Foundations
Using Templates to Customize WPF Controls
Charles Petzold

Code download available at: Foundations2007_01.exe (158KB)

Contents

With the release of Windows Vista™ and the Microsoft® .NET Framework 3.0 come a host of new technologies for developers to learn about, discuss, and use. New tools, libraries, and paradigms will change how you build managed apps, offering powerful possibilities. We're introducing this new monthly column to cover the underlying technologies you will use for developing your applications. Industry experts you are already familiar with will take turns delving into the Windows® Presentation Foundation, Windows Communication Foundation, and Windows Workflow Foundation. Let's get started.

Traditionally, customizing an existing control in Windows has been a four-step process. It begins with inspiration. Then comes the research and exploration. This inevitably leads to weeping and gnashing of teeth. And finally it concludes with a complete rewrite. Controls are often resistant to customization due to deeply inaccessible code that links the visuals of the control with its functionality. This code is vital to the control so it must be either completely accepted or completely bypassed and replaced.

The pain of control customization has evidently been felt by the developers of the Windows Presentation Foundation, available as part of the .NET Framework 3.0. They've come up with an exciting and powerful solution known as the "template."

The Windows Presentation Foundation template is at the same time so simple and so powerful that it actually took me a while to wrap my head around the concept. I quickly understood Windows Presentation Foundation styles (which are often confused with templates), but templates needed time to sink in.

Every predefined control in the Windows Presentation Foundation that has a visual appearance also has a template that entirely defines that appearance. This template is an object of type ControlTemplate that is set to the Template property defined by the Control class.

When you use a Windows Presentation Foundation control in your application, you can replace that default template with one of your own design. You retain the basic functionality of the control-including all the keyboard and mouse handling-but you can give it an entirely different look. This is what is meant when Windows Presentation Foundation controls are described (rather inelegantly) as "lookless." A control has a default look, but this look is not intrinsic to the inner workings of the control.

Writing templates in code is rather awkward. It is much easier to use Extensible Application Markup Language (XAML). Because templates can be represented entirely in XAML, visual design tools

will be available to help. If you're thinking about programming a custom control that will work like an existing Windows Presentation Foundation control but with a different look, stop now! You might very well be able to do it with a template.

The downloadable source code consists of seven standalone XAML files that I'll discuss throughout this column. No C# code was compiled during the making of this column! If you have the .NET Framework 3.0 SDK installed, you can edit the files using XAMLPad or XAML Cruncher, a similar program from my book, *Applications = Code + Markup*: *A Guide to the Microsoft Windows Presentation Foundation*.

The Windows Presentation Foundation supports other types of templates for displaying control content, but in this column I'll only discuss objects of type ControlTemplate.

Dumping the Defaults

Every predefined control in the Windows Presentation Foundation that has a visual appearance has a default template. If you are interested in writing custom templates, study these defaults.

The DumpControlTemplate from Chapter 25 of my book displays a control's default Template property in convenient XAML format. (You can download the source code.) But if you'd rather write a template dumper yourself, here's the code that performs the crucial step:

```
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;
settings.IndentChars = new string(' ', 4);
settings.NewLineOnAttributes = true;
StringBuilder strbuild = new StringBuilder();
XmlWriter xmlwrite = XmlWriter.Create(strbuild, settings);
XamlWriter.Save(ctrl.Template, xmlwrite);
```

This code assumes that ctrl is an instance of a class that derives from Control and that ctrl has been rendered on the screen. (It is my experience that the Template property will be null otherwise.) The XamlWriter.Save call raises an exception if the Template property is null, as it will be for some controls that don't have a visual appearance. At the end of this code, calling ToString on strbuild provides a complete XAML document that contains the template.

The XAML in this default template may be a little wordier than the XAML you might write. For example, you might write this:

```
<Trigger Property="IsEnabled" Value="False">
```

But in the XAML file generated by XamlWriter.Save, you'll see markup that looks like this:

```
<Trigger Property="UIElement.IsEnabled">
    <Trigger.Value>
        <s:Boolean>False</s:Boolean>
    </Trigger.Value>
```

The s prefix is defined with an xmlns namespace declaration for the .NET System namespace.

The Visual Tree

Let's begin by looking at a template for a fairly simple but non-trivial control: the CheckBox. Say you want something that looks a bit more dramatic than the standard CheckBox. Perhaps you want

the user's selection to be displayed as a big green checkmark or a big red X that appears over the entire content of the CheckBox. The file BigCheckCheckBox.xaml shows a template for such a CheckBox.

In a XAML file, a template is an element of type ControlTemplate. In a small Windows Presentation Foundation program, ControlTemplate elements are generally defined in a Resources section in the root element of a XAML file. For larger applications, or when defining templates that are shared among applications, ControlTemplate elements are in their own XAML files with root elements of ResourceDictionary.

In either case, the ControlTemplate element generally has three parts. It begins with an optional Resources section that can define styles or brushes used by the template. (The template in BigCheckCheckBox.xaml has no Resources section.) The template then provides the definition of the template's visual tree. This tree describes the desired appearance of the control as a layout of elements and perhaps other controls. The template concludes with a Triggers section, which indicates how elements of the visual tree change in response to changes in the control's properties. Figure 1 and Figure 2 show most of the ControlTemplate elements for the custom CheckBox. The code in the first figure shows the visual tree and the second figure continues the ControlTemplate element with the Triggers section.

The excerpt from BigCheckBox.xaml shown in Figure 1 illustrates the use of the ControlTemplate start tag and the visual tree. Because this template is a resource, it must contain an x:Key attribute with its resource name. You'll notice the TargetType attribute is used-this is not strictly required but it makes the rest of the template simpler by eliminating the need to preface every property with a class name.

The top-level element of this visual tree is a Border, an element that can play parent to a single child. In this sample, three properties of this Border are assigned with the TemplateBinding markup extension. TemplateBinding is used to bind properties of elements in the visual tree to properties of the control.

The TemplateBinding expressions are more interesting in the ContentPresenter element, which formats content for buttons and other controls that derive from ContentControl. It is ContentPresenter that allows just about anything to be displayed inside a Button or CheckBox. Notice that a TemplateBinding is used to set the Margin property of the ContentPresenter to the Padding property of the CheckBox control being templated. Margin is extra space outside an element; padding is space inside a control not occupied by its content, so you can see the rationale behind this binding.

The ContentPresenter appears inside a single-cell Grid panel along with another Border. Grid cells are often used to host multiple children that must appear in layers. The second Border therefore appears on top of the ContentPresenter. This Border is given a background that consists of a 50 percent opaque brush that contains a red X mark. Notice that the Path element that renders the red X mark is assigned a name of path. This name is referenced in the Triggers section of the template.

Template Triggers

The last section of a ControlTemplate is generally dedicated to Trigger elements. These change properties of the elements that make up the visual tree based on changes in properties of the control. Figure 2 shows a large portion of the Triggers section for the custom CheckBox template.

By default, the IsChecked property of the CheckBox is false and the CheckBox displays a red X mark. The first Trigger element indicates that when IsChecked becomes true, the Data property and

the Stroke property of the element named path should be set to display a green check mark. The next Trigger element accommodates a null value of IsChecked (this can occur when the CheckBox is set for tri-state operation) by displaying a blue question mark. The last Trigger element changes the Foreground property of the control to a gray brush when the IsEnabled property is false. (The BigCheckCheckBox.xaml file has an additional Trigger element that displays a dotted line around the control's content when the control has input focus.)

The Triggers section of a template can be quite extensive. A Trigger element for the IsMouseOver property is often included so the control reacts when the mouse passes over the control.

You can then reference the template in an element that creates a CheckBox, like so:

```
<CheckBox Template="{StaticResource templateBigCheck}" ...
```

Figure 3 shows three CheckBox controls displayed by the BigCheckCheckBox.xaml file. Two have text content and the third (which has its IsThreeState property set to true and its IsChecked property set to null) contains a bitmap.

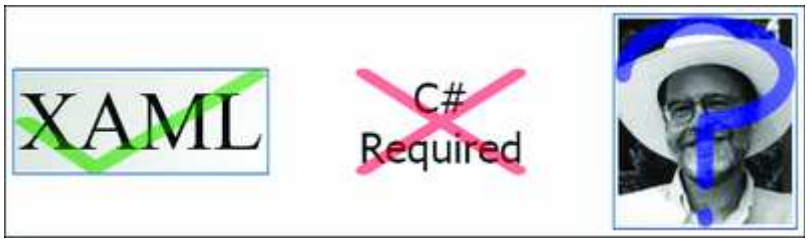

Figure 3 CheckBox Controls with a Custom Template (Click the image for a larger view)

Making Progress

You might be skeptical of how well the concept of templates can be adapted to more complex controls. After all, some controls have moving parts and more involved interaction with the user. To dissuade you from such skepticism, I'll spend the rest of this column looking at templates for the three controls that derive from RangeBase. These are ProgressBar, ScrollBar, and Slider.

To work properly, templates for more complex controls require certain types of elements with specific names. For example, a visual tree for a ProgressBar must have two elements of type FrameworkElement (or that derive from FrameworkElement) named PART_Track and PART_Indicator. These two elements are known as "named parts" of the template. The names are shown in the SDK documentation of the ProgressBar class as TemplatePart attributes. If your template does not include two elements with these names, the control won't work correctly.

If the ProgressBar is displayed in its default horizontal orientation, then the internal logic of the control sets the Width property of the element named PART_Indicator to a fraction of the ActualWidth property of the element named PART_Track. That fraction is based on the Minimum, Maximum, and Value properties of the ProgressBar. If the orientation of the ProgressBar is vertical, then the Height and ActualHeight properties of the two elements are used instead.

The ProgressBar control actually has two default templates for the two orientations. (This is true of ScrollBar and Slider, as well.) If you want your new ProgressBar to support both orientations, you should write two separate templates and select them in the Triggers section of a Style element that you also define for the ProgressBar.

Figure 4 is an excerpt from the BareBonesProgressBar.xaml file that shows a complete ProgressBar element with the ControlTemplate object as a property of that element. The file also includes a ScrollBar used to test the ProgressBar through a binding. As you manipulate the ScrollBar, the blue rectangle (the PART_Indicator element) varies in width from zero to the width of the red rectangle (the PART_Track element). Note that even though BareBonesProgressBar.xaml gives the PART_Track rectangle an explicit width, you should actually avoid explicit dimensions whenever possible.

It's more common for the indicator element to be inside the track element, and you'll probably consider using a Border element for the track. But beware: if you also give that Border a non-zero BorderThickness, then that thickness will be part of the Border element's total width, and the width inside the border will be slightly less. If you want to use a Border with a non-zero border thickness for display purposes, make the track another Border with a zero border thickness inside that first border, and put something else inside that second Border for the indicator. (And if you're using a Border element without making use of its border or background properties, consider using a Decorator, which is the borderless, background-less class from which Border descends.)

Figure 5 shows a rather extensive ControlTemplate object from the ThermometerProgressBar.xaml file. This template has a Resources section but no Triggers section. Although the visual tree makes use of some explicit coordinates for borders and corners, the overall dimensions of the ProgressBar are not defined. That responsibility is left to any markup that defines a ProgressBar using a template like this, for example:

```
<ProgressBar
       Template="{StaticResource
       templateThermometer}"
       Orientation="Vertical" Minimum="0"
       Maximum="100"
       Width="50" Height="350" ...
```

The resulting ProgressBar is shown in Figure 6. Notice that the Orientation property of this ProgressBar must be set to Vertical-otherwise it won't work properly. You can either remember to set the Orientation whenever you use this template, or you can define a Style for the ProgressBar that sets the Orientation and also references the template. (I'll show an example of this technique in a moment.)

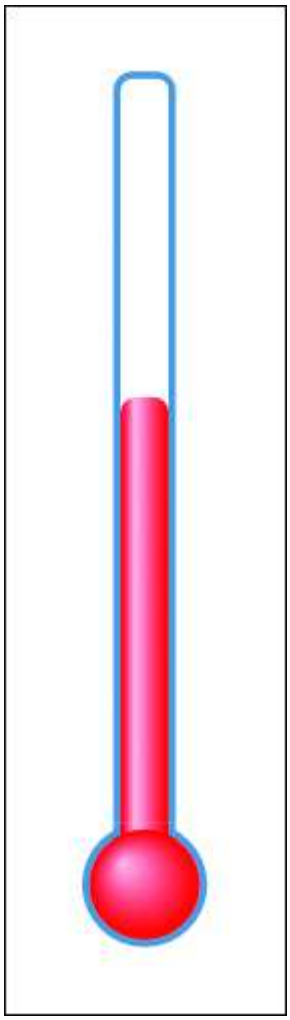

Figure 6

Among the downloadable code for this column is also SpeedometerProgressBar.xaml, which produces the more radical ProgressBar shown in Figure 7.

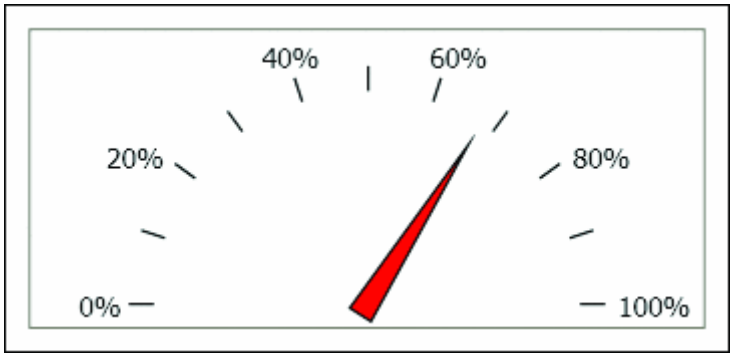

Figure 7 Speedometer ProgressBar

This one is a bit tricky. The template includes two rectangles that are invisible-they have no height or fill color or stroke color, as shown here:

```
<Rectangle Name="PART_Track" Width="180" />
<Rectangle Name="PART_Indicator" />
```

The width of the PART_Track element is set to the number of degrees in a semicircle. The Polygon

element for the red pointer is subjected to a RotateTransform whose Angle property is bound to the ActualWidth of PART_Indicator, like so:

```
<RotateTransform
   Angle="{Binding ElementName=PART_Indicator,
                   Path=ActualWidth}" />
```

Anatomy of a ScrollBar

Both ProgressBar and ScrollBar derive from the abstract RangeBase class. Just about all that RangeBase provides are Value properties, the ValueChanged event, and definitions of Minimum, Maximum, SmallChange, and LargeChange. (ProgressBar uses just the Minimum, Maximum, and Value properties.)

ScrollBar is more complex than ProgressBar, reacting to user input in several ways. This behavior is distributed among five child controls in the standard ScrollBar. The moveable Thumb is a control named Thumb. Directly on either side of the thumb are two RepeatButton controls that typically implement Page Up and Page Down commands. (RepeatButton is similar to the regular Button except that it responds to a sustained mouse press with multiple Click events.) These two RepeatButton controls change size whenever the Value property of the ScrollBar changes, and sometimes one or the other shrinks down to nothing. On the extreme ends of the ScrollBar are two more RepeatButton controls of constant size labeled with arrows that typically implement Line Up and Line Down commands.

The Thumb and inner RepeatButton controls are children of an element named Track, which is responsible for all of their interaction, movement, and size changes. Most of the core functionality and messy logic of the ScrollBar is handled by Track.

When you define a ControlTemplate for a ScrollBar, the visual tree needs to contain a Track element named PART_Track. That's the only required named part for a ScrollBar template.

Track is not a control. It derives instead from FrameworkElement, and it does not have a Template property because the Template property is defined by Control. Although you can't provide a template for Track, you can provide templates for the three controls that make up the Track element.

Track defines three properties that correspond to its three children: Thumb (of type Thumb), DecreaseRepeatButton (of type RepeatButton), and IncreaseRepeatButton (also of type RepeatButton). By default, these three properties are null, which means the template's visual tree should contain explicit definitions of these three controls. If you also want to provide the two buttons on the ends of the ScrollBar, the visual tree should include elements for these two buttons. You can give these five child controls their own templates, or you can use the existing templates and just assign some properties.

Button Commands

The four RepeatButton controls in the standard ScrollBar template must be able to communicate with the inner logic of ScrollBar. This is accomplished using predefined RoutedCommand objects that the ScrollBar class defines as static read-only fields. ScrollBar defines no fewer than 17 of these static read-only fields that correspond to various operations the ScrollBar can perform. Some of these commands are for use with the ScrollBar context menu; some are for a ScrollBar used as part of a ScrollViewer control. Still others, specifically those that begin with the words Line and Page, are for templates.

Figure 8 shows a ScrollBar template from the NoFrillsScrollBar.xaml file. The Track element and the two RepeatButton controls at the ends are organized in a Grid panel. Three properties of the Track element are set to two more RepeatButton controls and a Thumb.

This is a fully functional ScrollBar, but as you can see in Figure 9, the four RepeatButton controls look like regular buttons. The two buttons on the ends were originally quite tiny, so I increased their sizes by setting their content to pointing-hands characters taken from the Wingdings font. Other than that, none of the five controls have any properties set except for the Command property and the Grid.Column attached property.

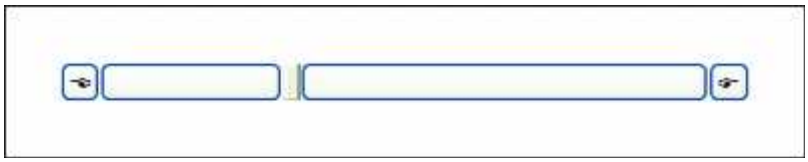

Figure 9 ScrollBar Based on Figure 8 (Click the image for a larger view)

The template in Figure 8 is intended for a horizontal ScrollBar. A vertical orientation would require three rows rather than three columns in the Grid, and it would reference RoutedCommand fields from the ScrollBar class that include the words Up and Down rather than Left and Right.

The default ScrollBar template uses the ScrollChrome class from the Microsoft.Windows.Themes namespace for rendering some of its controls. You'll also notice that the template seems to be much longer than it needs to be because many of the properties of the controls in the visual tree are defined through styles rather than attributes. For example, where you could use something like this:

```
<RepeatButton IsFocusable="False" ...
```

instead you see something like this:

```
<RepeatButton ... >
    <RepeatButton.Style>
        <Style TargetType="RepeatButton">
            ...
            <Setter Property="UIElement.IsFocusable">
                <Setter.Value>
                    <s:Boolean>False</s:Boolean>
                </Setter.Value>
            </Setter>
            ...
```

I suspect the default templates contain markup like this because these properties were originally defined in Style elements in the Resources section of the templates. But it's important to know that when you write your own templates, you can set properties directly in the elements.

The default template sets both the IsFocusable and IsTabStop properties of each RepeatButton to false-when you experiment with NoFrillsScrollBar.xaml, you'll see why. Click the leftmost button. Now press the Tab key a couple times. You'll see input focus shift from RepeatButton to RepeatButton, which is certainly not what's supposed to happen.

Because there are four RepeatButton controls in the template, it's probably easiest to use a Style element for setting uniform properties on all of them. You can add the following markup to NoFrillsScrollBar.xaml right after the ControlTemplate start tag to solve the focus problem:

```
<ControlTemplate.Resources>
    <Style TargetType="{x:Type RepeatButton}">
        <Setter Property="Focusable" Value="False" />
        <Setter Property="IsTabStop" Value="False" />
    </Style>
</ControlTemplate.Resources>
```

You can also use the Resources section of ControlTemplate to define templates for the child controls that make up the visual tree of the ScrollBar. I did this in the SpringLoadedScrollBar.xaml file. Figure 10 shows the visual tree section of that template.

Notice that the four RepeatButton elements in the template set the Foreground property to the Foreground property of the ScrollBar itself, and set a Template defined earlier in the Resources section. These other templates (not shown in Figure 10) use the Foreground property of the RepeatButton to color their visuals. Figure 11 shows two ScrollBar controls that use this template but with different Foreground settings.

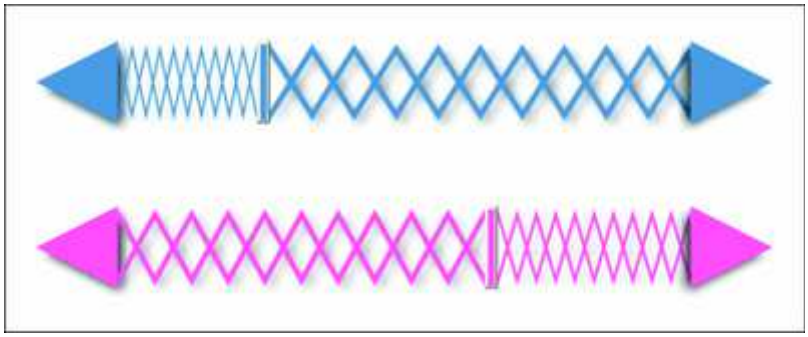

Figure 11 Using Different Foreground Settings (Click the image for a larger view)

I added DropShadowBitmapEffect elements to the four buttons. In fact, I liked the effect so much that I decided to add a Triggers section that sets an additional drop shadow when the mouse passes over the button.

The template for the spring-loaded ScrollBar assumes a horizontal orientation, and it looks best when the height is 50 device-independent units. These settings are not part of the template itself, and there's really no place for them in the template. You might want to consider putting them in a Style definition, as I'll demonstrate in a moment.

The size of the Thumb in the ScrollBar is based on the ScrollBar ViewportSize property and is commonly used to show the proportion of the document currently in view. I couldn't find a way to override the Thumb size without setting ViewportSize. If you want a larger Thumb, you should write a template for Slider instead-you can actually write a template that makes a Slider look pretty much just like a ScrollBar.

Creating a 3D Slider

Unlike ScrollBar, the default Slider control does not have two small change buttons placed on either end. But you can add such controls to a Slider template. The Slider defines six RoutedCommand objects as static get-only properties, and includes DecreaseSmall, IncreaseSmall, DecreaseLarge, and IncreaseLarge. The Slider can also display optional tick marks on one or both sides.

Perhaps the first thing you'll notice when you look at the default Slider template is that it's over a thousand lines long-that's over three times the length of the default ScrollBar template. Part of the

reason for this is that the Slider template doesn't rely on a class from Microsoft.Windows.Themes. Instead, everything is built right into the template.

This template defines different thumb shapes when tick marks are displayed on just one side of the Slider, and the template contains many gradient brushes used in coloring this thumb. If you want to include optional TickBar elements in your Slider template, they should have their Visibility properties set to Visibility.Collapsed. The Triggers section of the ControlTemplate should contain Trigger elements that set the Visibility properties to Visibility.Visible based on the TickPlacement property of Slider. For this reason, it's necessary to assign names to the TickBar elements, but they don't have to be special names.

For my custom vertical Slider, I wanted my Thumb to resemble a lever on a sound-mixing board. My goal was for this lever to look like an elevated piece of plastic with a three-dimensional appearance. To emulate three-dimensional perspective, it had to change shape as it moved up and down. You should see the bottom of the lever when it's pushed to the top of the Slider and the top of the lever when it's positioned at the bottom of the slider.

Windows Presentation Foundation has some 3D graphics capabilities that are ideal for a task of this nature. The Slider3D.xaml file contains the template. Figure 12 shows five of these 3D Slider controls set to different positions. Notice how the lever changes shape depending on its position.

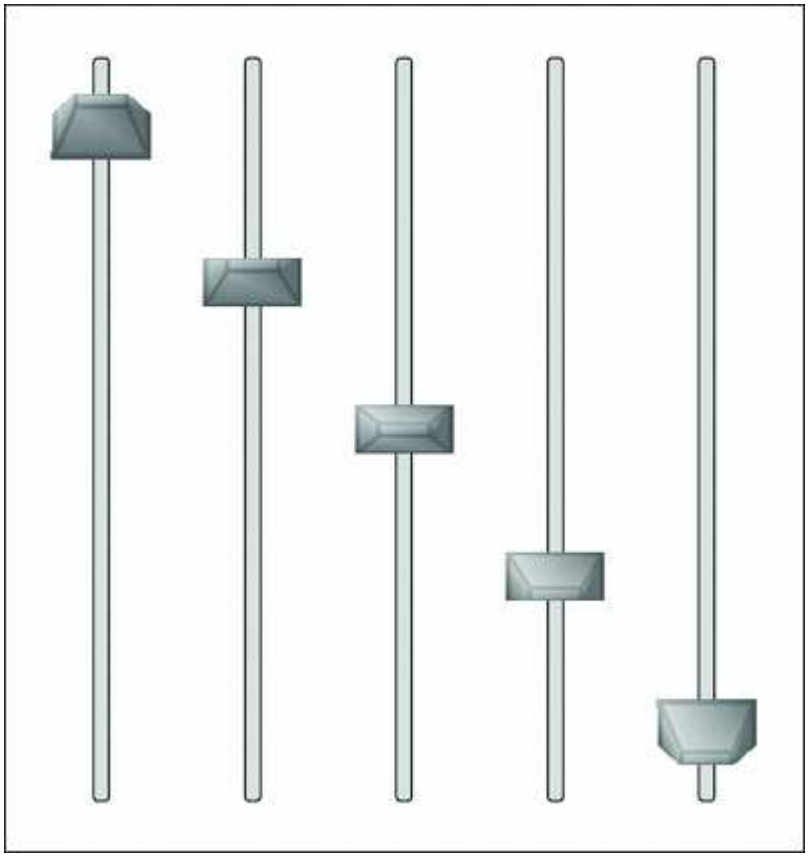

Figure 12 3D Slider Controls (Click the image for a larger view)

A vertical Slider generally begins with a three-column Grid panel; the two outer columns are for the two TickBar elements. In the default vertical Slider template, the center Grid contains a Border element that provides the image of the groove, and a Track element containing the Thumb and two RepeatButton controls. I took the same approach in my template and defined the two RepeatButton controls as just Border elements with Transparent backgrounds. They don't obscure the groove but

they still respond to the mouse.

I decided that the Thumb should be 50 pixels high (it's already 50 pixels wide by virtue of the ColumnDefinition in the Grid). And the visual tree contains another Border with a transparent Background. At this point, the Slider is fully functional, though you can't see the thumb. The child of this transparent Border is a Viewport3D element, and here's where it gets interesting.

The way I look at it, there are basically two parts to 3D graphics programming: the grunt work and the fiddling. The grunt work mostly involves defining MeshGeometry3D elements, which are the visible objects defined entirely as a collection of connected triangles in 3D coordinate space. For the Thumb, I defined an object that is a rectangular pyramid with the top cut off:

```
<MeshGeometry3D
    Positions="-2 -1 0, -1 -0.25 4, -2 1 0, -1 0.25 4,
               2 -1 0,  1 -0.25 4,  2 1 0,  1 0.25 4"

    TriangleIndices="0 1 2, 1 3 2, 0 2 4, 2 6 4,
                     0 4 1, 1 4 5, 1 5 7, 1 7 3,
                     4 6 5, 7 5 6, 2 3 6, 3 7 6"

    TextureCoordinates="0 1, 0.2 0.6, 0 0, 0.2 0.4,
                        1 1, 0.8 0.6, 1 0, 0.8 0.4" />
```

The Positions attribute consists of three-dimensional coordinates for each of the eight vertices of the figure. The bottom of the figure (where Z equals zero) extends from X values of -2 to +2 and Y values of -1 to +1. The top rectangle (where Z equals 4) extends from X values of -1 to +1 and Y values of -0.25 to +0.25.

Each triplet in the TriangleIndices attribute defines a surface of the figure in terms of indices into the Positions array. For example, the first TriangleIndices triplet is 0, 1, and 2-this refers to the first, second, and third vertices of the Positions collection. The TextureCoordinates collection contains a two-dimensional point for every three-dimensional vertex in the figure. These points correspond to points in a brush based on a GeometryDrawing used to cover the outside of the three-dimensional object:

```
<GeometryDrawing Brush="LightGray"
    Geometry="F 1 M 0 0 L 1 0 L 1 1 L 0 1 Z
              M 0.2 0.4
              L 0.8 0.4 0.8 0.6 0.2 0.6 Z
              M 0 0 L 0.2 0.4
              M 1 0 L 0.8 0.4
              M 1 1 L 0.8 0.6
              M 0 1 L 0.2 0.6">
    <GeometryDrawing.Pen>
        <Pen Brush="DarkGray"
             Thickness=".05" />
    </GeometryDrawing.Pen>
</GeometryDrawing>
```

This brush simply covers the whole object with a light-gray color and gives a dark-gray accent to the edges.

When I refer to fiddling with 3D graphics programming, I mean working with the lighting and camera. With lighting, you generally want a combination of DirectionalLight and AmbientLight. The former by itself is too stark; the latter tends to wash out everything. The camera in the template is initially set up to look straight down the Z axis at the lever, but it also has a rotation transform to view the lever from above and below. I couldn't get this to work with a TemplateBinding to the Value property of the Slider, so I used a regular binding instead:

```
<AxisAngleRotation3D Axis="1 0 0"
    Angle="{Binding RelativeSource={RelativeSource
        AncestorType={x:Type Slider}}, Path=Value}" />
```

Because the Value property determines the rotation angle of the camera, I needed to impose Minimum and Maximum properties of -25 and 25, respectively. This allows the camera to swing a total of 50 degrees-any more and the lever would get clipped. Less than 50 degrees is OK, but then the 3D effect is muted. The Slider3D.xaml file uses a Style to set these two properties as well as the Orientation property and the Template property, as shown here:

```
<Style x:Key="styleSlider3D"
       TargetType="Slider">
    <Setter Property="Orientation" Value="Vertical" />
    <Setter Property="Minimum" Value="-25" />
    <Setter Property="Maximum" Value="25" />
    <Setter Property="Template"
            Value="{StaticResource templateSlider3D}" />
</Style>
```

A Slider using this template actually refers to this style rather than the template.

Remember Moderation

As you've seen, a little hand-coded XAML is all you need to make standard controls look quite different. If you're like most programmers I know, you'll start horsing around and writing templates that violate all sense of logic and good taste.

Before you go crazy with templates, keep in mind that one of the great values of Windows is its consistent user interface. When things look familiar, users know what to expect. So keep your creative instincts thoroughly tempered and use templates in moderation. (The *MSDN®Magazine* editors made me grudgingly add this section.)

Send your questions and comments to mmnet30@microsoft.com.

Charles Petzold is a Contributing Editor to MSDN Magazine and the author of Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation (Microsoft Press, 2006). His Web site is www.charlespetzold.com.
