A WPF Image/Text Button

Some of our customers are asking us to give them a Windows 8 look and feel for their applications. This includes things like buttons, tiles, application bars, and other features. In this blog post I will describe how to create a button that looks similar to those you will find in a Windows 8 application bar.

In Figure 1 you can see two different kinds of buttons. In the top row is a WPF button where the content of the button includes a Border, an Image and a TextBlock. In the bottom row are four individual user controls that have a Windows 8 style look and feel. The “Edit” button in Figure 1 has the mouse hovering over it so you can see how it looks when the user is about to click on it.

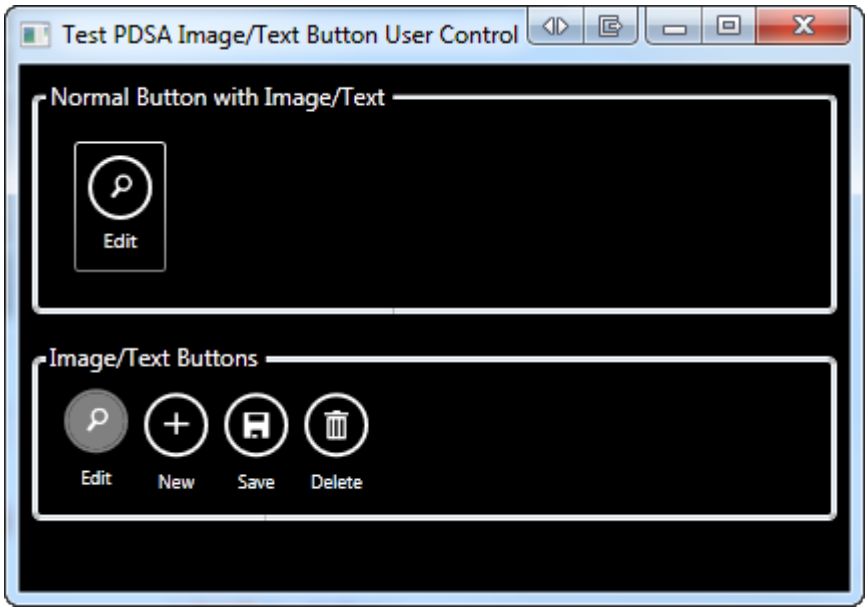

Figure 1: It is best to create a custom user control to get a more polished look and feel for a button control.

If you read my previous blog post on creating a custom Button user control, you will find this blog post very similar.

There are many ways to create custom buttons and there are advantages and disadvantages to each way. The purpose of this blog post is to present one method that is easily understood by almost any XAML programmer, and hopefully to those new to XAML as well. User controls have been around since the Visual Basic 4.0 days. Most developers understand the value of

using user controls. With XAML user controls you can put these controls into a WPF User Control Library or a Silverlight Class Library and reference those DLLs from any WPF or Silverlight application. This gives you great reusability.

The User Control

The XAML for this kind of Windows 8 application bar style-button is a little more complicated than the simple buttons shown in my previous blog posts. However, the basics are shown below:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <Border Grid.Row="0"
          Name="borMain"
          Style="{StaticResource
                   pdsaButtonImageTextBorderStyle}"
          MouseEnter="borMain_MouseEnter"
          MouseLeave="borMain_MouseLeave"
          MouseLeftButtonDown="borMain_MouseLeftButtonDown"
          ToolTip="{Binding Path=ToolTip}">

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

    <Image Source="{Binding Path=ImageUri}"
           Style="{StaticResource
                   pdsaButtonImageTextImageStyle}" />
  </Border>
  <TextBlock Grid.Row="1"
             Name="tbText"
             Style="{StaticResource
                      pdsaButtonImageTextTextBlockStyle}"
             Text="{Binding Path=Text}" />
</Grid>
```

There is a Grid, a Border, an Image and a TextBlock control all combined to form the buttons shown in row 2 of Figure 1. The above XAML is fairly easy to understand as this is just combining standard controls into a format that gives you the look required for your button. The Border, the Image and the TextBlock have a named style applied to them. The definition for this user control is in a DLL named PDSA.WPF. A default resource dictionary is included in the DLL where this user control is located to give you a default

look and feel; however, you can make a copy of this resource dictionary and change the look to meet your needs.

Adding the Visual State Manager

In the original blog post on creating a button user control I wrote code to change a button's state using C#. In this blog post I have replaced most of this code with XAML in the form of the Visual State Manager. A Visual State Manager (VSM) is a container for a storyboard in which you specify a series of actions to perform on different attributes of your controls. To give the user feedback when they hover over a button you use the Visual State Manager to change attributes of controls.

In the following VSM there are two visual states; MouseEnter and MouseLeave. The MouseLeave is empty which tells the VSM to return all properties changed during the MouseEnter back to their original values. In the MouseEnter state is where you modify three properties of the Border control. First you modify the BorderBrush color to the color specified in the style named "pdsaButtonImageTextBorderHoverColor". You also modify the Background color of the border to the color specified in the style name "pdsaButtonImageTextBackHoverColor". Finally, the Margin property of the Border control is modified slightly in order to make the button appear to move up.

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup Name="MouseStates">
    <VisualState Name="MouseEnter">
      <Storyboard>
        <ColorAnimation
           To="{StaticResource
                pdsaButtonImageTextBorderHoverColor}"
           Duration="0:0:00.1"
           Storyboard.TargetName="borMain"
           Storyboard.TargetProperty="BorderBrush.Color" />
        <ColorAnimation
           To="{StaticResource
                pdsaButtonImageTextBackHoverColor}"
           Duration="0:0:00.1"
           Storyboard.TargetName="borMain"
           Storyboard.TargetProperty="Background.Color" />
        <ThicknessAnimation
           To="{StaticResource
                pdsaButtonImageTextHoverThickness}"
           Duration="0:0:00.1"
           Storyboard.TargetName="borMain"
           Storyboard.TargetProperty="Margin" />
      </Storyboard>
    </VisualState>
    <VisualState Name="MouseLeave" />
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

The XAML below shows the default styles used in the Visual State Manager. These styles come from the PDSAButtonStyles.xaml resource dictionary contained in the PDSA.WPF dll.

```
<!-- Border color while hovering over button -->
<Color x:Key="pdsaButtonImageTextBorderHoverColor">
  Gray
</Color>
<!-- Background color while hovering over button -->
<Color x:Key="pdsaButtonImageTextBackHoverColor">
  Gray
</Color>
<!-- Thickness while hovering over button -->
<Thickness x:Key="pdsaButtonImageTextHoverThickness">
  4,2,4,4
</Thickness>
```

Writing the Mouse Events

To trigger the Visual State Manager to run its storyboard in response to the specified event, you 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);
}
```

Write code in the MouseLeave event and call the VisualStateManager’s GoToElementState method and specify “MouseLeave” as the state to go to.

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

The Default Resource Dictionary

Below is the definition of the resource dictionary file contained in the PDSA.WPF DLL. This dictionary is used as the default look and feel for any Image/Text Button control you add to a window or user control.

```
<ResourceDictionary  ...>
  <!-- ***************************** -->
  <!-- ** Image/Text Button Styles ** -->
  <!-- ***************************** -->
  <!-- Image/Text Button Border -->
  <Style TargetType="Border"
         x:Key="pdsaButtonImageTextBorderStyle">
    <Setter Property="Margin"
            Value="4" />
    <Setter Property="BorderBrush"
            Value="White" />
    <Setter Property="BorderThickness"
            Value="2" />
    <Setter Property="HorizontalAlignment"
            Value="Center" />
    <Setter Property="CornerRadius"
            Value="50" />
    <Setter Property="Width"
            Value="32" />
    <Setter Property="Height"
            Value="32" />
    <Setter Property="Background"
            Value="Transparent" />
  </Style>
  <!-- Image/Text Button Image -->
  <Style TargetType="Image"
         x:Key="pdsaButtonImageTextImageStyle">
    <Setter Property="Margin"
            Value="0" />
  </Style>
  <!-- Image/Text Button TextBlock -->
  <Style TargetType="TextBlock"
         x:Key="pdsaButtonImageTextTextBlockStyle">
    <Setter Property="Margin"
            Value="2" />
    <Setter Property="Foreground"
            Value="White" />
    <Setter Property="HorizontalAlignment"
            Value="Center" />
    <Setter Property="FontSize"
            Value="9" />
  </Style>
  <!-- Border color while hovering over button -->
  <Color x:Key="pdsaButtonImageTextBorderHoverColor">
    Gray
  </Color>
  <!-- Background color while hovering over button -->
  <Color x:Key="pdsaButtonImageTextBackHoverColor">
    Gray
  </Color>
  <!-- Thickness while hovering over button -->
  <Thickness x:Key="pdsaButtonImageTextHoverThickness">
    4,2,4,4
  </Thickness>
</ResourceDictionary>
```

Feel free to modify this resource dictionary, or copy it and modify your new copy in order to give another look and feel to these buttons. Keep the “x:Key” name the same, other than that, you can modify any other attribute.

Using the Button Control

Once you make a reference to the PDSA.WPF DLL from your WPF application you will see the “PDSAucButtonImageText” 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 add a reference to this resource dictionary in your Application Resources section defined in App.xaml.

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

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

```
<my:PDSAucButtonImageText ... />
```

There will be some other attributes set on the above XAML, but you just need to set the x:Name, the Text, 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 Edit button to be the following:

```
<my:PDSAucButtonImageText
     Name="btnEdit"
     ImageUri="/PDSA.WPF;component/Images/Edit_White.png"
     Text="Edit"
     Click="btnEdit_Click" />
```

The Text and ImageUri properties are dependency properties in the PDSAucButtonImageText user control. The x:Name and the ToolTip we get for free. Since a Border control does not have a Click event you will create one by using the MouseLeftButtonDown on the border to fire your custom “Click” event. Code the “Click” event in the PDSAucButtonImageText user control using the code shown below:

```
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);
}
```

Summary

This blog post built upon the previous posts where I explained how to build a button user control. The user control presented in this post adds both text and an image and adds a little XAML to the storyboard in the Visual State Manager. With the appropriate styles applied you can get a Windows 8 look and feel for these application bar buttons. Feel free to modify the styles to take on any look you want for your buttons.

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/Text Button” from the drop down list.