**An Animated Custom Panel base class for WPF and Silverlight**

[MCS UK Solution Development](http://blogs.msdn.com/257387/ProfileUrlRedirect.ashx)

12 Jan 2011 3:06 AM

* Comments [1](http://blogs.msdn.com/b/mcsuksoldev/archive/2011/01/12/an-animated-custom-panel-base-class-for-wpf-and-silverlight.aspx#comments#comments)

The full code base for this posting can be located [here](http://code.msdn.microsoft.com/Animated-Custom-Panel-base-89a7399c).

If you have used XAML at all, you will have used some of the built in layout panels such as Grid and StackPanel and if you are a developer you have probably developed a custom panel that implements your own layout logic. In case you haven’t, here’s a brief explanation. WPF and Silverlight use a two-pass scheme, where first all elements are measured by calling the panel’s MeasureOverride method. In this method you are supposed to call Measure on all your children and ask them how much space they would like. In the second pass, your ArrangeOverride method is called and the panel then actually performs the layout of its children, usually taking note of the child’s DesiredSize property and then calling Arrange passing the size and position of the children. There is really not much more to a custom panel than the ArrangeOverride and MeasureOverride methods but obviously the custom layout logic can be as complicated as you like.

Way back in 2006, Martin Grayson and I published [an article](http://www.codeproject.com/KB/WPF/Panels.aspx) showing an animated custom Fish-Eye panel that implemented a layout not dissimilar to Apple’s hyperbar on OSX that grows as you mouse over each element. Obviously we’ve done lots of custom panels since then, but one recurring theme is that really a panel should not modify its children. In the Fish-Eye example, the panel arranged all the children at (0,0) and applied animated transforms to move the children into place. If the child wanted to set its RenderTransform itself, it couldn’t, as the panel would overwrite it. Also creating lots of animations in code seemed a bit wasteful.

Another technique was used by a colleague, Stuart Richardson, which is not to use Animations at all, but hook CompositeTarget.Render and call InvalidateArrange on each frame redraw, then recalculate the positions of all the children and arrange them in the right place. This way you are not modifying the children at all, just positioning them. After we had successfully used this technique for many panels, Martin suggested a base class that would handle all the CompositeTarget.Render logic that panels requiring animation could inherit from. So I took the idea and wrote the AnimatedPanel class that this article is about.

Firstly let’s look at a simple panel. If you look at the code for AnimatedWrapPanel, you see it just wraps its children, but inherits from AnimatedPanel. If you run the sample, you’ll notice that although the panel wraps its children, if you resize the window and cause the panel to readjust, the transitions between the states are animated. This is the AnimatedPanel base class doing this.

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Windows.Controls;  
using System.Windows;  
  
namespace AnimatedPanels  
{  
 public class AnimatedWrapPanel : AnimatedPanel  
 {  
 protected override Size MeasureOverride(Size availableSize)  
 {  
 foreach (UIElement child in this.Children)  
 {  
 child.Measure(availableSize);  
 }  
  
 return base.MeasureOverride(availableSize);  
 }  
  
 protected override Size ArrangeOverride(Size finalSize)  
 {  
 double x = 0;  
 double y = 0;  
 double maxHeight = 0;  
 foreach (UIElement child in this.Children)  
 {  
 if (x + child.DesiredSize.Width >= finalSize.Width)  
 {  
 x = 0;  
 y += maxHeight;  
 maxHeight = 0;  
 }  
  
 base.AnimatedArrange(child, new Rect(x, y, child.DesiredSize.Width, child.DesiredSize.Height));  
 maxHeight = Math.Max(child.DesiredSize.Height, maxHeight);  
 x += child.DesiredSize.Width;  
   
 }  
  
 return base.ArrangeOverride(finalSize);  
 }  
 }  
}

Notice that the panel calls AnimatedArrange instead of the normal child.Arrange method that a normal panel would. It would have been nice to let the panel call child.Arrange so that the only code change to make it animate would be to change its inheritance, but I couldn’t find a way to reliably get the arranged size and position in the AnimatedPanel base class, so provided an explicit call for the panel to call instead of Arrange’ing its children.

Now the AnimatedPanel base class:

using System;  
using System.Collections.Generic;  
using System.Windows;  
using System.Windows.Controls;  
using System.Windows.Media;  
using System.Diagnostics;  
  
namespace AnimatedPanels  
{  
 public class AnimatedPanel : Panel  
 {  
 Dictionary<UIElement, Rect> targetPositions = new Dictionary<UIElement, Rect>();  
 Dictionary<UIElement, Rect> currentPositions = new Dictionary<UIElement, Rect>();  
 Dictionary<UIElement, Rect> startingPositions = new Dictionary<UIElement, Rect>();  
 DateTime lastUpdateTime = DateTime.Now;  
 DateTime endTime = DateTime.Now;  
   
 public bool IsAnimating { get; set; }  
  
 public TimeSpan Duration  
 {  
 get { return (TimeSpan)GetValue(DurationProperty); }  
 set { SetValue(DurationProperty, value); }  
 }  
  
 public static readonly DependencyProperty DurationProperty =  
 DependencyProperty.Register("Duration", typeof(TimeSpan), typeof(AnimatedPanel), new UIPropertyMetadata(TimeSpan.FromMilliseconds(500)));  
  
 protected override Size ArrangeOverride(Size finalSize)  
 {  
 DateTime now = DateTime.Now;  
  
 foreach (UIElement child in this.Children)  
 {  
 if (!this.targetPositions.ContainsKey(child))  
 throw new InvalidOperationException("Must call AnimatedPanel.AnimatedArrange for all children");  
  
 if (!this.currentPositions.ContainsKey(child))  
 this.currentPositions[child] = this.targetPositions[child];  
  
 if (!this.startingPositions.ContainsKey(child))  
 this.startingPositions[child] = this.targetPositions[child];  
 }  
   
 bool somethingMoved = false;  
 foreach (UIElement child in this.Children)  
 {  
 if (this.startingPositions.ContainsKey(child))  
 {  
 Rect s = this.startingPositions[child];  
 Rect t = this.targetPositions[child];  
 if (s.Left != t.Left || s.Top != t.Top || s.Width != t.Width || s.Height != t.Height)  
 {  
 somethingMoved = true;  
 break;  
 }  
 }  
 }  
  
 if (somethingMoved)  
 {  
 // Start animating (make endTime later than now)  
 this.IsAnimating = true;  
 this.lastUpdateTime = now;  
 this.endTime = this.lastUpdateTime.AddMilliseconds(this.Duration.TotalMilliseconds);  
 foreach (UIElement child in this.Children)  
 {  
 this.startingPositions[child] = this.targetPositions[child];  
 }  
 }  
  
 double timeRemaining = (this.endTime - now).TotalMilliseconds;  
  
 double deltaTimeSinceLastUpdate = (now - lastUpdateTime).TotalMilliseconds;  
 if (deltaTimeSinceLastUpdate > timeRemaining)  
 deltaTimeSinceLastUpdate = timeRemaining;  
  
 DateTime startTime = this.endTime.AddMilliseconds(-this.Duration.TotalMilliseconds);  
 double timeIntoAnimation = (now - startTime).TotalMilliseconds;  
  
 double fractionComplete;  
 if (timeRemaining > 0)  
 fractionComplete = GetCurrentValue(timeIntoAnimation, 0, 1, this.Duration.TotalMilliseconds);  
 else  
 fractionComplete = 1;  
  
// Debug.WriteLine("Arrange " + fractionComplete.ToString());  
  
 foreach (UIElement child in this.Children)  
 {  
 Rect t = this.targetPositions[child];  
 Rect c = this.currentPositions[child];  
 double left = ((t.Left - c.Left) \* fractionComplete) + c.Left;  
 Rect pos = new Rect(left, ((t.Top - c.Top) \* fractionComplete) + c.Top,  
 ((t.Width - c.Width) \* fractionComplete) + c.Width, ((t.Height - c.Height) \* fractionComplete) + c.Height);  
  
 child.Arrange(pos);  
 this.currentPositions[child] = pos;  
 }  
  
 this.lastUpdateTime = now;  
  
 CompositionTarget.Rendering -= CompositionTarget\_Rendering;  
  
 if (timeRemaining > 0)  
 {  
 CompositionTarget.Rendering += CompositionTarget\_Rendering;  
 }  
 else  
 {  
 this.IsAnimating = false;  
 }  
  
 Clean(this.startingPositions);  
 Clean(this.currentPositions);  
 Clean(this.targetPositions);  
  
 return base.ArrangeOverride(finalSize);  
 }  
  
 // Dictionary may reference children that have been removed  
 private void Clean(Dictionary<UIElement, Rect> dictionary)  
 {  
 if (dictionary.Count != this.Children.Count)  
 {  
 Dictionary<UIElement, Rect> newDictionary = new Dictionary<UIElement, Rect>();  
 foreach (UIElement child in this.Children)  
 {  
 newDictionary[child] = dictionary[child];  
 }  
  
 dictionary = newDictionary;  
 }  
 }  
  
 public void AnimatedArrange(UIElement child, Rect finalSize)  
 {  
 this.targetPositions[child] = finalSize;  
 }  
  
 void CompositionTarget\_Rendering(object sender, EventArgs e)  
 {  
 this.InvalidateArrange();  
 }  
  
 /\*\*  
 \* Default easing equation: EaseInOutQuad   
 \* Easing equation function for a quadratic (t^2) easing in/out: acceleration until halfway, then deceleration.  
 \*  
 \* time Current time (in milliseconds).  
 \* begin Starting value.  
 \* change Change needed in value.  
 \* duration Expected easing duration (in milliseconds).  
 \* @return The correct value.  
 \*/  
 public virtual double GetCurrentValue(double time, double begin, double change, double duration)  
 {  
 if ((time /= duration / 2) < 1)  
 return change / 2 \* time \* time + begin;  
 return -change / 2 \* ((--time) \* (time - 2) - 1) + begin;  
 }  
 }  
}

Firstly we have a Duration dependency property that controls the duration of transition animations. Then notice that we don’t even bother hooking the MeasureOverride method, it all happens in the ArrangeOverride method. If you look near the bottom of this method, you’ll see that we hook CompositionTarget\_Rendering and invalidate the arrange. CompositionTarget\_Rendering gets called for every frame of graphics as it is rendered so maybe 50 times per second. By invalidating the arrange, we cause the warp panel and the base class to rerun their arrange code. Naturally we only hook this when we are actually animating, and doing this gives a very fluid looking animation.

In our arrange override method, we first check our storage contains all the children. If a new child is added to the panel, we need to add it to the dictionaries so that nothing crashes. Note that new children are positioned immediately and snap into place. It is fairly easy to update their start position to make them animate into place if you prefer.

Next we check if the calling panel (AnimatedWrapPanel in this case) is asking to actually move one of the children, of if the arrangement is staying the same. Only if something moves do we start animating things into position.

Then it’s a simple matter of calculating the distance into the animation as a number between 0 and 1 (fractionComplete). Note that we do this by calling a function that can be overridden – GetCurrentValue(). The default implementation provides a quadratic easing function which is the most common easing function that is used for animations. This provides an acceleration at the beginning of the animation and a deceleration at the end, and gives a very natural look. You can override this function and provide any function you require.

We then use fractionComplete to adjust the position and size of all of our children and call Arrange. This will smoothly animate the children to the next location. Note that if the panel rearranges the children before the animation is complete, we just start another animation and move them to their new position – you can see this by resizing the window quickly – the children start to animate to one position then switch direction and move to the next. This was quite hard to achieve and took a lot of debugging so is something to watch for if you change the code.

Then we decide whether to continue hooking CompositionTarget\_Render or stop if the animation is complete, and we clean up our dictionaries of any children that have been removed from the panel. Not doing this would cause memory leaks as adding children then removing them from the panel would cause them to still be referenced and not garbage collected.

So now if you have a custom panel you want to make animated, just make it inherit from AnimatedPanel and make it call AnimatedArrange instead of Arrange and then you are done! Enjoy!

*Written by* [*Paul Tallett*](http://blogs.msdn.com/mcsuksoldev/archive/tags/Paul+Tallett/default.aspx)