Name	Description
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.

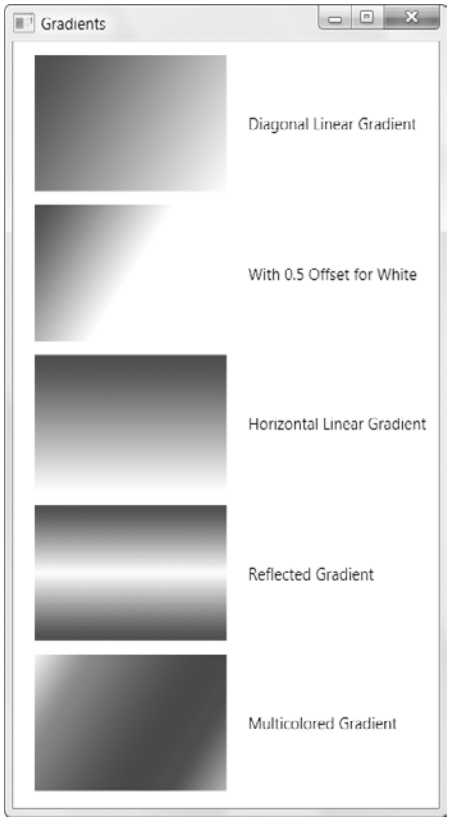

Figure 12-15. *A rectangle with different linear gradients*

To create this gradient, you need to add one GradientStop for each color. You also need to place each color in your gradient 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 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 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 give the offsets 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 RadialGradientBrush.GradientStops collection. Figure 12-17 illustrates how a radial gradient is filled.

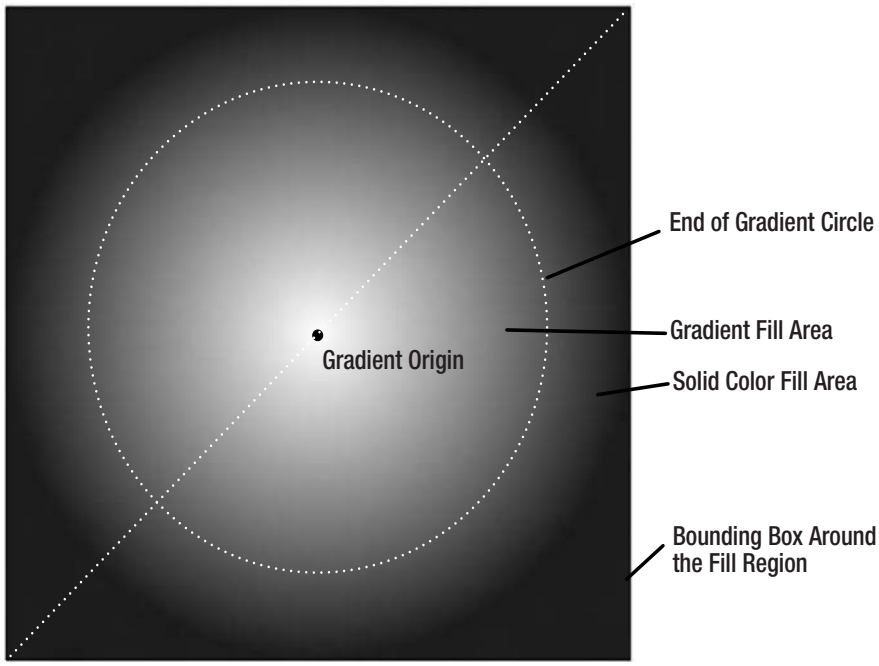

Figure 12-17. *How a radial gradient is filled*

You set the edge of the inner gradient circle 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.

The 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).

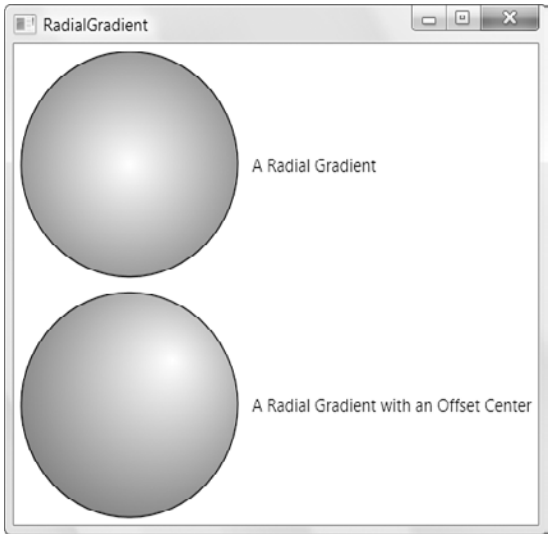

Figure 12-18. *Radial gradients*

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 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 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% 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.

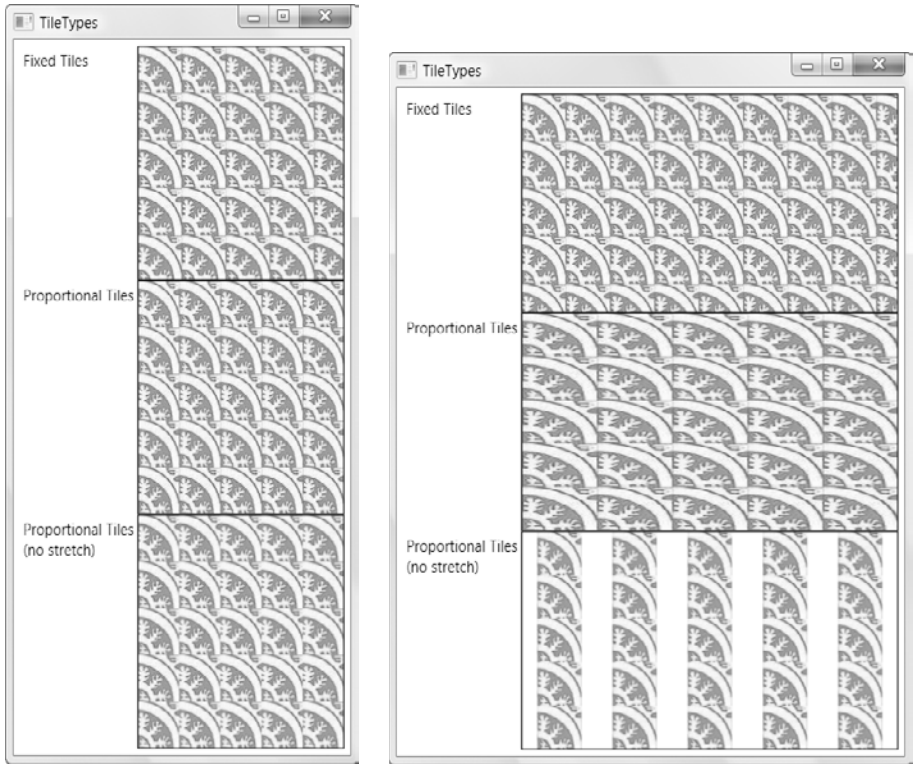

Figure 12-20. *Different ways to tile a rectangle*

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 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 in 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 in 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 different tiling options.

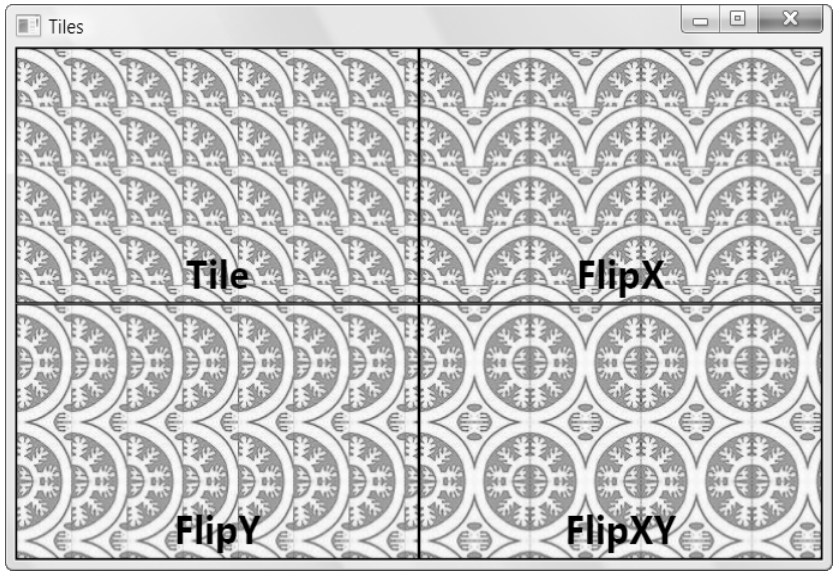

Figure 12-21. *Flipping tiles*

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 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 the "Bitmap Caching" section in Chapter 15.

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

Name	Description	Important Properties
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
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 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 different 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 different places. Using a transform, you can take a shape and then move it, rotate it, resize it, and so on.

■ **Tip** In order 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 a number of 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.

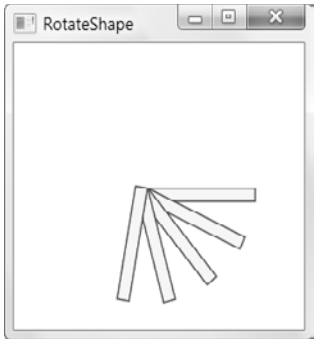

Figure 12-23. *Rotating a rectangle four times*

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.

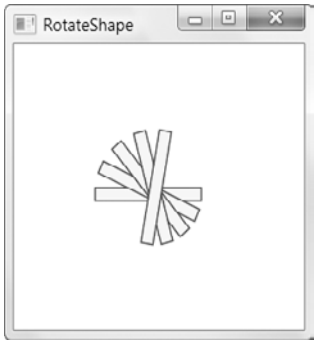

Figure 12-24. *Rotating a rectangle around its middle*

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 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 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

Unlike many older user interface technologies (for example, Windows Forms), 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% visible (10% 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.

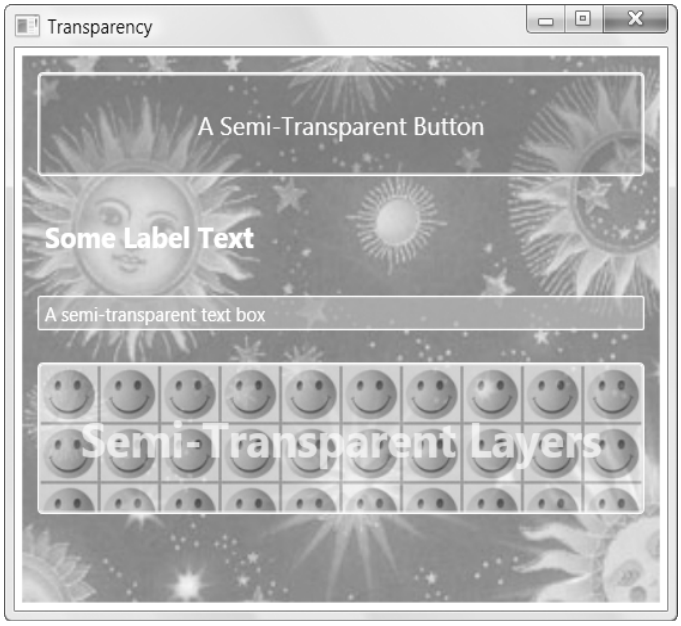

Figure 12-26. *A window with several semitransparent layers*

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.

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.