PART III

Drawing and Animation

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 such as 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 by using transforms. Finally, you'll see how to make shapes and other elements partially transparent.

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 the following 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.

■ **Tip** As you'll see in Chapter 14, it's possible to program at a lower level in WPF by 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).

The Shape Classes

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

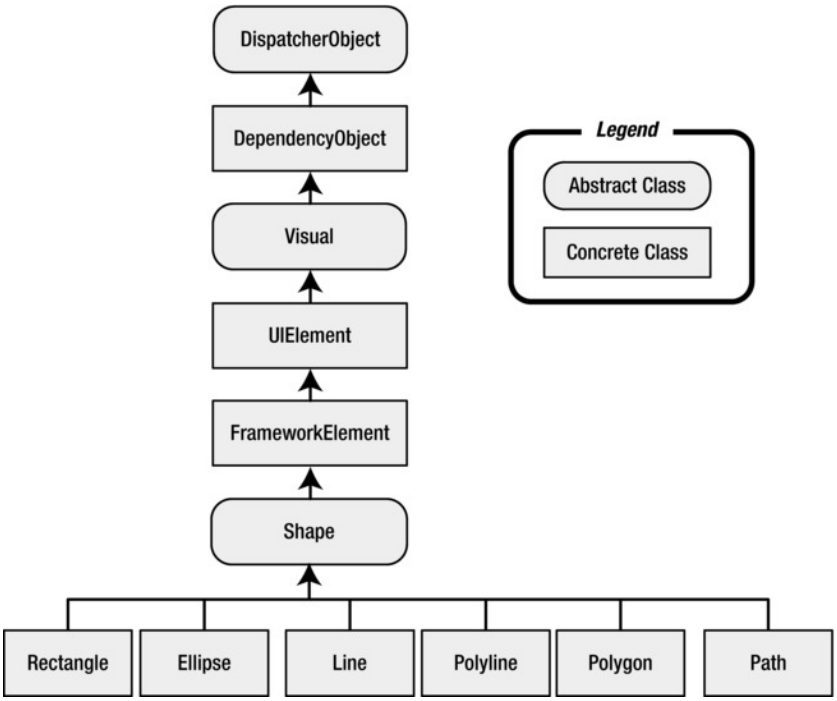

Figure 12-1. *The WPF shape classes*

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).
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 the contour of the edge where each dash line begins and ends.
StrokeLineJoin and StrokeMiterLimit	Determine the contour of the shape's corners. 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 by 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) by 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>
```

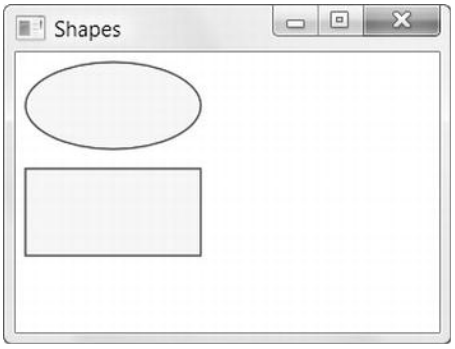

Figure 12-2. *Two simple shapes*

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 by 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.

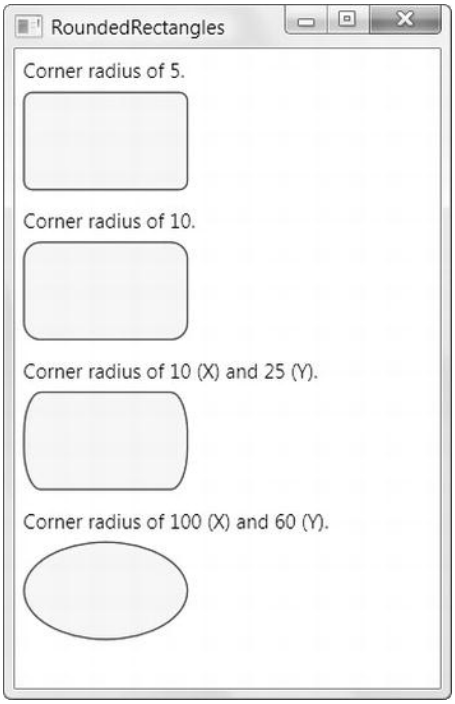

Figure 12-3. *Rounded corners*

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, in many cases 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), and 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 get only 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.

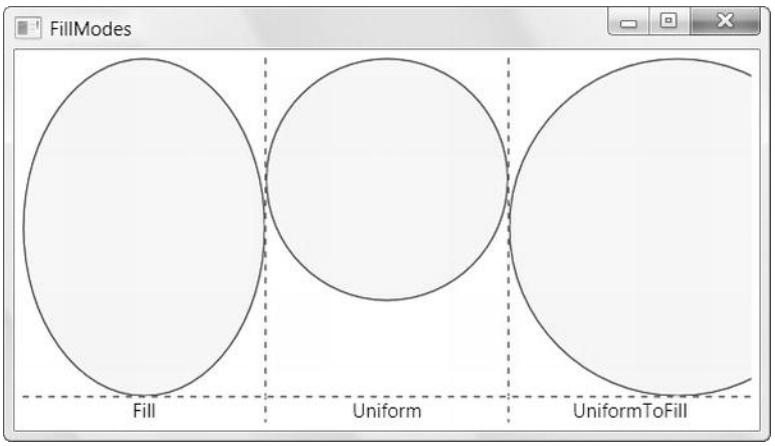

Figure 12-4. *Filling three cells in a Grid*

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 by 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).

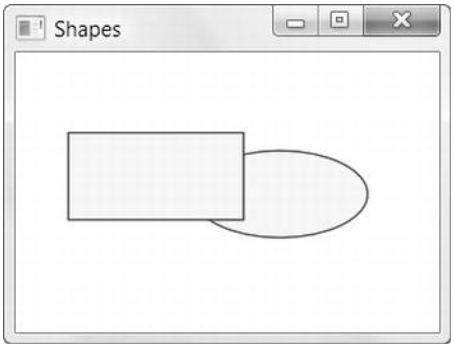

Figure 12-5. *Overlapping shapes in a Canvas*

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.

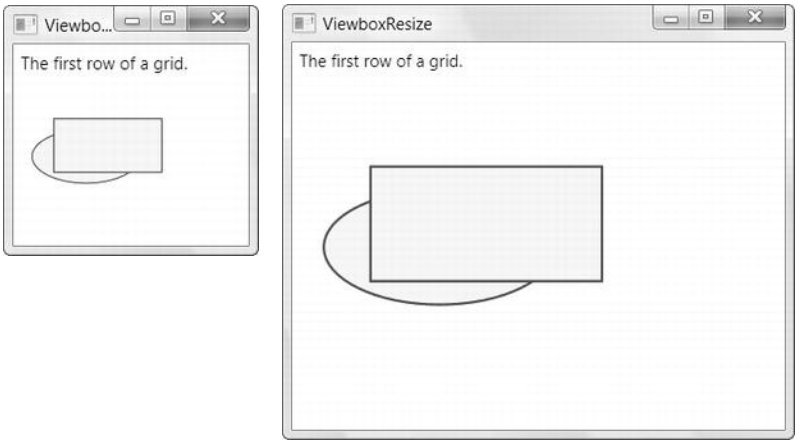

Figure 12-6. *Resizing with a Viewbox*

■ **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 by 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 200×150 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 0×0 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 0×0, 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 on top 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 by using the Points property. Technically, the Points property requires a PointCollection object, but you fill this collection in XAML by 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 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.

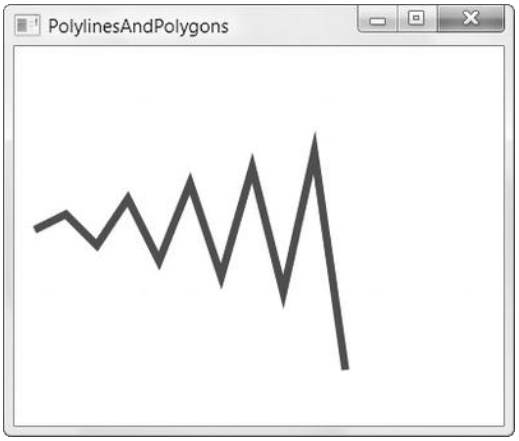

Figure 12-7. *A line with several segments*

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 by using the Fill brush. Figure 12-8 shows the previous Polyline as a Polygon with a yellow fill.

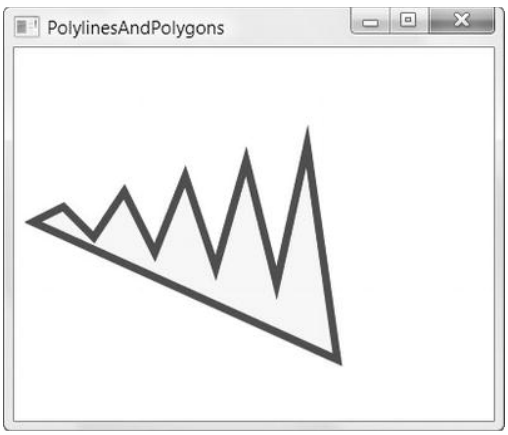

Figure 12-8. *A filled polygon*

■ **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 had an invisible line segment connecting the last point to the first point. This effect is of limited use.

In a simple shape with lines that 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.

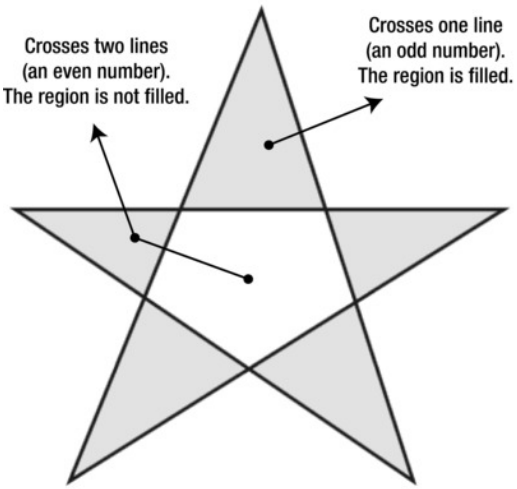

Figure 12-9. *Determining fill areas when FillRule is EvenOdd*

Every Polygon and Polyline includes a FillRule property, which lets you choose between two 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 crossed 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.)

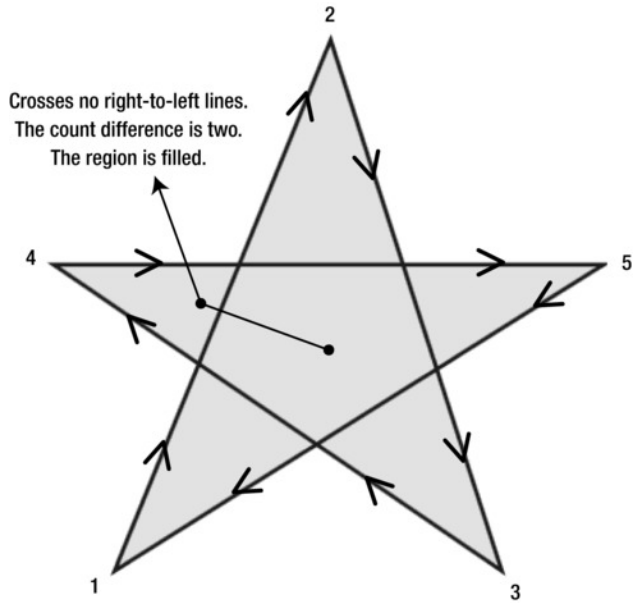

Figure 12-10. *Determining fill areas when FillRule is Nonzero*

■ **Note** If an odd number of lines are crossed to reach the outside of the shape, 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 by 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 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.

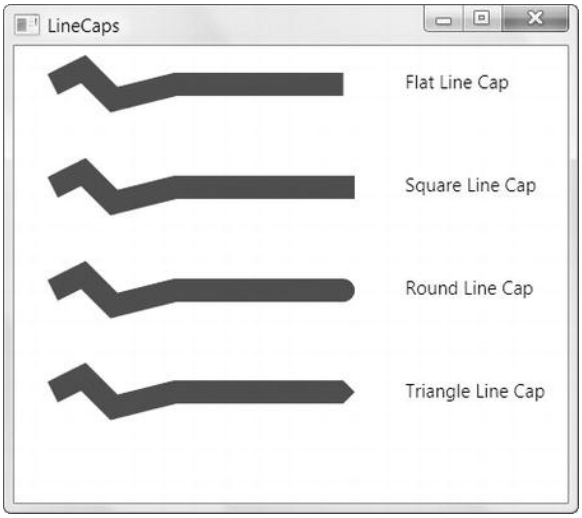

Figure 12-11. *Line caps*

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

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.

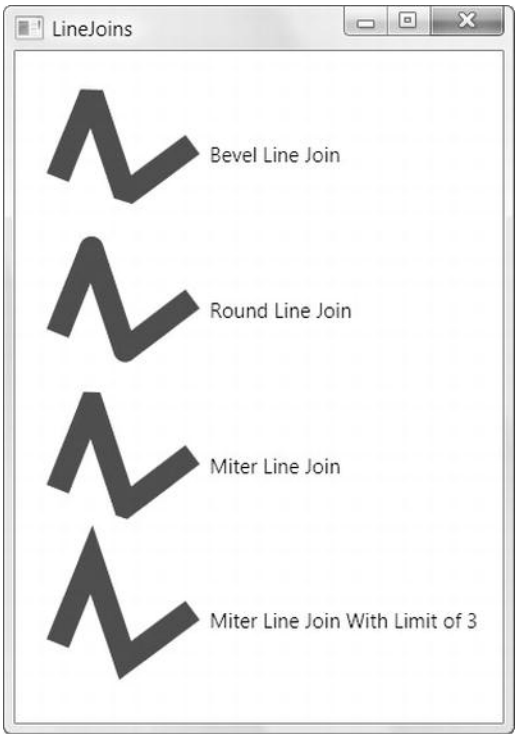

Figure 12-12. *Line joins*

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.

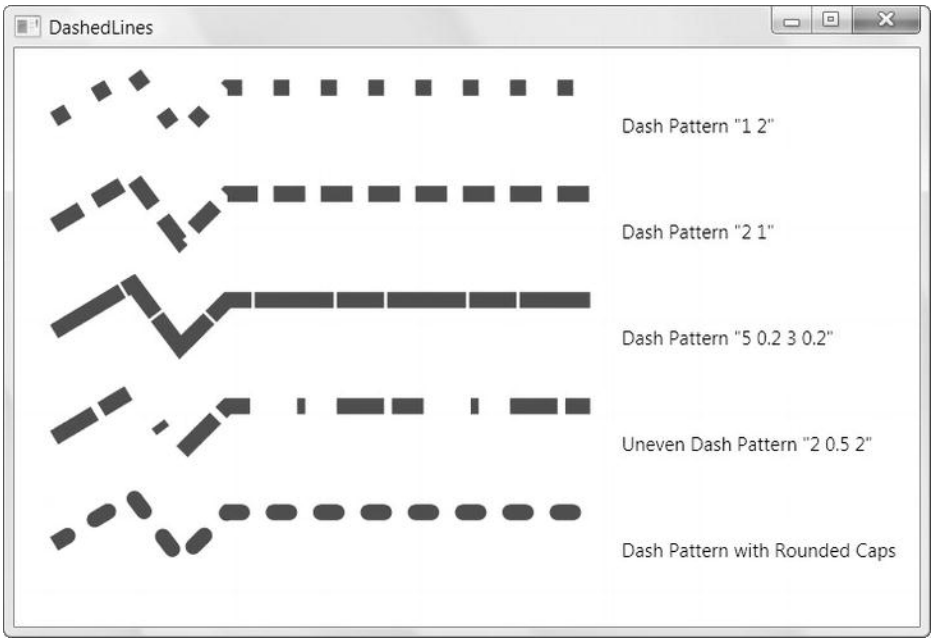

Figure 12-13. *Dashed lines*

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 by 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 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 63rd pixel with a value that's 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*

Using 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, several other classes 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).
RadialGradientBrush	Paints an area using a radial gradient fill, which is similar to a linear gradient, except that it radiates out in a circular pattern starting from a center point.
ImageBrush	Paints an area using an image that can be stretched, scaled, or tiled.
DrawingBrush	Paints an area using a Drawing object. This object can include shapes you've defined and bitmaps.
VisualBrush	Paints an area using a Visual object. Because all WPF elements derive from the Visual class, you can use this brush to copy part of your user interface (such as the face of a button) to another area. This is useful when creating fancy effects, such as partial reflections.
BitmapCacheBrush	Paints an area using the cached content from a Visual object. This makes it similar to VisualBrush, but more efficient if the graphical content needs to be reused in multiple places or repainted frequently.

The DrawingBrush is covered in Chapter 13, when you consider more optimized ways to deal with large numbers of graphics. In this section, you'll learn how to use the brushes that fill areas with gradients, images, and visual content copied from other elements.

■ **Note** All Brush classes are found in the System.Windows.Media namespace.

The SolidColorBrush

You've already seen how SolidColorBrush objects work with controls in Chapter 6. In most controls, setting the Foreground property paints the text color, and setting the Background property paints the space behind it. Shapes use similar but different properties: Stroke for painting the shape border and Fill for painting the shape interior.

As you've seen throughout this chapter, you can set both Stroke and Fill in XAML using color names, in which case the WPF parser automatically creates the matching SolidColorBrush object for you. You can also set Stroke and Fill in code, but you'll need to create the SolidColorBrush explicitly:

```
// Create a brush from a named color:
cmd.Background = new SolidColorBrush(Colors.AliceBlue);

// Create a brush from a system color:
cmd.Background = SystemColors.ControlBrush;

// Create a brush from color values:
int red = 0; int green = 255; int blue = 0;
cmd.Foreground = new SolidColorBrush(Color.FromRgb(red, green, blue));
```

The LinearGradientBrush

The LinearGradientBrush allows you to create a blended fill that changes from one color to another.

Here's the simplest possible gradient. It shades a rectangle diagonally from blue (in the top-left corner) to white (in the bottom-right corner):

```
<Rectangle Width="150" Height="100">
  <Rectangle.Fill>
    <LinearGradientBrush >
      <GradientStop Color="Blue" Offset="0"/>
      <GradientStop Color="White" Offset="1" />
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```

The top gradient in Figure 12-15 shows the result.

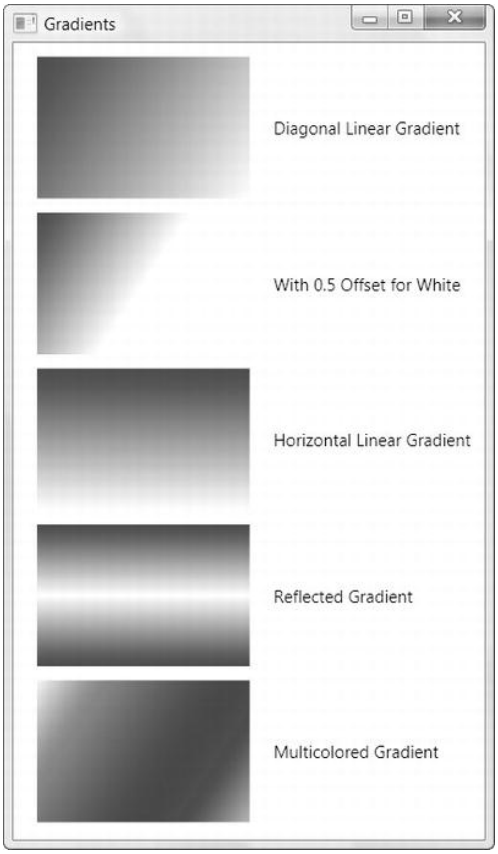

Figure 12-15. *A rectangle with different linear gradients*

To create this gradient, you need to add one GradientStop for each color. You also need to place each color in your gradient by using an Offset value from 0 to 1. In this example, the GradientStop for the blue color has an offset of 0, which means it's placed at the very beginning of the gradient. The GradientStop for the white color has an offset of 1, which places it at the end. By changing these values, you can adjust how quickly the gradient switches from one color to the other. For example, if you set the GradientStop for the white color to 0.5, the gradient would blend from blue (in the top-left corner) to white in the middle (the point between the two corners). The right side of the rectangle would be completely white. (The second gradient in Figure 12-15 shows this example.)

The previous markup creates a gradient with a diagonal fill that stretches from one corner to another. However, you might want to create a gradient that blends from top to bottom or side to side, or uses a different diagonal angle. You control these details by using the StartPoint and EndPoint properties of the LinearGradientBrush. These properties allow you to choose the point where the first color begins to change and the point where the color change ends with the final color. (The area in between is blended gradually.) However, there's one quirk: the coordinates you use for the starting and ending point aren't real coordinates. Instead, the LinearGradientBrush assigns the point (0, 0) to the top-left corner and (1, 1) to the bottom-right corner of the area you want to fill, no matter how high and wide it actually is.

To create a top-to-bottom horizontal fill, you can use a start point of (0, 0) for the top-left corner, and an end point of (0, 1), which represents the bottom-left corner. To create a side-to-side vertical fill (with no

slant), you can use a start point of (0, 0) and an end point of (1, 0) for the bottom-left corner. Figure 12-15 shows a horizontal gradient (it's the third one).

You can get a little craftier by supplying start points and end points that aren't quite aligned with the corners of your gradient. For example, you could have a gradient stretch from (0, 0) to (0, 0.5), which is a point on the left edge, halfway down. This creates a compressed linear gradient—one color starts at the top, blending to the second color in the middle. The bottom half of the shape is filled with the second color. But wait—you can change this behavior by using the LinearGradientBrush.SpreadMethod property. It's Pad by default (which means areas outside the gradient are given a solid fill with the appropriate color), but you can also use Reflect (to reverse the gradient, going from the second color back to the first) or Repeat (to duplicate the same color progression). Figure 12-15 shows the Reflect effect (it's the fourth gradient).

The LinearGradientBrush also allows you to create gradients with more than two colors by adding more than two GradientStop objects. For example, here's a gradient that moves through a rainbow of colors:

```
<Rectangle Width="150" Height="100">
  <Rectangle.Fill>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
      <GradientStop Color="Yellow" Offset="0.0" />
      <GradientStop Color="Red" Offset="0.25" />
      <GradientStop Color="Blue" Offset="0.75" />
      <GradientStop Color="LimeGreen" Offset="1.0" />
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```

The only trick is to set the appropriate offset for each GradientStop. For example, if you want to transition through five colors, you might give your first color an offset of 0, the second 0.25, the third 0.5, the fourth 0.75, and the fifth 1. Or if you want the colors to blend more quickly at the beginning and then end more gradually, you could set the offsets to 0, 0.1, 0.2, 0.4, 0.6, and 1.

Remember that Brushes aren't limited to shape drawing. You can substitute the LinearGradientBrush anytime you would use the SolidColorBrush—for example, when filling the background surface of an element (using the Background property), the foreground color of its text (using the Foreground property), or the fill of a border (using the BorderBrush property). Figure 12-16 shows an example of a gradient-filled TextBlock.

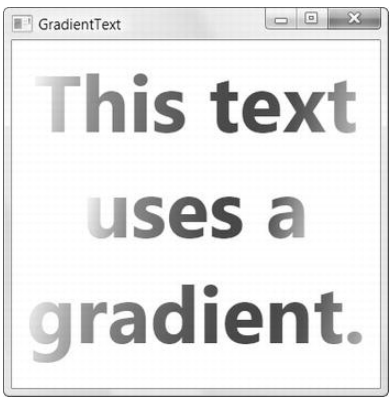

Figure 12-16. *Using the LinearGradientBrush to set the TextBlock.Foreground property*

The RadialGradientBrush

The RadialGradientBrush works similarly to the LinearGradientBrush. It also takes a sequence of colors with different offsets. As with the LinearGradientBrush, you can use as many colors as you want. The difference is how you place the gradient.

To identify the point where the first color in the gradient starts, you use the GradientOrigin property. By default, it's (0.5, 0.5), which represents the middle of the fill region.

■ **Note** As with the LinearGradientBrush, the RadialGradientBrush uses a proportional coordinate system that acts as though the top-left corner of your rectangular fill area is (0, 0) and the bottom-right corner is (1, 1). That means you can pick any coordinate from (0, 0) to (1, 1) to place the starting point of the gradient. In fact, you can even go beyond these limits if you want to locate the starting point outside the fill region.

The gradient radiates out from the starting point in a circular fashion. Eventually, your gradient reaches the edge of an inner gradient circle, where it ends. This center of this circle may or may not line up with the gradient origin, depending on the effect you want. The area beyond the edge of the inner gradient circle and the outermost edge of the fill region is given a solid fill using the last color that's defined in the RadialGradientBrush.GradientStops collection. Figure 12-17 illustrates how a radial gradient is filled.

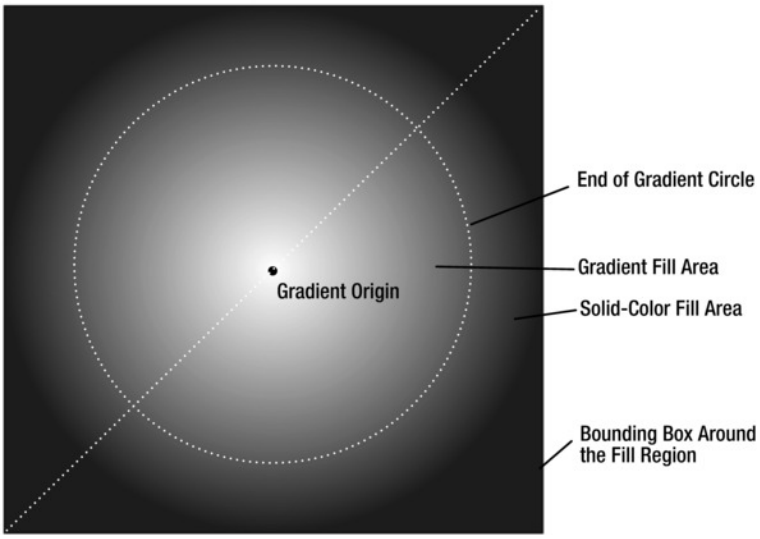

Figure 12-17. *How a radial gradient is filled*

You set the edge of the inner gradient circle by using three properties: Center, RadiusX, and RadiusY. By default, the Center property is (0.5, 0.5), which places the center of the limiting circle in the middle of your fill region and in the same position as the gradient origin.

RadiusX and RadiusY determine the size of the limiting circle, and by default, they're both set to 0.5. These values can be a bit unintuitive, because they're measured in relation to the *diagonal* span of your fill area (the length of an imaginary line stretching from the top-left corner to the bottom-right corner of your fill area). That means a radius of 0.5 defines a circle that has a radius that's half the length of this diagonal.

If you have a square fill region, you can use a dash of Pythagoras to calculate that this is about 0.7 times the width (or height) of your region. Thus, if you're filling a square region with the default settings, the gradient begins in the center and stretches to its outermost edge at about 0.7 times the width of the square.

■ **Note** If you trace the largest possible ellipse that fits in your fill area, that's the place where the gradient ends with your second color.

The radial gradient is a particularly good choice for filling rounded shapes and creating lighting effects. (Master artists use a combination of gradients to create buttons with a glow effect.) A common trick is to offset the GradientOrigin point slightly to create an illusion of depth in your shape. Here's an example:

```
<Ellipse Margin="5" Stroke="Black" StrokeThickness="1" Width="200" Height="200">
  <Ellipse.Fill>
    <RadialGradientBrush RadiusX="1" RadiusY="1" GradientOrigin="0.7,0.3">
      <GradientStop Color="White" Offset="0" />
      <GradientStop Color="Blue" Offset="1" />
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

Figure 12-18 shows this gradient, along with an ordinary radial gradient that has the standard GradientOrigin (0.5, 0.5).

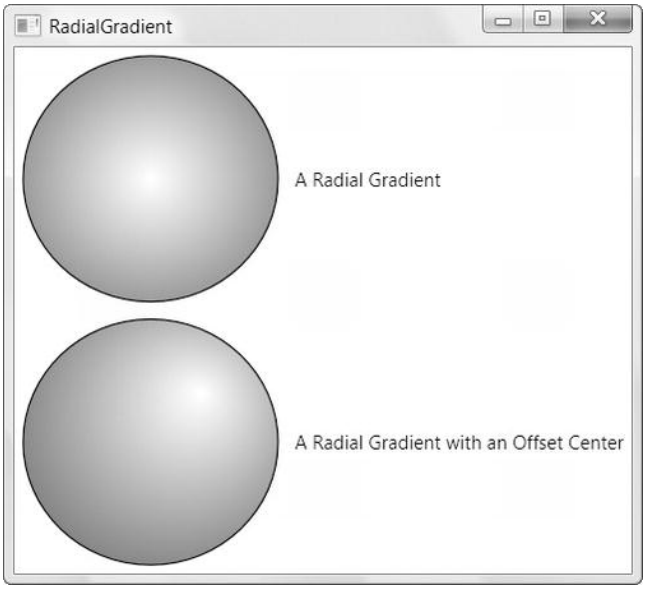

Figure 12-18. *Radial gradients*

The ImageBrush

The ImageBrush allows you to fill an area with a bitmap image. You can use most common file types, including BMP, PNG, GIF, and JPEG files. You identify the image you want to use by setting the ImageSource property. For example, this brush paints the background of a Grid by using an image named logo.jpg that's included in the assembly as a resource:

```
<Grid>
  <Grid.Background>
    <ImageBrush ImageSource="logo.jpg"></ImageBrush>
  </Grid.Background>
</Grid>
```

The ImageBrush.ImageSource property works in the same way as the Source property of the Image element, which means you can also set it by using a URI that points to a resource, an external file, or a web location. You can also create an ImageBrush that uses XAML-defined vector content by supplying a DrawingImage object for the ImageSource property. You might take this approach to reduce overhead (by avoiding the more costly Shape-derived classes), or if you want to use a vector image to create a tiled pattern. You'll learn more about the DrawingImage class in Chapter 13.

■ **Note** WPF respects any transparency information that it finds in an image. For example, WPF supports transparent areas in a GIF file and transparent or partially transparent areas in a PNG file.

In this example, the ImageBrush is used to paint the background of a cell. As a result, the image is stretched to fit the fill area. If the Grid is larger than the original size of the image, you may see resizing artifacts in your image (such as a general fuzziness). If the shape of the Grid doesn't match the aspect ratio of the picture, the picture will be distorted to fit.

You can control this behavior by modifying the ImageBrush.Stretch property, and assigning one of the values listed in Table 12-2, shown earlier in the chapter. For example, use Uniform to scale the image to fit the container, but keep the aspect ratio or None to paint the image at its natural size (in which case, part of it may be clipped to fit).

■ **Note** Even with a Stretch of None setting, your image may still be scaled. For example, if you've set your Windows system DPI setting to 120 dpi (also known as *large fonts*), WPF will scale up your bitmap proportionately. This may introduce some fuzziness, but it's a better solution than having your image sizes (and the alignment of your overall user interface) change on monitors with different DPI settings.

If the image is painted smaller than the fill region, the image is aligned according to the AlignmentX and AlignmentY properties. The unfilled area is left transparent. This occurs if you're using Uniform scaling and the region you're filling has a different shape (in which case, you'll get blank bars on the top or the sides). It also occurs if you're using None and the fill region is larger than the image.

You can also use the Viewbox property to clip out a smaller portion of the picture that you're interested in using. To do so, you specify four numbers that describe the rectangle you want to clip out of the source picture. The first two identify the top-left corner where your rectangle begins, and the following two numbers specify the width and height of the rectangle. The only catch is that the Viewbox uses a relative coordinate system, just like the gradient brushes. This coordinate system designates the top-left corner of your picture as (0, 0) and the bottom-right corner as (1, 1).

To understand how Viewbox works, take a look at this markup:

```
<ImageBrush ImageSource="logo.jpg" Stretch="Uniform"
  Viewbox="0.4,0.5 0.2,0.2"></ImageBrush>
```

Here, the Viewbox starts at (0.4, 0.5), which is almost halfway into the picture. (Technically, the X coordinate is 0.4× width and the Y coordinate is 0.5× width.) The rectangle then extends to fill a small box that's 20 percent as wide and tall as the total image (technically, the rectangle is 0.2× width long and 0.2× height tall). The cropped-out portion is then stretched or centered, based on the Stretch, AlignmentX, and AlignmentY properties. Figure 12-19 shows two rectangles that use different ImageBrush objects to fill themselves. The topmost rectangle shows the full image, while the rectangle underneath uses the Viewbox to magnify a small section. Both are given a solid black border.

■ **Note** The Viewbox property is occasionally useful when reusing parts of the same picture in different ways to create certain effects. However, if you know in advance that you need to use only a portion of an image, it obviously makes more sense to crop it down in your favorite graphics software.

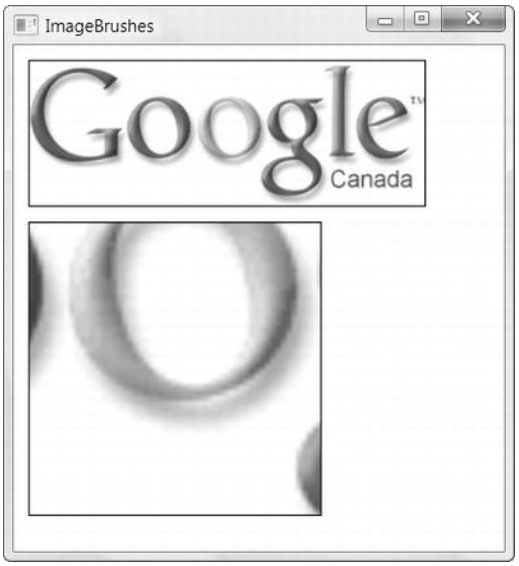

Figure 12-19. *Different ways to use an ImageBrush*

A Tiled ImageBrush

An ordinary ImageBrush isn't all that exciting. However, you can get some interesting effects by tiling your image across the surface of the brush.

When tiling an image, you have two options:

Proportionally sized tiles: Your fill area always has the same number of tiles. The tiles expand and shrink to fit the fill region.

Fixed-sized tiles: Your tiles are always the same size. The size of your fill area determines the number of tiles that appear.

Figure 12-20 compares the difference when a tile-filled rectangle is resized.

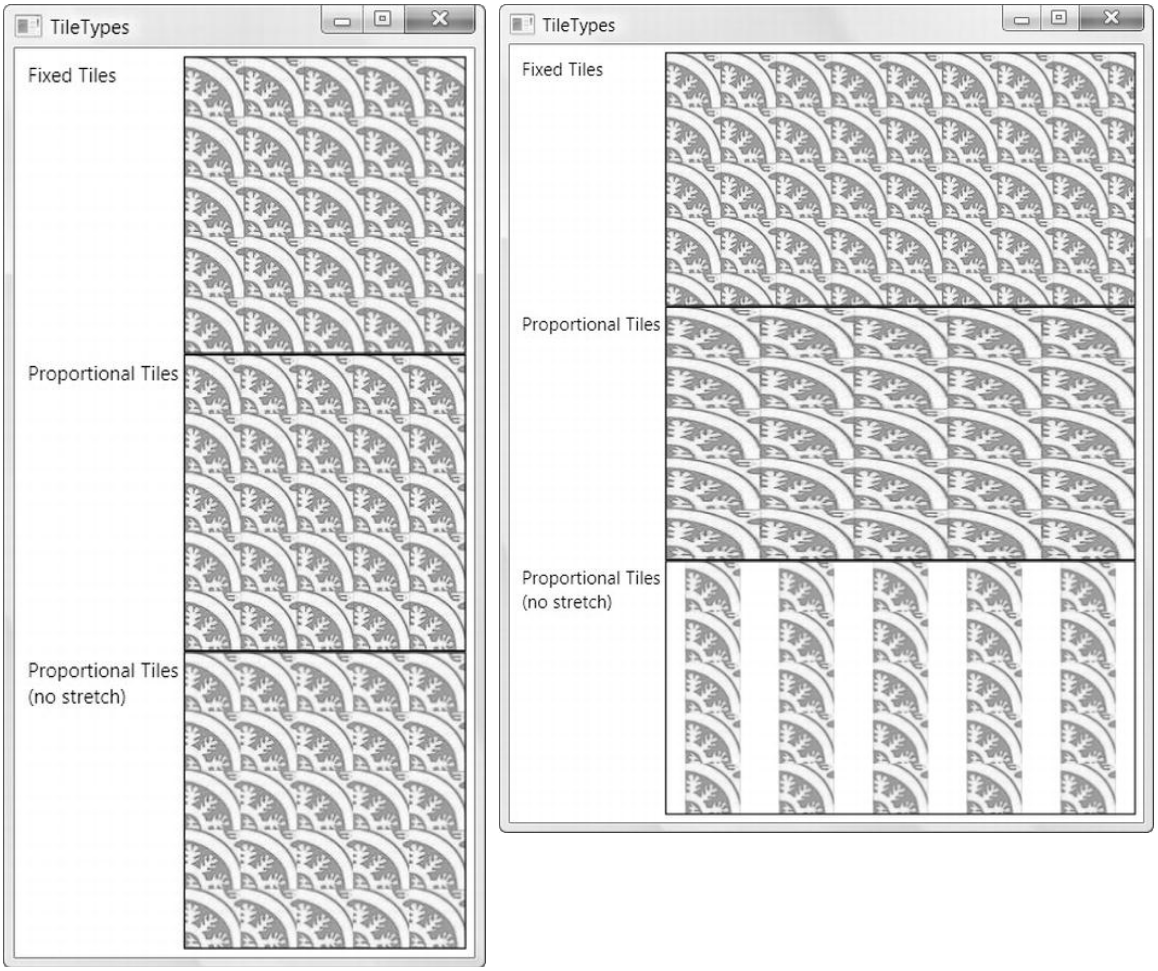

Figure 12-20. *Different ways to tile a rectangle*

To tile an image, you need to set the ImageSource property (to identify the image you want to tile) and the Viewport, ViewportUnits, and TileMode properties. These latter three properties determine the size of your tile and the way it's arranged.

You use the Viewport property to set the size of each tile. To use proportionately sized tiles, ViewportUnits must be set to RelativeToBoundingBox (which is the default). Then you define the tile size by using a proportional coordinate system that stretches from 0 to 1 in both dimensions. In other words, a tile that has a top-left corner at (0, 0) and a bottom-right corner at (1, 1) occupies the entire fill area. To get a tiled pattern, you need to define a Viewport that's smaller than the total size of the fill area, as shown here:

```
<ImageBrush ImageSource="tile.jpg" TileMode="Tile"
  Viewport="0,0 0.5,0.5"></ImageBrush>
```

This creates a Viewport box that begins at the top-left corner of the fill area (0, 0) and stretches down to the midpoint (0.5, 0.5). As a result, the fill region will always hold four tiles, no matter how big or small it is. This behavior is nice because it ensures that there's no danger of having part of a tile chopped off at the edge of a shape. (Of course, this isn't the case if you're using the ImageBrush to fill a nonrectangular area.)

Because the tile in this example is relative to the size of the fill area, a larger fill area will use a larger tile, and you may wind up with some blurriness from image resizing. Furthermore, if your fill area isn't perfectly square, the relative coordinate system is squashed accordingly, so each tiled square becomes a rectangle. This behavior is shown in the second tiled pattern in Figure 12-20.

You can alter this behavior by changing the Stretch property (which is Fill by default). Use None to ensure that tiles are never distorted and keep their proper shape. However, if the fill area isn't square, whitespace will appear between your tiles. This detail is shown in the third tiled pattern in Figure 12-20.

A third option is to use a Stretch value of UniformToFill, which crops your tile image as needed. That way, your tiled image keeps the correct aspect ratio and you don't have any whitespace between your tiles. However, if your fill area isn't a square, you won't see the complete tile image.

The automatic tile resizing is a nifty feature, but there's a price to pay. Some bitmaps may not resize properly. To some extent, you can prepare for this situation by supplying a bitmap that's bigger than what you need, but this technique can result in a blurrier bitmap when it's scaled down.

An alternate solution is to define the size of your tile in absolute coordinates, based on the size of your original image. To take this step, you set ViewportUnits to Absolute (instead of RelativeToBoundBox). Here's an example that defines a 32× 32 unit size for each tile and starts them at the top-left corner:

```
<ImageBrush ImageSource="tile.jpg" TileMode="Tile"
  ViewportUnits="Absolute" Viewport="0,0 32,32"></ImageBrush>
```

This type of tiled pattern is shown in the first rectangle in Figure 12-20. The drawback here is that the height and width of your fill area must be divisible by 32. Otherwise, you'll get a partial tile at the edge. If you're using the ImageBrush to fill a resizable element, there's no way around this problem, so you'll need to accept that the tiles won't always line up with the edges of the fill region.

So far, all the tiled patterns you've seen have used a TileMode value of Tile. You can change the TileMode to set how alternate tiles are flipped. Table 12-4 lists your choices.

Table 12-4. *Values from the TileMode Enumeration*

Name	Description
Tile	Copies the image across the available area
FlipX	Copies the image, but flips each second column vertically
FlipY	Copies the image, but flips each second row horizontally
FlipXY	Copies the image, but flips each second column vertically and each second row horizontally

This flipping behavior is often useful if you need to make tiles blend more seamlessly. For example, if you use FlipX, tiles that are side by side will always line up seamlessly. Figure 12-21 compares the tiling options.

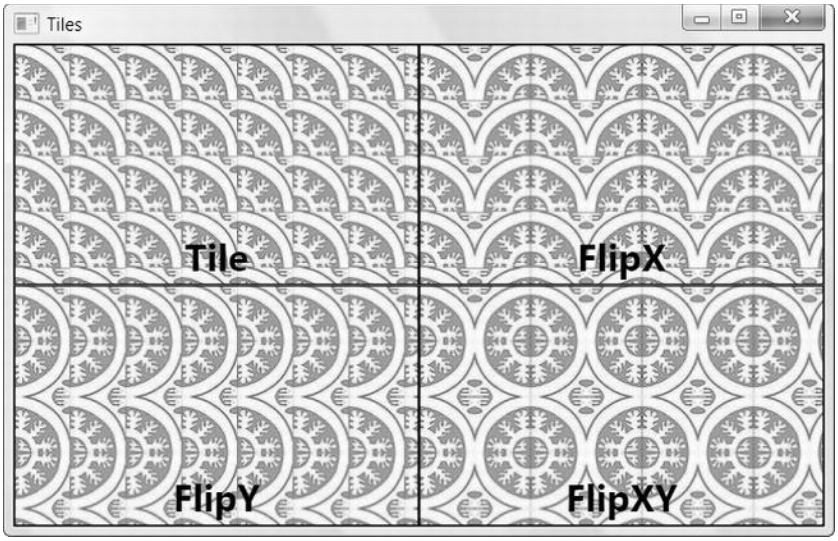

Figure 12-21. *Flipping tiles*

The VisualBrush

The VisualBrush is an unusual brush that allows you to take the visual content of an element and use it to fill any surface. For example, using a VisualBrush, you could copy the appearance of a button in a window to a region somewhere else in that same window. However, the button copy won't be clickable or interactive in any way. It's simply a copy of how your element looks.

For example, here's a snippet of markup that defines a button and a VisualBrush that duplicates the button:

```
<Button Name="cmd" Margin="3" Padding="5">Is this a real button?</Button>
<Rectangle Margin="3" Height="100">
  <Rectangle.Fill>
    <VisualBrush Visual="{Binding ElementName=cmd}"></VisualBrush>
  </Rectangle.Fill>
</Rectangle>
```

Although you could define the element you want to use in the VisualBrush itself, it's much more common to use a binding expression to refer to an element in the current window, as in this example. Figure 12-22 shows the original button (at the top of the window) and several differently shaped regions that are painted with a VisualBrush based on that button.

Figure 12-22. *Copying the visual for a button*

A VisualBrush watches for changes in the appearance of your element. For example, if you copy the visual for a button, and that button then receives focus, the VisualBrush repaints its fill area with the new visual—a focused button. The VisualBrush derives from TileBrush, so it also supports all the cropping, stretching, and flipping features you learned about in the previous section. If you combine these details with the transforms that you will learn about later in this chapter, you can easily use a VisualBrush to take element content and manipulate it beyond all recognition.

Because the content of a VisualBrush isn't interactive, you might wonder what purpose it has. In fact, the VisualBrush is useful in a number of situations where you need to create static content that duplicates the "real" content that's featured elsewhere. For example, you can take an element that contains a significant amount of nested content (even an entire window), shrink it down to a smaller size, and use it for a live preview. Some document programs do this to show formatting, Internet Explorer uses it to show previews of the documents in different tabs on the Quick Tabs view (hit Ctrl+Q), and Windows uses it to show previews of different applications in the taskbar.

You can use a VisualBrush in combination with animation to create certain effects (such as a document shrinking down to the bottom of your main application window). The VisualBrush is also the foundation for one of WPF's most notoriously overused effects—the live reflection, which you'll see in the following section (and the even worse live reflection of video content, which you'll see in Chapter 26).

The BitmapCacheBrush

The BitmapCacheBrush resembles the VisualBrush in many ways. While the VisualBrush provides a Visual property that refers to another element, the BitmapCacheBrush includes a Target property that serves the same purpose.

The key difference is that the BitmapCacheBrush takes the visual content (after it has been altered by any transforms, clipping, effects, and opacity settings) and asks the video card to store it in video memory. This way, the content can be redrawn quickly when needed, without requiring any extra work from WPF.

To configure bitmap caching, you set the BitmapCacheBrush.BitmapCache property (using, predictably, a BitmapCache object). Here's the simplest possible usage:

```
<Button Name="cmd" Margin="3" Padding="5">Is this a real button?</Button>
<Rectangle Margin="3" Height="100">
  <Rectangle.Fill>
    <BitmapCacheBrush Target="{Binding ElementName=cmd}"
     BitmapCache="BitmapCache"></BitmapCacheBrush>
  </Rectangle.Fill>
</Rectangle>
```

The BitmapCacheBrush has a significant drawback: the initial step of rendering the bitmap and copying it to video memory takes a short but noticeable amount of extra time. If you use the BitmapCacheBrush in a window, you'll probably notice that there's a lag before the window draws itself for the first time, while the BitmapCacheBrush is rendering and copying its bitmap. For this reason, the BitmapCacheBrush isn't much help in a traditional window.

However, bitmap caching is worth considering if you're making heavy use of animation in your user interface. That's because an animation can force your window to be repainted many times each second. If you have complex vector content, it may be faster to paint it from a cached bitmap than to redraw it from scratch. But even in this situation, you shouldn't jump to the BitmapCacheBrush just yet. You're much more likely to apply caching by setting the higher-level UIElement.CacheMode property on each element you want to cache (a technique described in Chapter 15). In this case, WPF uses the BitmapCacheBrush behind the scenes to get the same effect, but with less work.

Based on these details, it may seem that the BitmapCacheBrush isn't particularly useful on its own. However, it may make sense if you have a single piece of complex visual content that you need to paint in several places. In this case, you can save memory by caching it once with the BitmapCacheBrush, rather than separately for each element. Once again, the savings are not likely to be worth it, unless your user interface is also using animation. To learn more about bitmap caching and when to use it, refer to Chapter 15.

Using Transforms

Many drawing tasks can be made simpler with the use of a *transform*—an object that alters the way a shape or element is drawn by quietly shifting the coordinate system it uses. In WPF, transforms are represented by classes that derive from the abstract System.Windows.Media.Transform class, as listed in Table 12-5.

Table 12-5. *Transform Classes*

Name	Description	Important Properties
TranslateTransform	Displaces your coordinate system by some amount. This transform is useful if you want to draw the same shape in different places.	X, Y
RotateTransform	Rotates your coordinate system. The shapes you draw normally are turned around a center point you choose.	Angle, CenterX, CenterY
ScaleTransform	Scales your coordinate system up or down, so that your shapes are drawn smaller or larger. You can apply different degrees of scaling in the X and Y dimensions, thereby stretching or compressing your shape.	ScaleX, ScaleY, CenterX, CenterY

Name	Description	Important Properties
SkewTransform	Warps your coordinate system by slanting it a number of degrees. For example, if you draw a square, it becomes a parallelogram.	AngleX, AngleY, CenterX, CenterX
MatrixTransform	Modifies your coordinate system by using matrix multiplication with the matrix you supply. This is the most complex option; it requires some mathematical skill.	Matrix
TransformGroup	Combines multiple transforms so they can all be applied at once. The order in which you apply transformations is important because it affects the final result. For example, rotating a shape (with RotateTransform) and then moving it (with TranslateTransform) sends the shape off in a different direction than if you move it and *then* rotate it.	N/A

Technically, all transforms use matrix math to alter the coordinates of your shape. However, using the prebuilt transforms such as TranslateTransform, RotateTransform, ScaleTransform, and SkewTransform is far simpler than using the MatrixTransform and trying to work out the correct matrix for the operation you want to perform. When you perform a series of transforms with the TransformGroup, WPF fuses your transforms together into a single MatrixTransform, ensuring optimal performance.

■ **Note** All transforms derive from Freezable (through the Transform class). That means they have automatic change notification support. If you change a transform that's being used in a shape, the shape will redraw itself immediately.

Transforms are one of those quirky concepts that turn out to be extremely useful in a variety of contexts. Some examples include the following:

Angling a shape: So far, you've been stuck with horizontally aligned rectangles, ellipses, lines, and polygons. Using the RotateTransform, you can turn your coordinate system to create certain shapes more easily.

Repeating a shape: Many drawings are built using a similar shape in several places. Using a transform, you can take a shape and then move it, rotate it, resize it, and so on.

■ **Tip** To use the same shape in multiple places, you'll need to duplicate the shape in your markup (which isn't ideal), use code (to create the shape programmatically), or use the Path shape described in Chapter 13. The Path shape accepts Geometry objects, and you can store a Geometry object as a resource so it can be reused throughout your markup.

Animation: You can create sophisticated effects with the help of a transform, such as by rotating a shape, moving it from one place to another, and warping it dynamically.

You'll use transforms throughout this book, particularly when you create animations (Chapter 16) and manipulate 3-D content (Chapter 27). For now, you can learn all you need to know by considering how to apply a basic transform to an ordinary shape.

Transforming Shapes

To transform a shape, you assign the RenderTransform property to the transform object you want to use. Depending on the transform object you're using, you'll need to fill in different properties to configure it, as detailed in Table 12-5.

For example, if you're rotating a shape, you need to use the RotateTransform, and supply the angle in degrees. Here's an example that rotates a square by 25 degrees:

```
<Rectangle Width="80" Height="10" Stroke="Blue" Fill="Yellow"
  Canvas.Left="100" Canvas.Top="100">
  <Rectangle.RenderTransform>
    <RotateTransform Angle="25" />
  </Rectangle.RenderTransform>
</Rectangle>
```

When you transform a shape in this way, you rotate it about the shape's origin (the top-left corner). Figure 12-23 illustrates this by rotating the same square 25, 50, 75, and then 100 degrees.

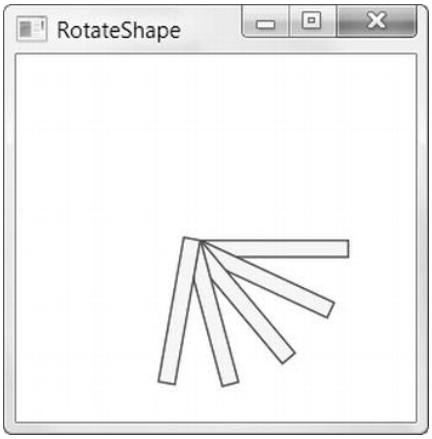

Figure 12-23. *Rotating a rectangle four times*

Sometimes you'll want to rotate a shape around a different point. The RotateTransform, like many other transform classes, provides a CenterX property and a CenterY property. You can use these properties to indicate the center point around which the rotation should be performed. Here's a rectangle that uses this approach to rotate itself 25 degrees around its center point:

```
<Rectangle Width="80" Height="10" Stroke="Blue" Fill="Yellow"
  Canvas.Left="100" Canvas.Top="100">
  <Rectangle.RenderTransform>
    <RotateTransform Angle="25" CenterX="45" CenterY="5" />
  </Rectangle.RenderTransform>
</Rectangle>
```

Figure 12-24 shows the result of performing the same sequence of rotations featured in Figure 12-23, but around the designated center point.

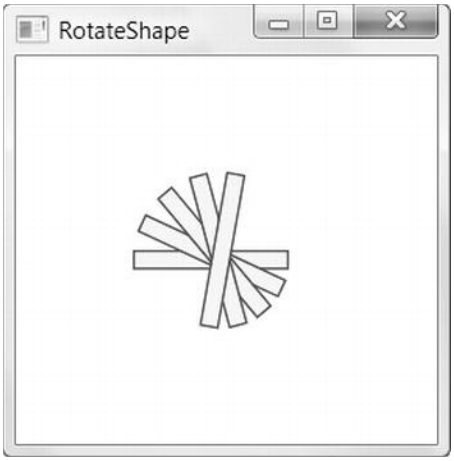

Figure 12-24. *Rotating a rectangle around its middle*

There's a clear limitation to using the CenterX and CenterY properties of the RotateTransform. These properties are defined using absolute coordinates, which means you need to know the exact center point of your content. If you're displaying dynamic content (for example, pictures of varying dimensions or elements that can be resized), this introduces a problem. Fortunately, WPF has a solution with the handy RenderTransformOrigin property, which is supported by all shapes. This property sets the center point by using a proportional coordinate system that stretches from 0 to 1 in both dimensions. In other words, the point (0, 0) is designated as the top-left corner and (1, 1) is the bottom-right corner. (If the shape region isn't square, the coordinate system is stretched accordingly.)

With the help of the RenderTransformOrigin property, you can rotate any shape around its center point by using markup like this:

```
<Rectangle Width="80" Height="10" Stroke="Blue" Fill="Yellow"
  Canvas.Left="100" Canvas.Top="100" RenderTransformOrigin="0.5,0.5">
  <Rectangle.RenderTransform>
    <RotateTransform Angle="25" />
  </Rectangle.RenderTransform>
</Rectangle>
```

This works because the point (0.5, 0.5) designates the center of the shape, regardless of its size. In practice, RenderTransformOrigin is generally more useful than the CenterX and CenterY properties, although you can use either one (or both) depending on your needs.

■ **Tip** You can use values greater than 1 or less than 0 when setting the RenderTransformOrigin property to designate a point that appears outside the bounding box of your shape. For example, you can use this technique with a RotateTransform to rotate a shape in a large arc around a very distant point, such as (5, 5).

Transforming Elements

The RenderTransform and RenderTransformOrigin properties aren't limited to shapes. In fact, the Shape class inherits them from the UIElement class, which means they're supported by all WPF elements, including buttons, text boxes, the TextBlock, entire layout containers full of content, and so on. Amazingly, you can rotate, skew, and scale any piece of WPF user interface (although in most cases you shouldn't).

RenderTransform isn't the only transform-related property that's defined in the base WPF classes. The FrameworkElement also defines a LayoutTransform property. LayoutTransform alters the element in the same way, but it performs its work before the layout pass. This results in slightly more overhead, but it's critical if you're using a layout container to provide automatic layout with a group of controls. (The shape classes also include the LayoutTransform property, but you'll rarely need to use it. You'll usually place your shapes specifically using a container such as the Canvas, rather than using automatic layout.)

To understand the difference, consider Figure 12-25, which includes two StackPanel containers (represented by the shaded areas), both of which contain a rotated button and a normal button. The rotated button in the first StackPanel uses the RenderTransform approach. The StackPanel lays out the two buttons as though the first button were positioned normally, and the rotation happens just before the button is rendered. As a result, the rotated button overlaps the one underneath. In the second StackPanel, the rotated button uses the LayoutTransform approach. The StackPanel gets the bounds that are required for the rotated button and lays out the second button accordingly.

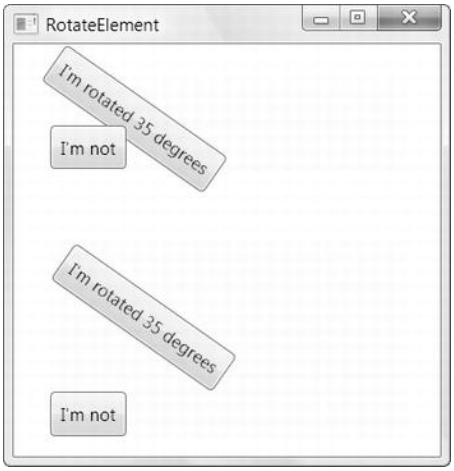

Figure 12-25. *Rotating buttons*

A few rare elements can't be transformed because their rendering work isn't native to WPF. Two examples are the WindowsFormsHost, which lets you place a Windows Forms control in a WPF window (a feat demonstrated in Chapter 30) and the WebBrowser element, which allows you to show HTML content.

To a certain degree, WPF elements aren't aware that they're being modified when you set the LayoutTransform or RenderTransform properties. Notably, transforms don't affect the ActualHeight and ActualWidth properties of the element, which continue to report their untransformed dimensions. This is part of how WPF ensures that features such as flow layout and margins continue to work with the same behavior, even when you apply one or more transforms.

Transparency

WPF supports true transparency. That means if you layer several shapes (or other elements) on top of one another and give them all varying levels of transparency, you'll see exactly what you expect. At its simplest, this feature gives you the ability to create graphical backgrounds that "show through" the elements you place on top. At its most complex, this feature allows you to create multilayered animations and other effects that would be extremely difficult in other frameworks.

Making an Element Partially Transparent

There are several ways to make an element semitransparent:

Set the Opacity property of your element: Every element, including shapes, inherits the Opacity property from the base UIElement class. *Opacity* is a fractional value from 0 to 1, where 1 is completely solid (the default) and 0 is completely transparent. For example, an opacity of 0.9 creates a 90 percent visible (10 percent transparency) effect. When you set the opacity this way, it applies to the visual content of the entire element.

Set the Opacity property of your brush: Every brush also inherits an Opacity property from the base Brush class. You can set this property from 0 to 1 to control the opacity of the content the brush paints—whether it's a solid color, gradient, or some sort of texture or image. Because you use a different brush for the Stroke and Fill properties of a shape, you can give different amounts of transparency to the border and surface area.

Use a color that has a nonopaque alpha value: Any color that has an alpha value less than 255 is semitransparent. For example, you could use a partially transparent color in a SolidColorBrush, and use that to paint the foreground content or background surface of an element. In some situations, using partially transparent colors will perform better than setting the Opacity property.

Figure 12-26 shows an example that has several semitransparent layers:

- The window has an opaque white background.
- The top-level StackPanel that contains all the elements has an ImageBrush that applies a picture. The Opacity of this brush is reduced to lighten it, allowing the white window background to show through.
- The first button uses a semitransparent red background color. (Behind the scenes, WPF creates a SolidColorBrush to paint this color.) The image shows through in the button background, but the text is opaque.
- The label (under the first button) is used as is. By default, all labels have a completely transparent background color.
- The text box uses opaque text and an opaque border but a semitransparent background color.
- Another StackPanel under the text box uses a TileBrush to create a pattern of happy faces. The TileBrush has a reduced Opacity, so the other background shows through. For example, you can see the sun at the bottom-right corner of the form.

- In the second StackPanel is a TextBlock with a completely transparent background (the default) and semitransparent white text. If you look carefully, you can see both backgrounds show through under some letters.

Figure 12-26. *A window with several semitransparent layers*

Here are the contents of the window in XAML:

```
<StackPanel Margin="5">
  <StackPanel.Background>
    <ImageBrush ImageSource="celestial.jpg" Opacity="0.7" />
  </StackPanel.Background>

  <Button Foreground="White" FontSize="16" Margin="10"
    BorderBrush="White" Background="#60AA4030"
    Padding="20">A Semi-Transparent Button</Button>
  <Label Margin="10" FontSize="18" FontWeight="Bold" Foreground="White">
    Some Label Text</Label>
  <TextBox Margin="10" Background="#AAAAAAAA" Foreground="White"
    BorderBrush="White">A semi-transparent text box</TextBox>

  <Button Margin="10" Padding="25" BorderBrush="White">
    <Button.Background>
      <ImageBrush ImageSource="happyface.jpg" Opacity="0.6"
       TileMode="Tile" Viewport="0,0,0.1,0.3"/>
    </Button.Background>
```

```
    <StackPanel>
      <TextBlock Foreground="#75FFFFFF" TextAlignment="Center"
       FontSize="30" FontWeight="Bold" TextWrapping="Wrap">
       Semi-Transparent Layers</TextBlock>
    </StackPanel>
  </Button>
</StackPanel>
```

Transparency is a popular WPF feature. In fact, it's so easy and works so well that it's a bit of a WPF user-interface cliché. For that reason, be careful not to overuse it.

Using Opacity Masks

The Opacity property makes *all* the content of an element partially transparent. The OpacityMask property gives you more flexibility. It makes specific regions of an element transparent or partially transparent, allowing you to achieve a variety of common and exotic effects. For example, you can use it to fade a shape gradually into transparency.

The OpacityMask property accepts any brush. The alpha channel of the brush determines where the transparency occurs. For example, if you use a SolidColorBrush that's set to a transparent color for your OpacityMask, your entire element will disappear. If you use a SolidColorBrush that's set to use a nontransparent color, your element will remain completely visible. The other details of the color (the red, green, and blue components) aren't important and are ignored when you set the OpacityMask property.

Using OpacityMask with a SolidColorBrush doesn't make much sense, because you can accomplish the same effect more easily with the Opacity property. However, OpacityMask becomes more useful when you use more exotic types of brushes, such as the LinearGradient or RadialGradientBrush. Using a gradient that moves from a solid to a transparent color, you can create a transparency effect that fades in over the surface of your element, like the one used by this button:

```
<Button FontSize="14" FontWeight="Bold">
  <Button.OpacityMask>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
      <GradientStop Offset="0" Color="Black"></GradientStop>
      <GradientStop Offset="1" Color="Transparent"></GradientStop>
    </LinearGradientBrush>
  </Button.OpacityMask>
  <Button.Content>A Partially Transparent Button</Button.Content>
</Button>
```

Figure 12-27 shows this button over a window that displays a picture of a grand piano.

Figure 12-27. *A button that fades from solid to transparent*

You can also use the OpacityMask property in conjunction with the VisualBrush to create a reflection effect. For example, the following markup creates one of WPF's most common effects—a text box with mirrored text. As you type, the VisualBrush paints a reflection of the text underneath. The VisualBrush paints a rectangle that uses the OpacityMask property to fade the reflection out, which distinguishes it from the real element above:

```
<TextBox Name="txt" FontSize="30">Here is some reflected text</TextBox>
<Rectangle Grid.Row="1" RenderTransformOrigin="1,0.5">
  <Rectangle.Fill>
    <VisualBrush Visual="{Binding ElementName=txt}"></VisualBrush>
  </Rectangle.Fill>
  <Rectangle.OpacityMask>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
      <GradientStop Offset="0.3" Color="Transparent"></GradientStop>
      <GradientStop Offset="1" Color="#44000000"></GradientStop>
    </LinearGradientBrush>
  </Rectangle.OpacityMask>
  <Rectangle.RenderTransform>
    <ScaleTransform ScaleY="-1"></ScaleTransform>
  </Rectangle.RenderTransform>
</Rectangle>
```

This example uses a LinearGradientBrush that fades from a completely transparent color to a partially transparent color, to make the reflected content more faded. It also adds a RenderTransform that flips the rectangle so the reflection is upside down. As a result of this transformation, the gradient stops must be defined in the reverse order. Figure 12-28 shows the result.

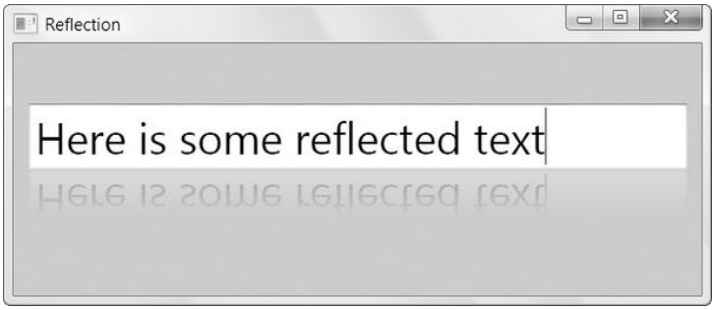

Figure 12-28. *VisualBrush + OpacityMask + RenderTransform = reflection effect*

Along with the gradient brushes and the VisualBrush, the OpacityMask property is often used with the DrawingBrush you'll learn about in the next chapter. This allows you to apply a shaped transparent region to an element.

The Last Word

In this chapter, you took a detailed look at WPF's support for basic 2-D drawing. You began by considering the simple shape classes. Next, you learned how to outline and fill shapes with brushes both simple and complex, and how to move, resize, rotate, and warp shapes with transforms. Finally, you took a quick look at transparency.

Your journey isn't finished yet. In the next chapter, you'll take a look at the Path, the most sophisticated of the shape classes, which lets you combine the shapes you've seen so far and add arcs and curves. You'll also consider how you can make more-efficient graphics with the help of WPF's Geometry and Drawing objects, and how you can export clip art from other programs.