Colors

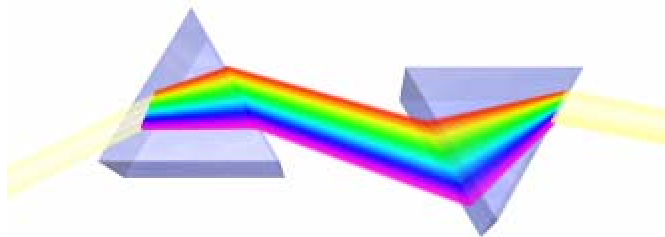
You can go for a long time in .NET programming and only use the colors provided as constants in the *Color* class. But I believe, as professional programmers, you should have a deeper understanding of the use of colors on a computer screen.

Review of third-grade physics. I was going to say eighth-grade, but search the Web for "prism Newton light spectrum"--I found the prism experiment for third-graders along with the following wonderful song:

Sing to the tune of "The Muffin Man" from Carson-Dellosa's *Hands-On Science Fair*
Oh, do you know the Spectrum Man,
The Spectrum Man,
The Spectrum Man?
Oh, do you know the Spectrum Man?
His name is Roy G. Biv.

[The source of this song was a web page that is no longer valid, so now I guess it is folklore..]

White light is made up of colored light. Newton discovered in 1666 that white light could be broken up by a prism into colors. The colors could then be put back together into white light again.



At http://micro.magnet.fsu.edu/primer/java/scienceopticsu/newton/ you can see a Java animation of Newton's experiment.

Further research showed that colors can be described by three numbers. The reason is that there are just three color-sensitive pigments in the human retina. A convenient choice of three numbers is the components of red,

green, and blue. Note that these are NOT the primary colors (pigments) you learned as a child. Colors of light combine differently than colors of pigment.

Red + blue = magenta
Blue + green = cyan
Red + green = yellow
Red + blue + green = white
Zero of red, blue, and green = black.

In .NET, or rather, "under the hood" of .NET, a 32-bit integer is used to code a color. There are 8 bits each for red, green, and blue, and 8 bits for a "alpha channel" (discussed below). You can construct an arbitrary color by either of these two constructors:

```
Color c = FromArgb(int a, int r, int g, int b);
Color c = FromArgb(int r, int g, int b);   // no alpha specified, opaque by
default
```

```
FromArgb(255,0,0)          pure, bright red
FromArgb(0,255,0)          pure, bright green
FromArgb(0,0,255)          pure, bright blue

FromArgb(255,255,0)        pure, bright yellow

FromArgb(0,0,0)            black
FromArgb(255,255255)       white
FromArgb(200,200,0)        pale yellow
FromArgb(128,128,128)      grey

FromArgb(50,50,50)         dark grey
FromArgb(200,200,200)      light grey

FromArgb(128,0,0)          dark red
FromArgb(255,128,0)        orange  (like yellow, but with more red in it)
```

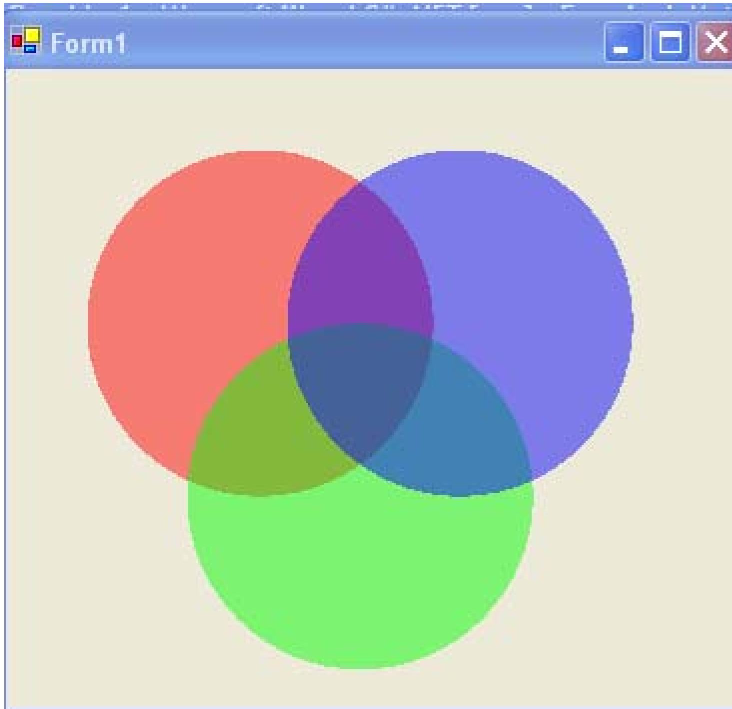
After a little practice, you'll find it easier to construct the exact color you want this way than by selecting from a long list of sometimes mysterious names. For example, what color is *Alice Blue* anyway? (It was named, incidentally, for the daughter of President Theodore Roosevelt, whose favorite color was blue.)

Transparent Colors

In computer graphics people speak of an "alpha channel". This is an 8-bit value governing the "transparency" of a color. 255 means fully opaque, 0 means fully transparent. The new graphics library GDI+ that come with .NET (and replaces the old GDI of Windows) supports colors with an alpha channel; this was never available in Windows before .NET. Here's the example from the Microsoft documentation. The first number passed to the constructor of the brushes is the alpha parameter, 120, about halfway between fully transparent and opaque.

```
{ Graphics g = e.Graphics;
  // Transparent red, green, and blue brushes.
  SolidBrush trnsRedBrush = new
SolidBrush(Color.FromArgb(120, 255, 0, 0));
  SolidBrush trnsGreenBrush = new
SolidBrush(Color.FromArgb(120, 0, 255, 0));
  SolidBrush trnsBlueBrush = new
SolidBrush(Color.FromArgb(120, 0, 0, 255));
  // Base and height of the triangle that is used to
position the
  // circles. Each vertex of the triangle is at the center
of one of the
  // 3 circles. The base is equal to the diameter of the
circles.
  float   triBase = 100;
  float   triHeight =
(float)Math.Sqrt(3*(triBase*triBase)/4);
  // Coordinates of first circle's bounding rectangle.
  float   x1 = 40;
  float   y1 = 40;
  // Fill 3 over-lapping circles. Each circle is a different
color.
  g.FillEllipse(trnsRedBrush, x1, y1, 2*triHeight,
2*triHeight);
  g.FillEllipse(trnsGreenBrush, x1 + triBase/2, y1 +
triHeight,
               2*triHeight, 2*triHeight);
  g.FillEllipse(trnsBlueBrush, x1 + triBase, y1,
2*triHeight, 2*triHeight);
}
```

Here's the output. You could not write this program in versions of Windows before .NET, since the alpha-channel is a feature of the GDI+ graphics library that wasn't in GDI graphics.



Lines and Polygons

So far, we have drawn only rectangles and ellipses. Now we'll draw lines and polygons.   As a demo program, we'll show *Polygon*. The main window shows a regular polygon with *n* sides. When you left-click in the polygon, the polygon is redrawn with *n+1* sides, up to some maximum number of sides, say 200. When you right-click in the polygon, the polygon is redrawn with *n-1* sides.   In other words, left-click increments the number of sides, and right-click decrements it. When you left-click or right-click outside the polygon, nothing happens—the click is ignored.

This program needs only two pieces of data:

```
private int m_nSides;    // number of sides
```

```
private int m_Radius;    // distance from center to each
vertex
```

Initialize them to 3 and 100, respectively.

Here's one way to paint the polygon, using the basic *Graphics* method *DrawLine*. Note that *DrawLine* takes a pen and two points—the obvious requirements for drawing a line! You can also use a pen and four integers, where the first two integers specify the starting point and the other two the ending point. Note also the use of various functions from the *Math* class; the necessity to pass *theta* in radians, and the need to close the polygon by connecting the last point to the starting point.

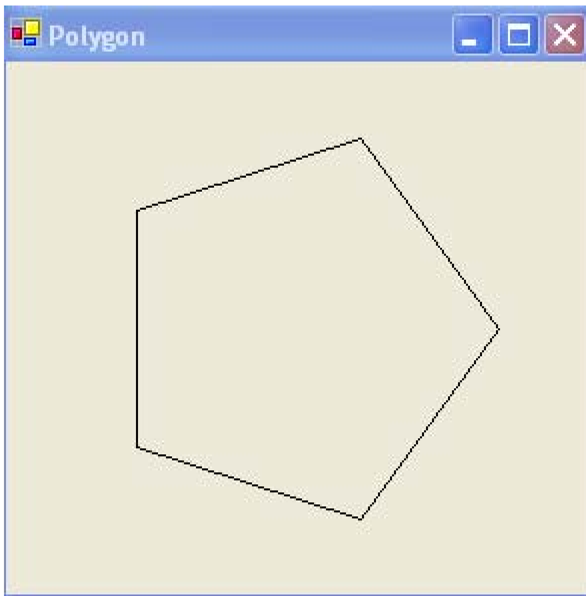
```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{ Pen p = new Pen(Color.Black);
  int i;
  Graphics g = e.Graphics;
  Point center,prev,next,start;
  center = new Point((ClientRectangle.Right +
ClientRectangle.Left)/2,
                     (ClientRectangle.Bottom +
ClientRectangle.Top)/2
                    );
  start = new Point(center.X + m_Radius, center.Y);
  prev = new Point();
  next = new Point();
  prev = start;
  for(i=1;i<m_nSides;i++)
     { double theta = i* 2* Math.PI / (float) m_nSides;
       next.X =  center.X + (int) Math.Round(m_Radius *
Math.Cos(theta));
       next.Y =  center.Y - (int) Math.Round(m_Radius *
Math.Sin(theta));
       g.DrawLine(p, prev,next);
       prev = next;
     }
  // Now close the polygon by connecting the last point to
the starting point
  g.DrawLine(p,prev,start);
}
```

To start with we can use the following simple *MouseDown* handler, which makes no effort to hit-test whether the click is inside the polygon or not:

```
private void Form1_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{ if(e.Button == MouseButtons.Left)
     ++m_nSides;
  else if(e.Button == MouseButtons.Right && m_nSides > 3)
     --m_nSides;
  Invalidate();
}
```

This program will fulfill the first two requirements, about what happens when you left-click or right-click inside the polygon, but it will not fulfill the third requirement, that clicks outside the polygon are to be ignored.

Here's a screen shot after two left-clicks:



OK, that's fine as far as it goes, but we haven't achieved the original goal, and doing so will require us to learn more GDI+ graphics, namely the *GraphicsPath* class and the *Region* class.

Here's another reason to learn about *GraphicsPath*: use a pen of width 15 pixels and see what the drawing looks like. To get a pen of width 15 pixels, use this constructor:

```
Pen p = new Pen(Color.Black,15);
```

You won't be happy with the appearance of your polygon! (Go on, type the code in and try it.)

A *GraphicsPath* object is essentially an array of *Point*, with each point labeled with a byte that tells how it should be connected to the previous point. The possible values of this byte include *PathPointType.Line*, which is what we need for drawing a polygon. Our improved program will have a member variable *m_thePolygon* of type *GraphicsPath*. We will use that object for hit-testing in the *MouseDown* handler, and for drawing in the *Paint* handler. The class *GraphicsPath* is not in the *System.Drawing* namespace, so we need a new command at the top of the file:

```
using System.Drawing.Drawing2D;
```

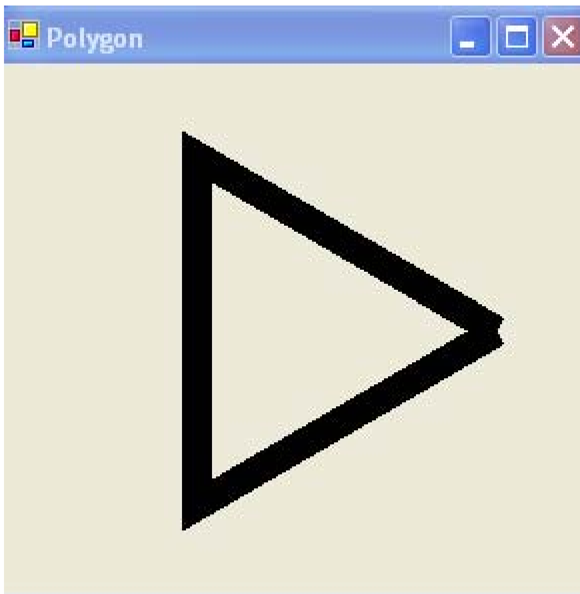
Here's the new *Paint* handler:

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{ Pen p = new Pen(Color.Black);
  int i;
  Graphics g = e.Graphics;
  Point center;
  center = new Point((ClientRectangle.Right +
ClientRectangle.Left)/2,
                     (ClientRectangle.Bottom +
ClientRectangle.Top)/2
                    );
  Point[] points = new Point[m_nSides+1];
  byte[] bytes = new byte[m_nSides+1];
  double theta;
  for(i=0;i<=m_nSides;i++)
```

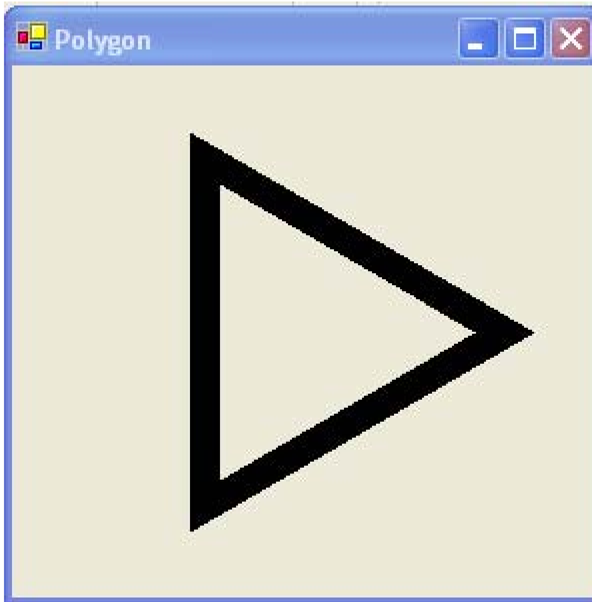
```
     { bytes[i] = (byte) PathPointType.Line;
       theta = i* 2* Math.PI / (float) m_nSides;
       points[i].X =  center.X + (int) Math.Round(m_Radius *
Math.Cos(theta));
       points[i].Y =  center.Y - (int) Math.Round(m_Radius *
Math.Sin(theta));
     }
  m_thePolygon = new GraphicsPath(points,bytes);
  g.DrawPath(p,m_thePolygon);
}
```

This time the loop goes up to *i<= m_nSides* so the last point is a copy of the starting point. We do the same work as before in computing the points, but we just store the coordinates in the *GraphicsPath* object. Then when we've computed all the points, we call *DrawPath.*

Replacing our old *Paint* handler with this one, we have the same functionality as before. You can't see any difference between *DrawPath* and the loop of *DrawLine* commands we had before. But try it with a pen of width 15 as before. It's much better, right? But still not perfect!
There's a problem where the last segment joins to the starting point:



You can fix this by extending the loop to *m_nSides+2*:



OK, so *GraphicsPath* helped with painting the polygon. Now for meeting requirement three about ignoring clicks outside the polygon and responding to clicks inside the polygon. We construct a *Region* from the *GraphicsPath*, and then, to test whether a point is in the given region, we use the *IsVisible* method of the *Region* class, which can accept a point (or in this case, two floats).

```
private void Form1_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{ Region r = new Region(m_thePolygon);
  if(r.IsVisible(e.X,e.Y) == false)
     return;  // ignore clicks outside the polygon
  if(e.Button == MouseButtons.Left)
     ++m_nSides;
  else if(e.Button == MouseButtons.Right && m_nSides > 3)
     --m_nSides;
  Invalidate();
}
```

Just for completeness: You can also pass a *Rectangle* instead of a *Point* to *IsVisible*. In that case, it tests whether any part of the rectangle is inside the region. The region constructor that takes a

GraphicsPath first closes the *GraphicsPath* object, if necessary, by connecting the last point to the first. In our example that wouldn't change the region anyway. Question: does *IsVisible* consider the points in the boundary of the polygon (the ones colored black) to be inside or outside the region? Run the program and find out. (That's the only way to find out, since the documentation of *IsVisible* is silent on that point.)

Regarding the design of the program: it's somewhat wasteful to be computing a new *GraphicsPath* object in *Form1_Paint*, and it also violates the principle that *Form1_Paint* should just present the data to the user, rather than compute the data. Well, in some sense the data is just *m_nSides* and *m_Radius*, and the *GraphicsPath* object is just a tool to present it. In fact, we could recompute it in *Form1_MouseDown* and then we wouldn't need *m_thePolygon* as a member variable. Alternately, we could keep *m_thePolygon* as a member variable, and recompute it only in *Form1_MouseDown* when the number of sides changes. Any of these alternate designs would work; the last suggestion is probably the best, but I didn't follow it here, because I wanted to show you the *Paint* handler first and only then introduce *Region*, so the way shown here seems to make it easier to explain the concepts required, even if it isn't the ideal final design.

More on Regions and GraphicsPath

The *GraphicsPath* constructor used above is only one way to create a *GraphicsPath* object. There are others that you should know. For example, you may want to create a circular or elliptical region—the above constructor won't do that job. There is no constructor that does it directly; instead, to create more complicated regions, you just call *new GraphicsPath()* with no arguments, and then you call various methods of the *GraphicsPath* class that add components to the path. For example, there is an *AddEllipse* method, which takes a rectangle

parameter. To create a circular region, you first create a square rectangle *square*, and then use this code:

```
GraphicsPath p = new GraphicsPath();
p.AddEllipse(square);
Region r = new Region(p);
```

There is also an *AddLine* method, which we could have used above to add lines repeatedly to an empty *GraphicsPath*, instead of using an array of points. There is also an *AddArc* method that can be used to add curved pieces to the path.

Subpaths and Figures

A *GraphicsPath* object is composed of subpaths. If the last point of the subpath connects to the starting point then the subpath is *closed.* A closed subpath is also called a *figure*.