Real-time rotation of the Windows Phone 8 Map Control

This article explains how to rotate the Windows Phone 8 Map Control object in real-time. The provided solution uses the Touch class to react on FrameReported events. It also shows a relatively simple way to test multi-touch behavior using the Windows Phone 8 emulator.

Motivation

Windows Phone 8 comes with a new Maps framework, and provides extended functionality compared to the Windows Phone 7.1 Maps components. One of these new features is that the developer can set the **Heading** property of the Map control. This property is described on the Maps and navigation for Windows Phone 8 page as: *This parameter specifies the directional heading that is pointing **up** on the map. It is represented in geometric degrees by a value that is between 0 and 360, indicating the number of degrees to rotate the map.*

I found this new feature very interesting and expected that soon the maps will be **rotatable**. In the end it turned out that it won't be the case...

Another reason to pick this topic is that I think there should be a lot more applications out there using multiple touch points. I was curious how simple it is to develop an app which can handle multiple touches simultaneously.

Preparation

In this section I list some of the new features of the new Maps API, then describe the steps to set up the development environment in order to be able to test multi-touch in the emulator and finally show the "theoretical" method, which will work for most of the `UIElement` objects, except for the Map control.

If you already own a Windows Phone 8 device and you are only curious about the map rotation, scroll to the **Solution** part.

New features in Windows Phone 8 Maps API

First of all, a subjective and short list of a few new features in the Location framework I found interesting:

- Improved ability for one-shot location acquisition
- Map control, Map tasks can make use of the downloaded offline maps (no data connection required)
- Background task support for location tracking applications
- "forcing" custom-style pushpins

Since these features are covered quite well in various tutorials in the internet, I will concentrate only on the map view setting feature.

Tip: For a comprehensive introduction to the new capabilities of the Location API, watch this excellent video from channel 9 presented by Microsoft Technical Evangelist Andy Wigley.

How to test multi-touch in the emulator

If you don't have a device you can test multi-touch behavior on the emulator - see How to test multi touch in the Windows Phone Emulator. Be warned, it will take some time to get used to.

How the easy way should look like...

At this point we are ready to implement our Windows Phone app with multi-touch. For this, I have created a basic sample app, following the instructions from the How to handle manipulation events for Windows Phone msdn page.

The idea was simple, I assign a RotateTransform object to my rectangle, and catch the PinchManipulation event's contact points' location. When the event starts (i.e. two fingers touch the display), I save the initial points as the base line, then on every manipulation delta, I compute the angle between the original line and the current line.

Initialization:

```
public MainPage()
{
    InitializeComponent();

    this.ManipulationDelta += this.PhoneApplicationPage_ManipulationDelta;
    this.ManipulationCompleted += this.PhoneApplicationPage_ManipulationCompleted;

    TransformGroup transformGroup = new TransformGroup();
    // I removed scale and move transform, as they were out-of-scope for me
    this.rotate = new RotateTransform();

    transformGroup.Children.Add(this.rotate);
    rectangle.RenderTransform = transformGroup;
}
```

The manipulation handler methods:

(C) Copyright Nokia 2013. All rights reserved.

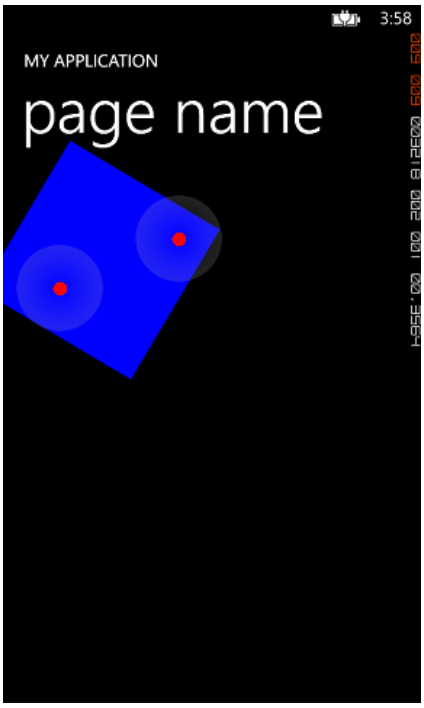
```
        private void PhoneApplicationPage_ManipulationCompleted(object sender, ManipulationCompletedEventArgs e)
        {
            lastAngle = this.rotate.Angle;
        }

        void PhoneApplicationPage_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
        {
            if (e.PinchManipulation != null)
            {
                double angle = angleBetween2Lines(e.PinchManipulation.Original, e.PinchManipulation.Current);
                this.rotate.Angle = lastAngle-angle;
            }
        }
```

I store the last angle in a separate field, in order to know where to start from when the user decides to rotate the object once again. Otherwise it would always "jump" back to its original position, because initially the difference between the base and current lines is 0.

Finally, the method to calculate the angle between the point sets (the formula can be found on the internet, it is basic trigonometry):

```
        public static double angleBetween2Lines(PinchContactPoints line1, PinchContactPoints line2)
        {
            if (line1 != null && line2 != null)
            {
                double angle1 = Math.Atan2(line1.PrimaryContact.Y - line1.SecondaryContact.Y,
                                           line1.PrimaryContact.X - line1.SecondaryContact.X);
                double angle2 = Math.Atan2(line2.PrimaryContact.Y - line2.SecondaryContact.Y,
                                           line2.PrimaryContact.X - line2.SecondaryContact.X);
                return (angle1 - angle2) * 180 / Math.PI;
            }
            else { return 0.0; }
        }
```

With these changes, I am able to rotate my rectangle by "pinching" and rotating the emulator display:



Tip: This simple method can be applied to any type of UIElement objects supporting Manipulation events. Try out to move the name tag to the "page name" text's object in the xaml code. The rotation will be applied on this text without any modification in the C# code.

Unfortunately, this method can't be applied to the Map Control object. I guess the manipulation is marked as completed as soon as the zoom value is calculated and set. So to achieve my goal, I needed to dig a bit deeper...

The solution

To calculate the "delta angle", I need two set of coordinates: one defining the base line and one for the updated one. The .Net API for Windows Phone provides an application-level service that processes touch input from the operating system and raises the Windows Phone-specific FrameReported event. The class is called Touch.

I will catch the FrameReported events and store the first two "touch points" (finger coordinates) in a map (Dictionary). As soon as I register the second finger's position (*TouchAction.Down* event), I calculate the base line, and keep it until both fingers are moving (*TouchAction.Move*). My custom manipulation is ending when one of the fingers report a *TouchAction.Up* event. To make my life easier I save the start coordinates and the current ones as *System.Windows.Shapes.Line* objects (the touch points of the two fingers define a line).

In the "move" phase, I simply use the slightly-modified version of the angleBetween2Lines function to calculate the angle and apply the calculated value to the map object. The rotate center is the center of the map at the point of the start of the event.

For the demonstration I used the Simple Map Control Sample from MSDN. I extended the code as follows:

(C) Copyright Nokia 2013. All rights reserved.

Step 1 Registering the event handler:

```
        public MainPage()
        {
            InitializeComponent();
            // This is the only line I need to add, the rest is from the sample app from the SDK
            rotationHelper = new RotationHelper(sampleMap);

            // Create the localized ApplicationBar.
            BuildLocalizedApplicationBar();

            // Get current location.
            sampleMap.ZoomLevel = 10;
            GetLocation();
        }
```

The event handler logic is implemented in a helper class:

```
        public RotationHelper(Map rotatedMap)
        {
            this.rotatedMap = rotatedMap;

            Touch.FrameReported += this.Touch_FrameReported;
        }
```

Step 2 Handling the FrameReported event:

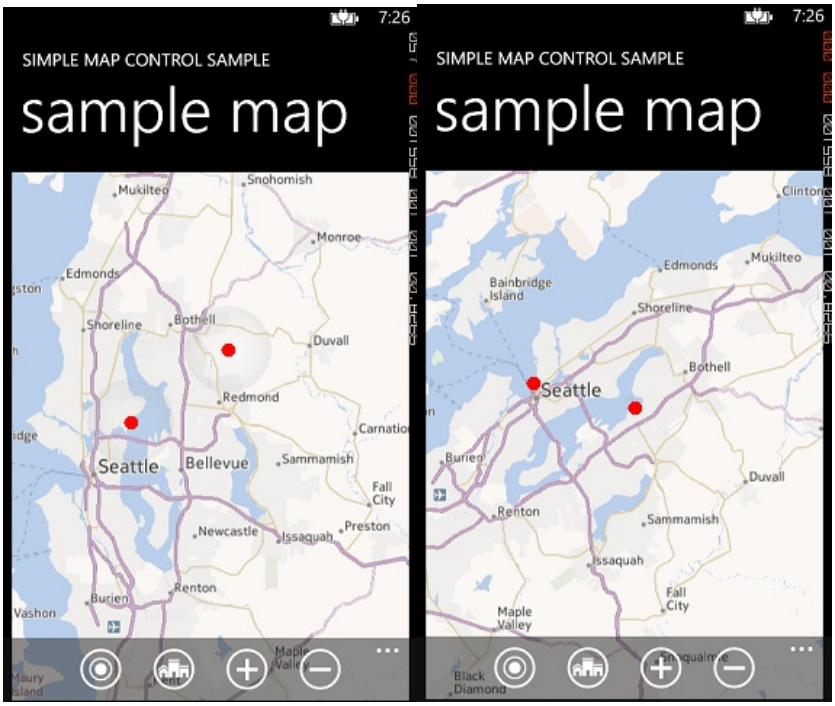
```
        public void Touch_FrameReported(object sender, TouchFrameEventArgs e)
        {
            bool moveOccurred = false;
            // this array contains all available touch devices, i.e. finger coordinates
            var pts = e.GetTouchPoints(rotatedMap);

            foreach (TouchPoint p in pts)
            {
                if (p.Action == TouchAction.Down)
                {
                    // I save the touch points into the startTouchPoints and deltaTouchPoints dictionaries
                    if (!startTouchPoints.ContainsKey(p.TouchDevice.Id))
                    {
                        startTouchPoints.Add(p.TouchDevice.Id, p);
                        deltaTouchPoints.Add(p.TouchDevice.Id, p);
                    }
                    // if I happen to have the second touch point, I save the base line and rotation center
                    if (startTouchPoints.Count > 1)
                    {
                        startLine = calculateLine(startTouchPoints);
                        Debug.WriteLine("Start line set to: {0:f}, {1:f}, {2:f}, {3:f}", startLine.X1, startLine.Y1, startLine.X2, s
                        rotateCenter = this.rotatedMap.Center;
                    }
                }
                else if (p.Action == TouchAction.Move)
                {
                    TouchPoint oldPositionTouchpoint;
                    // updating the touchpoints set
                    if (deltaTouchPoints.TryGetValue(p.TouchDevice.Id, out oldPositionTouchpoint))
                    {
                        deltaTouchPoints[p.TouchDevice.Id] = p;
                    }
                    // if we are already rotating, update the deltaLine
                    if (deltaTouchPoints.Count == 2)
                    {
                        deltaLine = calculateLine(deltaTouchPoints);
                        Debug.WriteLine("Delta line set to: {0:f}, {1:f}, {2:f}, {3:f}", deltaLine.X1, deltaLine.Y1, deltaLine.X2, d
                        moveOccurred = true;
                    }
                }
                // if the user lifts up one finger, we abort the manipulation and reset the values
                else if (p.Action == TouchAction.Up)
                {
                    startTouchPoints.Remove(p.TouchDevice.Id);
                    deltaTouchPoints.Remove(p.TouchDevice.Id);
                    startLine = null;
                    deltaLine = null;
                    moveOccurred = false;
                    this.lastSavedAngle = this.rotatedMap.Heading;
                }
            }
            // if we are in a move action, we should update the map
            if (moveOccurred)
            {
                double angle = angleBetween2Lines(startLine, deltaLine);
                Debug.WriteLine("Angle difference to startline: " + angle);
                this.rotatedMap.SetView(rotateCenter, this.rotatedMap.ZoomLevel, lastSavedAngle + angle, MapAnimationKind.Parabolic)
            }
        }
```

Please note the SetView function, which is used to update the map behavior on the fly. For the different options, you should check the Map class documentation (10 overloaded methods!).

In this example, I define only a new Heading for the map plus the animation type "Parabolic", leaving the map's current center and zoom level unchanged.

The test in the emulator looks like this:

(C) Copyright Nokia 2013. All rights reserved.



Conclusion

In this short tutorial I presented you a method to rotate almost any type of UIElement object in your Silverlight application based on multi-touch user input, and also shown how to do the same with the Maps control element introduced in Windows Phone 8. If you find any place for further "optimization", any problems in the code, or you have any questions regarding it, please add a comment to the page.

Keep in mind that the FrameReported event is known-to-be a performance killer, so use it with care (just check the "App performance considerations" article listed below). Hopefully we don't need to wait too much and the element itself will provide this option for all location-based app developers.

Thanks for reading!

Source Code

Please find test projects here: File:Basic UIElement rotation.zip, File:Map control rotation.zip Please note, that I'm "just" a java developer, so don't expect 100% clean C# code... and also forgive me for using the generated names at some places.

References

Although I was trying to link all external pages to my article, here are once again those which I found to be the most useful, extended by some "extra", which I can really recommend to take a look at. Tools, articles, documentation I used while working on this "project":

- Windows Phone Multi-Touch Manipulation
- How to handle manipulation events for Windows Phone
- Building Apps for Windows Phone 8 Jump Start
- Building Apps for Windows Phone 8 Jump Start-Maps and Location in Windows Phone 8
- App performance considerations for Windows Phone
- Windows Phone 8 Maps Control API
- Touch class
- FrameReported event

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.