Pop, Snap, and Slide

In this lab, we will code in bursts. Get the starting source code and follow along. I will commit changes after every burst, so you can get caught up if you get lost.

## Gitting the Source Code

If you have a git client installed, then clone the repository. Create a working branch. For example:

cd c:\projects

git clone git://github.com/dallasxaml/PopSnapAndSlide.git

cd PopSnapAndSlide

git checkout –b take1

After each successful burst, commit your changes. If you ever get lost, commit that branch, go back to master, and create a new one:

git add –A

git commit –m "I missed that."

git checkout master

git pull

git checkout –b take2

## Downloading the Source Code

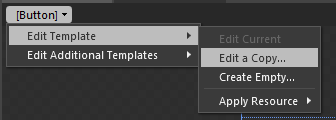
If you don’t have git installed, then go to the following URL and click on the “ZIP” button:

https://github.com/dallasxaml/PopSnapAndSlide

Unzip to your project folder. If you ever get lost, go back to the web page and download the zip again.

# Make Buttons Pop

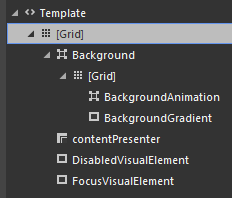
The visual state manager was born in Silverlight. Create a Silverlight project. No need to host the project in a new web site. Switch to Blend. Add a button to your main page. Then select the button, and use the template menu to edit a copy of the button template. Accept the default name (ButtonStyle1) and location (this document).



Now switch to the States tab and click through the visual states. To see the transition animation, click the transitions preview button.



There are several objects in the tree that are used just to provide visual cues.



These are always there. The visual states just change their color and opacity. Change the color of the BackgroundAnimation border and see how that affects the button’s visual behavior.

# Differences in WPF

Now do the same thing in WPF. What is different?

When you create a copy of the template in WPF, you can see that it’s not using the visual state manager. It’s using triggers. Change the background color in the MouseOver and Pressed states. The visual states override the triggers.

<Trigger Property="IsMouseOver" Value="true">

<Setter Property="Background" TargetName="border" Value="{StaticResource Button.MouseOver.Background}"/>

<Setter Property="BorderBrush" TargetName="border" Value="{StaticResource Button.MouseOver.Border}"/>

</Trigger>

Delete the triggers and then add a transition time to the states. Does it work as expected? The base style uses a brush resource for the background.

The WPF template does not have all of the objects that the Silverlight one had. So the visual states are more limited. If you want to add more objects, you could get buttons like Sliverlight. If you like that sort of thing.

# Make Navigation Slide

The visual state manager can help the user navigate through their data. Put all of the controls on the page, and then use the visual state manager to control which ones are visible.

Drop a couple of buttons onto the WPF application. Then add a couple of rectangles. The rectangles represent views that you want to navigate between. Color one of them blue and the other red.

Overlay the two rectangles on the right, and stack the buttons on the left. Set their opacity to 0 and their visual transform x to 150. Then create a couple of visual states called Red and Blue. In each visual state, set the opacity to 100 and the transform x to 0.

Now wire up the buttons so that they switch to the selected visual state.

    private void Red\_Click(object sender, System.Windows.RoutedEventArgs e)

    {

        VisualStateManager.GoToElementState(LayoutRoot, "Red", true);

    }

    private void Blue\_Click(object sender, System.Windows.RoutedEventArgs e)

    {

        VisualStateManager.GoToElementState(LayoutRoot, "Blue", true);

    }

There is a difference between GoToState and GoToElementState. GoToState is for controls, and GoToElementState is for applications.

# Make Windows Snap

Create a new Windows Store application using the Split template. Run the application and snap it. Notice how you can navigate through two screens when it is filled, but three screens when it is snapped.

The template injects a class called LayoutAwarePage into your code. This class actually has four responsibilities:

* Default view model
* Navigation keys
* Back stack
* Visual state switching

We’ll focus just on the last one. It changes the visual state based on the application view state:

* FullScreenLandscape
* Filled
* Snapped
* FullScreenPortrait

Take a look at the **ItemsPage.xaml**. It defines overrides for the Snapped and FullScreenPortrait states. They both change the style of the back button. FullScreenPortrait changes the padding inside the item grid view. Snapped goes further and hides the grid view to expose a list view instead.

Now take a look at **SplitPage.xaml**. It defines different visual states:

* FullScreenLandscapeOrWide
* FilledOrNarrow
* FullScreenPortrait\_Detail
* Snapped\_Detail

It produces these new states by overriding DetermineVisualState. This allows the page to simulate navigation between the full list and the selected item while in snapped mode. It overrides GoBack so that an item is deselected when the user hits the back button, rather than allowing navigation to occur.

# Composition and Visual States

Visual states can only affect things that you can name. This means that they cannot reach into child controls and affect their states. You have a couple of workarounds for this.

One option is to **subscribe to visual state changes in the child control**. Create a user control. Add it to the ItemsPage. Create a text block in the user control and set its text to “Unknown”. Then create two visual states called “FullScreenLandscape” and “Filled”. Set the text to “Full” in the “FullScreenLandscape” state, and “Filled” in the “Filled” state.

If you run the code now, you won’t get the visual state change. It doesn’t happen automatically. Instead, add this code to find the layout aware page into which this control is dropped.

    private LayoutAwarePage GetPage()

    {

        var ancestor = this.Parent as FrameworkElement;

        while (ancestor != null)

        {

            var container = ancestor as LayoutAwarePage;

            if (container != null)

                return container;

            ancestor = ancestor.Parent as FrameworkElement;

        }

        return null;

    }

Handle the Loaded and Unloaded event and call the StartLayoutUpdates and StopLayoutUpdates methods. Then give it a try.

Now drop this same control into the SplitPage and see if it works.

The second option is to **show and hide child controls** based on the visual state. This is what ItemsPage does to replace the grid view items with list view items. Without this, each item in the list or grid would have to subscribe to state changes with the parent.