[**Dan Crevier's VirtualizingTilePanel Mod**](http://www.cyqdata.com/cnblogs/article-detail-25097##)

[编辑](http://www.cyqdata.com/cnblogs/admin/article-edit-25097)2011-2-15 23:29:40[英文系列](http://www.cyqdata.com/cnblogs/article-cate-22)评论(0)浏览(33)

* [Download VirtualizingTilePanel - 4.21 KB](http://www.codeproject.com/KB/WPF/VirtualTilePanelFillMod/VirtualizingTilePanel.zip)

**Introduction**

This is barely worth an article but I'm posting it anyway because hopefully it will save time for others trying to implement the same mechanism. Dan Crevier has a wonderful post about how to implement a virtualizing tile/wrap panel in WPF ([link](http://blogs.msdn.com/b/dancre/archive/2006/02/16/implementing-a-virtualizingpanel-part-4-the-goods.aspx)). 99% of the code in this article belongs to him. My one issue with his implementation was that his layout logic can leave ugly empty space at the side of the items control. With Dan's logic you specify the width of each child item in the panel and the algorithm calculates how many children can occupy a row in the available space. With my logic you specify the number of children desired in each row and the algorithm calculates the width of each child.

**Using the code**

I'll just describe the changes from Dan's article. Two new dependency properties have been added. The Columns property sets the number of desired children on a row. The Tile property specifies which layout logic to execute. Set to true to use Dan's original logic, the Columns property if specified will be ignored in this case, otherwise set to false to use the changed logic.

The ChildSize property has been renamed to ChildHeight simply because this name was less confusing than ChildSize when writing the new logic; as under the new code it is actually only the height component that this property represents. Note that when using the original logic, this property still represents both the width and the height.

/// <summary>

/// Controls the size of the child elements.

/// </summary>

public static readonly DependencyProperty ChildHeightProperty

= DependencyProperty.RegisterAttached("ChildHeight", typeof(double), typeof(VirtualizingTilePanel),

new FrameworkPropertyMetadata(200.0d, FrameworkPropertyMetadataOptions.AffectsMeasure |

FrameworkPropertyMetadataOptions.AffectsArrange));

/// <summary>

/// Controls the number of the child elements in a row.

/// </summary>

public static readonly DependencyProperty ColumnsProperty

= DependencyProperty.RegisterAttached("Columns", typeof(int), typeof(VirtualizingTilePanel),

new FrameworkPropertyMetadata(10, FrameworkPropertyMetadataOptions.AffectsMeasure |

FrameworkPropertyMetadataOptions.AffectsArrange));

/// <summary>

/// If setting is true, the component will calulcate the number

/// of children per row, the width of each item is set equal

/// to the height. In this mode the columns property is ignored.

/// If the setting is false, the component will calulate the

/// width of each item by dividing the available size by the

/// number of desired rows.

/// </summary>

public static readonly DependencyProperty TileProperty

= DependencyProperty.RegisterAttached("Tile", typeof(bool), typeof(VirtualizingTilePanel),

new FrameworkPropertyMetadata(true, FrameworkPropertyMetadataOptions.AffectsMeasure |

FrameworkPropertyMetadataOptions.AffectsArrange));

/// <summary>

/// Gets or sets the height of each child.

/// </summary>

public double ChildHeight

{

get { return (double)GetValue(ChildHeightProperty); }

set { SetValue(ChildHeightProperty, value); }

}

/// <summary>

/// Gets or sets the number of desired columns.

/// </summary>

public int Columns

{

get { return (int)GetValue(ColumnsProperty); }

set { SetValue(ColumnsProperty, value); }

}

/// <summary>

/// Gets or sets whether the component is operating

/// in tile mode.If set to true, the component

/// will calulcate the number of children per row,

/// the width of each item is set equal to the height.

/// In this mode the Columns property is ignored. If the

/// setting is false, the component will calulate the

/// width of each item by dividing the available size

/// by the number of desired columns.

/// </summary>

public bool Tile

{

get { return (bool)GetValue(TileProperty); }

set { SetValue(TileProperty, value); }

}

And the layout specific code, which Dan had very nicely regionalised, has of course been modified.

/// <summary>

/// Calculate the extent of the view based on the available size

/// </summary>

/// <param name="availableSize">available size</param>

/// <param name="itemCount">number of data items</param>

/// <returns>Returns the extent size of the viewer.</returns>

private Size CalculateExtent(Size availableSize, int itemCount)

{

//If tile mode.

if (Tile)

{

//Gets the number of children or items for each row.

int childrenPerRow = CalculateChildrenPerRow(availableSize);

// See how big we are

return new Size(childrenPerRow \* this.ChildHeight,

this.ChildHeight \* Math.Ceiling((double)itemCount / childrenPerRow));

}

else

{

//Gets the width of each child.

double childWidth = CalculateChildWidth(availableSize);

// See how big we are

return new Size(this.Columns \* childWidth,

this.ChildHeight \* Math.Ceiling((double)itemCount / this.Columns));

}

}

/// <summary>

/// Get the range of children that are visible

/// </summary>

/// <param name="firstVisibleItemIndex">The item index of the first visible item</param>

/// <param name="lastVisibleItemIndex">The item index of the last visible item</param>

void GetVisibleRange(out int firstVisibleItemIndex, out int lastVisibleItemIndex)

{

//If tile mode.

if (Tile)

{

//Get the number of children

int childrenPerRow = CalculateChildrenPerRow(\_extent);

firstVisibleItemIndex = (int)Math.Floor(\_offset.Y / this.ChildHeight) \* childrenPerRow;

lastVisibleItemIndex = (int)Math.Ceiling((\_offset.Y + \_viewport.Height) / this.ChildHeight) \* childrenPerRow - 1;

ItemsControl itemsControl = ItemsControl.GetItemsOwner(this);

int itemCount = itemsControl.HasItems ? itemsControl.Items.Count : 0;

if (lastVisibleItemIndex >= itemCount)

lastVisibleItemIndex = itemCount - 1;

}

else

{

//Get the width of each child.

double childWidth = CalculateChildWidth(\_extent);

firstVisibleItemIndex = (int)Math.Floor(\_offset.Y / childWidth) \* this.Columns;

lastVisibleItemIndex = (int)Math.Ceiling((\_offset.Y + \_viewport.Height) / this.ChildHeight) \* this.Columns - 1;

ItemsControl itemsControl = ItemsControl.GetItemsOwner(this);

int itemCount = itemsControl.HasItems ? itemsControl.Items.Count : 0;

if (lastVisibleItemIndex >= itemCount)

lastVisibleItemIndex = itemCount - 1;

}

}

/// <summary>

/// Get the size of the each child.

/// </summary>

/// <returns>The size of each child.</returns>

Size GetChildSize(Size availableSize)

{

return new Size((Tile) ? this.ChildHeight :

CalculateChildWidth(availableSize), this.ChildHeight);

}

/// <summary>

/// Position a child

/// </summary>

/// <param name="itemIndex">The data item index of the child</param>

/// <param name="child">The element to position</param>

/// <param name="finalSize">The size of the panel</param>

void ArrangeChild(int itemIndex, UIElement child, Size finalSize)

{

if (Tile)

{

int childrenPerRow = CalculateChildrenPerRow(finalSize);

int row = itemIndex / childrenPerRow;

int column = itemIndex % childrenPerRow;

child.Arrange(new Rect(column \* this.ChildHeight, row \* this.ChildHeight,

this.ChildHeight, this.ChildHeight));

}

else

{

//Get the width of each child.

double childWidth = CalculateChildWidth(finalSize);

int row = itemIndex / this.Columns;

int column = itemIndex % this.Columns;

child.Arrange(new Rect(column \* childWidth, row \* this.ChildHeight,

childWidth, this.ChildHeight));

}

}

/// <summary>

/// Calculate the width of each tile by

/// dividing the width of available size

/// by the number of required columns.

/// </summary>

/// <param name="availableSize">The total layout size available.</param>

/// <returns>The width of each tile.</returns>

double CalculateChildWidth(Size availableSize)

{

return availableSize.Width / this.Columns;

}

/// <summary>

/// Helper function for tiling layout

/// </summary>

/// <param name="availableSize">Size available</param>

/// <returns>The number of tiles on each row.</returns>

int CalculateChildrenPerRow(Size availableSize)

{

// Figure out how many children fit on each row

int childrenPerRow;

if (availableSize.Width == Double.PositiveInfinity)

childrenPerRow = this.Children.Count;

else

childrenPerRow = Math.Max(1, (int)Math.Floor(availableSize.Width / this.ChildHeight));

return childrenPerRow;

}

**Points of Interest**

That's all there is to it. I'm sure some-one or some-all will point out the duplication of code within each method, definitely there is room to refactor into generic methods.

**History**