Special Effects

WPF provides built-in functionality that you can use to create graphically rich applications. Bitmap effects, transformations, a wide variety of brushes, transparency, and animation provide a vast toolset for creating unique special effects. Furthermore, you can combine any or all of these to create advanced special effects to enhance the look and feel of your applications.

In WPF, a brush is more than just a means of applying color to pixels. WPF provides a set of brushes that allow you to paint with color, gradient, image, drawing, and even visual objects as output. Painting with gradients provides you with a way to present glass effects or the illusion of depth and a third dimension. Painting with an image provides a means to stretch, tile, or fill an area with a specified bitmap. Most interesting in WPF is the ability to paint an area with any visual object defined in the application. The VisualBrush allows you to fill an area with a visual from another part of the applications visual tree. You can use this to create grand illusions of reflection or magnification in your UI. WPF provides a set of bitmap effects and transformations that you can apply to your elements. Specifically, the bitmap effects offer you a means to apply filters to your elements. Using bitmap effects you can, for example, apply drop shadows and blurring to graphical elements. Transformations, on the other hand, provide the opportunity to adjust the coordinates of an element. Using transformations, you can scale, skew, move, and rotate elements.

Using brushes, bitmap effects, and transformations in conjunction with animation, you will find yourself truly equipped with a powerful arsenal for creating visually stunning applications. For instance, if desired you could rotate an element that has a drop shadow applied by first using a transform in response to an event trigger fired by a button and then by using animation to control the speed of the rotation. Furthermore, you could accomplish all of this directly in XAML, if desired. This is powerful stuff!

In this chapter, you explore each of the concepts mentioned thus far, and a few others as well. Specifically, you will learn about the following:

- Brushes
- Bitmap effects
Chapter 6: Special Effects

- Transformations
- Opacity masks
- Combining effects

As you move through the chapter, your ability to create special effects will increase as you explore much of what is available to you in WPF. Finally, with all of the basics under your belt, you’ll combine these features to create some advanced special effects in WPF.

**Brushes**

In WPF, brushes are responsible for painting each visual element. If an element is visible on a screen, it is because a brush painted it. Brushes are responsible for painting both the Fill and Stroke of a visual element. Control properties, such as Background, Foreground, and Border, all use a brush as a mechanism to paint. Brushes are capable of painting solid colors, gradients, images, drawings, and even other visual objects.

WPF provides a number of brush classes you can use to paint pixels, all of which ultimately derive from the Brush base class. The brush classes reside in the System.Windows.Media namespace. The WPF framework provides six types of brushes you can use: SolidColorBrush, LinearGradientBrush, RadialGradientBrush, DrawingBrush, ImageBrush, and VisualBrush. The table that follows illustrates these brushes and the functionality provided by each.

<table>
<thead>
<tr>
<th>Brush</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SolidColorBrush</td>
<td>Paints a specific area of the screen with a solid color.</td>
</tr>
<tr>
<td>LinearGradientBrush</td>
<td>Paints a specific area of the screen with a linear gradient.</td>
</tr>
<tr>
<td>RadialGradientBrush</td>
<td>Paints a specific area of the screen with a radial gradient.</td>
</tr>
<tr>
<td>ImageBrush</td>
<td>Paints a specific area of the screen with an image.</td>
</tr>
<tr>
<td>DrawingBrush</td>
<td>Paints a specific area of the screen with a custom drawing.</td>
</tr>
<tr>
<td>VisualBrush</td>
<td>Paints a specific area of the screen with an object that derives from Visual.</td>
</tr>
</tbody>
</table>

As the preceding table illustrates, a brush can indeed provide output other than simply solid colors. Utilizing the various brushes available, you can create some interesting effects such as gradient and lighting effects, tiled backgrounds, thumbnail views, and visual reflection, among others. Brushes, along with other topics covered in this chapter, make up the primary toolset you can use to create a great UI.

**SolidColorBrush**

The most common brush, and the simplest to use, is the SolidColorBrush. A SolidColorBrush simply paints a specific area of the screen with a solid color. But don’t confuse a brush with a color: In WPF they are not the same. A brush is an object that tells the system to paint specific pixels with a specified
output defined by the brush. A SolidColorBrush paints a color to a specific area of the screen, such as Red, DarkRed, or even PapayaWhip. The output for a SolidColorBrush is a Color whereas the output for an ImageBrush is an Image.

A SolidColorBrush can be defined by simply providing a value for its Color property. In WPF, you can define color in a number of ways, such as declaring RGB or ARGB values, using hexadecimal notation, or by specifying a predefined color. In addition, you can supply an alpha value for the Color you define, providing transparency. (That explains the A in ARGB.) The Brush class also exposes an Opacity property you can use to define an alpha value for a brush.

Additionally, WPF also provides some handy classes for working with brushes and colors. The Brushes class, for example, exposes a set of predefined brushes based on solid colors. This provides a syntactical shortcut you can use for creating common solid color brushes. Similarly, the Colors class exposes a set of predefined colors.

Here's a quick example in which you can define a SolidColorBrush using the different methods just described:

```csharp
using System.Windows;
using System.Windows.Documents;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows.Shapes;

namespace WPFBrushes
{
    public partial class SolidColorBrushInCode : System.Windows.Window
    {
        public SolidColorBrushInCode()
        {
            InitializeComponent();
            this.Width = 600;
            this.Title = "SolidColorBrush Definition";
            StackPanel sp = new StackPanel();
            sp.Margin = new Thickness(4.0);
            sp.HorizontalAlignment = HorizontalAlignment.Left;
            sp.Orientation = Orientation.Vertical;
            TextBlock tb1 = new TextBlock(new Run("Predefined Brush \[ .Fill = Brushes.Red; \]"));
            Rectangle rect1 = new Rectangle();
            rect1.HorizontalAlignment = HorizontalAlignment.Left;
            rect1.Width = 60;
            rect1.Height = 20;
            rect1.Fill = Brushes.Red;
            TextBlock tb2 = new TextBlock(new Run("Brush from Predefined Color \[ .Fill = new SolidColorBrush(Colors.Green); \]"));
```
In the preceding code, you create five rectangles and define a SolidColorBrush for each of them:

- In rect1, you use a predefined SolidColorBrush from the Brushes class to define a red brush.
- For rect2, you create a new instance of a SolidColorBrush and pass a predefined color from the Colors class in the constructor.
For rect3, you use the method ColorFromRgb of the Color class to create a color from RGB values you’ve supplied, which you pass to the constructor of your SolidColorBrush.

In rect4, you’ll do the same thing, only in this case you’ll supply an alpha channel value to set the color transparency to 100. The range for alpha is 0–255, with zero being fully transparent (no color value).

For rect5, you create a Color from a hexadecimal string representation using a TypeConverter class provided by WPF, the ColorConverter.

You typically wouldn’t use hexadecimal notation and a TypeConverter in procedural code as it requires unnecessary overhead. The TypeConverter is really provided for using hexadecimal notation in XAML.

Figure 6-1 illustrates the results of running the sample application.

You can just as easily define and apply brushes using XAML. The following example is the XAML equivalent of the last example and will yield the same results illustrated in Figure 6-1.

```xml
<Window x:Class="WPFBrushes.SolidColorBrushInXAML"
     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     Title="SolidColorBrush" Height="300" Width="600">
  <StackPanel Margin="4" Orientation="Vertical" HorizontalAlignment="Left">
    <TextBlock Text="Predefined Brush [ .Fill = Brushes.Red; ]"/>
    <Rectangle HorizontalAlignment="Left" Width="60" Height="20" Fill="Red"/>
    <TextBlock Text="Brush from Predefined Color [ .Fill = new SolidColorBrush(Colors.Green); ]"/>
    <Rectangle Fill>
      <SolidColorBrush Color="Green"/>
    </Rectangle>
    <TextBlock Text="Brush from RGB Color [ .Fill = new SolidColorBrush(Color.FromArgb(0, 0, 255)); ]"/>
    <Rectangle Fill>
      <SolidColorBrush Color="Green"/>
    </Rectangle>
    <TextBlock Text="Brush from Hex Color [ .Fill = new SolidColorBrush(((Color)ColorConverter.ConvertFromString("#FFFFFFD5"))); ]"/>
  </StackPanel>
</Window>
```
Before you move on, take a quick look at defining a color’s alpha channel. The alpha channel specifies the level of transparency for a color. The values are 0–255, with zero being fully transparent. The following example creates a blue rectangle with white grid lines, and with three overlapping red rectangles. Each of the red rectangles sets a different value for transparency.
The example code defines a large blue rectangle with a grid pattern you’ll use as a background so you can see the opacity of each brush color. Next, place three smaller red rectangles over the grid pattern and change the alpha channel value for each color specified. Each of the red rectangles sets a different value for alpha as follows: 0, 125, and 255. Figure 6-2 illustrates the results.

When you run the example, you can see that the first rectangle shows none of the red color value. Its alpha value is zero, so it is fully transparent. The second rectangle has an alpha value of 125, and therefore is semi-transparent. The last rectangle has an alpha value of 255, so there is no transparency.
Chapter 6: Special Effects

![Figure 6-2](image-url)

**GradientBrush**

Now that you understand the roles of a brush and colors in WPF, and the `SolidColorBrush` in particular, you can take a look at gradient brushes. Gradient brushes paint an area with multiple colors. The colors blend together at specific points you specify in your brush definition. The points line up along an axis, referred to as the gradient-axis, the direction of which you also define. You can use gradients to achieve visually appealing effects, such as metal, glass, water, and shadows, or even to present the illusion of depth or a third dimension. WPF provides two flavors of gradient brushes that not surprisingly derive from the `GradientBrush` base class: `LinearGradientBrush` and `RadialGradientBrush`.

As its name implies, the `LinearGradientBrush` follows a linear axis. You can define the direction of the axis to achieve vertical, horizontal, or diagonal gradients. The gradient-axis is defined by two points, a `StartPoint` and an `EndPoint`. These points map to a 1x1 matrix. So a `StartPoint` of (0,0) and an `EndPoint` of (1,1) will produce a diagonal gradient, while a `StartPoint` of (0,0.5) and an `EndPoint` of (1,0.5) will produce a horizontal gradient. Along the axis you define a series of `GradientStop` objects, which are points on the axis where you want colors to blend and transition to other colors. You can define as many `GradientStop` objects as you like. A `GradientStop` has two properties of interest: `Color` and `Offset`. The `Offset` property defines a distance, ranging from 0 to 1, from the start point of the axis, from which the color specified in the `Color` property should begin.

The `RadialGradientBrush` is defined in a similar fashion; however, this brush blends colors in a radial pattern. A radial gradient is defined as a circle. The axis of the `RadialGradientBrush` starts from a point of origin you can define called the `GradientOrigin` and runs to the outer edge of the circle, the axis end point. You define `GradientStop` objects along the gradient-axis, just as you do for a `LinearGradientBrush`. You can define the circle of the gradient by specifying the `Center`, `RadiusX`, and `RadiusY` properties. The `Center` property specifies the center of the circle. The `RadiusX` and `RadiusY` properties specify the distance from center to the outer edge of the circle on the X and Y axis. This means you don’t have to define a perfect circle; it could be an ellipse.
Now, create an example using the `LinearGradientBrush` and `RadialGradientBrush`.

```xml
<Window x:Class="WPFBrushes.GradientBrushInXAML"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Gradient Brushes" Height="300" Width="300">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>

        <Rectangle Grid.Row="0" Grid.Column="0" Width="100" Height="100"
            StrokeThickness="4" Margin="4">
            <Rectangle.Fill>
                <LinearGradientBrush>
                    <GradientStop Color="Gray" Offset=".3"/>
                    <GradientStop Color="Black" Offset=".4"/>
                    <GradientStop Color="Gray" Offset=".8"/>
                </LinearGradientBrush>
            </Rectangle.Fill>
            <Rectangle.Stroke>
                <SolidColorBrush Color="Blue"/>
            </Rectangle.Stroke>
        </Rectangle>

        <Rectangle Grid.Row="0" Grid.Column="1" Width="100" Height="100"
            StrokeThickness="8" Margin="4">
            <Rectangle.Fill>
                <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                    <GradientStop Color="Gray" Offset=".3"/>
                    <GradientStop Color="Black" Offset=".4"/>
                    <GradientStop Color="Gray" Offset=".8"/>
                </LinearGradientBrush>
            </Rectangle.Fill>
            <Rectangle.Stroke>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Color="Red" Offset=".3"/>
                    <GradientStop Color="Blue" Offset=".4"/>
                </LinearGradientBrush>
            </Rectangle.Stroke>
        </Rectangle>

        <Rectangle Grid.Row="1" Grid.Column="0" Width="100" Height="100"
            StrokeThickness="4" Margin="4">
            <Rectangle.Fill>
                <RadialGradientBrush GradientOrigin="0.5,0.5" Center="0.5,0.5"
                    RadiusX="0.5" RadiusY="0.5">
                    <GradientStop Color="Black" Offset="0"/>
                </RadialGradientBrush>
            </Rectangle.Fill>
        </Rectangle>
    </Grid>
</Window>
```
The first two rectangles are filled by LinearGradientBrush objects, the first along a diagonal gradient-axis, and the second along a horizontal gradient-axis. The second rectangle also uses a LinearGradientBrush to fill its Stroke property with red and blue. The last two rectangles are filled by a RadialGradientBrush. The second rectangle uses a RadialGradientBrush to fill its Stroke property, creating an interesting pattern. Figure 6-3 illustrates the results of this example.

The GradientBrush examples presented thus far are intended to illustrate how LinearGradientBrush and RadialGradientBrush objects can be defined. The best way to get to know gradient brushes and the various properties covered in the chapter so far is to dive in and start using them. In the case of a LinearGradientBrush, change the StartPoint and EndPoint properties to get a feel for the gradient axis. Add variable numbers of GradientStops and play with the Color and Offset properties of each to see the results and how the colors transition. For a RadialGradientBrush, change the size of the circle and point of origin to get a feel for the many options available to you.
ImageBrush

The ImageBrush allows you to specify an image to be painted to an output area. Quite simply, the ImageBrush paints a specified area with its output, a bitmap. ImageBrush derives from TileBrush, so a pattern can be specified based on the image. Using the ImageBrush, you can specify an image to be used as the background of a button as follows:

```xml
<Window x:Class="WPFBrushes.ImageBrushInXAML"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ImageBrush" Height="425" Width="300">

    <StackPanel Margin="4">
        <StackPanel.Resources>
            <Style TargetType="Button">
                <Setter Property="Foreground" Value="White"/>
                <Setter Property="FontWeight" Value="DemiBold"/>
                <Setter Property="FontSize" Value="18"/>
                <Setter Property="Width" Value="250"/>
                <Setter Property="Height" Value="65"/>
                <Setter Property="Margin" Value="4"/>
            </Style>
        </StackPanel.Resources>

        <Button Content="FILL">
            <Button.Background>
                <ImageBrush ImageSource="Images/Flower.jpg" Stretch="Fill"/>
            </Button.Background>
        </Button>
    </StackPanel>
</Window>
```
Chapter 6: Special Effects

The preceding code defined five buttons. For each button, an ImageBrush was defined in order to be used for setting the button’s Background property. Also, the ImageSource property of each Brush was set to point at an image resource defined in your Visual Studio project. (Be sure to set the Build Action of the image to Resource.) Additionally, the Stretch property of each ImageBrush was set differently, in order to illustrate each of the values of the stretch enumeration. Figure 6-4 illustrates the results of running the application.

For the first button, setting the Stretch property of your ImageBrush to Fill will stretch the image to fill the button’s background. By default, the ImageBrush will stretch an image to fill its output area. In the second button, add an Opacity value of .25 to your ImageBrush. (All brushes expose an Opacity property.) In the third button, set the Stretch property to Uniform. Uniform will maintain the image ratio, but make sure to resize the image to fit into the buttons content area. In the fourth button, set the Stretch property to None, which has the effect of preserving the original image size and centering it in the output content area. In the last button, set the Stretch property to UniformToFill, which resizes the image to best-fit while preserving the original image ratio. In this case, the best-fit is still larger than the content area of the button, so the image is essentially clipped.
**DrawingBrush**

A `DrawingBrush` paints a specified area with a `Drawing` object. The `Drawing` class represents a two-dimensional drawing and is the base class for other drawing objects, including `GeometryDrawing`, `GlyphRunDrawing`, `ImageDrawing`, and `VideoDrawing`. The `GeometryDrawing` class allows you to define and render shapes with a specified Fill and Stroke, and the `GlyphRunDrawing` supports text operations. The latter two are self-explanatory. So, because the output of a `DrawingBrush` is a `Drawing` object, and because these various objects derive from `Drawing`, this means that a drawing brush can paint shapes, text, images, and video.

Additionally, one other class derives from `Drawing`, the `DrawingGroup` class. The `DrawingGroup` class allows you to group multiple `Drawing` objects together in order to create a composite `Drawing` object.

The following example applies a `DrawingBrush` to the background of a button. You will now create a `Drawing` using a `GeometryDrawing` object.

```xml
<Window x:Class="WPFBrushes.DrawingBrushInXAML"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DrawingBrush" Height="300" Width="300">
```
Chapter 6: Special Effects

The previous example uses a DrawingBrush to define the background of a button. In this example, you created your drawing as a resource, but you don’t have to — this is just for convenience should you want to reuse this drawing elsewhere. Here, simply use a couple of EllipseGeometry classes to define a couple of shapes in a GeometryDrawing. The GeometryDrawing object is the Drawing that will be used by the DrawingBrush you define.

Figure 6-5 illustrates the results of running the example.

Figure 6-5
The DrawingBrush is quite flexible and very powerful. It allows you to paint with many low-level objects from the WPF framework. These objects are not derivatives of UIElement, so they do not participate in the layout system, which is why they are so lightweight.

**VisualBrush**

The VisualBrush paints an area with any object that derives from Visual. Any visual element you specify, along with the children of that element, will be output by the VisualBrush. You can apply transformations to the VisualBrush in order to add additional effects if you choose. Note that the VisualBrush is “copying” the visual tree of an element, not the transformations, animations, events, and so on, which are associated with that element.

```xml
<Window x:Class="WPFBrushes.VisualBrushInXAML"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="VisualBrush" Height="500" Width="300">
  <Window.Resources>
    <DrawingBrush x:Key="MyCustomDrawing">
      <DrawingBrush.Drawing>
        <GeometryDrawing Brush="Red">
          <GeometryDrawing.Geometry>
            <GeometryGroup>
              <EllipseGeometry RadiusX="22" RadiusY="25" Center="25,50" />
              <EllipseGeometry RadiusX="22" RadiusY="55" Center="50,50" />
            </GeometryGroup>
          </GeometryDrawing.Geometry>
          <GeometryDrawing.Pen>
            <Pen Thickness="1.5" Brush="LightBlue" />
          </GeometryDrawing.Pen>
        </GeometryDrawing>
      </DrawingBrush.Drawing>
    </DrawingBrush>
  </Window.Resources>
  <Grid>
    <StackPanel Margin="4,4,4,4">
      <TextBlock Margin="4,4,4,4">Source Visual:</TextBlock>
      <Button Content="DrawingBrush" Height="125" Width="275" Name="MyVisual"
          Background="{(StaticResource MyCustomDrawing)}" />
      <TextBlock Margin="4,4,4,4">VisualBrush:</TextBlock>
      <Button Foreground="Blue" Height="125" Width="275">
        <Button.Background>
          <VisualBrush Visual="{(Binding ElementName=MyVisual)}" />
        </Button.Background>
      </Button>
      <TextBlock Margin="4,4,4,4">Tiled VisualBrush:</TextBlock>
      <Button Foreground="Blue" Height="125" Width="275">
        <Button.Background>
          <VisualBrush Visual="{(Binding ElementName=MyVisual)}" />
        </Button.Background>
      </Button>
    </StackPanel>
  </Grid>
</Window>
```
The previous example extends the DrawingBrush example you created previously. What you are doing here is adding two more buttons below the original button. Each of these new buttons uses a VisualBrush to paint its background. In this example, the second button simply uses the VisualBrush to essentially “copy” the first button and use it as its background. The third button does the same, only this time scales the “copy” to 25 percent and then tiles its background. The scaling is achieved using a ScaleTransform, which you’ll get to know later in this chapter. Figure 6-6 illustrates the results of running the example.
The background for the bottom two buttons is created using a VisualBrush. By default, when you mouse over a button, the button is highlighted yellow. Notice that if you mouse over the first button, the background of the other buttons becomes highlighted as well. In fact, in the third button, each of the tiles of the VisualBrush are highlighted. This is because the VisualBrush “copies” the entire visual tree of its source object. Part of the button’s visual tree is the border and the event trigger associated with it.

The VisualBrush is extremely powerful and opens the door for creating lots of special effects. For example, there are already a lot of examples on the Internet of using a VisualBrush to achieve a reflection effect. You will see how to use a VisualBrush later in this chapter in the Bouncing Ball example.

**Bitmap Effects**

WPF bitmap effects allow you to add artistic filters to your visual elements such as blurring, drop shadows, and glows. If you’re a designer, you can think of these as blending options. These effects can be applied to any Visual or UIElement in the WPF Framework. UIElement exposes a BitmapEffect property, whereas Visual exposes a VisualBitmapEffect property. Both of these properties accept a BitmapEffect object as a value.

Bitmap effects are classes in the WPF framework that derive from BitmapEffect base class. These are found in the System.Windows.Media.Effects namespace. Currently, the WPF Framework provides six flavors of bitmap effects, as illustrated in the table that follows.

<table>
<thead>
<tr>
<th>Bitmap Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BevelBitmapEffect</td>
<td>Creates a raised surface effect on an element.</td>
</tr>
<tr>
<td>BlurBitmapEffect</td>
<td>Creates a blurring effect on an element.</td>
</tr>
<tr>
<td>DropShadowBitmapEffect</td>
<td>Creates a shadow behind an element.</td>
</tr>
<tr>
<td>EmbossedBitmapEffect</td>
<td>Creates an illusion of depth on an element.</td>
</tr>
<tr>
<td>OuterGlowBitmapEffect</td>
<td>Creates a glowing effect around the perimeter of an element.</td>
</tr>
<tr>
<td>BitmapEffectGroup</td>
<td>Allows multiple effects to be applied to a BitmapEffect property.</td>
</tr>
</tbody>
</table>

A bitmap effect accepts a visual element as input, applies the effect, and produces a graphical result. Bitmap effects are software rendered and, therefore, can be expensive for large visual objects. Some effects work better on vector content and others work better with images, but they can all be applied to either. What works best in a given situation is completely up to you. Also, you can create custom bitmap effects, but an explanation of that is beyond the scope of this chapter.

The best way to get familiar with these effects is to get your hands dirty and write some code. Rather than create a separate code example for each of the bitmap effects, this chapter presents a single example to demonstrate all of them. In this code example, you will create a number of elements, bitmap effects, sliders, and text boxes, which will illustrate the various bitmap effects provided by WPF.

```xml
<Window x:Class="BitmapEffects.Window4"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```
Chapter 6: Special Effects

<Ellipse Height="100" Width="100" Fill="Blue" Stroke="Black">
    <Ellipse.BitmapEffect>
        <BlurBitmapEffect
            Radius="{(Binding Path=Value, ElementName=sliderBlur)}"
        />
    </Ellipse.BitmapEffect>
</Ellipse>

<TextBlock>Blur</TextBlock>

<StackPanel Grid.Column="0" Grid.Row="3"
            Orientation="Horizontal" HorizontalAlignment="Center">
    <Slider Minimum="0" Maximum="10" Name="sliderBlur" Value="14"/>
    <TextBox Text="{(Binding ElementName=sliderBlur, Path=Value)}"/>
</StackPanel>

<Grid Grid.Column="1" Grid.Row="0"
      Height="50" Width="200" HorizontalAlignment="Center">
    <Rectangle Height="50" Width="200" Fill="White" Stroke="Black">
        <Rectangle.BitmapEffect>
            <DropShadowBitmapEffect
                ShadowDepth="{(Binding Path=Value, ElementName=sliderDrop)}"
            />
        </Rectangle.BitmapEffect>
    </Rectangle>
    <TextBlock>Drop Shadow</TextBlock>
</Grid>

<StackPanel Grid.Column="1" Grid.Row="1"
            Orientation="Horizontal" HorizontalAlignment="Center">
    <Slider Minimum="0" Maximum="10" Name="sliderDrop" Value="14"/>
    <TextBox Text="{(Binding ElementName=sliderDrop, Path=Value)}"/>
</StackPanel>

<Grid Grid.Column="1" Grid.Row="2" HorizontalAlignment="Center">
    <Canvas>
        <Ellipse Height="100" Width="100" Fill="Blue" Stroke="Black">
            <Canvas.BitmapEffect>
                <OuterGlowBitmapEffect
                    GlowSize="{(Binding Path=Value, ElementName=sliderGlow)}"
                />
            </Canvas.BitmapEffect>
        </Ellipse>
        <TextBlock>Outer Glow</TextBlock>
    </Canvas>
    <Button Grid.Column="1" Grid.Row="2" Height="50" Width="200">
        Outer Glow
    </Button>
</Grid>

<StackPanel Grid.Column="1" Grid.Row="3"
            Orientation="Horizontal" HorizontalAlignment="Center">
    <Slider Minimum="0" Maximum="20" Name="sliderGlow" Value="14"/>
    <TextBox Text="{(Binding ElementName=sliderGlow, Path=Value)}"/>
</StackPanel>
In order to illustrate the various bitmap effects, you need some elements to apply them to. Create a couple of rectangles, a couple of ellipses, a button, and an image reference. Then, place all of these elements in a Grid for presentation alignment purposes. For each element, create a BitmapEffect object and then supply the specific effect to be used. For each of these effects, create a Slider object and bind the appropriate effect property to the slider’s Value property. For example, you can bind your drop shadows ShadowDepth property to your drop shadow slider. Finally, create a TextBox next to each slider and bind the TextBox.Text property to the slider’s Value property to display the value of the slider.

For the Emboss effect, you can use any image you like. Simply add the image to your Visual Studio project, right-click, select Properties, and then change the Build property to Resource.

Figure 6-7 illustrates the results of running the example.

Figure 6-7 illustrates the use of each bitmap effect provided by WPF. The example application suggests that these effects can be applied to any element, and that is, in fact, true. What this example does not show is that you can indeed combine bitmap effects and apply them as one to any visual element. To do this, you can add your effects to a BitmapEffectGroup class as follows:

```xml
<Rectangle.BitmapEffect>
    <BitmapEffectGroup>
        <BevelBitmapEffect
            BevelWidth="{Binding ElementName=sliderBevel, Path=Value}" />
        <OuterGlowBitmapEffect GlowColor="Aqua" GlowSize="3"/>
    </BitmapEffectGroup>
</Rectangle.BitmapEffect>
```
In the previous code, you simply created a BitmapEffectGroup object and then added two bitmap effects to it: one BevelBitmapEffect and one OuterGlowBitmapEffect. The ability to combine bitmap effects in WPF is very effective in creating an interesting UI.

**Transformations**

WPF transformations can be used to apply effects such as rotation, scaling, and skewing to visual elements. For example, using transformations you can rotate a button, scale text, or skew a rectangle to give the impression of depth. Taking this one step further, you can add animation to transformation to achieve even more effects. For example, you can magnify visual elements, move controls around in your UI (referred to as translation), and do just about anything else you can think of.
Transformation classes live in the System.Windows.Media namespace. WPF provides several transform classes for convenience that let you easily scale, translate, rotate, and skew objects. These classes are: ScaleTransform, TranslateTransform, RotateTransform, and SkewTransform and all derive from the base class Transform. The following table illustrates these transform classes available in WPF and the functionality provided by each.

<table>
<thead>
<tr>
<th>Transform</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RotateTransform</td>
<td>Rotates an element based on a center point you can define.</td>
</tr>
<tr>
<td>ScaleTransform</td>
<td>Resizes an element based on a center point you can define.</td>
</tr>
<tr>
<td>SkewTransform</td>
<td>Shears an element in a non-uniform manner.</td>
</tr>
<tr>
<td>TranslateTransform</td>
<td>Moves an element along an X and Y axis.</td>
</tr>
<tr>
<td>MatrixTransform</td>
<td>Provides a 3×3 matrix you can use to define a custom transformation.</td>
</tr>
</tbody>
</table>

I refer to the first four classes in the preceding table as convenience classes because transformations can be created directly using a MatrixTransform class if you understand the principles of mapping points from one coordinate space to another. The MatrixTransform class exposes six properties: m11, m12, m21, m22, offsetX, and offsetY, which let you define the values of a transform matrix, a 3×3 matrix of values that define a transformation. All transforms have a matrix. While creating transformations such as scale and translate is very easy using a MatrixTransform, rotate and skew are far more complex. This is why WPF offers the convenient transform classes that it does.

The following code creates a scale transform using a MatrixTransform object:

```xml
<Grid>
  <Rectangle Width="50" Height="50" Fill="#AAFF0000">
    <Rectangle.RenderTransform>
      <MatrixTransform>
        <Matrix M11=".75" M12="0" M21="0" M22=".75" OffsetX="-20" OffsetY="-20"/>
      </MatrixTransform>
    </Rectangle.RenderTransform>
  </Rectangle>
</Grid>
```

The following code creates the same scale transform using a ScaleTransform object:

```xml
<Grid>
  <Rectangle Width="50" Height="50" Fill="#AAFF0000">
    <Rectangle.RenderTransform>
      <ScaleTransform ScaleX=".75" ScaleY=".75" CenterX="-20" CenterY="-20"/>
    </Rectangle.RenderTransform>
  </Rectangle>
</Grid>
```
There are two types of transforms you can apply to elements in WPF: RenderTransform, defined by UIElement, and LayoutTransform, defined by FrameworkElement. A RenderTransform is applied to an element as the element is rendered. What this means is that the transform is applied after layout. Because of this, you may transform an element and find it overlaps other elements, or moves outside of the bounding box of its parent element. This is not necessarily a bad thing; in fact, it might be desirable in a specific situation. Alternatively, a LayoutTransform applies the transform to an element prior to layout. The WPF layout engine will take into account the transform during the Measure and Arrange phase of layout and will size and arrange the elements appropriately.

The following example creates three rectangles placed side by side. You can then place a RotateTransform on the rectangle in the center to rotate it 45 degrees. The rectangle will rotate 45 degrees as expected, but it overlaps the left-most rectangle. Again, this may be the desired effect.

```xml
<StackPanel Orientation="Horizontal">
  <Rectangle Width="50" Height="50" Fill="Blue"/>
  <Rectangle Width="50" Height="50" Fill="Red">
    <Rectangle.RenderTransform>
      <RotateTransform Angle="45"/>
    </Rectangle.RenderTransform>
  </Rectangle>
  <Rectangle Width="50" Height="50" Fill="Green"/>
</StackPanel>
```

In the following example, simply swap out the RenderTransform with a LayoutTransform. The result is that the layout engine takes into account the space required for the rotated rectangle, and sizes the controls respectfully. This results in no overlapping of elements.

```xml
<StackPanel Orientation="Horizontal">
  <Rectangle Width="50" Height="50" Fill="Blue"/>
  <Rectangle Width="50" Height="50" Fill="Red">
    <Rectangle.LayoutTransform>
      <RotateTransform Angle="45"/>
    </Rectangle.LayoutTransform>
  </Rectangle>
  <Rectangle Width="50" Height="50" Fill="Green"/>
</StackPanel>
```

Additionally, you can apply any number of transformations that you wish to a single element using a TransformGroup class. If you want to scale an item to twice its size and then rotate it 45 degrees, you can do this by adding both transforms to a TransformGroup object. You would then assign the TransformGroup object to the RenderTransform or LayoutTransform property of the element, and both would be applied.

```xml
<Rectangle Width="200" Height="50" Fill="Red">
  <Rectangle.RenderTransform>
    <TransformGroup>
      <ScaleTransform ScaleX="2" ScaleY="2"/>
      <RotateTransform Angle="45"/>
    </TransformGroup>
  </Rectangle.RenderTransform>
</Rectangle>
```

The rest of this section provides a thorough overview of the first four transform classes provided by WPF.
Chapter 6: Special Effects

TranslateTransform

The TranslateTransform is one of the simplest transformations you can perform. Similar in simplicity is the ScaleTransform, which will be covered next. The TranslateTransform simply moves (translates) an element along a two-dimensional X and Y axis. A positive X value moves the element to the right; a negative value moves it to the left. Similarly, a positive Y value moves the element down, and a negative value moves it up. This is really an offset from the element’s original position. TranslateTransform is really just a convenience wrapper for setting the OffsetX and OffsetY properties of the transform Matrix structure.

Most containers in WPF provide layout mechanisms that control the position of an element, such as Margin, HorizontalAlignment, and VerticalAlignment. Sometimes, however, you still cannot set a child element’s position as you would like. One solution is to apply a TranslateTransform to the element and then position the element to your liking. This is but one example of the use of a TranslateTransform. In an example later in this chapter, you’ll use a TranslateTransform with animation in order to make an element bounce up and down . . . so stay tuned.

In the following example, you create a ListBox and two sliders that you can use to move the ListBox from its original position.

```xml
<Window x:Class="WPFTransformations.Translate"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Translate Transform" Height="300" Width="300"
>
  <Window.Resources>
    <Style TargetType="TextBox">
      <Setter Property="FontFamily" Value="Verdana"/>
      <Setter Property="Margin" Value="4"/>
      <Setter Property="FontWeight" Value="DemiBold"/>
      <Setter Property="Width" Value="40"/>
    </Style>
    <Style TargetType="TextBlock">
      <Setter Property="FontFamily" Value="Verdana"/>
      <Setter Property="Margin" Value="4"/>
      <Setter Property="FontWeight" Value="DemiBold"/>
    </Style>
    <Style TargetType="Slider">
      <Setter Property="Margin" Value="4"/>
      <Setter Property="Width" Value="100"/>
      <Setter Property="Maximum" Value="100"/>
      <Setter Property="Minimum" Value="-100"/>
      <Setter Property="Value" Value="0"/>
      <Setter Property="TickFrequency" Value="2"/>
      <Setter Property="IsSnapToTickEnabled" Value="True"/>
    </Style>
  </Window.Resources>
  <Grid>
    <StackPanel Margin="8">
      <ListBox Height="100" Width="200" BorderBrush="Blue" BorderThickness="2">
        <ListBoxItem Content="Item 1" Background="Beige" Height="22"/>
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```
In the preceding code, you also created a Rectangle the same size as the ListBox. The rectangle will serve as a point of reference for you while you move the ListBox along the X and Y axis. You’ll also add a RenderTransform to the ListBox and specify a TranslateTransform transformation. The TranslateTransform exposes two properties you’re concerned with, X and Y. Both of these accept a double, and can be either positive or negative in value. In this example, you bind the X and Y properties of your transform to the Value property of the X and Y sliders. Figure 6-8 illustrates the results of running your example.
Moving the X and Y sliders moves the ListBox control accordingly. Because this transformation is applied after rendering, you’ll find that you can effectively move an element right out of its container. For example, change the Background of the topmost StackPanel to gray and rerun the application. You’ll see that you can move outside of the StackPanel bounds. This behavior exists because you are applying a RenderTransform.

Now, if you change the RenderTransform to a LayoutTransform, you’ll see a completely different behavior. When you move the sliders, nothing happens. This is by design because a LayoutTransform will ignore a TranslateTransform.

**ScaleTransform**

When you want to resize an element, you can use a ScaleTransform. Applying a scale transformation is also very simple. You can scale an object using a ScaleTransform by specifying the value for the ScaleX and ScaleY properties. As their names suggest, these properties represent factors for scaling along the X and Y axis. If you were to specify 2 for ScaleX and 2 for ScaleY, you would be scaling your element by 200 percent. Similarly, if you were to specify .5 for both properties, you would be resizing your element by 50 percent.

In addition to defining a scale factor for your element, you may also specify a point from which to scale. You can specify these values by setting the CenterX and CenterY properties of the ScaleTransform object. By default, these both have a value of zero. This means your element will scale from the top-left corner of the object (for example, the top-left corner will remain intact). If you wanted to scale from the center of the element, you would specify CenterX = Width/2 and CenterY = Height/2, where Width and Height are values of your element.

Similar to the prior example, you’ll now create a ListBox and some Slider objects you can use to scale the ListBox and define the center point from which you will scale, using the following code:
Chapter 6: Special Effects

Excerpted from Professional WPF Programming: .NET Development with the Windows Presentation Foundation
Wrox Press, www.wrox.com
This code is very similar to the last example. The primary difference is that this time you are using a ScaleTransform rather than a TranslateTransform. In addition you are adding a couple more sliders. This time the sliders represent the ScaleX, ScaleY, CenterX, and CenterY properties of the ScaleTransform. Figure 6-9 illustrates the results of running your example.

Running your program, you can see that by changing the ScaleX slider, the ListBox gets larger and smaller along the X axis. Similarly, changing the ScaleY slider stretches the ListBox along the Y axis. Your top-left corner, however, will not change, unless you move the CenterX and CenterY sliders away from zero. Once you change these, you’ll see you lose the default top-left corner position. Figure 6-9 illustrates this point. The original ListBox position is represented by the red rectangle in the center.
The term *skew* literally means to turn or place something at an angle or on its side. In a 2D world, this would be hard to do, but you can create an effect that will give the appearance of a skewed object. The SkewTransform will skew (a.k.a. shear) an element’s coordinates in a non-uniform manner. Skewing an element can create the illusion of depth or a third dimension. You can control the skewing of an element by defining four property values of the SkewTransform class: AngleX, AngleY, CenterX, and CenterY. AngleX and AngleY define the skew angle, while CenterX and CenterY the center point from which to skew. By default, all these properties are set to zero.

Next, modify your prior example to perform a skew transformation and change the sliders to manipulate the SkewTransform properties.

```xml
<Window x:Class="WPFTransformations.Transforms"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Skew Transform" Height="350" Width="300">
    <Window.Resources>
        <Style TargetType="TextBox">
            <Setter Property="FontFamily" Value="Verdana"/>
            <Setter Property="Margin" Value="4"/>
            <Setter Property="FontWeight" Value="DemiBold"/>
            <Setter Property="Width" Value="40"/>
        </Style>
    </Window.Resources>
```

Figure 6-9
Chapter 6: Special Effects

```xml
<Style TargetType="TextBlock">
    <Setter Property="FontFamily" Value="Verdana"/>
    <Setter Property="Margin" Value="4"/>
    <Setter Property="FontWeight" Value="DemiBold"/>
</Style>

<Style TargetType="Slider">
    <Setter Property="Margin" Value="4"/>
    <Setter Property="Width" Value="100"/>
    <Setter Property="Maximum" Value="100"/>
    <Setter Property="Minimum" Value="-100"/>
    <Setter Property="Value" Value="0"/>
    <Setter Property="TickFrequency" Value="2"/>
    <Setter Property="IsSnapToTickEnabled" Value="True"/>
</Style>

</Window.Resources>
<Grid>
    <StackPanel Margin="8">
        <ListBox Height="100" Width="200" BorderBrush="Blue" BorderThickness="2">
            <ListBoxItem Content="Item 1" Background="Beige" Height="22"/>
            <ListBoxItem Content="Item 2" Background="LightGray" Height="22"/>
            <ListBoxItem Content="Item 3" Background="Beige" Height="22"/>
            <ListBoxItem Content="Item 4" Background="LightGray" Height="22"/>
            <ListBox.RenderTransform>
                <SkewTransform
                    CenterX="{Binding Path=Value, ElementName=sliderSkewCX}"
                    CenterY="{Binding Path=Value, ElementName=sliderSkewCY}"
                    AngleX="{Binding Path=Value, ElementName=sliderSkewX}"
                    AngleY="{Binding Path=Value, ElementName=sliderSkewY}"/>
            </ListBox.RenderTransform>
        </ListBox>
        <TextBlock Height="65" Width="100"/>
        <Grid HorizontalAlignment="Center" Margin="2">
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition/>
                <ColumnDefinition Width="110"/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
            <TextBlock Grid.Row="0" Grid.Column="0" Text="Angle X:"/>
            <Slider Grid.Row="1" Name="sliderSkewX"/>
            <TextBox Grid.Row="0" Grid.Column="2" Text="{Binding Path=Value, ElementName=sliderSkewX}"/>
            <TextBlock Grid.Row="1" Grid.Column="0" Text="Angle Y:"/>
        </Grid>
    </StackPanel>
</Grid>
```

Excerpted from Professional WPF Programming: .NET Development with the Windows Presentation Foundation
Wrox Press, www.wrox.com
Again, this code is very similar to the other transform samples you’ve created. This time, however, you’re applying a skew transformation. Also, the previous code creates a slider for each value that affects the skew effect you place on the ListBox. Figure 6-10 illustrates the results of running the example.

![Figure 6-10](image_url)
Chapter 6: Special Effects

Running this sample and applying the various values, you can really get a sense of what “skewing” is. You can make an element appear to be on its side, tilted backward or forward giving the illusion of depth, or stretched in very odd ways. One of my favorite aspects of this example is the fact that the ListBox still works! You can select items, wire up routed events, and do any other thing you can think of with a ListBox control. That is the power of WPF.

RotateTransform

WPF also provides the RotateTransform class. As you’ve probably guessed already, the RotateTransform class allows you to rotate an element. There are three properties of interest: Angle, CenterX, and CenterY. The Angle property specifies the distance to rotate, for example, 45, 90, 180 degrees, and so on. Together, the CenterX and CenterY properties specify the center point around which the element will rotate. The default for all three properties is zero. To rotate an element on center, you would set CenterX = Width/2 and CenterY = Height/2, where Width and Height are values of the element. This assumes you are using a RenderTransform. If you rotate an element in a LayoutTransform, it will rotate on center because you cannot translate the element in this mode.

You don’t always know the height and width of your elements because the WPF layout engine may resize them during the Measure and Arrange phase of layout, which is performed at runtime. For this scenario, you can use a property called RenderTransformOrigin which is part of the UIElement class. RenderTransformOrigin is defined as a Point object. You’re not specifying a location when you set this value; rather, you are specifying a factor, between 0 and 1 for both X and Y, which you will use as the base for the transformation. RenderTransformOrigin can be applied to any transformation. So, to rotate on center, you simply supply the value of 0.5 for both X and Y of the Point structure.

```xml
<Window x:Class="WPFTransformations.Rotate"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Rotate Transform" Height="350" Width="300"
>
<Window.Resources>
<Style TargetType="TextBox">
  <Setter Property="FontFamily" Value="Verdana"/>
  <Setter Property="Margin" Value="4"/>
  <Setter Property="FontWeight" Value="DemiBold"/>
  <Setter Property="Width" Value="40"/>
</Style>
<Style TargetType="TextBlock">
  <Setter Property="FontFamily" Value="Verdana"/>
  <Setter Property="Margin" Value="4"/>
  <Setter Property="FontWeight" Value="DemiBold"/>
</Style>
<Style TargetType="Slider">
  <Setter Property="Margin" Value="4"/>
  <Setter Property="Width" Value="100"/>
  <Setter Property="Maximum" Value="100"/>
  <Setter Property="Minimum" Value="-100"/>
  <Setter Property="Value" Value="0"/>
  <Setter Property="TickFrequency" Value="2"/>
  <Setter Property="IsSnapToTickEnabled" Value="True"/>
</Style>
</Window.Resources>
```
This time you’re using a **RotateTransform**, and the sliders represent the **Angle**, **CenterX**, and **CenterY**. Moving the Angle slider will rotate the **ListBox**. Changing the **CenterX** and **CenterY** sliders will move the center point of the rotation. By this, the fourth example in this style, the rest of the code should look pretty familiar.

Figure 6-11 illustrates the results of running the example.

![Figure 6-11](image)

In Figure 6-11, you see that you’ve rotated the **ListBox** 154 degrees clockwise. In order to keep your element on the screen while rotating, you must change the **CenterX** and **CenterY** properties. If you didn’t, you wouldn’t see much of your **ListBox** at 154 degrees because, by default, you’d be rotating at the coordinates (0,0).

**Opacity Masks**

**Opacity masks** provide another way to create interesting effects in WPF. You can create the appearance of an element or image fading into the background. You can also create glass-like surface effects in order to make an element look glossy or semi-transparent. You can apply an opacity mask to any element via the **OpacityMask** property.
Typically, you create an opacity mask using a gradient, but you can use any type of Brush object as an opacity mask. The way an opacity mask works is based on the alpha value of each pixel defined in the brush. The actual color of the pixel is ignored; rather, the opacity mask simply determines the alpha value (opacity level) of each pixel. The value of the alpha is then applied to the element to which you are applying the opacity mask. If an area of the opacity mask’s brush is fully transparent, then the corresponding area of the element is made transparent.

The following example illustrates the use of an opacity mask. You’ll use a drawing brush to create a grid pattern to use as the background of a rectangle. You’ll then define an opacity mask for the rectangle, which will create a fading effect.

```xml
<Window x:Class="OpacityMasks.Window1"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="Opacity Masks" Height="300" Width="300">
  <Window.Resources>
    <DrawingBrush x:Key="Grid" Viewport="0,0,.1,.1" TileMode="Tile">
      <DrawingBrush.Drawing>
        <GeometryDrawing>
          <GeometryDrawing.Geometry>
            <GeometryGroup>
              <RectangleGeometry Rect="0, 0, 10, 10"/>
              <RectangleGeometry Rect="5, 5, 10, 10"/>
              <RectangleGeometry Rect="0, 5, 10, 10"/>
              <RectangleGeometry Rect="5, 0, 10, 10"/>
            </GeometryGroup>
          </GeometryDrawing.Geometry>
          <GeometryDrawing.Pen>
            <Pen Thickness=".5" Brush="Blue"/>
          </GeometryDrawing.Pen>
        </GeometryDrawing>
      </DrawingBrush.Drawing>
    </DrawingBrush>
  </Window.Resources>
  <Grid>
    <Rectangle VerticalAlignment="Center" HorizontalAlignment="Center" Width="200" Height="200" StrokeThickness="0" Fill="{(StaticResource Grid)}">
      <Rectangle.OpacityMask>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
          <GradientStop Color="Transparent" Offset="0"/>
          <GradientStop Color="Black" Offset=".1"/>
          <GradientStop Color="Transparent" Offset=".5"/>
          <GradientStop Color="Black" Offset=".9"/>
        </LinearGradientBrush>
      </Rectangle.OpacityMask>
    </Rectangle>
    <Rectangle VerticalAlignment="Center" HorizontalAlignment="Center" Width="205" Height="205" StrokeThickness="0">
      <Rectangle.Fill>
        <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,0.5">
          <GradientStop Color="White" Offset="0"/>
          <GradientStop Color="Transparent" Offset=".5"/>
        </LinearGradientBrush>
      </Rectangle.Fill>
    </Rectangle>
  </Grid>
</Window>
```
In the preceding example code, you create a `DrawingBrush` resource that defines your grid pattern. You then create a rectangle and assign your grid pattern to the `Fill` property as a `StaticResource`. You now have a visual element to play with. Next, you assign an opacity mask to the rectangle, defined as a `LinearGradientBrush`. You set the `StartPoint` and `EndPoint` properties of the brush to (0.5,0) and (0.5,1) so that you get a nice horizontal gradient. Next you define your gradient stops, and here’s where the opacity mask gets interesting. Notice that the colors you are using for your stops are `Black` and `Transparent`. The predefined color `Black` is defined by WPF as fully opaque, meaning it has no transparency. The predefined `Transparent` value, of course, exhibits full transparency. The result of this is that the `GradientStops`, which define `Black` as a color will expose the element underneath, while the `GradientStops`, which define `Transparent` as a color will expose whatever is behind the element (for example, make the portion of the element transparent as well).

To add a bit more style to your example, you create a second rectangle, whose fill is a `LinearGradientBrush`, in order to feather the left and right edges of your element. You set the `StartPoint` and `EndPoint` values to (0, 0.5) and (1, 0.5) to create a vertical gradient. Because this is not an opacity mask, you will use background color `White`, and the color `Transparent` to get a similar effect to that of the opacity mask. Because this rectangle will overlay your previous rectangle, you can simply define the edges of the gradient as white, and the center as transparent. Remember that this is not an opacity mask, so you’re not dealing with alpha values now; you’re dealing with color again.

Figure 6-12 illustrates the end result, which is a grid that appears to be burned into the background of your window.
Putting It All Together — Combining Effects

Now that you’ve explored many of the effects available to you in WPF, it’s time to build some examples that combine these effects. In this section, you create two projects that utilize many of the concepts that have been covered throughout this chapter. In the first example, you create a bouncing ball project. Your bouncing ball will bounce up and down on a glass table, casting a dynamic reflection. In the second example, you create an image viewer application. The image viewer displays a set of thumbnail images horizontally along the bottom of the window. When the mouse passes over one of the thumbnail images, the image will “fly in” to the image preview area, and the last image viewed will “fly out” of view.

Bouncing Ball with Reflection Example

We are all familiar with the Hello World scenario used to introduce a new language or platform in the programming world. I think the bouncing ball animation may be just as predominant in the graphics world. If not, maybe I can get the trend started as it is one of my favorite examples. The following example will combine animation, render transformations, opacity mask, and gradient brushes to create a bouncing ball. This bouncing ball will bounce up and down continuously on a plate of black glass, and its reflection will grow and shrink as the ball bounces. In addition, just to spice things up a bit, you’ll make your bouncing ball “draggable” horizontally.

Start by creating your XAML page. Open Visual Studio and create a new project using the .NET Framework 3.0 Windows Application template. Name the project BouncingBall and modify the default Window1.xaml as follows.

```
<Window x:Class="BouncingBall.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:BouncingBall"
    Title="Red Ball Bouncing on Glass"
    Height="400"
    Width="400"
    VerticalAlignment="Center"
    HorizontalAlignment="Center"
    Background="Black"
    MouseMove="BouncingBall_MouseMove"
    MouseDown="BouncingBall_MouseDown"
    MouseUp="BouncingBall_MouseUp"
    >

    <Window.Resources>
    <local:FlipConverter x:Key="FlipConverter" />
    </Window.Resources>

    <Grid VerticalAlignment="Stretch" HorizontalAlignment="Stretch">
    <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <Ellipse Grid.Row="0" Grid.Column="0" Fill="Red" Stroke="Red" StrokeThickness="5"
    Width="100" Height="50"/>
    </Grid>

    <TextBlock Grid.Row="0" Grid.Column="0" Text="Drag me!" FontWeight="Bold" FontSize="24"
    HorizontalAlignment="Center" VerticalAlignment="Center"/>
    </Window>
```
Chapter 6: Special Effects

First, modify the Window element as illustrated in the previous code. You are adding an XMLNamespace so you’ll be able to reference a TypeConverter that you’ll be creating shortly. Also, you’re adding events definitions for the MouseMove, MouseDown, and MouseUp events. These will wire-up your event handlers, which will provide the code to handle dragging your bouncing ball horizontally. Change the other Window attributes to match the previous code. You are also defining a single window resource, which contains your TypeConverter declaration. (You’ll create the TypeConverter shortly.) Last, you’ve added some row and column definitions to your Grid container.

```xml
<!-- Bouncing Ball Background -->
<Border Grid.Row="0" Grid.Column="0"
    x:Name="BouncingBallContainer"
    VerticalAlignment="Stretch"
    BorderBrush="White"
    BorderThickness="0"
>
    <Border.Background>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
            <GradientStop Color="Gray" Offset="0.8"/>
            <GradientStop Color="Black" Offset="0.99"/>
        </LinearGradientBrush>
    </Border.Background>
</Border>

Here, you’re simply creating a Border object, which will surround your bouncing ball and give you a chance to add a gradient background behind it. For the background, you define a simple LinearGradientBrush with a gradient flowing horizontally, made up of two colors, black and gray.

```xml
<!-- Bouncing Ball -->
<Canvas x:Name="BouncingBall"
    Background="Transparent"
    VerticalAlignment="Stretch">

    <Ellipse Width="100" Height="100"
        Fill="Red" Opacity=".8"/>
    <Ellipse Width="100" Height="100"
        Fill="Red" Opacity="1">
        <Ellipse.OpacityMask>
            <LinearGradientBrush StartPoint="0,1" EndPoint="0,0">
                <GradientStop Color="White" Offset="0.04"/>
                <GradientStop Color="#00000000" Offset="0.8"/>
            </LinearGradientBrush>
        </Ellipse.OpacityMask>
    </Ellipse>
    <Ellipse Width="60" Height="60"
        Fill="White" Opacity="0.3"
        Margin="20,0,0,0">
        <Ellipse.OpacityMask>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Color="White" Offset="0"/>
                <GradientStop Color="#00000000" Offset="0.8"/>
            </LinearGradientBrush>
        </Ellipse.OpacityMask>
    </Ellipse>
</Canvas>
```
Next, you are to define your bouncing ball. The bouncing ball is made up of three Ellipses and two opacity masks placed on a Canvas. Together, these elements make up the glassy, 3D visual effect of your ball. Opacity masks were covered earlier in the chapter, so you should be familiar with what’s happening thus far. This last example defines your ball, but now you’ll want to get it bouncing.

To get the ball bouncing, you’ve added a TranslateTransform to the outermost container for your bouncing ball, which is the Canvas. You aren’t actually moving anything in your transform definition; you’re letting the animation take care of this instead. This brings you to the next piece of functionality you need to add to your ball: the bounce. Before you can do that, however, you need to figure out when to start bouncing.

You want your animation to kick-off as soon as your Canvas completes loading. In your Canvas.Triggers object, you define a single EventTrigger, the target of which is the Canvas.Loaded event. When the Canvas.Loaded event fires, your EventTrigger will kick-off your Storyboard animation.

Finally, you need to create your animation. The animation will target the TranslateTransform you created earlier, specifically the Y property. You’ll move the ball along a Y axis from 1 to 55. This gets the ball bouncing up, but to get it back down you’ll need to do something else. To do this, you set the AutoReverse property to true. This will reverse the animation when it completes, sending the ball back down to its starting point. Finally, you want to keep this animation going, so you set the RepeatBehavior to a repeat value of Forever.
Next you need to create the reflection.

Here you are creating a `Border` object for the background glass effect. Within your black border, you are going to create the actual reflection of the bouncing ball. You use another `Border` object to define the reflection. You will use a `VisualBrush` to create the background of this border and, you guessed it, the visual you will target is your bouncing ball.
A VisualBrush is itself an object, so although it copies your bouncing ball, it does not copy the transform or animation. So you simply create a TranslateTransform for the reflection object. You don’t have to create another animation, however. If you were to create another animation, it might be hard to sync up the bounce with the reflection. In order to keep things in sync, you’ll take advantage of WPF data binding. You bind the X and Y properties of the TranslateTransform to the X and Y values defined in the bouncing ball TranslateTransform. That seems pretty great, but you want the reflection to translate in the opposite direction of your bouncing ball. When the bouncing ball goes up, the reflection should go down. In order to accomplish this, you need the inverse value of the Y property. For this, you’ll use a TypeConverter, which you’ll create shortly. You bind the X property as well so that when you drag the bouncing ball, your reflection will drag as well. In addition, you add a ScaleTransform to shrink up the reflection a bit, which will enhance your illusion of reflection on glass.

Finally, you also add an opacity mask to the reflection. This gives the impression of depth to the reflection.

Next you need to modify the Window1.xaml.cs code-behind file to add your dragging logic. You’ll simply define event handlers for MouseDown, MouseMove, and MouseUp events. In addition, you’ll create a TypeConverter, which simply returns the inverse value of an integer. This is part of how you get your reflected ball to move in the opposite direction of the main bouncing ball.

```csharp
using System;
using System.Windows;
using System.Windows.Data;
using System.Windows.Input;
using System.Windows.Media;

namespace BouncingBall
{
    public partial class Window1 : System.Windows.Window
    {
        private bool _dragging = false;

        public Window1()
        {
            InitializeComponent();
        }

        private void BouncingBall_MouseDown(object sender, MouseEventArgs e)
        {
            _dragging = true;
        }

        private void BouncingBall_MouseUp(object sender, MouseEventArgs e)
        {
            _dragging = false;
        }

        private void BouncingBall_MouseMove(object sender, MouseEventArgs e)
        {
            if (!_dragging)
```
The logic for dragging the ball is quite simple. You grab the X position of the mouse, the X position of your bouncing ball, and then perform a translate transform on the ball, using the formula (Mouse X Position – BouncingBall X Position).

You also finally get around to defining a TypeConverter, which will return the inverse of any integer passed to it. This is what you will use to get the opposite value of your bouncing ball TranslateTransform y-coordinate to your reflection. When the ball moves up 10 degrees, the reflection will move down 10 degrees (–10).

Figure 6-13 illustrates the results of running your example.

Of course Figure 6-13 can’t demonstrate the animation, but you get the idea. When you run the application, you can click any area of the window and drag the bouncing ball horizontally. The bouncing ball will follow the mouse. As the mouse moves horizontally, it continues to bounce up and down, and the reflection will follow.
Animated Image Viewer Example

In this example, you create an animated image viewer application. The structure of your application is going to be simple. You’re going to focus on bells and whistles rather than functionality. Your image viewer application will display a number of thumbnails across the bottom of the screen. When the mouse passes over a thumbnail, you’ll display the image at a larger size in a preview area, similar to the Windows Picture and Fax Viewer. When an image is selected, it will rotate into your preview area. While it rotates, the selected image will scale from its thumbnail size to full size. Similarly, when another image is selected, the current image will rotate and scale smaller until out of view. This gives the effect of an image “flying” into and out of the preview area. You’ll be using transformations and animation to make all of this happen. You’ll also use a VisualBrush to display the selected image in the preview area. Additionally, you’ll use a couple of bitmap effects to give your application some polish, and a Slider object to control the animation speed.

Now that you know what you are going to build, it’s time to jump into the code. Open Visual Studio and create a new project using the .NET Framework 3.0 Windows Application template. Name the project ImageViewer. You will need to add an Images directory to your project, and add some images to it. Make sure all images have their Build Action property set to Resource. Open the Window1.xaml file and modify it as follows:
Chapter 6: Special Effects

Up to this point, you’ve simply defined your Window element, a Style for your Image objects, and a Slider object to control the animation speed. The Slider control binds to a dependency property in your Window1 code-behind, which you’ll create shortly. Next you create your image thumbnails.

You are simply placing five images into a horizontal StackPanel and setting the Width property of each Image to 50. This is what we meant by not focusing on functionality. If you wanted to extend this example later, you could create a custom thumbnail Panel, read in files from disk, or load the images dynamically using an XMLDataProvider. You’re focusing on effects in this chapter so we’ll leave these details up to you. For now, this suits your purpose.
Here you simply add a TextBlock to display the image name when an image is loaded into the preview area. You're placing all of your elements in a DockPanel. You set the DockPanel.Dock property for your TextBlock to Bottom. You also set the DockPanel.Dock property to Bottom for the StackPanel. When two or more objects define the same value for the Dock property of a DockPanel, each will get docked in the order in which it is defined. So your StackPanel is docked to the bottom-most edge of your DockPanel because it is defined in your XAML file first, and your TextBlock is docked to the top edge of your StackPanel because it is defined second.

Finally, you define your image preview area as a Canvas, named imageCanvas. You'll use the canvas to display your image previews by painting the canvas's Background property with a VisualBrush that points to the selected thumbnail. Your canvas in this example is surrounded by two Border objects. The first Border you define will provide a white border area around your preview area. For the nested Border, you create a LinearGradientBrush to fill the background of your preview area. You also add a couple of bitmap effects, a drop shadow, and a slight bevel to give your preview area some depth and polish.

Now that you've defined your UI in XAML, let's jump to the code behind, where you'll add your transformation logic and animation to get the images "flying." Open up the Window1.xaml.cs file and modify it as follows.
using System.Windows.Media;

namespace ImageViewer
{
    public partial class Window1 : System.Windows.Window
    {
        object _sender;

        public Window1()
        {
            InitializeComponent();
            TransformGroup group = new TransformGroup();
            ScaleTransform scale = new ScaleTransform();
            group.Children.Add(scale);
            RotateTransform rot = new RotateTransform();
            group.Children.Add(rot);
            this.imageCanvas.RenderTransform = group;
        }

        public static DependencyProperty AnimationDurationProperty =
            DependencyProperty.Register("AnimationDuration", typeof(double),
            typeof(Window1), new PropertyMetadata(500.0, null));

        public double AnimationDuration
        {
            get { return (double)GetValue(AnimationDurationProperty); } 
            set { SetValue(AnimationDurationProperty, value); } 
        }

        private void Image_MouseEnter(object sender, MouseEventArgs e)
        {
            _sender = sender;
            if (this.imageCanvas.Background != null)
                FlyOutCurrentImage();
            else
                FlyInNewImage(sender);
        }
    }
}

Up to this point, you’ve simply defined some basic class constructs. In the Window1 constructor, you’ve added a TransformGroup to your imageCanvas. You’ve placed two transforms in the group—a ScaleTransform and RotateTransform. You’ve added a single dependency property named AnimationDurationProperty to your class. This will be used by the animation code you’ll write shortly. This is also the property that the Slider you created earlier binds to. Additionally, you create an event handler named Image_MouseEnter. This event will fire off the animation and transformation code whenever the mouse moves over a thumbnail.

Next, you define three methods: FlyInNewImage, FlyOutCurrentImage, and Animation_Completed. These methods control the animations and transformations you will apply to your preview area canvas.
private void FlyInNewImage(object sender)
{
    if (this.imageCanvas.RenderTransform == null)
        return;

    string imageName = ((Image)sender).Source.ToString();
    imageName = imageName.Substring(imageName.LastIndexOf("/") + 1);
    this.txtImageName.Text = imageName;

    VisualBrush vb = new VisualBrush((Visual)sender);
    this.imageCanvas.Background = vb;

    TransformGroup group = (TransformGroup)this.imageCanvas.RenderTransform;
    ScaleTransform scale = (ScaleTransform)group.Children[0];
    RotateTransform rot = (RotateTransform)group.Children[1];

    this.imageCanvas.RenderTransformOrigin = new Point(0, 0);
    scale.ScaleX = .5;
    scale.ScaleY = .5;
    rot.Angle = 45;

    DoubleAnimation rotAnimation = new DoubleAnimation(0, 
        TimeSpan.FromMilliseconds(AnimationDuration));
    rot.BeginAnimation(RotateTransform.AngleProperty, rotAnimation);
    DoubleAnimation scaleAnimation = new DoubleAnimation(1, 
        TimeSpan.FromMilliseconds(AnimationDuration));
    scale.BeginAnimation(ScaleTransform.ScaleXProperty, scaleAnimation);
    scale.BeginAnimation(ScaleTransform.ScaleYProperty, scaleAnimation);
}

The FlyInNewImage method accepts a parameter of type object, which will be the image that the mouse is over, and is passed in from your MouseEnter event handler. The FlyInNewImage method essentially does four things:

- Sets the Text property TextBlock to display the currently selected image name
- Adds the image the mouse is currently over to the Background property of the imageCanvas by setting the Visual property of the VisualBrush
- Sets up the scale and rotate transformation values for the imageCanvas
- Kicks off the animations.

private void FlyOutCurrentImage()
{
    if (this.imageCanvas.RenderTransform == null)
        return;

    TransformGroup group = (TransformGroup)this.imageCanvas.RenderTransform;
    ScaleTransform scale = (ScaleTransform)group.Children[0];
The **FlyOutCurrentImage** is very similar to the **FlyInNewImage** method. There are two key differences, however. First, the values you set for your transformations are different, and the **RenderTransformOrigin** is set to the upper-right corner of the preview area. Second, the **rotAnimation** object defines a callback method that will be called when the animation completes. This callback ensures the “fly out” will finish before the “fly in” begins. Finally, the **Animation_Completed** method simply calls the **FlyInNewImage** method.
As with the last example, Figure 6-14 doesn’t really do the application justice. When you run the application, and mouse over the image thumbnails, you’ll see the animation and transformations in action. Moving the Animation Duration slider to the right slows the animation speed and, of course, moving it left speeds the animation up. You can see the drop shadow and bevel of the preview area provide a nice framing effect for the image you are previewing.

Summary

WPF offers a lot of effects you can use to achieve some pretty cool UI. Using the foundations provided, the possibilities are truly endless. In this chapter you have explored all of the brushes, bitmap effects, and transformations provided by WPF. You’ve seen how they work, how you can apply them to elements in your UI, and how you can combine them to achieve a unique and visually appealing UI. You also explored triggering animations off of events and using animations to perform transformations. Finally, you created a couple of sample applications by combining the concepts that you have learned about in this chapter.

After reading this chapter you can now:

- Describe the various brushes available to you in WPF, including the SolidColorBrush, LinearGradientBrush, RadialGradientBrush, ImageBrush, DrawingBrush, and VisualBrush.
- Understand the functionality and capabilities of each Brush class.
- Describe the various bitmap effects provided by WPF including the BevelBitmapEffect, BlurBitmapEffect, DropShadowBitmapEffect, EmbossedBitmapEffect, and OuterGlowBitmapEffect.
- Understand the functionality and capabilities of each BitmapEffect class.
- Describe the various transformations provided by WPF, including TranslateTransform, ScaleTransform, SkewTransform, and RotateTransform.
- Understand the functionality and capabilities of each Transform class.
- Understand the difference between a LayoutTransform and a RenderTransform, and when to use each.
- Describe and use an opacity mask.