Game Design Methodologies for 2 in 1 Devices

Introduction
Laptops and tablets are wildly popular within different groups and demographics, mostly because they deliver specialized experiences to their users. The 2 in 1 has evolved from those two established computing form factors, coexisting in a single complementary platform combining the benefits of both usage models and more.

In this article, we focus on the importance of developing games for 2 in 1 form factors and improved user interface experience, which include:

- detecting the input mode (touch vs. keyboard/mouse) change
- predicting the mode (slate vs. clamshell)
- displaying the appropriate user interface

Form Factors and Usage
Traditional laptops have a “clamshell” form factor and are characterized as a mobile personal computer—a device with which you can perform all the tasks available on a desktop but in a mobile, lightweight, compact fashion. With the advent of Ultrabooks, laptops have become lighter, thinner, and with longer battery lives, while making significant strides in performance and functionality. For many users, laptops have become their preferred computing device because it allows them to perform all the tasks they would need on a desktop but with the convenience of mobility and flexibility.

Tablets, on the other hand, are the current pinnacle of mobile technology, utilizing a “slate” mode of input which relies on either handwriting recognition, touch screen keyboard, or an external keyboard. Tablets are compact, very lightweight, and extremely portable. However, they do not possess the processing power of a laptop, and they have limited functionality as a computing device. Tablets are ideal for casual web users who read the news or browse popular websites, for those who play "lightweight" games, and for others who want to watch TV or films while traveling. Additionally, tablets are often preferred by users in creative arts professions like design and music. [1]

The Two-Device Challenge
One of the problems in using two devices (a laptop to create content and a tablet to consume it) is the difficulty in sharing content across the devices with different operating systems and content formats. The advent of the 2 in 1 successfully bridges the gap between the two form factors and provides the convenience of both on a single device with one operating system.
Types of 2 in 1 Devices
There is a range of available 2 in 1 devices. Some have a screen that swivels, and others have a screen that folds back 180°, while a few have a screen that fully detaches. The one key feature that is common to all these models is the ability to quickly and seamlessly go from a fully featured Ultrabook to a convenient tablet and vice versa. Devices can be in one of two modes in this respect: Slate (a tablet-like mode where the primary method of input is touch) and Clamshell (a laptop-like mode where the primary method of input is keyboard/mouse/trackpad).

Any software running on these new devices should be able to identify the form factor and adapt the user interface to change in device state. For example, if the application or system software senses a touch input, it should automatically provide the option for an onscreen keyboard.
Adapting 2 in 1 Devices for Gaming

Games typically fall under two categories: (1) mouse-keyboard/controller-based traditional PC games or (2) touch-based mobile games. Since the 2 in 1 devices are capable of both, it is important to consider how a game would adapt when the user changes from slate to clamshell and back. This change can happen anytime, even in the middle of gameplay.

Special Considerations for a 2 in 1 Gaming Platform

Touch Mode

Touch is a new input mode that traditional games need to adopt. Touch input is available in both slate and clamshell modes, but adding touch interface doesn't necessarily mean that the game has to change its input or gameplay mode. Rather, it can be used to augment the gaming experience, because touch is more intuitive for many kinds of gameplay input. However, there are a few things to keep in mind when adding touch support to a game platform.

User Interface (UI) Design

Any touch-based user interface (UI) should follow tablet gaming design guidelines. For example, it is critical to have buttons that are sized and spaced appropriately for an average adult human finger. It is also important to make the buttons and forms immediately reactive to user input. Keep in mind that when the device is in slate mode, there is a good chance that the user is holding the device with both hands. Designing the UI control elements near the default hand grip locations makes it comfortable to use and play the game without having to balance with one hand and press a button in the middle of the screen. [2]

Figure 3: Source http://msdn.microsoft.com/en-us/library/windows/apps/hh465415.aspx
Seamless Transition of UI

The difference between a dedicated tablet or traditional laptop and a 2 in 1 is that the user can decide to switch from one mode to another. When this happens during a game, it's necessary to detect the change in form factor and provide the ideal user interface for the new mode.

For example, the game Defense Grid: The Awakening implements this transition beautifully. [3] When the game detects a touch input, it displays the slate UI. This UI consists of a column of easily selectable towers on the right edge of the screen near the bezel, a nearby options/game menu button, and a fast forward button at the lower left corner of the screen. The UI elements are placed near hand-grip points, as described above. When the game detects a keyboard, mouse, or trackpad input, the tablet UI elements fade away, and the mouse cursor is displayed. This makes device state transitions very fluid and non-intrusive.

![Figure 4: Slate mode UI elements when touch input detected](image1)

![Figure 5: Touch UI elements disappear and cursor appears on keyboard/mouse input](image2)

On some systems, it is possible to query the system through Windows* API functionality and determine the device state (slate or clamshell). But, this is not implemented across all platforms and cannot be guaranteed to work on older 2 in 1 systems.

Although uncommon, situations may exist where we want the interface to change before the user provides input. If the user closes the laptop and switches over to slate mode, we might want to display the slate user interface without requiring the user to touch the screen first. This is important if the game is in a mode where a negligent touch could be detrimental to the user’s gameplay. Although this factor could potentially be overcome by “special casing” the first touch input after keyboard/mouse usage as only a mode change indicator and not a real game update input, one can understand the desire to be automatically notified of a state change.
GetSystemMetrics is the Windows API to query for device status. [4] The relevant metrics are SM_CONVERTIBLESLATEMODE and SM_SYSTEMDOCKED. Querying for SM_CONVERTIBLESLATEMODE tells us if we are in slate mode or not in slate mode (i.e., in clamshell mode). [5]

```c
bool bSlateMode = (GetSystemMetrics(SM_CONVERTIBLESLATEMODE) == 0);
```

**Figure 6:** DO NOT USE! This API might return incorrect state on some systems. See below.

When this system metric changes, the system sends a broadcast message via a WM_SETTINGCHANGE message, with “ConvertibleSlateMode” in the LPARAM.

```c
case WM_SETTINGCHANGE:
    if(wcscmp(TEXT("ConvertibleSlateMode"), (TCHAR *) lParam) == 0)
        NotifySlateModeChange();
    break;
```

**Figure 7:** Use this mechanism instead. Guaranteed to be broadcast only when there is a true mode change.

Keep in mind that this functionality might not be available on some systems. There is no easy way for an application to detect this API query functionality. This means that the GetSystemMetrics() call could return a false state. On older devices, this might return 0, indicating slate mode, when in fact the device is in clamshell mode.

But, the WM_SETTINGCHANGE broadcast message with LPARAM value of “ConvertibleSlateMode” or “SystemDockMode” will only happen if this feature is supported. If the game detects these messages, it can correctly present the desired user interface to the user. Since this feature is not supported all the time, it is important to trigger the interface change based on detected user input mode changes (touch vs. keyboard) as a failsafe.
On Screen Keyboard

A physical keyboard might not be available in some situations. If the game requires text input, then it becomes important to support an On Screen Keyboard.

If your game runs in a window, the user can manually invoke the touch keyboard from the taskbar, but that is non-intuitive and doesn't help at all for full-screen games. You could write a custom keyboard overlay within your game. This approach can be challenging, but it might provide the best experience for the user.

The following code sample link shows how to bring up the Windows On Screen Keyboard in your DirectX game: https://software.intel.com/en-us/blogs/2013/06/11/touch-keyboard-access-for-windows-8-desktop-apps [6] This approach requires you to have the game run in Borderless Windowed FullScreen mode. Many desktop games will be written using full-screen exclusive mode. This is problematic when using built-in keyboards; either the invoked keyboard will be hidden or raising the keyboard will force the game out of full-screen mode. Transitions in/out of full-screen exclusive mode come with a raft of resource re-acquisition or re-sizing considerations and are generally a pain point that you don't want to add to your keyboard input handling. The solution demonstrated in this sample is to run the game in a maximized borderless window, rather than full-screen exclusive mode. The advantage is that raising the keyboard window does not disturb the game other than temporarily pushing it one level down the z-stack. The game continues to run in the background to process keyboard input, and it regains focus automatically as soon as the keyboard is dismissed.
AutoRotation/OrientationLock

All 2 in 1 devices come equipped with various sensors. One benefit of that is the AutoRotation feature. When in slate mode, the device can be used in either portrait or landscape orientation. If AutoRotation is enabled, a DirectX-based game would lose its D3D device and would have to recreate a new device with the new resolution as dictated by the orientation change. For example, the full screen dimension might change from 1600x900 to 900x1600. This is usually not desirable in a full-screen game. This is easily fixed for modern UI-based Windows applications by specifying orientation preferences in the application manifest. But, for desktop games, we can still get this functionality by directly looking up and calling an exported function in user32.dll. This function can then be used to disable autorotation by specifying the game’s orientation preferences. [7]

Figure 10: Code snippet to disable AutoRotation, and limit to Landscape orientation.

typedef enum ORIENTATION_PREFERENCE
{
    ORIENTATION_PREFERENCE_NONE              = 0x0,
    ORIENTATION_PREFERENCE_LANDSCAPE         = 0x1,
    ORIENTATION_PREFERENCE_PORTRAIT          = 0x2,
    ORIENTATION_PREFERENCE_LANDSCAPE_FLIPPED = 0x4,
    ORIENTATION_PREFERENCE_PORTRAIT_FLIPPED  = 0x8
} ORIENTATION_PREFERENCE;

typedef BOOL (WINAPI *pSDARP)(ORIENTATION_PREFERENCE orientation);

pSDARP pARP;

pARP = (pSDARP) GetProcAddress( GetModuleHandle(TEXT("user32.dll")),
                               "SetDisplayAutoRotationPreferences" );

if( pARP )
{
    pARP( (ORIENTATION_PREFERENCE)(ORIENTATION_PREFERENCE_LANDSCAPE |
                                             ORIENTATION_PREFERENCE_LANDSCAPE_FLIPPED) );
}
**Touch-Based Gestures**

Touch-based gestures make gameplay elements more intuitive. Depending on gameplay, touch input can be more intuitive than keyboard or mouse. In a real time strategy game, touch allows the player to make a lasso gesture to select units. These units can then be commanded to a target area by touching the mini-map or by pinch-zooming-out the main map and selecting an area that was originally outside the view. Also, using the RTS scenario, a user can select unit(s) and trace a special path on the screen for the units to use as their patrol path. These kinds of commands are hard to visualize and cumbersome to implement with traditional input systems but operate naturally in touch mode.

![Image: Civilization V added touch controls to provide a smooth touch-based gameplay experience.](image-url)

**Figure 11:** Civilization V added touch controls to provide a smooth touch-based gameplay experience.

Multi-finger touch adds many commands in a simple interface. For example, Civilization V implemented the following touch controls [8]:

- **Pinching** - Zooms the camera in and out
- **Dragging** - Provides info about the unit, or terrain the finger is hovering over
- **Two-finger scroll/pan** - Moves the camera up and down or left and right
- **Tap** - Selects a unit
- **Double tap** - Issues a move command to selected unit
- **Two-finger tap** - Exits current command without issuing order
- **Three-finger tap** - Closes open menus or open the game menu

Since touch is also available in clamshell mode, these new gestures augment the traditional interfaces and provide enhanced user experience.
Touch API Choices
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. [9]

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

Depending which level of abstraction you want, you may prefer the full control of WM_TOUCH over WM_POINTER and WM_GESTURE, although this requires more code. WM_GESTURE may be right for you if you can live with its limitations.

![Table comparing various Windows touch APIs](image)

*Figure 12: Comparing the various Windows touch APIs*
Here is a link to a code sample that demonstrates how to integrate touch into an application using both WM_GESTURE and WM_TOUCH APIs: https://software.intel.com/en-us/vcsource/samples/windows-7-touch [10]

**Packaging in Unity 3D**

Some versions of the Unity 3D engine do not process touch messages in applications running on Windows 7 and Windows 8 Desktop mode. Although the Unity engine does not process touch messages itself, it is possible to register the Unity window for touch from a plug-in. The trick is to create a plug-in that uses various Windows APIs to intercept the touch messages that are sent to the application. Once the window is registered, the plug-in provides access to the touch messages in a script in the Unity application.

The code sample for this touch plugin can be found here: https://software.intel.com/en-us/articles/adding-multi-touch-support-to-unity-games-on-microsoft-windows-7-and-windows-8-desktop [11]

**Latency Comparison of Input Devices**

![Figure 13: Overview of full hardware and software stack contributing towards touch response latency](image)

End-to-end latency can be anywhere from 50-100ms, depending on the device, but this is continually improving in touch devices [12]. This latency is the time taken from the recognition of an input from the touch to the effect displayed on the screen. This is not very different from a traditional mouse- or trackpad-based input. A traditional mouse has a hardware latency of ~8ms, and you still have the software stack latency in addition to that.

If the game depends on twitchy gameplay, touch will not be a good option. But, it could still be used for augmenting gameplay for normal controls.

**Companion App Mode**

Even if the main game relies on twitchy gameplay and does not adapt to touch-based play on a tablet, there could be elements of the game that play well in the touch-only mode. The gameplay may involve planning and character customization, viewing and publishing saved
state and replays, watching and socially interacting while other friends are playing matches, organizing inventory, sorting quests, etc. Most of these gameplay elements could be better enjoyed on a touch screen/tablet form factor.

Extracting these gameplay elements and adding a companion app that can be enjoyed in tablet mode provides users with a well-rounded usage experience.

Other Aspects to Consider
Tutorials or in-game Help dialog should refer to the correct mode (Touch/KeyBd/Controller), depending on which mode is currently being used. If possible, it is better to show both Touch and KeyBd/Mouse information so a user can transition from one to another without having to go back to search around for the controls again.

Adding unique accomplishments that players can achieve in touch mode will add to the gamer’s experience and invite traditional players to try out the new gameplay elements exposed through Touch.

About the Author
Doraisamy Ganeshkumar is a Senior Software Engineer on the Intel Developer Relations team. He helps PC game developers optimize games for Intel products. Outside of work, he likes to bike, hike, work on wood, and tinker with tech toys.

References


