Enabling Touch in Windows 8* Metro Style Apps with C#

By David Medawar

Objective

This article discusses how to enable touch for Windows 8* Metro style apps using the C# programming language in the Visual Studio* 11 IDE. Any prior touch API familiarity is sufficient, and this document is straightforward for those new to touch enabling. The APIs listed are current as of the Windows 8 Release Preview version of the operating system and the corresponding Microsoft Visual Studio 2012 RC IDE. The article covers how to bind user touch events using both the Extensible Application Markup Language (XAML) and “code-behind” (C# code) perspectives to create a rich user experience. A children’s math game is used as an example use case for touch.
# Table of Contents

1. Introduction

2. Handling Tap Events in XAML

3. Tap Event Handler Code in C#

4. Pointer Events

5. Manipulation Events

6. Testing Before Device Deployment: VS Emulator

7. Example Use Case: Children’s Math Game

8. A Preview of the Math Logic

9. Using Manipulation to Fling UI Elements

10. Summary
1. Introduction

More and more, touch-enabled applications are becoming commonplace as users demand a "hands on" experience. Windows 8 Metro style apps and the corresponding Visual Studio (VS) IDE align with these new demands by providing developers with touch APIs that enable consistent touch experiences for end users. This paper explains the exciting ways developers can enable touch in Metro style apps by looking at the XAML and code-behind C# perspectives. The combination of UI element XAML markup and C# code-behind bridge the user experience with the magic that occurs behind the scenes.

The source code in this document was written using the Windows 8 Release Preview (8400 build) released May 31st, 2012, Visual Studio Ultimate 2012 RC (11.0.50522.1 RCEL), and .NET Framework version 4.5.50501.

After a walkthrough of touch APIs, this document will show a game example.

2. Handling Tap Events in XAML

Let's say, for example, that the app at hand should do something whenever a user taps a UI element. Further, consider the case of an Image. Starting with the XAML declaration of the image, the respective tap event handler can be defined as follows:

```xml
<Image x:Name="c29" Height="100" Width="100" Tapped="Image_Tapped" Grid.Row="1" Grid.Column="8" />
```

In this example, an image named "c29" is defined, and parameters for its dimensions and grid position are provided. Note the use of:

```
Tapped="Image_Tapped"
```

This essentially indicates that whenever this particular image is tapped, the method Image_Tapped in the supporting C# code-behind will be invoked.

3. Tap Event Handler Code in C#

Using the example image XAML definition in the previous section, multiple UI elements can utilize the specified tap handler method. To do this, runtime code needs to determine which UI element was tapped. Thankfully, FrameworkElement can be used to make this process easy, and the developer code could look like this:
private void Image_Tapped(object sender, RoutedEventArgs e) {
    //find out which game object was clicked
    FrameworkElement source_image = e.OriginalSource as FrameworkElement;
    for (int row = 0; row < ROWS; row++)
    {
        for (int col = 0; col < COLS; col++)
        {
            //we found a match!
            if (source_image.Name.Equals(images[row, col].Name))
            {
                //handler logic
            }
        }
    }
}

Here, an array of images is used. Using the event argument passed into the routine, the developer can fetch the element that was touched and then use its name to find a match in the array.

In addition to Tapped as a keyword in XAML, DoubleTapped can be used as well. Other keywords, such as Clicked, are beyond the scope of this document.

4. Pointer Events

The Tapped keyword was quite simple to use. Now, it is time to peel the onion a bit. Please refer to this link:


Read the section titled "Using pointer events." This method can be used for detecting both touch and mouse inputs. The limitation however is that this API is limited to single finger touch. Take a peek at the next section for multi-touch handling.

Using Pointer is like Tapped in that it is as simple as using a keyword in XAML! The documentation describes the use of XAML keywords PointerPressed, PointerReleased, and PointerExited. Take a look at the MSDN article for an example of how the corresponding code-behind handler could work.

Please pay attention to the event handler signature. Notice that in the code example, PointerRoutedEventArgs is used. In the previous section, RoutedEventArgs was described. These different event handling classes provide developers with the flexibility of distinguishing various touch event types!

5. Manipulation Events
Here, things get really interesting. Manipulation events, not surprisingly, have even more XAML keywords (see the "Using manipulation events" section in the previously mentioned MSDN link).

To simplify things, consider a manipulation event as occurring in the following event phases:

**Phase I: ManipulationStarted.** This is initiated at the start of the manipulation. This happens when, say, the user holds a finger down on a UI element.

**Phase II: ManipulationDelta.** The manipulation continues as the user continues holding the UI element but is performing a drag. Even for simple hold-drag-release sequences, this event can be invoked many times. This takes careful code consideration.

**Phase III: ManipulationInertiaStarting.** This is invoked when the user finally releases all finger(s) from the screen, thus sending the UI element(s) into inertia.

Other manipulation event types are described in the MSDN link. Also note that manipulations can be tested with a mouse as well when using a scroll wheel.

Refer to the MSDN link for keyword usage in XAML. One interesting part about this API is that the developer can retrieve the velocity and change in position of the UI element throughout the manipulation duration. This allows for manipulating the UI element via matrix transformations, also described in the link.

Multi-touch is supported when handling manipulation events, but that is not specifically discussed here. Refer to the MSDN link for more details.

6. **Testing Before Device Deployment: VS Emulator**

After writing the touch detection glue, it is time to test! Manually transferring the app package to a device via wired method after every code build can get tiresome. There are two preferred methods from a productivity perspective either: use the VS Remote Tools (contains Remote Debugger), or simply deploy the code to the VS Emulator. In this paper, the latter method is discussed.

*Tip: In the Debug menu of VS, select the "<Project name> Properties" option in the drop-down bar, and then select the "Debug" option on the left pane. Here, a target device can be selected as either local machine (simulation in VS) or remote machine (wirelessly debug on a real device via Remote Debugging).*

Visual Studio lets the developer add breakpoints to the source code. One debug scenario may be to ensure that the touch event routine satisfies the following two properties: the routine is never entered when no touch event occurs, and the routine
is always entered whenever a touch event occurs. Simply add a breakpoint in VS to the start of the routine glue as follows:

```csharp
//called when one of the game images is clicked
private void Image_Tapped(object sender, RoutedEventArgs e)
{
    //find out which game object was clicked
    FrameworkElement source_image = e.OriginalSource as FrameworkElement;
}
```

After building the project successfully and selecting "Start Debugging" from the Debug menu in VS (or pressing the F5 key), the app starts running. Then, upon entry of the Image_Tapped routine, VS highlights the breakpoint when execution reaches the line:

```csharp
//called when one of the game images is clicked
private void Image_Tapped(object sender, RoutedEventArgs e)
{
    //find out which game object was clicked
    FrameworkElement source_image = e.OriginalSource as FrameworkElement;
}
```

7. Example Use Case: Children’s Math Game

In this section, the concepts described in the previous sections are applied to an example usage scenario: delivering a compelling children's math game. The game idea is as follows. The user is given the option to choose a difficulty level. In every level, the user selects operands (digits) and operators (+,-,\(*,/)\. The goal is to construct an arithmetic sequence that uses both operators and operands to create a sound mathematical equation. If the equation is correct, the tiles disappear; otherwise, nothing happens. The goal is to make all operand tiles disappear. It is possible to get stuck in a level if equations aren't constructed properly! Thus, the game requires thinking ahead.

When the app is launched, the user sees:
If, for instance, the user chooses level 3, the game board XAML is loaded and play begins:

On this game board, a correct equation could be constructed as follows:

\((1+2)-2 = 1\)

Note that the game does not use the formal order of operations—equations are processed from left to right. So, here, the user could choose the following tiles in this order:

1, +, 2, -, 2, 1

The game does not need an equal sign; the game logic tracks the cumulative total as the user continues selecting tiles. The following picture shows the partial selection of all but the final tile. The user simply taps on each tile to make the selection:
The selected tiles are shaded so they appear darker. Now, when the user selects "1", the equation is properly constructed, and the tiles disappear!

There is one problem. The game has no way to arithmetically eliminate all remaining operand tiles. Winning in the original level structure shown above is left as an exercise for the user!

8. A Preview of the Math Logic

The logic for handling equation construction has two main constituents:

Queue: selected tiles are pushed on the queue, and the queue is cleared when needed

Finite State Machine (FSM): maintains state to determine when the equation is completed

The FSM works as follows. The first element of an equation must obviously be an operand. Next, the user must select an operator. Then, an operand must be selected once again. This sequence of operand operator operand occurs at least once and may occur quite often for the more intensive computations. When the state machine detects that two consecutive operands are selected, this marks the end of the equation construction. A portion of the related code is now presented:

```java
//after selecting num in Q0, user must now specify an operator
    case selection_state.Q1:
```
if (v == value.DIV || v == value.MINUS || v == value.MULT || v == value.PLUS) {
    sel_state = selection_state.Q2; // move to next state
    move_queue.Enqueue(v); // add last user operation to the queue
    // set update flag for redraw
    updates_pending[row, col] = true;
}

// else, user selected a number, so we just stay in Q1
else {
    clearSelection();
    move_queue.Clear(); // wipe the moves the user has made
    state[row, col] = SELECTED.YES;
    sel_state = selection_state.Q1;
    move_queue.Enqueue(v); // add last user operation to the queue
    // set update flag for redraw
    updates_pending[row, col] = true;
}
break;

In the sample code, Q0 denotes the first state where the very first operand of the equation is expected. Here we see that in Q1, the code expects the user to select an operator. If so, the selection is queued, and the tile is marked for updating. The updating logic is an array that sets a Boolean for each tile so that on redraw, only marked tiles are updated for performance reasons.

Now if the user selected an operand instead, state does not change. This simply updates the equation such that the operand is now the most recent selection.

9. Using Manipulation to Fling UI Elements

This section is another case study example using the children's math game idea. As mentioned in a preview section, this code considers manipulation to occur in three phases (start, delta, inertia).

The code snippets in this section provide a sample of how a developer can use a "fling" gesture. This involves the user holding down a game tile and "flinging it" across the screen with a finger (multi-touch not discussed).

Recall in the previous section where the math game was discussed. An array of images was elicited as the data type storing several UI elements (images). The following code demonstrates how to define manipulations for these images.
//specify the manipulation events to be invoked for each game tile

for (x = 0; x < images.GetLength(0); x++)
{
    for (y = 0; y < images.GetLength(1); y++)
    {
        images[x, y].ManipulationStarted += manip_start;
        images[x, y].ManipulationDelta += manip_delta;
        images[x, y].ManipulationInertiaStarting += manip_complete;
    }
}

Here, to the right of += the event handlers are specified. For example, when manipulation starts on an image, the following routine is invoked:

//invoked when user begins a gesture on game tile
void manip_start(object sender, ManipulationStartedRoutedEventArgs e)
{
    //store the elapsed time to mark the beginning of this event
    start_ms = game_clock.Elapsed.TotalMilliseconds + (1000 *
        game_clock.Elapsed.TotalSeconds);
    //grab the image (game tile) where manipulation began!!!
    img = sender as Image;
    //retain image location in canvas at start of gesture
    initial_x = Canvas.GetLeft(img);
    initial_y = Canvas.GetTop(img);
    ...
}

The parameter sender is used to retrieve the image that was touched. The code also shows a sample of how one can capture the time when the manipulation started (this helps for velocity calculations when computing change in position over change in time in a given direction). Further, the Canvas class is used to capture the x and y coordinates of the touched image at the beginning of the manipulation. Note, as seen before in the previous sections, the developer can specify a specific event argument in the routine. In this case, the sample code uses ManipulationStartedRoutedEventArgs.

The body of the manip_delta routine is not shown here. When the user releases finger(s) from the screen, , the following snippet shows the event that is invoked and how to compute velocity. Then, it is left to the developer to decide how to create a friction event, such as linearly dampening the velocity over time to slow the flung object down.

//invoked when user ends a gesture on game tile...use final velocity for projection
void manip_complete(object sender, ManipulationInertiaStartingRoutedEventArgs e)
{
    //we will compute the average linear velocity as
    //change in position over time of the manipulation interval
    //note that velocity is a vector, not a scalar, as it
    //has both magnitude and directional components
}
Here, velocity is computed as change in total translation divided by the change in time. The translation is relative to the starting position of the image at the time of manipulation invocation. Alternatively, e.velocities can be used to directly access the velocity passed into the event. Remember that unlike speed (a scalar), velocity is a vector that has both magnitude and direction. The signs vel_x and vel_y clarify this.

10. Summary

This paper guided developers seeking to enable touch in Windows 8 Metro style apps. The paper started with the basics of defining a touch event for a UI element in XAML, and then provided an overview of defining the supporting code "glue" (C# code-behind). The discussion started with the simple case of detecting a single tap. An example of defining and using a double tap detection interval was also shown.

In addition to explaining some of the niceties of Metro style apps, the VS IDE was shown to be simple to use from both the API library perspective and the runtime debugging capabilities. This exciting new software bundle has plenty of headroom for the creative thinker looking to make the most appealing user experience from both a UI and touch sensor perspective. This paper showed an exciting example: using touch APIs to deliver a great learning game for children. Now, go have fun!

About the Author

David Medawar is a Software Engineer with Intel Corporation. He has been with the company for the last seven years, and his former development experience includes system BIOS and boot-loader enabling. David currently works on app enabling for Windows 8 and Android.
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 and the Intel logo are trademarks of Intel Corporation in the US and/or other countries.

Copyright © 2012 Intel Corporation. All rights reserved.

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

This sample source code is released under the Intel Sample Source Code License Agreement.