CHAPTER 8

An Example Control Using Just the Tools Palette

IN THIS CHAPTER

- Building a Gel Button from Scratch
- Introduction to the Timeline and Timeline Events (Triggers)
- A Look at RoutedEvents
- Creating a Control Template

My original plans for this chapter were that it be around 16 pages long, which was an estimate based on some of the first CTP's of what was called Expression Interactive Designer. Things have cleaned up a lot since those early days of the application but in saying this, there are a lot of new features that I want to show you. As a result, it may be a longer journey to Gel button utopia than was first planned, but with any luck you will have an even better experience.

Building a Gel Button from Scratch

It was a strange coincidence that when I first saw Expression Interactive Designer being demonstrated, I had just spent the better part of three days making what I thought was sure to be the most awesome Gel button making application in .NET. The application I was working on at the time called for a cutting-edge look and feel, and it was for one of those rare companies that didn't care how long things took as long as the results were mind blowing.

Three days. The results were brilliant because I had based the results on early Vista builds; but it took three days just to get to that point. I took a day off to go to a demonstration put on by Microsoft, and you can imagine my eyes when I saw a Gel button built from scratch in around 5 minutes. I knew then that what I had been asking for—for years—had finally arrived: an interface development tool built for designers but which, at the same time, allowed the developer to ensure functionality and maintain performance.

It was not even the fact that the Gel button shown in the Demonstration was that good, it was more about the fact

that the results had been stored in a template and could be used at will on any other button object.

The thing to remember about Blend is that everything you see and do can essentially be saved as a template and can be stored and retrieved from what is known as a *Resource Dictionary*. So you can design a complete series of buttons, drop-down lists, Treeview controls—whatever you want—and store these templates in your application or one of your Resource Dictionaries.

The term *template* is being used here rather generically, and you may think that I should have said *Skin*. Remember, however, what I said about everything being able to be saved as a template? Well, that includes data bindings, styles, brushes (made from the color palette), and the list goes on. So, you can see how the term *Skin* is incorrect.

It does get confusing at first when you are trying to get used to the idea of templates, what certain templates are for, what the difference is between a style and a control parts template, and so on. We will get to all of this in a bit. But for now, let's just get on with building the button and worry about the finer points of the templating side of things later. (Chapter 12, "Templates," details the different template types and how to create and modify them.)

The goal is to design and build a button similar to the one shown as the Play button in Figure 8.1, which is actually the button control used in Windows Media Player 11. As mentioned previously, without the help of Blend, you could work for days just trying to get all the lighting effects right to try and replicate this, but thankfully you don't have to worry about that any longer.

FIGURE 8.1 The visually stunning controls of Windows Media Player 11.

Working the Layers

You are going to use just the tools found in the Toolbox, so it is nice and easy for you to take this sample and apply it to other areas of design without having to worry so much about how other controls will integrate. You may think the subtitle of this area is a little strange, but as you will see, you will be building this control using carefully constructed layers, one on top of the other to give the overall appearance.

Before you begin, take a moment to understand the Brush Suggestion tables presented in this chapter (begin with Table 8.1). They will provide a little bit of insight into working with the Brushes. After this chapter, however, you are on your own!

All the information shown in the above table and others like it, correlate directly to the Brushes and Appearance panels located within the Properties palette. `Fill`, `Stroke`, `Foreground`, and other brush properties are shown as a suggestion in this format.

TABLE 8.1 Sample 1 Brush Suggestion

Brush Property	Brush Type	Overall Opacity	Stroke Thickness		
Fill	Linear Gradient	55%	4		
Gradient Stop	**Position**	**Red**	**Green**	**Blue**	**Alpha**
1	50%	140	26	204	100%
2	100%	255	255	255	100%

XAML Listing (Sample 1):

```
<LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
    <GradientStop Color="#FF8C1ACC" Offset="0.5"/>
    <GradientStop Color="#FFFFFFFF" Offset="1"/>
</LinearGradientBrush>
```

As you can see from the suggested Brush and in Figure 8.2, you are looking at setting the `Fill` property brush, using a Linear Gradient brush type, which has two Gradient Stops added to it, which is done by default.

FIGURE 8.2 All the relevant areas of the suggested Brush.

What if There Are More Than Two Color Selectors (Gradient Stops)?

In the event you see more than two Gradient Stops, you only need to click on the gradient display graph and another Gradient Stop will be created. To remove them, simply click and hold the Stop and then drag down and off the graph.

You should also note that individual Gradient Stops can have a different opacity value than the overall opacity, which is located in the Appearance panel. These individual Stops are also given a position that refers to the position on the color gradient graph, the far left of the graph representing 0% and the far right 100%.

As you can see in Figure 8.2, the first Gradient Stop is located approximately 50% along the graph, while the second remains firmly to the far right.

Also notice the Stroke thickness, which is again located in the Appearance panel. This panel also contains additional property settings for the `Stroke` property.

Lastly, if you do not see a value in the suggested Brush, it is safe to assume that you can use the default values that it has been given by Blend. In the case of a Brush being suggested as a Solid Color Brush type, you will not see Gradient Stops within the table because none are available with this type of brush.

Let's get into it!

1. Start by creating a new project in Blend. You can call it whatever you would like, but you are not going to be saving this application along the way. (You can always name the project if you are not confident of completing the chapter in one sitting.)
2. Make sure you have the Design Workspace selected by either depressing F6 (to cycle the workspace) or selecting it from the Windows menu.
3. Start by making the base of the button first. As you see, it is a circular shape, so select the Ellipse tool from the Toolbox and draw a shape of arbitrary size on your artboard. Make it nice and big, though, because it will scale how you want it to later.
4. The shading in this type of design is what gives the lighting effect, so pay close attention to it and your results should be even better than mine. Now, select the Ellipse element in the Objects and Timeline palette and right-click to display a context menu with the Rename option on it. Name this element ButtonBase.
5. Hopefully you have the Properties palette open; if not, select it from the Windows menu and find the Brushes area of the palette. First select the `Fill` Brush property. For now set it to a Solid Color Brush with a nice dark deep blue color. Take note of the color suggestions I make or change the color at will along the way (see Table 8.2).

   Next, you use the Pen tool in much the same way as you did in the previous chapter to create the shaded area that represents the gloss on the button. Be careful where you add the points, particularly the points along the center area of the button. The shape you create is critical to eliciting the correct effect. Figure 8.3 shows where I added points with the Pen tool, starting on the far left and eventually going full-circle and returning to that point to close the path object.

TABLE 8.2 Sample 2 Brush Suggestion

Brush Property	Brush Type	Overall Opacity	Stroke Thickness
Fill	Solid Color	100%	1
Red	**Green**	**Blue**	**Alpha**
0	25	158	100

XAML Listing (Sample 2):

```
          <Ellipse HorizontalAlignment="Left" VerticalAlignment="Top"
  Width="100" Height="100" Fill="#FF00199E" Stroke="#FF000000"/>
```

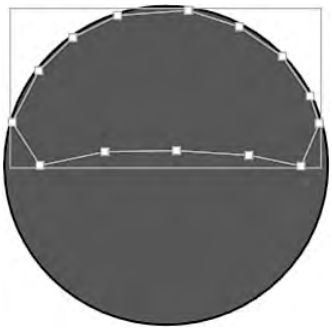

FIGURE 8.3 Add points with the Pen tool.

6. Now select the Direct Selection tool from the Toolbox palette and begin to smooth out the object. Use the image in Figure 8.1 as a reference. Remember, the goal is to create a nice-looking gloss object. At this point, your path object is most likely the same blue color as the ButtonBase element you created in the previous steps. You can leave it like that for the moment. I'll provide some brush suggestions a little later on. For now, rename this path element to Hood.
7. Next, add the Arrow element. Because you are enjoying using the Pen tool so much, you might as well get this part of the task completed. Select the Pen tool again and add the points for the arrow. When I created the arrow shown in Figure 8.4, I did not just add three points to make the arrow. Instead, I placed a point at the beginning and end of the straight line sections of the arrow, and then selected them to place a curve in the corners. Remember to select the first point with the Pen tool again to close the path element.
8. Rename the arrow element to Arrow. You have now completed all the shape elements required for the button. It probably looks like a blue blob at the moment; but remember that the magic is all in the colors and the shading. Figure 8.5 illustrates how mine turned out. Does yours look similar?

FIGURE 8.4 The completed arrow shape.

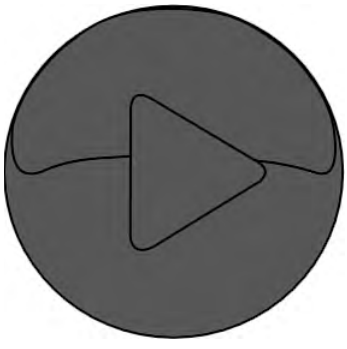

FIGURE 8.5 All the shapes completed.

9. Now, hide both the Arrow and the Hood elements by selecting the eye icon displayed next to the elements in the Objects and Timeline palette. Concentrate on each layer now, building the brushes as you go. Select the ButtonBase element in the Objects and Timeline palette and apply the following Brush suggestions in Table 8.3 to it.

TABLE 8.3 Sample 3 Brush Suggestion

Brush Property	Brush Type	Overall Opacity	Stroke Thickness		
Fill	Radial Gradient	100%	1		
Gradient Stop	**Position**	**Red**	**Green**	**Blue**	**Alpha**
1	50%	0	229	255	99%
2	100%	0	25	158	100%

XAML Listing (Sample 3):

```
<RadialGradientBrush>
    <GradientStop Color="#FC00E5FF" Offset="0.5"/>
    <GradientStop Color="#FF00199E" Offset="1"/>
</RadialGradientBrush>
```

10. Next, apply the Stroke property settings described in Table 8.4.

TABLE 8.4 Sample 4 Brush Suggestion

Brush Property	Brush Type	Overall Opacity	Stroke Thickness		
Fill	Linear Gradient	100%	1		
Gradient Stop	**Position**	**Red**	**Green**	**Blue**	**Alpha**
1	50%	255	255	255	100%
2	100%	29	37	194	100%

XAML Listing (Sample 4):

```
<LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
    <GradientStop Color="#FFFFFFFF" Offset="0.5"/>
    <GradientStop Color="#FF1D25C2" Offset="1"/>
</LinearGradientBrush>
```

11. Select the `Fill` property again. This time select the Brush Transform tool from the Toolbox. Figure 8.6 shows how to manipulate the brush to place the highlights in the required position.

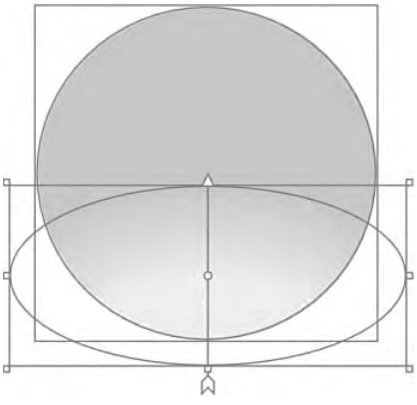

FIGURE 8.6 The Fill Brush Transform position.

12. With the Brush Transform tool still selected, change the property in the Brushes panel to `Stroke`. Manipulate the Brush highlights again to as close to the `Fill` property as possible. It is a little difficult because you do not get the bounding box that was supplied with the `Fill`, but you can just switch between the two properties to get a rough similarity.

13. Now make the Hood element visible again and apply the Brush suggestions included in Tables 8.5 and 8.6.

TABLE 8.5 Sample 5 Brush Suggestion

Brush Property	Brush Type	Overall Opacity	Stroke Thickness		
Fill	Radial Gradient	100%	1		
Gradient Stop	**Position**	**Red**	**Green**	**Blue**	**Alpha**
1	30%	20	71	206	12%
2	100%	255	255	255	95%

XAML Listing (Sample 5):

```
<RadialGradientBrush>
    <GradientStop Color="#1E1447CE" Offset="0.3"/>
    <GradientStop Color="#F2FFFFFF" Offset="1"/>
</RadialGradientBrush>
```

TABLE 8.6 Sample 6 Brush Suggestion

Brush Property	Brush Type	Overall Opacity	Stroke Thickness		
Stroke	Radial Gradient	100%	2		
Gradient Stop	**Position**	**Red**	**Green**	**Blue**	**Alpha**
1	55%	255	255	255	100%
2	100%	27	46	180	23%

XAML Listing (Sample 6):

```
<RadialGradientBrush>
    <GradientStop Color="#FFFFFFFF" Offset="0.55"/>
    <GradientStop Color="#3A1B2EB4" Offset="1"/>
</RadialGradientBrush>
```

Data Property Available in the Appearance Panel

Do you recall in Chapter 7, "Using the Blend Toolbox: Tools and Common Controls," I mentioned the mini-language used to create paths? You can see the syntax that has been applied for you in the Data properties located in the Appearance panel.

14. The secret to the Hood element is within the Brush transforms. Pay close attention to the following figures, which show you the `Fill` property transform (Figure 8.7) and the `Stroke` property transform (Figure 8.8).

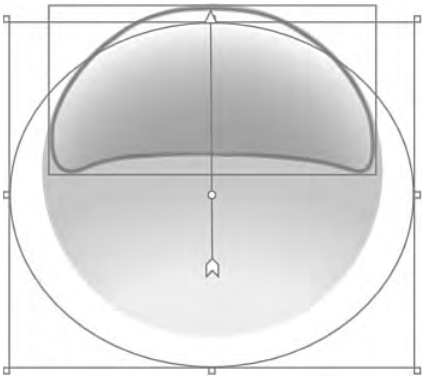

FIGURE 8.7 The Fill brush transform.

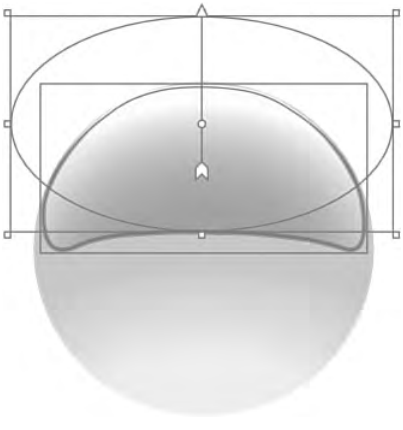

FIGURE 8.8 The Stroke property transform.

15. Make the Arrow element visible again and apply the Brush suggestion provided in Tables 8.7 and 8.8.

TABLE 8.7 Sample 7 Brush Suggestion

Brush Property	Brush Type	Overall Opacity	Stroke Thickness		
Fill	Radial Gradient	100%	1		
Gradient Stop	**Position**	**Red**	**Green**	**Blue**	**Alpha**
1	60%	255	255	255	100%
2	100%	186	186	186	100%

XAML Listing (Sample 7):

```
<RadialGradientBrush>
    <GradientStop Color="#FFFFFFFF" Offset="0.6"/>
    <GradientStop Color="#FFBABABA" Offset="1"/>
</RadialGradientBrush>
```

TABLE 8.8 Sample 8 Brush Suggestion

Brush Property	Brush Type	Overall Opacity	Stroke Thickness		
Stroke	Linear Gradient	100%	1		
Gradient Stop	**Position**	**Red**	**Green**	**Blue**	**Alpha**
1	0%	0	0	0	100%
2	100%	255	255	255	100%

XAML Listing (Sample 8):

```
<LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
    <GradientStop Color="#FF000000" Offset="0"/>
    <GradientStop Color="#FFFFFFFF" Offset="1"/>
</LinearGradientBrush>
```

16. Now apply the Brush Transform for the `Fill` property as shown in Figure 8.9.

With any luck, you now have a gorgeous Gel play button. I hope you also now have more confidence in your ability to create this style of element.

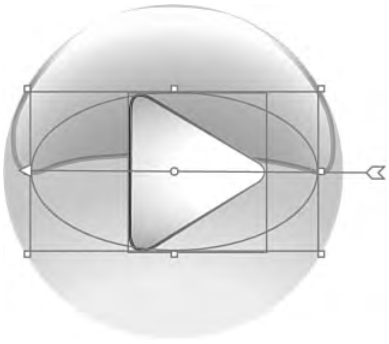

FIGURE 8.9 The Fill property transform.

If you haven't been able to re-create it exactly as shown, don't worry too much about it. You can always try again—and each time you do it, you will get better at placing the curves and the lines in the correct places.

If you want to cheat a little, which is fine by me, you can always just open a new project and type in the XAML provided in Listing 8.1 by hand. It might be quicker, however, to try to use the brushes again.

LISTING 8.1 Complex Button Shapes

```
<Grid x:Name="LayoutRoot">
    <Ellipse Opacity="1"
     StrokeThickness="1"
     Margin="249.746,123.849,245.592,188.287"
     x:Name="ButtonBase">
    <Ellipse.Fill>
        <RadialGradientBrush GradientOrigin="0.5,1.058">
            <RadialGradientBrush.RelativeTransform>
                <TransformGroup>
                    <ScaleTransform CenterX="0.5"
                         CenterY="0.5"
                         ScaleX="1.159"
                         ScaleY="0.535"/>
                    <SkewTransform AngleX="0"
                         AngleY="0"
                         CenterX="0.5"
                         CenterY="0.5"/>
                    <RotateTransform Angle="0"
                         CenterX="0.5"
                         CenterY="0.5"/>
                    <TranslateTransform X="0.005"
                         Y="0.306"/>
                </TransformGroup>
```

```
            </RadialGradientBrush.RelativeTransform>
            <GradientStop Color="#FC00E5FF" Offset="0.502"/>
            <GradientStop Color="#FF00199E" Offset="1"/>
        </RadialGradientBrush>
    </Ellipse.Fill>
    <Ellipse.Stroke>
        <LinearGradientBrush EndPoint="0.498,0.561"
             StartPoint="0.498,1.101">
            <GradientStop Color="#FFFFFFFF" Offset="0"/>
            <GradientStop Color="#FF1D25C2" Offset="1"/>
        </LinearGradientBrush>
    </Ellipse.Stroke>
    </Ellipse>
    <Path Opacity="1"
        Stretch="Fill"
        StrokeThickness="2"
        Margin="251.733,123.478,247.072,0"
        x:Name="Hood"
        VerticalAlignment="Top"
        Height="69.625"
        Data="M252.42643,173.06279 C254.48207,164.32278 258.4611,157.86994
264.19778,150.36784 268.71292,144.67081 272.29185,140.86135 277.38975,137.28246
283.23859,132.40728 290.1125,128.65259 298.49689,126.44613 307.17547,123.79786
325.28673,122.76574 335.8403,125.83263 345.03072,126.9941 352.00882,131.50965
357.1504,135.03366 362.63063,138.47065 367.103,142.89265 370.95122,146.68778
376.39771,152.59304 378.77487,156.69182 381.30189,162.02215 383.24375,166.02556
385.00012,170.21081
   385.56368,173.88104 387.58732,189.00662 382.95216,194.52641 376.1114,192.02416
368.52424,189.64282 361.15644,188.11773 354.01074,187.45957 343.37655,186.04817
333.10954,185.37263 323.262,185.53773 312.98013,185.00436 302.89249,185.17094
292.9937,186.01819 281.67796,186.41552 272.4693,188.33731 265.36788,191.78369
255.86256,195.09834 251.16319,187.8839 252.42643,173.06279 z">
        <Path.Fill>
            <RadialGradientBrush GradientOrigin="0.501,0.685">
                <RadialGradientBrush.RelativeTransform>
                    <TransformGroup>
                        <ScaleTransform CenterX="0.5"
                            CenterY="0.5"
                            ScaleX="1.239"
                            ScaleY="2.049"/>
                        <SkewTransform AngleX="0"
                            AngleY="0"
                            CenterX="0.5"
                            CenterY="0.5"/>
                        <RotateTransform Angle="0"
                            CenterX="0.5"
```

```
                        CenterY="0.5"/>
                    <TranslateTransform X="-0.002" Y="0.624"/>
                </TransformGroup>
            </RadialGradientBrush.RelativeTransform>
            <GradientStop Color="#1E1447CE" Offset="0.268"/>
            <GradientStop Color="#F2FFFFFF" Offset="1"/>
        </RadialGradientBrush>
    </Path.Fill>
    <Path.Stroke>
        <RadialGradientBrush GradientOrigin="0.5,0.677">
            <RadialGradientBrush.RelativeTransform>
                <TransformGroup>
                    <ScaleTransform
                        ScaleX="1.172"
                        ScaleY="1.277"/>
                    <SkewTransform AngleX="0"
                        AngleY="0"/>
                    <RotateTransform Angle="0"/>
                    <TranslateTransform X="-0.092" Y="-0.404"/>
                     </TransformGroup>
                </RadialGradientBrush.RelativeTransform>
                <GradientStop Color="#FFFFFFFF" Offset="0.55"/>
                <GradientStop Color="#3A1B2EB4" Offset="1"/>
            </RadialGradientBrush>
        </Path.Stroke>
    </Path>
    <Path Stretch="Fill"
        StrokeThickness="1"
        Margin="300.844,160.078,277.043,223.929"
        x:Name="Arrow"
        Data="M301.34447,218.38417 L301.51887,166.19908 C301.82203,160.49434
304.64037,159.30446 309.53978,161.85022 L351.5611,186.89926 C356.60548,189.32806
357.59647,192.21381 354.17685,195.5968 L310.58607,222.38521 C305.13799,224.84771
302.0932,223.45672 301.34447,218.38417 z">
    <Path.Fill>
        <RadialGradientBrush GradientOrigin="1.17,0.508">
            <RadialGradientBrush.RelativeTransform>
                <TransformGroup>
                    <ScaleTransform ScaleX="2.091" ScaleY="1"/>
                    <SkewTransform AngleX="0" AngleY="0"/>
                    <RotateTransform Angle="0"/>
                    <TranslateTransform X="-0.714" Y="-0.008"/>
                </TransformGroup>
            </RadialGradientBrush.RelativeTransform>
            <GradientStop Color="#FFFFFFFF" Offset="0.573"/>
            <GradientStop Color="#FFBABABA" Offset="1"/>
```

```
        </RadialGradientBrush>
    </Path.Fill>
    <Path.Stroke>
        <LinearGradientBrush EndPoint="1,0.5" StartPoint="0,0.5">
            <GradientStop Color="#FF000000" Offset="0"/>
            <GradientStop Color="#FFFFFFFF" Offset="1"/>
        </LinearGradientBrush>
    </Path.Stroke>
    </Path>
</Grid>
```

Now that you have the shapes needed for the button, what is missing is the functionality that a button is required to have. It would be a bit of a mess trying to move all these elements around from place to place. You also want to contain the design you have just slaved over and be able to apply it as a template to other button elements if you choose. To accomplish this portion of the task, embrace the magic of the Make Button tool.

Using the Make Button Tool

The Make Button tool does pretty much what its name implies. It turns UIElements into Button elements. It doesn't just rename the object type to achieve its goals; it adds triggers and declares events like *click*—all the things that you need for a button to function as you or anyone else using your button would expect.

You should (as good practice), test your elements to make sure they scale correctly. You should also Group all the elements together, using the *Group Into* option on the context menu of the Objects and Timeline category by taking the following steps:

1. Open the Objects and Timeline palette and select each of the three elements in the tree by holding down the Shift or Ctrl key. Make sure you only select the three elements you created.
2. Right-click on one of the highlighted elements. A popup context menu containing the Group Into menu item appears. Move your mouse over the item to reveal a further context menu with all the optional element types that you can select to contain your elements.
3. For this example select Grid. There are several reasons for this, but the most important is that the Grid element will allow the three elements to coexist with the same layout settings that they currently have. If you were to select Stack panel, you could find a fairly unique button being developed, so I wouldn't use that at this stage.
4. You now have a new element named [Grid]. The brackets indicate that it has child elements, namely your elements, contained within it. Rename the Grid element to MyGelPlayButton.
5. If you now select the MyGelPlayButton element, you can resize it at will and test the scaling ability of your element. Note that if you elongate the element, then the Hood doesn't look right—or the Arrow disappears all together if you make the MyGelPlayButton very thin. These design problems all need to be taken care of to

ensure that the button scales and still remains the same visually as it does at the original size you created it in. You can come back later and do this if you wish. Chapter 5, "UIElement: Control Embedding," for more details on Grid layout scenarios and Chapter 6, "Panel-Based Containers," for more information on object embedding. At this point, know that this type of combined element (Circular) is very hard to set correct scaling on, mainly because the center points of each element are in a different position.

6. With MyGelPlayButton selected in the Timeline palette, click on Tools from the main menu at the top of the application and then select Make Button. A dialog box displays that contains the information about the button as shown in Figure 8.10.

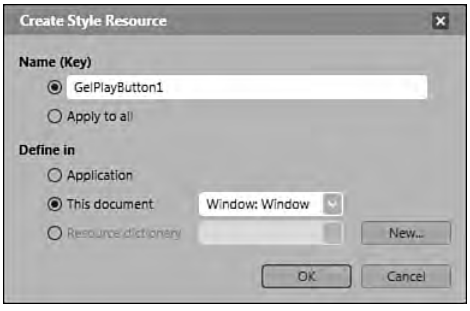

FIGURE 8.10 The Style Resource dialog box.

Notice that I changed the name of the button to *GelPlayButton1* in the Resource name (key) area of the dialog box. This is particularly important because, as you will find out soon enough, defining a manageable naming convention for Templates is half the battle to using them effectively.

Also note that you are defining the button element in this document only, which means that you can only apply this resource to other button elements in this window element. That is called *Scope*. You will see shortly how to put the button into a Resource Dictionary (RD) to enable you to use it anytime and at any place within your solution(s).

For now, make the changes as shown in Figure 8.10, and then click the OK button.

You should have noticed that a few interesting things have happened to the element. First, the text *Button* appears in the center of the button element. Second, if you look in the Timeline palette you will see that the Grid element is gone and has been replaced by the [Button] Button, which shows you that the type is now that of Button and the name is the default name of Button.

Why is there text in the center of the button? The Make Button tool applies default button elements within the button to ensure that the basics of a button are created. One of the default properties of a button (any button) is that it contains what is known as a *ContentPresenter* (see Chapter 6 for more details about content controls).

If you switch to the Properties palette for a minute, you should be able to navigate to the Common Properties panel. In here you will see a property named *Content* with the text Button displaying.

If you remove the text from this property, all is well and the button begins to resemble what it previously was. In the same area, just below the Content property is a property named *Cursor*. Select the drop-down list beside the label and look for the Hand cursor and select it.

Realistically, to make this button more reusable, you would probably make the button without the Triangle embedded and instead create the triangle as a separate path and add it as the Button's content property value instead of text. This way you could create different shapes to add to the Button without the need for re-creating the button for each symbol you wish to use.

Introduction to the Timeline and Timeline Events (Triggers)

You are now going to briefly see some features of the Animation Timeline as well as the timelines associated Event Triggers and Property Triggers, which are also collectively referred to as Triggers from time to time. Chapter 14, "Animations with Storyboards," provides more detailed information. You use these triggers to add the finishing touches that really set your button apart from other buttons.

Maybe you have used Flash previous to using the Blend application. In case you haven't, it is helpful to understand a little of what the Timeline does and how it ties in with any code you may wish to add to the object.

Timelines do indeed use measurements of time as an instrument of functionality. This really means that while you can create an animation of sorts with any object using the Timer class, you would realistically miss out on certain key points in time when the animation is meant to be applying certain property values like position or size or both for example. The Timeline-based solution provided in Blend is really simple to use; however, because of this simplicity, you also lose out on some of the functionality you either need to add within XAML or you can code the entire animation before you set it off in the .NET backend.

When you animate an element, you are, in point of fact, animating a certain property or set of properties on that element. For instance, if you animate the width of an element, you set the width property at the beginning of the animation to, perhaps, its current value of 50 and the ending point anywhere you wish, perhaps 150 wide. The animation backend uses an algorithm to work out exactly what width the element should be at all points along the duration of the animation to ensure that the end property value is met at the end of the specified duration. That provides a nice, smooth animated result.

Timeline events allow you to specify when an animation should start, pause, stop, resume, complete (skip to fill), or be removed. They are tied directly to a Timeline and are derived

(for lack of a better word) from events available to the object or element that you are intending to animate.

Look at the process like this:

1. Figure out which element event will trigger your animation.
2. When your chosen trigger fires, what object(s) or element(s) do you want to animate?
3. Which Timeline will contain the animation?
4. What action will be performed on the animation?

Using this process you can say: "When the Window.Loaded event is raised, I want the Timeline OnWindowLoaded to Begin animating my Button element". You will shortly see how this intuitive action list is implemented in the Blend UI. It really *is* as simple as that.

You can create as many Timelines as you wish and animate any number of element properties as the animation progresses. There are different types of animation described in Chapter 14. For now you will be using what is known as the *Key Frame* technique (see Figure 8.11). This means that you will be specifying what values certain properties are to contain at a particular point in the animation lifecycle on the Timeline (using a KeyFrame).

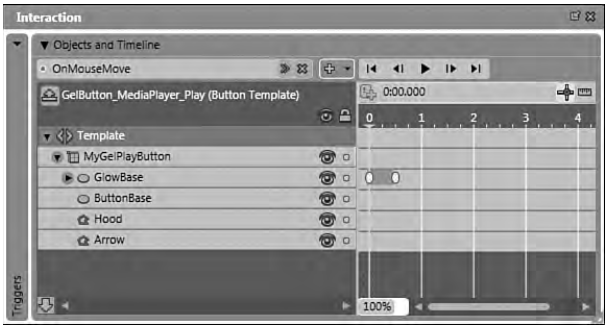

FIGURE 8.11 A Timeline with Key Frames set on it.

That is animation in a nutshell. You can only animate a property—and only certain types of properties. You can provide a start and/or end property value or values at arbitrary points in time; and last but not least, you specify a duration.

Animations can get quite complex. There are several other properties that can be set within the Blend animation set, but we will stick with using the Blend Timeline interface for animating the button.

Adding Animated Glow

Let's say that when the user's mouse moves over [the button], you want to get a glow to emanate from behind the button in order to provide a perception of depth as well as to provide valuable user feedback. You can create that effect using animation techniques.

Now that you have your button element viewable on the artboard, you have essentially created a template that the button visually represents (more on templates in Chapter 12). Follow these steps:

1. Select the button in the Objects and Timeline palette and right-click to display the context menu. Move your mouse over the Edit Control Parts (Template) menu item, which will bring up a submenu. In that submenu, select Edit Template.
2. You now can see the elements you created previously using the Make Button tool. There is also an additional element named [ContentPresenter] that was placed here by the Make Button tool. It is responsible for the text that is shown on the button. Select this new element, right-click on it and select Delete. There is no requirement to include text on this style of button.
3. It is now time to add an additional element to represent the Glow of the button. Because you want the glow to begin from the back of the button, you should select ButtonBase, right-click on it, and select Copy.
4. Now select and activate the MyGelPlayButton element so you can see the yellow bounding box around it on the artboard. Right-click on it and select Paste. The ButtonBase_Copy element is added, but it is the top layer of all the child levels. To remedy this, right-click the element and find the Order menu item, for which a submenu displays with an item named Send to back.
5. Rename the ButtonBase_Copy element to GlowBase.
6. The first action you are going to take with GlowBase is to resize it to the intended maximum size of the glow that the element will produce. Find the Properties palette and then the Layout panel. Make sure that both the `HorizontalAlignment` and `VerticalAlignment` properties are set to Stretch.
7. Just below the Layout Panel is the `Alignment` property called Margin with four input boxes representing the Top, Bottom, Left and Right margin values. Enter the value -50 in all four property inputs and you will create a fairly large Ellipse element. Do not be concerned that the Button will be this big. Regardless of any child element size or placement, the top level parent always maintains the Buttons footprint when being used in a scene.
8. Go back to the Timeline panel and select the eye icon next to the ButtonBase, Hood, and Arrow elements to set each to invisible. You need to have a clear image of the GlowBase element while still seeing the bounding box area of the Parent.
9. Make sure you once again have the GlowBase element selected in the Timeline palette and select the Brush Transform tool from the Toolbox. What you will see straight away is that the Gradient Brush needs to be re-transformed to make it work as a centered glow element.
10. Figure 8.12 illustrates how the brush gradient should be transformed. The main area of concern here is that the bottom of the transform arrow must be placed in the center of the active element bounding box. This will spread the gradient equally outward. Figure 8.12 has been purposely faded to show you the transform requirement; however, you should see a light blue glow in the center of the GlowBase

element and a darker blue outer area. You must ensure that the light blue area is inside the yellow bounding box.

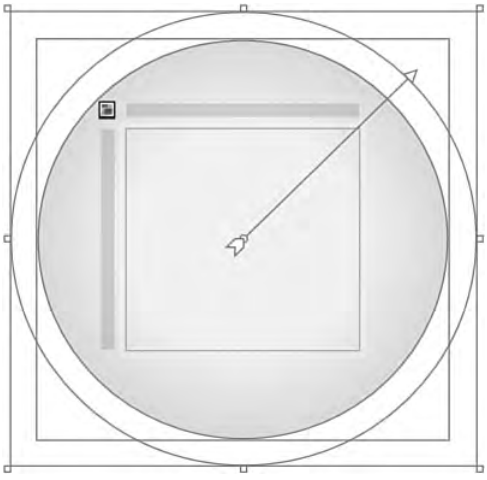

FIGURE 8.12 The Brush Transform requirement.

11. Return to the Brushes panel within the Properties palette and apply the Brush suggestions in Table 8.9.

TABLE 8.9 Sample 9 Brush Suggestion

Brush Property	Brush Type	Overall Opacity	Stroke Thickness		
Fill	Radial Gradient	100%	Set No Brush		
Gradient Stop	**Position**	**Red**	**Green**	**Blue**	**Alpha**
1	25%	6	196	227	100%
2	60%	255	255	255	0%

XAML Listing (Sample 9):

```
<RadialGradientBrush>
    <GradientStop Color="#FF06C4E3" Offset="0.25"/>
    <GradientStop Color="#00FFFFFF" Offset="0.6"/>
</RadialGradientBrush>
```

It is important that you remember to set the `Stroke` brush property to *No Brush*.

The next step is the animation requirements. If you do not already have your workspace configured for animation, select Window from the menu at the top of the screen and then select Animation Workspace or press F6.

At the bottom of the screen two palettes are displayed: an Interaction palette and a Results palette. Select the Interaction palette, which displays the Object and Timeline panel as well as the Trigger panel.

Figure 8.13 shows that just under the Objects and Timeline label there is a box with the words *(No Storyboard open)* written in it. You are going to add a new Timeline first so you have something to work with. Figure 8.13 also shows the drop-down control allowing you to create a new Timeline. See Chapter 3, "Expression Blend Panels," for a detailed description of the Objects and Timeline category as well as the related Storyboard picker control.

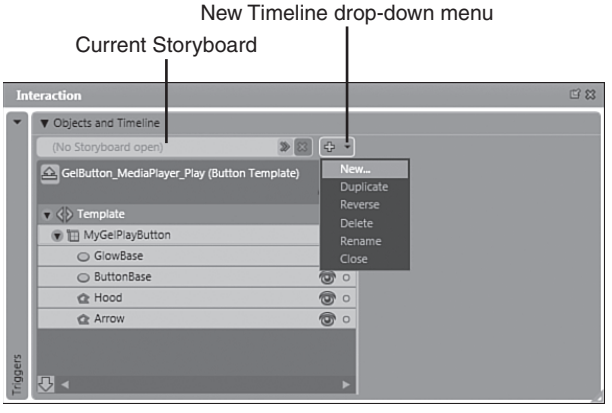

FIGURE 8.13 The areas of interest in the Objects and Timeline category.

A dialog box is displayed that asks you for the name of the new Timeline. Enter **OnMouseMove** and click the OK button. Your timeline appears to the right side of the visual element tree. You may also notice that the artboard now has a red highlighted border with a label in the top left of the artboard saying *Timeline recording is on*. This red light means that changes made now to any property of the element that is currently selected will be recorded in a KeyFrame on the timeline.

Knowing that you want to make the glow appear when the user moves the mouse over the button, you can determine that a Property Trigger, something like `ButtonBase.IsMouseOver` is in order. This property (when evaluated to true) lets you know when to start the animation. All you need to do now is define the OnMouseOver Timeline to show the glow effect.

Let's go through adding the correct trigger step-by-step.

1. Select the element in the Objects and Timeline panel for which you want to view the property values in order to start the animation. I suggest that you use the ButtonBase element.

2. Now switch over to looking at the Triggers panel. A number of predefined property events are already listed, as shown in Figure 8.14. All these predefined events are courtesy of the Make Button tool; however, for future reference, you can click on the button with the label + Property Trigger and then select from the Trigger display list the property you want to use.

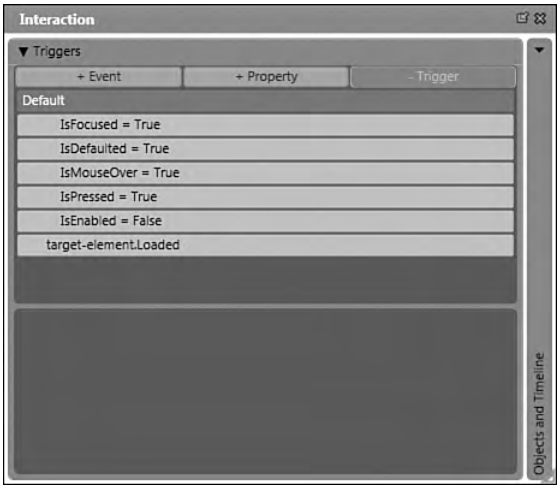

FIGURE 8.14 The Trigger list with predefined property events.

3. Search the Trigger list for one that has the label IsMouseOver = True and then click on it. You will notice the panel below called Properties when active shows a number of headings. The one named as Activated when should already have an entry with three fields, each indicating a drop-down list is available. When you select the first entry (target-element), you should see the ButtonBase element listed. Select ButtonBase from the list.
4. The next field is the property name, which is already set with the correct value, the IsMouseOver requirement. Ensure that the last field (actual property value) is set to True.
5. You are now ready to set the actions that you want to happen when the property value resolves. When activating, you should see a plus (+) sign next to the label Actions. Clicking the plus sign enables you to add a new action. Click on the sign and you add a new action with two available fields, which again indicate that a drop-down list is available. The first field represents the Timeline on which you want the action to be performed. You should see the name of the timeline you declared earlier: OnMouseMove. Also note that from this list you can create a new timeline if one is required.

   Figure 8.15 shows an example of where you are and the action options available to you. Because this is the first action planned, set the action to Begin. Notice that in the list of triggers in the top panel, there is now a lightning bolt next to the trigger. That symbol tells you there is a valid action assigned to this trigger.

FIGURE 8.15 The timeline action options.

6. You may now want to think about what is going to happen to the button when the user moves the mouse away. You can't leave the animation at its end point, because that will leave the glow visible. You must add another Trigger representing the action to take when the IsMouseOver property value is False. Click on the + Property button at the top of the Triggers panel. A new trigger is added to the list below with a red circle next to it (by default, Timeline recording is on).
7. Set the target element, the property, and the value in the Activated when area of the Properties when active panel.
8. After setting the values to ButtonBase, IsMouseOver and False, add another action for the trigger. Remember to select the timeline OnMouseMove and then set the action to Stop. This will return the timeline to its first position ready to be executed again.

Now that you have finished adding the triggers, it is time to turn your attention to the animation area of the Object and Timelines panel and really bringing the GlowBase element to life. You should still have the MyGelPlayButton element activated. If not, then select and activate it now.

It is now time to add the animation of the GlowBase element:

1. If you only see the words No Storyboard open in the Object and Timeline category, select the previously declared timeline OnMouseMove using the Storyboard picker, which will automatically turn on Timeline recording. If recording has not come on, click Ctrl+R.
2. The timeline reappears in the right side of the Objects and Timelines panel, and you can clearly see the orange line representing the time position (also called the Playhead position). Above the orange line you should see another icon with a green

plus symbol. This adds a KeyFrame to the animation path for the element you are animating. Before clicking it, make sure you have the GlowBase element selected and that the timeline playhead position (the orange line) is on the position 0. When ready, add your first KeyFrame.

3. Adding a KeyFrame does a number of things:

   a. It ensures that the object or element containing the KeyFrame will look exactly as it did when the KeyFrame was added (not withstanding any changes to the element properties).

   b. Each time your timeline is returned to the beginning, any element(s) without KeyFrames will stay at their present settings, regardless of the changes being made to elements with KeyFrames. This is called Hand-off animation and is what enables you to link (or transition) between timelines, specifically when you don't know what the previous property state of the animated object(s) will be.

Figure 8.16 shows various elements of the Objects and Timelines panel. Study it and find the Timeline Zoom control. This tool enables you to view time at different scaled ratios so you may either add KeyFrames closer together or animate for a longer period of time, as is shown in the timeline window. The callouts on this image are discussed in detail in Chapter 3.

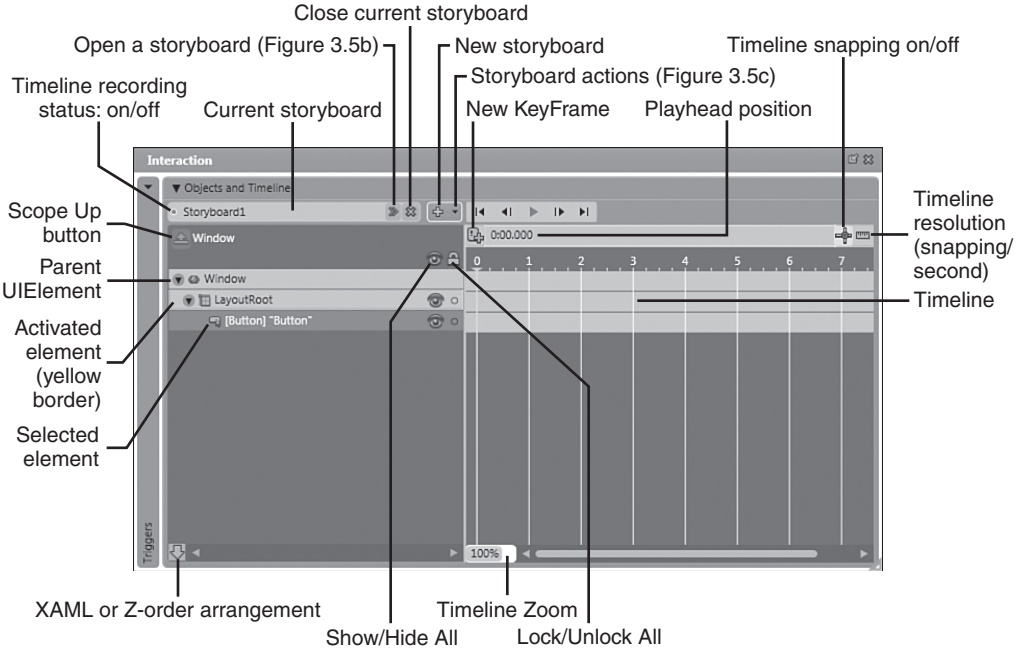

FIGURE 8.16 Various elements of the Objects and Timeline panel.

1. Apply the Timeline Zoom until you see the setting of 400% in the Zoom control and the Timeline markers show half-second intervals. You want the glow on the button to appear quite quickly. Drag the timeline position indicator (the orange line) to the position of 0.5. After making sure that the GlowBase element is still selected, add a second KeyFrame to the timeline. Next to the added KeyFrame button, notice the time indicator label, which shows the value of the Playhead position. After moving the position indicator to the required position, the Playhead position now displays 0:00.500. You can double-click this label and manually enter a position value, which is handy for those animations requiring greater levels of accuracy.
2. Now that you have added an additional KeyFrame, you need to set the properties of the element to the desired result. As this is the last KeyFrame position in this timeline, this KeyFrame is sometime referred to as the Fillframe; hence, the trigger action option to Skip to Fill. Open the Properties palette and select the Brush panel.
3. After selecting the `Fill` brush property, notice the color selectors that make up the gradient of the glow element. Move the left selector to the 45% position, making sure the solid area of the color gradient stays within the yellow bounding box of the parent element. Now move the right selector to the 95% position on the color graph. That was simple, wasn't it?
4. Last but not least, click on the Scope Up button to return to the design scene. You are now ready to test the button. Either select Project from the menu at the top of the screen, and then select Test Project from the menu that appears or depress the F5 button.

Did you notice that nothing happens if you move the mouse over the Hood or Arrow element? This is a serious issue that you must now address. Thanks to the elegant event structure employed by WPF, a very clean method exists to fix it.

Had This Problem Before with CLR Controls?

For those that have built a lot of controls before using pre-Framework 3.0 controls, especially complex controls, you may have come across this issue before. The solution was to capture the MouseOver event for every object on the control and then funnel the events to a singular method to handle it. If you have had this problem before, you are going to love this!

Before you get confused that all of a sudden I am talking about events when you just worked on Property triggers, understand the fact that an event fired somewhere in order to set a property to True or False.

A Look at RoutedEvents

RoutedEvents are very special to me for a number of reasons, but mostly because of similar issues facing large complex controls that I have constructed in the past that had similar issues. It was a situation that I dreaded every time I got a spec for a new control that essentially had layers of UIElements.

The name actually says it all. RoutedEvents move events along a Route until they reach their final destination—or until they are used by a subscribing handle.

RoutedEvents work in one of three ways:

1. Bubbling
2. Tunneling
3. Direct

Unlike CLR events (the events used by developers in previous .NET Framework versions) that only notify a specific subscriber, bubbling RoutedEvents, for example, travel along every node of the visual tree, right up to the root element. Tunneling RoutedEvents work in reverse, traveling downward through the element children until they find a subscriber. Direct RoutedEvents act almost identical to the old CLR events.

This means that you could subscribe to a MouseOver event at the root element of the tree (for an object embedded several items deep) and still get notified when the event is raised. How awesome is that? Well, it will mean more to developers....

So taking this into perspective, your `IsMouseOver` property is not True when you mouseover the ButtonBase element if the Arrow or Hood element is on top of the ButtonBase element. You need to be listening to the IsMouseOver property (set by the MouseOver event) deeper toward the root of the control itself.

I wonder: How many of you have thought far enough ahead to see another potential problem in this situation? Even though you can listen to the event on the root element, the GlowBase element is currently larger in size compared to the root element, so it effectively extends the boundary on which events will be raised.

What you need to do is make the initial size of the GlowBase element the same size or smaller than the parent root element. Then, when it is enlarged, listen for a mouse movement on it (the glow). This would indicate to you that the user has indeed moved the mouse off the button element, even though the mouse is still over the top of the GlowBase element. If you are lost, relax. It will all make sense shortly.

To make this work is fairly trivial. All you need to do is go back into the button template, change the element that the property triggers are listening on, and change the default size of the GlowBase element.

1. Select the button element in the Objects and Timelines panel. As previously described, you could right-click to bring up the context menu and select the Edit Control Parts (Template) menu item, and then the Edit Template submenu item to get into the template. But this time, direct your focus to the top of the artboard where you see a collection of tabs relating to your currently selected control.

   Because you have edited the template and/or Style of this control before, you now have access to a shortcut between the three viewable layers of the control, as shown

in Figure 8.17. You can click on any of these layers at anytime from now on instead of having to scope up and do multiple selects to access the templates of the control.

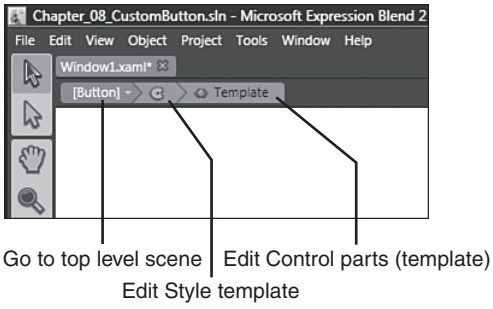

FIGURE 8.17 The improved template access workflow.

2. Select the Control parts (template) from the template level tabs.
3. Select the GlowBase element in the Timelines panel. Open the Properties palette and find the Layout panel located within it.
4. No timelines are selected yet because you are changing the default property values for the element. Once you have found the Margin properties, notice that they are still set to –50, as they were when you initially created the element. Set these all to 0.

   Now select the OnMouseMove Timeline from the Storyboard picker.
5. You should select each one of the KeyFrames added and right-click to reveal the context menu, where you can select Delete.
6. Move the Playhead position back to position 0, and after ensuring the Timeline recording is on and the GlowBase element is selected, click to add a new KeyFrame.
7. Advance the Playhead position to 0:0.500 and again add a new KeyFrame.
8. Locate the Margin properties for this element and now add the value –50 to each property.
9. Open the Brushes panel within the Properties palette and select the `Fill` brush property. Move the second Gradient Stop to a position of 90%—or until you are happy with the level of glow showing from behind the Button.
10. In the Object and Timeline category, select and activate the parent element, MyGelPlayButton.
11. Scroll down the trigger list and find the two property triggers, each with lightning bolts next to their names.
12. Select the first trigger IsMouseOver = True, and notice that the property values and action reappear in the subpanel.

13. You only need to change the value in the Activated when area, which at present shows ButtonBase as the element of choice. By selecting the drop-down list item, the parent element, MyGelPlayButton, in the list below. Click this item to make it the value.
14. Now select the GlowBase element in the Timeline panel and select the second IsMouseOver = False trigger.
15. Change the element that will activate this trigger to GlowBase.
16. Click on the Scope Up button to return to the scene and test the button.

With any luck you are now the proud owner of a Button that looks pretty sweet and has a properly working mouse move trigger that causes a glow to emanate from behind the button element.

Creating a Control Template

As you will know by now, I am a huge advocate for the use of Resource Dictionaries (RD) and templates in general. They are used throughout the application. RD's will save a substantial amount of design/development time, not to mention enable the visual continuity of an application to be honed to an extraordinary degree.

To be truthful, I must say that you have already created a control template, but it is only available as a local template because of the way you first created it. I can tell you that the Microsoft development team has worked extremely hard to make the creation, management, and use of templates as easy as possible. I, for one, am very impressed thus far.

Take a quick look at putting this button template into a Resource Dictionary and making it available to every application you write from now on. You can simply add a button element to your application and then apply this control template, or you could derive new versions and again add them to the Resource Dictionary.

Chapter 12 goes into greater detail about Resource Dictionaries and the various templates that you can create using Expression Blend; so for now, let's take the most direct root to get the required result.

1. Open the Objects and Timelines panel on the Interactive palette and select the Button element.
2. Right-click to view the context menu and look for the Edit Control Parts (Template) menu item. When you put your mouse over it, a submenu appears. Select Edit a copy.
3. The Create Style Resource dialog is displayed. From here you can create a new Resource Dictionary to store this control template.
4. Within the Define in group box at the lower half of the dialog box, click the New button.
5. The Add New Item dialog shown in Figure 8.18 is displayed. Figure 8.18 also shows that I have named this RD, GelButttons.xaml. You can ultimately name your dictionary whatever you want; but, for now, just stick with the suggestion.

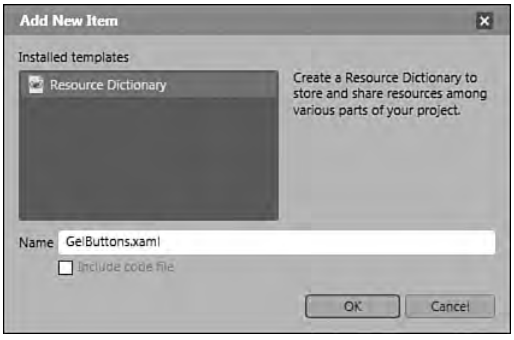

FIGURE 8.18 Creating a Resource Dictionary takes no special skills!

6. The Resource Dictionary is placed, by default, in the root directory of your project and is included in your open project resources. Back to the Add New Item dialog, notice that the Define in group box is already switched to use the RD you just created. The only thing left to do is to give the template a good name. Something like GelButtonMediaPlayerPlay perhaps? You can choose whatever you want, but do try to be descriptive.

Why Are Underscores a Bad Thing in Resource Names?

Resource names with underscores can cause confusion inside the Blend IDE. It appears that the development team did not switch off the mnemonics on the menu item template they used. When you go to view your list of resources, if you have named anything with an underscore in it, the first underscore will be missing and you will see an underscore under the first letter after where the underscore was originally created.

You can test this by naming a resource with an underscore and then applying that resource to a control such as a button. Then go back to the Edit Control Parts (template) menu item and view the Applied Resource list.

Considering that a designer could be creating the resource before sending it to someone else, any documentation may not reflect the correct name. This is a small issue, I know, but it can become annoying. I did report this to the team, but they only consider it to be an esthetic defect. For now, it is yet to be fixed.

7. Click OK and the interface will change dramatically in front of you. This is because you are now in resource-editing mode, which means you can only select and edit parts of the given template. You are all done with this at present; but later on, if you decide to enhance or change your button, you will see this same environment change. To leave and return to the scene, click on the Scope Up button within the Objects and Timelines panel or select the Go to top level scope tab, as shown in Figure 8.17.

And that, as they say, is the ball game! How simple was that? It used to be a lot more difficult than that, let me tell you.

You can now send that Resource Dictionary to other people who will be able to reproduce not only the button style, but also any functionality that you added to the template. In this case the Make Button tool did all the work. But it doesn't stop you from doing the exact same thing.

Now you are going to use the template in the next example to test that the template is valid and that the events that the button contains function properly.

Testing the Template and RoutedEvents

You are going to test the events of the Button element, but I also wanted to show you a little experiment that confirms the capability of the RoutedEvents, specifically the Tunneling and Bubbling events.

Tunneling events should always be named *Preview{event name}*; so, for example, the MouseDown event has an attached Tunneling event named PreviewMouseDown. Tunneling events are always the paired event of a standard counterpart. Their purpose is to be raised first, to enable parent elements to act on the event raised by the child before the child actually changes.

This could be handy if you wanted to validate some property after a button was clicked but before the Click event was raised and be able to handle the event if so required. The Bubbling counterpart always fires after the Tunneling event, unless an action causes the event to be cancelled.

1. Open a new project and add the XAML from Listing 8.2. Use only the markup from the "LayoutRoot" element down, but do note the Window dimension is 230 × 330.

LISTING 8.2 The Custom Button

```
<Window
... { Blend created definitions here }
    Width="230"
    Height="330" >

    <Grid x:Name="LayoutRoot">
        <Button
            x:Name="btnEventButton"
            VerticalAlignment="Top"
            Height="120"
            HorizontalAlignment="Center"
            Margin="0,10,0,0"
            Width="130"
            Content="Button"/>
        <ListBox
            Margin="0,144.8,0,0"
            x:Name="listBox_EventListener"
            IsSynchronizedWithCurrentItem="True"/>
            </Grid>
</Window>
```

2. You should end up with a Window containing two elements that looks like Figure 8.19. You are going to test the Resource Dictionary and control template with this button element. Find the previous example application and copy the Resource Dictionary file you created to the root or to a subroot folder of this project.

FIGURE 8.19 A simple test rig.

Managing Resources

It is a smart idea to create a folder somewhere in which you can store all your Resource Dictionaries. It does not take long to build up a number of these files, and it can be a nightmare trying to remember where each file is located and what each one contained.

3. Click on the Project menu item at the top of the application and then select Add Existing Item... or press Ctrl + I to add the Resource Dictionary to the project.

Why the Root Folder?

Expression Blend allows you to either add resources to the project or to link to existing files as a quasicopy-type method. If you link to an existing file resource, Blend creates a copy of the resource and adds it to the Root folder of the application.

If you try to link to a resource file that is already in the root of the application, you will receive an error dialog telling you that the file already exists in the location. For the type of file that a Resource Dictionary is, I would advise you to make a copy of the file, place it in the root or a subroot folder, but always add the file as a resource to the project rather than linking to the file.

The downside of linking is also that if the linked file is ever removed from the original path, the application will almost certainly fail with a file not found error.

4. Locate/Show the Resources palette, which is shown in Figure 8.20. You will see the layout of the application as it stands. Select the relevant scope for the resource, which in this case is the Window element. Right-click it to reveal the Link to Resource Dictionary menu item. From the submenu, select your Resource Dictionary.

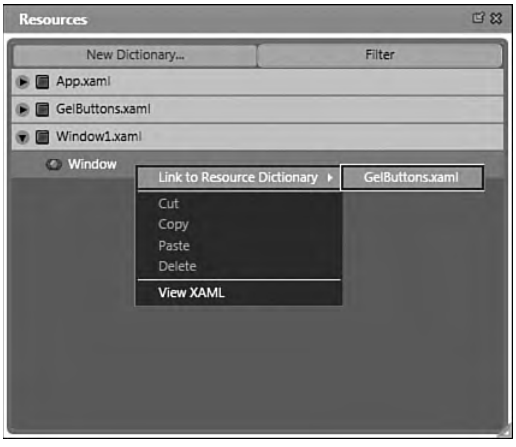

FIGURE 8.20 Linking to a Resource Dictionary.

5. Make sure you have the Selection tool chosen from the toolbox and click on the button element in the window on your artboard to select it. Open the Properties palette. In the Search box, type the word **style**. This returns only the relevant properties.
6. You are now going to apply the style template (from the previously created button) to the original standard button template that exists in this current project from the Resource Dictionary you just added. Notice that beside each property, a little square that is either filled (a white square denotes this property is applied in the xaml) or unfilled (default is assumed) is present. A click of the little box displays the Advanced property options menu for the associated property, as shown in

Figure 8.21. Figure 8.21 also shows how to select the template that was previously created and named GelButtonMediaPlayerPlay. Go ahead and select the template.

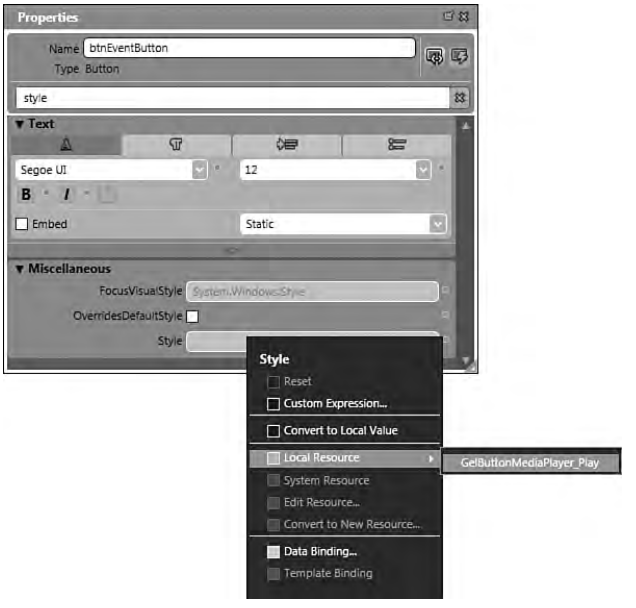

FIGURE 8.21 The Advanced menu for the Style property.

7. With any luck, your button looks similar to how it was at the end of the previous project, although you may have to tinker with the layout properties, width, height, and margins to get it back to just how you want it. After you have the button sorted out, make sure that it is still the selected item in the project and then open the Properties palette again.
8. You are about to add some events for the button element. Before you do so, save the project.
9. At the top of the palette, the name of the button element btnEventButton is displayed. To the right of this, are two icons. The one on the left, which is currently selected, allows you to see the properties for the selected element. The one on the right with the little lightning strike in the icon image represents the events that are available for this element. Select the event icon now.
10. Scroll down until you find the MouseMove event name, and then click into the input box beside it and type **BubbleMouseMove** and press Enter. Visual Studio launches with an open copy of the project and the event handler code shown in Listing 8.3 already inserted into the code editor.

LISTING 8.3 Sample Application

```
        private void BubbleMouseMove(object sender,
         System.Windows.Input.MouseEventArgs e)
        {

        }
```

1. The first thing to do here is to build the solution from the Build menu or press the F6 button. Once done, the build should succeed and you are ready to add the event handler code. Add the code in the event handler method, as shown in Listing 8.4.

LISTING 8.4 The MouseMove Event Handler

```
        private void BubbleMouseMove(object sender,
System.Windows.Input.MouseEventArgs e)
        {
            this.listBox_EventListener.BeginInit();
            this.listBox_EventListener.Items.Add("Bubble MouseMove");
            this.listBox_EventListener.EndInit();
        }
```

2. Now return to Blend. This time, scroll down the event list until you find the event named PreviewMouseMove and add the following event name: TunnelMouseMove.
3. Again, you should be able to see the event handler in Visual Studio. Enter the code as shown in Listing 8.5.

LISTING 8.5 Adding Items to the ListBox

```
        private void TunnelMouseMove(object sender,
System.Windows.Input.MouseEventArgs e)
        {
            this.listBox_EventListener.BeginInit();
            this.listBox_EventListener.Items.Add("Tunnel MouseMove");
            this.listBox_EventListener.EndInit();
        }
```

4. Select Save All from the File menu in Visual Studio, and then return to the Blend environment.

Time now to test the application by pressing F5. Move your mouse over the button. If the application is working properly, the ListBox will fill up with messages in the order in which they fire, as shown in Figure 8.22. Scroll the ListBox to the top if necessary, and you can see that the Tunnel Event message is shown first and the messages continue on all the way down the list.

FIGURE 8.22 The event messages are added as they fire.

Summary

That was a simple demonstration of the effect of RoutedEvents, not to mention the application of a template stored in a Resource Dictionary. Creating buttons and other templates is a "fun" by-product of working with Blend. The very first question almost all designers ask when I introduce them to Blend is: How do I create a button? The next thing they want to know is how to put a rollover effect on it!

I cannot stress enough that using Resource Dictionaries is the method of choice for future development teams to embrace. It is one of the foundation features of the application to enable designers and developers to work together on the same project.

Teams implement their desired processes out of necessity to fulfill a project requirement. As a result, there is no *one* right way to enable collaborative solution development using Resource Dictionaries alone. But now you know how create and modify a template within one. After a little practice you will wonder how you ever worked without them. They are that powerful of a tool in WPF projects.