Introduction to GDIPlusX

Doug Hennig
Stonefield Software Inc.
2323 Broad Street
Regina, SK Canada S4P 1Y9
Email: dhennig@stonefield.com
Web sites: www.stonefield.com
www.stonefieldquery.com
Blog: DougHennig.BlogSpot.com
Twitter: DougHennig

GDIPlusX is a VFPX project that exposes GDI+ to VFP applications as a set of VFP class libraries. GDIPlusX makes it easy to add new graphical abilities to your applications, allowing you to provide a fresher and more powerful user interface. This session provides an introduction to GDIPlusX, going through the basics of GDI+, looking at the classes in GDIPlusX, and going through various samples of how to use GDIPlusX in VFP applications.

Introduction

GDI+ is the part of Windows responsible for displaying information on devices such as screens and printers. As its name suggests, GDI+ is the successor to Graphics Device Interface (GDI), the graphics application programming interface (API) included with earlier versions of Windows.

GDI+ allows developers to display information on a display device without worrying about the details of the device. The developer calls methods of the GDI+ classes and those methods in turn make the appropriate calls to specific device drivers. GDI+ insulates the application from the graphics hardware, and this insulation allows developers to create device-independent applications.

Although the GDI+ API is well-documented, calling it from VFP applications can be quite complex, as you have to keep track of things such as handles for every object you use. Also, the API consists of function calls, whereas most VFP developers prefer to work with objects. Fortunately, three expert VFP developers, Craig Boyd, Bo Durban, and Cesar Chalom, spent a lot of time and hard work to create a set of wrapper classes for the GDI+ classes. This set is called GDIPlusX.

One of the most interesting things about GDIPlusX is that it was designed to as closely match as possible the GDI+ wrapper classes the .Net framework provides. The class, property, and method names of GDIPlusX mirror those in the System.Drawing namespace in .Net. The advantage of this is that VFP developers can use the thousands of examples of GDI+ code available for .Net in VFP with very little translation required.

GDIPlusX is a project of VFPX (http://vfpx.codeplex.com), the collaborative community site for VFP add-ons and extensions. You can download GDIPlusX from its page on VFPX and review the many sample and reference articles available. The sample code accompanying this document includes the latest version of GDIPlusX (as of the time of writing).

Using GDIPlusX in your applications

Although you can add the PRGs containing the classes that make up GDIPlusX to your project and include them in your executable, GDIPlusX is also provided as a compiled APP, System.APP, you can simply distribute with your application. Somewhere in your application, DO System.APP. That adds a new member to _SCREEN, System, which is an instance of the xfcSystem class, defined in System.PRG. From there, you simply reference _SCREEN.System and VFP provides full IntelliSense on its members.

GDIPlusX concepts

GDIPlusX provides an object-oriented wrapper for the classes in GDI+. Like the .Net classes, GDIPlusX is organized into a set of “namespaces.” The concept of a namespace doesn’t really exist in VFP, but think of them as a containership hierarchy. At the top of the hierarchy is the System class. It doesn't have many properties or methods itself, but its three main contained classes, Drawing, Imaging, and Text do.

GDIPlusX methods are quite flexible in terms of the parameters they accept. For example, one way to draw a rectangle is to pass the DrawRectangle method of a Graphics object the location and size of the rectangle using four values: the X and Y location of the upper left corner of the rectangle and the width and height:

```
loGfx.DrawRectangle(loPen, 20, 20, 420, 220)
```

However, we can also create a GDIPlusX Rectangle object with the same location and size and pass it to DrawRectangle instead:

```
loRect = _screen.System.Drawing.Rectangle.New(20, 20, 420, 220)
loGfx.DrawRectangle(loPen, loRect)
```

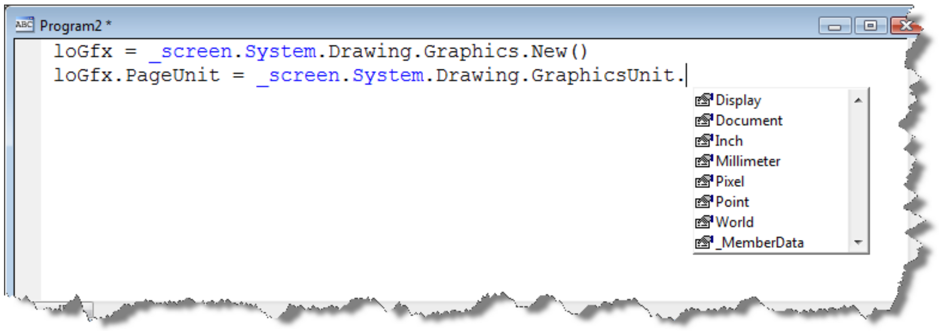
Some method parameters must be objects. For example, in the two DrawRectangle statements, the first parameter is a reference to a Pen object which tells DrawRectangle how wide to draw and with which color. Colors are specified using a GDIPlusX Color object, either one of the members of the Color class (such as System.Drawing.Color.Red) or by passing the desired colors to one of the methods of the Color class, such as FromRGB:

```
loColor = _screen.System.Drawing.Color.FromRGB(rgb(100, 150, 175))
```

Not all parameters are required in every method. Default values, such as 0 for X and Y values, are used in place of missing parameters.

Many numeric properties of VFP objects must be one of a small range of fixed values. For example, Form.BorderStyle can only be 0 (no border), 1 (fixed single), 2 (fixed dialog), or 3 (sizable). Rather than having to remember that 3 means "sizable," the Properties window shows you a drop-down list of the possible values and their meanings. You can think of these values as being "enumerations." VFP doesn't have the concept of enumerations, but .Net does, so rather than having to remember the proper values for each property, you set the property to one of the possible enumerations. Fortunately, the creators of GDIPlusX provide us with something similar: they've exposed members of some classes as if they were enumerations.

For example, the PageUnit property of the Graphics object determines whether drawing units are in pixels, inches, millimeters, and so on. Rather than having to remember that 4 means inches, you can use the Inch member of System.Drawing.GraphicsUnit, which contains 4. IntelliSense makes it seem like VFP supports enumerations, as you can see in **Figure 1**.



```
loGfx = _screen.System.Drawing.Graphics.New()
loGfx.PageUnit = _screen.System.Drawing.GraphicsUnit.
```

Figure 1. GDIPlusX has enumerations that make setting the values of properties much easier.

The Graphics object

All drawing in GDI+ is done with a Graphics object. In GDIPlusX, this is represented by the xfcGraphics class in System.Drawing.PRG.

There are several ways you can obtain a Graphics object:

- From a window handle: pass System.Drawing.Graphics.FromHWnd the handle for the window, such as _SCREEN.HWnd to draw to the screen or MyForm.HWnd to draw directly onto a form.
- From a device context handle: pass System.Drawing.Graphics.FromHDC the device context handle to draw to that device. This isn't commonly used with GDIPlusX.
- From an image: pass System.Drawing.Graphics.FromImage a reference to a previously created GDIPlusX image object.
- From nothing: call System.Drawing.Graphics.New to obtain a Graphics object that isn't connected to anything. This obviously isn't terribly useful unless you can later connect the object to a drawing surface. One use of this is in a ReportListener: you can set the Handle property of the Graphics object to the content of the GDIPlusGraphics property of the ReportListener to allow the Graphics object to draw on the same image as the ReportListener does.

We'll see lots of examples of obtaining and using GDIPlusX Graphics objects in this document.

Drawing units

The position of things you draw in GDI+ is determined by X and Y coordinates, where X is the distance to the right of the left edge and Y is the distance down from the top edge of the drawing surface. By default, GDI+ uses pixels as the units for coordinates, but since it's a device independent graphics system, it can also use other units, such as millimeter, inches, and points (1/72 of an inch). Unlike pixels, which are dependent on the resolution of the device, drawings made with these other units will be the same size on different devices, such as the screen and a printer. To change the units used for drawing, set the PageUnit property of the Graphics object to an enumeration of System.Drawing.GraphicsUnit.

The following code, taken from Units.PRG, shows how to draw using different units. Although we haven't discussed pens, rectangles, or how to draw yet, the code should be easy to follow. The results are shown in **Figure 2**.

```
local loGFX as xfcGraphics of Source\System.Drawing.PRG, ;
   loPen as xfcPen of Source\System.Drawing.PRG

* Ensure GDIPlusX is available.

do System.app
clear
with _screen.System.Drawing as xfcDrawing of Source\System.Drawing.PRG

* Create a Graphics object that draws on _SCREEN.

   loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Create a red pen. The New method of System.Drawing.Pen creates a pen using a
* specified color and width. Note that the color is specified as a color
* brush. In this case, we'll use the Red member of System.Drawing.Color.

   loPen = .Pen.New(.Color.Red, 6)

* Draw a rectangle using the pen. The other parameters for DrawRectangle
* specify the X and Y location of the upper-left corner and its width and
* height.

   loGfx.DrawRectangle(loPen, 10, 10, 200, 100)

* Change the drawing units to inches, create another pen, and draw a 4" by 1"
* rectangle.

   loGfx.PageUnit = .GraphicsUnit.Inch
   loPen = .Pen.New(.Color.Blue, 0.1)
   loGfx.DrawRectangle(loPen, 0.1, 1.3, 4.0, 1.0)

* Draw an 80 x 60 mm rectangle.

   loGfx.PageUnit = .GraphicsUnit.Millimeter
   loPen = .Pen.New(.Color.Green, 1.0)
   loGfx.DrawRectangle(loPen, 4.0, 63.0, 80.0, 60.0)
endwith
```

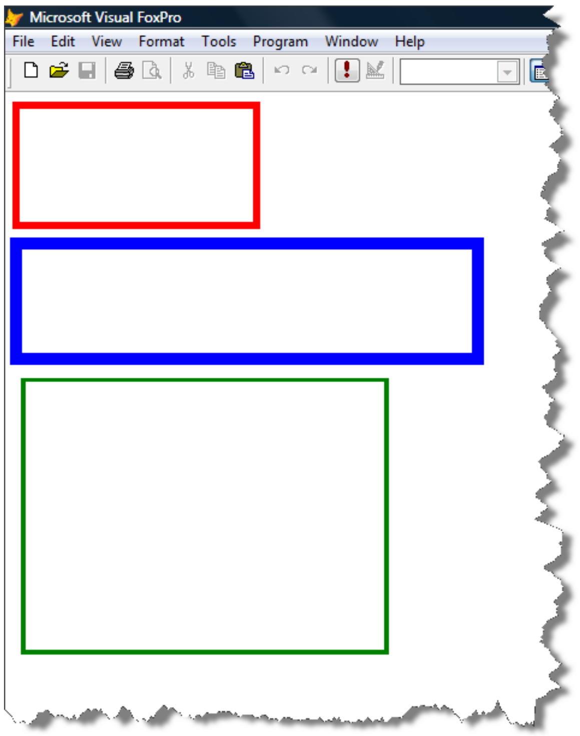


Figure 2. You can specify what units are used for values in GDIPlusX.

Points, sizes, and rectangles

GDIPlusX provides several classes representing a location (a **point**), which has X and Y components, a **size**, which has height and width components, and a **rectangle**, which has both location and size components.

Points

A coordinate is represented in GDIPlusX by an instance of either a Point or PointF object (xfcPoint and xfcPointF classes, defined in System.Drawing.PRG). Both of these objects have X and Y properties. The difference between Point and PointF is that Point stores values as integers while PointF stores floating point values.

Many GDIPlusX methods can accept either X and Y values or Point/PointF objects. For example, the following code, taken from PointSizeRectangle.PRG, draws two lines on the screen, the first using X and Y values and the second a Point object:

```
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object that draws on _SCREEN.

    loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Draw a line using X and Y values.
```

```
   loGfx.DrawLine(.Pens.Black, 10, 10, 200, 200)

* Draw a line using Point objects.

   loPoint1 = .Point.New(10, 20)
   loPoint2 = .Point.New(200, 210)
   loGfx.DrawLine(.Pens.Black, loPoint1, loPoint2)
endwith
```

Sizes

A size is an object specifying the height and width of something. Like points, there are two size objects in GDIPlusX, Size and SizeF (xfcSize and xfcSizeF in System.Drawing.PRG), both of which have Height and Width properties. We'll see the use of sizes later.

Rectangles

You can specify the location and size of a drawing object using a Rectangle or RectangleF object (xfcRectangle and xfcRectangleF in System.Drawing.PRG). To create a rectangle, call System.Drawing.Rectangle.New or System.Drawing.RectangleF.New, passing the X and Y values for the upper-left corner of the rectangle, followed by the width and height. The values are written to the X, Y, Width, and Height properties, respectively, of the Rectangle/RectangleF object. Rectangles have other useful properties as well: Location, which is a Point or PointF object containing the X and Y values; Size, a Size or SizeF object containing the Height and Width values; and Left, Right, Top, and Bottom, which contain the appropriate values for the rectangle.

The following code, which was taken from PointSizeRectangle.PRG, shows how to create and use a rectangle:

```
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object that draws on _SCREEN.

   loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Draw a red rectangle using X, Y, height, and width values.

   loGfx.DrawRectangle(.Pens.Red, 20, 20, 420, 220)

* Draw a blue rectangle using a Rectangle object.

   loRect = .Rectangle.New(40, 40, 100, 100)
   loGfx.DrawRectangle(.Pens.Blue, loRect)
endwith
```

Rectangles have some useful methods. Contains returns .T. if the specified rectangle (specified as X, Y, width, and height values or another rectangle object) is enclosed in the current rectangle. Union returns a new rectangle large enough to encompass the two specified rectangles. Intersect replaces the rectangle with the intersection of itself and the specified rectangle. IntersectsWith returns .T. if the rectangle intersects with the specified rectangle. Code in PointSizeRectangle.PRG shows some of these methods:

```
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a rectangle.

   loRect = .Rectangle.New(40, 40, 100, 100)

* Create a new rectangle and see if it's contained with the first one.

   loRect2 = .Rectangle.New(60, 60, 10, 10)
   messagebox('loRect2 ' + iif(loRect.Contains(loRect2), 'is', 'is not') + ;
      ' contained in loRect')

* Create a rectangle that unions two others.

   loRect2 = .Rectangle.New(60, 60, 200, 200)
   loRect3 = loRect2.Union(loRect, loRect2)
   messagebox('Top: ' + transform(loRect3.Top) + chr(13) + ;
      'Left: ' + transform(loRect3.Left) + chr(13) + ;
      'Bottom: ' + transform(loRect3.Bottom) + chr(13) + ;
      'Right: ' + transform(loRect3.Right), 0, 'After Union')

* See if the rectangles intersect.

   messagebox('loRect and loRect2 ' + ;
      iif(loRect.IntersectsWith(loRect2), 'do', 'do not') + ' intersect')
endwith
```

Stroking and filling

Drawing a line or shape on a Graphics object is called *stroking*. A pen object specifies how a line or shape is drawn: the color of the line, the width of the line, and so on. *Filling* means filling an enclosed shape, such as a rectangle or ellipse, with a color, pattern, or image. A brush object specifies how to fill a shape.

The Graphics object has methods for drawing a filling a variety of lines and shapes. The Draw methods expect the first parameter is a pen object and the Fill methods expect a brush object; we'll discuss pens and brushes later in this document.

- **DrawLine(oPen, oPoint, oPoint)** or **DrawLine(oPen, X1, Y1, X2, Y2)**: draws a line between the two points.
- **DrawRectangle(oPen, X, Y, Width, Height)** or **DrawRectangle(oPen, oRectangle)**: draws a rectangle, starting from the specified upper-left corner, of the desired height and width.
- **FillRectangle(oBrush, X, Y, Width, Height)** or **FillRectangle(oPen, oRectangle)**: fills a rectangle using the specified brush.

- **DrawEllipse(oPen, X, Y, Width, Height)** or **DrawEllipse(oPen, oRectangle)**: draws a ellipse bounded by the specified rectangle.
- **FillEllipse(oBrush, X, Y, Width, Height)** or **FillEllipse(oPen, oRectangle)**: fills an ellipse.
- **DrawPolygon(oPen, aPoints)**: draws a polygonal shape. The 2-dimensional array contains the X and Y coordinates of the corner points; DrawPolygon draws lines between the points.
- **FillPolygon(oBrush, aPoints)**: fills a polygon.
- **DrawCurve(oPen, aPoints)**: draws a type of curve called a "cardinal spline;" see http://www.bobpowell.net/cardinalspline.htm for information about and http://msdn.microsoft.com/en-us/library/system.drawing.graphics.drawcurve(VS.80).aspx for details about additional parameters.
- **DrawClosedCurve(oPen, aPoints)**: draws a closed curve; see http://msdn.microsoft.com/en-us/library/system.drawing.graphics.drawclosedcurve(VS.80).aspx for details about additional parameters.
- **DrawBezier(oPen, oPoint1, oPoint2, oPoint2, oPoint4)** or **DrawBezier(oPen, X1, Y1, X2, Y2, X3, Y3, X4, Y4)**: draws a Bezier curve; see http://www.bobpowell.net/bezier.htm for information on Bezier curves.
- **DrawPath(oPen, oPath)**: draws along a path, which is a collection of lines and shapes; see http://www.bobpowell.net/path.htm for information about paths.
- **FillPath(oBrush, oPath)**: fills a path.
- **DrawArc(oPen, oRectangle, nStartAngle, nSweepAngle)** or **DrawArc(oPen, X, Y, Width, Height, nStartAngle, nSweepAngle)**: draws an arc segment, which is a section of an ellipse, bounded by the specified rectangle, starting at the specified start angle and sweeping to the specified end angle.
- **DrawPie(oPen, oRectangle, nStartAngle, nSweepAngle)** or **DrawPie(oPen, X, Y, Width, Height, nStartAngle, nSweepAngle)**: draws a pie-shaped area. Similar to DrawArc, except it also draws two lines, one from each end of the arc to the center of the ellipse specified by the rectangle.
- **FillPie(oBrush, oRectangle, nStartAngle, nSweepAngle)** or **FillPie(oBrush, X, Y, Width, Height, nStartAngle, nSweepAngle)**: fills a pie-shaped area.
- **FillRegion(oBrush, oRegion)**: fills a region; see http://www.bobpowell.net/whatregion.htm for information on regions.

You can combine calls to some of these methods. For example, if you want a red rectangle to have a black border, call FillRectangle with a red pen and DrawRectangle with a black one and the same rectangle dimensions.

Here's an example, taken from Graphics.PRG, which shows several of these methods:

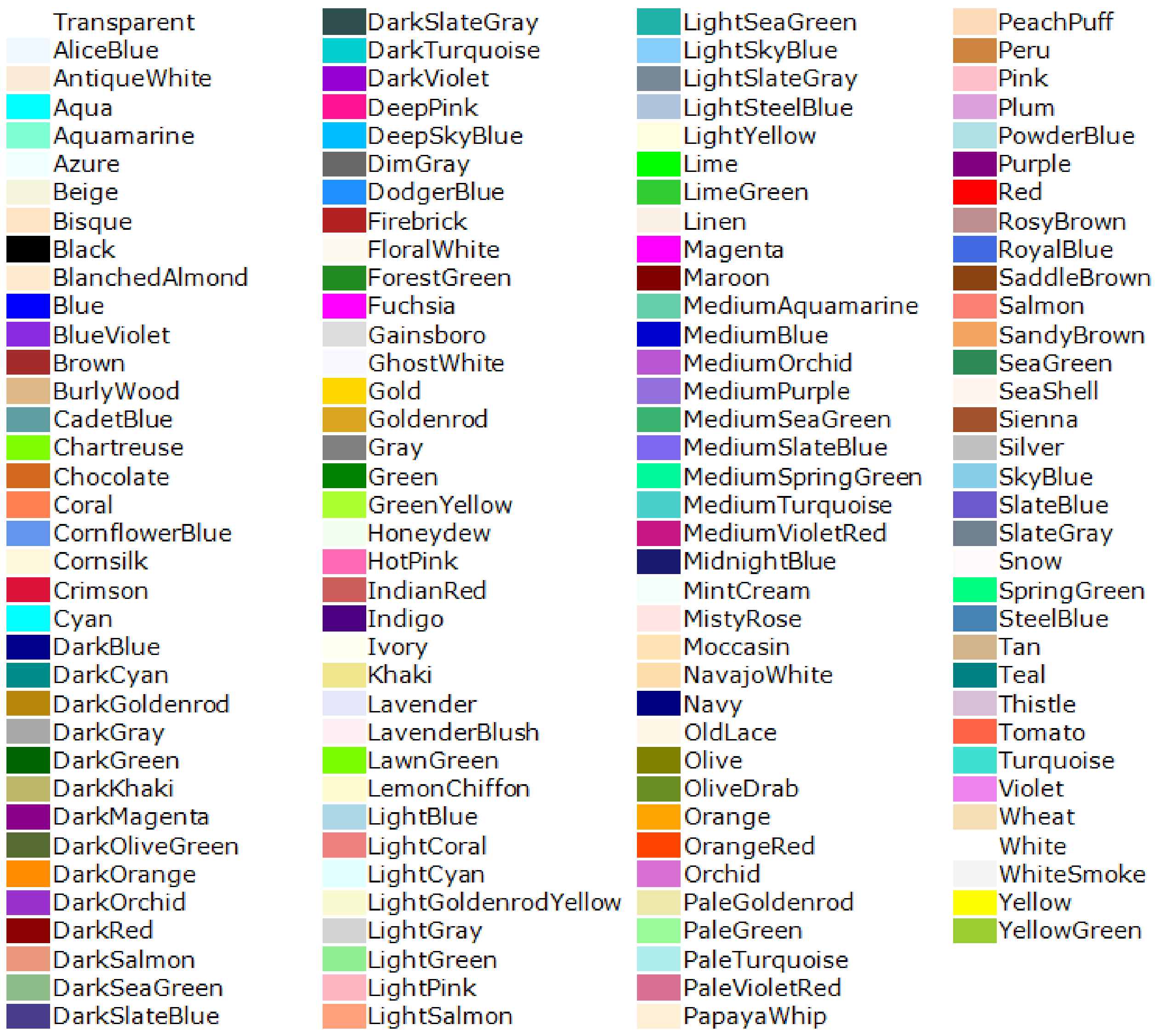
```
local loGFX as xfcGraphics of Source\System.Drawing.PRG, ;
   loPen as xfcPen of Source\System.Drawing.PRG, ;
   laPoints[5]

* Ensure GDIPlusX is available.

do System.app
clear
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object that draws on _SCREEN.

   loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Draw a line. System.Drawing.Pens provides a set of pens with pre-defined
* colors, such as black. The other parameters for DrawLine are the X and Y
* locations of the endpoints of the line.

   loGfx.DrawLine(.Pens.Black, 10, 10, 400, 400)

* Fill an ellipse using a blue brush. System.Drawing.Brushes provides a set of
* brushes with pre-defined colors, such as blue. The other parameters are the X
* and Y location of the upper-left corner of the ellipse and its width and
* height.

   loGfx.FillEllipse(.Brushes.Blue, 30, 30, 400, 200)

* Create a red pen. The New method of System.Drawing.Pen creates a pen using a
* specified color and width. Note that the color is specified as a color
* brush. In this case, we'll use the Red member of System.Drawing.Color.

   loPen = .Pen.New(.Color.Red, 2)

* Draw a rectangle using the pen. As with ellipses, the parameters specify the
* X and Y location of the upper-left corner and its width and height.

   loGfx.DrawRectangle(loPen, 20, 20, 420, 220)

* Create a closed polygon, fill it with tomato red, and stroke it with a
* 4-pixel tan pen.

   laPoints[1] = .Point.New(220, 250)
   laPoints[2] = .Point.New(400, 300)
   laPoints[3] = .Point.New(400, 250)
   laPoints[4] = .Point.New(220, 300)
   laPoints[5] = .Point.New(220, 250)
   loGfx.FillPolygon(.Brushes.Tomato, @laPoints)
   loGfx.DrawPolygon(.Pen.New(.Color.Tan, 4), @laPoints)
endwith
```

Working with color

VFP developers are usually familiar with color settings. The native RGB() function returns a numeric color value from three specified integers (0 – 255) representing the red, green, and blue components of the color. GDI+ adds an additional layer: the transparency of the color, also known as the alpha value or alpha channel. Alpha is also specified using an integer from 0 – 255, where 0 is completely transparent and 255 is opaque.

GDIPlusX has a Color object (xfcColor in System.Drawing.PRG) which represents a color. A color object is most commonly used as a parameter for creating brushes and pens. Its R, G, B, and A properties contain the red, green, blue, and alpha values of the color, and ARGB contains the combined values. There are several ways you can create a color object:

- Like the RGB() function, the FromRGB method of System.Drawing.Color accepts red, green, and blue values, but returns a color object of the specified color.
- FromARGB accepts alpha, red, green, and blue values and returns a color object. Alternatively, it can accept an alpha value and a color object for the red, green, and blue values.
- FromName accepts a known color name as a string, such as “red,” “green,” or “black.” The known names and their colors are shown in **Figure 3**, which was taken from http://www.bobpowell.net/colour.htm.
- Color contains members with the known names, such as System.Drawing.Color.Firebrick and System.Drawing.Color.Gold, which return color objects with those colors.
- FromKnownColor accepts a numeric value representing either a known color or a system color (such as the color for the border of active windows), and returns the appropriate color object. You don’t have to know the numeric values for the colors; instead, use one of the enumerations of System.Drawing.KnownColor, such as System.Drawing.KnownColor.ActiveBorder or System.Drawing.KnownColor.Green.



Transparent
AliceBlue
AntiqueWhite
Aqua
Aquamarine
Azure
Beige
Bisque
Black
BlanchedAlmond
Blue
BlueViolet
Brown
BurlyWood
CadetBlue
Chartreuse
Chocolate
Coral
CornflowerBlue
Cornsilk
Crimson
Cyan
DarkBlue
DarkCyan
DarkGoldenrod
DarkGray
DarkGreen
DarkKhaki
DarkMagenta
DarkOliveGreen
DarkOrange
DarkOrchid
DarkRed
DarkSalmon
DarkSeaGreen
DarkSlateBlue
DarkSlateGray
DarkTurquoise
DarkViolet
DeepPink
DeepSkyBlue
DimGray
DodgerBlue
Firebrick
FloralWhite
ForestGreen
Fuchsia
Gainsboro
GhostWhite
Gold
Goldenrod
Gray
Green
GreenYellow
Honeydew
HotPink
IndianRed
Indigo
Ivory
Khaki
Lavender
LavenderBlush
LawnGreen
LemonChiffon
LightBlue
LightCoral
LightCyan
LightGoldenrodYellow
LightGray
LightGreen
LightPink
LightSalmon
LightSeaGreen
LightSkyBlue
LightSlateGray
LightSteelBlue
LightYellow
Lime
LimeGreen
Linen
Magenta
Maroon
MediumAquamarine
MediumBlue
MediumOrchid
MediumPurple
MediumSeaGreen
MediumSlateBlue
MediumSpringGreen
MediumTurquoise
MediumVioletRed
MidnightBlue
MintCream
MistyRose
Moccasin
NavajoWhite
Navy
OldLace
Olive
OliveDrab
Orange
OrangeRed
Orchid
PaleGoldenrod
PaleGreen
PaleTurquoise
PaleVioletRed
PapayaWhip
PeachPuff
Peru
Pink
Plum
PowderBlue
Purple
Red
RosyBrown
RoyalBlue
SaddleBrown
Salmon
SandyBrown
SeaGreen
SeaShell
Sienna
Silver
SkyBlue
SlateBlue
SlateGray
Snow
SpringGreen
SteelBlue
Tan
Teal
Thistle
Tomato
Turquoise
Violet
Wheat
White
WhiteSmoke
Yellow
YellowGreen

Figure 3. Known color names and their colors.

In addition to the R, G, B, and A properties, Color has several other properties you may find useful. Name contains the known name of the color if such a name was used to create the color object. IsKnownColor is .T. if the color is known. IsSystemColor is .T. if the color is a system color. The Equals method returns .T. if the specified color object is the same color as the current object.

Color.PRG starts by drawing ellipses and rectangles in random colors at random places on the screen (see **Figure 4**), then draws a color chart similar to that in **Figure 3**.

```
local loGFX as xfcGraphics of Source\System.Drawing.PRG, ;
   loBrush as xfcBrush of Source\System.Drawing.PRG

* Ensure GDIPlusX is available.

do System.app
```

```
clear
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object that draws on _SCREEN.

   loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Seed the random number generator.

   rand(-1)

* Draw filled ellipses and rectangles of random colors and transparencies at
* random places on the screen.

   for lnI = 1 to 100
      loBrush = .SolidBrush.New(.Color.FromARGB(rand() * 255, rand() * 255, ;
         rand() * 255, rand() * 255))
      if mod(int(rand() * 100), 2) = 1
         loGfx.FillRectangle(loBrush, rand() * _screen.Width, ;
            rand() * _screen.Height, rand() * 200, rand() * 200)
      else
         loGfx.FillEllipse(loBrush, rand() * _screen.Width, ;
            rand() * _screen.Height, rand() * 200, rand() * 200)
      endif mod(int(rand() * 100), 2) = 1
   next lnI

* Wait for a keypress, then clear the screen.

   wait window
   clear

* Create a color chart. Start by creating a font object and starting X and Y
* values.

   loFont = .Font.New('Segoe UI', 10)
   lnX    = 0
   lnY    = 20
   for lnI = 1 to 174  && there are 174 known colors

* Create a color object from the known color number and get its name.

      loColor = .Color.FromKnownColor(lnI)
      lcName  = loColor.Name

* Create a brush of the specified color and create a small filled rectangle.

      loBrush = .SolidBrush.New(loColor)
      loGfx.FillRectangle(loBrush, lnX, lnY, 10, 10)

* Draw the name of the color beside the rectangle.

      loGfx.DrawString(lcName, loFont, .Brushes.Black, lnX + 10, lnY - 4)

* Increment the Y position and every 20 colors, move to the next column.
```

```
      if lnI/20 = int(lnI/20)
         lnX = lnX + 150
         lnY = 20
      else
         lnY = lnY + 20
      endif lnI/20 = int(lnI/20)
   next lnI
endwith
```

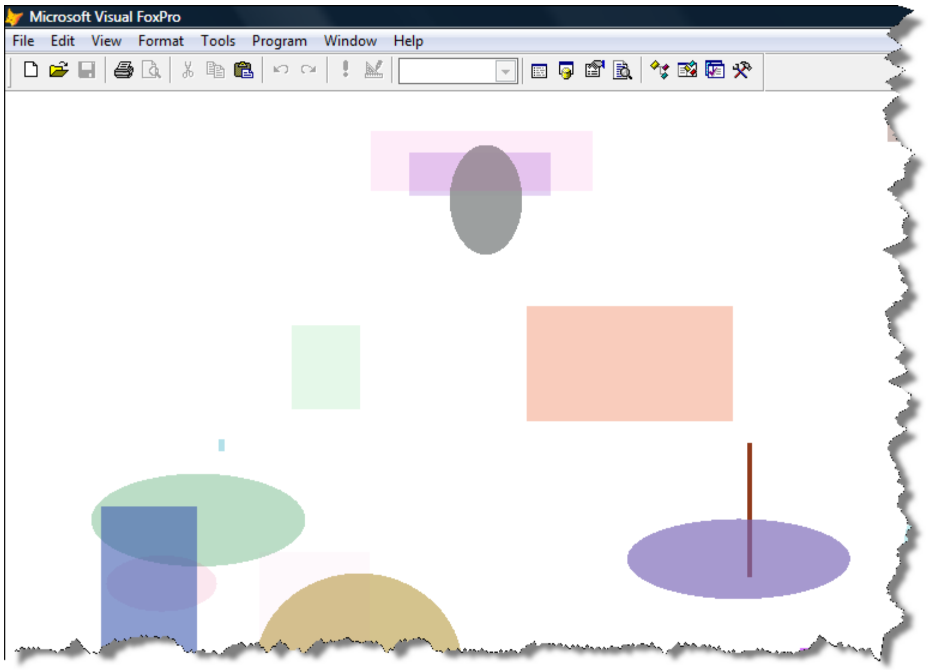


Figure 4. Color.PRG shows how to use different colors to draw objects.

Pens

Pens are used to stroke lines and shapes. Pens have several attributes that control how they are used, such as width, style, color, and end caps. You can even create a compound pen, one that has sections that draw and others that don't; that isn't covered in this document, but you can see http://www.bobpowell.net/pensone.htm for details.

Standard pens

As we've seen earlier, System.Drawing contains a set of standard pens, one for each system color. They are referenced using System.Drawing.Pens.*ColorName*, where *ColorName* is the name of the desired color. For example, PointSizeRectangle.PRG has a statement that draws a line using a black pen:

```
loGfx.DrawLine(.Pens.Black, 10, 10, 200, 200)
```

The pens are all one unit wide. The type of unit depends on the setting of the Graphics.PageUnit property we saw earlier.

Pen width

If you want to draw using a pen wider than one unit, create a new pen using System.Drawing.Pen.New, specifying a color and pen width. For example, code we saw earlier in Graphics.PRG created two pens, one (a 2-pixel red pen) which was stored in a variable and the other (a 4-pixel tan pen) passed directly to a draw method:

```
loPen = .Pen.New(.Color.Red, 2)
loGfx.DrawPolygon(.Pen.New(.Color.Tan, 4), @laPoints)
```

End caps

The StartCap and EndCap properties determine what shapes are drawn at the ends of lines drawn by the pen. Set the values of these properties to enumerations of System.Drawing.Drawing2D.LineCap, such as System.Drawing.Drawing2D.LineCap.Round. StartCap and EndCap can be the same or different. The default value for both properties is System.Drawing.Drawing2D.LineCap.Flat.

The following code, taken from Pens.PRG, creates the lines shown in **Figure 5**. Note that all lines are specified as the same length yet some appear longer. The end caps are added to the ends of the lines, so larger ones, such as the various anchor caps, extend beyond the line.

```
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object that draws on _SCREEN.

   loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Draw some reference lines.

   loGfx.DrawLine(.Pens.Black, 100, 0, 100, 600)
   loGfx.DrawLine(.Pens.Black, 500, 0, 500, 600)

* Create a 20-pixel red pen.

   loPen = .Pen.New(.Color.Red, 20)

* Draw lines with various caps.

   loPen.StartCap = .Drawing2D.LineCap.Flat
   loPen.EndCap   = .Drawing2D.LineCap.Flat
   loGfx.DrawLine(loPen, 100, 100, 500, 100)

   loPen.StartCap = .Drawing2D.LineCap.Square
   loPen.EndCap   = .Drawing2D.LineCap.Square
   loPen.Color    = .Color.Green
   loGfx.DrawLine(loPen, 100, 130, 500, 130)

   loPen.StartCap = .Drawing2D.LineCap.Round
   loPen.EndCap   = .Drawing2D.LineCap.Round
   loPen.Color    = .Color.Orange
   loGfx.DrawLine(loPen, 100, 160, 500, 160)

   loPen.StartCap = .Drawing2D.LineCap.Triangle
   loPen.EndCap   = .Drawing2D.LineCap.Triangle
   loPen.Color    = .Color.Purple
   loGfx.DrawLine(loPen, 100, 190, 500, 190)

   loPen.StartCap = .Drawing2D.LineCap.SquareAnchor
   loPen.EndCap   = .Drawing2D.LineCap.SquareAnchor
   loPen.Color    = .Color.Teal
   loGfx.DrawLine(loPen, 100, 220, 500, 220)

   loPen.StartCap = .Drawing2D.LineCap.RoundAnchor
   loPen.EndCap   = .Drawing2D.LineCap.RoundAnchor
   loPen.Color    = .Color.Blue
   loGfx.DrawLine(loPen, 100, 260, 500, 260)

   loPen.StartCap = .Drawing2D.LineCap.DiamondAnchor
   loPen.EndCap   = .Drawing2D.LineCap.DiamondAnchor
   loPen.Color    = .Color.Pink
   loGfx.DrawLine(loPen, 100, 300, 500, 300)

   loPen.StartCap = .Drawing2D.LineCap.ArrowAnchor
   loPen.EndCap   = .Drawing2D.LineCap.ArrowAnchor
   loPen.Color    = .Color.Salmon
   loGfx.DrawLine(loPen, 100, 340, 500, 340)

* StartCap and EndCap don't have to be the same.

   loPen.StartCap = .Drawing2D.LineCap.RoundAnchor
   loPen.EndCap   = .Drawing2D.LineCap.ArrowAnchor
   loPen.Color    = .Color.GoldenRod
   loGfx.DrawLine(loPen, 100, 380, 500, 380)
 endwith
```

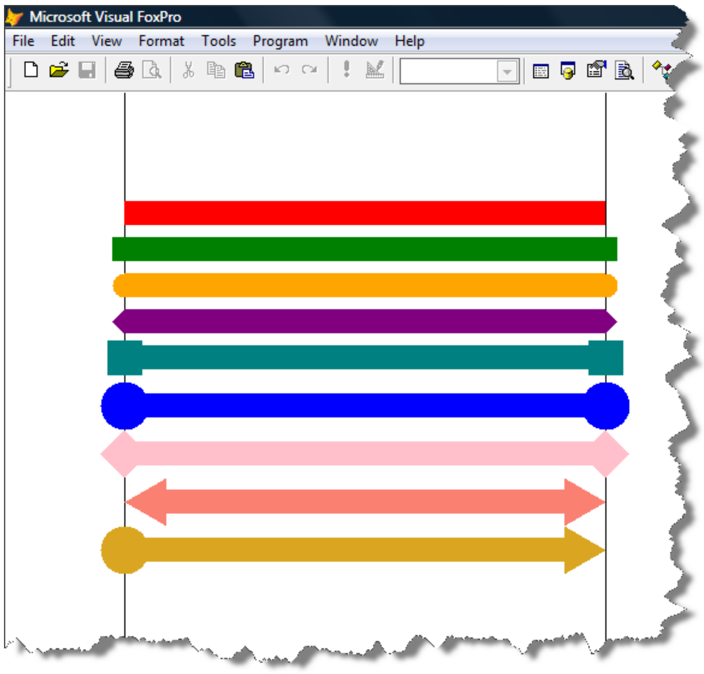


Figure 5. Pens support a variety of end caps.

You can also create your own custom end caps, but that's beyond the scope of this document.

Line style

You can create lines with dots and dashes by setting the DashStyle property to an enumeration of System.Drawing.Drawing2D.DashStyle, such as System.Drawing.Drawing2D.DashStyle.DashDot. In addition, you can specify the end cap used for the dots and dashes by setting the DashCap property to an enumeration of System.Drawing.Drawing2D.DashCap. The following code, taken from PenDashStyle.PRG, creates the drawing shown in **Figure 6**.

```
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object that draws on _SCREEN.

    loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Draw some reference lines.

    loGfx.DrawLine(.Pens.Black, 100, 0, 100, 600)
    loGfx.DrawLine(.Pens.Black, 500, 0, 500, 600)

* Create a black pen.

    loPen = .Pen.New(.Color.Black, 20)

* Draw lines with various styles.
```

```
    loPen.DashStyle = .Drawing2D.DashStyle.Dash
    loGfx.DrawLine(loPen, 100, 100, 500, 100)

    loPen.DashStyle = .Drawing2D.DashStyle.Dot
    loPen.Color     = .Color.Blue
    loGfx.DrawLine(loPen, 100, 130, 500, 130)

    loPen.DashStyle = .Drawing2D.DashStyle.DashDot
    loPen.Color     = .Color.Green
    loGfx.DrawLine(loPen, 100, 160, 500, 160)

    loPen.DashStyle = .Drawing2D.DashStyle.DashDotDot
    loPen.Color     = .Color.Purple
    loGfx.DrawLine(loPen, 100, 190, 500, 190)

* Set the end caps for the dashes.

    loPen.DashStyle = .Drawing2D.DashStyle.Dash
    loPen.DashCap   = .Drawing2D.DashCap.Round
    loPen.Color     = .Color.Teal
    loGfx.DrawLine(loPen, 100, 220, 500, 220)

    loPen.DashCap   = .Drawing2D.DashCap.Triangle
    loPen.Color     = .Color.Salmon
    loGfx.DrawLine(loPen, 100, 250, 500, 250)

* Set the start, end, and dash caps.

    loPen.SetLineCap(.Drawing2D.LineCap.ArrowAnchor, ;
       .Drawing2D.LineCap.RoundAnchor, .Drawing2D.DashCap.Round)
    loPen.Color = .Color.Orange
    loGfx.DrawLine(loPen, 100, 280, 500, 280)
endwith
```

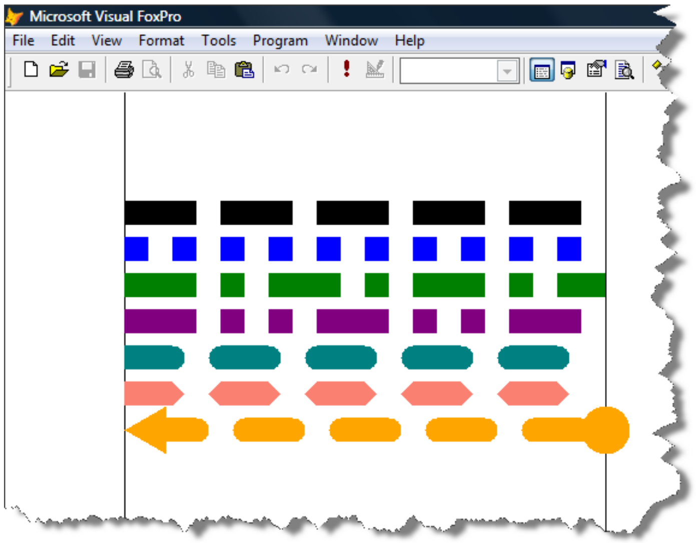


Figure 6. You can style pens with dashes and dots.

Brushes

Brushes fill an enclosed shape, such as a rectangle or ellipse, with a color, pattern, or image. There are several kinds of brushes, including solid (which fill with a solid color), hatch (which fill with a hatch pattern), gradient (which fill with gradient colors), and image (which fill from an image). Although some brushes are in the System.Drawing namespace, others are in System.Drawing.Drawing2D.

Standard brushes

As with pens, System.Drawing contains a set of standard solid brushes. They are referenced using System.Drawing.Brushes.*ColorName*, where *ColorName* is the name of the desired color. For example, PointSizeRectangle.PRG has statements that fill images with blue and tomato red brushes:

```
loGfx.FillEllipse(.Brushes.Blue, 30, 30, 400, 200)
loGfx.FillPolygon(.Brushes.Tomato, @laPoints)
```

Solid brushes

System.Drawing.SolidBrush.New creates a new brush that fills a region with the specified solid color. We saw the use of solid brushes earlier in Colors.PRG:

```
loBrush = .SolidBrush.New(loColor)
loGfx.FillRectangle(loBrush, lnX, lnY, 10, 10)
```

Solid brushes can, of course, fill with solid or transparent colors. The code in SolidBrush.PRG creates the image shown in **Figure 7**.

```
* Draw a red rectangle.

   loBrush = .SolidBrush.New(.Color.Red)
   loGfx.FillRectangle(loBrush, 50, 50, 300, 300)

* Draw a blue ellipse.

   loBrush.Color = .Color.Blue
   loGfx.FillEllipse(loBrush, 0, 100, 400, 100)

* Draw a semi-transparent ellipse.

   loBrush.Color = .Color.FromARGB(80, .Color.Green)
   loGfx.FillEllipse(loBrush, 0, 200, 400, 100)
```

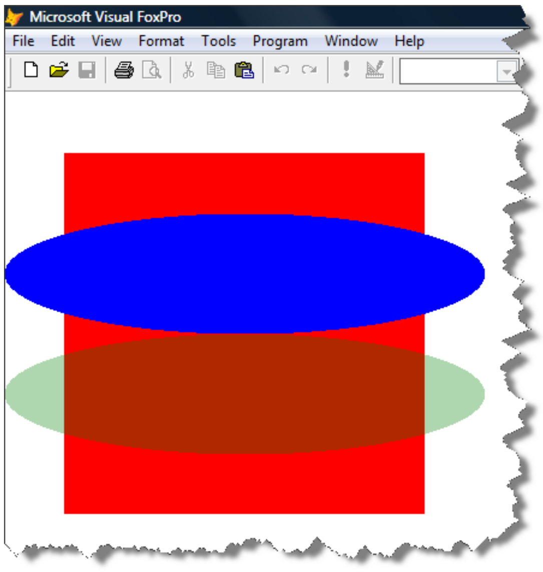


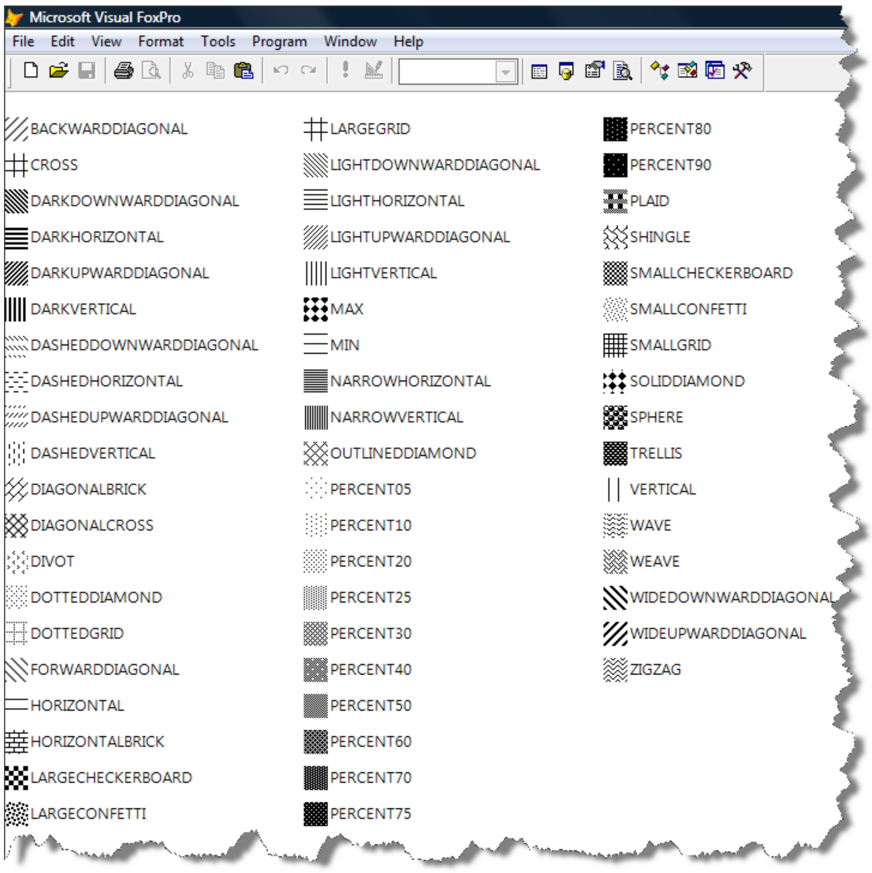
Figure 7. Solid brushes can fill with solid or transparent color.

Hatch brushes

System.Drawing.Drawing2D.HatchBrush.New creates a new brush that fills a region with the specified hatch pattern and color. The hatch pattern is an enumeration of System.Drawing.Drawing2D.HatchStyle, such as System.Drawing.Drawing2D.HatchStyle.Spheres.

The code shown below, taken from HatchBrush.PRG, creates the hatch pattern chart shown in **Figure 8**.

```
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object that draws on _SCREEN.

   loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Create a hatch chart. Start by creating a font object and starting X and Y
* values.

   loFont = .Font.New('Segoe UI', 10)
   lnX    = 0
   lnY    = 20

* Fill an array with the names of the hatch patterns.

   laPatterns = _screen.System.Enum.GetNames(.Drawing2D.HatchStyle)

* Create a listing of patterns.

   for lnI = 1 to alen(laPatterns)

* Create a brush of the specified pattern and create a small filled rectangle.
* For some reason, the "Total" pattern appears in the enums but doesn't work.

      lcName  = laPatterns[lnI]
      if lcName <> 'TOTAL'
          lnStyle = .Drawing2D.HatchStyle.&lcName
          loBrush = .Drawing2D.HatchBrush.New(lnStyle, .Color.Black)
          loGfx.FillRectangle(loBrush, lnX, lnY, 20, 20)

* Draw the name of the color beside the rectangle.

          loGfx.DrawString(lcName, loFont, .Brushes.Black, lnX + 20, lnY)

* Increment the Y position and every 20 patterns, move to the next column.

          if lnI/20 = int(lnI/20)
             lnX = lnX + 250
             lnY = 20
          else
             lnY = lnY + 30
          endif lnI/20 = int(lnI/20)
      endif lcName <> 'TOTAL'
   next lnI
endwith
```



BACKWARDDIAGONAL
CROSS
DARKDOWNWARDDIAGONAL
DARKHORIZONTAL
DARKUPWARDDIAGONAL
DARKVERTICAL
DASHEDDOWNWARDDIAGONAL
DASHEDHORIZONTAL
DASHEDUPWARDDIAGONAL
DASHEDVERTICAL
DIAGONALBRICK
DIAGONALCROSS
DIVOT
DOTTEDDIAMOND
DOTTEDGRID
FORWARDDIAGONAL
HORIZONTAL
HORIZONTALBRICK
LARGECHECKERBOARD
LARGECONFETTI
LARGEGRID
LIGHTDOWNWARDDIAGONAL
LIGHTHORIZONTAL
LIGHTUPWARDDIAGONAL
LIGHTVERTICAL
MAX
MIN
NARROWHORIZONTAL
NARROWVERTICAL
OUTLINEDDIAMOND
PERCENT05
PERCENT10
PERCENT20
PERCENT25
PERCENT30
PERCENT40
PERCENT50
PERCENT60
PERCENT70
PERCENT75
PERCENT80
PERCENT90
PLAID
SHINGLE
SMALLCHECKERBOARD
SMALLCONFETTI
SMALLGRID
SOLIDDIAMOND
SPHERE
TRELLIS
VERTICAL
WAVE
WEAVE
WIDEDOWNWARDDIAGONAL
WIDEUPWARDDIAGONAL
ZIGZAG

Figure 8. There are a lot of hatch patterns available for HatchBrush.

Gradient brushes

A gradient brush is one that changes color as it progresses across the shape it fills. As the brush starts filling, it uses one color, then transitions so by the end of the fill, a second color is used. For example, the form shown in **Figure 11** has a gradient from blue at the top to white at the bottom.

There are two types of gradient brushes: linear and path. A linear gradient brush changes color as it moves horizontally, vertically, or along a line. A path gradient brush changes color as it moves about a path. We won't discuss path brushes in this document; see http://www.functionx.com/vcnet/gdi+/gradientbrush.htm and http://msdn.microsoft.com/en-us/library/7fswd1t7.aspx for information.

To create a simple linear gradient brush, pass the following parameters to System.Drawing.Drawing2D.LinearGradientBrush.New: two points (or a rectangle), the starting color, the ending color, and the transition mode. The first point (or Top and Left properties of the rectangle) specifies the point where the drawing starts and the starting color is applied. The second point (or Bottom and Right properties of the rectangle) specifies the point where the drawing stops and the ending color is applied. The mode determines the direction of the transition and is an enumeration of System.Drawing.Drawing2D.LinearGradientMode: Horizontal, Vertical, ForwardDiagonal, or BackwardDiagonal. The default is horizontal if it isn't specified.

The following code, taken from LinearGradientBrush.PRG, creates the image shown in **Figure 9**. Notice what happens with the second rectangle when the gradient brush is smaller than the rectangle it fills.

```
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object that draws on _SCREEN.

   loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Draw a rectangle with a gradient from red to white.

   loRect  = .Rectangle.New(50, 50, 300, 100)
   loBrush = .Drawing2D.LinearGradientBrush.New(loRect, .Color.Red, ;
      .Color.White)
   loGfx.FillRectangle(loBrush, loRect)

* Change the brush colors and draw a bigger rectangle.

   loBrush.LinearColors[1] = .Color.Blue
   loBrush.LinearColors[2] = .Color.Black
   loGfx.FillRectangle(loBrush, 50, 200, 500, 100)
endwith
```

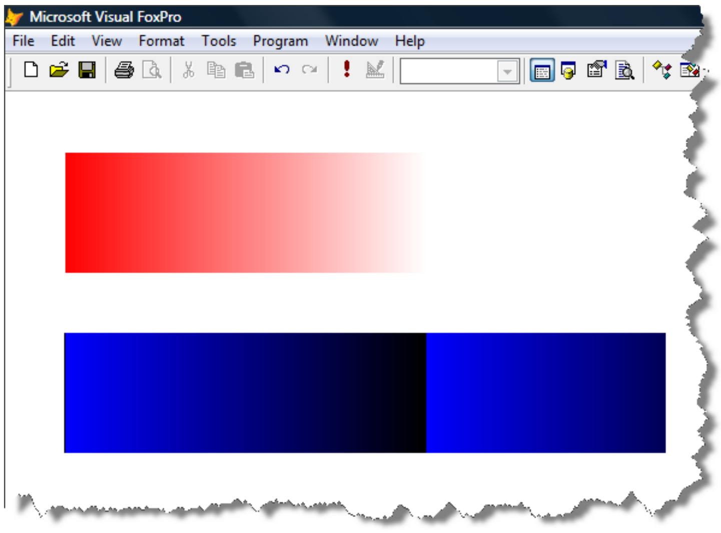


Figure 9. Linear gradient brushes fill regions with gradient colors.

If you want a different angle of transition than LinearGradientMode provides, specify an angle, measured clockwise from the X axis, instead of the mode and pass .T. for the final parameter. An angle of 0 is the same as horizontal, 45 is the same as ForwardDiagonal, 90 is the same as Vertical, and 135 is the same as BackwardDiagonal. The following code, also taken from LinearGradientBrush.PRG, creates the image shown in **Figure 10**.

```
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object that draws on _SCREEN.

    loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Create a font object.

    loFont = .Font.New('Segoe UI', 16)

* Show various gradient modes.

    lnMode  = .Drawing2D.LinearGradientMode.Horizontal
    loRect  = .Rectangle.New(50, 50, 200, 200)
    loBrush = .Drawing2D.LinearGradientBrush.New(loRect, .Color.Red, ;
        .Color.White, lnMode)
    loGfx.FillRectangle(loBrush, loRect)
    loGfx.DrawString('Horizontal', loFont, .Brushes.Black, 50, 50)

    lnMode  = .Drawing2D.LinearGradientMode.ForwardDiagonal
    loRect  = .Rectangle.New(300, 50, 200, 200)
    loBrush = .Drawing2D.LinearGradientBrush.New(loRect, .Color.Red, ;
        .Color.White, lnMode)
    loGfx.FillRectangle(loBrush, loRect)
    loGfx.DrawString('ForwardDiagonal', loFont, .Brushes.Black, 300, 50)

    lnMode  = .Drawing2D.LinearGradientMode.Vertical
    loRect  = .Rectangle.New(50, 300, 200, 200)
```

```
   loBrush = .Drawing2D.LinearGradientBrush.New(loRect, .Color.Red, ;
      .Color.White, lnMode)
   loGfx.FillRectangle(loBrush, loRect)
   loGfx.DrawString('Vertical', loFont, .Brushes.Black, 50, 300)

   lnMode  = .Drawing2D.LinearGradientMode.BackwardDiagonal
   loRect  = .Rectangle.New(300, 300, 200, 200)
   loBrush = .Drawing2D.LinearGradientBrush.New(loRect, .Color.Red, ;
      .Color.White, lnMode)
   loGfx.FillRectangle(loBrush, loRect)
   loGfx.DrawString('BackwardDiagonal', loFont, .Brushes.Black, 300, 300)

* Use an angle to customize how the transition works.

   loRect  = .Rectangle.New(50, 550, 200, 200)
   loBrush = .Drawing2D.LinearGradientBrush.New(loRect, ;
      .Color.Red, .Color.White, 225, .T.)
   loGfx.FillRectangle(loBrush, loRect)
   loGfx.DrawString('Custom (225)', loFont, .Brushes.Black, 50, 550)
endwith
```

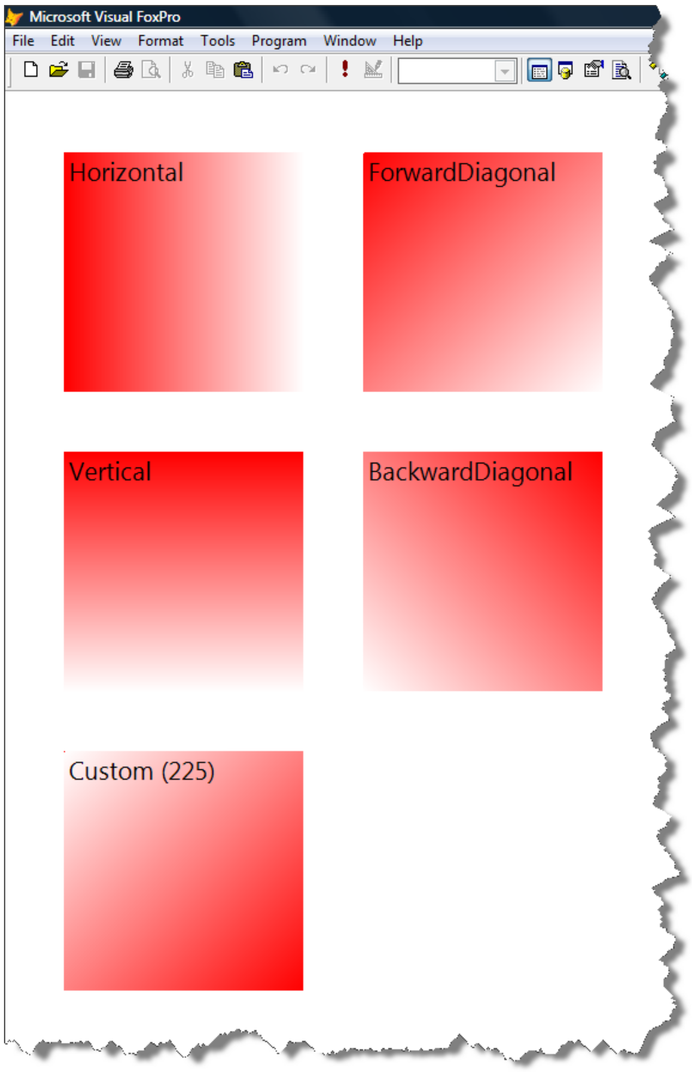


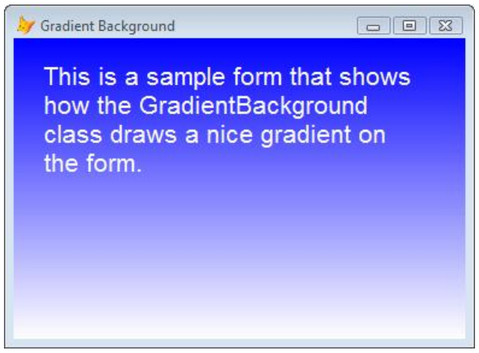
Figure 10. Different types of linear gradient brushes.

GradientImage, in GradientImage.VCX, is a simple Image subclass you can drop on a form to display a gradient image. Set nFromColor and nToColor to the RGB values for the starting and ending colors for the gradient. You can also set nGradientMode to the desired gradient mode; the default is -1, which means automatically use a horizontal gradient if the image is wider than it is tall or a vertical gradient if it's taller than wide. The Init method calls CreateGradientImage, which has the following code:

```
local lnWidth, ;
   lnHeight, ;
   lnMode, ;
   lnFromColor, ;
   lnToColor, ;
   loBitmap, ;
   loRect, ;
   loGfx, ;
   loBrush
```

```
* Ensure System is available.

do System.app
with _screen.System.Drawing

* If we're supposed to automatically determine the gradient mode, use a
* horizontal gradient if width > height or vertical if height > width.

   lnWidth  = This.Width
   lnHeight = This.Height
   do case
      case This.nGradientMode <> -1
         lnMode = This.nGradientMode
      case lnWidth > lnHeight
         lnMode = .Drawing2D.LinearGradientMode.Horizontal
      otherwise
         lnMode = .Drawing2D.LinearGradientMode.Vertical
   endcase

* Get the colors to use.

   if vartype(This.nFromColor) = 'N'
      lnFromColor = This.nFromColor
   else
      lnFromColor = evaluate(This.nFromColor)
   endif vartype(This.nFromColor) = 'N'
   if vartype(This.nToColor) = 'N'
      lnToColor = This.nToColor
   else
      lnToColor = evaluate(This.nToColor)
   endif vartype(This.nToColor) = 'N'

* Create a bitmap and a rectangle the size of this control.

   loBitmap = .Bitmap.New(lnWidth, lnHeight)
   loRect   = .Rectangle.New(0, 0, lnWidth, lnHeight)

* Create a graphics object.

   loGfx = .Graphics.FromImage(loBitmap)

* Create a gradient brush.

   loBrush = .Drawing2D.LinearGradientBrush.New(loRect, ;
      .Color.FromRgb(lnFromColor), .Color.FromRgb(lnToColor), lnMode)

* Fill the rectangle with the gradient brush.

   loGfx.FillRectangle(loBrush, loRect)

* Put the image into PictureVal.

   This.PictureVal = loBitmap.GetPictureValFromHBitmap()
endwith
return
```

GradientBackground is a subclass of GradientImage that simply sizes the image so it fills the form it's on. **Figure 11** shows an example of a form containing a GradientBackground object.



This is a sample form that shows how the GradientBackground class draws a nice gradient on the form.

Figure 11. Simply drop GradientBackground onto a form to create a gradient background on the form.

Texture brushes

System.Drawing.TextureBrush.New creates a texture brush, one that fills a region using an image. Pass it a reference to a loaded image. For example, this code, taken from TextureBrush.PRG, fills a couple of shapes from some images; the result is in **Figure 12**.

```
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object that draws on _SCREEN.

   loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Load an image, create a TextureBrush from it, and fill a rectangle.

   loImage = .Bitmap.New('AntiqueGreen.jpg')
   loBrush = .TextureBrush.New(loImage)
   loGfx.FillRectangle(loBrush, 50, 50, 200, 200)

* Use a different image in an ellipse.

   loImage = .Bitmap.New('RoseTea.jpg')
   loBrush = .TextureBrush.New(loImage)
   loGfx.FillEllipse(loBrush, 250, 50, 200, 200)
endwith
```

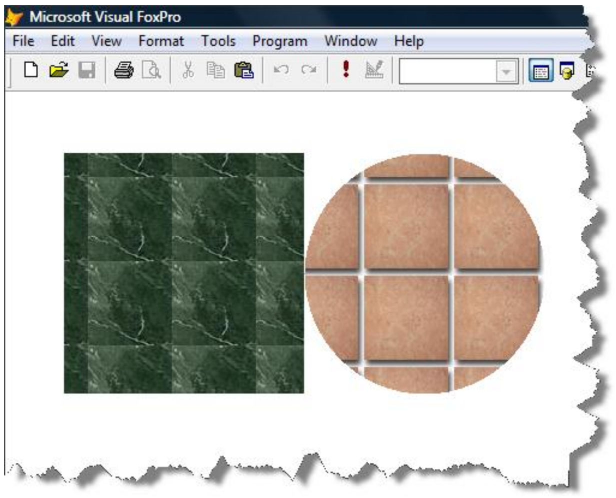


Figure 12. TextureBrush fills a region with an image.

The WrapMode property, which should be set to an enumeration of System.Drawing.Drawing2D.WrapMode, determines how the image is tiled.

Combining pens and brushes

Pens have a Brush property that determines how the pen fills the line it draws. By default, Brush contains a SolidBrush with the same color as the pen, but you can create interesting effects by replacing that brush with a different one. The code in Pens&Brushes.PRG creates the image shown in **Figure 13**.

```
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object that draws on _SCREEN.

    loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Draw a blue rectangle with a hatch brush pen. Notice how the brush color
* overrides the pen color.

    loPen   = .Pen.New(.Color.Red, 6)
    loBrush = .Drawing2D.HatchBrush.New(.Drawing2D.HatchStyle.ZigZag, ;
        .Color.Blue)
    loPen.Brush = loBrush
    loGfx.DrawRectangle(loPen, 50, 50, 100, 100)
endwith
```

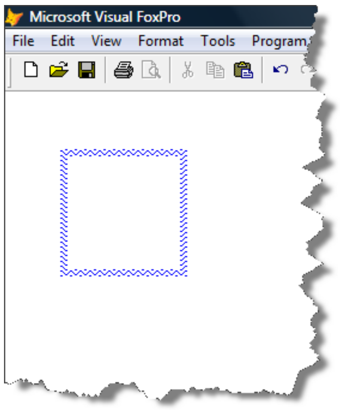


Figure 13. You can combine pens and brushes for interesting effects.

A simple pie chart

Let's tie together a few of the things we've discussed so far. PieChart.PRG creates the simple pie chart shown in **Figure 14**. As you can see, there isn't a lot of code involved.

```
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object that draws on _SCREEN.

   loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Create an array of values and colors we'll use for pie slices.

   dimension laValues[5, 2]
   laValues[1, 1] = 15
   laValues[1, 2] = .Color.Red
   laValues[2, 1] = 30
   laValues[2, 2] = .Color.Blue
   laValues[3, 1] = 25
   laValues[3, 2] = .Color.Green
   laValues[4, 1] = 10
   laValues[4, 2] = .Color.Salmon
   laValues[5, 1] = 20
   laValues[5, 2] = .Color.Teal

* Draw each slice of the pie.

   loBrush = .SolidBrush.New()
   lnStart = 0
   for lnI = 1 to alen(laValues, 1)
      lnValue       = laValues[lnI, 1]
      lnAngle       = lnValue * 360 / 100
      loBrush.Color = .Color.FromARGB(160, laValues[lnI, 2])
      loGfx.FillPie(loBrush, 50, 50, 300, 300, lnStart, lnAngle)
      lnStart = lnStart + lnAngle
   next lnI
endwith
```

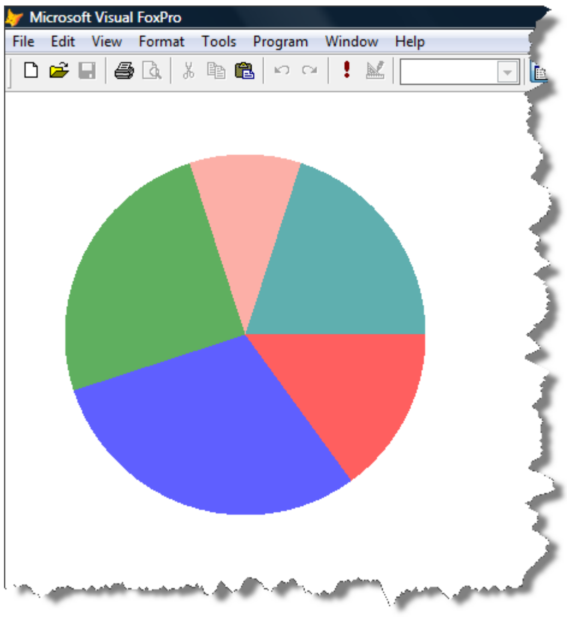


Figure 14. This simple pie chart was drawn using GDIPlusX.

Text

GDIPlusX provides classes to display text in a variety of fonts, sizes, and styles. This includes the ability to add text to images and with some advanced techniques, draw text in interesting ways, such as shadowed, curved, or haloed.

Start by creating a font object. The simplest way is to pass System.Drawing.Font.New the name of a font family and the font size, and optionally the style, which is an enumeration of System.Drawing.FontStyle (Regular, Bold, Italic, BoldItalic, Underline, and Strikeout):

```
loFont = .Font.New('Tahoma', 16, .FontStyle.Bold)
```

To draw the desired text, call the DrawString method of a Graphics object, passing it the text, a font object, a brush, and a Point, Rectangle, or X and Y values indicating the location to draw the text. DrawString word-wraps the text as necessary to fit within the rectangle if one is specified. For example:

```
loRect  = .Rectangle.New(10, 10, 200, 150)
loBrush = .SolidBrush.New(.Color.Red)
loGfx.DrawString(lcText, loFont, loBrush, loRect)
```

To justify the text other than the default left, create a StringFormat object, set its Alignment property to an enumeration of System.Drawing.StringAlignment, and pass it to DrawString. For example:

```
loStringFormat = .StringFormat.New()
loStringFormat.Alignment = .StringAlignment.Center
loGfx.DrawString(lcText, loFont, loBrush, loRect, loStringFormat)
```

The enumeration names of System.Drawing.StringAlignment are a little odd. Instead of Left or Right, use Near or Far. These names are used because left and right are reversed in right-to-left languages.

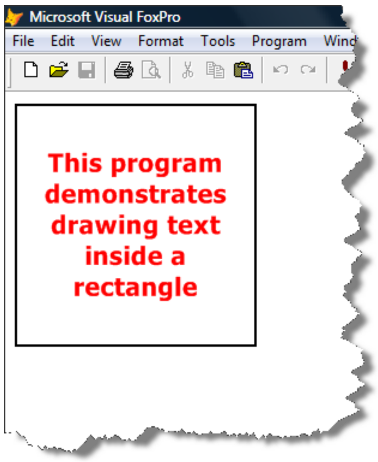
In addition to horizontal alignment, you can also set the vertical alignment by setting the string format object's LineAlignment to an enumeration of StringAlignment. You can also specify how the string is trimmed if it's too long to fit in the rectangle by setting the string format object's Trimming property to an enumeration of StringTrimming.

Text often renders better if you set the TextRenderingHint property of the Graphics object to System.Drawing.Drawing2D.SmoothingMode.AntiAlias.

Here's an example, taken from DemoText.PRG, which draws a string inside a rectangle on _SCREEN. The result is shown in **Figure 15**.

```
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Create a Graphics object. We'll create it by calling FromHWnd and passing it
* the handle (HWnd) to _SCREEN, which means we'll be drawing directly on
* _SCREEN.

   loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Use anti-aliasing for better results.

   loGfx.TextRenderingHint = .Drawing2D.SmoothingMode.AntiAlias

* The text to draw.

   lcText = 'This program demonstrates drawing text inside a rectangle'

* Create a font object. The New method of System.Drawing.Font creates a font
* object using specified font, size, and style. Note that style is specified as
* an enumeration; in this case, the Bold member of System.Drawing.FontStyle.

   loFont = .Font.New('Tahoma', 16, .FontStyle.Bold)

* Create a rectangle object to hold the dimensions we'll draw at.

   loRect = .Rectangle.New(10, 10, 200, 200)

* Create a red solid brush.

   loBrush = .SolidBrush.New(.Color.Red)

* Create a StringFormat object and specify center alignment both horizontally
* and vertically.

   loStringFormat               = .StringFormat.New()
   loStringFormat.Alignment     = .StringAlignment.Center
   loStringFormat.LineAlignment = .StringAlignment.Center

* Draw the text using the desired font and brush in the desired dimensions.
```

```
   loGfx.DrawString(lcText, loFont, loBrush, loRect, loStringFormat)

* Create a 2-pixel black pen.

   loPen = .Pen.New(.Color.Black, 2)

* Draw a rectangle using the pen at the desired location and size.

   loGfx.DrawRectangle(loPen, loRect)
endwith
```



This program demonstrates drawing text inside a rectangle

Figure 15. Drawing text using GDIPlusX methods.

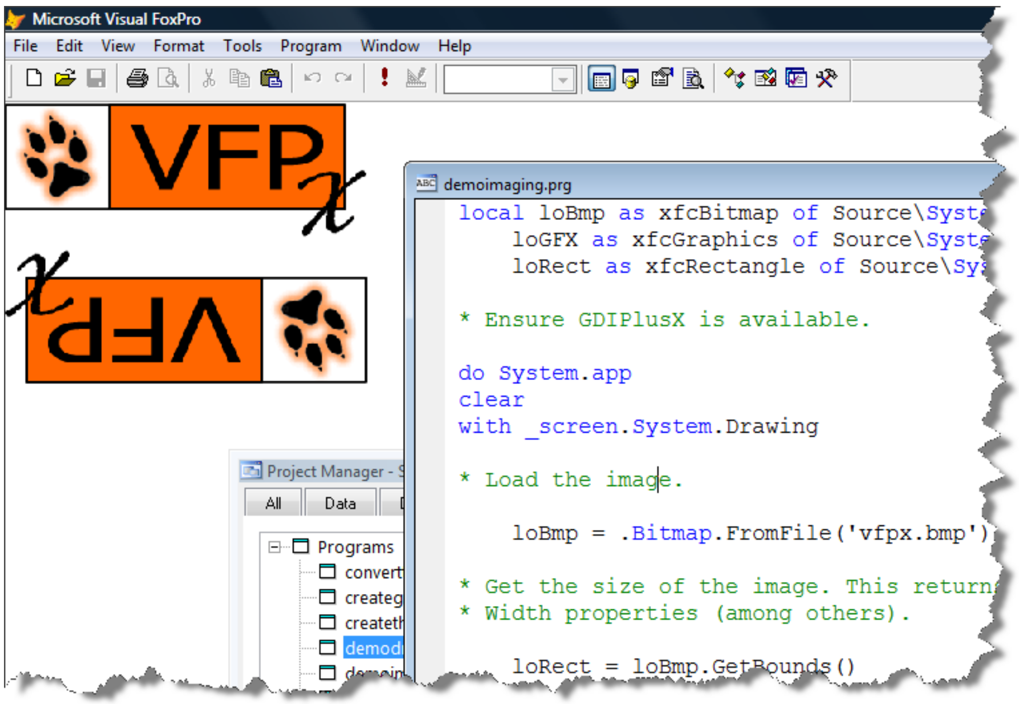
Images

It doesn't make sense to display certain kinds of pictures, such as icons and photographs, using vector graphics. Instead, images of this type are stored as bitmaps, which are arrays of numbers representing the colors of individual dots on the screen.

GDI+ provides several classes devoted to displaying and manipulating images. GDIPlusX has wrapper classes in the System.Drawing and System.Drawing.Imaging namespaces. DemoImaging.PRG provides an example that displays an image on _SCREEN, both as it normally appears and rotated 180 degrees (**Figure 16**):

```
local loBmp as xfcBitmap of Source\System.Drawing.PRG, ;
   loRect as xfcRectangle of Source\System.Drawing.PRG, ;
   loGfx as xfcGraphics of Source\System.Drawing.PRG

* Ensure GDIPlusX is available.

do System.app
clear
with _screen.System.Drawing as xfcDrawing of Source\System.PRG

* Load the image.
```

```
   loBmp = .Bitmap.FromFile('vfpx.bmp')

* Get the size of the image. This returns a rectangle object with Height and
* Width properties (among others).

   loRect = loBmp.GetBounds()

* Create a Graphics object. We'll create it by calling FromHWnd and passing it
* the handle (HWnd) to _SCREEN, which means we'll be drawing directly on
* _SCREEN.

   loGfx = .Graphics.FromHWnd(_screen.HWnd)

* Draw the image on the screen starting from the upper-left corner (loRect.X
* and Y are 0 by default) using the dimensions of the image.

   loGfx.DrawImage(loBmp, loRect)

* Rotate the image 180 degrees without flipping it.

   loBmp.RotateFlip(.RotateFlipType.Rotate180FlipNone)

* Draw the rotated image below the first one.

   loRect.X = loRect.Height + 10
   loGfx.DrawImage(loBmp, loRect)
endwith
```



```
local loBmp as xfcBitmap of Source\Syste
    loGFX as xfcGraphics of Source\Syste
    loRect as xfcRectangle of Source\Sy

* Ensure GDIPlusX is available.

do System.app
clear
with _screen.System.Drawing

* Load the image.

    loBmp = .Bitmap.FromFile('vfpx.bmp')

* Get the size of the image. This return
* Width properties (among others).

    loRect = loBmp.GetBounds()
```

Figure 16. Drawing images using GDIPlusX.

Images are also important because, although the examples in this document draw on _SCREEN for demonstration purposes, it's unlikely you'll do that in a real application. Instead, you'll likely draw on an image and then display that image using a VFP Image object. For example, the CreateGradientImage code we saw earlier draws a gradient on an image, which is then used as the source for the PictureVal property of an Image object. The nice thing about this is that the drawing code doesn't change, just the creation of the Graphics object used to do the drawing.

To draw on an image, start by creating a Bitmap object. You can either create a new one or load an image file if you want to draw on an existing image. Then use Graphics.FromImage to create a Graphics object from the Bitmap object. After using the Graphics object to do the desired drawing, you can save the drawing to a file using Bitmap.Save or display it by calling GetPictureValFromHBitmap and putting the results into the PictureVal property of an Image object as CreateGradientImage does.

To specify the type of image file to save to, pass Bitmap.Save a member of System.Drawing.Imaging.ImageFormat. For example, the following code, taken from CreateGradient.PRG, determines what type of image to create based on the extension of the specified file name:

```
lcExt = upper(justext(tcFileName))
do case
   case lcExt = 'PNG'
      loFormat = .Imaging.ImageFormat.Png
   case lcExt = 'BMP'
      loFormat = .Imaging.ImageFormat.Bmp
   case lcExt = 'GIF'
      loFormat = .Imaging.ImageFormat.Gif
   case inlist(lcExt, 'JPG', 'JPEG')
      loFormat = .Imaging.ImageFormat.Jpeg
   case lcExt = 'ICO'
      loFormat = .Imaging.ImageFormat.Icon
   case inlist(lcExt, 'TIF', 'TIFF')
      loFormat = .Imaging.ImageFormat.Tiff
   case lcExt = 'WMF'
      loFormat = .Imaging.ImageFormat.Wmf
endcase
loBmp = .Bitmap.New(tnWidth, tnHeight)
* Drawing code goes here
loBmp.Save(tcFileName, loFormat)
```

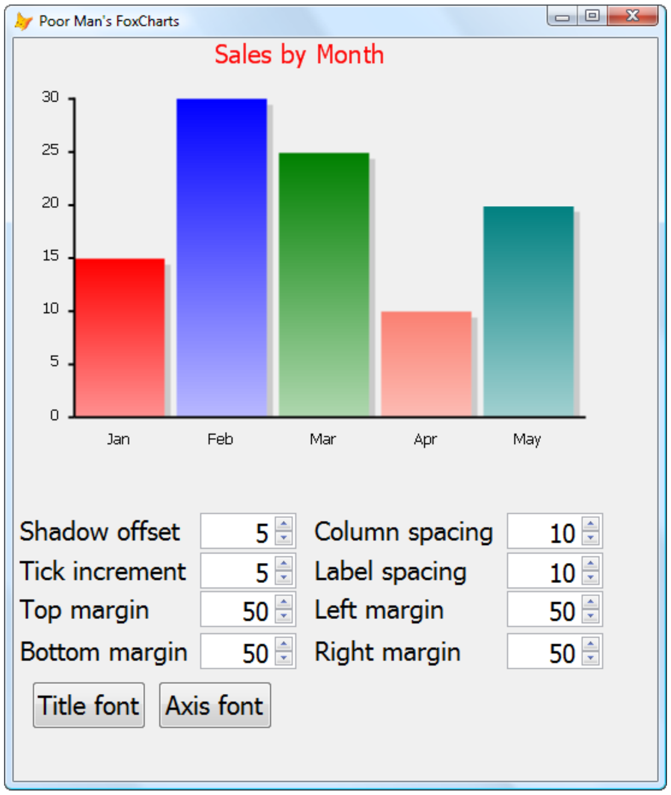
imgCanvas

Because displaying a drawing on a VFP Image is a common thing to do, GDIPlusX includes a subclass of an Image called imgCanvas (in GDIPlusX.VCX) that does all the work of creating a Graphics object and sending the drawing to the Image. Drop imgCanvas on a form, size it as desired, and put code in BeforeDraw to do the drawing, using This.oGfx as the Graphics object. Other than removing the statement creating a Graphics object and using "This.oGfx" instead of “loGfx”, the code in all of the samples included with this document could be used as is in an imgCanvas object.

Here are some tips about using imgCanvas:

- To redraw the image, such as if something affecting the drawing changes, call the Draw method. Depending on your drawing, you may need to call Clear first.
- The Resize event calls This.Draw, so your BeforeDraw code automatically redraws the image. You may wish to set Anchor to an appropriate value (such as 15 to resize both horizontally and vertically) so resizing the form automatically resizes the image.
- Because resizing the image forces it to be redrawn, you may want to use the Height and Width to determine the size or scaling for the drawing.

Let’s look at a more complex sample that ties together many of the things we’ve seen in GDIPlusX. ColumnChart.SCX is a “poor man’s FoxCharts.” FoxCharts, another VFPX project, uses GDIPlusX to draw beautiful 3-D charts. ColumnChart.SCX just does a simple column chart from hard-coded data, but serves as a example for drawing in GDIPlusX and uses imgCanvas as its drawing surface. **Figure 17** shows what the form looks like when you run it. Try resizing the form, changing the values of the spinners, or changing the fonts by clicking the buttons. We won’t look at the code in BeforeDraw here because it’s fairly long; however, it’s relatively simple code and well-commented.



Shadow offset 5
Tick increment 5
Top margin 50
Bottom margin 50
Column spacing 10
Label spacing 10
Left margin 50
Right margin 50

Figure 17. ColumnChart.SCX ties together many of the GDIPlusX concepts.

Using .Net examples

As I mentioned at the beginning of this white paper, one of the advantages of developing GDIPlusX to closely mimic the .Net namespaces is the ability to use the thousands of .Net samples available in VFP with only a little translation.

As an example, here's some VB.Net code taken from http://www.bobpowell.net/highqualitythumb.htm. This code reduces an image by the specified percentage.

```
Public Function GenerateThumbnail(original As Image, percentage As Integer) As Image
    If percentage < 1 Then
        Throw New Exception("Thumbnail size must be at least 1% of the original size")
    End If
    Dim tn As New Bitmap(CInt(original.Width * 0.01F * percentage), _
        CInt(original.Height * 0.01F * percentage))
    Dim g As Graphics = Graphics.FromImage(tn)
    g.InterpolationMode = InterpolationMode.HighQualityBilinear
    g.DrawImage(original, New Rectangle(0, 0, tn.Width, tn.Height), _
        0, 0, original.Width, original.Height, GraphicsUnit.Pixel)
    g.Dispose()
    Return CType(tn, Image)
End Function
```

Even if you're not familiar with VB.Net, the GDIPlusX part of the code should be fairly clear, Here's the VFP equivalent, line by line:

```
*Public Function GenerateThumbnail(original As Image, percentage As Integer) As
Image
function GenerateThumbnail(original, percentage)

*If percentage < 1 Then
*  Throw New Exception("Thumbnail size must be at least 1% of the original size")
*End If
if percentage < 1
   error "Thumbnail size must be at least 1% of the original size"
endif

* New code: ensure GDIPlusX is available.
do System
with _screen.System.Drawing as xfcDrawing of source\System.Drawing.prg
* End of new code

*  Dim tn As New Bitmap(CInt(original.Width * 0.01F * percentage), _
*     CInt(original.Height * 0.01F * percentage))
   tn = .Bitmap.New(int(original.Width * 0.01 * percentage), ;
      int(original.Height * 0.01 * percentage))

*  Dim g As Graphics = Graphics.FromImage(tn)
   gx = .Graphics.FromImage(tn)
      && need different variable name because "g" looks like a workarea

*  g.InterpolationMode = InterpolationMode.HighQualityBilinear
   gx.InterpolationMode = .Drawing2D.InterpolationMode.HighQualityBilinear

*  g.DrawImage(original, New Rectangle(0, 0, tn.Width, tn.Height), _
*     0, 0, original.Width, original.Height, GraphicsUnit.Pixel)
   gx.DrawImage(original, .Rectangle.New(0, 0, tn.Width, tn.Height), ;
      0, 0, original.Width, original.Height, .GraphicsUnit.Pixel)

*  g.Dispose()
   gx.Dispose()

* New code: finish WITH
endwith
* End of new code

*Return CType(tn, Image)
return tn

*End Function
endfunc
```

As you can see, most of the changes are due to difference between VFP and VB.Net syntax. The GDIPlusX calls are almost identical between the two languages. The major differences are:

- In VB.Net, members of namespaces added as references can be specified without the complete namespace, such as “Rectangle.” In VFP, you need to use the WITH statement and precede members with a period, such as “.Rectangle.”
- Some classes are deeper in the hierarchy that System.Drawing, so, as in the case of “InterpolationMode,” you have to specify the additional class name (“.Drawing2D. InterpolationMode” in this case).
- VB.Net syntax to create a new object is “New *ClassName*,” such as “New Rectangle.” In VFP, call the New method of the GDIPlusX class, such as “.Rectangle.New.”

Resources

Several VFP developers have great blogs that focus on GDIPlusX, including Cesar Chalom (http://weblogs.foxite.com/vfpimaging), Bernard Bout (http://weblogs.foxite.com/bernardbout), and Bo Durban (http://blog.moxiedata.com).

The GDIPlusX References page on VFPX (http://vfpx.codeplex.com/Wiki/View.aspx?title=GDIPlusX%20References) has links to many GDI+ and GDIPlusX articles. Also, GDIPlusX comes with a lot of samples that show individual techniques, so it’s worth trying them out.

I found the various GDI+ tutorials at www.bobpowell.net to be very informative and well-written. They’re in C# and VB.Net but are easy to follow and convert to VFP.

Summary

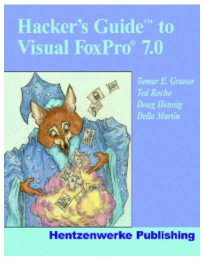
GDIPlusX is one of the most important projects on VFPX because it forms the foundation for several other projects. GDIPlusX makes it easy to add new graphical abilities to your applications, allowing you to provide a fresher and more powerful user interface. Because it has the same structure as the GDI+ .Net classes, you can use the thousands of .Net samples available in your own VFP code.

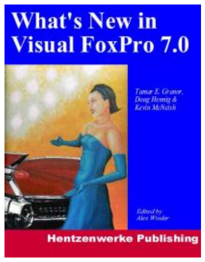
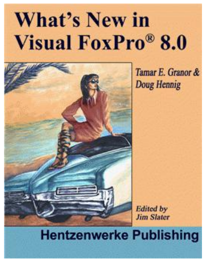
Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. and Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of “Making Sense of Sedna and SP2,” the “What’s New in Visual FoxPro” series (the latest being “What’s New in Nine”), “Visual FoxPro Best Practices For The Next Ten Years,” and “The Hacker’s Guide to Visual FoxPro 7.0.” He was the technical editor of “The Hacker’s Guide to Visual FoxPro 6.0” and “The Fundamentals.” All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He has been a Microsoft Most Valuable Professional (MVP) since 1996. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://fox.wikis.com/wc.dll?Wiki~FoxProCommunityLifetimeAchievementAward~VFP).




What's New in Visual FoxPro 7.0
Tamar E. Granor, Doug Hennig & Kevin McNeish
Hentzenwerke Publishing

What's New in Visual FoxPro® 8.0
Tamar E. Granor & Doug Hennig
Edited by Jim Slater
Hentzenwerke Publishing