Advanced Touch Gestures API Overview, from iOS* to Windows* 8 Store Apps

Objective

Developers looking to port their existing iOS* apps to Windows* 8 Store Apps face several challenges. One of these challenges is porting existing touch detection code. In this article, we use a simple photo viewer as the preexisting iOS application, and we create a similar app on the Windows 8 side. Other application design models such as games are not discussed here.

We will also provide an overview of the touch API differences between the two platforms and show you how to port your apps. Specifically, we discuss how to port tap, swipe, pinch/zoom, and rotation gestures across the platforms. C# is used as the Windows 8 programming language.
Table of Contents

1. Introduction
2. Starting with a Completed iOS Touch App
3. A Touch API Mapping Table (High Level)
4. Porting Tap Gestures
5. Porting Swipe Gestures
6. Porting Pinch/Zoom Gestures
7. Porting Rotation Gestures
8. Summary
9. Appendix A

1. Introduction

Ultrabook™ devices, tablets, phones, and other touch-enabled devices have emerged in the mobile computing market. These devices support an innovative software usage model: apps that respond to user touch input. Today, touch-enabled apps are commonplace; users can browse the web, make purchases, and do so many other things with the use of simple swipe and drag gestures. Of course, this is all made possible with great application design practices in mind.

This article provides an overview of porting pre-existing iOS touch code to Windows 8. From a design perspective, developers should be cognizant of the fact that touch gestures are indistinguishable across platforms. For example, if a user swipes an iPad* screen or performs a two-finger rotation gesture, the gestures would be performed the same way on an Ultrabook device. OS and application responses to touch (such as Charms in Windows 8) may differ across platforms, and we will describe these in detail below.

While the primary programming language for iOS apps is Objective-C*, Windows 8 offers several options such as Visual Basic*, C#, C++, etc. Here, the programming language of choice will be C#. While several gestures exist, this article covers porting guidelines for the following: swipe, pinch/zoom, and rotation. At a high level, the end user doesn’t perceive these gestures to be different when comparing iOS to Windows 8 apps, but programmatically, there are quite a few differences. The details are discussed below.

2. Starting with a Completed iOS Touch App

This article assumes that you are porting a pre-existing iOS app to Windows 8. For example, let’s call it PhotoGesture. A screenshot is presented below:
The bottom button toggles the gesture detection mode. The available modes are: rotation, pinch/zoom, and swipe. For swipe, the text to the right informs the user about the direction of the last swipe detected (up, down, left, right). Thus, not only does this app support single finger swipe, but also two finger rotation and two finger pinch/zoom as well.

This article assumes that you are already familiar with creating an app as shown above. In case you need a primer, here are the prerequisites for this article:


- Overview of the various UIGestureRecognizer classes with code examples for gesture detection: http://www.techotopia.com/index.php/Identifying_iPhone_Gestures_using_iOS_6_Gesture_Recognizers


### 3. A Touch API Mapping Table (High Level)

In Table 3.1 we show a high level comparison between iOS and Windows 8 touch APIs. The table isn’t all inclusive. For more information, refer to the previous section for more iOS APIs or below for links pertaining to Windows 8.

**Table 3.1: Touch API Mapping Table (High Level)**

<table>
<thead>
<tr>
<th>Gesture(s)</th>
<th>API Family (iOS)</th>
<th>API Family (Windows 8)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Tap</td>
<td>Action, Outlet</td>
<td>Click, OnTapped (XAML), pointer events+, manipulation events+</td>
<td>The most basic gesture considered to be a discrete event</td>
</tr>
<tr>
<td>Swipe</td>
<td>UISwipeGestureRecognizer</td>
<td>Pointer events, manipulation events+</td>
<td>Considered a sequence of press, drag, release actions</td>
</tr>
<tr>
<td>Two finger rotation</td>
<td>UIRotationGestureRecognizer</td>
<td>Manipulation events</td>
<td>Multi touch gesture</td>
</tr>
<tr>
<td>Two-finger pinch/zoom</td>
<td>UIPinchGestureRecognizer</td>
<td>Manipulation events</td>
<td>Another multi touch gesture</td>
</tr>
</tbody>
</table>

*+*: Optional

The following Windows 8 guide provides the framework for all touch APIs that will be discussed in the remainder of this article:


This guide provides great supplemental information on Windows 8 touch by providing an example children's math game implementation:

Single tap and swipe gestures only require one finger and are thus easier to implement. For single finger tap, iOS uses both target-action and outlet design patterns. The practice is to use one or both depending on the UI element semantics. For example, a button only needs an action, whereas a textbox would use an outlet. Correspondingly, on the Windows 8 side, there are three ways of handling single tap as shown in the table above. The easiest way is to use a pre-defined XAML keyword such as onTapped. Pointer events can also be used with finger press and release corresponding to separate event triggers, although they are not needed for simple tap detection. Manipulation event implementation is similar to implementing pointer events in the most basic form of press/release. The press would correspond to ManipulationStarted while the release would correspond to ManipulationCompleted. Of course, this isn’t required for handling simple tap events.

For swipe, note that it’s not discrete since dragging is continuous. iOS has a predefined UISwipeGestureRecognizer class for handling this. For Windows 8, a single pre-defined XAML keyword for tap detection won’t suffice. At a minimum, we need two or more XAML keywords utilizing either pointer events or at a deeper level, manipulation events. Refer to this link for more information:


Similarly, manipulation events have keywords.

A note regarding pointer events: don’t assume that when a pointer event completes that it fires PointerPressed. There are many factors that govern which event fires such as which device is being used. This will be discussed in the “Porting Swipe Gestures” section below. For now, please refer to this link for more information:


Windows 8 Manipulation events are considered to be the most advanced APIs in that we must use these for handling multi touch gestures. While the table above provides predefined multi-touch APIs for detecting rotation and pinch/zoom in iOS, we get to handle these manually with manipulation events in Windows 8. This is discussed in more detail below.

The following sections dive into the porting exercises. A sample Windows 8 app, SensorDemo, is used in conjunction with PhotoGesture mentioned above. For iOS, the storyboard designer is used in the code sections below.

### 4. Porting Tap Gestures

In our example, the tap gesture is for the bottom button shown in Figure 1.1. For iOS, designing a button starts with the storyboard. After setting up the target action for the button, the corresponding view controller header file’s code for the iOS side looks like the following:

```
//for button that changes selected gestures
-(IBAction)modeChanged:(UIButton *)sender;
```
When the user clicks the button, the following implementation in the .m file handles the event:

```objective-c
-(IBAction)modeChanged:(UIButton *)sender {
    //handler code here...
}
```

**Figure 4.1: Button for Mode Selection**

On the Windows 8 C# side, design begins with XAML. The developer opens the Toolbox and then drags a button into the design pane. The framework then auto-populates the .xaml file with the button specification. In this example, the `click` keyword is then manually added to the XAML code in order to specify the handler that manages button click/tap events:

```xml
<Button x:Name="btnMode" Content="Mode Button" HorizontalAlignment="Left" Height="66" VerticalAlignment="Top" Width="270" Click="ButtonToggleFiltering" Foreground="White" Background="RoyalBlue" Canvas.Left="10" Canvas.Top="692" Margin="10,692,0,0" Style="{StaticResource MyButtonsStyle}"/>
```

**Figure 4.2: Mode Change Handler**

Assuming the XAML file name is `file.xaml`, you can then add the click handler code to `file.xaml.cs` as follows:

```csharp
private void ButtonToggleFiltering(object sender, RoutedEventArgs e) {
    //handler code here
}
```

**Figure 4.3: Button Specification in XAML**

**Figure 4.4: Click Handler**

### 5. Porting Swipe Gestures

The following screen shot is taken from the iOS sample app side:
Four distinct swipe gesture recognizers are used in iOS since any given swipe gesture recognizer can only detect at most one swipe direction. Thus, the collection of recognizers is used here to detect the up, down, left, and right swipe directions.

Here is a snippet from the iOS view controller header file:

```cpp
// we use one swipe gesture instance per direction we wish to detect
// since one recognizer instance can only detect one direction of
// choice
- (IBAction)onSwipeUp:(UISwipeGestureRecognizer *)sender;
- (IBAction)onSwipeLeft:(UISwipeGestureRecognizer *)sender;
- (IBAction)onSwipeRight:(UISwipeGestureRecognizer *)sender;
- (IBAction)onSwipeDown:(UISwipeGestureRecognizer *)sender;
```

Then, the implementation for one of the handlers looks like the following in the .m file:

```cpp
- (IBAction)onSwipeUp:(UISwipeGestureRecognizer *)sender {
    if(mode == 2) {
        direction = sender.direction;
        if(direction == UISwipeGestureRecognizerDirectionUp) {
            last_swipe.text = @"Up";
        }
    }
}
```

Notice how, like the previous example, a “sender” object is used, and in this case, the `direction` property allows for proper swipe detection.

While four swipe recognizers were needed on the iOS side, this isn’t the case for Windows 8. Once again, we start with the XAML design. In this sample, pointer events are used. Recall that in section 3,
a note of caution is made as to which keywords to use. Since events such as `PointerPressed` aren’t always guaranteed to fire upon swipe, here, we simply use all possible keywords (highlighted below) for handling pointer events in a robust, platform-neutral way. This ensures that we don’t miss handling events. It’s up to you to determine which subset of keywords to use based on the platform at hand:

```xml
<Image x:Name="imageToRotate" Source="Assets/Ultrabook-Arrow.png" HorizontalAlignment="Center" Stretch="None" VerticalAlignment="Center" ManipulationMode="All" ManipulationDelta="manip_delta" PointerEntered="pressed" PointerPressed="pressed" PointerCanceled="released" PointerCaptureLost="released" PointerReleased="released" PointerExited="released" Height="682" Canvas.Top="10" Width="922" Canvas.Left="264" Margin="250,-73,194,159"/>

Figure 5.4: Pointer Event Specification in XAML++
```

In this sample, note that an image is used. Also, note that while multiple keywords are used, we share the handler name. There is no necessity to use a separate handler for each keyword, but it may be desired based on the application design.

Here, manipulation mode was specified in a way that all manipulation types are detected. This can instead be limited to rotation, scale, etc. For the complete list of manipulation modes supported, refer to this link: [http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.input.manipulationmodes](http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.input.manipulationmodes)

For Windows 8, the `pressed` and `released` handlers are presented below:

```csharp
void pressed(object sender, PointerRoutedEventArgs e)
{
    if (_currentSensorMode == SensorMode.TOUCH_SWIPE)
    {
        begin_swipe_x = e.GetCurrentPoint(this.imageToRotate).Position.X;
        begin_swipe_y = e.GetCurrentPoint(this.imageToRotate).Position.Y;
    }
}

void released(object sender, PointerRoutedEventArgs e)
{
    if (_currentSensorMode == SensorMode.TOUCH_SWIPE)
    {
        end_swipe_x = e.GetCurrentPoint(this.imageToRotate).Position.X;
        end_swipe_y = e.GetCurrentPoint(this.imageToRotate).Position.Y;
        //let’s determine if there was more of a coordinate change in the
        //x direction or y direction to better
        //choose one of four directions as feedback to the user who has swept across the screen
    }
}
```
bool x_axis = false;

if(Math.Abs(Math.Floor(begin_swipe_x-end_swipe_x)) > Math.Abs(Math.Floor(begin_swipe_y-end_swipe_y)))
	x_axis = true;

if(x_axis && end_swipe_x - begin_swipe_x > MIN_THRESHOLD)
	swipe_status.Text = "RIGHT"; //swipe right
if(x_axis && begin_swipe_x - end_swipe_x > MIN_THRESHOLD)
	swipe_status.Text = "LEFT"; //swipe left
if(!x_axis && end_swipe_y - begin_swipe_y > MIN_THRESHOLD)
	swipe_status.Text = "DOWN"; //swipe down
if(!x_axis && begin_swipe_y - end_swipe_y > MIN_THRESHOLD)
	swipe_status.Text = "UP"; //swipe up
}

Figure 5.5: Handler Code+++ 

Compared to the previous section, the event argument has changed to PointerRoutedEventArgs. The code above first notes the touch coordinates when swiping begins, and then it captures the final coordinates when swiping completes. Using a threshold and axial direction, these two coordinate pairs are then compared to determine the swipe direction. The user must move more than MIN_THRESHOLD along any given axis to register a swipe.

Notice how in the Windows 8 case, you have more flexibility and control for precisely detecting a swipe since the threshold can be specified, etc. Note also that four separate recognizers were not needed. There are also other properties that are easily accessible, such as velocity data.

6. Porting Pinch/Zoom Gestures

We now discuss the pinch/zoom recognizer on the iOS side. Officially, the recognizer is called the "Pinch Gesture Recognizer," but we will refer to it as pinch/zoom gesture recognizer since the recognizer really handles both touch motions.
Here is the view controller code for the iOS side:
- (IBAction)onPinch:(UIPinchGestureRecognizer *)sender;

Here is the corresponding implementation code:

```objc
// for pinch zoom
CGFloat scale = 1.0; // used for pinch/zoom image scale
CGFloat orig_width, orig_height; // original dimensions for image view
CGFloat old_width, old_height; // used for resizing origin change
CGFloat old_origin_x, old_origin_y; // previous origin of image
...

- (IBAction)onPinch:(UIPinchGestureRecognizer *)sender {
  if(mode == 1) {
    // first time through, assuming image size > 0
    if(orig_width == 0 && orig_height == 0) {
      fr = _img.frame; // the encompassing frame for our UIImageView
      orig_width = fr.size.width;
      orig_height = fr.size.height;
    }
    scale = sender.scale; // scale change
    fr = _img.frame;
    // if needed, refer to Appendix A for the details of calculating new
    // origin and dimension of scaled image
    // calculate scale and origin here...
  }
}
```

In this sample code, pinch/zoom is performed on an image. The image frame is first obtained. Then, the `sender.scale` property is read to determine by what scale the user has pinched or zoomed. Given the scale value provided as input into the event handler, we wish to compute the new (x,y) origin denoted by (fr.origin.x, fr.origin.y). For these additional details, please look through Appendix A.
For the Windows 8 side, with no surprise, design starts with XAML (same as the above, but this time highlighting manipulation keyword for emphasis):

```xml
<Image x:Name="imageToRotate" Source="Assets/Ultrabook-Arrow.png" HorizontalAlignment="Center" Stretch="None" VerticalAlignment="Center" ManipulationMode="All" ManipulationDelta="manip_delta" PointerEntered="pressed" PointerPressed="pressed" PointerCanceled="released" PointerCaptureLost="released" PointerReleased="released" PointerExited="released" Height="682" Canvas.Top="10" Width="922" Canvas.Left="264" Margin="250,-73,194,159"/>
```

*Figure 6.4: Manipulation Specification in XAML++*

On the Windows 8 side, since pinch/zoom doesn’t necessitate handling when the event starts or ends, we can simply treat it as a continuous event where *manip_delta* is continuously fired so long as the gesture continues. It is however perfectly acceptable to use the other manipulation phases as discussed in the links above.

Here is the Windows 8 C# code for the routine:

```csharp
//moving image while holding down pointer
void manip_delta(object sender, ManipulationDeltaRoutedEventArgs e)
{
    ... 
    if (_currentSensorMode == SensorMode.TOUCH_PINCH)
    {
        ScaleTransform tran = new ScaleTransform();
        //scale in/out from center of image
        tran.CenterX = imageToRotate.ActualWidth / 2;
        tran.CenterY = imageToRotate.ActualHeight / 2;
        tran.ScaleX = e.Cumulative.Scale;
        tran.ScaleY = e.Cumulative.Scale;
        //update the on-screen image using the transform
        imageToRotate.RenderTransform = tran;
    }
}
```

*Figure 6.5: Manipulation Delta Handler Code+++*

Once again, take note of the change for the event handler type. In this code example, since the transform origin is taken to be the center of the image, we don’t need to do any mathematical tricks to fixate the origin as we needed to in the iOS code above. The `e.cumulative.Scale` property accumulates the overall scale change for this gesture event so long as the user continues it. Thus, this is why it suffices to just use `ManipulationDelta`. 

7. Porting Rotation Gestures

We now move on to the final porting exercise: porting rotation code. Here is the iOS snapshot:

![iOS Sample View Controller Scene](image)

**Figure 7.1: Rotation Gesture Recognizer (Photo source: XCode*)

The iOS view controller header file code follows:

```objective-c
- (IBAction)onRotation:(UIRotationGestureRecognizer *)sender;
```

**Figure 7.2: On Rotation ++++

Here is the corresponding implementation in the .m file:

```objective-c
//for rotation
CGFloat angle = 0.0; //rotation angle for image
CGFloat last_angle = 0.0; //orientation of image at end of last gesture
...

- (IBAction)onRotation:(UIRotationGestureRecognizer *)sender {
    if(mode == 0) {
        angle = sender.rotation;
        //time to apply rotation transform to the image
        CGAffineTransform transformer = CGAffineTransformMakeRotation(last_angle + angle);
        [_img setTransform:transformer];
        if(sender.state == UIGestureRecognizerStateEnded) {
            last_angle = last_angle + angle;
        }
    }
}
```

**Figure 7.3: iOS Sample Implementation+++
The purpose of \texttt{last_angle} is to allow the user to rotate the image some, and then upon rotating the image again, the starting orientation of the image is the ending orientation of the previous rotation event. This is done so that the image doesn’t appear to “jump” between separate rotation events.

Now for the C# side. Instead of adding the manipulation events via XAML, we show the alternative way you can specify the manipulation handlers: by using the C# code-behind method:

\begin{verbatim}
imageToRotate.ManipulationStarted += manip_start;
imageToRotate.ManipulationDelta += manip_delta;
\end{verbatim}

\textbf{Figure 7.4: Specifying Manipulation Events Programmatically} +++

Notice that we continue to specify gesture handlers for the same image as in the previous sections. In this sample, the same “delta” routine is used as in the previous section. However, for the sake of exercise, we also use a separate handler for when the manipulation starts. Here is the rest of the code in the .cs file:

\begin{verbatim}
//the start of the gesture event: touching the image
void manip_start(object sender, ManipulationStartedRoutedEventArgs e)
{
   //there may have been a previous touch event that rotated the
   //image....let's let the orientation of the image at the end
   //of previous event be the start for this one to avoid a
   //frame jump to original upright orientation
   if (_currentSensorMode == SensorMode.TOUCH_ROTATE)
   {
      RotateTransform tran = new RotateTransform();
      tran.Angle = _curAngle;

      //rotate about the center of the image
      tran.CenterX = imageToRotate.ActualWidth / 2;
      tran.CenterY = imageToRotate.ActualHeight / 2;

      //update the on-screen image using the transform
      imageToRotate.RenderTransform = tran;
   }
}

void manip_delta(object sender, ManipulationDeltaRoutedEventArgs e)
{
   if (_currentSensorMode == SensorMode.TOUCH_ROTATE)
   {
      //time to rotate the image!
      RotateTransform tran = new RotateTransform();
      _curAngle += e.Delta.Rotation;
      tran.Angle = _curAngle;

      //rotate about the center of the image
      tran.CenterX = imageToRotate.ActualWidth / 2;
      tran.CenterY = imageToRotate.ActualHeight / 2;

      //update the on-screen image using the transform
      imageToRotate.RenderTransform = tran;
   }
}
\end{verbatim}
//update the on-screen image using the transform
imageToRotate.RenderTransform = tran;
}

Figure 7.5: The Corresponding Windows 8 Handler Code

Unlike the previous Windows 8 code example, here, we use e.Delta.Rotation. Rather than use a cumulative value to directly assign the orientation, we instead keep applying the small angular change that occurs every time the event is fired. The _curAngle variable then tracks the overall angular change relative to the upright (0 angle) orientation. Of course, you don't need to handle the code this way, but we see that Windows 8 provides us with some great flexibility!

8. Summary

This article summarized the essential steps needed to port preexisting iOS touch code to the Windows 8 platform when handling tap, rotate, pinch/zoom, and swipe. The code examples for Windows 8 incorporated both XAML and C# design methodologies, and we showed that Windows 8 has flexibility in terms of the APIs that can be used to solve the porting challenges. We saw a case where handling swipe required only one recognizer for Windows 8. We also saw how with manipulation events, the same event handler can be shared among different gesture types with some logic added to distinguish them (e.g., the gesture mode we are in as shown in the code snippets above). Windows 8 thus allows touch gestures to be extended programmatically from other platforms like iOS to continue providing the end user with a rich user experience!

9. Appendix A

For scaling an image in iOS, the reason we need to compute the new location of a scaled image’s top left corner is because when scaling the image, we want its center to remain fixed. Otherwise, by default, without doing the adjustment, the image will scale from its top left corner, rather than from the center. It was noted above that this adjustment is not needed when scaling images in Windows 8. Here is sample iOS code provided as a reference for calculating a scaled image’s new position and dimensions.

old_origin_x = fr.origin.x;
old_origin_y = fr.origin.y;
old_width = fr.size.width;
old_height = fr.size.height;

//ensure that the center of the image is fixed even when
//rescaled so that it appears to scale from the center out
//rather than a corner
fr.size.width = orig_width * (scale/sqrt(2)); //rescale the image
fr.size.height = orig_height * (scale/sqrt(2));
fr.origin.x = old_origin_x - ((fr.size.width - old_width)/2);
fr.origin.y = old_origin_y - ((fr.size.height - old_height)/2);
Figure 9.1: Adjusting the Result of a Scale

The following picture is a pictorial description of what’s happening in the above code:

Figure 9.2: Scaled Image

Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information
here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to:
http://www.intel.com/design/literature.htm

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel, Ultrabook, and the Intel logo are trademarks of Intel Corporation in the US and/or other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2012 Intel Corporation. All rights reserved.

++This sample source code includes XAML code automatically generated by Visual Studio IDE and is released under the Intel OBL Sample Source Code License (MS-LPL Compatible)

+++This sample source code is released under the Microsoft Limited Public License (MS-LPL)

++++This sample source code is released under the Intel IBL Apple Inc. Software License Agreement for XCode Agreement