Palo Alto Research Center

# Stylus User Interfaces for Manipulating Text

David Goldberg and Aaron Goodisman

XEROX

# Stylus User Interfaces for Manipulating Text

David Goldberg and Aaron Goodisman *

CSL-91-9 September 1991 [P91-00110]

© Copyright 1991 Xerox Corporation. All rights reserved.

**Abstract:** This paper is concerned with pen-based (also called stylus-based) computers. Two of the key questions for such computers are how to interface to handwriting recognition algorithms, and whether there are interfaces that can effectively exploit the differences between a stylus and a keyboard/mouse.

We describe prototypes that explore each of these questions. Our text entry tool is designed around the idea that handwriting recognition algorithms will always be error prone, and has a different flavor from existing systems. Our prototype editor goes beyond the usual gesture editors used with styli and is based on the idea of leaving the markups visible.

**CR Categories and Subject Descriptors:** H.5.2 **[Information Interfaces and Presentation]:** User Interfaces - Input devices and stategies; I.7.1 **[Text Processing]:** Text Editing;

**General Terms:** Human Factors

**Additional Keywords and Phrases:** stylus, two-view editor, character recognition.

This paper appeared in the Proceedings of the Fourth Annual ACM Symposium on User Interface Software and Technology, pp 127 - 135.

* also with Department of Computer Science, Massachussetts Institute of Technology, Cambridge, MA 02139

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

# 1 Introduction

Common experience with keyboards and pens suggest that keyboards are optimized for text entry, and pens for drawing. With the appearance of small, portable pen-based computers such as the GridPad, whose primary input device is a stylus (i.e. an electronic pen) used directly on a display, we are faced with the problem of how to use a stylus to manipulate text. This paper studies two aspects of this problem.

First, what is a good user interface to a handwriting recognizer? There have been many papers on handwriting recognition (see [11] for a survey), but very few on user interfaces to recognizers (the most completely described interface is PenPoint [3]). Our system is based on the philosophy that recognizers will always make errors, and that the interface must be designed from the beginning to accomodate that fact. This results in a substantially different interface from PenPoint (and was developed before the publication of [3]).

The second problem is this: can we find an interface that exploits the differences between a stylus and a mouse/keyboard? If we cannot, then there is a real question as to whether pen-based computers are viable, since text entry is slower and more error prone with a stylus than with a keyboard, and since it is feasible to build a notebook sized pen-based computer with a keyboard and thumb operated trackball with similar functionality to a mouse (such as the portable Macintosh).

Two of the primary differences between a stylus and a mouse are (1) styli have much finer control than mice (try writing your signature with a mouse), and (2) while it is easy to manipulate three buttons on a mouse, it is difficult to use the same hand to both hold a stylus and press one of three buttons. Most previous work with styli has focused on using gestures as editing commands ([2], [3], [4], [8], [14] are examples). This works well because it doesn't require buttons, and it is easier to draw gestures with a stylus than a mouse. However, all these editors simply provide a different way to do the same tasks that can be performed with mice and keyboards. We propose that a markup editing interface (which extends gesture based editors by leaving the gestures visible) is more appropriate for pen computers, because it can perform some common tasks that are very awkward with traditional keyboard/mouse systems.

There is a continuum of evaluation techniques for studying user interfaces,

ranging from formal user studies to informal testing by a few users. Since the space of stylus-based user interfaces is just beginning to be explored, we felt that informal observation of users was the appropriate way to evaluate our prototypes. Once the informal exploration has identified a few promising UI techniques, then formal studies will become more important.

This paper has four parts. First, we describe the hardware used in our studies. Next, we classify the different types of recognition systems and explain the choices we made for our system. In the third part, we describe our text entry program, which uses a number of techniques to compensate for the error-prone analogue nature of handwriting recognition. Finally, we describe our markup editor, and single out what we feel is a key design principle for such editors. More details on the programs can be found in [6].

# 2 The Hardware

We did all our studies with a *scratchpad*, which is a PARC-built peripheral to a Sun Microsystems SPARCstation. A scratchpad consists of an 1120 by 780 LCD display, on top of which is a transparent tablet made by Scriptel Corporation. The display is driven by a custom SBus card. To SPARCstation software, the scratchpad looks like a second display, and our programs wrote directly onto the display using Sun's pixrect library.

The Scriptel stylus we used for these studies is tethered to the tablet, and has a microswitch in the tip to detect when the stylus is in contact with the tablet, but has no buttons on the side of the barrel (the newest styli from Scriptel do have a side button). The stylus plugs into an RS-232 port at the back of the SPARCstation, and reports the location of the pen at a rate of 200 locations/second, with a resolution of about 400 dots/inch, which is roughly four times the resolution of the display. The stylus can accurately report its position even when it is above the display, that is, when the tip of the stylus is not in contact with the tablet. Each location report contains a bit telling whether or not the tip switch is depressed.

Because the Scriptel tablet lies between the pen and the display surface, there is a fair amount of parallax between the tip of the pen and the display. Although we tried to compensate for this by displaying a cursor, most users had some difficulty pointing the stylus because of the parallax.

# 3 Parameters of Text Entry Systems

There are five major parameters that characterize handwriting recognition systems.

**Printing/Cursive Script** Electronic styli can detect when the pen is in contact with the tablet. This gives a way to mark the beginning and ending of strokes, namely by when the pen is lifted off the tablet. In printing the pen is lifted between each character (and also perhaps between strokes within a character). In cursive script writing, the pen will not always lift between two characters.

**Upper/Mixed Case** Does the system recognize only upper case letters, or does it accept mixed case?

**Boxed/Unboxed** The key issue is who decides how to group strokes into letters. In a boxed system, each letter is in a separate box, so the user groups strokes into letters by writing different letters in different boxes. In an unboxed system, the system provides only a line to write on rather than a row of boxes, and the system itself must compute how to group strokes into letters.

**Recognition Feedback** Does the system provide feedback after each letter, immediately displaying the result of the recognition? Or does the user signal the system after entering a unit of text, at which point the system recognizes that whole block of text at once.

**Writer Dependent/Independent** Is the system writer independent, or does it require each user to first train the system to learn his handwriting.

## 3.1 Discussion

Here are the choices we made for our text entry tool. Our system recognizes printed text. Cursive script recognition was not chosen because there are no script systems known to the authors that have consistently high recognition rates.

Our system recognizes mixed case. Although some commercial systems (Pencept, GridPad) recognize upper case only, we feel that mixed case systems are necessary for the acceptance of stylus text input.

Even though our recognizer can perform in an unboxed mode (that is, it can do the segmentation itself), we chose to do all our UI studies in boxed mode. There were several reasons for this. First, recognition in boxed mode is more accurate, because there are no segmentation errors. Second, correction and editing are greatly simplified when each letter is in a box.[1] Based on our informal studies, we identified a third advantage to boxed input, which may be most important of all: when users are given only lines to write on, some writers frequently slip into a script mode where they join pairs of letters. For recognizers that can recognize only discrete characters, this obviously results in poor recognition rates. Boxes greatly reduce the chance that a writer will join pairs of letters.

Our system recognizes one letter at a time. We did not study tradeoffs of recognition by blocks versus by character, however we can make the following observation. In our system, you can correct at any time (and we provide two correction gestures: one to delete a letter and one to open up space). Systems that recognize a block at a time essentially require a mode for correction, since you can't correct until you see that the recognizer has made an error, and thus you can't correct without first instructing the system to perform the recognition. However, block recognition does provide more context to help in performing the recognition.

## 3.2 Writer Independence

The last item on our list is the most interesting tradeoff. If writer independent systems had the same accuracy of writer dependent ones, then writer independent systems would be the obvious choice. However, writer dependent systems are more accurate. This is only partly due to imperfect recognition technology. Consider the pair of letters $u$ and $v$, written by two different writers, and the unidentified letter shown in Figure 1.

If the unidentified character was printed by the first writer, it is obviously a $v$, whereas for the second writer it is a $u$. This illustrates that even for

[1] PenPoint finesses this point by allowing unboxed input, but then after recognition, redisplaying using boxes.

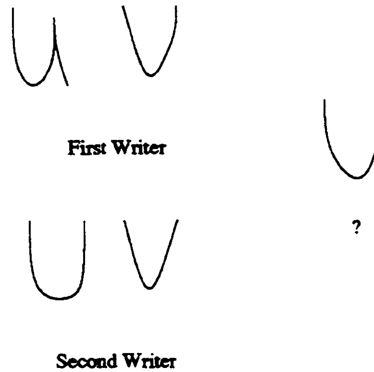


Figure 1: Lower case u, v for two different writers

human recognition of letters, knowledge about the writer is important. The importance of writer-dependent information is even more crucial for mixed case systems than for upper-case only, for the simple reason that there are more easily confused pairs like *u* and *v*.

Our system is writer dependent and requires training. This is partly because we wanted to focus on user interfaces and minimize the effort on building a recognizer. A trained recognizer is easier to write because it doesn't require collecting a large database of user writing styles. But we also feel that trained systems (or hybrid systems that modify their behavior with training) are necessary for achieving high accuracy.

There are two disadvantages to trained systems: first, they must be trained, and second, before using them, you must identify yourself to the system so that it can locate your training database. For pen-based computers that will be used for several hours a day, the overhead of a training session is small. Hybrid systems have the advantage that they can be used immediately, but we believe that virtually all users will eventually want to go through a training session to improve accuracy.

The potentially larger problem with trainable systems is that you must identify yourself to the system before you begin writing. Similar issues arise in speech recognition ([5]), but the inconvenience of identification with stylus systems is much less than with speech systems. First, there is often no overhead to begin using a speech system: you simply begin speaking. Thus the relative overhead of an extra identification step can be significant. Imagine

an elevator where you speak the floor number. Having to first identify yourself would take longer than performing the actual task of speaking a floor number! For stylus applications, there is built-in overhead, because you have to pick up the pen and place it on the writing surface. Thus the relative cost of identifying yourself is less. Second, with speech there is a problem when several people alternate speaking (say at a meeting). Identifying the speaker after each speaker change is a significant problem. With multiple writers, the common case will be for each writer to have a separate pen, and thus once a mapping between people and pens has been established, there is no further overhead to identification.

Once you've decided to use a trained system, there is the question of how to do the training. Perhaps the best method is to do it automatically, so that each time the user draws a letter, that is used as training data. The disadvantage of this technique is that a letter may be misrecognized, and if the user doesn't bother to correct it then incorrect training data will be used. Our system requires explicit training instead. With explicit training, we believe that it is important to make training as simple as possible. Thus we do not require the user to invoke a special program to train. Instead, whenever the user is entering text, he can get a display which enables him to examine and modify his training samples via a single tap. Details are in section 4.2. Experience with our prototype shows that users often write somewhat differently in a formal training session than when performing real-life tasks. Thus the accuracy of the training data improves if it is based on input during actual tasks, and that in turn is more likely if there if very low overhead to modifying training samples.

# 4 The Text Entry Program

The primary difference between keyboard and styli are that keyboards are discrete. When you hit the $A$ key, you will always get the character $A$. On the other hand, when you write the letter $A$, because recognizers are heuristic, sometimes you won't get $A$, but rather $H$ or some other incorrect letter. Even the best recognizers will occasionally misinterpret a letter that seems unambiguous to the writer. Thus a successful user interface to a character recognizer must have a strategy for overcoming the analogue nature of handprinted input. Our text entry prototype uses two major techniques to

improve accuracy.

## 4.1 Segmenting Cursor

First, it uses boxed entry, which eliminates segmentation errors and discourages run-on pairs of letters. Some users, however, find it awkward to write in an input area filled with boxes. So we developed a technique we call the segmenting cursor[2] to gain the segmenting advantages that boxes give, without having to fill the input area with boxes.

The basic idea is simple. The user is presented with a single box. This is like the cursor in computer keyboard systems, except that this box is the outline of a box, rather than a solid box. The user writes the first letter inside the cursor/box. When all the strokes of the first letter are completed, the user begins the first stroke of the next letter outside and to the right of the box. At this point, the box/cursor moves to surround the letter now being written.

In practice, we have discovered that a few refinements to this basic idea are necessary. First of all, if the user is allowed to write on more than one line, there should be a box at the beginning of the line following the line being written on. That way, when the user wants to skip to the next line, he knows where to write the first character af that line.

Second, we found that it helps if the cursor is actually a pair of boxes. The user writes the first letter in the left-most box. As soon as he sets the stylus into the second box, the first box disappears and a new box appears to the right of the second box. The reason this helps users is that at the moment the user lifts the pen from the last stroke of a letter, he would like to see the following box. But the system cannot know to create the next box until the pen touches down outside the current box. Thus having a pair of boxes ensures that when a user wants to write a new letter, the box for it will be there in time to aim the beginning of the first stroke of that new letter.

Finally, if the writer wants to leave a space to begin a new word, it helps to have a visible box where the new word will appear. Thus, the initial cursor has two boxes, but after the user begins to write, we put up a third box. See figure 2.

[2] This was based on a suggestion by David Gifford of MIT, although we later discovered a similar idea in the patent literature [12].

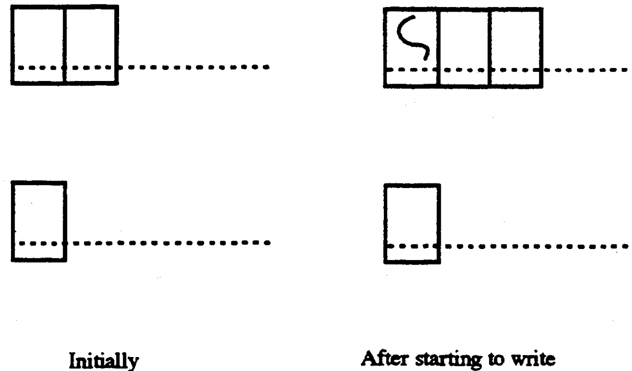


Figure 2: Segmenting Cursor

Some users find that erasing of old boxes and drawing of new boxes distracting. Having the "buffer" of boxes as just described helps minimize this problem, because the drawing of the new cursor/box does not appear directly under where the pen is writing. Although many users find the segmenting cursor a pleasant entry interface, even with this buffering some still find the moving cursor distracting. Thus a system that uses the segmenting cursor will probably want to provide a mode where the writing area is filled with traditional boxes.

## 4.2 Tap-Correction

One obvious way to deal with the fact that recognizers make errors, is to leverage off the fact that when the recognizer is wrong, its second guess is usually right. Our first attempt to exploit this idea was as follows: after printing a letter, the user would be presented with the first guess by having his drawn letter replaced by the recognizer's best guess. We also presented two smaller boxes containing the second and third choices (see Figure 3). If the first guess was correct, the user would go on and print the next letter. If the first guess was not correct, then the user could either redraw the letter, or touch one of the buttons with alternate choices. Our expectation was that most of the time, the correct choice would be in one of the smaller boxes, and that the user save effort by clicking on the appropriate small box, rather than having to reprint the letter.

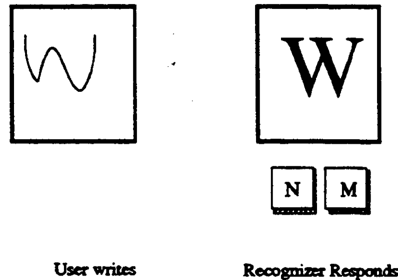


Figure 3: Initial attempt at correction

However, our informal experiments showed that users rarely clicked on the small correction boxes. It seems that the effort of reading what was in the boxes and deciding if they were correct required too much cognitive overhead, and so users prefered to simply rewrite the letter.

With that failure behind us, we tried a new idea. The simplest gesture to make with a stylus is to tap, and so in our next system if a letter was misrecognized, the user could tap on it and the system would display the recognizer's second choice. This was much more successful. When the recognizer was wrong, but the second choice was correct, users tended to perceive this as not really being an error. We call this system "tap-correction".

This technique does have a downside, which is illustrated by the letter *i*. Normally, the system echos pen motion, spreading "ink" to mark the path of the stylus, just as a real pen would. When the system recognizes a letter, it replaces the ink with a nicely drawn version of the recognized letter either when the user begins the next letter, or when one second has elapsed with no input. So if a user writes a lower case *i* by first drawing the vertical stroke and then a dot, the recognition will not occur until he moves on to the next letter (or pauses). However, if a user writes slowly, and one second elapses between the time of the vertical stroke and the dot, the dot will be intepreted to mean "choose the second choice", rather than as dotting the *i*. Our experience suggests that this is not a major problem, and can be minimized by allowing users to change the one second time-out.

The next issue is what meaning to assign to a second tap. There are three

possible actions: display the third best recognition, clear the writing area so the user can rewrite the letter, or just escape into a system that displays a keyboard so the user can select the key with his letter. We chose to have the second tap clear the display, allowing the user to rewrite the character. At the same time, a small box appears. Tapping on this box jumps to a display that not only contains a keyboard, but also displays the reference characters and buttons to modify them. This provides the easy access to reference characters that was mentioned earlier.

We also found that after the second tap, when the user was going to rewrite the letter, it was useful to redisplay the original ink. This has two advantages. First, sometimes a user would forget what letter he was writing. This usually happened when the user wrote a stretch of letters, and then went back to correct the errors. Seeing the redisplayed original letter was often a faster way to recall the original letter than rederiving it from context. Second, redisplaying the ink gives the user useful feedback on why the recognizer failed. After the recognizer has replaced the users ink with an incorrect guess, there is no longer any hint as to why the recognizer failed. When the ink is redisplayed, it is often obvious what the problem was. The input letter may have been poorly formed, or perhaps the user drew a letter that is ambiguous (e.g. $u$, $v$). Redisplaying the original ink helps remove an air of mystery from the whole system.

One important aspect of tap-correction is that if you make an error, you cannot immediately rewrite the letter: you must tap twice before you can rewrite. However, in practice a double tap is a sufficiently simple gesture, so that this is not an obstacle to directly proceeding to a rewrite. The advantage of this approach is that users are "trained" to use the built-in fast correction: you must go thru the "tap for second choice" step before you can rewrite a letter.

# 5 The Editing Program

Most tasks performed on a computer involve some text creation. Although there are some people who have a strong aversion to keyboards and prefer handwriting recognition to typing, for most users the fact that handwriting recognition is slower and more error prone than typing is a major drawback to stylus systems. Thus if the stylus is to become widely used as an input

device, we must identify common tasks for which a stylus is superior to a keyboard (and/or mouse). Our candidate for such a t¡ask is a markup editor, others are discussed in [16].

A typical scenario that can benefit from markup editing is when two people collaborate (possibly at remote sites) on a document. Author one produces a draft and gives it to author two. With current tools, author two has two choices. He can print the draft and mark it up with a pen, sending the marked up paper back to author one, or he can receive the draft online, edit it, and then resend it to author one. In the first case, author one can clearly see the changes, but must key in the edits by hand. In the second case the changes are already keyed in, but there is no easy way to see the changes.[3] A markup editor combines the best features of both approaches. The edits are made using marks such as strikeout to erase a word. The editor does **not** perform the edit and erase the word. Instead, it recognizes the edit (or gesture), and signifies its recognition by replacing the strikeout ink with a nicely drawn strikeout, in complete analogy with a character recognizer that replaces the ink of a hand drawn letter with one from a nicely tuned font. Because the gesture was recognized, the edit that it stands for can later be carried out automatically. Markup editors combine the ideas of gesture editing with two-view editors ([1] is a recent example) in that there are two views of the document: one with the edits as gesture marks, and the other with the edits applied.

In most previous work on gesture-based text editors, the gesture commands have either been carried out immediately ([2], [3], [4], [8], [14]), or the marks have been left unintepreted ([13]). The work of Suenaga and Nagura ([9]) is closest to ours. They mark up a printed (possibly handwritten) document with gestures, and then read the document and marks with a FAX receiver. The marks are interpreted, and a revised document is then printed on a FAX transmitter. [4]

Experience with our prototype suggests that markup editors have two

---

[3]Of course it is possible to develop a program that takes two input files and tries to show the changes by marking up the original, but this seems inferior because (1) edits can be made in more than one way, and the precise edit is lost and (2) marginal notes or "comments" (that is ink that is meant for the reader but performs no editing function) are lost.

[4]After completing this paper, we received a copy of [7], which describes the use of markup in a collaborative editor.

significant advantages that might make markup editors desirable even for ordinary editing tasks. First of all, undo operations become much more flexible. Traditionally, editors let you undo operations in reverse order, but you since can't see what those edits were, you can only infer them from watching the undo. A few editors (such as Tioga [10]) will display a list of your recent edits, but only as textual descriptions which are not the same as the mouse/keyboard actions that created the edits. With our editor, when a user undoes an edit, he makes a rubout gesture on top of the edit mark, so there is no ambiguity about what is being undone. Furthermore, the edits can be undone in any order: there is no longer a constraint that the edits be undone in the reverse order to which they were made. Of course, there is the complication of two different edit operations that interact,[5] but no matter what solution is provided to this problem, we feel that "random access" undo is a significant advantage to this type of editor.

The second advantage of markup editors is that they tie in very nicely with version control. At a certain point, the document will be filled with marks, and the user will wish to see what the document looks like when the edits are applied. This provides a natural checkpoint at which to make a new version of the document.

## 5.1 Analogies with Paper

One popular approach to stylus-based editors is to mimic what people do with pencil and paper ([15]). Although this is a valuable approach, we want to emphasize the opposite. After all, if a stylus-based system can do only what paper and pencil does, then why not just use paper? We propose that an important design principal for stylus editors (and stylus interfaces in general) is to try to go beyond what can be done with paper and pencil.

An example of an interface that uses a stylus like a pencil is Wang FreeStyle ([13]). In this system, to erase you turn the pen over and rub, just as you would with a pencil. Although this technique is easy to remember, it is not the best approach for stylus editors for the following reasons.

FreeStyle allows you to make only uninterpreted marks on paper, so the only operation (besides marking) is erasing. With a stylus text editor, how-

[5] For example, if you move a block of text, and then edit that text, what happens when you undo the move operation? Will that replace the original copy or the edited copy?

ever, you should also be able to move, copy and perform other editing actions. It doesn't make sense to single out one editing action (erasing) to be performed in a special way. Furthermore, the potential advantage of erasing with the other end of a stylus is that it is easy to remember; however, this advantage is minimal because remembering that action doesn't carry over to the other editing actions (eg. copy, move) that a stylus can perform.

Erasing by rubbing perpetuates an action which is clumsy and awkward to perform with a pencil. In one of our prototypes, we used the action of clicking a side button on a pen to indicate erase. Users loved it, because a simple click is much simpler and faster than having to turn a pencil (or stylus) around in your hand. This isn't to say that the best use of a side button is for erasing, but simply to indicate that a stylus affords much simpler methods of erasing than the use of more metaphorical techniques.

Here are two examples that arise in editing, which illustrate the principle that editor interfaces should go beyond simply mimicking paper and pencil practice.

First, consider moving text. The common practice with paper is to circle the source, then draw an arrow to the destination. A markup editor could use this as the gesture that it leaves on the screen. But it is possible to go beyond this and do something that can't be done easily with paper: the editor can also insert a copy of the source at the destination (preferably in a different font). This is especially useful when the source and destination are separated by many lines. Placing a copy of the inserted text at the insertion point is a simple idea, but one that is easy to overlook when merely imitating paper practice.

A second example is selection. With paper, selection tends to be done by circling, and this has been adopted by some systems (for example [4]). But as other have observed (e.g. [14]), it is awkward for selections that span multiple lines. A better idea is inspired by mouse-based systems: simply indicate the beginning and end of the selection, and let the computer highlight the rest.

# 6 Summary

We draw two main conclusions from our work.

First, it is possible to design a text entry interface that compensates for the imperfections of handwriting recognition algorithms. Unlike PenPoint,

our prototype allows users to correct errors without having to rewrite a misrecognized letter; in our system, a single tap corrects. We also observed that writer-dependent, boxed-text entry was more accurate than unboxed text entry, and we devised techniques (e.g. the segmenting cursor and the ability to modify the reference characters at any time) for making this type of input work more smoothly than the obvious implementation.

Second, we believe we have identified a type of stylus usage (markup editing) that can accomplish many editing tasks more easily than can be done with paper or with a keyboard/mouse system. An important principle in designing stylus-based user interfaces is to focus not on imitating paper, but rather on performing tasks that are difficult to do with paper. The two-view nature of our markup editor is an example of something that cannot be done with a paper-only system.

# 7 Acknowledgements

We would like to thank Jim Gasbarro who designed the scratchpad hardware, Russell Brown who implemented the character recognizer, Bill Buxton for many enlightening conversations, and Dan Swinehart for a careful reading of the manuscript.

# References

[1] Avrahami, G., Kenneth Brooks, M. H. Brown. "A Two-view approach to constructing user interfaces," *Computer Graphics* 23(3), 1989, pp. 137-146.

[2] E.R. Broklehurst. "The NPL electronic paper project," *Int. J. Man-Machine Studies* 34(1), 1991, pp. 69-95.

[3] Robert Carr and Dan Shafer. *The Power of PenPoint*, Addison-Wesley, 1991.

[4] Michael L. Coleman. "Text Editing on a Graphic Display Device Using Hand-drawn Proofreader's symbols," *Pertinent Concepts In Computer Graphics, Proceedings of the Second Univeristy of Illinios Conference on Computer Graphics*, 1969, pp. 282-290.

[5] Gearge R. Doddington and Thomas B. Schalk. "Speech recognition: turning theory to practice," *IEEE Spectrum*, 18(9), September, 1981, pp. 26-32.

[6] Aaron Goodisman. *A Stylus-Based User Interface for Text: Entry and Editing*, MIT Masters Thesis, 1991.

[7] Gary Hardock. "Design Issues for Line-Driven Text Editing / Annotation Systems," *Proceedings of Graphics Interface '91*, Calgary, Alberta, June 3-7 1991, Morgan Kaufmann, pp 77-84.

[8] Arto Kankaanpaa. "FIDS – A Flat-Panel Interactive Display System," *IEEE Computer Graphics and Applications*, 8(2), March 1988, pp. 71-82.

[9] Yasuhito Suenaga and Masakazu Nagura. "A facsimile based manuscript layout and editing system by auxiliary mark recognition," *Proceedings 5th International Conference on Pattern Recognition*, Miami Beach, Florida, December 1980, pp. 856 - 858.

[10] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. "A Structural View of the Cedar Programming Environment," *ACM Transactions on Programming Languages and Systems*, 8(4), October 1986, pp. 419-490.

[11] C.C. Tappert, C.Y. Suen, and T. Wakahara. "On-Line Handwriting Recognition — A Survey," *Ninth International Conference on Pattern Recognition*, 1989, pp. 1123-1132.

[12] Fumio Togawa and Hitoshi Hirose. "Handwritten Character-Recognizing Apparatus for Automatically Generating and Displaying Character Frames," *United States Patent 4,953,225*, August 28, 1990.

[13] Wang FreeStyle *SigGraph Video Review* 45(3), 1989.

[14] L.K. Welbourn, R.J. Whitrow. "A Gesture Based Text Editor," *Proc. of the Fourth Conference of the British Computer Society of Human Computer Interaction*, 1988, pp. 363-371.

[15] Catherine G. Wolf and Palmer Morrel-Samuels. "The Use of Hand-drawn Gestures for Text-Editing," *International Journal of Man-Machine Studies*, 27(11), 1987, pp. 91 - 102.

[16] Catherine G. Wolf, James R. Rhyne and Hamed A. Ellozy. "The Paper-Like Interface," in *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*, Edited by G. Salvendy and M. J. Smith, Elsevier Science Publishers, 1989, pp. 494 - 501.

XEROX Stylus User Interfaces for Manipulating Text David Goldberg & Aaron Goodisman CSL-91-9