WPF and user Interactivity Part IV: Attached property for moving and resizing shapes

Posted by Bechir Bejaoui in Articles | WPF on April 28, 2010
Tags: Attached property for moving and resizing shapes, wpf, WPF and user Interactivity

In this article, I will refactor the previous code by representing it as an attached property.

Source: http://www.c-sharpcorner.com/uploadfile/yougerthen/wpf-and-user-interactivity-part-iv-attached-property-for-moving-and-resizing-shapes/

Reader Level:

Download Files:

- **Graphics06.zip**

In a previous article, I've shown how to deal with shapes, I mean how to move them and resize them using a couple of mouse event handlers. This article is, in fact, an extension of the previous one; therefore it is highly recommended to take a look on the previous article.

In this article, I will refactor the previous code by representing it as an attached property. The result is as same as the published in the previous article. But this once no business logic is implemented at the code behind level of the window, all business logic is wrapped up within an attached property.

First let's add a class that will hold all the business logic to process shapes either by resizing or moving. Then, let's add an enumeration that will represent either the move or the resize mode. The user could experience exclusively one mode per time interval. The user sets the desired mode through the context menu.

```
namespace Graphics6.AttachedProperties
{
   public class DragDropResizeClass
   {
   }
   public enum Mode {Move,Resize }
}
```

The given class holds two properties, one is a Boolean attached property that will expose all the business logic that enables processing shapes within the canvas or the scene once its value is set to true. If it is set to false then shapes remain static, in other word, the user can't move them or resize them. The other property is static too. It is type of Mode (The above defined enumeration), this one will help to tell the attached property which is the actual defined mode by the user.

First let's define the first property

```
static public Mode ProcessMode { get; set; }
```

It defines the mode either resize or move one.

Second, this is the attached property

```
public static bool GetActivated(DependencyObject obj)
 {return (bool)obj.GetValue(ActivatedProperty);}
public static void SetActivated(DependencyObject obj, bool
value){obj.SetValue(ActivatedProperty, value);}
// Using a DependencyProperty as the backing store for Activated.  This enables animation,
styling, binding, etc...
public static readonly DependencyProperty ActivatedProperty =
          DependencyProperty.RegisterAttached("Activated",
                                              typeof(bool),
                 typeof(DragDropResizeClass), metadata);
```

As you can Remarque, we used our defined metadata, in fact, this Meta data will hold all the business logic, it is composed by two main parts, namely the default value which is false, the second part is the property changed callback delegate.

```
static UIPropertyMetadata metadata = new UIPropertyMetadata(false,

           new PropertyChangedCallback(OnPropertyValueChanged));
```

The property changed callback points to the below method:

```
static void OnPropertyValueChanged(DependencyObject d,
DependencyPropertyChangedEventArgs args)
 {
   Canvas canvas = d as Canvas;
   if ((bool)args.NewValue==true)
   {
   canvas.MouseLeftButtonUp +=
   new MouseButtonEventHandler(canvas_MouseLeftButtonUp);
   canvas.MouseLeftButtonDown +=
   new MouseButtonEventHandler(canvas_MouseLeftButtonDown);
   canvas.MouseMove +=
   new MouseEventHandler(canvas_MouseMove);
   }
    else
    {
             canvas.MouseLeftButtonUp -=
             new MouseButtonEventHandler(canvas_MouseLeftButtonUp);
             canvas.MouseLeftButtonDown -=
             new MouseButtonEventHandler(canvas_MouseLeftButtonDown);
             canvas.MouseMove -=
             new MouseEventHandler(canvas_MouseMove);
    }
 }
```

The above methods enables to subscribe or unsubscribe the three events, namely the mouse left button down, up and the mouse move to help deal with shapes within the scene, the event handler are exposed as follow:

```
static UIPropertyMetadata metadata = new UIPropertyMetadata(false,
          new PropertyChangedCallback(OnPropertyValueChanged));
```

The property changed callback points to the bellow method:

```
static void OnPropertyValueChanged(DependencyObject d,
DependencyPropertyChangedEventArgs args)
 {
   Canvas canvas = d as Canvas;
   if ((bool)args.NewValue==true)
   {
   canvas.MouseLeftButtonUp +=
   new MouseButtonEventHandler(canvas_MouseLeftButtonUp);
   canvas.MouseLeftButtonDown +=
   new MouseButtonEventHandler(canvas_MouseLeftButtonDown);
   canvas.MouseMove +=
   new MouseEventHandler(canvas_MouseMove);

   }
    else
    {
            canvas.MouseLeftButtonUp -=
            new MouseButtonEventHandler(canvas_MouseLeftButtonUp);
            canvas.MouseLeftButtonDown -=
            new MouseButtonEventHandler(canvas_MouseLeftButtonDown);
            canvas.MouseMove -=
            new MouseEventHandler(canvas_MouseMove);
    }
 }
```

The above methods enables to subscribe or unsubscribe the three events, namely the mouse left button down, up and the mouse move to help deal with shapes within the scene, the event handler are exposed as follow:

```
 static void canvas_MouseLeftButtonUp(object sender,
      MouseEventArgs args)
  {
         Canvas Scene = sender as Canvas;
         //First Block
         HitTestResult result = VisualTreeHelper.HitTest(Scene,
                   Mouse.GetPosition(Scene));
         Path path = result.VisualHit as Path;

         //Second block
         if (ProcessMode == Mode.Move)
```

```
{
    if (path.Data.GetType() == typeof(EllipseGeometry))
    {
        EllipseGeometry geometry =
        new EllipseGeometry(new Point(50, 50), 50, 50);
        if ((path.Tag as double[]).Length>0)
        {
            geometry.RadiusX =(path.Tag as double[])[0];;
            geometry.RadiusY =(path.Tag as double[])[1];
        }
        geometry.Transform =
       new TranslateTransform {
       X = CurrentPoint.X - geometry.RadiusX,
       Y = CurrentPoint.Y - geometry.RadiusY };
        Path FinalPath = new Path {
        Fill = Brushes.Red,
        Stroke = Brushes.Black,
        Data = geometry };
        Scene.Children.Add(FinalPath);
        Scene.Children.Remove(path);
    }
    if (path.Data.GetType() == typeof(RectangleGeometry))

    {
        RectangleGeometry geometry =
        new RectangleGeometry(new Rect(0, 0, 150, 100));
        if ((Rect)path.Tag != null)
        {
            geometry.Rect = (Rect)path.Tag;
        }
        geometry.Transform =
        new TranslateTransform {
        X = CurrentPoint.X - geometry.Rect.Width / 2,
        Y = CurrentPoint.Y - geometry.Rect.Height / 2 };
        Path FinalPath = new Path { Fill = Brushes.Blue,
                            Stroke = Brushes.Black,
                            Data = geometry };
        Scene.Children.Add(FinalPath);
        Scene.Children.Remove(path);
    }
}

//Third block
if (ProcessMode== Mode.Resize)
{
    Geometry geometry = path.Data;
    if (path.Data.GetType() == typeof(EllipseGeometry))
    {
        path.Fill = Brushes.Red;
        (geometry as EllipseGeometry).RadiusX =
                (path.Tag as double[])[0];
        (geometry as EllipseGeometry).RadiusY =
                 (path.Tag as double[])[1];
        }
        if (path.Data.GetType() == typeof(RectangleGeometry))
        {
            path.Fill = Brushes.Blue;
           (geometry as RectangleGeometry).Rect =
             (Rect)path.Tag;
        }
    }

}
static void canvas_MouseLeftButtonDown(object sender,
                                       MouseEventArgs args)
{
    Canvas Scene = sender as Canvas;
    //The first block
    HitTestResult result =
    VisualTreeHelper.HitTest(Scene, Mouse.GetPosition(Scene));
    Path path = result.VisualHit as Path;

    //The second block
    if (path.Data.GetType() == typeof(EllipseGeometry))
    {
        Dimensions[0] = (path.Data as EllipseGeometry).RadiusX;
        Dimensions[1] = (path.Data as EllipseGeometry).RadiusY;
        path.Tag = Dimensions;
    }

    if (path.Data.GetType() == typeof(RectangleGeometry))
    {
        rect = (path.Data as RectangleGeometry).Rect;
        path.Tag = rect;
    }

    //The third block
    path.Fill = Brushes.Violet;

    //The fourth block
    if (StartPoint == null)
    {
        StartPoint = args.GetPosition(Scene);
    }
    StartPoint = CurrentPoint;
}
static void canvas_MouseMove(object sender, MouseEventArgs args)
{
    Canvas Scene = sender as Canvas;
    //First block
    CurrentPoint = args.GetPosition(Scene);

    //Second block
    HitTestResult result =
    VisualTreeHelper.HitTest(Scene, Mouse.GetPosition(Scene));
    Path path = result.VisualHit as Path;

    //Third block
    if (ProcessMode == Mode.Move)
    {
        if (Mouse.LeftButton == MouseButtonState.Pressed)
        {
            transform.Matrix =
            new Matrix(1, 0, 0, 1,
            CurrentPoint.X - StartPoint.X,
            CurrentPoint.Y - StartPoint.Y);
            path.RenderTransform = transform;
        }
    }

    //Fourth block

    {
        Geometry geomerty = path.Data;
        if (Mouse.LeftButton == MouseButtonState.Pressed)
        {
            if (geomerty.GetType() == typeof(EllipseGeometry))
            {
                EllipseGeometry currentShape =
               geomerty as EllipseGeometry;
               currentShape.RadiusX =
               CurrentPoint.X - currentShape.Center.X;
               currentShape.RadiusY =
               CurrentPoint.Y - currentShape.Center.Y;
               double[] Dimensions =
               new double[2] {
               currentShape.RadiusX, currentShape.RadiusY };
               path.Tag = Dimensions;
            }
            if (geomerty.GetType() == typeof(RectangleGeometry))
            {
                RectangleGeometry currentShape =
                    geomerty as RectangleGeometry;
                Vector vector =
                    CurrentPoint - currentShape.Rect.Location;
                Rect rect =
                   new Rect(currentShape.Rect.Location, vector);
                currentShape.Rect = rect;
                path.Tag = rect;
            }

        }
    }
}
```

The event handlers are explained in advantage in the previous article. This will lead to the exact result of the previous article. In the next article, I will demonstrate how to deal with the drag and drop activities in more realistic scenario.

Good dotnetting!!!