CODE PROJECT® For those who code

Articles » Platforms, Frameworks & Libraries » Windows Presentation Foundation » General

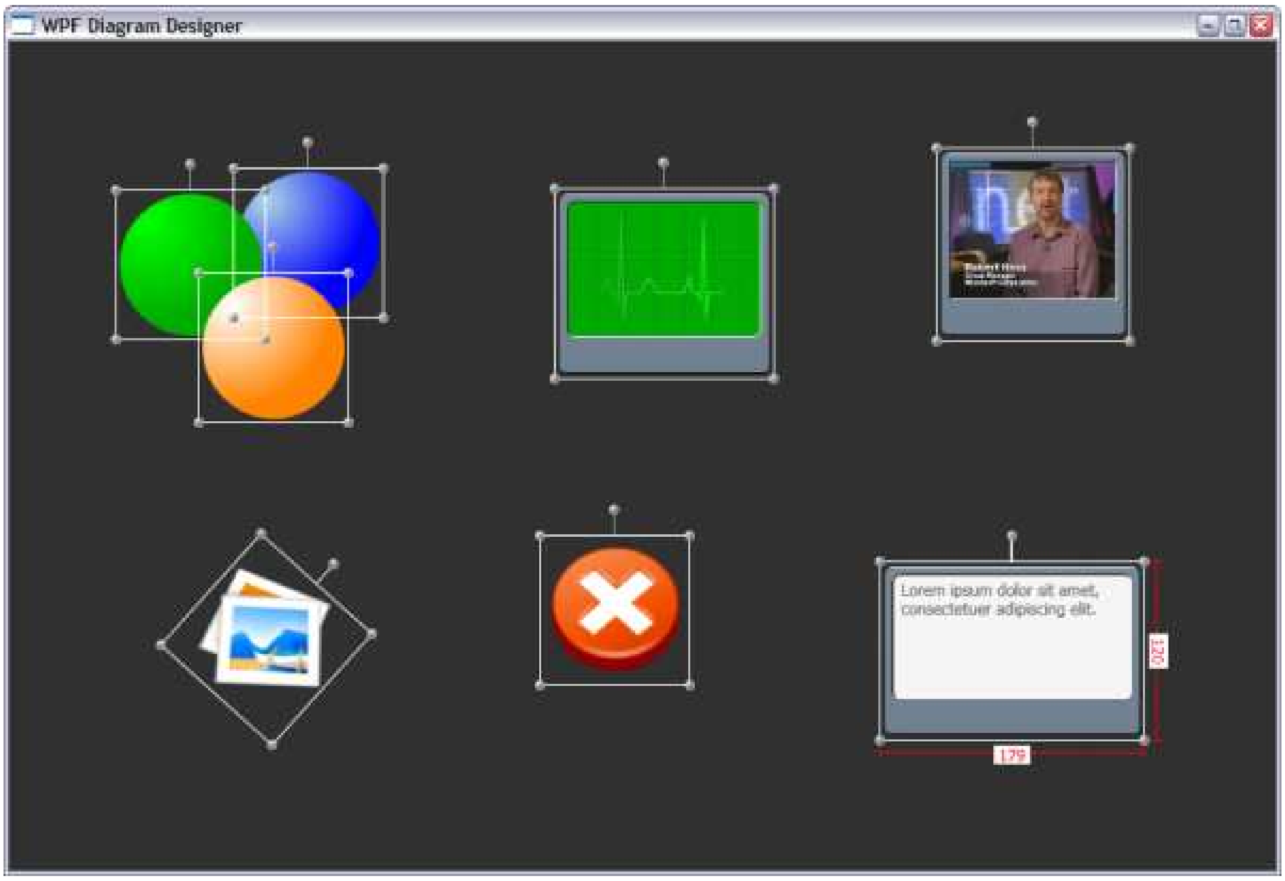
WPF Diagram Designer: Part 1

By **sukram**, 23 Aug 2008

★ ★ ★ ★ ★ 4.97 (170 votes)

Download source - 629.79 KB

Download binaries - 611.23 KB

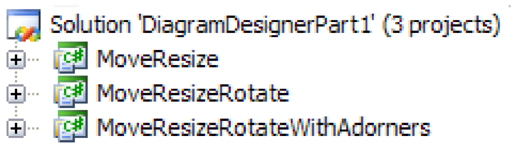


Introduction

In this article, I will show you how to move, resize and rotate objects of any type on a canvas. For this, I will provide two different solutions - the first one without and then one with WPF Adorners.

About the Code

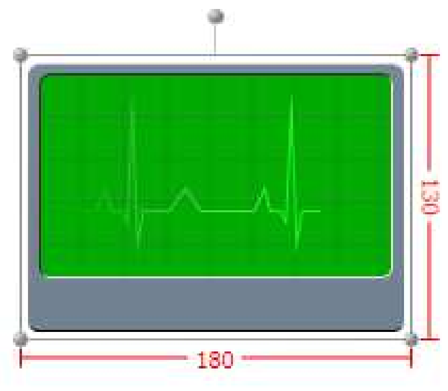
The attached Visual Studio Express 2008 solution consists of three projects:


Solution 'DiagramDesignerPart1' (3 projects)
MoveResize
MoveResizeRotate
MoveResizeRotateWithAdorners

`MoveResize`: This version shows you how to move and resize objects without WPF Adorners.

`MoveResizeRotate`: In addition, this project provides rotation of objects, still without WPF Adorners. Rotation has some minor side effects that need to be considered when moving or resizing objects. You can easily track these side effects when you compare this project with the previous one.

`MoveResizeRotateWithAdorners`: The third project finally shows you how to move, resize and rotate items with the help of WPF Adorners. It also gives you an example of how Adorners can be used to provide visual feedback to indicate the actual size of an object during a resize operation.



Preparations

We start with a simple diagram:

```
<Canvas>
  <Ellipse Fill="Blue"
           Width="100"
           Height="100"
           Canvas.Top="100"
           Canvas.Left="100"/>
</Canvas>
```

You might not be impressed by this diagram, nevertheless it is a good starting point. It is easy to understand and it has all a basic diagram needs to have: a drawing canvas with a shape. But you are right, this diagram isn't really useful - it's just too static.

So let's start with some preparations by wrapping the ellipse into a `ContentControl`:

```
 <Canvas>
   <ContentControl Width="100"
                   Height="100"
                   Canvas.Top="100"
                   Canvas.Left="100">
      <Ellipse Fill="Blue"/>
   </ContentControl>
</Canvas>
```

Not much better you may say, we still can't move the ellipse, so what is it good for? Well, the ContentControl serves as a container for the object that we want to place on the canvas and it is actually this ContentControl that we are going to move, resize and rotate! And because the content of a ContentControl can be of any type, we will be able to move, resize and rotate objects of any type on our canvas!

Note: Because of its key role, this ContentControl is also referred to as DesignerItem.

We conclude our preparations by assigning a ControlTemplate to the DesignerItem. This introduces a further level of abstraction, so that from now on we will just expand this template and leave the DesignerItem and its content completely untouched.

```
 <Canvas>
   <Canvas.Resources>
      <ControlTemplate x:Key="DesignerItemTemplate" TargetType="ContentControl">
         <ContentPresenter Content="{TemplateBinding ContentControl.Content}"/>
      </ControlTemplate>
   </Canvas.Resources>
   <ContentControl Name="DesignerItem"
                   Width="100"
                   Height="100"
                   Canvas.Top="100"
                   Canvas.Left="100"
                   Template="{StaticResource DesignerItemTemplate}">
      <Ellipse Fill="Blue"/>
   </ContentControl>
</Canvas>
```

Now that we have finished our preparations, we are ready to bring some activity on the canvas.

Move

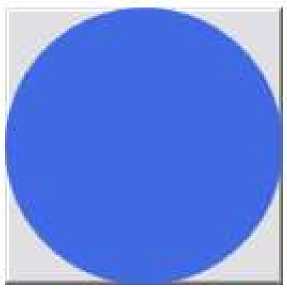
There is a control in WPF about which the MSDN documentation says: " ...represents a control that lets the user drag and resize controls." That seems to be a perfect candidate for our job. It is the Thumb control and here is how we are going to use it:

```
public class MoveThumb : Thumb
{
    public MoveThumb()
    {
        DragDelta += new DragDeltaEventHandler(this.MoveThumb_DragDelta);
    }

    private void MoveThumb_DragDelta(object sender, DragDeltaEventArgs e)
    {
        Control item = this.DataContext as Control;

        if (item != null)
        {
            double left = Canvas.GetLeft(item);
            double top = Canvas.GetTop(item);

            Canvas.SetLeft(item, left + e.HorizontalChange);
            Canvas.SetTop(item, top + e.VerticalChange);
        }
    }
}
```

The MoveThumb is inherited from Thumb and it provides just an implementation of the DragDelta event handler. Within the event handler, first the DataContext is cast to a ContentControl and then its position is updated according to the horizontal and vertical drag change. You may have already guessed that the control retrieved from the DataContext is our DesignerItem, but where does it come from? You can find the answer if you look at the updated DesignerItem's template:

```
 <ControlTemplate x:Key="DesignerItemControlTemplate" TargetType="ContentControl">
  <Grid>
    <s:DragThumb DataContext="{Binding RelativeSource={RelativeSource TemplatedParent}}"
                 Cursor="SizeAll"/>
    <ContentPresenter Content="{TemplateBinding ContentControl.Content}"/>
  </Grid>
</ControlTemplate>
```

Here you see that the MoveThumb's DataContext property is bound to the templated parent, which is of course our DesignerItem. Note that we have added a Grid as the layout panel for the template, which allows both the ContentPresenter and the MoveThumb to take in the complete DesignerItem's real estate. Now we can compile and run the code.



As a result, we get a blue ellipse on top of a gray MoveThumb. If you play around with it, you will notice that you can actually grab and drag the object, but only where the gray MoveThumb is visible. That's because the ellipse hinders the mouse events to make its way through to the MoveThumb. We can easily change this behaviour by setting the IsHitTest property of the ellipse to false.

```
 <Ellipse Fill="Blue" IsHitTestVisible="False"/>
```

The MoveThumb has inherited its style from the base Thumb class, which is not really appealing in our case. For this, we create a new template consisting of a transparent rectangle only. A more general solution would be to create a default style for the MoveThumb class, but for the moment a customized template will do.

Now the DesignerItem's control template looks like this:

```
 <ControlTemplate x:Key="MoveThumbTemplate" TargetType="{x:Type s:MoveThumb}">
  <Rectangle Fill="Transparent"/>
</ControlTemplate>

<ControlTemplate x:Key="DesignerItemTemplate" TargetType="Control">
   <Grid>
     <s:MoveThumb Template="{StaticResource MoveThumbTemplate}"
        DataContext="{Binding RelativeSource={RelativeSource TemplatedParent}}"
        Cursor="SizeAll"/>
     <ContentPresenter Content="{TemplateBinding ContentControl.Content}"/>
   </Grid>
</ControlTemplate>
```

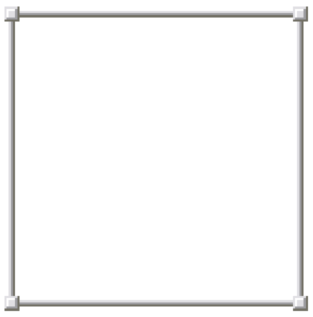
That's all we need to move items on a canvas, now I will show you how to resize objects.

Resize

You remember that the MSDN documentation promised that the Thumb control would let the user drag and resize controls? So, we stick with the Thumb control and build a control template, named ResizeDecoratorTeamplate:

```
<ControlTemplate x:Key="ResizeDecoratorTemplate" TargetType="Control">
 <Grid>
   <Thumb Height="3" Cursor="SizeNS" Margin="0 -4 0 0"
          VerticalAlignment="Top" HorizontalAlignment="Stretch"/>
   <Thumb Width="3" Cursor="SizeWE" Margin="-4 0 0 0"
          VerticalAlignment="Stretch" HorizontalAlignment="Left"/>
   <Thumb Width="3" Cursor="SizeWE" Margin="0 0 -4 0"
          VerticalAlignment="Stretch" HorizontalAlignment="Right"/>
   <Thumb Height="3" Cursor="SizeNS" Margin="0 0 0 -4"
          VerticalAlignment="Bottom"  HorizontalAlignment="Stretch"/>
   <Thumb Width="7" Height="7" Cursor="SizeNWSE" Margin="-6 -6 0 0"
          VerticalAlignment="Top" HorizontalAlignment="Left"/>
   <Thumb Width="7" Height="7" Cursor="SizeNESW" Margin="0 -6 -6 0"
          VerticalAlignment="Top" HorizontalAlignment="Right"/>
   <Thumb Width="7" Height="7" Cursor="SizeNESW" Margin="-6 0 0 -6"
          VerticalAlignment="Bottom" HorizontalAlignment="Left"/>
   <Thumb Width="7" Height="7" Cursor="SizeNWSE" Margin="0 0 -6 -6"
          VerticalAlignment="Bottom" HorizontalAlignment="Right"/>
  </Grid>
</ControlTemplate>
```

Here you see a control template that consists of a grid filled up with a bunch of 8 Thumb controls, which should work as resize handles. By setting the Thumb properties like we did above, we achieved a layout that results in something that looks like a real resize decorator:



Amazing, isn't it. But so far it is only a fake, because there is no event handler that would handle the DragDelta events of the Thumbs. For this, we replace the Thumb objects by ResizeThumbs:

```
public class ResizeThumb : Thumb
{
    public ResizeThumb()
    {
        DragDelta += new DragDeltaEventHandler(this.ResizeThumb_DragDelta);
    }

    private void ResizeThumb_DragDelta(object sender, DragDeltaEventArgs e)
    {
        Control item = this.DataContext as Control;

        if (item != null)
        {
            double deltaVertical, deltaHorizontal;

            switch (VerticalAlignment)
            {
                case VerticalAlignment.Bottom:
                    deltaVertical = Math.Min(-e.VerticalChange,
                        item.ActualHeight - item.MinHeight);
                    item.Height -= deltaVertical;
                    break;
                case VerticalAlignment.Top:
                    deltaVertical = Math.Min(e.VerticalChange,
                        item.ActualHeight - item.MinHeight);
                    Canvas.SetTop(item, Canvas.GetTop(item) + deltaVertical);
                    item.Height -= deltaVertical;
                    break;
                default:
                    break;
            }

            switch (HorizontalAlignment)
            {
                case HorizontalAlignment.Left:
                    deltaHorizontal = Math.Min(e.HorizontalChange,
                        item.ActualWidth - item.MinWidth);
                    Canvas.SetLeft(item, Canvas.GetLeft(item) + deltaHorizontal);
                    item.Width -= deltaHorizontal;
                    break;
                case HorizontalAlignment.Right:
                    deltaHorizontal = Math.Min(-e.HorizontalChange,
                        item.ActualWidth - item.MinWidth);
                    item.Width -= deltaHorizontal;
                    break;
                default:
                    break;
            }
        }

        e.Handled = true;
    }
}
```

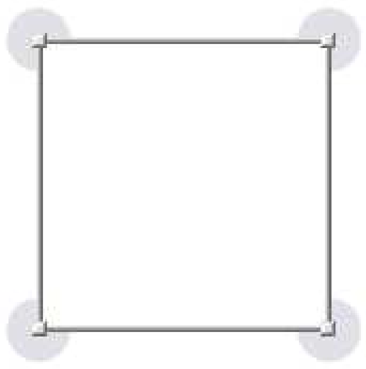
The `ResizeThumbs` just update the `DesignerItem`'s width, height and/or position, depending on the `ResizeThumb`'s vertical and horizontal alignment. Now let's integrate the resize decorator into the `DesignerItem`'s control template by adding a `Control` object with the `ResizeDecoratorTemplate`.

```
<ControlTemplate x:Key="DesignerItemTemplate" TargetType="ContentControl">
  <Grid DataContext="{Binding RelativeSource={RelativeSource TemplatedParent}}">
    <s:MoveThumb Template="{StaticResource MoveThumbTemplate}" Cursor="SizeAll"/>
    <Control Template="{StaticResource ResizeDecoratorTemplate}"/>
    <ContentPresenter Content="{TemplateBinding ContentControl.Content}"/>
  </Grid>
</ControlTemplate>
```

Perfect, now we can move and resize objects. Next comes rotation of objects.

Rotate

To provide rotation of objects, we follow the same solution path as in the chapter before, but this time we create a `RotateThumb` and arrange four instances of it in a control template named `RotateDecoratorTemplate`. Together with the resize decorator, it looks like this:



The structure of the code for RotateThumb and the RotateDecoratorTemplate is very similar to what we have seen in the chapter before, so I will not list the code here.

Note: My first approach to drag, resize and rotate items was to use WPF's TranslateTransform, ScaleTransform and RotateTransform. But that turned out to be the wrong way, because Transforms in WPF do not really change an object's properties like width or height, WPF Transforms are **just** a rendering issue. So I didn't use TranslateTransform and ScaleTransform to drag and resize items, but I had to use RotateTransform because of no alternative.

DesignerItem Style

For convenience, we wrap the DesignerItem's control template into a style, where we also set various properties like MinWidth, MinHeight and RenderTransformOrigin. A trigger allows us to make the resize and rotate decorator visible only when the item is selected, which is indicated by the attached property Selector.IsSelected.

Note: WPF comes with a class named Selector, which is a control that allows you to select items from among its child elements. I do not make use of this control in this article, but I use the attached Selector.IsSelected property to mimic selection.

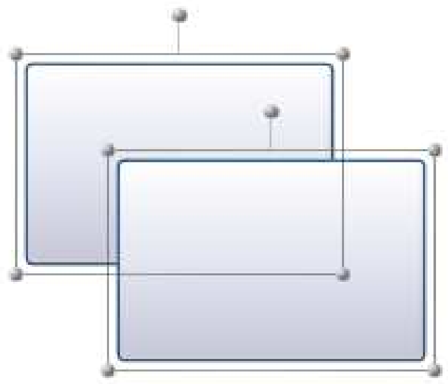
```
<Style x:Key="DesignerItemStyle" TargetType="ContentControl">
 <Setter Property="MinHeight" Value="50"/>
 <Setter Property="MinWidth" Value="50"/>
 <Setter Property="RenderTransformOrigin" Value="0.5,0.5"/>
 <Setter Property="Template">
   <Setter.Value>
     <ControlTemplate TargetType="ContentControl">
       <Grid DataContext="{Binding RelativeSource={RelativeSource TemplatedParent}}">
         <Control x:Name="RotateDecorator"
                  Template="{StaticResource RotateDecoratorTemplate}"
                  Visibility="Collapsed"/>
         <s:MoveThumb Template="{StaticResource MoveThumbTemplate}"
                      Cursor="SizeAll"/>
         <Control x:Name="ResizeDecorator"
                  Template="{StaticResource ResizeDecoratorTemplate}"
                  Visibility="Collapsed"/>
         <ContentPresenter Content="{TemplateBinding ContentControl.Content}"/>
       </Grid>
       <ControlTemplate.Triggers>
         <Trigger Property="Selector.IsSelected" Value="True">
           <Setter TargetName="ResizeDecorator"
               Property="Visibility" Value="Visible"/>
           <Setter TargetName="RotateDecorator"
               Property="Visibility" Value="Visible"/>
         </Trigger>
       </ControlTemplate.Triggers>
```

```
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

That's it. Now we can move, resize and rotate objects. One has to realize that a few lines of XAML code together with three classes provide all we need to do that! Best of all, we don't need to touch the objects themselves: all the behaviour is completely wrapped into a `ControlTemplate`.

Adorner Based Solution

In this chapter, I present a solution that raises the resize and rotate decorators to the `AdornerLayer` so that they are rendered on top of all other items.



The adorner based solution is best explained by showing you the resulting `DesignerItem`'s control template:

```
 <ControlTemplate x:Key="DesignerItemTemplate" TargetType="ContentControl">
  <Grid DataContext="{Binding RelativeSource={RelativeSource TemplatedParent}}">
    <s:MoveThumb Template="{StaticResource MoveThumbTemplate}" Cursor="SizeAll"/>
    <ContentPresenter Content="{TemplateBinding ContentControl.Content}"/>
    <s:DesignerItemDecorator x:Name="decorator" ShowDecorator="true"/>
  </Grid>
  <ControlTemplate.Triggers>
    <Trigger Property="Selector.IsSelected" Value="True">
      <Setter TargetName="decorator" Property="ShowDecorator" Value="true"/>
    </Trigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

This template is similar to what we had in the previous chapter, except that the resize and rotate decorators are replaced by an instance of a new class named `DesignerItemDecorator`. This class is derived from `Control` and has no own default style, instead the class provides an adorner that becomes visible when the boolean `ShowAdorner` property gets `true`.

```
 public class DesignerItemDecorator : Control
{
    private Adorner adorner;

    public bool ShowDecorator
    {
        get { return (bool)GetValue(ShowDecoratorProperty); }
        set { SetValue(ShowDecoratorProperty, value); }
    }
    public static readonly DependencyProperty ShowDecoratorProperty =
        DependencyProperty.Register
            ("ShowDecorator", typeof(bool), typeof(DesignerItemDecorator),
        new FrameworkPropertyMetadata
            (false, new PropertyChangedCallback(ShowDecoratorProperty_Changed)));


    private void HideAdorner()
    {
        ...
    }

    private void ShowAdorner()
    {
        ...
    }

    private static void ShowDecoratorProperty_Changed
        (DependencyObject d, DependencyPropertyChangedEventArgs e)
    {
        DesignerItemDecorator decorator = (DesignerItemDecorator)d;
        bool showDecorator = (bool)e.NewValue;

        if (showDecorator)
        {
            decorator.ShowAdorner();
        }
        else
        {
            decorator.HideAdorner();
        }
    }
}
```

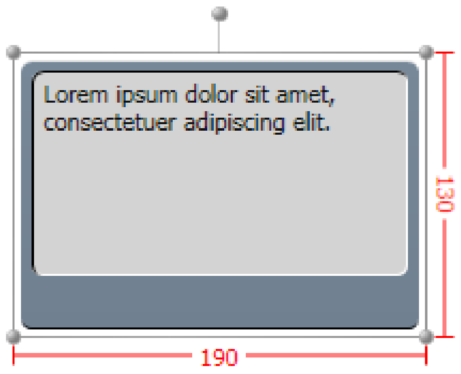
The adorner that becomes visible when the `DesignerItem` is selected is of type `DesignerItemAdorner` and is derived from `Adorner`:

```
 public class DesignerItemAdorner : Adorner
{
    private VisualCollection visuals;
    private DesignerItemAdornerChrome chrome;

    protected override int VisualChildrenCount
    {
        get
        {
            return this.visuals.Count;
        }
    }

    public DesignerItemAdorner(ContentControl designerItem)
         : base(designerItem)
    {
        this.chrome = new DesignerItemAdornerChrome();
        this.chrome.DataContext = designerItem;
        this.visuals = new VisualCollection(this);
    }

    protected override Size ArrangeOverride(Size arrangeBounds)
    {
        this.chrome.Arrange(new Rect(arrangeBounds));
        return arrangeBounds;
    }

    protected override Visual GetVisualChild(int index)
    {
        return this.visuals[index];
    }
}
```

You see that this adorner has a single visual child of type `DesignerItemAdornerChrome`, which is actually the control that provides the drag handles to resize and rotate items. This chrome control has a default style which arranges `ResizeThumb`s and `RotateThumb`s objects in a way that is similar to what we have seen in the previous chapter, so I will not repeat that code here.

Custom Adorners

You can, of course, add your own customized adorners to a `DesignerItem`. As an example, I have added an adorner that displays width and height while resizing an object. For more details, please see the attached code. If you have questions, feel free to ask.

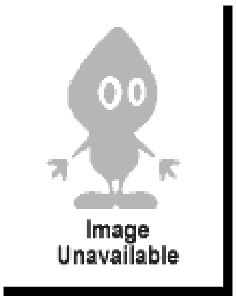


History

- 10th January, 2008 -- Original version
- 18th January, 2008 -- Update: Introduced `ContentControl` as designer item
- 5th February, 2008 -- Update: Added rotation of items
- 22nd August, 2008 -- Update: Added adorner based solution

License

About the Author



sukram

Austria

No Biography provided

Comments and Discussions

120 messages have been posted for this article Visit **http://www.codeproject.com/Articles/22952/WPF-Diagram-Designer-Part-1** to post and view comments on this article, or click **here** to get a print view with messages.