Drawing

For this chapter you will need the following name spaces:

```
using System;
using System.IO;
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;
using System.Drawing.Text;
```

Creating a Metafile or Bitmap

To start a drawing one needs a Graphics object to draw upon. This Graphics object can be obtained from several sources, an existing Metafile or Bitmap file could be one source for it. We could also obtain it from an object in our application that has a Graphics object, if the intension is to draw to its Graphics object. Here we will start by creating a Metafile in memory. The following code creates a Matafile “mf”and then obtains its graphics object “g” from it:

```
Graphics gTemp = CreateGraphics();
IntPtr ipHdc = gTemp.GetHdc();
int PlotWidth = 300;
int PlotHeight = 300;
Metafile mf = new Metafile(new MemoryStream(),ipHdc, new
      Rectangle(0,0, PlotWidth , PlotHeight),
      MetafileFrameUnit.Pixel,EmfType.EmfPlusDual,"Plot");
gTemp.ReleaseHdc(ipHdc);
gTemp.Dispose();
Graphics g = Graphics.FromImage(mf);
```

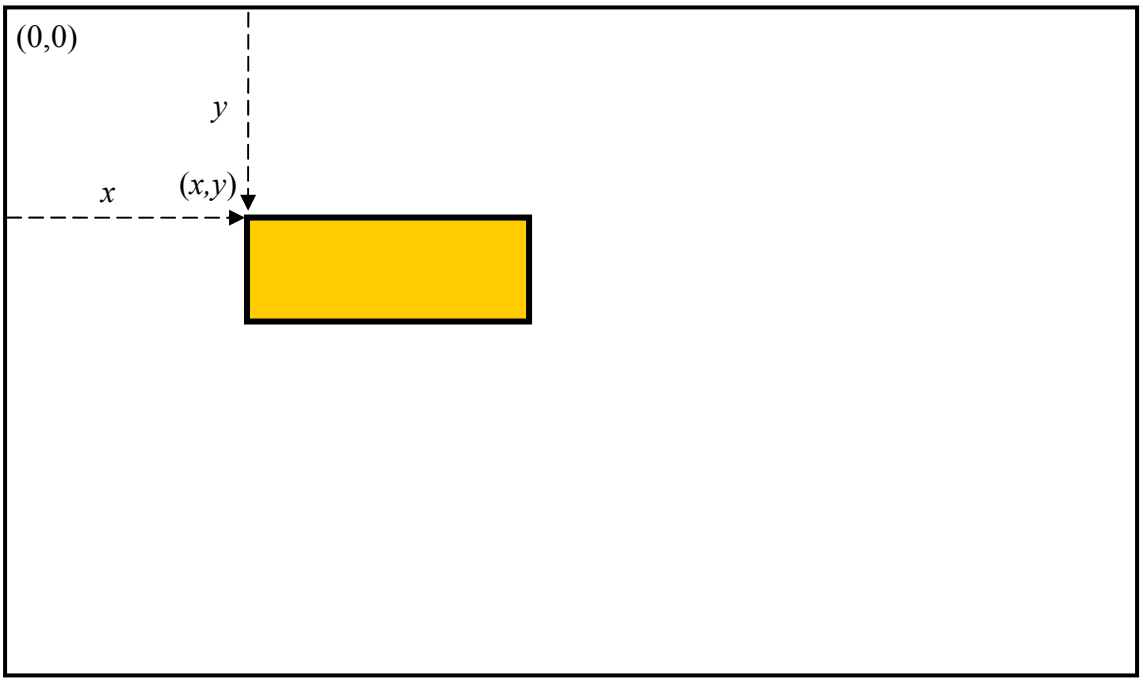
Not explaining all that was done, it can be seen that a Graphics “gTemp” was created, from which we obtained the Hdc, created the Metafile, and then disposed of “gTemp” and then got the Graphics “g” of “mf” to draw on.

The creation of a Bitmap is simpler, but the Metafile is normally a better method of creating and storing graphics files. The Bitmap does not scale well since as we increase the scaling the Bitmap starts of appear coarse. This is because the Bitmap stores the graphics at a fixed resolution. The Metafile, being a vector format, stores the objects through their descriptions and renders them each time. This makes the Metafile drawing look sharp at all magnifications. To create a Bitmap “bm” of 300x300 and get its Graphics object “g” we can use the following code:

```
int PlotWidth = 300;
int PlotHeight = 300;
Bitmap bm = new Bitmap(PlotWidth, PlotHeight);
Graphics g = Graphics.FromImage(bm);
```


Positioning of objects

When drawing on the Graphics object it should be understood that the coordinates are measured from the top left corner of the Graphics object and dimensions are measured down for the height and to the right for the width. As shown in the drawing, objects drawn on the Graphics are positioned based on the position of their top left corner relative to the top left corner of the Graphics object.



Drawing on the Graphics object

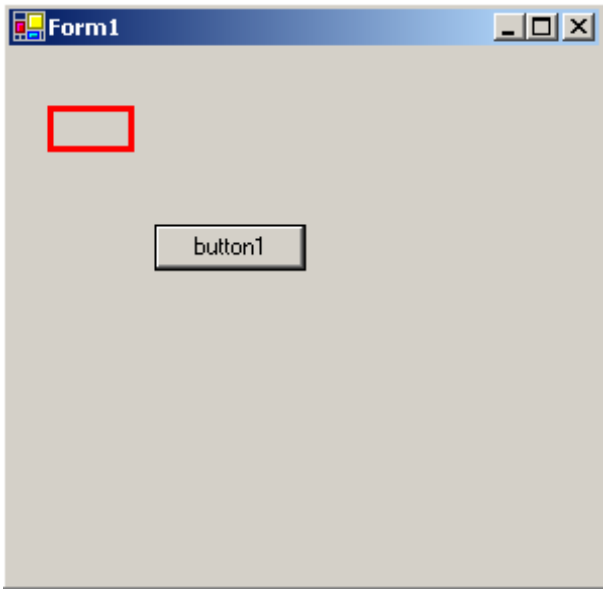
To draw an object such as a rectangle on the Graphics object you need a Pen to draw with. The following tells the Graphics object "g" to draw a rectangle of width 40 and height 20 with a red pen of thickness 3

```
Pen myPen = new Pen(Color.Red, 3);
int TopLeftCornerX = 20;
int TopLeftCornerY = 30;
int Width = 40;
int Height = 20;
g.DrawRectangle(myPen, TopLeftCornerX, TopLeftCornerY, Width , Height);
g.Dispose();

this.BackgroundImage = mf;
```

Note that after the drawing is complete, the Graphics needs to be Disposed. In this example, the Metafile is assigned to the BackgroundImage. If you put this in the Click

event of the button on your application, it will draw a rectangle on your window as follows:



Saving Metafiles and Bitmaps

The Metafile and Bitmap objects have a Save method that allows saving to a file. For example, to save our Metafile to the file "myMetafile.wmf" in the current directory we could use the following command line:

```
mf.Save("myMetafile.wmf");
```

If we do so, the following picture would be stored:

Note that the picture is as large as the space shown, but since it has no boarders and no background color, we cannot see it.

Getting the Graphics object from a file

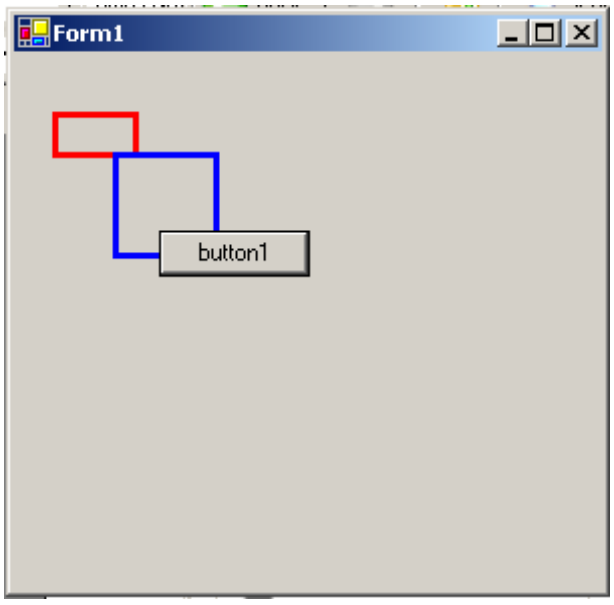
In place of creating a Metafile or Bitmap, and then getting the Graphics object, one can get the Graphics object directly from an existing Metafile or Bitmap file. For example, the following code gets a Bitmap "bm" from the image "myMetafile.wmf" and draws a new rectangle to it, and then sets it as the background image of the form.

```
Bitmap bm = new Bitmap("myMetafile.wmf");
Graphics g = Graphics.FromImage(bm);

Pen myPen = new Pen(Color.Blue, 3);
int TopLeftCornerX = 50;
int TopLeftCornerY = 50;
int Width = 50;
int Height = 50;
g.DrawRectangle(myPen, TopLeftCornerX, TopLeftCornerY, Width , Height);
g.Dispose();

this.BackgroundImage = bm;
```

The result of this would be for our application to draw the two rectangles as follows:



Pens and Brushes

Pens and brushes are fundamental to making drawings as pens are used to draw lines and brushes are used to fill areas.

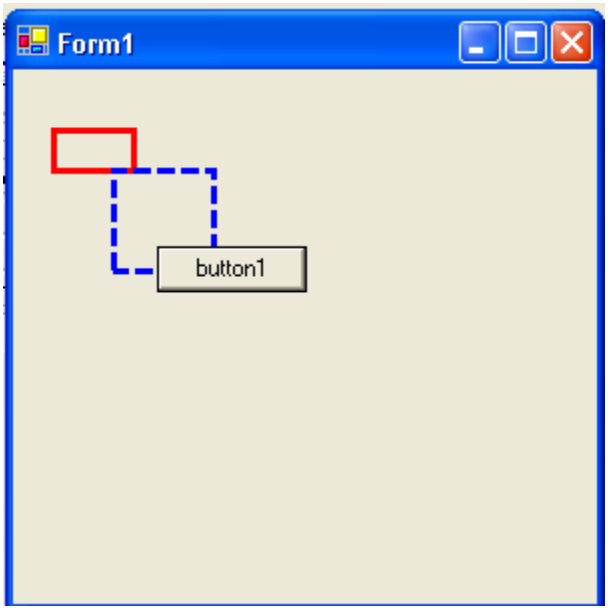
We have already seen the use of pens in our two examples. A typical declaration of a pen includes a color, a line width and a line type. The color and line width can be declared in the Pen constructor. A typical declaration and instantiation of the Pen would be as follows:

```
Pen myPen = new Pen(Color.Blue, 3);
```

This will create an instance of Pen named "myPen" which will draw blue lines of width three. In addition, one of a number of line types can be selected for myPen through command such as:

```
myPen.DashStyle = DashStyle.Dash;
```

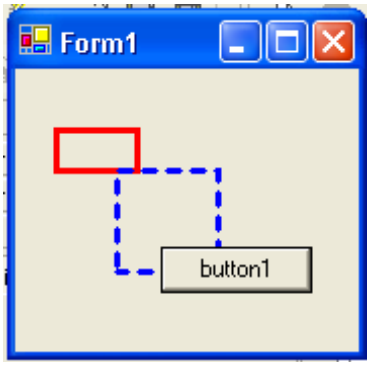
This sets the DashStyle to one of several line styles including: Custom, Solid, Dash, DashDot, DashDotDot, Dot. If you change the DashStyle to Dash before drawing the rectangle in the example, the result will be the following:



Also, the dash cap can be set to: Flat, Round, and Triangle. The command for this would be:

```
myPen.DashCap = DashCap.Triangle;
```

which would result in the following result:

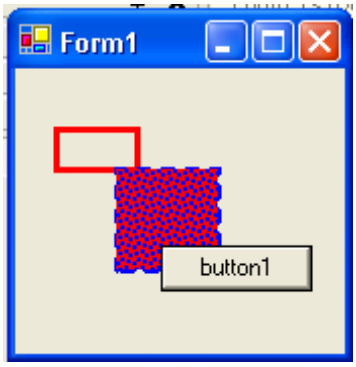


Pens can also be constructed from brushes. The brush is to fill an object and a pen constructed in this way will expose the fill in the places that a line is drawn.

The Brush class itself is an abstract class so it cannot be instantiated. Classes derived from it are the SolidBrush, HatchBrush, TextureBrush, LinearGradientBrush, and PathGradientBrush. To instantiate a brush you need to instantiate it as one of these types of brushes. A typical Brush definition and construction can be seen in the following example in which a brush "myBrush" is defined with a Large Confetti pattern constructed from a foreground color blue and background color red. This is then used to fill the rectangle drawn in the last example.

```
Color forColor = Color.Blue;
Color backColor = Color.Red;
Brush myBrush = new HatchBrush(HatchStyle.LargeConfetti, forColor, backColor);
g.FillRectangle(myBrush, TopLeftCornerX, TopLeftCornerY, Width , Height);
```

If these lines are added to the last example, the following will result:



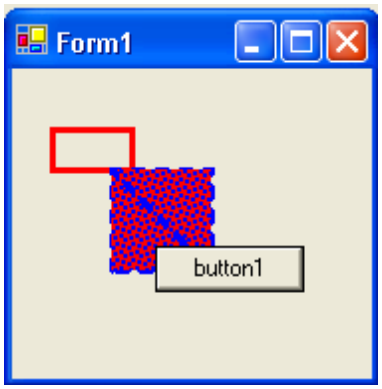
Drawing Shapes

From the graphics object of an image we get a set of shapes that can be drawn. These are: DrawArc, DrawBezier, DrawBezier, DrawClosedCurve, DrawCurve, DrawEllipse, DrawIcon, DrawImage, DrawLine, DrawLines, DrawPath, DrawPie, DrawPolygon, DrawRectangle, DrawRectangles, DrawString.

We have already seen a typical declaration to draw a rectangle. To draw a line along the diagonal of our rectangle we can use the following commands:

```
int point1X =TopLeftCornerX;
int point1Y =TopLeftCornerY;
int point2X =TopLeftCornerX+Width;
int point2Y =TopLeftCornerY+Height;
g.DrawLine(myPen, point1X, point1Y, point2X , point2Y);;
```

The result of adding this to our program would be:



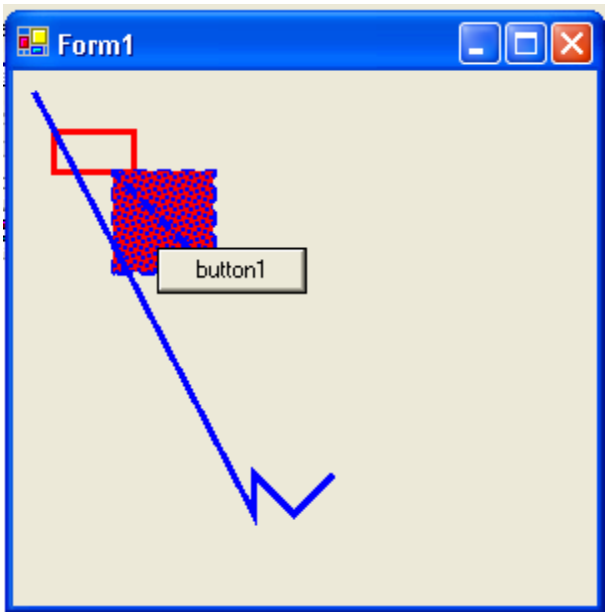
This same result can be obtained by using a Point in place of declaring x and y components of each point. In this case we could use the commands:

```
Point point1 = new Point(TopLeftCornerX, TopLeftCornerY);
Point point2 = new Point(TopLeftCornerX+Width, TopLeftCornerY+Height);
g.DrawLine(myPen, point1, point2);
```

The Point allows us to chain a set of points together as an array. This is useful when drawing multiple lines as in the following example:

```
Point[] points = new Point[5];
points[0] = new Point(10,10);
points[1] = new Point(120,220);
points[2] = new Point(120,200);
points[3] = new Point(140,220);
points[4] = new Point(160,200);
myPen.DashStyle = DashStyle.Solid;
g.DrawLines(myPen, points);
```

The result of this is to draw a solid line connecting the points to give the result:



A variation of Point is PointF that is similar but allows using floating point values to be declared for the coordinates in place of integers. PointF can in many places be used in place of Point.

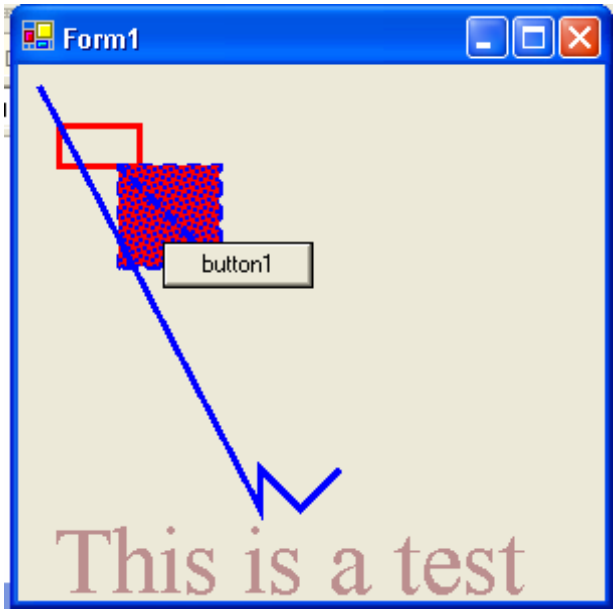
Filling drawing object is done in the same way as described in the above example.

Writing text

Text can be put on a picture. To write text one needs to select the FontFamily and size, define a Font, select a Brush, and position the top left corner. This is done in the following code:

```
FontFamily myFontFamily = new FontFamily("Times New Roman");
int myFontSize = 35;
Font myFont = new Font(myFontFamily, myFontSize);
Point myPoint = new Point(10,220);
myBrush = new SolidBrush(Color.RosyBrown);
g.DrawString("This is a test", myFont, myBrush, myPoint);
```

The result of this will be

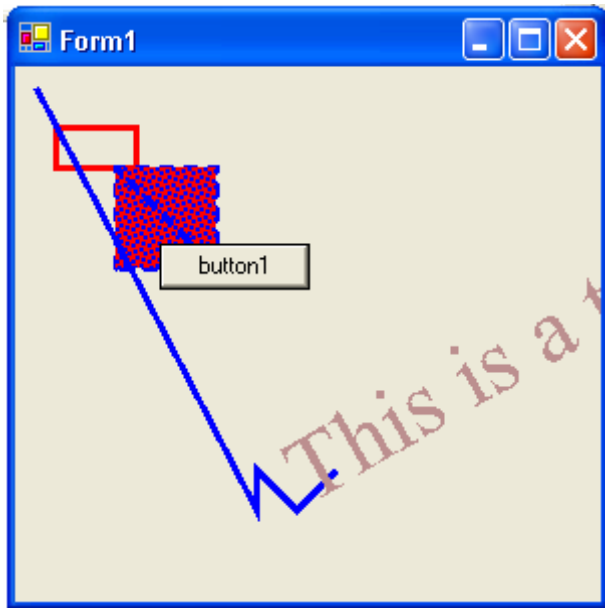


Translation and rotation

Rotation and translations can be done be using TranslateTransform and RotateTransform before the point in the code that a rotation or translation is to be done. For example, if the text on the last example is to be rotated about the origin an angel of 30 degrees counter clockwise, it can be done using the code

```
FontFamily myFontFamily = new FontFamily("Times New Roman");
int myFontSize = 35;
Font myFont = new Font(myFontFamily, myFontSize);
Point myPoint = new Point(10,220);
myBrush = new SolidBrush(Color.RosyBrown);
g.RotateTransform(-30);
g.DrawString("This is a test", myFont, myBrush, myPoint);
```

The result of this can be seen in the following:



It is important to note that the rotation does not affect objects that are drawn before the RotateTransform method, but all objects that are drawn after calling this method are rotated. For this reason, if the rotation is to be applied only to one part of the drawing, it is necessary to use RotateTransform to rotate back after the completion of the drawing of all the parts that need to be rotated.

Further drawing methods

There is a lot of drawing tools that cannot be described here due to the limited scope of these notes. A good reference for drawing is “Programming Microsoft Windows with C#” by Charles Petzold, Microsoft Press, 2002.