

How To Implement Custom Cursors In Silverlight 2

Jürgen Baurle

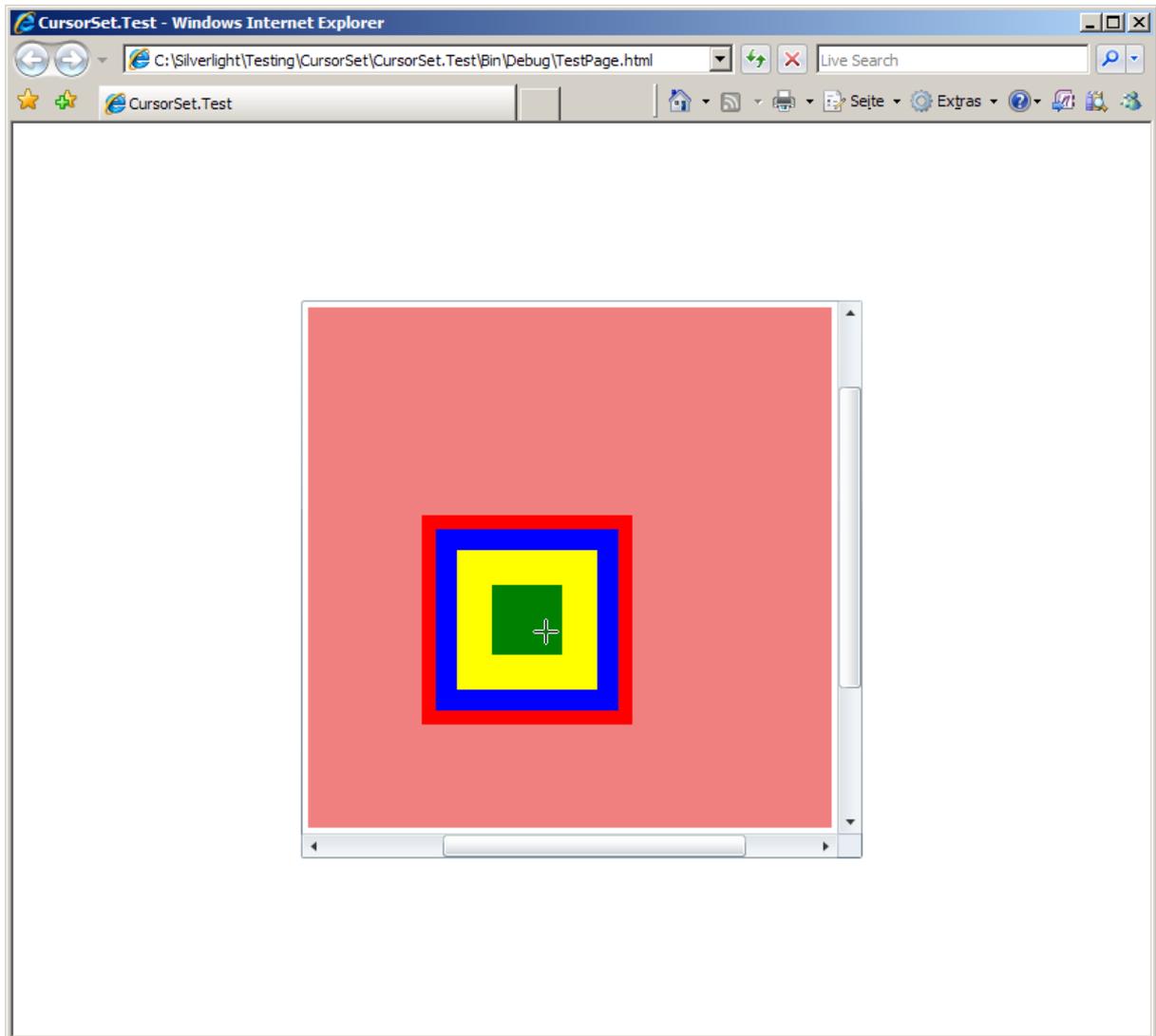
December 2008

Parago Media GmbH & Co. KG

Introduction

Silverlight 2 supports just a small set of default cursor types. WPF (Windows Presentation Foundation) is offering developers an extended set of default cursor types and in addition it is an easy task to create custom cursors for your WPF application.

However it is quite simple to simulate custom cursors in Silverlight 2. There are different implementation approaches to display the cursors. The solution described in this article is based on the Popup class and a fix pre-defined set of custom cursor images, which allows ease of use. Let's start with a screenshot showing a custom cursor type (a cross) on the green rectangle:



The custom cursor implementation is working with a stack of UI elements all using different cursors and within the scroll viewer. The XAML code of the page shown above looks as follows:

```

...
<Grid Background="White">
  <ScrollViewer Width="400" Height="400" ... >
    <Grid>
      <Rectangle parago:CursorSet.ID="SizeAll" Fill="LightCoral" Width="600" ... />
      <Rectangle parago:CursorSet.ID="Cross" Fill="Red" Width="150" Height="150" ... />
      <Rectangle parago:CursorSet.ID="Hand" Fill="Blue" Width="130" Height="130" ... />
      <Rectangle parago:CursorSet.ID="IBeam" Fill="Yellow" Width="100" Height="100" ... />
      <Rectangle parago:CursorSet.ID="Cross" Fill="Green" Width="50" Height="50" ... />
    </Grid>
  </ScrollViewer>
</Grid>
...

```

The code shows that setting the custom cursors is as simple as setting the default and build-in cursors in Silverlight 2. The standard cursor types like Hand or IBeam can also be defined as an ID and will be converted internally to set the Cursor property. So, from developer point of view there is just one technique to use for setting cursors for UI elements and no need to add any additional elements in the XAML code.

How is this implemented? There is a class named CursorSet which implements an attached property called ID. Using an attached property in this implementation allows to query the UI element for cursor settings. The CursorSet class maintains a static list of cursor IDs, a list of active elements (with a cursor ID attached) named ActiveElements, as well as a static popup and canvas instance for displaying the actual cursor image:

```

public class CursorSet
{
    const string ImagePath = "/Resources/Cursors/";

    internal static Popup Popup;
    internal static Canvas AdornerLayer;
    internal static List<string> IDs;
    internal static Dictionary<FrameworkElement, ContentControl> ActiveElements;

    #region public string ID (attached)

    public static string GetID(DependencyObject d)
    {
        return (string)d.GetValue(IDProperty);
    }

    public static void SetID(DependencyObject d, string id)
    {
        d.SetValue(IDProperty, id);
    }

    public static readonly DependencyProperty IDProperty = DependencyProperty.RegisterAttached(
        "ID",
        typeof(string),
        typeof(CursorSet),
        new PropertyMetadata(new PropertyChangedCallback(OnIDPropertyChanged)));

    static void OnIDPropertyChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
    {
        FrameworkElement element = d as FrameworkElement;
        string id = e.NewValue as string;

        if(IsValidID(id))
        {
            Cursor cursor;

            if(IsSystemType(id, out cursor))
            {
                element.Cursor = cursor;
                RemoveMouseEventHandlers(element);

                if(ActiveElements.ContainsKey(element))
                    ActiveElements.Remove(element);
            }
            else
            {
                AddMouseEventHandlers(element);

                ContentControl control = CreateControl(id);
            }
        }
    }
}

```

```

        if(ActiveElements.ContainsKey(element))
            ActiveElements[element] = control;
        else
            ActiveElements.Add(element, control);
    }
}
else
{
    element.Cursor = null;
    RemoveMouseEventHandlers(element);

    if(ActiveElements.ContainsKey(element))
        ActiveElements.Remove(element);
}
}

#endregion

...
}

```

For more information about attached properties in Silverlight see the MSDN documentation or this great article on the SilverlightShow.net website: <http://www.silverlightshow.net/items/Attached-Properties-in-Silverlight.aspx>.

An important part plays the handling of changes to the attached property ID. The method `OnIDPropertyChanged` is called once a cursor ID is newly assigned to an UI element (*parago:CursorSet.ID="Cross"*) or if it has changed. The method is checking in the first place if the defined cursor ID is valid, then it will check if the ID is the name of a default cursor type. If the ID is identified as a default cursor than the method will set the `Cursor` property of the UI element and will remove all eventually existing mouse event handler registrations, otherwise a new instance of the class `ContentControl` will be created for that element with the desired cursor ID as image. The instance will then be added to `ActiveElements` list and the `CursorSet` class will register for the `MouseEnter`, `MouseMove` and `MouseLeave` events of the given UI element.

For each custom ID a PNG image must be existing in the folder defined by the string constant `ImagePath`. Right now there are only two cursors in the sample project available, but it is easy to extend the collection of custom cursors.

Since we need to render a cursor on top of all other elements and independent of the layout of those elements, using the build-in popup class is perfect. The `CursorSet` popup itself contains a canvas element as child to set the cursor image position by coordinates. The size of the popup should fill the Silverlight content area completely. Therefore the `CursorSet` registers the `OnContentResized` method with the `Application.Current.Host.Content.Resized` event and updates the size of the popup and its canvas accordingly.

```

...

static void OnContentResized(object sender, EventArgs e)
{
    if(AdornerLayer != null)
    {
        AdornerLayer.Width = Application.Current.Host.Content.ActualWidth;
        AdornerLayer.Height = Application.Current.Host.Content.ActualHeight;
    }
}

static void EnsurePopup()
{
    if(Popup == null || AdornerLayer == null)
    {
        AdornerLayer = new Canvas()
        {
            IsHitTestVisible = false,
            Width = Application.Current.Host.Content.ActualWidth,
            Height = Application.Current.Host.Content.ActualHeight
        };

        Popup = new Popup
        {
            IsHitTestVisible = false,
            Child = AdornerLayer
        };
    }
}

```

...

The listing above also shows the method `EnsurePopup` which initially creates the popup and canvas instance. Important to note is that the property `IsHitTestVisible` must be set to `false`, in order to avoid conflicting with the underlying elements.

The mouse event handlers for the events `MouseEnter`, `MouseMove` and `MouseLeave` are defined as follows:

...

```
static void OnElementMouseEnter(object sender, MouseEventArgs e)
{
    EnsurePopup();

    FrameworkElement element = sender as FrameworkElement;
    ContentControl control = ActiveElements[element];

    element.Cursor = Cursors.None;
    AdornerLayer.Children.Add(control);

    Point p = e.GetPosition(null);
    Canvas.SetTop(control, p.Y);
    Canvas.SetLeft(control, p.X);

    Popup.IsOpen = true;
}

static void OnElementMouseMove(object sender, MouseEventArgs e)
{
    FrameworkElement element = sender as FrameworkElement;
    ContentControl control = ActiveElements[element];

    Point p = e.GetPosition(null);
    Canvas.SetTop(control, p.Y);
    Canvas.SetLeft(control, p.X);
}

static void OnElementMouseLeave(object sender, MouseEventArgs e)
{
    FrameworkElement element = sender as FrameworkElement;
    ContentControl control = ActiveElements[element];

    element.Cursor = null;
    AdornerLayer.Children.Remove(control);

    Popup.IsOpen = false;
}

...
```

Within the `MouseEnter` event handler `OnElementMouseEnter`, at first the code will ensure the popup instance exists and is setup by calling the `EnsurePopup` method (see code above). Since the event handlers are registered only for custom cursors and not for the default and built-in cursors, the method will get the `ContentControl` instance from the `ActiveElements` list (generated when the property `ID` is set) and will add it to the canvas named `AdornerLayer` of the popup. Next the position of the cursor image will be adapted to the current mouse pointer position and the `Cursor` property of the UI element will be set to the value `None` to avoid multiple cursors. Then the popup will be displayed.

The `MouseMove` event handler `OnElementMouseMove` just will update the position of the cursor image within the popup canvas accordingly to the current mouse position. Once the mouse pointer leaves the UI element, the `ContentControl` with the cursor image will be removed from the popup canvas and the popup itself will be closed.

That's it.

Summary

In overall its easy to create new and custom cursors for your Silverlight 2 application. Just copy the files from the sample project in your project and start using it.

Contact Information

If you have any feedback or suggestions, please feel free to contact me:

Jürgen Bäurle

jbaurle@parago.de

<http://www.parago.de/jbaurle>

Parago Media GmbH & Co. KG

Im Wengert 3 | 71336 Waiblingen, Germany | Phone +49.7146.861803 | Internet <http://www.parago.de>