Programming GDI+ in Vulcan.Net

Chris Pyrgas

Introduction

GDI+ is the successor to the old GDI (Graphics Programming Interface), responsible for drawing and painting on the screen and the printer. In this paper we will discuss the basics of GDI+ and demonstrate how to use it in Vulcan.Net, through its managed class interface.

GDI+ Structure

The function/handle interface model of GDI is replaced in GDI+ with a class interface model. GDI+ is exposed to the programmer through a set of classes, counting more than a hundred members that provide an easy way to use the numerous features of GDI+. Those features include basic ones like drawing simple shapes (lines, rectangles etc), text and images, and extend to more advanced like transparency, alpha blending, painting with gradient colors and drawing of complex shapes. Also almost every method in GDI+ classes is overloaded with many versions, offering different ways to accomplish the same task.

NameSpaces

A Namespace can be described as a way to group together classes that are used for a similar purpose. A namespace can contain any number of classes that may span around many DLL files. Most basic GDI+ classes (Point, Rectangle, Color, Brush etc) are grouped into the namespace named “System.Drawing”, while some classes implementing advanced functionality are included in the following namespaces:

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

The classes and methods included in the above namespaces are declared in a single DLL file named “System.Drawing.dll”. This is contained in the Global Assembly Cache (GAC) folder, usually located at “C:\WINDOWS\GAC” in every Windows PC with the .Net Framework installed.

Framework app

In order to demonstrate using GDI+, we need to create a simple framework application. The following code creates and shows a simple form on the screen:

```
REFERENCES "System.Windows.Forms.dll"
REFERENCES "System.Drawing.dll"

FUNCTION Start() AS VOID
    LOCAL oForm AS SampleForm
    oForm:=SampleForm{}
    oForm:Show()
    System.Windows.Forms.Application.Run(oForm)
RETURN

CLASS SampleForm INHERIT System.Windows.Forms.Form
CONSTRUCTOR() CLASS SampleForm
    SUPER()
    SELF:Text:='GDI Sample'
    SELF:ClientSize:=System.Drawing.Size{400,400}
RETURN
```

The first two lines of code instruct the compiler that we want to use some classes contained in the two referred DLL files, “System.Windows.Forms.dll” and “System.Drawing.dll”. The first one contains classes from the same named namespace System.Windows.Forms, while the second contains classes from the various System.Drawing.* namespaces mentioned above.

System.Windows.Forms.Form and System.Drawing.Size are the fully qualified names of the Form and Size classes, contained in System.Windows.Forms.Form and System.Drawing namespaces respectively. Since we will be making much use of classes in those namespaces, there fortunately exists a way to tell the compiler that the namespace part of those classes should be implied, saving us a lot of typing. This is achieved with the ‘USING <NameSpace>” compiler directive:

```
REFERENCES "System.Windows.Forms "
REFERENCES "System.Drawing "

USING System.Windows.Forms
USING System.Drawing

FUNCTION Start() AS VOID
    LOCAL oForm AS SampleForm
    oForm:=SampleForm{}
    oForm:Show()
    oForm:DoSomeDrawing()
    Application.Run(oForm)
RETURN

CLASS SampleForm INHERIT Form
CONSTRUCTOR() CLASS SampleForm
    SUPER()
    SELF:Text:='GDI Sample'
    SELF:ClientSize:=Size{600,400}
RETURN

METHOD DoSomeDrawing() AS VOID CLASS SampleForm
RETURN
```

Also note that in the above sample, we have used another version of the ***REFERENCES*** *directive, which uses an* ***assembly*** *name (“System.Windows.Forms”, “System.Drawing”) instead of a DLL filename. For more information about the REFERENCES directive, please refer to the Vulcan.NET documentation.*

Obtaining a Graphics object

In the code above we have added a call to `DoSomeDrawing()` method in our form after showing it, that will be used for some simple painting. In order to draw something in our form, we need to get a Graphics object. This is the equivelant of a Device Context (DC) in GDI, and contains numerous methods and properties for drawing shapes, text and images in many different styles. We can obtain a Graphics object, associated with our form, by using the form's `CreateGraphics()` method:

```
METHOD DoSomeDrawing() AS VOID CLASS SampleForm

    LOCAL oGraphics AS Graphics

    oGraphics := SELF:CreateGraphics()
    // do some drawing
    oGraphics:Dispose()

RETURN
```

When we're done with our painting, we need to release the resources used by this Graphics object, by calling its `Dispose()` method.

Drawing lines

Now let's do our first simple drawing. We will use Graphics object's `DrawLine()` method to draw a line on the form surface. One of the overloads of `DrawLine()` is declared as follows (translated from C# into Vulcan.NET code) :

```
METHOD DrawLine(oPen AS Pen, x1 AS INT, y1 AS INT, x2 AS INT, y2 AS INT ) ;
        AS VOID CLASS Graphics
```

This method draws a line from (x1,y1) to (x2,y2) coordinate pairs using an instance of the Pen class. By default the (0,0) point is located at the upper left corner of the form's client area.

Pen class

The Pen object describes the style that will be used when drawing a line. The simplest Pen class constructor needs a Color object passed to it:

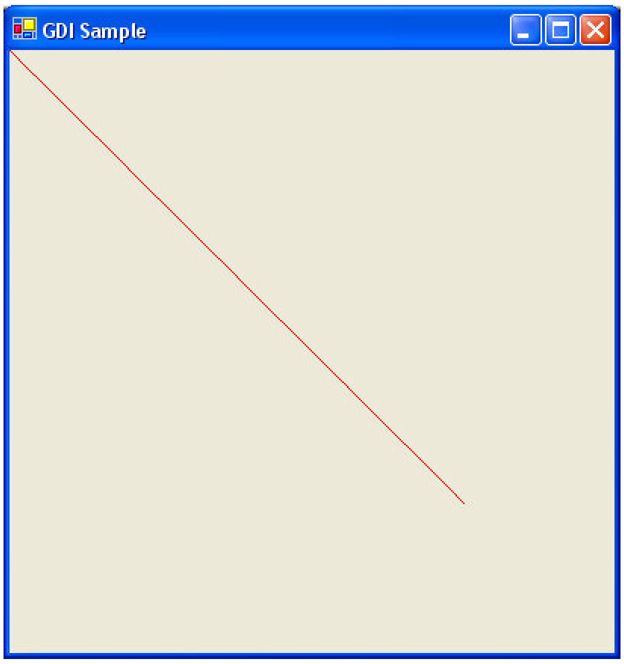
```
CONSTRUCTOR ( oColor AS Color) CLASS Pen
```

Color is actually a .Net Structure, but we can treat it like a class in Vulcan.NET. Many predefined Color objects (called Static Properies) with names like Color.Red, Color.Blue etc are already available for use, but we can also create custom colors by mixing the specified Red, Green and Blue values using the `Color.FromArgb(Red, Green, Blue)` function (Static Method in .Net).

In our example we will just use a Pen object with a Red color. The code for drawing a line is the following:

```
METHOD DoSomeDrawing() AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    LOCAL oPen AS Pen
    oGraphics := SELF:CreateGraphics()
    oPen := Pen{ Color.Red }
    oGraphics:DrawLine(oPen, 0, 0, 300, 300)
    oGraphics:Dispose()
RETURN
```

If you add this code in the framework app and run it, the result will be a form with a diagonal line drawn on it :



Drawing lines using Point objects

Let's have a look at another overload of the `DrawLine()` method :

```
METHOD DrawLine(oPen AS Pen, pt1 AS Point, pt2 AS Point) AS VOID CLASS Graphics
```

This overload requires two instances of the Point class, instead of four integers. The point class (again a structure in .Net terms) represents a pair of (x,y) coordinates; pt1 is the point where the line begins and pt2 is the ending point. The most commonly used Point class Constructor is declared as:

```
CONSTRUCTOR (x AS INT, y AS INT) CLASS Point
```

The above example can be rewritten using Points in the following way:

```
METHOD DoSomeDrawing() AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    LOCAL oPen AS Pen
    LOCAL oStartPoint,oEndPoint AS Point
    oGraphics := SELF:CreateGraphics()
    oPen := Pen{ Color.Red}
    oStartPoint := Point{ 0, 0 }
    oEndPoint := Point{ 300, 300 }
    oGraphics:DrawLine(oPen, oStartPoint, oEndPoint)
    oGraphics:Dispose()
RETURN
```

Pen properties

The result with a simple line is not very impressive, but there are ways to make our lines look a bit fancier.

Width

We can change a line's thickness from only one pixel wide to any size, using the Width property:

```
oPen:Width := 5  // set pen width to 5 pixels
```

Color

After instantiating a Pen object, we can change its drawing color at will, by simply reassigning a different **Color** value to its Color property:

```
oPen:Color := Color.Blue  // set pen's color to blue
```

DashStyle

Except for solid lines, we can draw dotted or dashed lines using the DashStyle property. In this property we can assign a member of the **DashStyle** enumeration (Enumerations can be described as a number of DEFINE's declared for a similar purpose, grouped under the same base name.). Members of **DashStyle** include:

- DashStyle.Solid (solid line)
- DashStyle.Dot (dotted line)
- DashStyle.Dash (dashed line)
- DashStyle.DashDot (line with alternating dots and dashes)

An example of assigning a DashStyle value to a Pen:

```
oPen:DashStyle := DashStyle.Dash // set dash style to dashes
```

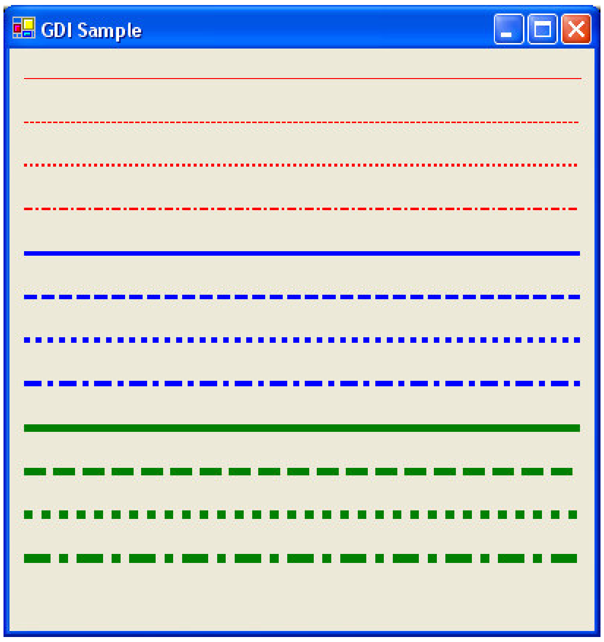
The following version of `DoSomeDrawing()` demonstrates usage of the Width, Color and DashStyle properties:

```
METHOD DoSomeDrawing() AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    LOCAL oPen AS Pen
    LOCAL n AS INT
    oGraphics := SELF:CreateGraphics()
    oPen := Pen{ Color.Empty }
    FOR n:=0 UPTO 11
            DO CASE
            CASE n <= 3
                    oPen:Color := Color.Red
            CASE n <= 7
                    oPen:Color := Color.Blue
            OTHERWISE
                    oPen:Color := Color.Green
            END CASE
            DO CASE
            CASE n % 4 == 0
                    oPen:DashStyle := DashStyle.Solid
            CASE n % 4 == 1
                    oPen:DashStyle := DashStyle.Dash
            CASE n % 4 == 2
                    oPen:DashStyle := DashStyle.Dot
            CASE n % 4 == 3
                    oPen:DashStyle := DashStyle.DashDot
            END CASE
            oPen:Width := n / 2 + 1
            oGraphics:DrawLine(oPen, 10, 20 + n * 30, 390, 20 + n * 30)
    NEXT
    oGraphics:Dispose()
RETURN
```

DashStyle enumeration is contained in System.Drawing.Drawing2D namespace, so we also need to add the following line of code in the beginning of our source:

```
USING System.Drawing.Drawing2D
```

After making the above changes in our code and running it, the output should look like this:



Using OnPaint() callback method for drawing operations

Drawing with the method described above (by creating a Graphics objects with `CreateGraphics()`) has a big disadvantage; everything we draw on the form is volatile. If the user moves another window over our form and removes it back, the lines we've drawn will disappear. The solution is to do all drawing operations within the Form's `OnPaint()` callback method, which is automatically called every time Windows decides that a part of our Form's contents or all of it needs to be repainted. The OnPaint method is declared as:

```
METHOD OnPaint ( e AS PaintEventArgs ) AS VOID CLASS Form
```

PaintEventArgs is a class that holds information about which part of the form needs painting (**ClipRect** property) and provides us a Graphics object that we can use to paint the form's surface. When we obtain a graphics object through the OnPaint method, we don't need to (and should not) release it with its `Dispose()` method. A sample using `OnPaint()` looks like this:

```
METHOD OnPaint(e AS PaintEventArgs) AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    oGraphics := e:Graphics
    // do some drawing here
RETURN
```

Here is also the full previous line sample, rewritten using `OnPaint()` method instead of `DoSomeDrawing()`:

```
REFERENCES "System.Windows.Forms "
REFERENCES "System.Drawing "

USING System.Windows.Forms
USING System.Drawing
USING System.Drawing.Drawing2D

FUNCTION Start() AS VOID
    LOCAL oForm AS SampleForm
    oForm:=SampleForm{}
    oForm:Show()
    Application.Run(oForm)
RETURN

CLASS SampleForm INHERIT Form
CONSTRUCTOR() CLASS SampleForm
    SUPER()
    SELF:Text:='GDI Sample'
    SELF:ClientSize:=Size{400,400}
RETURN

METHOD OnPaint(e AS PaintEventArgs) AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    LOCAL oPen AS Pen
    LOCAL n AS INT
    oGraphics := e:Graphics
    oPen := Pen{ Color.Empty }
    FOR n:=0 UPTO 11
            DO CASE
            CASE n <= 3
                    oPen:Color := Color.Red
            CASE n <= 7
                    oPen:Color := Color.Blue
            OTHERWISE
                    oPen:Color := Color.Green
            END CASE
            DO CASE
            CASE n % 4 == 0
```

```
                oPen:DashStyle := DashStyle.Solid
        CASE n % 4 == 1
                oPen:DashStyle := DashStyle.Dash
        CASE n % 4 == 2
                oPen:DashStyle := DashStyle.Dot
        CASE n % 4 == 3
                oPen:DashStyle := DashStyle.DashDot
        END CASE
        oPen:Width := n / 2 + 1
        oGraphics:DrawLine(oPen, 10, 20 + n * 30, 390, 20 + n * 30)
    NEXT
RETURN
```

The result on screen will be exactly the same as before, only this time the lines will be redrawn each time a part of the form gets invalidated.

Drawing Shapes

Enough is enough with simple lines! Let’s move on now and draw some a bit more complex shapes like rectangles, circles and ellipses. The Graphics class’ method used for drawing rectangles is (not surprisingly!) named `DrawRectangle()`:

```
METHOD DrawRectangle(oPen AS Pen, x AS INT, y AS INT, width AS INT, height AS INT) ;
            AS VOID CLASS Graphics
```

This method draws a rectangle starting at (x,y) with the specified width and height, using a Pen object. Again we can assign to the Pen object the styles (color, width etc) described before for painting lines.

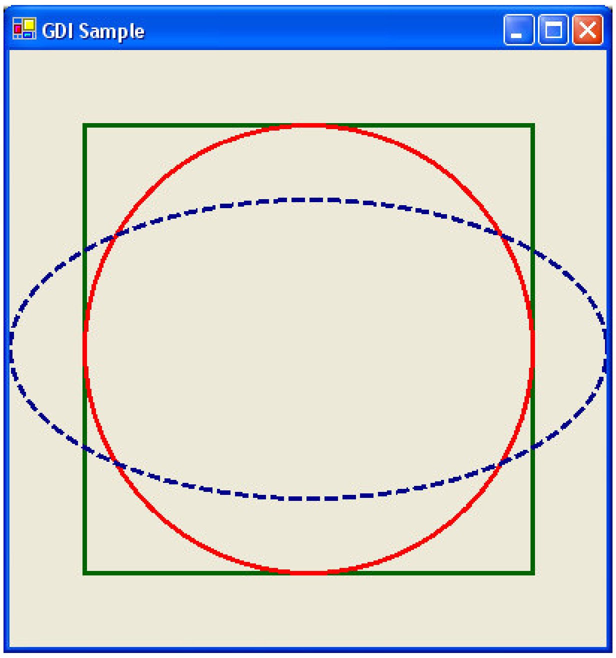
Circles and ellipses can be drawn with the `DrawEllipse()` method:

```
METHOD DrawEllipse(oPen AS Pen, x AS INT, y AS INT, width AS INT, height AS INT) ;
            AS VOID CLASS Graphics
```

The (x,y) and (width, height) pairs define the bounding rectangle in which the ellipse will be drawn into. If the width value equals to the height value the shape drawn will be a circle, otherwise it will be a long or tall ellipse. The following version of the `OnPaint()` method draws a rectangle on screen, a circle bound to it and an ellipse drawn with a dashed Pen. It also demonstrates using another constructor of Pen class that accepts two arguments, a Color and a Width value (in a FLOAT number):

```
METHOD OnPaint(e AS PaintEventArgs) AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    LOCAL oPen AS Pen
    oGraphics := e:Graphics

    oPen := Pen { Color.DarkGreen , 3.0}
    oGraphics:DrawRectangle(oPen, 50, 50, 300, 300)

    oPen:Color := Color.Red
    oGraphics:DrawEllipse(oPen, 50, 50, 300, 300)

    oPen:Color := Color.DarkBlue
    oPen:DashStyle := DashStyle.Dash
    oGraphics:DrawEllipse(oPen, 0, 100, 400, 200)
RETURN
```

The screen output:



Using Rectangle class

Rectangles and ellipses can be also drawn with another overload that uses a **Rectangle** class as an argument, instead of the (x,y) and (width, height) pairs:

```
METHOD DrawRectangle(oPen AS Pen, oRect AS Rectangle) AS VOID CLASS Graphics

METHOD DrawEllipse(oPen AS Pen, oRect AS Rectangle) AS VOID CLASS Graphics
```

Rectangle class defines a rectangle by its starting location, width and height. A rectangle object can be instantiated using one of its two available constructor methods:

```
CONSTRUCTOR (x AS INT, y AS INT, width AS INT, height AS INT) CLASS Rectangle
CONSTRUCTOR (location AS Point, size AS Size) CLASS Rectangle
```

The previous example can be written in a better way with the Rectangle class:

```
METHOD OnPaint(e AS PaintEventArgs) AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    LOCAL oPen AS Pen
    LOCAL oRectangle AS Rectangle
    oGraphics := e:Graphics

    oPen := Pen { Color.DarkGreen , 3.0}
    oRectangle := Rectangle { 50, 50, 300, 300 }
    oGraphics:DrawRectangle(oPen, oRectangle)

    oPen:Color := Color.Red
    oGraphics:DrawEllipse(oPen,  oRectangle)

    oPen:Color := Color.DarkBlue
    oPen:DashStyle := DashStyle.Dash
    oGraphics:DrawEllipse(oPen, 0, 100, 400, 200)
RETURN
```

More drawing methods

Except for rectangles and ellipses, the Graphics class provides methods for drawing a vast number of other objects like polygons, arcs, pies, curves, splines and more. You can use them in a very similar way to the methods described above and the .Net Framework SDK Documentation is a great source of information on their usage.

Drawing Filled Objects

Almost every `Draw…()` method of the Graphics object has a corresponding `Fill…()` method that draws the respective closed shape filed with some color (or in more complex ways as we will see below).

The methods for drawing filled rectangles and ellipses are:

```
METHOD FillRectangle(oBrush AS Brush, x AS INT, y AS INT, width AS INT, height AS INT) ;
          AS VOID CLASS Graphics

METHOD FillEllipse(oBrush AS Brush, x AS INT, y AS INT, width AS INT, height AS INT) ;
          AS VOID CLASS Graphics
```

And their overloads using a Recangle object:

```
METHOD FillRectangle(oBrush AS Brush, oRect AS Rectangle) AS VOID CLASS Graphics

METHOD FillEllipse(oBrush AS Brush, oRect AS Rectangle) AS VOID CLASS Graphics
```

Brushes

Both fill methods require a **Brush** object as their first argument that describes the method used to fill a shape. Brush is an abstract class, which means it cannot be instantiated directly; instead we need to use one of the classes that inherit from the Brush class:

- SolidBrush
- TextureBrush
- HatchBrush
- LinearGradientBrush
- PathGradientbrush

We will take a closer look at the simplest subclass, SolidBrush:

SolidBrush

SolidBrush class is used for painting shapes filled with a single color. It has only one Constructor method available, which requires a **Color** object as its single argument:

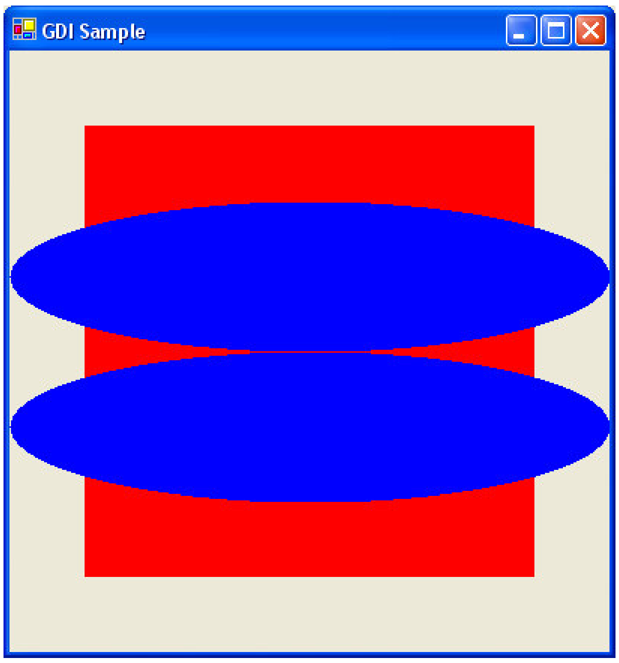
```
CLASS SolidBrush INHERIT Brush

CONSTRUCTOR ( oColor AS Color) CLASS SolidBrush
```

So in order to create a Solidbrush object, we need to use for example:

```
LOCAL oBrush AS SolidBrush
oBrush := SolidBrush { Color.Yellow } // creates a solid yellow brush
```

Now let's put the pieces together. The following version of `OnPaint()` method draws a filled rectangle and two ellipses on our form:

```
METHOD OnPaint(e AS PaintEventArgs) AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    LOCAL oBrush AS SolidBrush
    oGraphics := e:Graphics

    oBrush := SolidBrush { Color.Red }
    oGraphics:FillRectangle(oBrush, 50, 50, 300, 300)

    oBrush:Color := Color.Blue

    oGraphics:FillEllipse(oBrush, 0, 100, 400, 100)
    oGraphics:FillEllipse(oBrush, 0, 200, 400, 100)
RETURN
```



Transparent colors/brushes

Shapes can be also filled with semi-transparent brushes, using a GDI+ feature named **Alpha Blending**. We create a transparent brush by using a transparent color to instantiate it. A transparent color can be obtained with each of the following two `Color.FromArgb()` functions (static class methods):

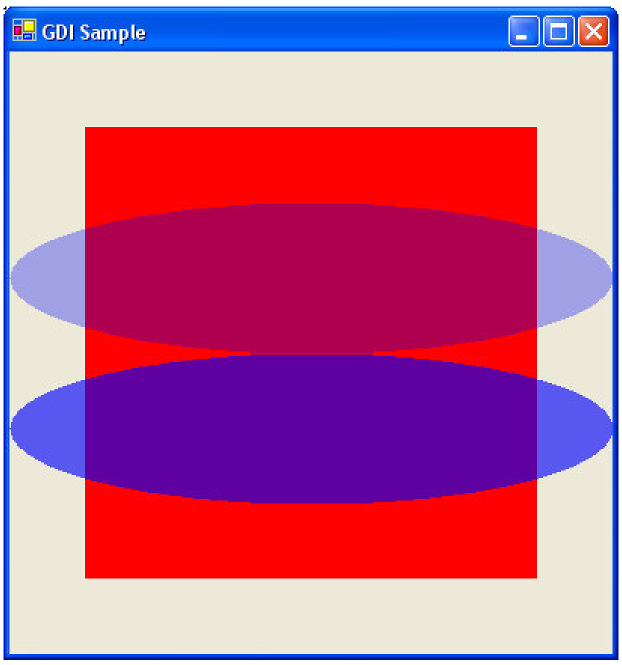
```
FUNCTION Color.FromArgb (alpha AS INT, red AS INT, green AS INT, blue AS INT) AS Color
FUNCTION Color.FromArgb (alpha AS INT, oColor AS Color) AS Color
```

The first overload creates a Color object using the specified **alpha** value and the red, green and blue component values (all ranging from 0 to 255), while the second receives a Color object and returns the same color with the specified alpha value applied to it. The alpha value represents the level of transparency; a value of 0 represents a totally transparent color and a value of 255 means the color is opaque:

```
oColor := Color.FromArgb(128, Color.Blue)  // create a half-transparent blue color
oBrush := SolidBrush { oColor }  // create a half-transparent solid blue brush
```

The following code draws again a solid red filled rectangle, but uses two different levels of transparency for the blue ellipses:

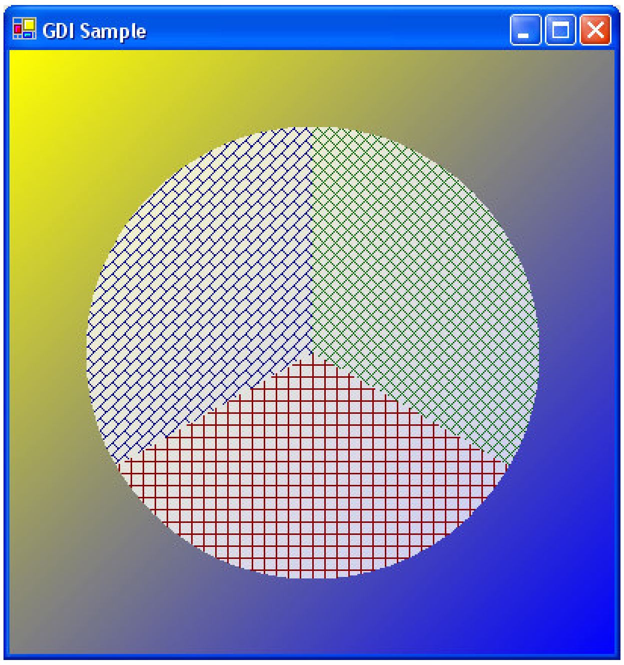
```
METHOD OnPaint(e AS PaintEventArgs) AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    LOCAL oBrush AS SolidBrush
    oGraphics := e:Graphics

    oBrush := SolidBrush { Color.Red }
    oGraphics:FillRectangle(oBrush, 50, 50, 300, 300)

    oBrush:Color := Color.FromArgb(80, Color.Blue)
    oGraphics:FillEllipse(oBrush, 0, 100, 400, 100)

    oBrush:Color := Color.FromArgb(160, Color.Blue)
    oGraphics:FillEllipse(oBrush, 0, 200, 400, 100)
RETURN
```



Other Brush types

A SolidBrush object is a great tool for simple drawing, but we can get a lot more impressive results using one of the other more sophisticated **Brush** subclasses. In-depth discussions of those brush styles goes beyond the scope of this tutorial, but let's have a look at a small sample using a **LinearGradientBrush** and a **HatchBrush.** Hopefully this will act as an appetizer for you to have a closer look at them!

```
METHOD OnPaint(e AS PaintEventArgs) AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    LOCAL oLBrush AS LinearGradientBrush
    LOCAL oHBrush AS HatchBrush
    LOCAL oBackColor AS Color
    oGraphics := e:Graphics

    oLBrush := LinearGradientBrush { Point{0,0} , Point{400,400} ,;
                    Color.Yellow , Color.Blue}
    oGraphics:FillRectangle(oLBrush, 0, 0, 400, 400)

    oBackColor := Color.FromArgb(192 , Color.White)

    oHBrush := HatchBrush { HatchStyle.Cross , Color.DarkRed , oBackColor }
    oGraphics:FillPie(oHBrush, 50, 50, 300, 300 , 30 ,120)

    oHBrush := HatchBrush { HatchStyle.DiagonalBrick , Color.DarkBlue , oBackColor }
    oGraphics:FillPie(oHBrush, 50, 50, 300, 300 , 150 ,120)

    oHBrush := HatchBrush { HatchStyle.OutlinedDiamond , Color.DarkGreen , oBackColor }
    oGraphics:FillPie(oHBrush, 50, 50, 300, 300 , 270 ,120)
RETURN
```



Drawing Text

Drawing text is a very easy thing to do, using `DrawString()` method. The following overload draws a string with the specified **Font** at the location specified by oPoint, and paints it with a **Brush** object (again any of the Brush types mentioned above):

```
METHOD DrawString(cText AS STRING, oBrush AS Brush, oFont AS Font, oPoint AS Point) ;
            AS VOID CLASS Graphics
```

Font class

A Font object can be instantiated in many ways. The simplest Font class constructor requires a font name and a font size (as a float number) :

```
CONSTRUCTOR (cFontName AS STRING, fSize AS FLOAT) CLASS Font
```

An exmple on creating a font:

```
oFont := Font{ 'Arial' , 10.0 } // create an "Arial" font with a 10 point size
```

Now let's see how we can draw a simple string on our form:

```
METHOD OnPaint(e AS PaintEventArgs) AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    LOCAL oBrush AS SolidBrush
    LOCAL oFont AS Font
    oGraphics := e:Graphics

    oBrush := SolidBrush{ Color.Blue }
    oFont:=Font {'Arial',15.0}

    oGraphics:DrawString ('Drawing some text' , oFont , oBrush , Point{50,50} )
RETURN
```

FontStyle

There are many situations when we need to print text in **Bold**, with *Italic* style, or underlined. In order to make use of such text drawing styles, we need to use another Font class constructor:

```
CONSTRUCTOR (cFontName AS STRING, fSize AS FLOAT, eStyle AS FontStyle) CLASS Font
```

FontStyle is an enumeration that has the following members:

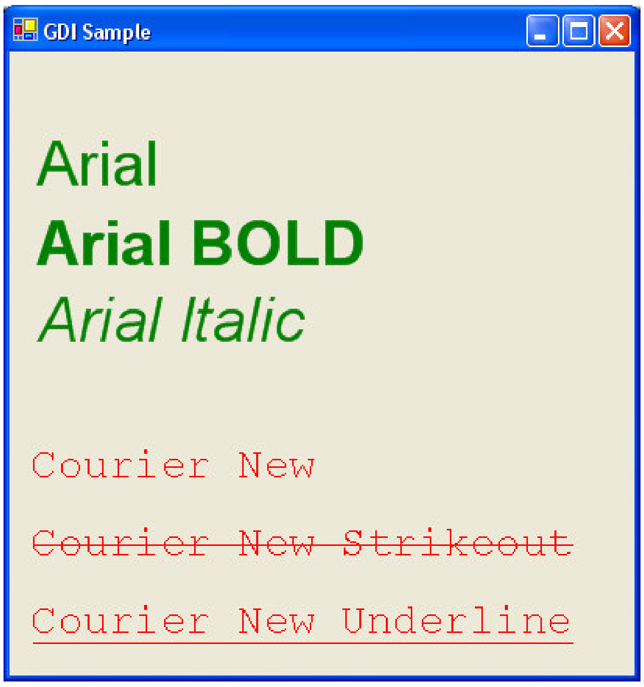
- Regular (the default)
- Bold
- Ialic
- Underline
- Strikeout

Font styles can be mixed together to define more complex styles:

```
oFont := Font{ 'Courier New' , 12 , FontStyle.Bold + FontStyle.Italic }
// create a "Courier New" font with a 12 point size and Bold,Italic style
```

The following sample draws text on our form with different font types, colors and styles:

```
METHOD OnPaint(e AS PaintEventArgs) AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    LOCAL oBrush AS SolidBrush
    LOCAL oFont AS Font
    oGraphics := e:Graphics

    oBrush := SolidBrush{ Color.Green }

    oFont:=Font {'Arial',30.0}
    oGraphics:DrawString ('Arial' , oFont , oBrush , Point{10,50} )
    oFont:=Font {'Arial',30.0, FontStyle.Bold}
    oGraphics:DrawString ('Arial BOLD' , oFont , oBrush , Point{10,100} )
    oFont:=Font {'Arial',30.0, FontStyle.Italic}
    oGraphics:DrawString ('Arial Italic' , oFont , oBrush , Point{10,150} )

    oBrush := SolidBrush{ Color.Red }

    oFont:=Font {'Courier New',20.0}
    oGraphics:DrawString ('Courier New' , oFont , oBrush , Point{10,250} )
    oFont:=Font {'Courier New',20.0, FontStyle.Strikeout}
    oGraphics:DrawString ('Courier New Strikeout' , oFont , oBrush , Point{10,300} )
    oFont:=Font {'Courier New',20.0, FontStyle.Underline}
    oGraphics:DrawString ('Courier New Underline' , oFont , oBrush , Point{10,350} )
RETURN
```



Important note: in the code snippets we have seen so far, we used local variables inside `OnPaint()` method to hold our GDI+ objects (fonts, brushes, pens etc). This is helpful for keeping the samples small and easy to follow, but it is not good programming practice; every time `OnPaint()` is called, those objects are created and destroyed when they get out of scope, resulting to resource and performance issues. In a real-life application, GDI+ objects should be generally declared at the class level (EXPORTs, PROTECTs etc) and created only once.

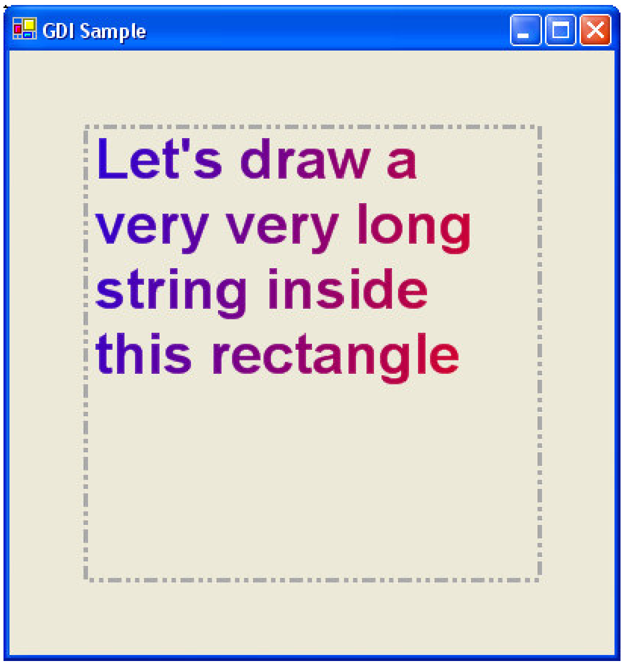
Drawing text inside a rectangle

Text may also be drawn inside a virtual rectangle, by using another overload of the `DrawString()` method:

```
METHOD DrawString(cText AS STRING, oBrush AS Brush, oFont AS Font, oRect AS Rectangle) ;
           AS VOID CLASS Graphics
```

This version requires a **Rectangle** object instead of a **Point**. It draws text starting from the upper-left corner of the rectangle and wraps text when it reaches its right side. The next sample showcases this functionality; it creates a **Rectangle** object that is used to draw a rectangle on the form and then draws a long string inside the same rectangle. Just in order to make the result a bit fancier, text is drawn using a **LinearGradientbrush**:

```
METHOD OnPaint(e AS PaintEventArgs) AS VOID CLASS SampleForm
    LOCAL oGraphics AS Graphics
    LOCAL oRectangle AS Rectangle
    LOCAL oPen AS Pen
    LOCAL oBrush AS LinearGradientBrush
    LOCAL oFont AS Font
    LOCAL cText AS STRING
    oGraphics := e:Graphics

    cText := "Let's draw a very very long string inside this rectangle"
    oBrush := LinearGradientBrush{ Point { 0 , 0 } , Point { 400 , 100 } ,;
                              Color.Blue , Color.Red }
    oPen  := Pen{ Color.DarkGray}
    oPen:Width := 3
    oPen:DashStyle := DashStyle.DashDotDot

    oRectangle := Rectangle{ 50 , 50 , 300 , 300 }

    oGraphics:DrawRectangle( oPen , oRectangle )

    oFont:=Font{ 'Arial', 28.0 , FontStyle.Bold }
    oGraphics:DrawString( cText , oFont , oBrush , oRectangle )
RETURN
```



Let's draw a very very long string inside this rectangle

Drawing Images

The **Graphics** class includes a very powerful method for drawing images, named, as one would expect, `DrawImage()`. This method has literally dozens of overloads and in its simplest version it requires an **Image** object and a (x,y) coordinate pair, specifying the position where the image should be drawn (in its original size):

```
METHOD DrawImage(oImage AS Image, x AS INT, y AS INT) CLASS Graphics
```

Bitmap class

Image is an abstract class, which again means that it can not be instantiated directly; instead we will use its most important subclass, **Bitmap**, which encapsulates a bitmap image. We usually instantiate a Bitmap object by loading a bitmap file from disc, using the following constructor method:

```
CONSTRUCTOR (cFileName AS STRING) CLASS Bitmap
```

The cFileName parameter must contain the (fully or partially qualified) filename of a bitmap file. GDI+ supports all commonly used file formats, like .bmp, .tiff, .jpg, .png etc:

```
oBitmap := Bitmap{ ‘C:\PICTURES\MyPhoto.bmp’} // fully specified filename

oBitmap := Bitmap{ ‘MyPhoto.jpg’}   // bitamp file is loaded from the current folder
```

Putting the pieces together, let’s see a sample `OnPaint()` method, drawing an Image on the form:

```
Method OnPaint(e As PaintEventArgs) As Void Class SampleForm
    Local oBitmap As Bitmap

    oBitmap := Bitmap{'image.jpg'}

    e:Graphics:DrawImage( oBitmap , 10 , 10)
Return
```

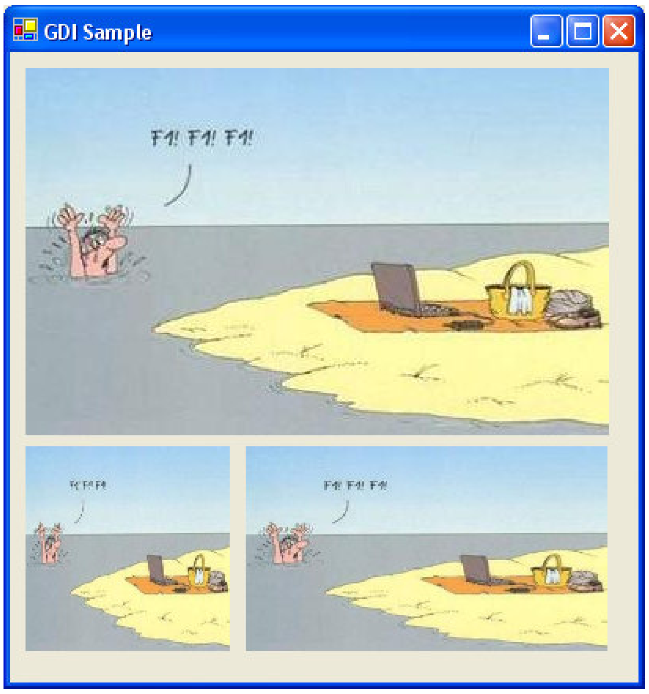
This simply draws a JPG image loaded from disc, in its original size. The Image can also be drawn in a different size (scaled or stretched), using another `DrawImage()` overload:

```
METHOD DrawImage(oImage AS Image, x AS INT, y AS INT, width AS INT, height AS INT) ;
    CLASS Graphics
```

The following full sample draws three times the same Image on the form; once in its original size and two times scaled in different sizes. It also demonstrates using a class variable for holding the image, loading it from disc every time `OnPaint()` is called (in our previous sample) was obviously very bad practice:

```
References "System.Windows.Forms"
References "System.Drawing"

Using System.Windows.Forms
Using System.Drawing

Function Start() As Void
    Local oForm As SampleForm
    oForm:=SampleForm{}
    oForm:Show()
    Application.Run(oForm)
Return
```

```
Class SampleForm Inherit Form
    Protect oBitmap As Bitmap

Constructor() Class SampleForm
    Super()
    Self:Text := 'GDI Sample'
    Self:oBitmap := Bitmap{'image.jpg'}
    Self:ClientSize := Size{400,400}
Return

Method OnPaint(e As PaintEventArgs) As Void Class SampleForm

    e:Graphics:DrawImage( Self:oBitmap , 10 , 10)

    e:Graphics:DrawImage( Self:oBitmap ,  10 , 250 , 130 , 130 )

    e:Graphics:DrawImage( Self:oBitmap ,  150 , 250 , 230 , 130 )

Return
```



Conclusion

This paper has presented the basic GDI+ features and demonstrated how to use them with Vulcan.NET. However, there is still a huge part that has been left uncovered. Hopefully this paper will achieve its goal, acting as a staring point for you to continue exploring the amazing world of GDI+.