Using GDI+ on Windows Mobile

Alex Feinman
OpenNETCF Consulting
October 2007

What is GDI+

GDI+ is a library created to add rich drawing and imaging feature to the basic set of Windows APIs. It has been introduced in Windows 2000 timeframe and eventually became a basis of System.Drawing and System.Drawing.Imaging namespaces. GDI+ is a dual API. It has a set of rich objects, such as Graphics, Bitmap, Pen, Brush etc as well, as a flat API suitable for use in legacy C or for P/Invoking. Starting with Windows Mobile 5 (Pocket PC), GDI+ ships on Windows Mobile devices. Despite the fact that the desktop .NET drawing and imaging support is built on top of GDI+, the Compact Framework does not use it. This in part explains somewhat more Spartan selection of graphic features available in Compact framework. Of course one of the main reasons for this was the lack of universal availability of GDI+. E.g it is not present on 2003 and 2003 SE devices, any version of smartphone (including Windows Mobile 6 Standard) or basic Windows CE.

It is also important to understand that Windows Mobile version of GDI+ lacks many features of its desktop counterpart. Although it is 100% API-compatible, a lot of calls is simply not implemented. All it does is returning the corresponding status (NotImplemented) to the caller.

Why GDI+

As I understand it, the need for GDI+ has been driven by inking support. Transcriber (an Input Method supporting cursive recognition) and in WinMo 6 WISPLite (inking and recognition support) both use GDI+ internally.

GDI+ allows drawing with antialiasing, transparency and some limited 2D transformation support. Currently, there is no other way to draw an antialiased polyline, curve or other object in Windows CE, than by using GDI+.

GDI+ supports Rectangles, Pies, Ellipses, Beziers as part of a Path object. It also supports various special types of pens and brushes including gradients.

Things notably missing from Windows Mobile version of GDI+ are:

- Text support
- World Transform support. In general transforms are only supported on a Path object
- Many image operations

From the performance standpoint GDI+ is a slow-ish API. It needs to be used judiciously and sparingly. Caching complex objects as bitmaps also helps.

How

API access

GDI+ API consists of 2 parts. One is GDIPLUS.DLL, which exports so-called flat API – a large number of C functions that comprise the entire functional set of the API. The other is a set of C++ objects, found in the include directory of Platform SDK, which encapsulates flat API functionality, provides memory management, cleanup and some useful shortcuts. When using GDI+ from Compact Framework, it is relatively easy to P/Invoke the flat API. The class-based API is a different thing. It is absolutely, 100% incompatible with Compact Framework. It can however be recreated in C# and given its usefulness, it should be.

Caution needs to be exercised, when using GDI+. The code needs to make sure that the platform indeed has GDIPLUS.DLL and be prepared to deal with the lack of it, for example by falling back to less rich, traditional methods of drawing.

Session initialization and termination

Before GDI+ can be used, a call to “GdiplusStartup” needs to be made:

```
        GdiplusStartupInput input = new GdiplusStartupInput();
        GdiplusStartupOutput output;
        GpStatus stat = NativeMethods.GdiplusStartup(out token, input, out output);
```

The above code initializes GDI+ subsystem. The parameters can be ignored – there is not much use to them unless you have a debug build of GDIPLUS.DLL, which is not likely. This is also a good place to add try/catch and watch for MissingMethodException. Getting one would typically indicate that your platform does not have GDIPLUS.DLL available.

When the application is shutting down (or GDI+ is not needed anymore), the following call is required:

```
        NativeMethods.GdiplusShutdown(token);
```

Following this call no other GDI+ calls are allowed.

Managed wrapper

To be able to use GDI+ in a Compact Framework application, we need to wrap most of the flat API and port the object model. This is done for you in OpenNETCF.GDIPlus.dll assembly. It has a class called NativeMethods, where all of the flat API is exposed. It also has objects matching GDI+ objects. Among those are:

- GraphicsPlus

- ImagePlus
- BitmapPlus
- GraphicsPath
- BrushPlus and its derivative classes:
    - SolidBrushPlus
    - HatchBrush
    - LinearGradientBrush
    - PathGradientBrush
    - TextureBrushPlus
- PenPlus
- RegionPlus

Drawing with GDI+

All drawing in GDI+ is done using GraphicsPlus object, similar to how in .NET all drawing is done using Graphics object. In order to obtain GraphicsPlus object an appropriate constructor needs to be used. For example to draw inside OnPaint override (or wherever .NET Graphics object is available), one would use a constructor that takes HDC:

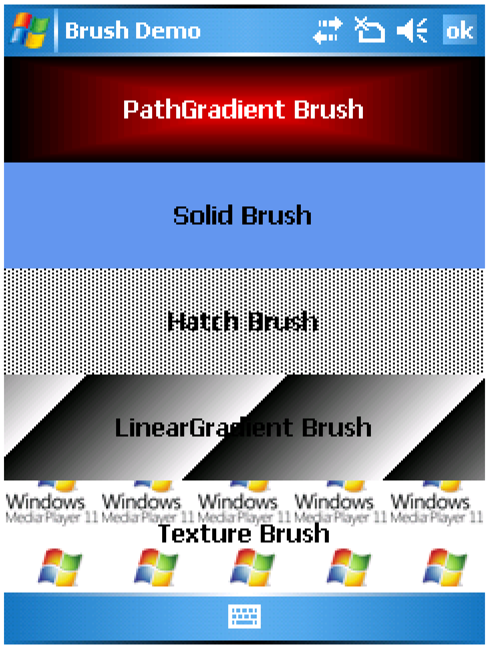
```
protected override void OnPaint(PaintEventArgs e)
{
    IntPtr hdc = e.Graphics.GetHdc();
    using(GraphicsPlus g = new GraphicsPlus(hdc))
        PaintControl(g);
    e.Graphics.ReleaseHdc(hdc);
}
```

If the drawing target is an off-screen bitmap, then there is a choice of creating a .NET Graphics object on it followed by the above, or using Bitmap.GetHbitmap() and then constructing GraphicsPlus object on HBITMAP. There are some other choices, but mostly of limited use.

While its desktop peer supports many settings, the Windows Mobile version of Graphics object supports only one – smoothing mode. It can be sent to None, Antialias, High Speed and High Quality. Naturally, antialised output takes the longest time to render.

Brushes

Brush objects in GDI+ can be used to draw directly on the screen by filling various shapes as well as to create pens that use brush textures to fill the line. Below are several brush examples:



In order to create the above brushes the following code was used:

1. PathGradient brush

```
// Create rectangular path
GraphicsPath path = new GraphicsPath(FillMode.FillModeAlternate);
path.AddRectangle(new GpRectF( 0, 0, ClientRectangle.Width,
    ClientRectangle.Height / 5));

// Create rectangular gradient brush
// with red in center and black in the corners
brPathGrad = new PathGradientBrush(path);
brPathGrad.SetCenterColor(Color.Red);
int count = 2;
brPathGrad.SetSurroundColors(new Color[] { Color.Black, Color.Black },
    ref count);
```

2. Solid Brush

```
brSolid = new SolidBrushPlus(Color.CornflowerBlue);
```

3. Hatch Brush

```
brHatch = new HatchBrush(HatchStyle.HatchStyle25Percent,
    Color.Black, Color.White);
```

4. Linear Gradient

```
brLinGrad = new LinearGradientBrush(new GpPointF(0, 0),
    new GpPointF(50, 50), Color.Black, Color.White);
```

5. Texture brush

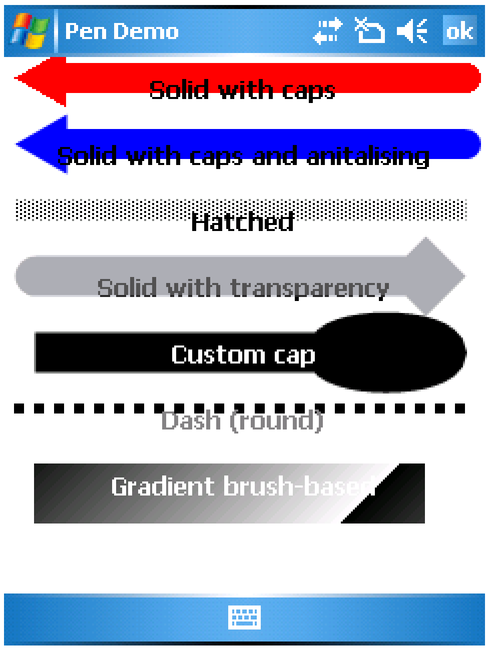
```
StreamOnFile sf = new StreamOnFile(bitmapPath);
ImagePlus img = new ImagePlus(sf, false);
brTexture = new TextureBrushPlus(img, WrapMode.WrapModeTile);
```

Pens

Pens in GDI+ support thickness, transparency, antialiasing, custom caps, and some other features. In addition a pen can be created solid, hatched and generally based on any brush one can create. Using pens it is possible to draw lines, rectangles, ellipses, beziers and otherwise random paths.

Pens support start and end caps. Caps can be predefined or entirely custom, based on a path object.

Below is a sample of the output from various pens used under various conditions:



Here is how each pen has been created and used:

1. Solid with caps. Standard caps are used – round and arrow

```
brSolid = new SolidBrushPlus(Color.CornflowerBlue);
penSolid = new PenPlus(Color.Red, 10);
penSolid.SetEndCap(LineCap.LineCapRound);
penSolid.SetStartCap(LineCap.LineCapArrowAnchor);
```

2. Solid with caps and antialiasing. This one is the same as before except it is drawn with antialiasing

```
g.SetSmoothingMode(SmoothingMode.SmoothingModeAntiAlias);
```

```
penSolid.SetColor(Color.Blue);
g.DrawLine(penSolid, 5, rcf.Top + 10, rc.Width - 10, rcf.Top + 10);
```

3. Hatched (25%)

```
brHatch = new HatchBrush(HatchStyle.HatchStyle25Percent,
    Color.Black, Color.White);
penHatch = new PenPlus(brHatch, 10);
```

4. Solid with transparency

```
penSolidTrans = new PenPlus(Color.FromArgb(-0x5f7f7f7f), 10);
```

5. Custom cap. The custom cap has been created out of a path object consisting of a single ellipse

```
penSolidCustomCap = new PenPlus(Color.Black, 20);
GraphicsPath path = new GraphicsPath(FillMode.FillModeAlternate);
path.AddEllipse(-0.5f, -1.5f, 1, 3);
CustomLineCap cap = new CustomLineCap(null,path, LineCap.LineCapFlat, 0);
penSolidCustomCap.SetCustomEndCap(cap);
```

6. Dash

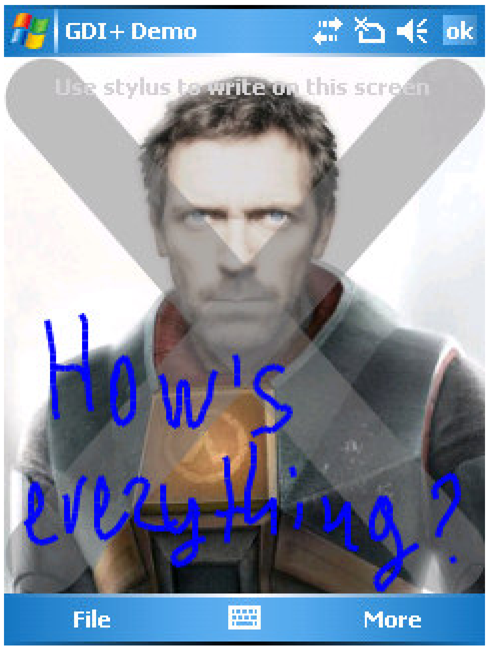
```
penDash = new PenPlus(Color.Black, 5);
penDash.SetDashStyle(DashStyle.DashStyleDot);
```

7. Gradient brush-based

```
brGrad = new LinearGradientBrush(
    new GpPointF(0, 0), new GpPointF(100, 100),
    Color.Black, Color.White);
penGradient = new PenPlus(brGrad, 30);
```

Demo application

This whitepaper includes a demo application. The application demonstrates the above techniques as well as shows how to use GDI+ to capture ink with smoothing and antialiasing.



Conclusion

GDI+ is available on modern Windows Mobile platform that support stylus input. It allows creating graphic applications that are hard or impossible to implement otherwise. Using GDI+ in conjunction with Compact .NET Framework allows a developer to create a rich graphic application without expending substantial effort.