A WPF Image Button

Instead of a normal button with words, sometimes you want a button that is just graphical. Yes, you can put an Image control in the Content of a normal Button control, but you still have the button outline, and trying to change the style can be rather difficult. Instead I like creating a user control that simulates a button, but just accepts an image. Figure 1 shows an example of three of these custom user controls to represent minimize, maximize and close buttons for a borderless window. Notice the highlighted image button has a gray rectangle around it. You will learn how to highlight using the VisualStateManager in this blog post.

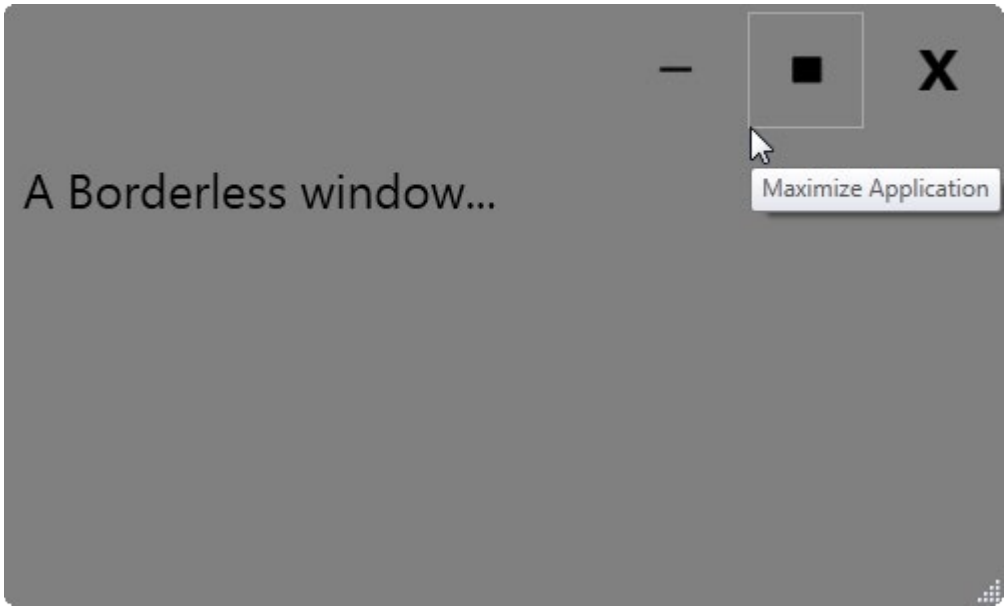

Figure 1: Creating a custom user control for things like image buttons gives you complete control over the look and feel.

I would suggest you read my previous blog post on creating a custom Button user control as that is a good primer for what I am going to expand upon in this blog post. You can find this blog post at http://weblogs.asp.net/psheriff/archive/2012/08/10/create-your-own-wpf-button-user-controls.aspx.

The User Control

The XAML for this image button user control contains just a few controls, plus a Visual State Manager. The basic outline of the user control is shown below:

```
<Border Grid.Row="0"
        Name="borMain"
        Style="{StaticResource pdsaButtonImageBorderStyle}"
        MouseEnter="borMain_MouseEnter"
        MouseLeave="borMain_MouseLeave"
        MouseLeftButtonDown="borMain_MouseLeftButtonDown">

  <VisualStateManager.VisualStateGroups>
  ... MORE XAML HERE ...
  </VisualStateManager.VisualStateGroups>

  <Image Style="{StaticResource pdsaButtonImageImageStyle}"
         Visibility="{Binding Path=Visibility}"
         Source="{Binding Path=ImageUri}"
         ToolTip="{Binding Path=ToolTip}" />
</Border>
```

There is a Border control named **borMain** and a single Image control in this user control. That is all that is needed to display the buttons shown in Figure 1. The definition for this user control is in a DLL named PDSA.WPF. The Style definitions for both the Border and the Image controls are contained in a resource dictionary names PDSAButtonStyles.xaml. Using a resource dictionary allows you to create a few different resource dictionaries, each with a different theme for the buttons.

The Visual State Manager

To display the highlight around the button as your mouse moves over the control, you will need to add a Visual State Manager group. Two different states are needed; MouseEnter and MouseLeave. In the MouseEnter you create a ColorAnimation to modify the BorderBrush color of the Border control. You specify the color to animate as “DarkGray”. You set the duration to less than a second. The TargetName of this storyboard is the name of the Border control “borMain” and since we are specifying a single color, you need to set the TargetProperty to “BorderBrush.Color”. You do not need any storyboard for the MouseLeave state. Leaving this VisualState empty tells the Visual State Manager to put everything back the way it was before the MouseEnter event.

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup Name="MouseStates">
    <VisualState Name="MouseEnter">
      <Storyboard>
        <ColorAnimation
            To="DarkGray"
            Duration="0:0:00.1"
            Storyboard.TargetName="borMain"
            Storyboard.TargetProperty="BorderBrush.Color" />
      </Storyboard>
    </VisualState>
    <VisualState Name="MouseLeave" />
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Writing the Mouse Events

To trigger the Visual State Manager to run its storyboard in response to the specified event, you need to respond to the MouseEnter event on the Border control. In the code behind for this event call the GoToElementState() method of the VisualStateManager class exposed by the user control. To this method you will pass in the target element (“borMain”) and the state (“MouseEnter”). The VisualStateManager will then run the storyboard contained within the defined state in the XAML.

```
private void borMain_MouseEnter(object sender,
 MouseEventArgs e)
{
  VisualStateManager.GoToElementState(borMain,
    "MouseEnter", true);
}
```

You also need to respond to the MouseLeave event. In this event you call the VisualStateManager as well, but specify “MouseLeave” as the state to go to.

```
private void borMain_MouseLeave(object sender,
 MouseEventArgs e)
{
  VisualStateManager.GoToElementState(borMain,
    "MouseLeave", true);
}
```

The Resource Dictionary

Below is the definition of the PDSAButtonStyles.xaml resource dictionary file contained in the PDSA.WPF DLL. This dictionary can be used as the default look and feel for any image button control you add to a window.

```
<ResourceDictionary  ... >
  <!-- ************************ -->
  <!-- ** Image Button Styles ** -->
  <!-- ************************ -->
  <!-- Image/Text Button Border -->
  <Style TargetType="Border"
         x:Key="pdsaButtonImageBorderStyle">
    <Setter Property="Margin"
            Value="4" />
    <Setter Property="Padding"
            Value="2" />
    <Setter Property="BorderBrush"
            Value="Transparent" />
    <Setter Property="BorderThickness"
            Value="1" />
    <Setter Property="VerticalAlignment"
            Value="Top" />
    <Setter Property="HorizontalAlignment"
            Value="Left" />
    <Setter Property="Background"
            Value="Transparent" />
  </Style>
  <!-- Image Button -->
  <Style TargetType="Image"
         x:Key="pdsaButtonImageImageStyle">
    <Setter Property="Width"
            Value="40" />
    <Setter Property="Margin"
            Value="6" />
    <Setter Property="VerticalAlignment"
            Value="Top" />
    <Setter Property="HorizontalAlignment"
            Value="Left" />
  </Style>
</ResourceDictionary>
```

Using the Button Control

Once you make a reference to the PDSA.WPF DLL from your WPF application you will see the “PDSAucButtonImage” control appear in your Toolbox. Drag and drop the button onto a Window or User Control in your application. I have not referenced the PDSAButtonStyles.xaml file within the control itself so you do need to add a reference to this resource dictionary somewhere in your application such as in the App.xaml.

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary
        Source="/PDSA.WPF;component/PDSAButtonStyles.xaml" />
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

This will give your buttons a default look and feel unless you override that dictionary on a specific Window or User Control or on an individual button. After you have given a global style to your application and you drag your image button onto a window, the following will appear in your XAML window.

```
<my:PDSAucButtonImage ... />
```

There will be some other attributes set on the above XAML, but you simply need to set the x:Name, the ToolTip and ImageUri properties. You will also want to respond to the Click event procedure in order to associate an action with clicking on this button. In the sample code you download for this blog post you will find the declaration of the Minimize button to be the following:

```
<my:PDSAucButtonImage
       x:Name="btnMinimize"
       Click="btnMinimize_Click"
       ToolTip="Minimize Application"
       ImageUri="/PDSA.WPF;component/Images/Minus.png" />
```

The ImageUri property is a dependency property in the PDSAucButtonImage user control. The x:Name and the ToolTip we get for free. You have to create the Click event procedure yourself. This is also created in the PDSAucButtonImage user control as follows:

```
private void borMain_MouseLeftButtonDown(object sender,
 MouseButtonEventArgs e)
{
  RaiseClick(e);
}

public delegate void ClickEventHandler(object sender,
  RoutedEventArgs e);
public event ClickEventHandler Click;

protected void RaiseClick(RoutedEventArgs e)
{
  if (null != Click)
    Click(this, e);
}
```

Since a Border control does not have a Click event you will create one by using the MouseLeftButtonDown on the border to fire an event you create called “Click”.

Summary

Creating your own image button control can be done in a variety of ways. In this blog post I showed you how to create a custom user control and simulate a button using a Border and Image control. With just a little bit of code to respond to the MouseLeftButtonDown event on the border you can raise your own Click event. Dependency properties, such as ImageUri, allow you to set attributes on your custom user control. Feel free to expand on this button by adding additional dependency properties, change the resource dictionary, and even the animation to make this button look and act like you want.

NOTE: You can download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select “Tips & Tricks”, then select “A WPF Image Button” from the drop down list.