|  |  |
| --- | --- |
|  | [**WPF Tutorial - Implementing IScrollInfo**](http://www.switchonthecode.com/tutorials/wpf-tutorial-implementing-iscrollinfo) |

Posted in:

* [Tutorials](http://www.switchonthecode.com/tutorial)
* [.NET 3.5](http://www.switchonthecode.com/frameworks/dotnet/dotnet-35)
* [C#](http://www.switchonthecode.com/languages/csharp)
* [WPF](http://www.switchonthecode.com/frameworks/wpf)
* [XAML](http://www.switchonthecode.com/languages/xaml)

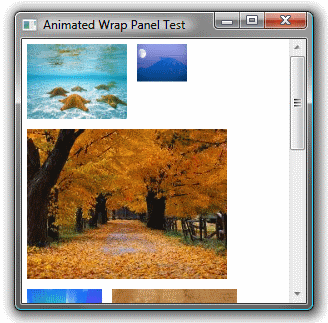
**Related Posts**

* [WPF Tutorial - Creating A Custom Panel Control](http://www.switchonthecode.com/tutorials/wpf-tutorial-creating-a-custom-panel-control)
* [WPF Tutorial - Using A Visual Collection](http://www.switchonthecode.com/tutorials/wpf-tutorial-using-a-visual-collection)
* [WinForms Tutorial - Manage Your Own Double Buffering](http://www.switchonthecode.com/tutorials/winforms-tutorial-manage-your-own-double-buffering)
* [Creating an XP Style WPF Button with Silverlight](http://www.switchonthecode.com/tutorials/creating-an-xp-style-wpf-button-with-silverlight)

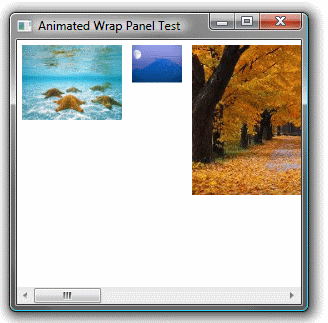
The [ScrollViewer](http://msdn.microsoft.com/en-us/library/system.windows.controls.scrollviewer.aspx) in WPF is pretty handy (and quite flexible) - especially when compared to what you had to work with in WinForms ([ScrollableControl](http://msdn.microsoft.com/en-us/library/system.windows.forms.scrollablecontrol.aspx)). 98% of the time, I can make the ScrollViewer do what I need it to for the given situation. Those other 2 percent, though, can get kind of hairy. Fortunately, WPF provides the [IScrollInfo](http://msdn.microsoft.com/en-us/library/system.windows.controls.primitives.iscrollinfo.aspx) interface - which is what we will be talking about today.

So what is IScrollInfo? Well, it is a way to take over the logic behind scrolling, while still maintaining the look and feel of the standard ScrollViewer. Now, first off, why in the world would we want to do that? To answer that question, I'm going to take a an example from a tutorial that is over a year old now - [Creating a Custom Panel Control](http://www.switchonthecode.com/tutorials/wpf-tutorial-creating-a-custom-panel-control). In that tutorial, we created our own custom WPF panel (that animated!). One of the issues with that panel though (and the WPF [WrapPanel](http://msdn.microsoft.com/en-us/library/system.windows.controls.wrappanel.aspx) in general) is that you have to disable the horizontal scrollbar if you put the panel in a ScrollViewer.

If you don't, you go from something that looks like this:

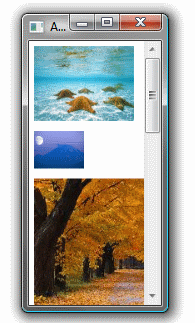


To something that looks like this:



And that kind of really defeats the purpose of a wrap panel.

The problem with disabling the horizontal scroll bar altogether is a situation like this:



In that case, you would really like a horizontal scroll bar to be there, but not change the wrapping behavior. And to get that behavior, you have to write your own custom scroll logic using IScrollInfo.

Ok, time to dive into the code. First, let's take a look at what methods IScrollInfo requires us to implement:

public class AnimatedWrapPanel : IScrollInfo  
{  
  public void LineDown(){ }  
  
  public void LineLeft(){ }  
  
  public void LineRight(){ }  
  
  public void LineUp(){ }  
  
  public void MouseWheelDown() { }  
  
  public void MouseWheelLeft() { }  
  
  public void MouseWheelRight() { }  
  
  public void MouseWheelUp() { }  
  
  public void PageDown() { }  
  
  public void PageLeft() { }  
  
  public void PageRight() { }  
  
  public void PageUp() { }  
  
  public ScrollViewer ScrollOwner { get; set; }  
  
  public bool CanHorizontallyScroll { get; set; }  
  
  public bool CanVerticallyScroll { get; set; }  
  
  public double ExtentHeight { get; }  
  
  public double ExtentWidth { get; }  
  
  public double HorizontalOffset { get; }  
  
  public double VerticalOffset { get; }  
  
  public double ViewportHeight { get; }  
  
  public double ViewportWidth { get; }  
  
  public Rect MakeVisible(Visual visual, Rect rectangle)  
  { }  
  
  public void SetHorizontalOffset(double offset)  
  { }  
  
  public void SetVerticalOffset(double offset)  
  { }  
}

Wow! Thats quite a lot of stuff there. But don't worry - almost all of it is your basic simple fill in the blank. For instance, take all the 'Up', 'Down', 'Left', 'Right' methods. Those methods give you fine grained control over how much your panel will scroll when the user clicks the up/down buttons on the scroll bar, or scrolls their mouse wheel. But for our purposes, they can be filled in pretty easily:

public class AnimatedWrapPanel : IScrollInfo  
{  
  private const double LineSize = 16;  
  private const double WheelSize = 3 \* LineSize;  
  
  public void LineDown()  
  { SetVerticalOffset(VerticalOffset + LineSize); }  
  
  public void LineUp()  
  { SetVerticalOffset(VerticalOffset - LineSize); }  
  
  public void LineLeft()  
  { SetHorizontalOffset(HorizontalOffset - LineSize); }  
  
  public void LineRight()  
  { SetHorizontalOffset(HorizontalOffset + LineSize); }  
  
  public void MouseWheelDown()  
  { SetVerticalOffset(VerticalOffset + WheelSize); }  
  
  public void MouseWheelUp()  
  { SetVerticalOffset(VerticalOffset - WheelSize); }  
  
  public void MouseWheelLeft()  
  { SetHorizontalOffset(HorizontalOffset - WheelSize); }  
  
  public void MouseWheelRight()  
  { SetHorizontalOffset(HorizontalOffset + WheelSize); }  
  
  public void PageDown()  
  { SetVerticalOffset(VerticalOffset + ViewportHeight); }  
  
  public void PageUp()  
  { SetVerticalOffset(VerticalOffset - ViewportHeight); }  
  
  public void PageLeft()  
  { SetHorizontalOffset(HorizontalOffset - ViewportWidth); }  
  
  public void PageRight()  
  { SetHorizontalOffset(HorizontalOffset + ViewportWidth); }  
  
  public ScrollViewer ScrollOwner { get; set; }  
  
  public bool CanHorizontallyScroll { get; set; }  
  
  public bool CanVerticallyScroll { get; set; }  
  
  public double ExtentHeight { get; }  
  
  public double ExtentWidth { get; }  
  
  public double HorizontalOffset { get; }  
  
  public double VerticalOffset { get; }  
  
  public double ViewportHeight { get; }  
  
  public double ViewportWidth { get; }  
  
  public Rect MakeVisible(Visual visual, Rect rectangle)  
  { }  
  
  public void SetHorizontalOffset(double offset)  
  { }  
  
  public void SetVerticalOffset(double offset)  
  { }  
}

Just set up some constants for the amount to scroll per line and per wheel click, and away we go! Thats over half the methods down already. Now let's take care of some of those pesky properties:

public class AnimatedWrapPanel : IScrollInfo  
{  
  private const double LineSize = 16;  
  private const double WheelSize = 3 \* LineSize;  
  
  private bool \_CanHorizontallyScroll;  
  private bool \_CanVerticallyScroll;  
  private ScrollViewer \_ScrollOwner;  
  private Vector \_Offset;  
  private Size \_Extent;  
  private Size \_Viewport;  
  
  public void LineDown()  
  { SetVerticalOffset(VerticalOffset + LineSize); }  
  
  public void LineUp()  
  { SetVerticalOffset(VerticalOffset - LineSize); }  
  
  public void LineLeft()  
  { SetHorizontalOffset(HorizontalOffset - LineSize); }  
  
  public void LineRight()  
  { SetHorizontalOffset(HorizontalOffset + LineSize); }  
  
  public void MouseWheelDown()  
  { SetVerticalOffset(VerticalOffset + WheelSize); }  
  
  public void MouseWheelUp()  
  { SetVerticalOffset(VerticalOffset - WheelSize); }  
  
  public void MouseWheelLeft()  
  { SetHorizontalOffset(HorizontalOffset - WheelSize); }  
  
  public void MouseWheelRight()  
  { SetHorizontalOffset(HorizontalOffset + WheelSize); }  
  
  public void PageDown()  
  { SetVerticalOffset(VerticalOffset + ViewportHeight); }  
  
  public void PageUp()  
  { SetVerticalOffset(VerticalOffset - ViewportHeight); }  
  
  public void PageLeft()  
  { SetHorizontalOffset(HorizontalOffset - ViewportWidth); }  
  
  public void PageRight()  
  { SetHorizontalOffset(HorizontalOffset + ViewportWidth); }  
  
  public ScrollViewer ScrollOwner  
  {  
    get { return \_ScrollOwner; }  
    set { \_ScrollOwner = value; }  
  }  
  
  public bool CanHorizontallyScroll  
  {  
    get { return \_CanHorizontallyScroll; }  
    set { \_CanHorizontallyScroll = value; }  
  }  
  
  public bool CanVerticallyScroll  
  {  
    get { return \_CanVerticallyScroll; }  
    set { \_CanVerticallyScroll = value; }  
  }  
  
  public double ExtentHeight  
  { get { return \_Extent.Height; } }  
  
  public double ExtentWidth  
  { get { return \_Extent.Width; } }  
  
  public double HorizontalOffset  
  { get { return \_Offset.X; } }  
  
  public double VerticalOffset  
  { get { return \_Offset.Y; } }  
  
  public double ViewportHeight  
  { get { return \_Viewport.Height; } }  
  
  public double ViewportWidth  
  { get { return \_Viewport.Width; } }  
  
  public Rect MakeVisible(Visual visual, Rect rectangle)  
  { }  
  
  public void SetHorizontalOffset(double offset)  
  { }  
  
  public void SetVerticalOffset(double offset)  
  { }  
}

Pretty much we just needed to set up backing fields for all those properties. The property names are pretty self explanatory - "Extent" is the total size of the panel, while "Viewport" is the amount that is visible on screen. "Offset" is the amount that the viewport is offset from 0,0 - i.e., how far scrolled down/right we are.

What is left are the more complicated parts of the interface. First, let's fill out the SetHorizontalOffset and SetVerticalOffset calls:

public void SetHorizontalOffset(double offset)  
{  
  offset = Math.Max(0, Math.Min(offset, ExtentWidth - ViewportWidth));  
  if (offset != \_Offset.Y)  
  {  
    \_Offset.X = offset;  
    InvalidateArrange();  
  }  
}  
  
public void SetVerticalOffset(double offset)  
{  
  offset = Math.Max(0, Math.Min(offset, ExtentHeight - ViewportHeight));  
  if (offset != \_Offset.Y)  
  {  
    \_Offset.Y = offset;  
    InvalidateArrange();  
  }  
}

In both cases, we force the offset into a valid range, and then if it is different than the current offset, we set it as the new offset and invalidate the arrange of the panel (so that the items on the panel will get moved appropriately).

What's left on the interface is MakeVisible, which is what gets called to scroll an item into view. The code in there is just a bunch of math to calculate new scroll offsets - I'm not going to walk through it, but you can check it out in the full code farther down the tutorial.

So the interface is fully implemented. But sadly, that isn't enough - we still have to do things like calculate the Viewport and the Extent, as well as modify the MeasureOverride and ArrangeOverride to deal with their own scroll behavior.

If you take a look at the code in [Creating a Custom Panel Control](http://www.switchonthecode.com/tutorials/wpf-tutorial-creating-a-custom-panel-control), the following code might look very familiar. This is because I tried to modify the code for the original animated wrap panel as little as possible - see if you can spot the changes:

protected override Size MeasureOverride(Size availableSize)  
{  
  double curX = 0, curY = 0, curLineHeight = 0, maxLineWidth = 0;  
  foreach (UIElement child in Children)  
  {  
    child.Measure(InfiniteSize);  
  
    if (curX + child.DesiredSize.Width > availableSize.Width)  
    { *//Wrap to next line*  
      curY += curLineHeight;  
      curX = 0;  
      curLineHeight = 0;  
    }  
  
    curX += child.DesiredSize.Width;  
    if (child.DesiredSize.Height > curLineHeight)  
    { curLineHeight = child.DesiredSize.Height; }  
      
    if (curX > maxLineWidth)  
    { maxLineWidth = curX; }  
  }  
  
  curY += curLineHeight;  
  
  VerifyScrollData(availableSize, new Size(maxLineWidth, curY));  
  
  return \_Viewport;  
}  
  
protected override Size ArrangeOverride(Size finalSize)  
{  
  if (this.Children == null || this.Children.Count == 0)  
  { return finalSize; }  
  
  TranslateTransform trans = null;  
  double curX = 0, curY = 0, curLineHeight = 0, maxLineWidth = 0;  
  
  foreach (UIElement child in Children)  
  {  
    trans = child.RenderTransform as TranslateTransform;  
    if (trans == null)  
    {  
      child.RenderTransformOrigin = new Point(0, 0);  
      trans = new TranslateTransform();  
      child.RenderTransform = trans;  
    }  
  
    if (curX + child.DesiredSize.Width > finalSize.Width)  
    { *//Wrap to next line*  
      curY += curLineHeight;  
      curX = 0;  
      curLineHeight = 0;  
    }  
  
    child.Arrange(new Rect(0, 0,   
      child.DesiredSize.Width, child.DesiredSize.Height));  
  
    trans.BeginAnimation(TranslateTransform.XProperty,   
      new DoubleAnimation(curX - HorizontalOffset, \_AnimationLength),   
      HandoffBehavior.Compose);  
    trans.BeginAnimation(TranslateTransform.YProperty,   
      new DoubleAnimation(curY - VerticalOffset, \_AnimationLength),   
      HandoffBehavior.Compose);  
  
    curX += child.DesiredSize.Width;  
    if (child.DesiredSize.Height > curLineHeight)  
    { curLineHeight = child.DesiredSize.Height; }  
  
    if (curX > maxLineWidth)  
    { maxLineWidth = curX; }  
  }  
  
  curY += curLineHeight;  
  VerifyScrollData(finalSize, new Size(maxLineWidth, curY));  
  
  return finalSize;  
}

MeasureOverride is almost identical to the old code, except for two things. One, we keep track of the max row width, to correctly calculate the horizontal extent of the panel. Two, we have a call to VerifyScrollData at the end of the method - a method we have not seen yet (but will be taking a look at soon).

ArrangeOverride has a couple more changes. Again, we are keeping track of the max row width, and calling VerifyScrollData. But we are also modifying the positions at which the items are placed by the amount of the scroll offset. This is because since we are in charge of scrolling behavior, we are also in charge of making sure items are place correctly according to the scrolling behavior.

Ok, now for that VerifyScrollData method:

protected void VerifyScrollData(Size viewport, Size extent)  
{  
  if (double.IsInfinity(viewport.Width))  
  { viewport.Width = extent.Width; }  
  
  if (double.IsInfinity(viewport.Height))  
  { viewport.Height = extent.Height; }  
  
  \_Extent = extent;  
  \_Viewport = viewport;  
  
  \_Offset.X = Math.Max(0,   
    Math.Min(\_Offset.X, ExtentWidth - ViewportWidth));  
  \_Offset.Y = Math.Max(0,   
    Math.Min(\_Offset.Y, ExtentHeight - ViewportHeight));  
  
  if (ScrollOwner != null)  
  { ScrollOwner.InvalidateScrollInfo(); }  
}

It is this function that sets the viewport and extent fields. It also coerces the offsets to be within the correct ranges (changes to the extent/viewport can make a previously valid offset incorrect). Finally, if there is a scroll owner currently attached, we call [InvalidateScrollInfo](http://msdn.microsoft.com/en-us/library/system.windows.controls.scrollviewer.invalidatescrollinfo.aspx).  
This makes sure that the scrolviewer is displaying the right ranges and positions for the scrollbars.

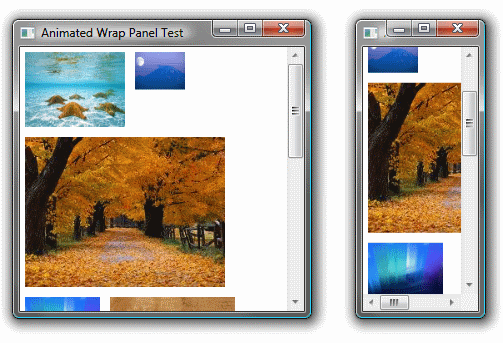
And that is it for implementing your own implementation of IScrollInfo. Here is all the code together:

using System;  
using System.Windows;  
using System.Windows.Controls;  
using System.Windows.Controls.Primitives;  
using System.Windows.Media;  
using System.Windows.Media.Animation;  
  
namespace AnimatedWrapPanel  
{  
  public class AnimatedWrapPanel : Panel, IScrollInfo  
  {  
    private static Size InfiniteSize =  
      new Size(double.PositiveInfinity, double.PositiveInfinity);  
    private const double LineSize = 16;  
    private const double WheelSize = 3 \* LineSize;  
  
    private bool \_CanHorizontallyScroll;  
    private bool \_CanVerticallyScroll;  
    private ScrollViewer \_ScrollOwner;  
    private Vector \_Offset;  
    private Size \_Extent;  
    private Size \_Viewport;  
  
    private TimeSpan \_AnimationLength = TimeSpan.FromMilliseconds(200);  
  
    protected override Size MeasureOverride(Size availableSize)  
    {  
      double curX = 0, curY = 0, curLineHeight = 0, maxLineWidth = 0;  
      foreach (UIElement child in Children)  
      {  
        child.Measure(InfiniteSize);  
  
        if (curX + child.DesiredSize.Width > availableSize.Width)  
        { *//Wrap to next line*  
          curY += curLineHeight;  
          curX = 0;  
          curLineHeight = 0;  
        }  
  
        curX += child.DesiredSize.Width;  
        if (child.DesiredSize.Height > curLineHeight)  
        { curLineHeight = child.DesiredSize.Height; }  
          
        if (curX > maxLineWidth)  
        { maxLineWidth = curX; }  
      }  
  
      curY += curLineHeight;  
  
      VerifyScrollData(availableSize, new Size(maxLineWidth, curY));  
  
      return \_Viewport;  
    }  
  
    protected override Size ArrangeOverride(Size finalSize)  
    {  
      if (this.Children == null || this.Children.Count == 0)  
      { return finalSize; }  
  
      TranslateTransform trans = null;  
      double curX = 0, curY = 0, curLineHeight = 0, maxLineWidth = 0;  
  
      foreach (UIElement child in Children)  
      {  
        trans = child.RenderTransform as TranslateTransform;  
        if (trans == null)  
        {  
          child.RenderTransformOrigin = new Point(0, 0);  
          trans = new TranslateTransform();  
          child.RenderTransform = trans;  
        }  
  
        if (curX + child.DesiredSize.Width > finalSize.Width)  
        { *//Wrap to next line*  
          curY += curLineHeight;  
          curX = 0;  
          curLineHeight = 0;  
        }  
  
        child.Arrange(new Rect(0, 0,   
          child.DesiredSize.Width, child.DesiredSize.Height));  
  
        trans.BeginAnimation(TranslateTransform.XProperty,   
          new DoubleAnimation(curX - HorizontalOffset, \_AnimationLength),   
          HandoffBehavior.Compose);  
        trans.BeginAnimation(TranslateTransform.YProperty,   
          new DoubleAnimation(curY - VerticalOffset, \_AnimationLength),   
          HandoffBehavior.Compose);  
  
        curX += child.DesiredSize.Width;  
        if (child.DesiredSize.Height > curLineHeight)  
        { curLineHeight = child.DesiredSize.Height; }  
  
        if (curX > maxLineWidth)  
        { maxLineWidth = curX; }  
      }  
  
      curY += curLineHeight;  
      VerifyScrollData(finalSize, new Size(maxLineWidth, curY));  
  
      return finalSize;  
    }  
      
    #region Movement Methods  
    public void LineDown()  
    { SetVerticalOffset(VerticalOffset + LineSize); }  
  
    public void LineUp()  
    { SetVerticalOffset(VerticalOffset - LineSize); }  
  
    public void LineLeft()  
    { SetHorizontalOffset(HorizontalOffset - LineSize); }  
  
    public void LineRight()  
    { SetHorizontalOffset(HorizontalOffset + LineSize); }  
  
    public void MouseWheelDown()  
    { SetVerticalOffset(VerticalOffset + WheelSize); }  
  
    public void MouseWheelUp()  
    { SetVerticalOffset(VerticalOffset - WheelSize); }  
  
    public void MouseWheelLeft()  
    { SetHorizontalOffset(HorizontalOffset - WheelSize); }  
  
    public void MouseWheelRight()  
    { SetHorizontalOffset(HorizontalOffset + WheelSize); }  
  
    public void PageDown()  
    { SetVerticalOffset(VerticalOffset + ViewportHeight); }  
  
    public void PageUp()  
    { SetVerticalOffset(VerticalOffset - ViewportHeight); }  
  
    public void PageLeft()  
    { SetHorizontalOffset(HorizontalOffset - ViewportWidth); }  
  
    public void PageRight()  
    { SetHorizontalOffset(HorizontalOffset + ViewportWidth); }  
    #endregion  
  
    public ScrollViewer ScrollOwner  
    {  
      get { return \_ScrollOwner; }  
      set { \_ScrollOwner = value; }  
    }  
  
    public bool CanHorizontallyScroll  
    {  
      get { return \_CanHorizontallyScroll; }  
      set { \_CanHorizontallyScroll = value; }  
    }  
  
    public bool CanVerticallyScroll  
    {  
      get { return \_CanVerticallyScroll; }  
      set { \_CanVerticallyScroll = value; }  
    }  
  
    public double ExtentHeight  
    { get { return \_Extent.Height; } }  
  
    public double ExtentWidth  
    { get { return \_Extent.Width; } }  
  
    public double HorizontalOffset  
    { get { return \_Offset.X; } }  
  
    public double VerticalOffset  
    { get { return \_Offset.Y; } }  
  
    public double ViewportHeight  
    { get { return \_Viewport.Height; } }  
  
    public double ViewportWidth  
    { get { return \_Viewport.Width; } }  
  
    public Rect MakeVisible(Visual visual, Rect rectangle)  
    {  
      if (rectangle.IsEmpty || visual == null   
        || visual == this || !base.IsAncestorOf(visual))  
      { return Rect.Empty; }  
  
      rectangle = visual.TransformToAncestor(this).TransformBounds(rectangle);  
  
      Rect viewRect = new Rect(HorizontalOffset,   
        VerticalOffset, ViewportWidth, ViewportHeight);  
      rectangle.X += viewRect.X;  
      rectangle.Y += viewRect.Y;  
      viewRect.X = CalculateNewScrollOffset(viewRect.Left,   
        viewRect.Right, rectangle.Left, rectangle.Right);  
      viewRect.Y = CalculateNewScrollOffset(viewRect.Top,   
        viewRect.Bottom, rectangle.Top, rectangle.Bottom);  
      SetHorizontalOffset(viewRect.X);  
      SetVerticalOffset(viewRect.Y);  
      rectangle.Intersect(viewRect);  
      rectangle.X -= viewRect.X;  
      rectangle.Y -= viewRect.Y;  
  
      return rectangle;  
    }  
  
    private static double CalculateNewScrollOffset(double topView,   
      double bottomView, double topChild, double bottomChild)  
    {  
      bool offBottom = topChild < topView && bottomChild < bottomView;  
      bool offTop = bottomChild > bottomView && topChild > topView;  
      bool tooLarge = (bottomChild - topChild) > (bottomView - topView);  
  
      if (!offBottom && !offTop)  
      { return topView; } *//Don't do anything, already in view*  
  
      if ((offBottom && !tooLarge) || (offTop && tooLarge))  
      { return topChild; }  
  
      return (bottomChild - (bottomView - topView));  
    }  
  
    protected void VerifyScrollData(Size viewport, Size extent)  
    {  
      if (double.IsInfinity(viewport.Width))  
      { viewport.Width = extent.Width; }  
  
      if (double.IsInfinity(viewport.Height))  
      { viewport.Height = extent.Height; }  
  
      \_Extent = extent;  
      \_Viewport = viewport;  
  
      \_Offset.X = Math.Max(0,   
        Math.Min(\_Offset.X, ExtentWidth - ViewportWidth));  
      \_Offset.Y = Math.Max(0,   
        Math.Min(\_Offset.Y, ExtentHeight - ViewportHeight));  
  
      if (ScrollOwner != null)  
      { ScrollOwner.InvalidateScrollInfo(); }  
    }  
  
    public void SetHorizontalOffset(double offset)  
    {  
      offset = Math.Max(0,   
        Math.Min(offset, ExtentWidth - ViewportWidth));  
      if (offset != \_Offset.Y)  
      {  
        \_Offset.X = offset;  
        InvalidateArrange();  
      }  
    }  
  
    public void SetVerticalOffset(double offset)  
    {  
      offset = Math.Max(0,   
        Math.Min(offset, ExtentHeight - ViewportHeight));  
      if (offset != \_Offset.Y)  
      {  
        \_Offset.Y = offset;  
        InvalidateArrange();  
      }  
    }  
  }  
}

Now, in order to use a class that implements IScrollInfo, you do have to do one other thing - you have to remember to set the property [CanContentScroll](http://msdn.microsoft.com/en-us/library/system.windows.controls.scrollviewer.cancontentscroll.aspx) to true on the ScrollViewer surrounding the instance of your class. This signifies to the ScrollViewer that the content can control its own scroll behavior - letting the logic that you have written work its magic. So with that property set, your XAML might look something like this:

<Window x:Class="AnimatedWrapPanel.Window1"  
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
   xmlns:ARP="clr-namespace:AnimatedWrapPanel"  
   Title="Animated Wrap Panel Test" Height="300" Width="300">  
  <ScrollViewer CanContentScroll="True"   
               HorizontalScrollBarVisibility="Auto"  
               VerticalScrollBarVisibility="Auto">  
    <ARP:AnimatedWrapPanel>  
      <Image Source="Images\Aquarium.jpg" Stretch="Uniform"   
            Width="100" Margin="5"/>  
      <Image Source="Images\Ascent.jpg" Stretch="Uniform"   
            Width="50" Margin="5" />  
      <Image Source="Images\Autumn.jpg" Stretch="Uniform"   
            Width="200" Margin="5"/>  
      <Image Source="Images\Crystal.jpg" Stretch="Uniform"   
            Width="75" Margin="5"/>  
      <Image Source="Images\DaVinci.jpg" Stretch="Uniform"   
            Width="125" Margin="5"/>  
      <Image Source="Images\Follow.jpg" Stretch="Uniform"   
            Width="100" Margin="5"/>  
      <Image Source="Images\Friend.jpg" Stretch="Uniform"   
            Width="50" Margin="5"/>  
      <Image Source="Images\Home.jpg" Stretch="Uniform"   
            Width="150" Margin="5"/>  
      <Image Source="Images\Moon flower.jpg" Stretch="Uniform"   
            Width="100" Margin="5"/>  
    </ARP:AnimatedWrapPanel>  
  </ScrollViewer>  
</Window>

And this leaves you a panel that can scroll horizontally, but only when it needs to:



Well, that about wraps it up. As always, you can grab the source for the example below if you want to play with the code on your own. Leave any comments or questions you might have , and I'll do my best to answer them.