

Arcs, Pies, and Area Fills

We have already seen how to draw a polygon using a *GraphicsPath* object, and how to use a *GraphicsPath* to create a region that corresponds to an ellipse, in earlier examples.

There is also a method *DrawPolygon(Pen pen, Point[] points)* that might sometimes be useful. It connects the points in the given array, joining the lines nicely and connecting the last point to the first one automatically. No doubt it works by creating a *GraphicsPath* from the array of points and then calling *CloseFigure()*, and then *DrawPath*.

Question: Why didn't we use *DrawPolygon* in the example program that draws a polygon and hit-tests whether the mouse is clicked inside or outside the polygon?

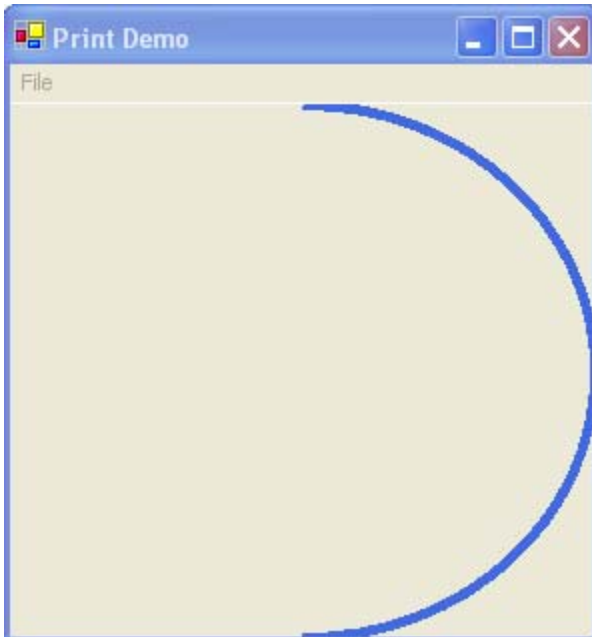
Answer: We could have, but we needed the *GraphicsPath* anyway in order to construct a region to use for hit-testing; so we might as well just call *DrawPath*.

We have also already used *DrawEllipse*. But what if you need to draw just *part* of a circle, or part of an ellipse? The function *DrawArc* can do that:

DrawArc(Pen p, Rectangle r, float startAngle, float sweepAngle)

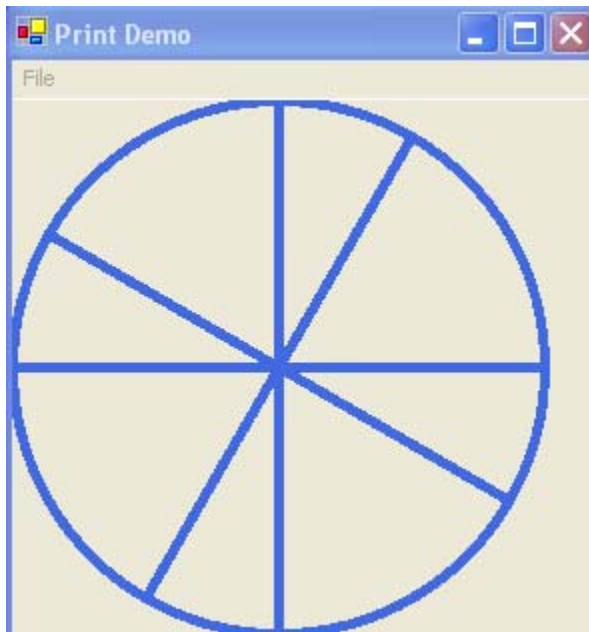
The *Rectangle* argument can also be replaced by four numbers. The two angle arguments are measured increasing in the clockwise direction, which is backwards from the usual convention in mathematics. Zero *startAngle* means on the positive x-axis relative to the center of the *Rectangle* argument, which is the ellipse's center too. The second angle argument measures the length of the arc to be drawn (in degrees), not the ending point. Here's an example:

```
private void PrintOrDraw(Graphics g)
{ Pen p = new Pen(Color.RoyalBlue,5);
  g.DrawArc(p,ClientRectangle,-90,180);
}
```



Called in a similar way are the *DrawPie* methods. These draw not only the arc, but two lines connecting the endpoints of the arc to the center of the ellipse in question. Here's an example:

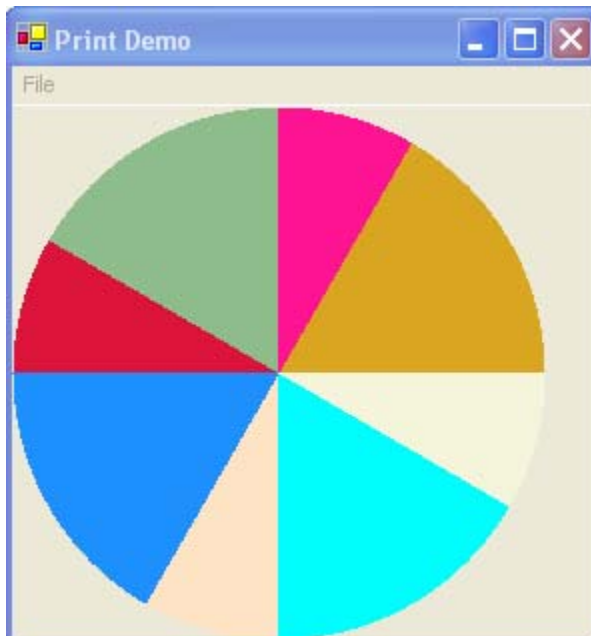
```
private void PrintOrDraw(Graphics g)
{ int[] angles = {30,60,30,60,30,60,30,60};
  int size = Math.Min(ClientRectangle.Width,
ClientRectangle.Height);
  Rectangle r = new Rectangle(0,0,size,size);
  Pen p = new Pen(Color.RoyalBlue,5);
  int theta = 0;
  for(int i=0;i<angles.Length;i++)
  { g.DrawPie(p,r,theta,angles[i]);
    theta += angles[i];
  }
}
```



You can see that the pen is drawing an arc 5 pixels thick centered at the specified coordinates, so where the specified coordinates take their max and min values, some pixels outside the “bounding rectangle” are colored, or (in this case) are *not* colored, because they would lie outside the window.

Just there are both *DrawEllipse* and *FillEllipse*, there is also *FillPie*:

```
private void PrintOrDraw(Graphics g)
{
    int[] angles = {30,60,30,60,30,60,30,60};
    Brush[] brushes = {Brushes.Beige, Brushes.Aqua,
Brushes.Bisque, Brushes.DodgerBlue,
        Brushes.Crimson,
Brushes.DarkSeaGreen,Brushes.DeepPink, Brushes.Goldenrod};
    int size = Math.Min(ClientRectangle.Width,
ClientRectangle.Height);
    Rectangle r = new Rectangle(0,0,size,size);
    Pen p = new Pen(Color.RoyalBlue,5);
    int theta = 0;
    for(int i=0;i<angles.Length;i++)
        { g.FillPie(brushes[i],r,theta,angles[i]);
          theta += angles[i];
        }
}
```



Our next example shows how to add hit-testing to this program, so that when you click in one of the pie-shaped regions, that region turns yellow, and when you click outside the circle, all the regions are shown as in their original colors again. First, we change the local variables *angles* and *r* to be member variables *m_angles* and *m_theRect*, so they can be accessible in the *MouseDown* handler. Then we write a *MouseDown* handler that goes through the pieces of pie one at a time, constructs a *GraphicsPath*, using the *AddPie* method, and then a region from the *GraphicsPath*, and then uses the region's *IsVisible* method to do the hit-testing.

```
private void Form1_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{ GraphicsPath p;
  Region r;
  int i;
  int theta = 0;
  int oldHit = m_theHit; // which region is already
yellow (selected)
  for(i=0;i<m_angles.Length;i++)
  { p = new GraphicsPath();
    p.AddPie(m_theRect,theta,m_angles[i]);
```

```

        r = new Region(p);
        if(p.IsVisible(e.X,e.Y))
            { m_theHit = i;
              if(m_theHit != oldHit)
                  { Invalidate(); // don't invalidate if the
selection isn't changing
                    return;
                  }
            }
        theta += m_angles[i];
    }
    m_theHit = -1;
    if(m_theHit != oldHit)
        Invalidate();
}

```

Garbage collection takes care of deleting all these *GraphicPath* and *Region* objects that we create in this code.

Finally we will discuss the *FillMode*. You will have observed from Visual Studio's prompts that *FillPath* has a version with an extra argument for a *FillMode*. There are just two possible things you can put for this argument:

FillMode.Alternate (the default; you get this if you don't use this argument)

FillMode.Winding.

Even though this does not come up very often, you should understand what the *FillMode* does. Remember that a *GraphicsPath* describes a geometric figure drawn by a sequence of connected line segments or arcs (or Bezier curves, but we haven't discussed that). If you call *FillPath*, the graphics path is automatically closed, with the last point connected to the first. If these segments or arcs never cross each other, then the entire figure has a well-defined "inside" and "outside", and *FillPath* has a clear task: color the inside. But if they do intersect each other, as in a five-pointed star drawn with five intersecting lines, it is not so obvious what points are *inside* and what points are *outside*. If the

fill mode is *FillMode.Alternate*, then point p is *inside* the path if a line from p to infinity intersects an odd number of boundaries. It can be proved that if we count “intersections” correctly, the number of intersections does not depend on which line we use from p to infinity. According to this definition, the interior pentagon of a five-pointed star is *outside* the star.

If you call *FillPath* or *FillPolygon* and the result surprises you by leaving some parts unfilled, you may want to try *FillMode.Winding*. This will fill all the areas that would be filled under *FillMode.Alternate*, but it may fill more areas as well. Here is how *inside* is defined under this fill mode: When we connect a point p to infinity, we consider the arcs and line segments of the path to have a direction (the direction they are traced out when drawing the path). We count the intersections with boundary line segments and arcs more carefully, counting each one with a plus or minus sign according as it crosses the line to infinity from right to left, or left to right. If the sum of the intersections counted this way is zero then the point is *outside*, and is not filled. For example, the center pentagon of a five-pointed star will be filled.