Control Templates

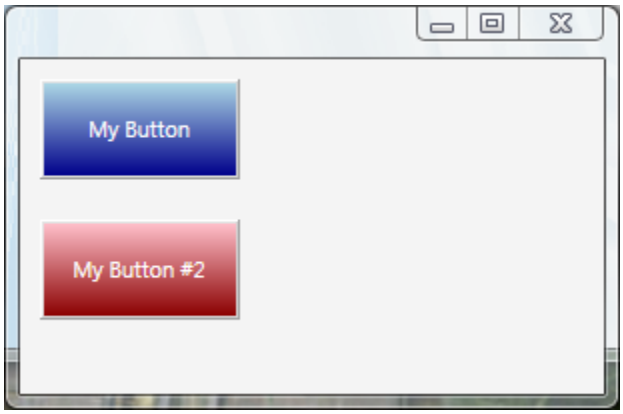
Estimated time for completion: 60 minutes

Goals:

- To understand how to change the visual appearance of controls.
- To utilize triggers to mimic the built-in behaviors of controls.

Overview:

This lab will introduce control templates which can be used to change the visual appearance of controls. You will start with a window containing two buttons that look like:



Notice that the buttons have a nice gradient background, but still look like rectangular buttons. Our goal with this exercise will be to apply a control template to the buttons to give them a more exciting look:

These buttons are made of multiple overlapping rounded rectangles. A bottom rectangle defines a "Shadow", and then the background rectangle draws the proper brush colors followed by the actual button content. Finally, a highlight rectangle is placed on the top in order to generate the "reflection" effect shown.

The new template will be applied through a Style, and then you will apply trigger effects to manage the focus and up/down state of the Button. Finally, you will change the focus rectangle so it surrounds the new shape instead of the default rectangle.

You can try to achieve the above effect yourself, or follow the below detailed instructors. You might also try using Blend (right click on the buttons and select "Edit Control Parts").

Part 1 – Creating the basic button theme

In this part, you will create a basic theme and apply it using a style.

Steps:

1. Open the **CoolButton.sln** starter project located in the **before** folder associated with the solution.
2. Run the application to see the starting picture.

1. Open the app.xaml file.
2. Add a `Style` definition into the resources.
    a. Set the `TargetType` to be a button.
    b. Add a `Setter` to set the `Template` property to a `DynamicResource` named "CoolTemplate".

```
<Style TargetType="{x:Type Button}">
   <Setter Property="Template" Value="{DynamicResource CoolTemplate}" />
</Style>
```

3. Next, add a `ControlTemplate` to the application resources.
    a. Set the `x:Key` to be "CoolTemplate" to match the style key.
    b. Set the `TargetType` to be a button.
4. Place a `Grid` as the root tag in the `ControlTemplate` – we will overlay various things in the rectangle to create our cool button.
    a. Add two rows to the grid of equal size.

```
<ControlTemplate x:Key="CoolTemplate" TargetType="{x:Type Button}">
   <Grid>
      <Grid.RowDefinitions>
         <RowDefinition />
```

```
        <RowDefinition />
      </Grid.RowDefinitions>
   </Grid>
</ControlTemplate>
```

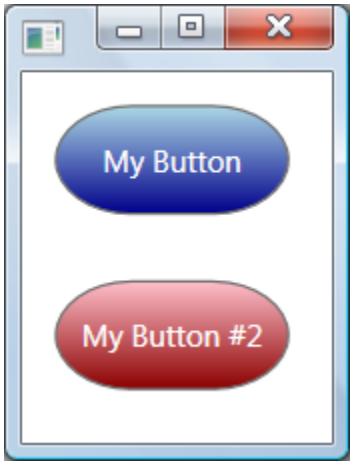
5. Add a `Rectangle` to the `Grid` that spans both rows.
    a. Give it a name of "background".
    b. Set the `Margin` to "3" to give it a little space.
    c. Set the `Stroke` to be template bound to the `BorderBrush` of the button.
    d. Set the `Fill` to be template bound to the `Background` of the button.
    e. Set the `RadiusX` and `RadiusY` to be "30".

```
<Rectangle Stroke="{TemplateBinding BorderBrush}"
      Fill="{TemplateBinding Background}"
      RadiusX="30" RadiusY="30" Margin="3"
      Grid.RowSpan="2" x:Name="background" />
```

6. Next, put a `TextBlock` into the `Grid` which also spans both rows.
    a. Set the `HorizontalAlignment` and `VerticalAlignment` to "Center"
    b. Set the `Text` property to be template bound to the `Content` of the button.

```
<TextBlock Grid.RowSpan="2"
      HorizontalAlignment="Center" VerticalAlignment="Center"
      Text="{TemplateBinding Content}" />
```

7. Run the application
    a. Notice that we now have a rounded button which utilizes the background defined on our brush. It should look something like:



Part 2 – Utilizing the ContentPresenter

In this part, you will allow the custom rendering to host content other than text.

Steps:

1. The `TextBox` isn't really the proper way to display the text in this case – it's working here because all we've defined is text on the button.
    a. Change one of the buttons to have an image as the content and run the application to see the problem.
2. Remove the `TextBox` from the `Grid` in the template and replace it with a `ContentPresenter`, it should also span both rows.
    a. Set the name to "content"
    b. Set the `Margin` to "1" to add a little space around the content.
3. In order to completely capture the button appearance you should really template bind several properties of the button to the content presenter:
    a. Bind the `HorizontalAlignment` to the button's `HorizontalContentAlignment` property.
    b. Bind the `VerticalAlignment` to the button's `VerticalContentAlignment` property.
    c. Bind the `ContentTemplate` property.
    d. Bind the `ContentTemplateSelector` property.

```
<ContentPresenter x:Name="content" Margin="1" Grid.RowSpan="2"
    HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
    VerticalAlignment="{TemplateBinding VerticalContentAlignment}"
    ContentTemplate="{TemplateBinding ContentTemplate}"
    ContentTemplateSelector="{TemplateBinding ContentTemplateSelector}" />
```

4. Run the application again – the image should now appear properly.

Part 3 – Making it look cool

In this part, you will complete the custom rendering by applying a shadow and a highlight effect to the button.

Steps:

1. Add a new `Rectangle` into the `Grid` as the first element – it should be placed above the "background" rectangle. It should span both rows of the `Grid`.
2. This will be the shadow for the button, so give it a name of "shadow".
    a. Set the `RadiusX` and `RadiusY` to "30".

b. Set the `Margin` to "1".

c. Set the `Fill` property to "Black"

d. Set the `Opacity` to ".75" to make it slightly opaque.

3. Run the application and see the effect you've created. We now have a nice border around the button.

4. Next, add a final `Rectangle` into the `Grid` after the `ContentPresenter`. This will be our highlight for the button – name it "highlight". It should only be in the first row of the Grid.

    a. Set the `RadiusX` and `RadiusY` to "40" to give it a more rounded appearance.

    b. Set the `Margin` to "9,1" to push it into the button space on the left and right.

    c. Set the `Fill` property to a `LinearGradientBrush`

        i. The `StartPoint` should be "0,0.5" and the `EndPoint` should be "1,0.5".

        ii. The `Gradient` should go from "#F0DDDDDD" at `Offset` "0" to "Transparent" at `Offset` ".75".

```
<Rectangle x:Name="highlight"
           RadiusX="40" RadiusY="40" Margin="9,1">
   <Rectangle.Fill>
      <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,0.5">
         <GradientStop Color="#F0DDDDDD" Offset="0"/>
         <GradientStop Color="Transparent" Offset=".75"/>
      </LinearGradientBrush>
   </Rectangle.Fill>
</Rectangle>
```

5. Run the application – it should now have a very nice looking "glow" to it.

Part 4 – Adding behavior

In this part, you will begin to add some behavior into the Control Template so that it behaves like a button.

Steps:

1. Run the application and attempt to click the button.

    a. Notice that it *does* respond to the `Click` event, but we've lost the visual "down" behavior. This is because we are drawing the button ourselves.

    b. It also has no mouse over behavior like our Vista buttons do.

2. Open window1.xaml and set one of the buttons to be disabled by setting the `IsEnabled` property to "false".

3. Run the application again – can you tell that it is disabled? Again, we have no information in the template to handle this case. This part will fix these problems using triggers.
4. Open the app.xaml file again and locate the Control Template you've been working on.
5. Go to the bottom of the template – just after the `</Grid>` closing tag.
6. Add a new start and end tag for `ControlTemplate.Triggers`.
7. Add a new `Trigger` into the triggers collection.
    a. Set the `Property` to "IsEnabled".
    b. Set the `Value` to "False".
    c. Add a `Setter` which sets the `Fill` property of the "background" `Rectangle` to "#FFC0C0C0" to make it a light grayish color. To do this, set the `TargetName` to "background".
8. Add another trigger into the collection
    a. Set the `Property` to "IsPressed".
    b. Set the `Value` to "True".
    c. Add a setter which sets the `Visibility` of the highlight to "Collapsed". To do this, set the `TargetName` to "highlight" so that it impacts that portion of the template.
    d. Add a setter which applies a `ScaleTransform` as a `RenderTransform` and scales the button to 95% of the current size. This is done by setting the `ScaleX` and `ScaleY` properties to ".95".
9. Add another trigger into the collection.
    a. Set the `Property` to "IsMouseOver".
    b. Set the `Value` to "True".
    c. Add a setter which changes the "shadow" `Fill` color to "Goldenrod".

```
<ControlTemplate.Triggers>
  <Trigger Property="IsEnabled" Value="False">
     <Setter Property="Fill" Value="#FFC0C0C0" TargetName="background"/>
  </Trigger>
  <Trigger Property="IsPressed" Value="True">
     <Setter Property="Visibility" Value="Collapsed" TargetName="highlight"/>
     <Setter Property="RenderTransform">
        <Setter.Value>
           <ScaleTransform ScaleX=".95" ScaleY=".95" />
        </Setter.Value>
     </Setter>
  </Trigger>
  <Trigger Property="IsMouseOver" Value="True">
```

```
    <Setter Property="Fill" Value="Goldenrod" TargetName="shadow" />
  </Trigger>
</ControlTemplate.Triggers>
```

10. Run the application and verify that all three behaviors are now working properly.

Solution

There is a full solution to this lab located in the **after** folder associated with the solution.