tablet PC developer

http://www.devx.com Printed from http://www.devx.com/TabletPC/Article/21302

The Challenge of Designing Interfaces for the Tablet PC

Designing a usable interface for a Tablet PC program is considerably trickier than simply drawing blue lines across a yellow background and calling it a legal pad. Consider these challenges, and learn how to overcome them.

by Larry O'Brien

The pen is a harsh mistress. While the portability and direct manipulation of the Tablet PC form-factor are a dream come true for many users, designing a usable interface for a Tablet PC program is considerably trickier than simply drawing blue lines across a yellow background and calling it a legal pad. UI design is made harder because the Tablet PC is frequently compared to the versatile blank sheet of paper.

The first major challenge in designing and effective UI is obvious the moment you use a tablet: parallax. The Tablet PC's display is actually a couple of millimeters below the screen surface. Because your eye rarely happens to be precisely perpendicular to the pen's location on the display, this implies that the pen's physical location on the screen surface generally appears offset from the location of the mouse pointer on the display. This phenomenon exists with PDAs and touch-screen devices such as ATM displays, but the tablet's larger display area, varied viewing angles, and generally higher display resolution of the Tablet PC make the parallax more apparent. Figure 1 shows a parallax offset of 15 pixels or so when the pen moves towards the upper-right-hand corner of the display. Note the relative size of the offset, the mouse pointer, and the form's control boxes: having to compensate a few pixels might sound like a trivial adjustment, but after using a Tablet PC for even a few hours, you come to appreciate how many widgets on an interface are really quite small. So the first rule of Tablet PC UI design is: Make Big Controls and Hotspots.

Obeying this rule helps address another challenge offered by the Tablet PC, one that is less obvious than parallax: the "natural" size of writing on a Tablet PC is larger than what you're used to on paper. A paper contract may have an ample signature area of 3" x ½" (roughly 75mm x 12 mm.), a similarly sized area on a Windows Form feels undeniably cramped. I believe this is a result of the combination of parallax and the relatively low resolution of the display compared with paper. The two factors require larger strokes to reflect the subtleties of cursive writing. After more than a year of using a Tablet PC, I've found that my ~9 ¾" x 7 ¼" screen (12" diagonal) captures about the same amount of writing as I put on a 6 ½" x 4 ½" notepad. This ratio, which seems fairly constant in my practice, is the basis of a second rule: Budget your ink-capturing controls at 150% of the space you'd allocate for them on paper.

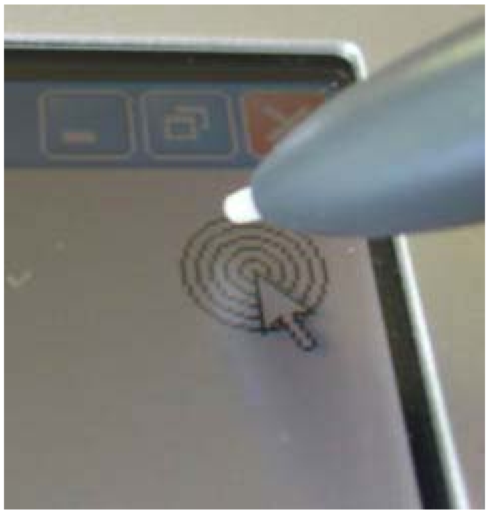


Figure 1. The distance between the pen tip and the actual screen creates a parallax that can amount to 15 pixels or more. So, controls and hotspots should be made large.

The next limitation on UI's is one that jumps out the first time you surf the World Wide Web with a Tablet PC: It's succinctly articulated as "You can't see through your hand." Many Websites place menus along the left side of the screen and cascade sub-options to the right. This is fine with a mouse interface, but is frustrating for right-handed pen users, as sub-menus are obscured by the pen-holding hand (Figure 2). This problem can cause trouble anytime your interface requires pointing; but it is especially troublesome when the UI design uses dynamic elements that move to the right or, to a lesser extent, downward.

The final important challenge of UI design for Tablet PC is that the pen is a significantly "overloaded" device. It takes on the chores of pointing, input, and command selection. Entering short amounts of text with a

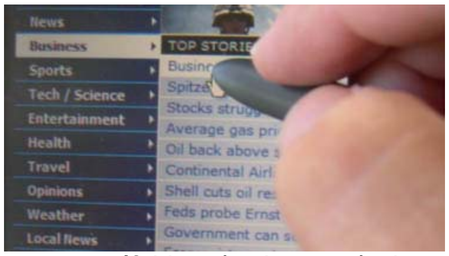


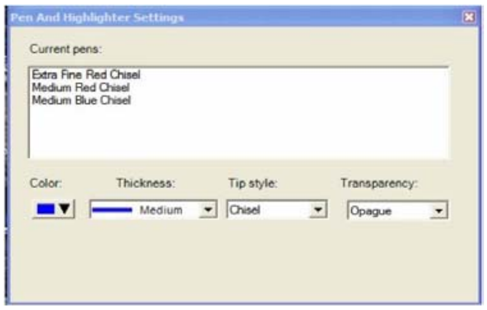
Figure 2. If this selection results in another sub-menu, a right-handed user cannot see it because his hand obscures it.

pen is a fine thing and drawing is downright pleasurable; but other than beyond those two activities, using a pen to control an application can be a hassle. Less is more when it comes to the pen: focus on the core value that the pen-and-tablet combination is providing you (mobility, discreteness, ink capture, and text recognition) and strive to expose those benefits immediately and directly. Avoid dialog boxes, deep menus, and complex navigation, all of which are more annoying on a Tablet than on a keyboard-and-mouse system. Use the pen, therefore, as much as possible for writing and as little as possible for selecting and pointing.

Starting In
Pragmatically, the first thing to do when developing for the Tablet PC is to set the **Font** property of your **Form**s to a larger size than the default 8.25 points. I generally kick it up to 12 or even 14 points. The default font for a Windows Forms **Control** is that of its containing **Control**, so setting the **Font** property of a **Form** is a quick way to "cascade" a change throughout a **Form**.

As to selecting a pen (that is, as it appears on the screen) the Tablet PC SDK does not provide a standard dialog for selecting a combination of values for the **DrawingAttributes** that make up a particular pen. Two UI idioms are emerging in response to this need: a **ToolBar** that presents a limited number of ready-made pens, and a dialog box that allows for separate specification of **Color, PenTip, Width**, and **Transparency**. (Microsoft Windows Journal has a dialog of this kind.) The **ToolBar** with a small selection of pens appears to be gaining acceptance as the primary idiom, although I hope a standard control will appear in future Tablet PC SDKs. In the meantime, Figure 3 shows a **PenSelectionPanel** custom control that I wrote that can be incorporated into either a dialog or tool window. My **PenSelectionPanel** allows the user to compose the **DrawingAttributes** properties, stores specifications in a **ListBox**, and fires a **PenChosen** event that contains the **DrawingAttributes** as part of the event arguments. (All of the source code and files needed to generate the dialog in Figure 3 can be downloaded from here)

Once you have a set of **DrawingAttributes**, apply them to the **DefaultDrawingAttributes** property of the ink-collecting object in your form: either an **InkCollector**, an **InkOverlay**, or an **InkPicture**. Selecting an appropriate ink-collecting class was covered in my previous article.



Extra Fine Red Chisel
Medium Red Chisel
Medium Blue Chisel

Color: | Thickness: Medium | Tip style: Chisel | Transparency: Opague

Figure 3. There is no standard dialog for selecting a pen, although several common choices are emerging. However, you can feel free to design your own dialog until the Tablet PC SDK comes up with a standardized version.

Saving and Restoring Ink
Saving ink is simple: the **Ink** class has **Save()** and **Load()** methods that return and consume **byte** arrays. The **Save** method takes as a parameter a value from the **PersistenceFormat** enumeration: **InkSerializedFormat, Base64InkSerializedFormat, Gif**, and **Base64Gif.** The **Base64X** values are used for embedding ink directly into XML format. An interesting implementation fillip of **PersistenceFormat.Gif** is that native ink-as-ink data is stored in the header fields of the GIF data, allowing you to use this bitmap format as a "transport" for the vector-based ink data! There are a few caveats: calling **Save(PersistenceFormat.Gif)** with an **Ink** object that contains no data throws an **Exception** with the alarming error message "Catastrophic failure."

Similarly, you must call **Load()** only with an Ink object that contains no data, otherwise an **Exception** is thrown with the message "The parameter is incorrect." However, you cannot assign the **Ink** property of an ink-collecting object while ink-collection is enabled. Thus, to load **Ink**, you must use a code block similar to this one:

```
myInkCollector.Enabled = false;
myInkCollector.Ink = new Ink();
myInkCollector.Ink.Load(myByteArray);
myInkCollector. Enabled = true;
```

A common desire is to use **Ink** to annotate existing graphics. The **InkPicture** control is the tool for this job, with the graphics stored in the **Image** property and the **Ink** stored in the **Ink** property. (Remember that ink collection is done in a transparent z-layer above the **Control**, which is why **InkCollector** and **InkOverlay** can be so flexibly added to any **Control**.) While the separation of **Ink** and **Image** is great for editing purposes, you will often want to composite the two together into a bitmap. Listing 1 is an application that demonstrates the process.

First, you use the "Load Image" button to select a picture for **inkPicture1**. Then, draw on it with the pen. Finally, press the "Composite" to create a new bitmap that is loaded into the **Image** property of **pictureBox1**.

Let's step through the behavior of **button2_Click()**: first, we change the **AntiAliased** property of all the **Strokes** in **inkPicture1** to **false**, otherwise composited artifacts will occur. Second, we create a **Bitmap** from the ink data, using **Ink.Save()**. Third, we composite this newly created bitmap with the underlying bitmap. This is done in the function **CompositeInk()**.

CompositeInk()'s task is to align and scale the rasterized ink bitmap with the background image. It does this by using the **GetBoundingBox()** method of the **Strokes** collection to set two **Point** objects inkOrigin and inkExtent. The value of these Points is in ink's native HIMETRIC space, which has to be converted into appropriate pixel values. This is the role of the **Renderer()** class and its **InkSpaceToPixel()** method. With the bounding box of the ink now transformed into pixel coordinates, **Graphics.FromImage(dest)** creates a **Graphics** object for drawing the ink bitmap onto the **dest** bitmap.

After **CompositeInk()** completes, **pictureBox1**'s **Image** property is set to the composited bitmap. Compositing ink into a bitmap removes editability, of course, but is vastly more shareable.

Input and Edit Text

The Tablet PC SDK provides an **InkEdit** control that accepts ink and, after a programmer-defined **RecoTimeout**, places the "top string" or best guess of the recognizer into an underlying **EditBox** control. While impressive the first time you use it, it doesn't provide any pen-based editing facilities, which limits its practical use. The forthcoming Windows XP Tablet PC Edition 2005 has significantly improved handwriting recognition and a dramatically improved Tablet Input Panel (TIP). The new TIP provides three choices for input: a QWERTY keyboard, a free-form text recognition panel with editing capability, and a "comb" control for handwriting on a letter-by-letter basis (perfect for filling out formatted fields).

The TIP can be used by all applications, but recognition improves dramatically when the TIP is given context information about what type of data is expected for an input field. The SDK for Windows Tablet PC Edition 2005 comes with a context tagging tool that makes it very easy to add context-specific information to any application: essentially, you start up the application, drag-and-drop a "finder" icon onto the field for which you want to set context, and then choose either a preset context (like "City" or "Phone number") or input a regular expression. The tool saves the set of contexts in an XML file that the TIP automatically loads thereafter. It's a very clean model and is the best way to support ink for filling out forms.

However, not all contexts can be described statically. Let's say that you want to bias a field towards Contacts from your Outlook file: the context-tagging tool isn't going to help you. You can programmatically control text recognition by creating a **PenInputPanel** object, associating it with a particular **Control**, and then settings its **Factoid** property. For some reason, the **TabletInputPanel.Factoid** property takes a string, not an object of type **Factoid**. The acceptable values of the **Factoid** vary from world region to region, but the most important for dynamic context is generally "WordList" which strongly biases the recognizer to a set of specified values. Unfortunately, the **TabletInputPanel** simply does not support the "WordList" factoid. (If you want to use a WordList factoid without the TIP, see Listing 3 in my previous article.)

and Having Writ, Moves On

Designing for the Tablet PC is a challenge. While traditional applications are so standardized that we can fool ourselves that we're designing decent UIs, the ways in which people use a Tablet PC are so varied, and evolving so fast, that there are few patterns to guide the developer toward success. However, there are two things going for the Tablet PC programmer: The Tablet PC SDK is a tremendously powerful and easy-to-use SDK, and the average Tablet PC owner is enthusiastic about the form-factor. Good UI design will keep that user happy.

Larry O'Brien is the founding editor of Software Development Magazine. When not working on his Tablet PC projects, he is the

Windows columnist for SD Times. He can be reached through his website, http://www.knowing.net.