Create your own WPF Button User Controls

In Figure 1 you can see examples of the standard WPF Button controls. You can add a drop shadow and you can change the color, but you can't change much else without creating a whole new control template. For example, you are unable to modify the BorderBrush or the BorderThickness properties of the Button control. Additionally you might want to use some other animation than the default, which again requires you to change the control template.

Sometimes all you want to do is to just have some simple buttons where you can modify the border brush and the thickness and have different color options via styles. I have found that instead of working with the whole control template thing, just creating a User Control is sometimes much easier.

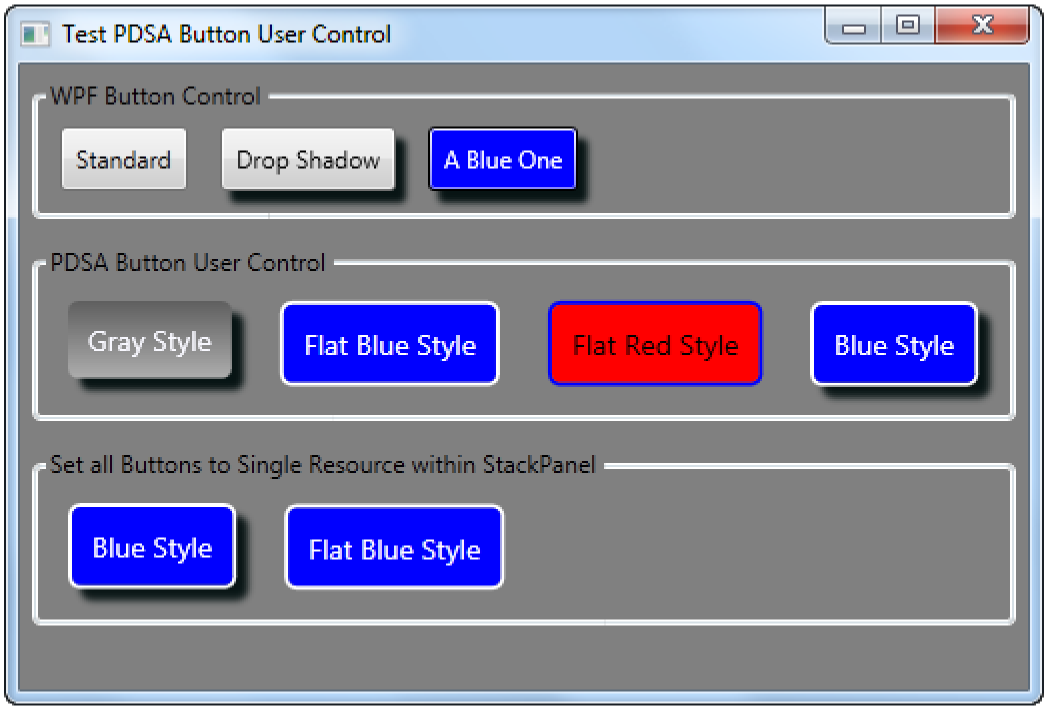

Figure 1: The normal WPF Buttons cannot be styled very well.

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. By using resource dictionaries and appropriate use of styles you can make your user controls very customizable as shown in Figure 1. The 2nd row of buttons you see are the same button user control with just different styles applied.

The User Control

The XAML for your Button user control is actually very simple. You use a single Border control and a TextBlock control within that Border as shown in Listing 1.

```
<Border Name="hoverBorder"
        Style="{DynamicResource hoverBorder}"
        MouseDown="hoverBorder_MouseDown"
        MouseUp="hoverBorder_MouseUp"
        MouseEnter="hoverBorder_MouseEnter"
        MouseLeave="hoverBorder_MouseLeave"
        MouseLeftButtonUp="hoverBorder_MouseLeftButtonUp">
  <TextBlock Text="{Binding Path=Text}"
             Name="tbText"
             Style="{DynamicResource hoverTextBlock}" />
</Border>
```

Listing 1: The XAML for a Button user control is just a Border and TextBlock

The definition for this user control is in a DLL named PDSA.WPF. Notice that there are Style definitions for both the Border and the TextBlock. We used styles contained in a resource dictionary as opposed to setting up dependency properties for each individual attribute that we might want to set. This allows us to create a few different resource dictionaries, each with a different theme for the buttons. You can see the three different themes we created in Figure 1. The **Gray Style** button uses a resource dictionary that is contained in the PDSA.WPF DLL. The other two styles are in the main project and can be referenced from your App.xaml or from within the Window/User Control where you need the button.

Changing Colors in Response to Mouse Events

When a user moves over a button or presses a button you should give some visual feedback to that user. You can do this with animation using the Visual State Manager or Event Triggers in WPF. To keep things simple for this blog post, I simply respond to the various mouse events and change the Background property to a different color. When the mouse is pressed, I also change the Foreground color of the Text in the TextBlock control. The code for each of the mouse events is shown below.

```
private void pdsaButtonBorderStyle_MouseEnter(
 object sender, MouseEventArgs e)
{
  pdsaButtonBorderStyle.Background =
   (Brush)this.FindResource("pdsaButtonOverStyle");
}

private void pdsaButtonBorderStyle_MouseLeave(
 object sender, MouseEventArgs e)
{
  pdsaButtonBorderStyle.Background =
   (Brush)this.FindResource("pdsaButtonNormalStyle");
}

private void pdsaButtonBorderStyle_MouseDown(
 object sender, MouseButtonEventArgs e)
{
  // Save old Foreground Brush
  _TextBrush = tbText.Foreground;

  pdsaButtonBorderStyle.Background =
   (Brush)this.FindResource("pdsaButtonPressedStyle");
  tbText.Foreground =
   (SolidColorBrush)this.FindResource(
     "pdsaButtonTextBlockStylePressed");
}

private void pdsaButtonBorderStyle_MouseUp(
 object sender, MouseButtonEventArgs e)
{
  RestoreNormal();
}

private void RestoreNormal()
{
  pdsaButtonBorderStyle.Background =
   (Brush)this.FindResource("pdsaButtonNormalStyle");

  tbText.Foreground = _TextBrush;
}
```

Notice that in the code above that you use the FindResource() method instead of accessing the this.Resources[] collection. That is because you want to be able to set the resource dictionary at any level, not just on this user control. FindResource() will search upward through your UI tree looking for a resource that match the names you see that start with “pdsa”.

You will need a Click event that you can raise when the user clicks on the button. Here is the definition for that Click event.

```
private void pdsaButtonBorderStyle_MouseLeftButtonUp(
 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);

  RestoreNormal();
}
```

The Default Resource Dictionary

Below is the definition of the resource dictionary file contained in the PDSA.WPF DLL. This dictionary can be used as the default look and feel for any button control you add to a window. I have included two additional resource dictionaries in the main project as examples of how you can change the resources to give your buttons a different look. You need to keep the **x:Key** names the same, but you can change any of the attributes of color or thickness that you want. You can even change from Gradient colors to a SolidColorBrush as you can see I did when you look at the different resource dictionaries.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/
                           winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <!-- Hover TextBlock Style -->
  <Style TargetType="TextBlock"
         x:Key="pdsaButtonTextBlockStyle">
    <Setter Property="Foreground"
            Value="GhostWhite" />
    <Setter Property="Margin"
            Value="10" />
    <Setter Property="FontSize"
            Value="14" />
  </Style>
  <!-- Hover Brush for Pressed Hover Button Text Block -->
  <SolidColorBrush x:Key="pdsaButtonTextBlockStylePressed"
                   Color="Black" />
  <!-- Hover Border Thickness -->
  <Thickness x:Key="pdsaButtonBorderStyleThickness"
             Bottom="0"
             Left="0"
             Right="0"
             Top="0" />
  <!-- Hover Border Brush -->
  <SolidColorBrush x:Key="pdsaButtonBorderBrushStyle"
                   Color="Black" />
  <!-- Style for when hovering over button -->
  <RadialGradientBrush x:Key="pdsaButtonOverStyle">
    <GradientStop Color="#FF5F5F5F"
                  Offset="0" />
    <GradientStop Color="#FFADADAD"
                  Offset="1" />
  </RadialGradientBrush>
  <!-- Style for when button is pressed -->
  <LinearGradientBrush EndPoint="1,0.5"
                       StartPoint="0,0.5"
                       x:Key="pdsaButtonPressedStyle">
    <GradientStop Color="#FF5F5F5F"
                  Offset="0" />
    <GradientStop Color="#FFADADAD"
                  Offset="1" />
  </LinearGradientBrush>
  <!-- Style for normal button -->
  <LinearGradientBrush EndPoint="0.5,1"
                       StartPoint="0.5,0"
                       x:Key="pdsaButtonNormalStyle">
    <GradientStop Color="#FF5F5F5F"
                  Offset="0" />
    <GradientStop Color="#FFADADAD"
                  Offset="1" />
  </LinearGradientBrush>
  <!-- Overall Style for Hover Button -->
  <Style TargetType="Border"
         x:Key="pdsaButtonBorderStyle">
    <Setter Property="Margin"
            Value="6" />
    <Setter Property="CornerRadius"
```

```
            Value="5" />
    <Setter Property="Background"
            Value="{StaticResource pdsaButtonNormalStyle}" />
    <Setter Property="BorderBrush"
            Value="{StaticResource
                    pdsaButtonBorderBrushStyle}" />
    <Setter Property="BorderThickness"
            Value="{StaticResource
                    pdsaButtonBorderStyleThickness}" />
  </Style>
</ResourceDictionary>
```

You can look at the other two Resource Dictionary files when you download the sample.

Using the Button Control

Once you make a reference to the PDSA.WPF DLL from your WPF application you will see the “PDSAucButton” 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/User Control or on an individual button. The “Gray Style” button shown in Figure 1 is what the default button looks like after setting the above code in your App.xaml and then dragging a button onto a Window.

If you wish to give a specific style to just a single button you can override the default by using the code below:

```
<my:PDSAucButton HorizontalAlignment="Left"
                  Margin="6"
                  Text="Flat Blue Style"
                  x:Name="btn2"
                  Click="btn2_Click"
                  VerticalAlignment="Top">
  <my:PDSAucButton.Resources>
    <ResourceDictionary Source="FlatBlueButtonStyles.xaml" />
  </my:PDSAucButton.Resources>
</my:PDSAucButton>
```

If you want to override a series of buttons within one specific StackPanel, or within a specific Window or User Control, you simply set the Resources section for that control as shown below:

```
<StackPanel Orientation="Horizontal"
            VerticalAlignment="Top">
  <!-- Set Styles for All Buttons.
       This overrides the App.xaml styles -->
  <StackPanel.Resources>
    <ResourceDictionary Source="FlatBlueButtonStyles.xaml" />
  </StackPanel.Resources>
  <my:PDSAucButton HorizontalAlignment="Left"
                   Margin="6"
                   Text="Blue Style"
                   x:Name="btn11"
                   Effect="{StaticResource mainDropShadow}"
                   VerticalAlignment="Top" />
  <!-- Can assign a custom set of styles -->
  <my:PDSAucButton HorizontalAlignment="Left"
                   Margin="6"
                   Text="Flat Blue Style"
                   x:Name="btn12"
                   VerticalAlignment="Top" />
</StackPanel>
```

In the above sample, all buttons within this StackPanel control are styled using the resource dictionary FlatBlueButtonStyles.xaml.

Summary

Creating your own button control can be done in a variety of ways. You can create a new Button control template, apply styles to change just a few things about a Button, create your own Custom control that inherits from Button, or build a User Control as in this blog post. Which method you choose depends a little on how comfortable you are with each method, and how much control you want over the final details of your new Button. The approach outlined in this blog post is simple and easy to understand and almost anyone can use this technique right away. Feel free to add your own animation to this control or maybe add dependency properties to control the BorderBrush, BorderThickness or any other properties 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 “Create your own WPF Buttons” from the drop down list.