Comparing Touch Coding Techniques - Windows 8 Desktop Touch Sample

Abstract
There are three ways to support touch input and gestures in Microsoft Windows 8* Desktop apps: Using the WM_POINTER, WM_GESTURE, or WM_TOUCH messages. The featured sample code teaches you how to implement a similar user interface with each.

WM_POINTER is the simplest to code and supports the richest set of gestures, but only runs on Windows 8. WM_GESTURE is easy to code and is backwards compatible with Windows 7*, but has some gesture limitations. WM_TOUCH is also backwards compatible with Windows 7 but requires a lot of code, because you must write your own gesture and manipulation recognizers from lower-level touch events.

Overview
Media and the web have been abuzz about touch input, and new, attractive Ultrabook™ device designs with touch screens arrive all the time. Your customers may already be familiar with using touch devices and a rich set of gestures to interact with apps. Microsoft Windows 8 provides new ways to use touch input in your app with two separate user interfaces, one for Windows 8 Store apps, and another for Windows 8 Desktop apps. There are advantages to each, but one of the more important features of Windows 8 Desktop apps is that they can be backwards compatible with Windows 7. Existing Windows apps, which are being upgraded with touch, have a more direct path if they target Windows 8 Desktop with one of these techniques. For these reasons, here we study how to implement touch on Windows 8 Desktop.

Windows 8 Desktop has three ways to implement touch. The WM_POINTER, WM_GESTURE and WM_TOUCH techniques can each give your customers roughly the same user experience. Which should you use? The sample code featured with this article will help you compare the code required to implement each technique. We’ll look at the pros and cons of each technique.

Welcome to the Sample
The sample uses one of the three techniques to implement a familiar set of gestures. You can switch between the techniques with a quick code change.

To get started, open the sample solution in Visual Studio* 2012, build it, and run it on your touch-enabled Windows 8 system. Now that the app is running, you’ll see a set of colored blocks that you can manipulate, some descriptive text about which technique the app is using, and a list of the gestures you
may use. Try each of the gestures listed on screen: Drag a block with one finger to pan it on top of another, pinch to zoom in and out, and so on.

Figure 1: The sample on startup

Once you are familiar with the app’s behaviors, look in InteractionContextSample.cpp to find the main code. You’ll find typical startup and message-handling code, and a few implementation details.

Key parts of the code are wrapped in `ifdefs`, to let you change which of the three message types and APIs the sample uses. It’s easy to switch between the three different implementations by changing the `defines` that the code uses. See `stdafx.h` for the `defines`.

```cpp
// Defines which API to use, interaction context by default
// Undefining GESTURE and TOUCH leaves the default
//#define USE_WM_GESTURE
//#define USE_WM_TOUCH
```

Figure 2: Define these values in `stdafx.h` to change the touch API

By default, the sample uses `WM_POINTER` messages to receive touch input, and uses the Interaction Context functions for recognizing gestures and manipulations from those messages. You can change `stdafx.h` to modify that behavior so you can see the other APIs in action. Define `USE_WM_GESTURE` to
use the WM_GESTURE code, or define USE_WM_TOUCH to use the WM_TOUCH code instead. Once you change these defines, rebuild the solution to experiment with the differences. The samples have the descriptive text on-screen to highlight the Windows message technique being used and list the ways you may interact with the objects.

With a typical structure for desktop Windows apps, the sample has a WndProc function to handle incoming Windows messages. The message handling code is different for the three implementations. For each implementation, there’s some similar initialization code, both for the objects being drawn and for the background. Look for code like this in each case.

```c
//Initialize each object
for(int i = 0; i < DRAWING_OBJECTS_COUNT; i++)
{
    g_objects[i].Initialize(hWnd, g_D2DDriver);
    g_zOrder[i] = i;
}
```

```c
// Initialize background
g_background.Initialize(hWnd, g_D2DDriver);
```

```c
// Set Descriptive Text
WCHAR text[256] = L"Mode: WM_POINTER*
Pan: One Finger
Zoom: Pinch
Rotate: Two Fingers
Color: One Finger Tap
Reset: Press and Hold"
memcpy(g_szDescText, text, 256);
```

**Figure 3: Example initialization code, from WM_POINTER case**

Because the objects may overlap, the sample maintains a Z-order so that touch events go to only the top object. This code shows the g_zOrder array of objects that are used to keep track of the squares on the screen (plus the g_background background object).

:**DrawingObject.cpp** contains the mechanics of object creation and manipulation (for all message types).

Inertia is very useful for making touch implementations “feel” right. When you pan or rotate an object, inertia can allow it to keep moving, like a physical object.

Let’s study each implementation in more detail.

**WM_POINTER with Interaction Context**

**WM_POINTER** messages are Microsoft’s recommended way to handle touch input for Windows 8 Desktop. More than a single message, this set of messages includes **WM_POINTERDOWN**, **WM_POINTERUP**, and so on; together we’ll call them **WM_POINTER**. These messages are only supported on Windows 8, so if you want backward compatibility with Windows 7, you’ll need to choose one of the other two techniques. These **WM_POINTER** messages give you high-level access to touch events. They’re used with the Interaction Context.

But what’s the Interaction Context? It’s the Windows 8 Desktop mechanism that detects gestures and handles manipulations. It works together with the **WM_POINTER** messages to give full gesture support.
There’s no special initialization code. Your app will receive WM_POINTER messages at startup. This is the sample’s default touch message. There is one bit of optional initialization code added to this case, however. If you want to use mouse input as if it were touch input, call this function.

    //Enable mouse to be pointer type
    EnableMouseInPointer(TRUE);

**Figure 4: WM_POINTER mouse setup (optional)**

This call lets the mouse generate WM_POINTER messages, so separate mouse message handling is unnecessary. Without it, you could still turn mouse input into touch or gesture input, but you would have to write code to handle the various mouse messages. You get this handling for free by calling this function. This is a unique advantage of using WM_POINTER.

Look at the initialization code in **DrawingObject.cpp** to see how the drawing objects are created. For example, manipulation events (translate, rotate, scale) get inertia, via its flags here.

When your app uses WM_POINTER, it will receive groups of gesture events. Under most conditions, this will be a sequence starting with a single WM_POINTERDOWN, followed by any number of WM_POINTERUPDATE messages, and finishing with a WM_POINTERUP. You may see some other cases, like WM_POINTERCAPTURECHANGED, when the window loses ownership of its pointer.

Look at the code for these messages in **WndProc**. The code to handle each message begins with a call to **GetPointerInfo**, which fills a POINTER_INFO struct describing the touch. The code cycles through the objects and checks if the touch applies to the object. If it does, the touch is processed for that object.

Let’s look at the WM_POINTERDOWN case, but they’re all similar.

```cpp
    case WM_POINTERDOWN:
        // Get frame id from current message
        if (GetPointerInfo(GET_POINTERID_WPARAM(wParam), &pointerInfo))
        {
            // Iterate over objects, respecting z-order: from the one at the top to the one at the bottom.
            for (int i = 0; i < DRAWING_OBJECTS_COUNT; i++)
            {
                int iz = g_zOrder[i];
                if (g_objects[iz].HitTest(pointerInfo.ptPixelLocation))
                {
                    // Object was hit, add pointer so Interaction Context can process
                    hr = ddPointerInteractionContext(,
                        g_objects[iz].GetInteractionContext(),
                        GET_POINTERID_WPARAM(wParam));
                    if (SUCCEEDED(hr))
                    {
                        // Bring the current object to the front
                        MoveToFront(iz);
                        g_objects[iz].AddPointer(GET_POINTERID_WPARAM(wParam));
                        ProcessPointerFrames(wParam, pointerInfo, iz);
                    }
                    break;
                }
            }
        }
```
else if (i == DRAWING_OBJECTS_COUNT-1)
{
    // No objects found in hit testing. Assign this pointer to background.
    hr = ddPointerInteractionContext(g_background.GetInteractionContext(),
                                       GET_POINTERID_WPARAM(wParam));
    if (SUCCEEDED(hr))
    {
        g_background.AddPointer(GET_POINTERID_WPARAM(wParam));
        ProcessBackgroundPointerFrames(wParam, pointerInfo);
    }
}
break;

Figure 5: WM_POINTERDOWN processing, showing hit testing and processing

If an object is hit, we add the object to the list of objects that need to process this touch event, using AddPointerInteractionContext. The Interaction Context collects objects that it will affect. Here, the hit detection is done by checking the touch coordinates against the object’s bounding box. This is adequate since the objects are solid rectangles; more complex objects will require a more complex hit test. Once detected, we keep track of the touch ID from GET_POINTERID_WPARAM(wParam). Future touch events can avoid the hit detection and check this ID to see if they apply to this object. If the touch did not hit any of the objects, it hit the background.

WM_POINTERDOWN begins a new gesture. This gesture can change the Z-order. One such event is panning a single object, which may drop the object on top of another. If the Z-order changes, we call ChangeZOrder. It moves this object to the top, and re-orders the rest toward the background.

Now that the objects are in the right order, we handle the touch with the object’s ProcessPointerFrames or ProcessBackgroundPointerFrames. These functions are straightforward. They check if the touch is already being handled, get a list of all touch events in this pointer frame (usually one), and process the touch on this object (with ProcessPointerFramesInteractionContext).

The other cases are similar. Pointer updates (WM_POINTERUPDATE) arrive during a gesture, and the end of a gesture is marked by a WM_POINTERUP message. The UPDATE case doesn’t require a hit test, since we can tell by the ID whether this gesture is already applying to this object. To keep track of the UP case, we just remove the pointer from the interaction context with RemovePointerInteractionContext. If this gesture is one that affected the Z-order, the Z-order was already rearranged when the WM_POINTERDOWN message arrived, so the UPDATE and UP cases don’t need to change Z-order.

To see when the gestures are finalized, look in DrawingObject.cpp at the OnInteractionOutputCallback method. Various tap events move directly to handling functions. The rest are manipulation events, and the code calls OnManipulation to process them. Manipulation may contain any combination of pan, scale and rotate, and that function implements all three together. Manipulation events have inertia (due to their setup options), so the gesture will have INTERACTION_FLAG_INERTIA set in its interactionFlags.
argument. The code starts a short timer to manage the inertia (giving the object time to move after the touch is released). If this timer is started, the WndProc will receive a WM_TIMER message later when the timer expires, and the code calls ProcessInertialInteractionContext on all affected objects.

All the system-supported gestures are handled inside this call, so there’s no gesture-specific code here.

**WM_GESTURE**

Now, let’s look at the WM_GESTURE code. This is a straightforward interface that was first implemented on Windows 7, which you may also use on Windows 8 Desktop. Of the backwards-compatible message types, it’s the easiest to code, but it comes with the most restrictions. You’ll see here how the gesture set is slightly different than the WM_POINTER gestures, gestures may only work on one object at a time, and complex gestures may only have two types of gesture per gesture event.

WM_GESTURE provides high-level access to system-supported gestures. There’s very little initialization code required. Your app simply needs to register for the gestures it will use.

```c
// WM_GESTURE configuration
DWORD panWant = GC_PAN
| GC_PAN_WITH_SINGLE_FINGER_VERTICALLY
| GC_PAN_WITH_SINGLE_FINGER_HORIZONTALLY
| GC_PAN_WITH_INERTIA;
GESTURECONFIG gestureConfig[] =
{
    { GID_PAN, panWant, GC_PAN_WITH_GUTTER },
    { GID_ZOOM, GC_ZOOM, 0 },
    { GID_ROTATE, GC_ROTATE, 0 },
    { GID_TWOFINGERTAP, GC_TWOFINGERTAP, 0 },
    { GID_PRESSANDTAP, GC_PRESSANDTAP, 0 }
};
SetGestureConfig(hWnd, 0, 5, gestureConfig, sizeof(GESTURECONFIG));
```

**Figure 6: Gesture init code for WM_GESTURE**

The supported gestures don’t exactly match those used by WM_POINTER or other common touch environments. These two finger tap and press-and-tap gestures have not gained widespread use. On the other hand, some of the gestures in common use (double-tap) are not supported by this interface. While it’s not shown here, you can use mouse click events to supplement the WM_GESTURE gesture set and easily implement those gestures for your app.

If you analyze the sample when using WM_GESTURE, you will discover that gestures only work on one object at a time, and that complex gestures (pan, zoom, and rotate) are limited to two types per gesture. Even with these restrictions, WM_GESTURE gives you the simplest way to support touch on Windows 7 and Windows 8 Desktop. As an added bonus, this interface implements inertia with no additional code.

In WndProc, the section for handling WM_GESTURE is simple. Gesture details are fetched with GetGestureInfo.

```c
GESTUREINFO  gestureInfo;
```
ZeroMemory(&gestureInfo, sizeof(GESTUREINFO));
gestureInfo.cbSize = sizeof(GESTUREINFO);
GetGestureInfo((HGESTUREINFO)1Param, &gestureInfo);

**Figure 7: Fetching details on this gesture**

With the gesture info, we can check details of this gesture. As with other gesture message sequences, these gesture messages contain an initial message, a sequence of ongoing messages, and a final message.

```
switch(gestureInfo.dwID)
{
    case GID_BEGIN:
        p.x = gestureInfo.ptsLocation.x;
        p.y = gestureInfo.ptsLocation.y;
        for (int i=0; i<DRAWING_OBJECTS_COUNT; i++)
        {
            int iz = g_zOrder[i];
            if (g_objects[iz].HitTest(p))
            {
                MoveToFront(iz);
                g_objects[iz].AddPointer(gestureInfo.dwInstanceID);
                break;
            }
            else if (i == DRAWING_OBJECTS_COUNT-1)
            {
                // No objects found in hit test, assign to background
                g_background.AddPointer(gestureInfo.dwInstanceID);
            }
        }
        break;
}
```

**Figure 8: Capturing the beginning of a gesture**

First, we capture the starting location. As in the previous section, when a gesture starts, the code compares the coordinates of the gesture with each of the objects. If the touch hit one of the objects (or the background) change the Z-order starting with that object, move it to the front and rearrange the other objects toward the background. Then, add this pointer’s ID to it.

As later messages in the sequence arrive, they’re simple to process. Note which objects the pointer affects (by checking the ID), and process the gesture on them. There’s a separate function for each of the gesture types. Look at the **OnTranslate**, **OnScale** and **OnRotate** functions in the sample for the details of how each gesture changes the objects. Other gestures use the same support functions the sample uses for the other message types.

When a gesture is complete, remove the pointer from any objects it affects.

**WM_GESTURE** has its limitations, but supporting gestures with **WM_GESTURE** is straightforward and easy to code.
**WM_TOUCH**

**WM_TOUCH** provides the complete solution for backwards-compatible touch support with Windows 7. Unlike the previous message types, this message doesn’t give gesture notification. It notifies your app for every touch event. Your code must collect the events and recognize its own gestures. While this can involve a fair amount of code, it also gives you precise control over gestures types and details.

Let’s start with a bit of configuration. Call **RegisterTouchWindow** to ensure that your app receives **WM_TOUCH** messages (otherwise, your app will receive **WM_GESTURE** or **WM_POINTER** messages).

The sample creates manipulation processor objects to handle the actual object manipulations, and inertia processor objects to support inertia on the objects as they’re manipulated. Each is created for every screen object (and the background).

```c
hr = CoCreateInstance(CLSID_ManipulationProcessor, NULL, CLSCTX_INPROC_SERVER, IID_IUnknown, (VOID**)(&g_pIManipulationProcessor[i]));
hr = CoCreateInstance(CLSID_InertiaProcessor, NULL, CLSCTX_INPROC_SERVER, IID_IUnknown, (VOID**)(&g_pIInertiaProcessor[i]));

g_pInertiaEventSink[i] = new CManipulationEventSink(g_pIInertiaProcessor[i], hWnd, &g_objects[i], NULL);
g_pManipulationEventSink[i] = new CManipulationEventSink(g_pIManipulationProcessor[i], g_pIInertiaProcessor[i], g_pInertiaEventSink[i], hWnd, &g_objects[i], NULL);
```

**Figure 9: Creating manipulation and inertia processors**

The manipulation and inertia code is in **CManipulationEventSink.cpp** and provides deeper initialization details.

With the app initialized, it’s time to start receiving touch messages. There’s a single case in **WndProc** to receive **WM_TOUCH** messages. We’ll discuss later how the **WM_TIMER** message is used in this case, too.

Once the **WndProc** receives a **WM_TOUCH** message, call **GetTouchInput** to discover the full set of all touch contacts in this message. Iterating across the contacts, the high-level code will now look familiar. First, capture the coordinates of this touch contact. Check if this is a “down” touch by comparing the touch input’s **dwFlags** with **TOUCHEVENTF_DOWN**. Loop through the objects, and check each to see if the touch hit the object. If it did, change the Z-order so the object pops to the top, add the pointer to the list of ones affecting this object, and then call the manipulation processor’s **ProcessDownWithTime** function. This passes a timestamp along with the object to be manipulated.
Looking closer, it’s a bit more complex than you’ve seen with the other cases. That manipulation processor function will in turn call into the `CManipulationEventSink` objects. During a manipulation, the code first calls `ManipulationStarted`. This will start a hold timer so we can detect when a touch has been held “long enough” to count as a hold (set here to 1000 milliseconds or 1 second). The details of that manipulation are done via `ManipulationDelta`, which kills any existing hold timer, then passes regular manipulations on to the drawing object to finish. After the manipulation is done (signaled by calling `ManipulationCompleted`), you may continue to see calls here, so the end of this function continues to call the object’s `OnManipulation` function as long as the inertia sink object is emitting more manipulation calls.

There’s a fair amount of code here to handle the specifics; if you’re using `WM_TOUCH`, read up on the manipulation and inertia processors and study the contents of `CManipulationEventSink.cpp` closely.

That walks you briefly through `TOUCHEVENTF_DOWN`. There’s similar code for `TOUCHEVENTF_MOVE` and `TOUCHEVENTF_UP`.

**Which to use?**

Which API should you use for Windows 8 Desktop? That depends. By now, you should have a sense of the common points between the implementations, as well as the differences.

If you want to have backward compatibility between Windows 8 Desktop and Windows 7, then you may not use `WM_POINTER`. Only `WM_GESTURE` and `WM_TOUCH` will run on both. If you only wish to support Windows 8 Desktop, then `WM_POINTER` is clearly best because it’s the easiest to implement.

Looking deeper, let’s compare the different message types.

<table>
<thead>
<tr>
<th></th>
<th>WM_POINTER</th>
<th>WM_GESTURE</th>
<th>WM_TOUCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction</td>
<td>Gesture</td>
<td>Gesture</td>
<td>Touch contact but this gives full control</td>
</tr>
<tr>
<td>Ease of coding</td>
<td>Easy</td>
<td>Easy</td>
<td>Complex</td>
</tr>
<tr>
<td>Completenss</td>
<td>Manipulation Easy</td>
<td>Easy</td>
<td>Complex</td>
</tr>
<tr>
<td></td>
<td>Inertia</td>
<td>Automatic</td>
<td>Complex</td>
</tr>
<tr>
<td></td>
<td>Gesture set</td>
<td>Complete</td>
<td>Limited</td>
</tr>
<tr>
<td></td>
<td>Manipulate multiple objects Many</td>
<td>One</td>
<td>You implement</td>
</tr>
<tr>
<td></td>
<td>Concurrent gestures Any</td>
<td>Limited to 2</td>
<td>You implement</td>
</tr>
<tr>
<td></td>
<td>Mouse as pointer input Trivial</td>
<td>You implement</td>
<td>You implement</td>
</tr>
</tbody>
</table>

*Figure 10: Comparing message types*
Depending which level of abstraction you want, you may prefer the full control of WM_TOUCH over WM_POINTER and WM_GESTURE, although you will find yourself writing a lot more code. If you can live with its limitations, WM_GESTURE may be right for you.

Write your app’s touch support with the techniques that are best for you!

About the Author
Paul Lindberg is a Senior Software Engineer in Developer Relations at Intel. He helps game developers all over the world to ship kick-ass games and other apps that shine on Intel platforms.

References


Touch Samples – http://software.intel.com/en-us/articles/touch-samples
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.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBCONTRACTORS, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENCE IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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

Intel, the Intel logo, VTune, Cilk and Xeon are trademarks of Intel Corporation in the U.S. and other countries.

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

Copyright© 2012 Intel Corporation. All rights reserved.

Optimization Notice