this class and has links to all its members.

System.Drawing.Color,
: Describes this class and has links to all its members.

System.Drawing.Drawing2D.Matrix

Describes this class and has links to all its members.

System.Windows.Forms.TextRenderer
Describes this class and has links to all its members.

Related Sections

Using Managed Graphics Classes.
Contains links to topics that demonstrate how to use the **Graphics** programming interface.

Lines, Curves, and Shapes

.NET Framework (current version)

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

In This Section

Vector Graphics Overview
: Discusses vector graphics.

Pens, Lines, and Rectangles in GDI+
: Discusses drawing lines and rectangles.

Ellipses and Arcs in GDI+
: Defines arcs and ellipses and identifies the classes needed to draw them.

Polygons in GDI+
: Defines polygons and identifies the classes needed to draw them.

Cardinal Splines in GDI+
: Defines cardinal splines and identifies the classes needed to draw them.

Bézier Splines in GDI+
: Defines Bezier splines and identifies the classes needed to draw them.

Graphics Paths in GDI+
: Describes paths and how to create and draw them.

Brushes and Filled Shapes in GDI+
: Describes brush types and how to use them.

Open and Closed Curves in GDI+
: Defines open and closed curves and how to draw and fill them.

Regions in GDI+
: Describes the methods associated with regions.

Restricting the Drawing Surface in GDI+
: Describes clipping and how to use it.

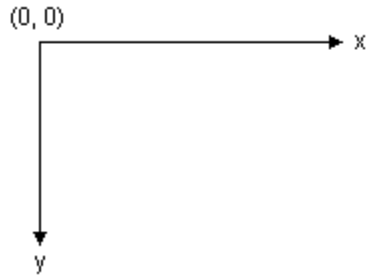
Antialiasing with Lines and Curves
: Defines antialiasing and how use antialiasing when drawing lines and curves.

Vector Graphics Overview

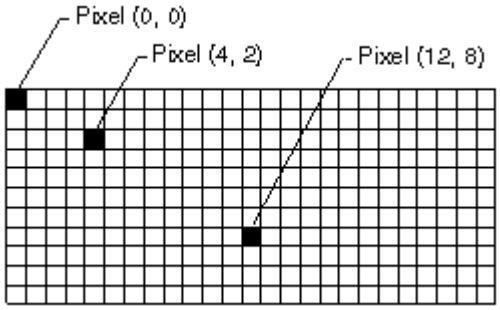
.NET Framework (current version)

GDI+ draws lines, rectangles, and other shapes 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.

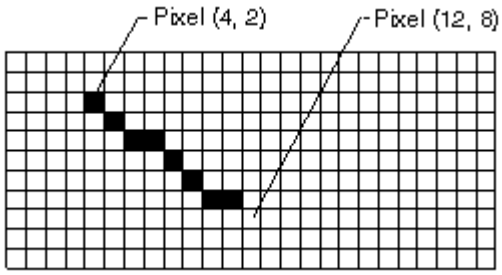
The Building Blocks of GDI+



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



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



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
- Bezier splines

Methods For Drawing with a Graphics Object

The Graphics class in GDI+ provides the following methods for drawing the items in the previous list: DrawLine, DrawRectangle, DrawEllipse, DrawPolygon, DrawArc, DrawCurve (for cardinal splines), and DrawBezier. Each of these methods is overloaded; that is, each method supports several different parameter lists. For example, one variation of the DrawLine method receives a Pen object and four integers, while another variation of the DrawLine method receives a Pen object and two Point objects.

The methods for drawing lines, rectangles, and Bézier splines have plural companion methods that draw several items in a single call: DrawLines, DrawRectangles, and DrawBeziers. Also, the DrawCurve method has a companion method, DrawClosedCurve, that closes a curve by connecting the ending point of the curve to the starting point.

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

VB

```
myGraphics.DrawRectangle(myPen, 20, 10, 100, 50)
```

See Also

System.Drawing.Graphics
System.Drawing.Pen
Lines, Curves, and Shapes
How to: Create Graphics Objects for Drawing

Pens, Lines, and Rectangles in GDI+

.NET Framework (current version)

To draw lines with GDI+ you need to create a Graphics object and a Pen object. The Graphics object provides the methods that actually do the drawing, and the Pen object stores attributes, such as line color, width, and style.

Drawing a Line

To draw a line, call the DrawLine method of the Graphics object. The Pen 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):

VB

```
myGraphics.DrawLine(myPen, 4, 2, 12, 6)
```

DrawLine is an overloaded method of the Graphics class, so there are several ways you can supply it with arguments. For example, you can construct two Point objects and pass the Point objects as arguments to the DrawLine method:

VB

```
Dim myStartPoint As New Point(4, 2)
Dim myEndPoint As New Point(12, 6)
myGraphics.DrawLine(myPen, myStartPoint, myEndPoint)
```

Constructing a Pen

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

VB

```
Dim myPen As New Pen(Color.Blue, 2)
myGraphics.DrawLine(myPen, 0, 0, 60, 30)
```

Dashed Lines and Line Caps

The Pen object also exposes properties, such as DashStyle, that you can use to specify features of the line. The following

example draws a dashed line from (100, 50) to (300, 80):

VB

```
myPen.DashStyle = DashStyle.Dash
myGraphics.DrawLine(myPen, 100, 50, 300, 80)
```

You can use the properties of the Pen object to set many more attributes of the line. The StartCap and EndCap properties specify the appearance of the ends of the line; the ends can be flat, square, rounded, triangular, or a custom shape. The LineJoin property lets you specify whether connected lines are mitered (joined with sharp corners), beveled, rounded, or clipped. The following illustration shows lines with various cap and join styles.

Drawing a Rectangle

Drawing rectangles with GDI+ is similar to drawing lines. To draw a rectangle, you need a Graphics object and a Pen object. The Graphics object provides a DrawRectangle method, and the Pen object stores attributes, such as line width and color. The Pen 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:

VB

```
myGraphics.DrawRectangle(myPen, 100, 50, 80, 40)
```

DrawRectangle is an overloaded method of the Graphics class, so there are several ways you can supply it with arguments. For example, you can construct a Rectangle object and pass the Rectangle object to the DrawRectangle method as an argument:

VB

```
Dim myRectangle As New Rectangle(100, 50, 80, 40)
myGraphics.DrawRectangle(myPen, myRectangle)
```

A Rectangle object has methods and properties for manipulating and gathering information about the rectangle. For example, the Inflate and Offset methods change the size and position of the rectangle. The IntersectsWith method tells you whether the rectangle intersects another given rectangle, and the Contains method tells you whether a given point is inside the rectangle.

See Also

System.Drawing.Graphics

System.Drawing.Pen
System.Drawing.Rectangle
How to: Create a Pen
How to: Draw a Line on a Windows Form
How to: Draw an Outlined Shape

Ellipses and Arcs in GDI+

.NET Framework (current version)

You can easily draw ellipses and arcs using the DrawEllipse and DrawArc methods of the Graphics class.

Drawing an Ellipse

To draw an ellipse, you need a Graphics object and a Pen object. The Graphics object provides the DrawEllipse method, and the Pen object stores attributes, such as width and color, of the line used to render the ellipse. The Pen 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 illustration shows an ellipse along with its bounding rectangle.

The following example draws an ellipse; the bounding rectangle has a width of 80, a height of 40, and an upper-left corner of (100, 50):

VB

```
myGraphics.DrawEllipse(myPen, 100, 50, 80, 40)
```

DrawEllipse is an overloaded method of the Graphics class, so there are several ways you can supply it with arguments. For example, you can construct a Rectangle and pass the Rectangle to the DrawEllipse method as an argument:

VB

```
Dim myRectangle As New Rectangle(100, 50, 80, 40)
myGraphics.DrawEllipse(myPen, myRectangle)
```

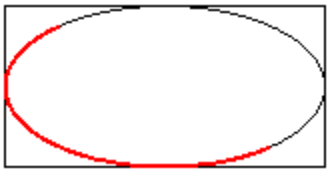
Drawing an Arc

An arc is a portion of an ellipse. To draw an arc, you call the DrawArc method of the Graphics class. The parameters of the DrawArc method are the same as the parameters of the DrawEllipse 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:

VB

```
myGraphics.DrawArc(myPen, 100, 50, 140, 70, 30, 180)
```

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



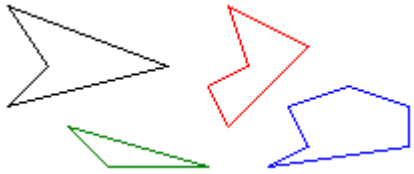
See Also

System.Drawing.Graphics
System.Drawing.Pen
Lines, Curves, and Shapes
How to: Create Graphics Objects for Drawing
How to: Create a Pen
How to: Draw an Outlined Shape

Polygons in GDI+

.NET Framework (current version)

A polygon is a closed shape 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. The following illustration shows several polygons.



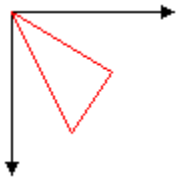
Drawing a Polygon

To draw a polygon, you need a Graphics object, a Pen object, and an array of Point (or PointF) objects. The Graphics object provides the DrawPolygon method. The Pen object stores attributes, such as width and color, of the line used to render the polygon, and the array of Point objects stores the points to be connected by straight lines. The Pen object and the array of Point 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).

VB

```
Dim myPointArray As Point() = _
   {New Point(0, 0), New Point(50, 30), New Point(30, 60)}
myGraphics.DrawPolygon(myPen, myPointArray)
```

The following illustration shows the polygon.



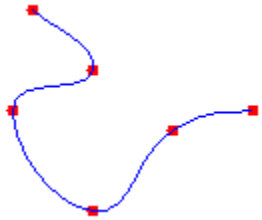
See Also

System.Drawing.Graphics
System.Drawing.Pen
Lines, Curves, and Shapes
How to: Create a Pen

Cardinal Splines in GDI+

.NET Framework (current version)

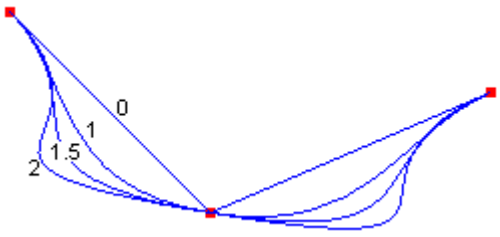
A cardinal spline is a sequence of individual curves joined to form a larger curve. The spline is specified by an array of points and a tension parameter. A cardinal spline passes smoothly through each point in the array; there are no sharp corners and no abrupt changes in the tightness of the curve.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



Physical and Mathematical Splines

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 pen or 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. 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.



The four splines in the preceding illustration 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 an instance of the Graphics class, a Pen, and an array of Point objects The instance of the Graphics class provides the DrawCurve method, which draws the spline, and the Pen stores attributes of the spline, such as line width and color. The array of Point objects stores the points that the curve will pass through. The following

code example shows how to draw a cardinal spline that passes through the points in `myPointArray`. The third parameter is the tension.

VB

```
myGraphics.DrawCurve(myPen, myPointArray, 1.5F)
```

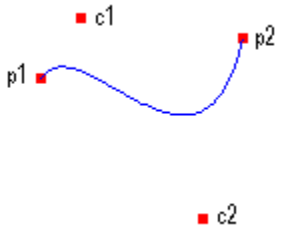
See Also

Lines, Curves, and Shapes
Constructing and Drawing Curves

Bézier Splines in GDI+

.NET Framework (current version)

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 does not 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.



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. The tangent line at the endpoint p2 is the line drawn from c2 to p2.

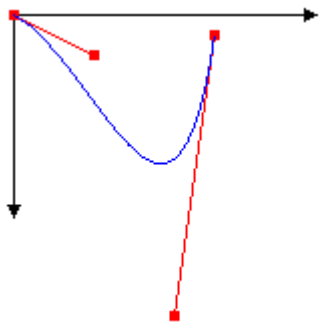
Drawing Bézier Splines

To draw a Bézier spline, you need an instance of the Graphics class and a Pen. The instance of the Graphics class provides the DrawBezier method, and the Pen stores attributes, such as width and color, of the line used to render the curve. The Pen 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):

VB

```
myGraphics.DrawBezier(myPen, 0, 0, 40, 20, 80, 150, 100, 10)
```

The following 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 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 following illustration.

See Also

- System.Drawing.Graphics
- System.Drawing.Pen
- Lines, Curves, and Shapes
- Constructing and Drawing Curves
- How to: Create Graphics Objects for Drawing
- How to: Create a Pen

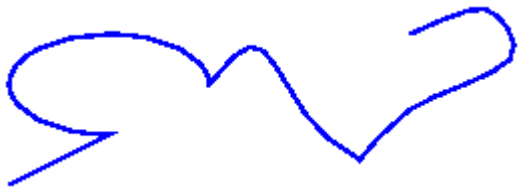
Graphics Paths in GDI+

.NET Framework (current version)

Paths are formed by combining lines, rectangles, and simple curves. Recall from the Vector Graphics Overview 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 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 DrawPath method of the Graphics class. The following illustration shows a path created by combining a line, an arc, a Bézier spline, and a cardinal spline.



Using a Path

The GraphicsPath class provides the following methods for creating a sequence of items to be drawn: AddLine, AddRectangle, AddEllipse, AddArc, AddPolygon, AddCurve (for cardinal splines), and AddBezier. Each of these methods is overloaded; that is, each method supports several different parameter lists. For example, one variation of the AddLine method receives four integers, and another variation of the AddLine method receives two Point objects.

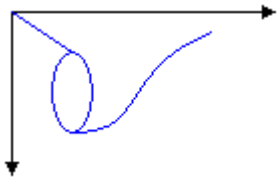
The methods for adding lines, rectangles, and Bézier splines to a path have plural companion methods that add several items to the path in a single call: AddLines, AddRectangles, and AddBeziers. Also, the AddCurve and AddArc methods have companion methods, AddClosedCurve and AddPie, that add a closed curve or pie to the path.

To draw a path, you need a Graphics object, a Pen object, and a GraphicsPath object. The Graphics object provides the DrawPath method, and the Pen object stores attributes, such as width and color, of the line used to render the path. The GraphicsPath object stores the sequence of lines and curves that make up the path. The Pen object and the GraphicsPath object are passed as arguments to the DrawPath method. The following example draws a path that consists of a line, an ellipse, and a Bézier spline:

VB

```
myGraphicsPath.AddLine(0, 0, 30, 20)
myGraphicsPath.AddEllipse(20, 20, 20, 40)
myGraphicsPath.AddBezier(30, 60, 70, 60, 50, 30, 100, 10)
myGraphics.DrawPath(myPen, myGraphicsPath)
```

The following illustration shows the path.



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.

VB

```
myGraphicsPath.AddPath(graphicsPath1, False)
myGraphicsPath.AddPath(graphicsPath2, False)
```

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:

VB

```
Dim myGraphicsPath As New GraphicsPath()

Dim myPointArray As Point() = { _
   New Point(5, 30), _
   New Point(20, 40), _
   New Point(50, 30)}

Dim myFontFamily As New FontFamily("Times New Roman")
Dim myPointF As New PointF(50, 20)
Dim myStringFormat As New StringFormat()

myGraphicsPath.AddArc(0, 0, 30, 20, -90, 180)
myGraphicsPath.StartFigure()
myGraphicsPath.AddCurve(myPointArray)
myGraphicsPath.AddString("a string in a path", myFontFamily, _
   0, 24, myPointF, myStringFormat)
myGraphicsPath.AddPie(230, 10, 40, 40, 40, 110)
myGraphics.DrawPath(myPen, myGraphicsPath)
```

The following illustration shows the path. Note that a path does not have to be connected; the arc, cardinal spline, string,

and pie are separated.

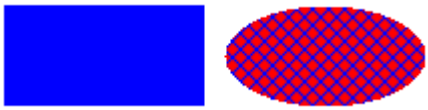
See Also

System.Drawing.Drawing2D.GraphicsPath
System.Drawing.Point
Lines, Curves, and Shapes
How to: Create Graphics Objects for Drawing
Constructing and Drawing Paths

Brushes and Filled Shapes in GDI+

.NET Framework (current version)

A closed shape, such as a rectangle or an ellipse, consists of an outline and an interior. The outline is drawn with a pen and the interior is filled with a brush. GDI+ provides several brush classes for filling the interiors of closed shapes: SolidBrush, HatchBrush, TextureBrush, LinearGradientBrush, and PathGradientBrush. All of these classes inherit from the Brush class. The following illustration shows a rectangle filled with a solid brush and an ellipse filled with a hatch brush.



Solid Brushes

To fill a closed shape, you need an instance of the Graphics class and a Brush. The instance of the Graphics class provides methods, such as FillRectangle and FillEllipse, and the Brush stores attributes of the fill, such as color and pattern. The Brush is passed as one of the arguments to the fill method. The following code example shows how to fill an ellipse with a solid red color.

VB

```
Dim mySolidBrush As New SolidBrush(Color.Red)
myGraphics.FillEllipse(mySolidBrush, 0, 0, 60, 40)
```

Note

In the preceding example, the brush is of type SolidBrush, which inherits from Brush.

Hatch Brushes

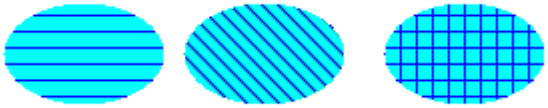
When you fill a shape with a hatch brush, you specify a foreground color, a background color, and a hatch style. The foreground color is the color of the hatching.

VB

```
Dim myHatchBrush As _
   New HatchBrush(HatchStyle.Vertical, Color.Blue, Color.Green)
```

GDI+ provides more than 50 hatch styles; the three styles shown in the following illustration are Horizontal,

ForwardDiagonal, and Cross.



Texture Brushes

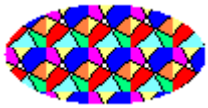
With a texture brush, you can fill a shape with a pattern stored in a bitmap. For example, suppose the following picture is stored in a disk file named `MyTexture.bmp`.

The following code example shows how to fill an ellipse by repeating the picture stored in `MyTexture.bmp`.

VB

```
Dim myImage As Image = Image.FromFile("MyTexture.bmp")
Dim myTextureBrush As New TextureBrush(myImage)
myGraphics.FillEllipse(myTextureBrush, 0, 0, 100, 50)
```

The following illustration shows the filled ellipse.



Gradient Brushes

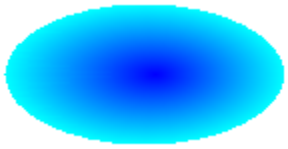
GDI+ provides two kinds of gradient brushes: linear and path. You can use a linear gradient brush to fill a shape with color that changes gradually as you move across the shape horizontally, vertically, or diagonally. The following code example shows how to fill an ellipse with a horizontal gradient brush that changes from blue to green as you move from the left edge of the ellipse to the right edge.

VB

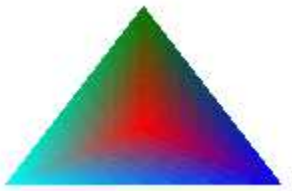
```
Dim myLinearGradientBrush As New LinearGradientBrush( _
   myRectangle, _
   Color.Blue, _
   Color.Green, _
   LinearGradientMode.Horizontal)
myGraphics.FillEllipse(myLinearGradientBrush, myRectangle)
```

The following illustration shows the filled ellipse.

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



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



See Also

System.Drawing.SolidBrush
System.Drawing.Drawing2D.HatchBrush
System.Drawing.TextureBrush
System.Drawing.Drawing2D.LinearGradientBrush
Lines, Curves, and Shapes
How to: Draw a Filled Rectangle on a Windows Form
How to: Draw a Filled Ellipse on a Windows Form

Open and Closed Curves in GDI+

.NET Framework (current version)

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

Managed Interface for Curves

Closed curves have an interior and therefore can be filled with a brush. The Graphics class in GDI+ provides the following methods for filling closed shapes and curves: FillRectangle, FillEllipse, FillPie, FillPolygon, FillClosedCurve, FillPath, and FillRegion. Whenever you call one of these methods, you must pass one of the specific brush types (SolidBrush, HatchBrush, TextureBrush, LinearGradientBrush, or PathGradientBrush) as an argument.

The FillPie method is a companion to the DrawArc 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:

VB

```
myGraphics.FillPie(mySolidBrush, 0, 0, 140, 70, 0, 120)
myGraphics.DrawArc(myPen, 0, 0, 140, 70, 0, 120)
```

The following 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.

VB

```
Dim myPointArray As Point() = _
   {New Point(0, 0), New Point(60, 20), New Point(40, 50)}
myGraphics.DrawClosedCurve(myPen, myPointArray)
myGraphics.FillClosedCurve(mySolidBrush, myPointArray)
```

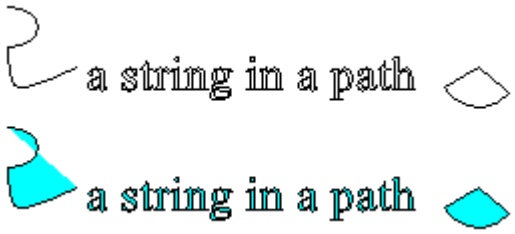
The FillPath method fills the interiors of the separate pieces of a path. If a piece of a path doesn't form a closed curve or

shape, the FillPath method automatically closes that piece of the path before filling it. The following example draws and fills a path that consists of an arc, a cardinal spline, a string, and a pie:

VB

```
Dim mySolidBrush As New SolidBrush(Color.Aqua)
Dim myGraphicsPath As New GraphicsPath()

Dim myPointArray As Point() = { _
   New Point(15, 20), _
   New Point(20, 40), _
   New Point(50, 30)}

Dim myFontFamily As New FontFamily("Times New Roman")
Dim myPointF As New PointF(50, 20)
Dim myStringFormat As New StringFormat()

myGraphicsPath.AddArc(0, 0, 30, 20, -90, 180)
myGraphicsPath.AddCurve(myPointArray)
myGraphicsPath.AddString("a string in a path", myFontFamily, _
   0, 24, myPointF, myStringFormat)
myGraphicsPath.AddPie(230, 10, 40, 40, 40, 110)

myGraphics.FillPath(mySolidBrush, myGraphicsPath)
myGraphics.DrawPath(myPen, myGraphicsPath)
```

The following illustration shows the path with and without the solid fill. Note that the text in the string is outlined, but not filled, by the DrawPath method. It is the FillPath method that paints the interiors of the characters in the string.



See Also

System.Drawing.Drawing2D.GraphicsPath
System.Drawing.Pen
System.Drawing.Point
Lines, Curves, and Shapes
How to: Create Graphics Objects for Drawing
Constructing and Drawing Paths

Regions in GDI+

.NET Framework (current version)

A region is a portion of the display area of an output device. 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.

Using Regions

Regions are often used for clipping and hit testing. Clipping involves restricting drawing to a certain region of the display area, usually the portion that needs to be updated. Hit testing involves checking to determine 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 a path. You can also create complex regions by combining existing regions. The Region class provides the following methods for combining regions: Intersect, Union, Xor, Exclude, and Complement.

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 shown in the preceding illustration.

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 Graphics object, a Brush object, and a Region object. The Graphics object provides the FillRegion method, and the Brush object stores attributes of the fill, such as color or pattern. The following example fills a region with a solid color.

VB

```
myGraphics.FillRegion(mySolidBrush, myRegion)
```

See Also

System.Drawing.Region
Lines, Curves, and Shapes
Using Regions

Restricting the Drawing Surface in GDI+

.NET Framework (current version)

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

Clipping with Regions

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 Graphics object, set its Clip property, and then call the drawing methods of that same Graphics object:

VB

```
myGraphics.Clip = myRegion
myGraphics.DrawLine(myPen, 0, 0, 200, 200)
```

See Also

System.Drawing.Graphics
System.Drawing.Region
Lines, Curves, and Shapes
Using Regions

Restricting the Drawing Surface in GDI+

.NET Framework (current version)

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

Clipping with Regions

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 Graphics object, set its Clip property, and then call the drawing methods of that same Graphics object:

VB

```
myGraphics.Clip = myRegion
myGraphics.DrawLine(myPen, 0, 0, 200, 200)
```

See Also

System.Drawing.Graphics
System.Drawing.Region
Lines, Curves, and Shapes
Using Regions

Images, Bitmaps, and Metafiles

.NET Framework (current version)

The **Image** class is an abstract base class that provides methods for working with raster images (bitmaps) and vector images (metafiles). The **Bitmap** class and the Metafile class both inherit from the **Image** class. The **Bitmap** class expands on the capabilities of the **Image** class by providing additional methods for loading, saving, and manipulating raster images. The Metafile class expands on the capabilities of the **Image** class by providing additional methods for recording and examining vector images.

In This Section

Types of Bitmaps
: Discusses the various image formats.

Metafiles in GDI+
: Discusses GDI+ support for metafiles.

Drawing, Positioning, and Cloning Images in GDI+
: Discusses methods for drawing vector and raster images with managed code.

Cropping and Scaling Images in GDI+
: Discusses methods for cropping and scaling vector and raster images with managed code

Reference

Image
: Describes this class and has links to all of its members.

Bitmap
: Describes this class and has links to all of its members

Related Sections

Working with Images, Bitmaps, Icons, and Metafiles
: Contains links to topics that demonstrate how to use images in your application.

Types of Bitmaps

.NET Framework (current version)

A bitmap is an array of bits that specify 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

Disk files that store bitmaps usually contain one or more information blocks that store information such as the 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, FFFFFF 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 FF0000 FFFFFF FF0000 FFFFFF FF0000 FFFFFF 00FF00
00FF00 FFFFFF FF0000 FFFFFF FF0000 FFFFFF FF0000 00FF00
0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF
```

Graphics File Formats

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

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. BMP files are usually not compressed and, therefore, are not well suited for transfer across the Internet.

Graphics Interchange Format (GIF)

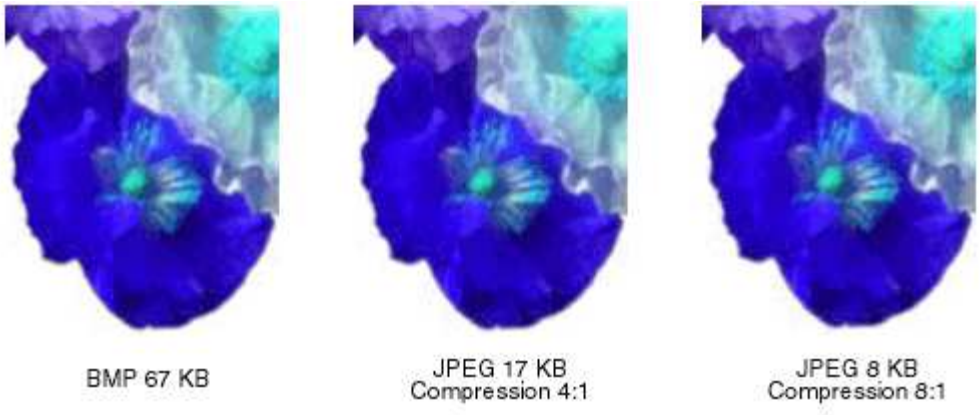
GIF is a common format for images that appear on Web pages. GIFs work well for line drawings, pictures with blocks of solid color, and pictures with sharp boundaries between colors. GIFs are compressed, but no information is lost in the compression process; a decompressed image is exactly the same as 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)

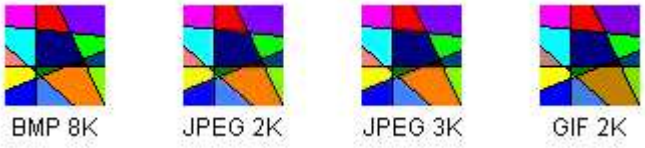
JPEG is a compression scheme that works well for natural scenes such as scanned photographs. Some information is

lost in the compression process, but often the loss is imperceptible to the human eye. JPEGs store 24 bits per pixel, so they are capable of displaying more than 16 million colors. 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 tend 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 grayscales 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.

See Also

System.Drawing.Image
System.Drawing.Bitmap
System.Drawing.Imaging.PixelFormat
Images, Bitmaps, and Metafiles
Working with Images, Bitmaps, Icons, and Metafiles

Metafiles in GDI+

.NET Framework (current version)

GDI+ provides the Metafile 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 Metafile object can be stored in memory or saved to a file or stream.

Metafile Formats

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

- Windows Metafile (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 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 displays a metafile that was previously saved as a file. The metafile is displayed with its upper-left corner at (100, 100).

VB

```
Public Sub Example_DisplayMetafile(ByVal e As PaintEventArgs)
    Dim myGraphics As Graphics = e.Graphics
    Dim myMetafile As New Metafile("SampleMetafile.emf")
    myGraphics.DrawImage(myMetafile, 100, 100)
End Sub
```

See Also

Images, Bitmaps, and Metafiles

Drawing, Positioning, and Cloning Images in GDI+

.NET Framework (current version)

You can use the Bitmap class to load and display raster images, and you can use the Metafile class to load and display vector images. The Bitmap and Metafile classes inherit from the Image class. To display a vector image, you need an instance of the Graphics class and a Metafile. To display a raster image, you need an instance of the Graphics class and a Bitmap. The instance of the Graphics class provides the DrawImage method, which receives the Metafile or Bitmap as an argument.

File Types and Cloning

The following code example shows how to construct a Bitmap from the file Climber.jpg and displays the bitmap. The destination point for the upper-left corner of the image, (10, 10), is specified in the second and third parameters.

VB

```
Dim myBitmap As New Bitmap("Climber.jpg")
myGraphics.DrawImage(myBitmap, 10, 10)
```

The following illustration shows the image.

You can construct Bitmap objects from a variety of graphics file formats: BMP, GIF, JPEG, EXIF, PNG, TIFF, and ICON.

The following code example shows how to construct Bitmap objects from a variety of file types and then displays the bitmaps.

VB

```
Dim myBMP As New Bitmap("SpaceCadet.bmp")
Dim myGIF As New Bitmap("Soda.gif")
Dim myJPEG As New Bitmap("Mango.jpg")
Dim myPNG As New Bitmap("Flowers.png")
Dim myTIFF As New Bitmap("MS.tif")
```

```
myGraphics.DrawImage(myBMP, 10, 10)
myGraphics.DrawImage(myGIF, 220, 10)
myGraphics.DrawImage(myJPEG, 280, 10)
myGraphics.DrawImage(myPNG, 150, 200)
myGraphics.DrawImage(myTIFF, 300, 200)
```

The Bitmap class provides a Clone method that you can use to make a copy of an existing Bitmap. The Clone method has a source rectangle parameter that you can use to specify the portion of the original bitmap that you want to copy. The following code example shows how to create a Bitmap by cloning the top half of an existing Bitmap. Then both images are drawn.

VB

```
Dim originalBitmap As New Bitmap("Spiral.png")
Dim sourceRectangle As New Rectangle(0, 0, originalBitmap.Width, _
   CType(originalBitmap.Height / 2, Integer))

Dim secondBitmap As Bitmap = originalBitmap.Clone(sourceRectangle, _
   PixelFormat.DontCare)

myGraphics.DrawImage(originalBitmap, 10, 10)
myGraphics.DrawImage(secondBitmap, 150, 10)
```

The following illustration shows the two images.

See Also

Images, Bitmaps, and Metafiles
How to: Create Graphics Objects for Drawing
Working with Images, Bitmaps, Icons, and Metafiles

Cropping and Scaling Images in GDI+

.NET Framework (current version)

You can use the DrawImage method of the Graphics class to draw and position vector images and raster images. DrawImage is an overloaded method, so there are several ways you can supply it with arguments.

DrawImage Variations

One variation of the DrawImage method receives a Bitmap and a Rectangle. The rectangle specifies the destination for the drawing operation; that is, it specifies the rectangle in which to draw the image. 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 code example shows how to draw the same image three times: once with no scaling, once with an expansion, and once with a compression:

VB

```
Dim myBitmap As New Bitmap("Spiral.png")

Dim expansionRectangle As New Rectangle(135, 10, _
   myBitmap.Width, myBitmap.Height)

Dim compressionRectangle As New Rectangle(300, 10, _
   CType(myBitmap.Width / 2, Integer), CType(myBitmap.Height / 2, Integer))

myGraphics.DrawImage(myBitmap, 10, 10)
myGraphics.DrawImage(myBitmap, expansionRectangle)
myGraphics.DrawImage(myBitmap, compressionRectangle)
```

The following illustration shows the three pictures.

Some variations of the DrawImage method have a source-rectangle parameter as well as a destination-rectangle parameter. The source-rectangle parameter specifies the portion of the original image to draw. The destination rectangle specifies the rectangle in which to draw that portion of the image. If the size of the destination rectangle is different from the size of the source rectangle, the picture is scaled to fit the destination rectangle.

The following code example shows how to construct a Bitmap 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.

VB

```
Dim myBitmap As New Bitmap("Runner.jpg")

' One hand of the runner
Dim sourceRectangle As New Rectangle(80, 70, 80, 45)

' Compressed hand
Dim destRectangle1 As New Rectangle(200, 10, 20, 16)

' Expanded hand
Dim destRectangle2 As New Rectangle(200, 40, 200, 160)

' Draw the original image at (0, 0).
myGraphics.DrawImage(myBitmap, 0, 0)

' Draw the compressed hand.
myGraphics.DrawImage( _
   myBitmap, destRectangle1, sourceRectangle, GraphicsUnit.Pixel)

' Draw the expanded hand.
myGraphics.DrawImage( _
   myBitmap, destRectangle2, sourceRectangle, GraphicsUnit.Pixel)
```

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

See Also

Images, Bitmaps, and Metafiles
Working with Images, Bitmaps, Icons, and Metafiles

Coordinate Systems and Transformations

.NET Framework (current version)

GDI+ provides a world transformation and a page transformation so that you can transform (rotate, scale, translate, and so on) the items you draw. The two transformations also allow you to work in a variety of coordinate systems.

In This Section

Types of Coordinate Systems
: Introduces coordinates systems and transformations.

Matrix Representation of Transformations
: Discusses using matrices for coordinate transformations.

Global and Local Transformations
: Discusses global and local transformations.

Reference

Matrix
: Encapsulates a 3-by-3 affine matrix that represents a geometric transform.

Related Sections

Using Transformations in Managed GDI+
: Provides a list of topics that provide more information about how to use matrix transformations.

About GDI+ Managed Code
: Contains a list of topics describing the graphics constructs you can use in the .NET Framework.

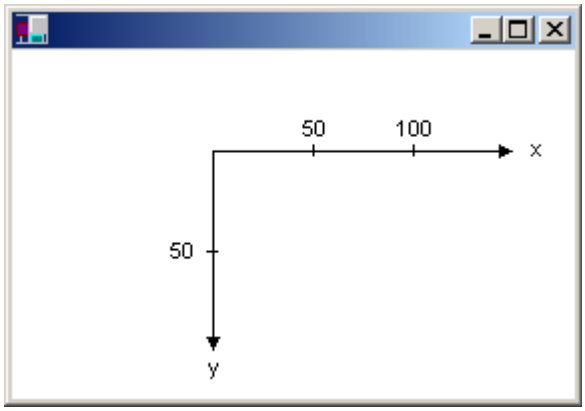
Types of Coordinate Systems

.NET Framework (current version)

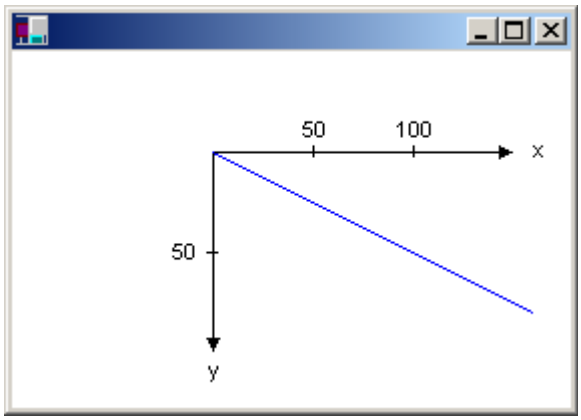
GDI+ uses three coordinate spaces: world, page, and device. World coordinates are the coordinates used to model a particular graphic world and are the coordinates you pass to methods in the .NET Framework. Page coordinates refer to the coordinate system used by a drawing surface, such as a form or control. Device coordinates are the coordinates used by the physical device being drawn on, such as a screen or sheet of paper. When you make the call `myGraphics.DrawLine(myPen, 0, 0, 160, 80)`, the points that you pass to the DrawLine method—`(0, 0)` and `(160, 80)`—are in the world coordinate space. Before GDI+ can draw the line on the screen, the coordinates pass through a sequence of transformations. One transformation, called the world transformation, converts world coordinates to page coordinates, and another transformation, called the page transformation, converts page coordinates to device coordinates.

Transforms and Coordinate Systems

Suppose you want to work with a coordinate system that has its origin in the body of the client area rather than the upper-left corner. Say, for example, that you want the origin to be 100 pixels from the left edge of the client area and 50 pixels from the top of the client area. The following illustration shows such a coordinate system.



When you make the call `myGraphics.DrawLine(myPen, 0, 0, 160, 80)`, you get the line shown in the following illustration.



The coordinates of the endpoints of your line in the three coordinate spaces are as follows:

World	(0, 0) to (160, 80)
Page	(100, 50) to (260, 130)
Device	(100, 50) to (260, 130)

Note that the page coordinate space has its origin at the upper-left corner of the client area; this will always be the case. Also note that because the unit of measure is the pixel, the device coordinates are the same as the page coordinates. If you set the unit of measure to something other than pixels (for example, inches), then the device coordinates will be different from the page coordinates.

The world transformation, which maps world coordinates to page coordinates, is held in the Transform property of the Graphics class. In the preceding example, the world transformation is a translation 100 units in the x direction and 50 units in the y direction. The following example sets the world transformation of a Graphics object and then uses that Graphics object to draw the line shown in the preceding figure:

VB

```
myGraphics.TranslateTransform(100, 50)
myGraphics.DrawLine(myPen, 0, 0, 160, 80)
```

The page transformation maps page coordinates to device coordinates. The Graphics class provides the PageUnit and PageScale properties for manipulating the page transformation. The Graphics class also provides two read-only properties, DpiX and DpiY, for examining the horizontal and vertical dots per inch of the display device.

You can use the PageUnit property of the Graphics class to specify a unit of measure other than the pixel.

Note

You cannot set the PageUnit property to World, as this is not a physical unit and will cause an exception.

The following example draws a line from (0, 0) to (2, 1), where the point (2, 1) is 2 inches to the right and 1 inch down from the point (0, 0):

VB

```
myGraphics.PageUnit = GraphicsUnit.Inch
myGraphics.DrawLine(myPen, 0, 0, 2, 1)
```

Note

If you don't specify a pen width when you construct your pen, the preceding example will draw a line that is one inch wide. You can specify the pen width in the second argument to the Pen constructor:

VB

```
Dim myPen As New Pen(Color.Black, 1 / myGraphics.DpiX)
```

If we assume that the display device has 96 dots per inch in the horizontal direction and 96 dots per inch in the vertical direction, the endpoints of the line in the preceding example have the following coordinates in the three coordinate spaces:

World	(0, 0) to (2, 1)
Page	(0, 0) to (2, 1)
Device	(0, 0, to (192, 96)

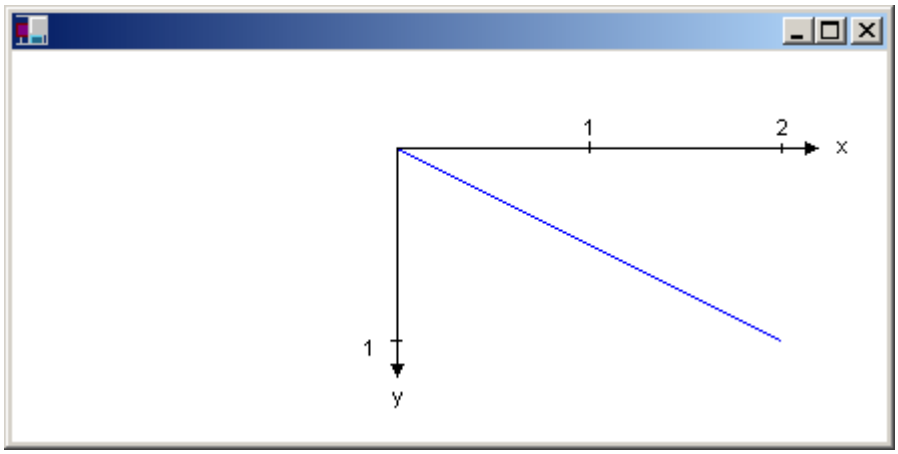
Note that because the origin of the world coordinate space is at the upper-left corner of the client area, the page coordinates are the same as the world coordinates.

You can combine the world and page transformations to achieve a variety of effects. For example, suppose you want to use inches as the unit of measure and you want the origin of your coordinate system to be 2 inches from the left edge of the client area and 1/2 inch from the top of the client area. The following example sets the world and page transformations of a Graphics object and then draws a line from (0, 0) to (2, 1):

VB

```
myGraphics.TranslateTransform(2, 0.5F)
myGraphics.PageUnit = GraphicsUnit.Inch
myGraphics.DrawLine(myPen, 0, 0, 2, 1)
```

The following illustration shows the line and coordinate system.



If we assume that the display device has 96 dots per inch in the horizontal direction and 96 dots per inch in the vertical direction, the endpoints of the line in the preceding example have the following coordinates in the three coordinate spaces:

World	(0, 0) to (2, 1)
Page	(2, 0.5) to (4, 1.5)
Device	(192, 48) to (384, 144)

See Also

Coordinate Systems and Transformations
Matrix Representation of Transformations

Matrix Representation of Transformations

.NET Framework (current version)

An *m*×*n* matrix is a set of numbers arranged in *m* rows and *n* columns. The following illustration shows several matrices.

$$\begin{bmatrix} 3 & 1 & 4 \\ 2 & 5 & 0 \end{bmatrix}_{2\ 3} \quad \begin{bmatrix} 1 & 3 \\ 2 & 8 \\ 0 & 4 \\ 5 & 6 \end{bmatrix}_{4\ 2} \quad \begin{bmatrix} 1.6 & 0.2 & 1.0 \end{bmatrix}_{1\ 3}$$

$$\begin{bmatrix} 2 & 0 \\ 0 & 3.5 \end{bmatrix}_{2\ 2} \quad \begin{bmatrix} 5 \\ \\ 3 \end{bmatrix}_{2\ 1} \quad \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 40 & 20 & 1 \end{bmatrix}_{3\ 3}$$

You can add two matrices of the same size by adding individual elements. The following illustration shows two examples of matrix addition.

$$\begin{bmatrix} 5 & 4 \end{bmatrix} + \begin{bmatrix} 20 & 30 \end{bmatrix} = \begin{bmatrix} 25 & 34 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 1 & 3 \end{bmatrix} + \begin{bmatrix} 2 & 4 \\ 1 & 5 \\ 0 & 6 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 1 & 7 \\ 1 & 9 \end{bmatrix}$$

An *m*×*n* matrix can be multiplied by an *n*×*p* matrix, and the result is an *m*×*p* matrix. The number of columns in the first matrix must be the same as the number of rows in the second matrix. For example, a 4×2 matrix can be multiplied by a 2×3 matrix to produce a 4×3 matrix.

Points in the plane and rows and columns of a matrix can be thought of as vectors. For example, (2, 5) is a vector with two components, and (3, 7, 1) is a vector with three components. The dot product of two vectors is defined as follows:

(a, b) • (c, d) = ac + bd

(a, b, c) • (d, e, f) = ad + be + cf

For example, the dot product of (2, 3) and (5, 4) is (2)(5) + (3)(4) = 22. The dot product of (2, 5, 1) and (4, 3, 1) is (2)(4) + (5)(3) + (1)(1) = 24. Note that the dot product of two vectors is a number, not another vector. Also note that you can calculate the dot product only if the two vectors have the same number of components.

Let A(i, j) be the entry in matrix A in the ith row and the jth column. For example A(3, 2) is the entry in matrix A in the 3rd row and the 2nd column. Suppose A, B, and C are matrices, and AB = C. The entries of C are calculated as follows:

C(i, j) = (row i of A) • (column j of B)

The following illustration shows several examples of matrix multiplication.

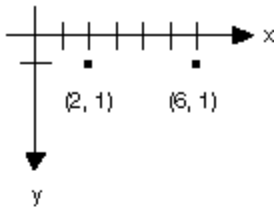
$$\underset{1\times2}{\begin{bmatrix}2 & 3\end{bmatrix}} \quad \underset{2\times1}{\begin{bmatrix}2\\4\end{bmatrix}} = \underset{1\times1}{\begin{bmatrix}16\end{bmatrix}}$$

$$\underset{1\times3}{\begin{bmatrix}1 & 3 & 2\end{bmatrix}} \quad \underset{3\times2}{\begin{bmatrix}1 & 0\\2 & 4\\5 & 1\end{bmatrix}} = \underset{1\times2}{\begin{bmatrix}17 & 14\end{bmatrix}}$$

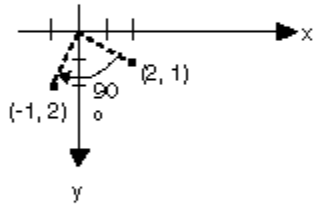
$$\underset{2\times3}{\begin{bmatrix}2 & 5 & 1\\4 & 3 & 1\end{bmatrix}} \quad \underset{3\times3}{\begin{bmatrix}1 & 0 & 0\\0 & 2 & 0\\2 & 3 & 1\end{bmatrix}} = \underset{2\times3}{\begin{bmatrix}4 & 13 & 1\\6 & 9 & 1\end{bmatrix}}$$

If you think of a point in a plane as a 1×2 matrix, you can transform that point by multiplying it by a 2×2 matrix. The following illustration shows several transformations applied to the point (2, 1).

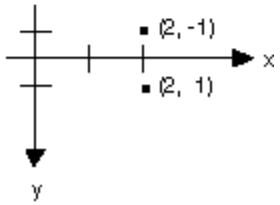
Scale $\begin{bmatrix}2 & 1\end{bmatrix}\begin{bmatrix}3 & 0\\0 & 1\end{bmatrix} = \begin{bmatrix}6 & 1\end{bmatrix}$



Rotate 90 º $\begin{bmatrix}2 & 1\end{bmatrix}\begin{bmatrix}0 & 1\\-1 & 0\end{bmatrix} = \begin{bmatrix}-1 & 2\end{bmatrix}$

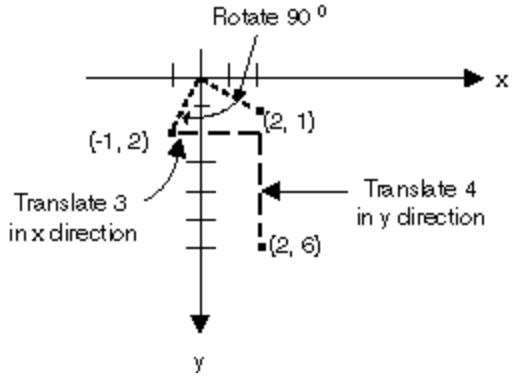


Reflect across x-axis $\begin{bmatrix}2 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix} = \begin{bmatrix}2 & -1\end{bmatrix}$



All of the transformations shown in the preceding figure are linear transformations. Certain other transformations, such as translation, are not linear, and cannot be expressed as multiplication by a 2×2 matrix. Suppose you want to start with the point (2, 1), rotate it 90 degrees, translate it 3 units in the x direction, and translate it 4 units in the y direction. You can accomplish this by using a matrix multiplication followed by a matrix addition.

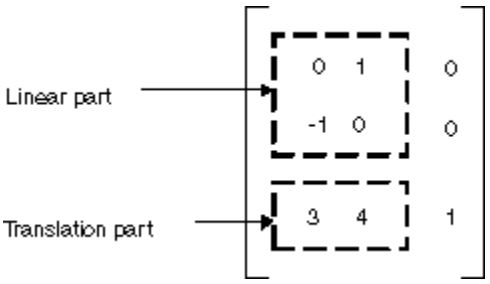
$$\begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} + \begin{bmatrix} 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 6 \end{bmatrix}$$



A linear transformation (multiplication by a 2×2 matrix) followed by a translation (addition of a 1×2 matrix) is called an affine transformation. An alternative to storing an affine transformation in a pair of matrices (one for the linear part and one for the translation) is to store the entire transformation in a 3×3 matrix. To make this work, a point in the plane must be stored in a 1×3 matrix with a dummy 3rd coordinate. The usual technique is to make all 3rd coordinates equal to 1. For example, the point (2, 1) is represented by the matrix [2 1 1]. The following illustration shows an affine transformation (rotate 90 degrees; translate 3 units in the x direction, 4 units in the y direction) expressed as multiplication by a single 3×3 matrix.

$$\begin{bmatrix} 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 3 & 4 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 6 & 1 \end{bmatrix}$$

In the preceding example, the point (2, 1) is mapped to the point (2, 6). Note that the third column of the 3×3 matrix contains the numbers 0, 0, 1. This will always be the case for the 3×3 matrix of an affine transformation. The important numbers are the six numbers in columns 1 and 2. The upper-left 2×2 portion of the matrix represents the linear part of the transformation, and the first two entries in the 3rd row represent the translation.



In GDI+ you can store an affine transformation in a Matrix object. Because the third column of a matrix that represents an affine transformation is always (0, 0, 1), you specify only the six numbers in the first two columns when you construct a Matrix object. The statement `Matrix myMatrix = new Matrix(0, 1, -1, 0, 3, 4)` constructs the matrix shown in the preceding figure.

Composite Transformations

A composite transformation is a sequence of transformations, one followed by the other. Consider the matrices and transformations in the following list:

Matrix A	Rotate 90 degrees

Matrix B	Scale by a factor of 2 in the x direction
Matrix C	Translate 3 units in the y direction

If we start with the point (2, 1) — represented by the matrix [2 1 1] — and multiply by A, then B, then C, the point (2, 1) will undergo the three transformations in the order listed.

[2 1 1]ABC = [-2 5 1]

Rather than store the three parts of the composite transformation in three separate matrices, you can multiply A, B, and C together to get a single 3×3 matrix that stores the entire composite transformation. Suppose ABC = D. Then a point multiplied by D gives the same result as a point multiplied by A, then B, then C.

[2 1 1]D = [-2 5 1]

The following illustration shows the matrices A, B, C, and D.

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -2 & 0 & 0 \\ 0 & 3 & 1 \end{bmatrix}$$

A B C = D

The fact that the matrix of a composite transformation can be formed by multiplying the individual transformation matrices means that any sequence of affine transformations can be stored in a single Matrix object.

Caution

The order of a composite transformation is important. In general, rotate, then scale, then translate is not the same as scale, then rotate, then translate. Similarly, the order of matrix multiplication is important. In general, ABC is not the same as BAC.

The Matrix class provides several methods for building a composite transformation: Multiply, Rotate, RotateAt, Scale, Shear, and Translate. The following example creates the matrix of a composite transformation that first rotates 30 degrees, then scales by a factor of 2 in the y direction, and then translates 5 units in the x direction:

VB

```
Dim myMatrix As New Matrix()
myMatrix.Rotate(30)
myMatrix.Scale(1, 2, MatrixOrder.Append)
myMatrix.Translate(5, 0, MatrixOrder.Append)
```

The following illustration shows the matrix.

$$\begin{bmatrix} \cos 30° & 2\sin 30° & 0 \\ -\sin 30° & 2\cos 30° & 0 \\ 5 & 0 & 1 \end{bmatrix} \approx \begin{bmatrix} 0.866 & 1.0 & 0 \\ -0.5 & 1.73 & 0 \\ 5 & 0 & 1 \end{bmatrix}$$

See Also

Coordinate Systems and Transformations
Using Transformations in Managed GDI+

Global and Local Transformations

.NET Framework (current version)

A global transformation is a transformation that applies to every item drawn by a given Graphics object. In contrast, a local transformation is a transformation that applies to a specific item to be drawn.

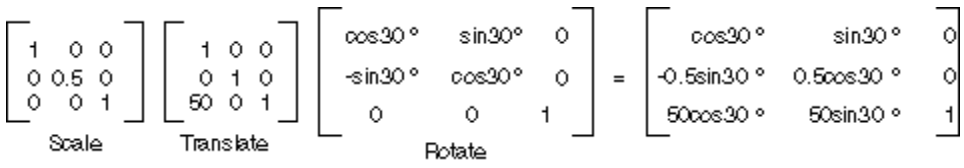
Global Transformations

To create a global transformation, construct a Graphics object, and then manipulate its Transform property. The Transform property is a Matrix object, so it can hold any sequence of affine transformations. The transformation stored in the Transform property is called the world transformation. The Graphics class provides several methods for building up a composite world transformation: MultiplyTransform, RotateTransform, ScaleTransform, and TranslateTransform. The following example draws an ellipse twice: once before creating a world transformation and once after. The transformation first scales by a factor of 0.5 in the y direction, then translates 50 units in the x direction, and then rotates 30 degrees.

VB

```
myGraphics.DrawEllipse(myPen, 0, 0, 100, 50)
myGraphics.ScaleTransform(1, 0.5F)
myGraphics.TranslateTransform(50, 0, MatrixOrder.Append)
myGraphics.RotateTransform(30, MatrixOrder.Append)
myGraphics.DrawEllipse(myPen, 0, 0, 100, 50)
```

The following illustration shows the matrices involved in the transformation.



Note

In the preceding example, the ellipse is rotated about the origin of the coordinate system, which is at the upper-left corner of the client area. This produces a different result than rotating the ellipse about its own center.

Local Transformations

A local transformation applies to a specific item to be drawn. For example, a GraphicsPath object has a Transform method that allows you to transform the data points of that path. The following example draws a rectangle with no transformation and a path with a rotation transformation. (Assume that there is no world transformation.)

VB

```
Dim myMatrix As New Matrix()
myMatrix.Rotate(45)
myGraphicsPath.Transform(myMatrix)
myGraphics.DrawRectangle(myPen, 10, 10, 100, 50)
myGraphics.DrawPath(myPen, myGraphicsPath)
```

You can combine the world transformation with local transformations to achieve a variety of results. For example, you can use the world transformation to revise the coordinate system and use local transformations to rotate and scale objects drawn on the new coordinate system.

Suppose you want a coordinate system that has its origin 200 pixels from the left edge of the client area and 150 pixels from the top of the client area. Furthermore, assume that you want the unit of measure to be the pixel, with the x-axis pointing to the right and the y-axis pointing up. The default coordinate system has the y-axis pointing down, so you need to perform a reflection across the horizontal axis. The following illustration shows the matrix of such a reflection.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Next, assume you need to perform a translation 200 units to the right and 150 units down.

The following example establishes the coordinate system just described by setting the world transformation of a Graphics object.

VB

```
Dim myMatrix As New Matrix(1, 0, 0, -1, 0, 0)
myGraphics.Transform = myMatrix
myGraphics.TranslateTransform(200, 150, MatrixOrder.Append)
```

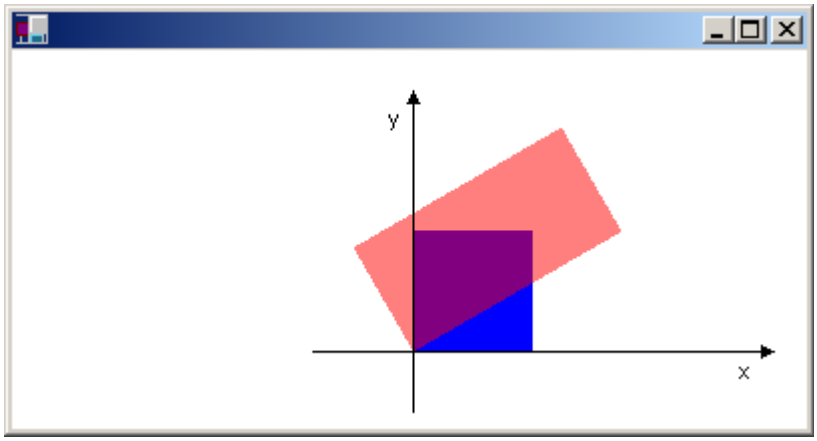
The following code (placed at the end of the preceding example) creates a path that consists of a single rectangle with its lower-left corner at the origin of the new coordinate system. The rectangle is filled once with no local transformation and once with a local transformation. The local transformation consists of a horizontal scaling by a factor of 2 followed by a 30-degree rotation.

VB

```
' Create the path.
Dim myGraphicsPath As New GraphicsPath()
Dim myRectangle As New Rectangle(0, 0, 60, 60)
myGraphicsPath.AddRectangle(myRectangle)

' Fill the path on the new coordinate system.
' No local transformation
myGraphics.FillPath(mySolidBrush1, myGraphicsPath)

' Set the local transformation of the GraphicsPath object.
Dim myPathMatrix As New Matrix()
myPathMatrix.Scale(2, 1)
myPathMatrix.Rotate(30, MatrixOrder.Append)
```

```
    myGraphicsPath.Transform(myPathMatrix)

    ' Fill the transformed path on the new coordinate system.
    myGraphics.FillPath(mySolidBrush2, myGraphicsPath)
```

The following illustration shows the new coordinate system and the two rectangles.



See Also

Coordinate Systems and Transformations
Using Transformations in Managed GDI+