CHAPTER 12

Shapes, Brushes, and Transforms

In many user interface technologies, there's a clear distinction between ordinary controls and custom drawing. Often, the drawing features are used only in specialized applications—for example, games, data visualization, physics simulations, and so on.

WPF has a dramatically different philosophy. It handles prebuilt controls and custom-drawn graphics in the same way. Not only will you use WPF's drawing support to create graphically rich visuals for your user interface, but you'll also use it to get the most out of other features like animation (Chapter 15) and control templates (Chapter 17). In fact, WPF's drawing support is equally important whether you're creating a dazzling new game or just adding polish to an ordinary business application.

In this chapter, you'll explore WPF's 2-D drawing features, starting with the basic elements for shape drawing. Next, you'll consider how to paint their borders and interiors with brushes. Then you'll learn how to rotate, skew, and otherwise manipulate shapes and elements using transforms. Finally, you'll see how to make shapes and other elements partially transparent.

What's New The basic 2-D drawing features in WPF haven't changed in version 4. The only addition is a new BitmapCacheBrush (covered in this chapter) that allows you to convert complex graphical content to a bitmap that's cached on your video card. This technique improves performance in certain specialized scenarios.

Understanding Shapes

The simplest way to draw 2-D graphical content in a WPF user interface is to use *shapes*—dedicated classes that represent simple lines, ellipses, rectangles, and polygons. Technically, shapes are known as drawing *primitives*. You can combine these basic ingredients to create more complex graphics.

The most important detail about shapes in WPF is that they all derive from FrameworkElement. As a result, shapes *are* elements. This has a number of important consequences:

- **Shapes draw themselves.** You don't need to manage the invalidation and painting process. For example, you don't need to manually repaint a shape when content moves, the window is resized, or the shape's properties change.
- **Shapes are organized in the same way as other elements.** In other words, you can place a shape in any of the layout containers you learned about in Chapter 3. (Although the Canvas is obviously the most useful container because it allows you to place shapes at specific coordinates, which is important when you're building a complex drawing out of multiple pieces.)
- **Shapes support the same events as other elements.** That means you don't need to go to any extra work to deal with focus, key presses, mouse movements, and mouse clicks. You can use the same set of events you would use with any element, and you have the same support for tooltips, context menus, and drag-and-drop operations.

This model is dramatically different than those in earlier user interface technologies, such as Windows Forms. Those frameworks do most of their work using a traditional windowing model (through User32), which would be incredibly inefficient if applied to pieces of graphical content, such as individual lines and squares. Additionally, the window model requires that each element "own" a small section of screen real estate, which makes it difficult to add transparency and use anti-aliasing around the edges of a nonrectangular shape.

Because of these limitations, older frameworks use the lower-level GDI/GDI+ model for custom drawing. This requires more work and provides far fewer high-level features.

■ **Tip** As you'll see in Chapter 14, it's still possible to program at a lower level in WPF using the *visual layer*. This lightweight model improves performance if you need to create huge numbers of elements (say, thousands of shapes), and you don't need all the features of the UIElement and FrameworkElement classes (such as data binding and event handling). However, visual layer programming still works at a higher level than GDI/GDI+. Most important, WPF still manages the redrawing processing automatically. You simply supply the content.

The Shape Classes

Every shape derives from the abstract System.Windows.Shapes.Shape class. Figure 12-1 shows the inheritance hierarchy for shapes.

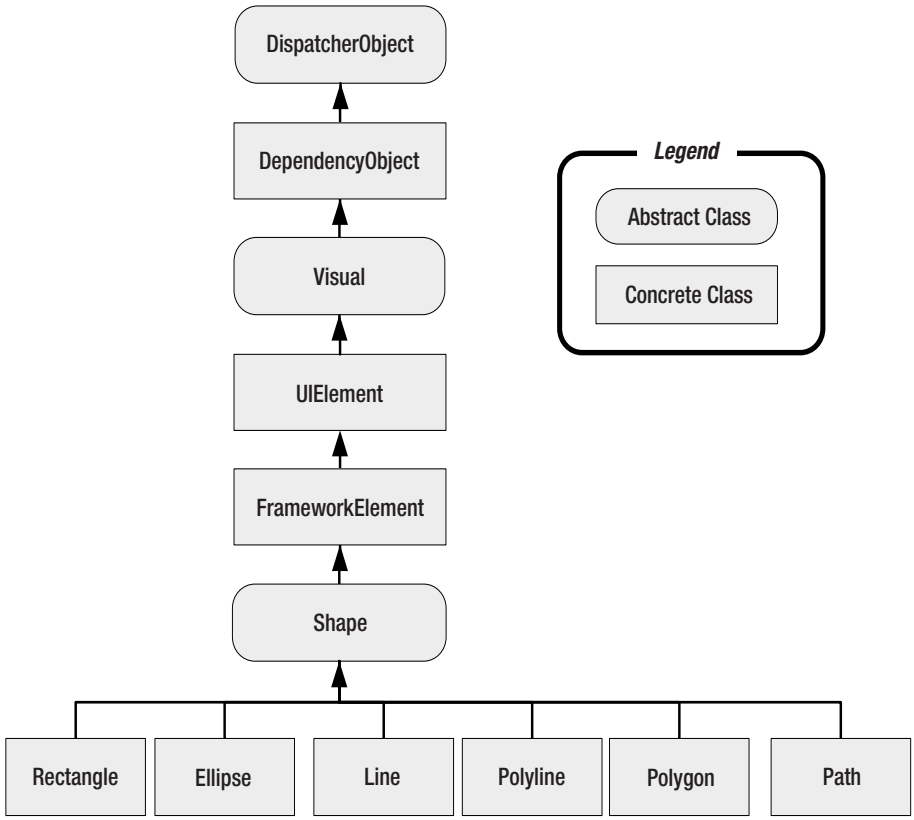

Figure 12-1. *The WPF shape classes*

As you can see, a relatively small set of classes derive from the Shape class. Line, Ellipse, and Rectangle are straightforward. Polyline is a connected series of straight lines. Polygon is a closed shape made up of a connected series of straight lines. Finally, the Path class is an all-in-one superpower that can combine basic shapes in a single element.

Although the Shape class can't do anything on its own, it defines a small set of important properties, which are listed in Table 12-1.

Table 12-1. *Shape Properties*

Name	Description
Fill	Sets the brush object that paints the surface of the shape (everything inside its borders).
Stroke	Sets the brush object that paints the edge of the shape (its border).

Name	Description
StrokeThickness	Sets the thickness of the border, in device-independent units. When drawing a line, WPF splits the width on each side. So a line that's 10 units wide gets 5 units of space on each side of where a single-unit line would be drawn. If you give a line an odd-number thickness, the line will have a fractional width on each side. For example, an 11-unit line has 5.5 units of space on each side. This pretty much guarantees that the line won't line up evenly with the display pixels of your monitor, even if it's running at 96-dpi resolution, so you'll end up with a slightly fuzzy anti-aliased edge. You can use the SnapsToDevicePixels property to clean this up if it bothers you (as described in the section "Pixel Snapping" later in this chapter).
StrokeStartLineCap and StrokeEndLineCap	Determine the contour of the edge of the beginning and end of the line. These properties have an effect only for the Line, the Polyline, and (sometimes) the Path shapes. All other shapes are closed, and so have no starting and ending point.
StrokeDashArray, StrokeDashOffset, and StrokeDashCap	Allow you to create a dashed border around a shape. You can control the size and frequency of the dashes and how the edge where each dash line begins and ends is contoured.
StrokeLineJoin and StrokeMiterLimit	Determine the contour of the corners of a shape. Technically, these properties affect the *vertices* where different lines meet, such as the corners of a Rectangle. These properties have no effect for shapes without corners, such as Line and Ellipse.
Stretch	Determines how a shape fills its available space. You can use this property to create a shape that expands to fit its container. You can also force a shape to expand in one direction using a Stretch value for the HorizontalAlignment or VerticalAlignment properties (which are inherited from the FrameworkElement class).
DefiningGeometry	Provides a Geometry object for the shape. A Geometry object describes the coordinates and size of a shape without including the UIElement plumbing, such as the support for keyboard and mouse events. You'll use geometries in Chapter 13.
GeometryTransform	Allows you to apply a Transform object that changes the coordinate system that's used to draw a shape. This allows you to skew, rotate, or displace a shape. Transforms are particularly useful when animating graphics. You'll learn about transforms later in this chapter.
RenderedGeometry	Provides a Geometry object that describes the final, rendered shape. Geometries are described in Chapter 13.

In the following sections, you'll consider the Rectangle, Ellipse, Line, and Polyline. Along the way, you'll learn the following fundamentals:

- How to size shapes and organize them in a layout container
- How to control which regions of a complex shape are filled in
- How to use dashed lines and different line ends (or "caps")
- How to neatly align shape edges along pixel boundaries

You'll take a look at the more sophisticated Path class in Chapter 13.

Rectangle and Ellipse

The Rectangle and Ellipse are the two simplest shapes. To create either one, set the familiar Height and Width properties (inherited from FrameworkElement) to define the size of your shape, and then set the Fill or Stroke property (or both) to make the shape visible. You're also free to use properties such as MinHeight, MinWidth, HorizontalAlignment, VerticalAlignment, and Margin.

■ **Note** If you fail to set the Stroke or Fill property, your shape won't appear at all.

Here's a simple example that stacks an ellipse on a rectangle (see Figure 12-2) using a StackPanel:

```
<StackPanel>
  <Ellipse Fill="Yellow" Stroke="Blue"
   Height="50" Width="100" Margin="5" HorizontalAlignment="Left"></Ellipse>
  <Rectangle Fill="Yellow" Stroke="Blue"
   Height="50" Width="100" Margin="5" HorizontalAlignment="Left"></Rectangle>
</StackPanel>
```

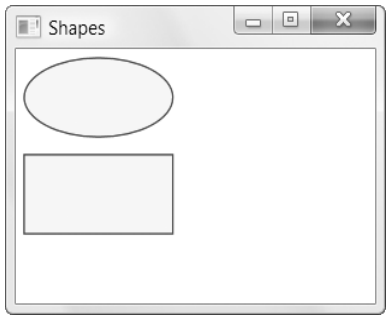

Figure 12-2. *Two simple shapes*

The Ellipse class doesn't add any properties. The Rectangle class adds just two: RadiusX and RadiusY. When set to nonzero values, these properties allow you to create nicely rounded corners.

You can think of RadiusX and RadiusY as describing an ellipse that's used just to fill in the corners of the rectangle. For example, if you set both properties to 10, WPF draws your corners using the edge of a circle that's 10 units wide. As you make your radius larger, more of your rectangle will be rounded off. If you increase RadiusY more than RadiusX, your corners will round off more gradually along the left and right sides and more sharply along the top and bottom edge. If you increase the RadiusX property to match your rectangle's width, and increase RadiusY to match its height, you'll end up converting your rectangle into an ordinary ellipse.

Figure 12-3 shows a few rectangles with rounded corners.

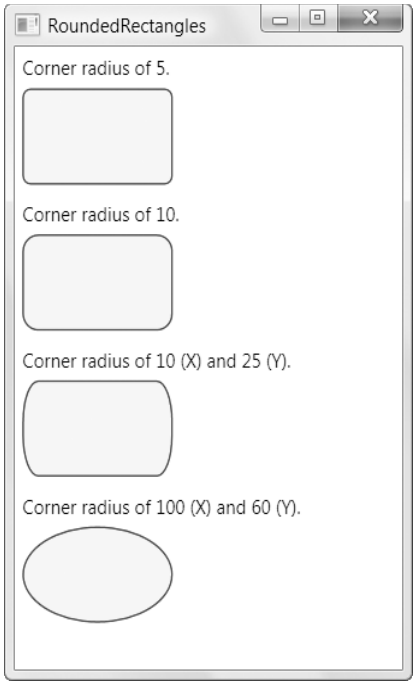

Figure 12-3. *Rounded corners*

Sizing and Placing Shapes

As you already know, hard-coded sizes are usually not the ideal approach to creating user interfaces. They limit your ability to handle dynamic content, and they make it more difficult to localize your application into other languages.

When drawing shapes, these concerns don't always apply. Often, you'll need tighter control over shape placement. However, there are many cases where you can make your design a little more flexible. Both the Ellipse and the Rectangle have the ability to size themselves to fill the available space.

If you don't supply the Height and Width properties, the shape is sized based on its container. In the previous example, removing the Height and Width values (and leaving out the MinHeight and MinWidth

values) will cause the shapes to shrink to a vanishingly small size, because the StackPanel is sized to fit its content. However, if you force the StackPanel to take the full width of the window (by setting its HorizontalAlignment property to Stretch), then also set the HorizontalAlignment property of the ellipse to Stretch and remove the ellipse's Width property, the ellipse will take the full width of the window.

A better example can be made with the Grid container. If you use the proportional row-sizing behavior (which is the default), you can create an ellipse that fills a window with this stripped-down markup:

```
<Grid>
  <Ellipse Fill="Yellow" Stroke="Blue"></Ellipse>
</Grid>
```

Here, the Grid fills the entire window. The Grid contains a single proportionately sized row, which fills the entire Grid. Finally, the ellipse fills the entire row.

This sizing behavior depends on the value of the Stretch property (which is defined in the Shape class). By default, it's set to Fill, which stretches a shape to fill its container if an explicit size isn't indicated. Table 12-2 lists all your possibilities.

Table 12-2. *Values for the Stretch Enumeration*

Name	Description
Fill	Your shape is stretched in width and height to fit its container exactly. (If you set an explicit height and width, this setting has no effect.)
None	The shape is not stretched. Unless you set a nonzero width and height (using the Height and Width or MinHeight and MinWidth properties), your shape won't appear.
Uniform	The width and height are sized up proportionately until the shape reaches the edge of the container. If you use this with an ellipse, you'll end up with the biggest circle that fits in the window. If you use it with a rectangle, you'll get the biggest possible square. (If you set an explicit height and width, your shape is sized within those bounds. For example, if you set a Width of 10 and a Height of 100 for a rectangle, you'll only get a 10 × 10 square.)
UniformToFill	The width and height are sized proportionately until the shape fills all the available height and width. For example, if you place a rectangle with this size setting into a window that's 100 × 200 units, you'll get a 200 × 200 rectangle, and part of it will be clipped off. (If you set an explicit height and width, your shape is sized within those bounds. For example, if you set a Width of 10 and a Height of 100 for a rectangle, you'll get a 100 × 100 rectangle that's clipped to fit an invisible 10 × 100 box.)

Figure 12-4 shows the difference between Fill, Uniform, and UniformToFill.

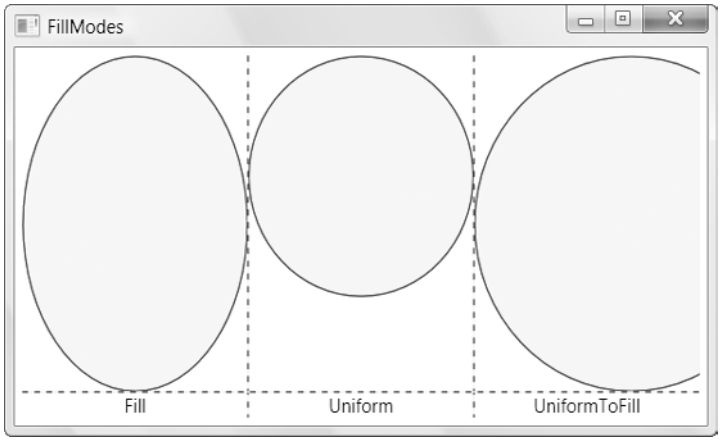

Figure 12-4. *Filling three cells in a Grid*

Usually, a Stretch value of Fill is the same as setting both HorizontalAlignment and VerticalAlignment to Stretch. The difference occurs if you choose to set a fixed Width or Height on your shape. In this case, the HorizontalAlignment and VerticalAlignment values are simply ignored. However, the Stretch setting still has an effect—it determines how your shape content is sized within the bounds you've given it.

■ **Tip** In most cases, you'll size a shape explicitly or allow it to stretch to fit. You won't combine both approaches.

So far, you've seen how to size a Rectangle and an Ellipse, but what about placing them exactly where you want them? WPF shapes use the same layout system as any other element. However, some layout containers aren't as appropriate. For example, the StackPanel, DockPanel, and WrapPanel often aren't what you want because they're designed to separate elements. The Grid is a bit more flexible because it allows you to place as many elements as you want in the same cell (although it doesn't let you position squares and ellipses in different parts of that cell). The ideal container is the Canvas, which forces you to specify the coordinates of each shape using the attached Left, Top, Right, or Bottom properties. This gives you complete control over how shapes overlap:

```
<Canvas>
  <Ellipse Fill="Yellow" Stroke="Blue" Canvas.Left="100" Canvas.Top="50"
    Width="100" Height="50"></Ellipse>
    <Rectangle Fill="Yellow" Stroke="Blue" Canvas.Left="30" Canvas.Top="40"
      Width="100" Height="50"></Rectangle>
</Canvas>
```

With a Canvas, the order of your tags is important. In the previous example, the rectangle is superimposed on the ellipse because the ellipse appears first in the list, and so is drawn first (see Figure 12-5).

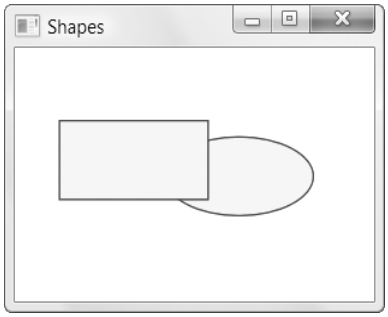

Figure 12-5. *Overlapping shapes in a Canvas*

Remember that a Canvas doesn't need to occupy an entire window. For example, there's no reason that you can't create a Grid that uses a Canvas in one of its cells. This gives you the perfect way to lock down fixed bits of drawing logic in a dynamic, free-flowing user interface.

Scaling Shapes with a Viewbox

The only limitation to using the Canvas is that your graphics won't be able to resize themselves to fit larger or smaller windows. This makes perfect sense for buttons (which don't change size in these situations), but not necessarily for other types of graphical content. For example, you might create a complex graphic that you want to be resizable so it can take advantage of the available space.

In situations like these, WPF has an easy solution. If you want to combine the precise control of the Canvas with easy resizability, you can use the Viewbox element.

The Viewbox is a simple class that derives from Decorator (much like the Border class you first encountered in Chapter 3). It accepts a single child, which it stretches or shrinks to fit the available space. Of course, that single child can be a layout container, which can hold a number of shapes (or other elements) that will be resized in sync. However, it's more common to use the Viewbox for vector graphics than for ordinary controls.

Although you could place a single shape in a Viewbox, that doesn't provide any real advantage. Instead, the Viewbox shines when you need to wrap a group of shapes that make up a drawing. Typically, you'll place a Canvas inside a Viewbox, and place your shapes inside the Canvas.

The following example puts a Canvas-containing Viewbox in the second row of a Grid. The Viewbox takes the full height and width of the row. The row takes whatever space is left over after the first autosized row is rendered. Here's the markup:

```
<Grid Margin="5">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="*"></RowDefinition>
  </Grid.RowDefinitions>

  <TextBlock>The first row of a Grid.</TextBlock>

  <Viewbox Grid.Row="1" HorizontalAlignment="Left" >
    <Canvas Width="200" Height="150">
      <Ellipse Fill="Yellow" Stroke="Blue" Canvas.Left="10"  Canvas.Top="50"
```

```
        Width="100" Height="50" HorizontalAlignment="Left"></Ellipse>
      <Rectangle Fill="Yellow" Stroke="Blue" Canvas.Left="30"  Canvas.Top="40"
        Width="100" Height="50" HorizontalAlignment="Left"></Rectangle>
    </Canvas>
  </Viewbox>
</Grid>
```

Figure 12-6 shows how the Viewbox adjusts itself as the window is resized. The first row is unchanged. However, the second row expands to fill the extra space. As you can see, the shape in the Viewbox changes proportionately as the window grows.

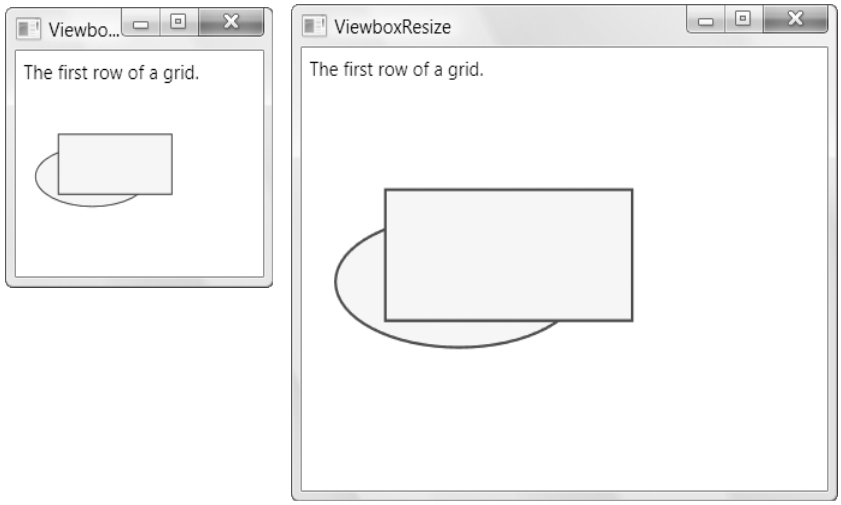

Figure 12-6. *Resizing with a Viewbox*

■ **Note** The scaling that the Viewbox does is similar to the scaling you see in WPF if you increase the system DPI setting. It changes every onscreen element proportionately, including images, text, lines, and shapes. For example, if you place an ordinary button in a Viewbox, the sizing will affect its overall size, the text inside, and the thickness of the border around it. If you place a shape element inside, the Viewbox resizes its inside area and its border proportionately, so the larger your shape grows, the thicker its border will be.

By default, the Viewbox performs proportional scaling that preserves the aspect ratio of its contents. In the current example, that means that even if the shape of the containing row changes (growing wider or taller), the shapes inside won't be distorted. Instead, the Viewbox uses the largest scaling factor that fits inside the available space. However, you can change this behavior using the Viewbox.Stretch property. By default, it's set to Uniform, but you can use any of the values from Table 12-2. Change it to Fill, and the content inside the Viewbox is stretched in both directions to fit the available space exactly, even if it mangles your original drawing. You can also get slightly more control by using the

StretchDirection property. By default, this property takes the value Both, but you can use UpOnly to create content that can grow but won't shrink beyond its original size, and use DownOnly to create content that can shrink but not grow.

In order for the Viewbox to perform its scaling magic, it needs to be able to determine two pieces of information: the ordinary size that your content would have (if it weren't in a Viewbox) and the new size that you want it to have.

The second detail—the new size—is simple enough. The Viewbox gives the inner content all the space that's available, based on its Stretch property. That means the bigger the Viewbox, the bigger your content.

The first detail—the ordinary, non-Viewbox size, is implicit in the way you define the nested content. In the previous example, the Canvas is given an explicit size of 200 by 150 units. Thus, the Viewbox scales the image from that starting point. For example, the ellipse is initially 100 units wide, which means it takes up half the allotted Canvas drawing space. As the Canvas grows larger, the Viewbox respects these proportions, and the ellipse continues to take half the available space.

However, consider what happens if you remove the Width and Height properties from the Canvas. Now the Canvas is given a size of 0 by 0 units, so the Viewbox cannot resize it, and your nested content won't appear. (This is different from the behavior you get if you have the Canvas on its own. That's because even though the Canvas is still given a size of 0 by 0, your shapes are allowed to draw outside the Canvas area as long as the Canvas.ClipToBounds property hasn't been set to true. The Viewbox isn't as tolerant of this error.)

Now consider what happens if you wrap the Canvas inside a proportionately sized Grid cell and you don't specify the size of the Canvas. If you aren't using the Viewbox, this approach works perfectly well—the Canvas is stretched to fill the cell, and the content inside is visible. But if you place all this content in a Viewbox, this strategy fails. The Viewbox can't determine the initial size, so it can't resize the Grid appropriately.

You can get around this problem by placing certain shapes (such as the Rectangle and Ellipse) directly in an autosized container (such as the Grid). The Viewbox can then evaluate the minimum size the Grid needs to fit its content and scale it up to fit what's available. However, the easiest way to get the size you really want in a Viewbox is to wrap your content in an element that has a fixed size, whether it's a Canvas, a button, or something else. This fixed size then becomes the initial size that the Viewbox uses for its calculations. Hard-coding a size in this way won't limit the flexibility of your layout, because the Viewbox is sized proportionately based on the available space and its layout container.

Line

The Line shape represents a straight line that connects one point to another. The starting and ending points are set by four properties: X1 and Y1 (for the first point) and X2 and Y2 (for the second point). For example, here's a line that stretches from (0, 0) to (10, 100):

```
<Line Stroke="Blue" X1="0" Y1="0" X2="10" Y2="100"></Line>
```

The Fill property has no effect for a line. You must set the Stroke property.

The coordinates you use in a line are relative to the top-left corner where the line is placed. For example, if you place the previous line in a StackPanel, the coordinate (0, 0) points to wherever that item in the StackPanel is placed. It might be the top-left corner of the window, but it probably isn't. If the StackPanel uses a nonzero Margin, or if the line is preceded by other elements, the line will begin at a point (0, 0) some distance down from the top of the window.

However, it's perfectly reasonable to use negative coordinates for a line. In fact, you can use coordinates that take your line out of its allocated space and draw overtop of any other part of the window. This isn't possible with the Rectangle and Ellipse shapes you've seen so far. However, there's

also a drawback to this model, which is that lines can't use the flow content model. That means there's no point setting properties such as Margin, HorizontalAlignment, and VerticalAlignment on a line, because they won't have any effect. The same limitation applies to the Polyline and Polygon shapes.

■ **Note** You can use the Height, Width, and Stretch properties with a line, although it's not terribly common. The basic technique is to use the Height and Width to determine the space that's allocated to the line, and then use the Stretch property to resize the line to fill this area.

If you place a Line in a Canvas, the attached position properties (such as Top and Left) still apply. They determine the starting position of the line. In other words, the two line coordinates are offset by that amount. Consider this line:

```
<Line Stroke="Blue" X1="0" Y1="0" X2="10" Y2="100"
 Canvas.Left="5" Canvas.Top="100"></Line>
```

It stretches from (0, 0) to (10, 100), using a coordinate system that treats the point (5, 100) on the Canvas as (0, 0). That makes it equivalent to this line, which doesn't use the Top and Left properties:

```
<Line Stroke="Blue" X1="5" Y1="100" X2="15" Y2="200"></Line>
```

It's up to you whether you use the position properties when you place a Line on a Canvas. Often, you can simplify your line drawing by picking a good starting point. You also make it easier to move parts of your drawing. For example, if you draw several lines and other shapes at a specific position in a Canvas, it's a good idea to draw them relative to a nearby point (by using the same Top and Left coordinates). That way, you can shift that entire part of your drawing to a new position as needed.

■ **Note** There's no way to create a curved line with Line or Polyline shapes. Instead, you need the more advanced Path class described in Chapter 13.

Polyline

The Polyline class allows you to draw a sequence of connected straight lines. You simply supply a list of X and Y coordinates using the Points property. Technically, the Points property requires a PointCollection object, but you fill this collection in XAML using a lean string-based syntax. You simply need to supply a list of points and add a space or a comma between each coordinate.

A Polyline can have as few as two points. For example, here's a Polyline that duplicates the first line you saw in this section, which stretches from (5, 100) to (15, 200):

```
<Polyline Stroke="Blue" Points="5 100 15 200"></Polyline>
```

For better readability, use commas in between each X and Y coordinate:

```
<Polyline Stroke="Blue" Points="5,100 15,200"></Polyline>
```

And here's a more complex Polyline that begins at (10, 150). The points move steadily to the right, oscillating between higher Y values such as (50, 160) and lower ones such as (70, 130):

```
<Canvas>
  <Polyline Stroke="Blue" StrokeThickness="5" Points="10,150 30,140 50,160 70,130
90,170 110,120 130,180 150,110 170,190 190,100 210,240" >
  </Polyline>
</Canvas>
```

Figure 12-7 shows the final line.

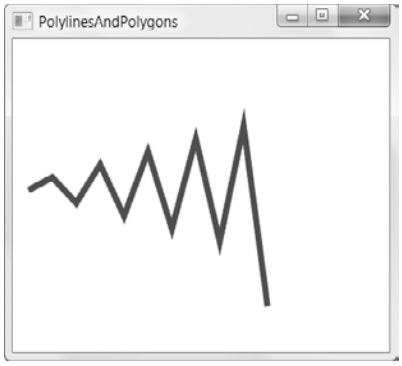

Figure 12-7. *A line with several segments*

At this point, it might occur to you that it would be easier to fill the Points collection programmatically, using some sort of loop that automatically increments X and Y values accordingly. This is true if you need to create highly dynamic graphics—for example, a chart that varies its appearance based on a set of data you extract from a database. But if you simply want to build a fixed piece of graphical content, you won't want to worry about the specific coordinates of your shapes at all. Instead, you (or a designer) will use another tool, such as Expression Design, to draw the appropriate graphics, and then export to XAML.

Polygon

The Polygon is virtually the same as the Polyline. Like the Polyline class, the Polygon class has a Points collection that takes a list of coordinates. The only difference is that the Polygon adds a final line segment that connects the final point to the starting point. (If your final point is already the same as the first point, the Polygon class has no difference from the Polyline class.) You can fill the interior of this shape using the Fill brush. Figure 12-8 shows the previous Polyline as a Polygon with a yellow fill.

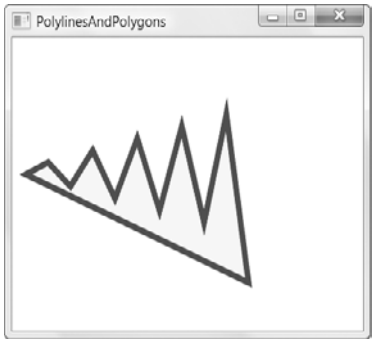

Figure 12-8. *A filled polygon*

■ **Note** Technically, you can set the Fill property of a Polyline as well. In this situation, the Polyline fills itself as though it were a Polygon—in other words, as though it has an invisible line segment connecting the last point to the first point. This effect is of limited use.

In a simple shape where the lines never cross, it's easy to fill the interior. However, sometimes you'll have a more complex Polygon where it's not necessarily obvious which portions are "inside" the shape (and should be filled) and which portions are outside.

For example, consider Figure 12-9, which features a line that crosses more than one other line, leaving an irregular region at the center that you may or may not want to fill. Obviously, you can control exactly what gets filled by breaking this drawing down into smaller shapes, but you may not need to do that.

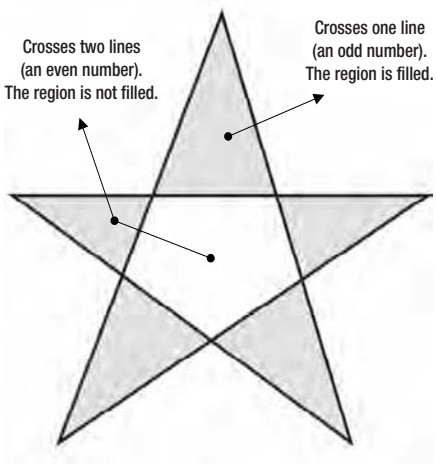

Figure 12-9. *Determining fill areas when FillRule is EvenOdd*

Every Polygon and Polyline includes a FillRule property, which lets you choose between two different approaches for filling in regions. By default, FillRule is set to EvenOdd. In order to decide whether to fill a region, WPF counts the number of lines that must be crossed to reach the outside of the shape. If this number is odd, the region is filled in; if it's even, the region isn't filled. In the center area of Figure 12-9, you must cross two lines to get out of the shape, so it's not filled.

WPF also supports the Nonzero fill rule, which is a little trickier. Essentially, with Nonzero, WPF follows the same line-counting process as EvenOdd, but it takes into account the direction that each line flows. If the number of lines going in one direction (say, left to right) is equal to the number going in the opposite direction (right to left), the region is not filled. If the difference between these two counts is not zero, the region is filled. In the shape from the previous example, the interior region is filled if you set the FillRule property to Nonzero. Figure 12-10 shows why. (In this example, the points are numbered in the order they are drawn, and arrows show the direction in which each line is drawn.)

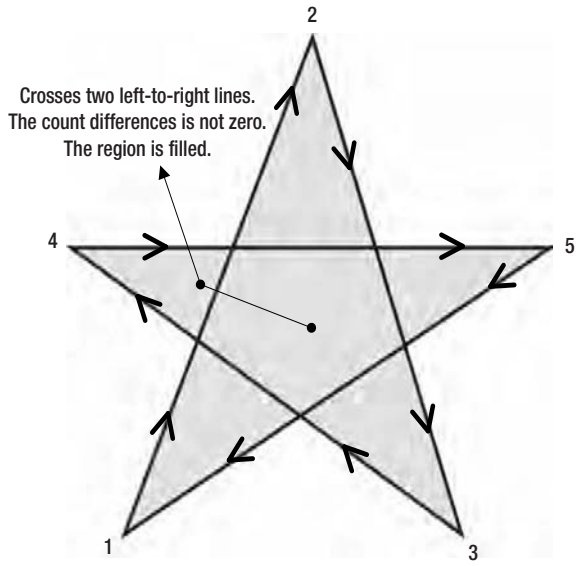

Figure 12-10. *Determining fill areas when FillRule is Nonzero*

■ **Note** If there are an odd number of lines, the difference between the two counts can't be zero. Thus, the Nonzero fill rule always fills at least as much as the EvenOdd rule, plus possibly a bit more.

The tricky part about Nonzero is that its fill settings depend on *how* you draw the shape, not what the shape itself looks like. For example, you could draw the same shape in such a way that the center isn't filled (although it's much more awkward—you would begin by drawing the inner region, and then you would draw the outside spikes in the reverse direction).

Here's the markup that draws the star shown in Figure 12-10:

```
<Polygon Stroke="Blue" StrokeThickness="1" Fill="Yellow"
 Canvas.Left="10" Canvas.Top="175" FillRule="Nonzero"
 Points="15,200 68,70 110,200 0,125 135,125">
</Polygon>
```

Line Caps and Line Joins

When drawing with the Line and Polyline shapes, you can choose how the starting and ending edge of the line is drawn using the StartLineCap and EndLineCap properties. (These properties have no effect on other shapes because they're closed.)

Ordinarily, both StartLineCap and EndLineCap are set to Flat, which means the line ends immediately at its final coordinate. Your other choices are Round (which rounds the corner off gently), Triangle (which draws the two sides of the line together in a point), and Square (which ends the line with a sharp edge). All of these values add length to the line—in other words, they take it beyond the position where it would otherwise end. The extra distance is half the thickness of the line.

■ **Note** The only difference between Flat and Square is the fact that the square-edged line extends this extra distance. In all other respects, the edge looks the same.

Figure 12-11 shows different line caps at the end of a line.

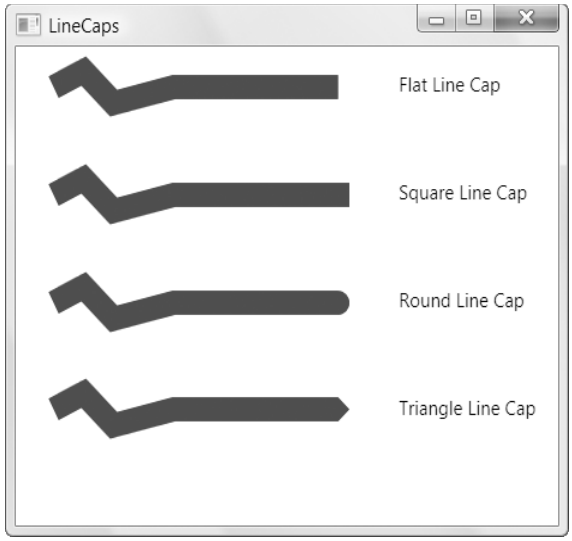

Figure 12-11. *Line caps*

All shapes except Line allow you to tweak how their corners are shaped using the StrokeLineJoin property. You have three choices: Miter (the default), uses sharp edges, Bevel cuts off the point edge, and Round rounds it out gently. Figure 12-12 shows the difference.

When using mitered edges with thick lines and very small angles, the sharp corner can extend an impractically long distance. In this case, you can use Bevel or Round to pare down the corner. Or you could use the StrokeMiterLimit, which automatically bevels the edge when it reaches a certain maximum length. The StrokeMiterLimit is a ratio that compares the length used to miter the corner to half the thickness of the line. If you set this to 1 (which is the default value), you're allowing the corner to extend half the thickness of the line. If you set it to 3, you're allowing the corner to extend to 1.5 times the thickness of the line. The last line in Figure 12-12 uses a higher miter limit with a narrow corner.

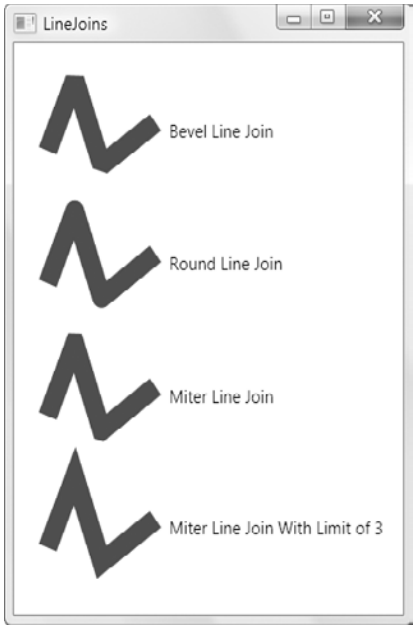

Figure 12-12. *Line joins*

Dashes

Instead of drawing boring solid lines for the borders of your shape, you can draw *dashed lines*—lines that are broken with spaces according to a pattern you specify.

When creating a dashed line in WPF, you aren't limited to specific presets. Instead, you choose the length of the solid segment of the line and the length of the broken (blank) segment by setting the StrokeDashArray property. For example, consider this line:

```
<Polyline Stroke="Blue" StrokeThickness="14" StrokeDashArray="1 2"
  Points="10,30 60,0 90,40 120,10 350,10">
</Polyline>
```

It has a line value of 1 and a gap value of 2. These values are interpreted relative to the thickness of the line. So if the line is 14 units thick (as in this example), the solid portion is 14 units, followed by a blank portion of 28 units. The line repeats this pattern for its entire length.

On the other hand, if you swap these values around like so:

```
StrokeDashArray="2 1"
```

you get a line that has 28-unit solid portions broken by 12-unit spaces. Figure 12-13 shows both lines. As you'll notice, when a very thick line segment falls on a corner, it may be broken unevenly.

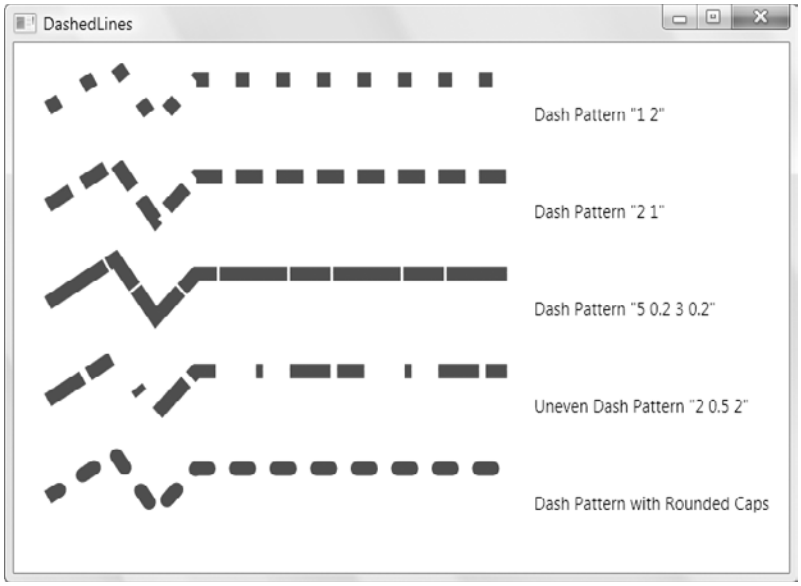

Figure 12-13. *Dashed lines*

There's no reason that you need to stick with whole number values. For example, this StrokeDashArray is perfectly reasonable:

```
StrokeDashArray="5 0.2 3 0.2"
```

It supplies a more complex sequence—a dashed line that's 5 × 14 length, then a 0.2 × 15 break, followed by a 3 × 14 length and another 0.2 × 14 length. At the end of this sequence, the line repeats the pattern from the beginning.

An interesting thing happens if you supply an odd number of values for the StrokeDashArray. Take this one for example:

```
StrokeDashArray="3 0.5 2"
```

When drawing this line, WPF begins with a 3-times-thickness line, followed by a 0.5-times-thickness space, followed by a 2-times-thickness line. But when it repeats the pattern, it starts with a gap, meaning you get a 3-times-thickness *space*, followed by a 0.5-times-thickness line, and so on. Essentially, the dashed line alternates its pattern between line segments and spaces.

If you want to start midway into your pattern, you can use the StrokeDashOffset property, which is a 0-based index number that points to one of the values in your StrokeDashArray. For example, if you set StrokeDashOffset to 1 in the previous example, the line will begin with the 0.5-thickness space. Set it to 2, and the line begins with the 2-thickness segment.

Finally, you can control how the broken edges of your line are capped. Ordinarily, it's a straight edge, but you can set the StrokeDashCap to the Bevel, Square, and Triangle values you considered in the previous section. Remember that all of these settings add one half the line thickness to the end of your dash. If you don't take this into account, you might end up with dashes that overlap one another. The solution is to add extra space to compensate.

■ **Tip** When using the StrokeDashCap property with a line (not a shape), it's often a good idea to set the StartLineCap and EndLineCap to the same values. This makes the line look consistent.

Pixel Snapping

As you know, WPF uses a device-independent drawing system. You specify sizes for things like fonts and shapes using "virtual" pixels, which are the same size as normal pixels on ordinary 96-dpi displays but are scaled up on higher DPI displays. In other words, a rectangle you draw that's 50 pixels wide might actually be rendered using more or fewer pixels, depending on the device. This conversion between device-independent units and physical pixels happens automatically, and you usually don't need to think about it.

The ratio of pixels between different DPI settings is rarely a whole number. For example, 50 pixels at 96 dpi become 62.4996 pixels on a 120-dpi monitor. (This isn't an error condition; in fact, WPF always allows you to use fractional double values when supplying a value in device-independent units.) Obviously, there's no way to place an edge on a point that's between pixels. WPF compensates by using anti-aliasing. For example, when drawing a red line that's 62.4992 pixels long, WPF might fill the first 62 pixels normally and then shade the sixty-third pixel with a value that's in between the line color (red) and the background. However, there's a catch. If you're drawing straight lines, rectangles, or polygons with square corners, this automatic anti-aliasing can introduce a tinge of blurriness at the edges of your shape.

You might assume that this problem appears only when you're running an application on a display that has display resolution that's *not* 96 dpi. However, that's not necessarily the case, because all shapes can be sized using fractional lengths and coordinates, which causes the same issue. And although you probably won't use fractional values in your shape drawing, *resizable* shapes—shapes that are stretched because they size along with their container or they're placed in a Viewbox—will almost always end up with fractional sizes. Similarly, odd-numbered line thicknesses create a line that has a fractional number of pixels on either side.

The fuzzy edge issue isn't necessarily a problem. In fact, depending on the type of graphic you're drawing, it might look quite normal. However, if you don't want this behavior, you can tell WPF not to use anti-aliasing for a specific shape. Instead, WPF will round the measurement to the nearest device pixel. You turn on this feature, which is called *pixel snapping*, by setting the SnapsToDevicePixels property of a UIElement to true.

To see the difference, look at the magnified window in Figure 12-14, which compares two rectangles. The bottom one uses pixel snapping, and the top one doesn't. If you look carefully, you'll see a thin edge of lighter color along the top and left edges of the unsnapped rectangle.

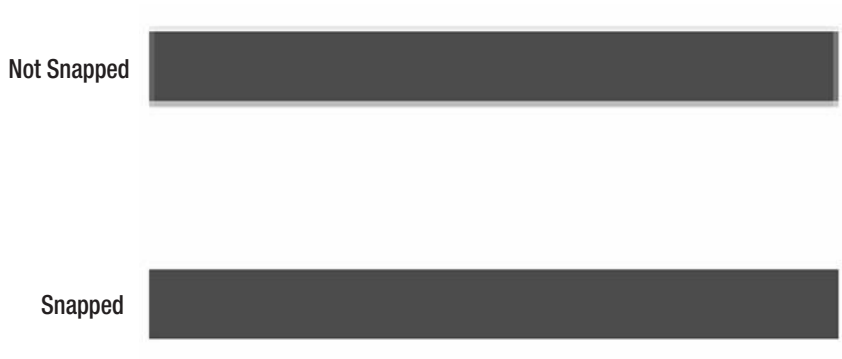

Figure 12-14. *The effect of pixel snapping*

Brushes

Brushes fill an area—whether it's the background, foreground, or border of an element, or the fill or stroke of a shape. The simplest type of brush is SolidColorBrush, which paints a solid, continuous color. When you set the Stroke or Fill property of a shape in XAML, there's a SolidColorBrush at work, doing the painting behind the scenes.

Here are a few more fundamental facts about brushes:

- Brushes support change notification because they derive from Freezable. As a result, if you change a brush, any elements that use that brush repaint themselves automatically.
- Brushes support partial transparency. All you need to do is modify the Opacity property to let the background show through. You'll try out this approach at the end of this chapter.
- The SystemBrushes class provides access to brushes that use the colors defined in the Windows system preferences for the current computer.

Although SolidColorBrush is indisputably useful, there are several other classes that inherit from System.Windows.Media.Brush and give you more exotic effects. Table 12-3 lists them all.

Table 12-3. *Brush Classes*

Name	Description
SolidColorBrush	Paints an area using a single continuous color.
LinearGradientBrush	Paints an area using a gradient fill, a gradually shaded fill that changes from one color to another (and, optionally, to another and then another, and so on).