Writing Handwritten Messages on a Small Touchscreen

Wolf Kienzle
Microsoft Research
Redmond, WA, USA
wkienzle@microsoft.com

Ken Hinckley
Microsoft Research
Redmond, WA, USA
kenh@microsoft.com

ABSTRACT
We present a method for composing handwritten messages on a small touchscreen device. A word is entered by drawing overlapped, screen sized letters on top of each other. The system does not require gestures or timeouts to delimit characters within a word—it automatically segments the overlapping strokes and renders the message in real-time as the user is writing. The auto-segmentation algorithm was designed for practicality; it is extremely simple, requires only public domain data for training, and runs very fast on low-power devices. Drawings may also be included with the text. Experimental data indicates the effectiveness of our system, even for novice users.

Author Keywords
handwriting; ink; messaging; notes; mobile input

ACM Classification Keywords
H.5.2. [Information interfaces and presentation]: User Interfaces – Input devices and strategies, Interaction styles

INTRODUCTION AND RELATED WORK
Handwriting offers advantages over typing for personal communication. Handwritten notes contain subtle personal cues through writing style and drawings that cannot be expressed by typed text. Many of today's communication devices—notably smartphones and tablet computers—have touchscreens capable of recording handwriting, and several commercial applications employ handwriting for note taking and messaging. Unfortunately, handwriting on a touchscreen requires either a stylus, or a surface large enough for a finger to write with sufficient precision. This poses a problem for the small screens of mobile devices.

In this paper we present a new method for entering handwritten notes on a small touchscreen. It allows the user to write words with their finger by drawing screen-sized characters on top of each other (Fig. 1). Our key contributions are 1) a straightforward interface that does not require the user to delimit characters by means of gestures or timeouts, and 2) a segmentation algorithm which is simple to implement, requires only public available training data, and runs extremely fast on low-power devices.

MobileHCI '13, August 27 - 30 2013, Munich, Germany

http://dx.doi.org/10.1145/2493190.2493200

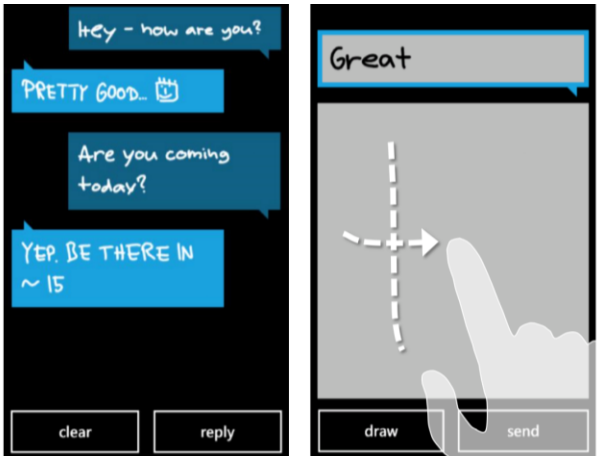

Fig. 1. Chat application for handwritten messages. *Left*: Conversation view. *Right:* A user writing the letter *t*.

Several commercial applications support small-screen writing. In *Use Your Handwriting* [7], the user writes characters in the phone's landscape mode. Touching the far left side of the screen (or waiting for a timeout) accepts the writing and clears the screen for more input. A downside of this approach is that the user must organize her input spatially such that 1) the "accept and clear" action is triggered, and 2) the proper amount of space surrounds each segment, since segments are concatenated by including leading and trailing whitespace. Also, the landscape orientation necessitates frequent rotation of the device. Another similar application, *Handwriting* [6], uses a two-finger swipe gesture to arrange ink segments on the screen. By contrast, our method does not require the user to manually arrange text segments, or attend to timeouts or trigger locations on the screen. Instead, we automatically detect overlapping segments and present the "untangled" result to the user in real-time, without a noticeable delay.

Automatic segmentation has been studied in mobile text entry: so-called *overlapped* recognition systems have been developed for English [1], Japanese [3], and Chinese [4]. Unistroke alphabets avoid the character segmentation problem, and can allow eyes-free input [12], but require the user to learn a new gesture set to enter text. *PocketTouch* [11] allows the user to write overlapping characters on a touch surface through fabric. It segments lowercase English characters using stroke classifiers and a set of simple rules, then converts them to text with a handwriting recognizer.

The key difference between recognition approaches and our method is that *we never recognize characters*—we only

detect character boundaries to untangle and render the handwritten message. Keeping users' personal handwriting in this way introduces an aesthetic quality that is not found in recognition systems (but see [13]). Furthermore, stroke segmentation is a much simpler problem than handwriting recognition. This keeps the computation efficient, allowing for true real-time feedback (at ~40Hz), even with our prototype implementation. Also, this approach facilitates unusual words or expressive spellings (e.g., "whoooah"), since our technique does not require a dictionary.

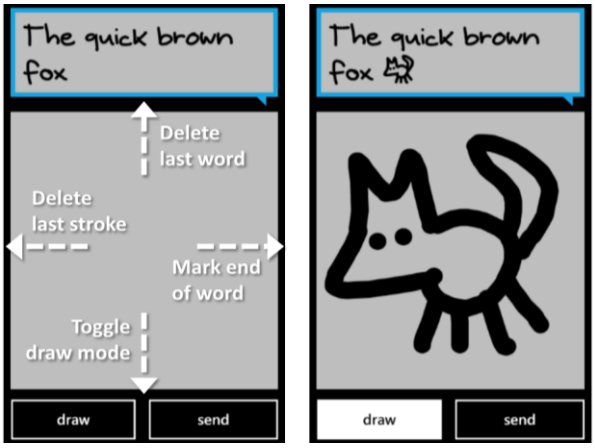

Fig. 2. ***Left:*** **Swipes across the writing pad are used to delete strokes, mark word ends, and toggle draw mode (*right*).**

USER INTERFACE

The proposed interface consists of a large writing pad (5cm × 5cm) and a display panel at the top (Fig. 1, *right*). The display panel is two lines of text high (2cm) and can be scrolled vertically using an up/down swipe gesture.

Writing Pad

The user can employ her normal writing style to make strokes on the writing pad, using upper and lower case English letters, numbers, and punctuation. The writing pad provides space for about 1-3 characters, depending on how large the user writes. The user can position and space characters horizontally without restriction; vertically, the full height of the writing pad always corresponds to one line of text. We provide swipe gestures (Fig. 2) to delete the last stroke (left), delete the last word (up), mark the end of a word (right), and switch between text and draw mode (down). To avoid accidental activation, swipe gestures must cross the outer boundary of the writing pad.

Display Panel

The display panel shows up to two lines of handwritten text. The current stroke (the one that is being drawn on the writing pad) gets updated upon every touch event on the writing pad. This allows the user to visually verify her input in real-time. Stroke segmentation and layout runs on a background thread that updates the display panel asynchronously. It gets triggered (if not currently busy) by every touch event and takes about 20ms to run. As a result, the display panel shows the final composed message evolving in real-time as the user writes on the writing pad.

To lay out a segmented word, we group strokes by character and align the character bounding boxes horizontally, separated by small spaces (about one stroke width). A wider horizontal space is inserted at the end of each word. Character positions are not adjusted vertically. Also, we retain the relative stroke positions from the writing pad *within* each character. In some cases the assumptions behind this simple scheme do not hold, e.g., if a user puts down the device in the middle of a symbol and tries to finish later. For the first version of our system, however, we found this simple and predictable approach work well. In the future this may be replaced by a more sophisticated layout algorithm.

Note that we do not provide visual feedback on the writing pad while the user articulates character strokes. During preliminary tests we experimented with a fading ink trace on the writing pad. It was unclear whether this helped users write more accurately. Also, some users reported that they felt more efficient attending to the display panel, which also shows their input in real-time, and in context of the entire word. For the present version of our system we decided to promote this behavior and make the interface less visually busy by not showing ink on the writing pad.

Draw Mode

The interface allows the user to add small drawings to the text. A drawing differs from a character in that stroke segmentation is turned off during its composition. To compose a drawing, the user brings the interface into draw mode by using the down gesture, or by toggling the draw button. In draw mode, ink is displayed on the writing panel, as shown in Fig. 2 (right). This is based on the assumption that unlike characters, drawings are custom creations that require potentially many strokes and a magnified view during composition. The draw mode toggle button was added to indicate that the system is in draw mode even if there are no strokes on the writing pad. Deletion gestures can be used in draw mode as well. To finish a drawing, the user either toggles the draw mode button or makes a word-end gesture. Drawings are treated as words when the message is displayed, i.e., they are separated from adjacent words or drawings by whitespace. There is no limit on the number of drawings in a message.

Every stroke has a bit associated with it that indicates whether it was entered in draw or text mode. Assume the user "backspaces" through a number of strokes and eventually deletes the last stroke of a drawing: at that point the system switches to draw mode and displays all strokes of that drawing on the writing pad, so that the entire drawing can be edited. Likewise, the system reverts to text mode if the user "backspaces" through the entire drawing.

Tap-to-Correct

As with any system that interprets natural user input, there are cases when the output does not match the user's intent. For our segmentation algorithm, errors result in characters rendered on top of each other, or broken up into pieces.

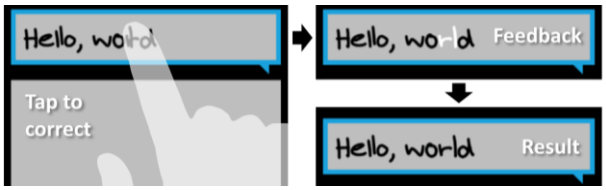

Fig. 3. Correction of a segmentation error. *Left:* user taps on mis-segmented part of the word. *Top right:* best alternate segmentation (see text) is used and affected strokes are highlighted for 200ms. *Bottom right:* result after correction.

We allow the user to correct segmentation errors by tapping on the mis-segmented part of the word in the display panel (Fig. 3). When the display panel is tapped, we find the most likely segmentation that differs at the tapped location (approx. 5mm radius), but keeps other parts of the word unchanged. This is described in more detail in the algorithm section below. The affected strokes are highlighted for a short amount of time to indicate what has changed. Tap corrections can be made any time anywhere on the display panel. The tap gesture has no effect on drawings.

SEGMENTATION ALGORTIHM

In our approach, the user only marks word boundaries by means of the space gesture ("mark end of word"); ink strokes within a word are automatically segmented into characters using the following method.

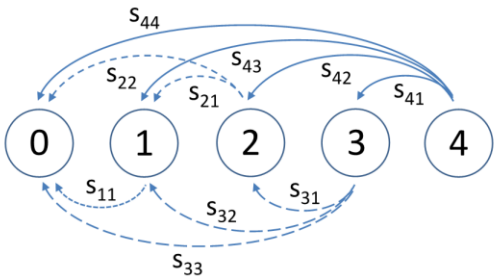

Fig. 4: The stroke segmentation graph for $N = 4$ strokes. Any path from node *4* to node *0* represents a stroke segmentation. Its score is the sum of traversed edge weights s_{ik}.

Assume we want to segment a word consisting of N strokes. For every stroke $i = 1...N$ we compute four scores s_{ik} $k = 1...4$ describing how likely stroke i is the last stroke in a k-stroke character. Now consider a graph (Fig. 4) with $N + 1$ nodes, one for each stroke $i = 1...N$ plus one extra start node labeled '0', and with directed edges pointing from each node back to its (up to) four predecessors. An edge from node i to node $i - k$ corresponds to grouping strokes $(i - k + 1),...,i$ into one character and is weighted by the corresponding score s_{ik}. Every path from node N to node '0' represents a segmentation of our N strokes into characters. To determine the most likely segmentation we find the path with the maximum sum of edge weights. This is a standard shortest path problem, solved efficiently using dynamic programming in $O(N)$ time and space [9].

Character Models

End-of-character scores are computed as $s_{ik} = f_k(v_i)$, where v_i is a vector of features describing stroke i, and f_k is a statistical model of how likely a stroke is the last one in a k-stroke character. The feature vector v_i contains 269 values, including the bounding box position, dimensions, area, aspect ratio, the stroke length, an eight-bin slope histogram, and slopes at the beginning and end of the stroke. It also holds the same numbers for up to two neighbor strokes in both directions, $(i - 2),...,(i + 2)$, provided these exist (missing values are set to -1). Finally, it includes bounding box features for the union and intersection of neighboring bounding boxes. We did not include timing information or features that depend on sampling rates; those can vary across devices. More features could be added if higher model accuracy is needed.

We use boosted decision stumps for our four end-of-character models: $f_k(v_i) = \sum_{j=1..n} h_{jk}(v_i)$, i.e. sums of n step functions h_{jk} on the individual dimensions of v_i. Models are trained discriminatively with *AdaBoost* [5], using all end of-k-stroke-character strokes in the training data as positive examples and the remaining strokes as negative examples.

Model Training

In a preliminary experiment, five participants wrote phrases on our test device. We recorded stroke data, manually labeled character boundaries, and trained our four character models. The accuracy of the resulting system was good, but not sufficient. To get significantly more training data we decided to simulate strokes using a publicly available data set of 11,000 handwritten English characters from 60 writers [2] and a corpus of the 25,000 most frequent words in English Twitter conversations [10]. To simulate a word entered on our writing pad, we draw a random word from the Twitter corpus. Then, for each character we draw a set of ink strokes from the handwriting data set. The position and size of the simulated strokes are randomly perturbed such that their mean and variance match the data from our preliminary (but real) data set. Relative stroke positions within a character are not randomized. This corresponds to the same assumption we make during rendering, i.e. that relative stroke positions are preserved within a character.

We generated 500,000 strokes and trained our character models with $n = 256$ stumps each. The training code was written in *Matlab* and took less than a day to run on a desktop computer. Segmentation accuracy of the full system was validated using a hold out set of 10,000 simulated words. We found that 7.7% of the validation words had segmentation errors. The system performance on real user data is described in the user study below.

CORRECTION ALGORITHM

When the user taps the display panel to correct a segmentation error, we find the closest word to the tap location and divide its strokes into three categories: strokes for which the segmentation should change (within 5mm of the tap location), may change (adjacent to must-change strokes), or is frozen (everything else). Then we compute the $m = 32$ most likely segmentations that are feasible, i.e. leave the frozen strokes' segmentation unchanged. We

achieve this by removing edges from the segmentation graph (Fig. 4) that would lead to alternate segmentations for frozen strokes, and finding the *m* highest scoring remaining paths. The correction segmentation is defined as the path that 1) changes at least one of the strokes that should change and 2) has the smallest impact on strokes that may change. This rule favors localized corrections, but also allows larger areas to change if the local change requires it. The full correction computation takes only about 25ms.

The corrected segmentation may be the one intended by the user. Depending on future experimental data we may need to extend the correction interface. However, anecdotal evidence indicates that the current approach is surprisingly effective considering its simple design (a single tap). A more sophisticated correction method may not be an improvement if it requires giving up this simplicity.

USER STUDY

We conducted a small field experiment to find out the degree to which users are interested in communicating via handwriting, and how effective they are at composing messages using our interface. We recruited five female and seven male subjects, between age 28 and 50, one of them left-handed. Nine subjects reported that they type messages on their phone at least once a day. We used an *HTC 7 Trophy* smartphone with the chat application shown in Fig. 1. After a one minute tutorial on how to write and use gestures, participants practiced using three randomly selected phrases from [8]. Then they entered ten more phrases for which we recorded gestures and timings. Subjects were asked to verify that messages are segmented correctly before they hit the "send" button.

Overall, subjects indicated they would use the system for personal correspondence (avg = 6.25, SD = 0.98, on a 7-point Likert scale, 1=strongly disagree, 7=strongly agree), and found it easy to use (avg = 6.41, SD = 0.67). In free-form comments, users liked the personal aspect (6x) and the simple/fun experience (4x). On the less positive side, users commented that letters were sometimes mis-segmented when written in a particular style (3x) and that sliding a finger required more manual effort than typing (3x).

Text entry speeds were averaged over the ten trials and ranged between 9.0 and 20.8 words per minute among the twelve subjects (avg = 12.7, SD = 3.5, median=11.1). Averaged across subjects, about one in eight words contained deletion or correction gestures (avg = 12.5%, SD = 9.4%, median = 10.1%, min = 2.7%, max = 30%). This is higher than the estimated 7.7% word error rate from our simulations. It indicates that our simulations do not capture some aspects of the real world, e.g., particular writing styles. The gap could also be caused by a different effect: for example, when writing the word 'cat', the vertical stroke of the 't' is often merged with the 'a' to form a 'd', since this is the most likely configuration at this point. As soon as the 't' gets crossed, however, it gets segmented correctly. We found that some subjects tend to correct prematurely in these cases, i.e. they delete and re-enter a stroke instead of providing the next stroke to resolve the ambiguity. This effect is not captured by our simulation.

CONCLUSION

We have developed a method for entering handwritten notes on a mobile touchscreen device. Initial feedback suggests that even novice users are able to use it effectively. Since we do not recognize or produce *text*, but rather only human-readable segmentations of strokes, there is not a clear basis to compare our error rates with those of existing text entry techniques, such as touch-screen keyboards. In practice, as observed in our studies, users found the word error rate (12.5%) well within the usable range—and similar to that of many touchscreen text entry techniques reported in the literature. The only errors in our system are either too much, or too little, white space between adjacent strokes. We observed that such errors have little impact on readability, or may even go unnoticed; by contrast, character-level errors in text entry systems can easily change meaning of words, especially in conjunction with a dictionary. Furthermore, if the user chooses to correct an error, a single tap on the mis-segmented character often suffices, which is a small cost compared to (say) correcting a misrecognized word in a handwriting recognition system.

We are exploring new application contexts, and ways to refine the technique with longitudinal data. More generally, our work illustrates an interesting class of technologically-mediated communication—one that preserves personal expression in conveying a message to another *human*—rather than imposing the taxes on users' time and attention that full-on handwriting recognition and error correction require for translation to machine-readable text.

REFERENCES

1. A. Bharath and S. Madhvanath. Freepad: a novel handwriting-based text input for pen and touch interfaces. IUI 2008, pp. 297–300.
2. D. Llorens et al. The UJI penchars database: A pen-based database of isolated handwritten characters. LREC '08 (Lang. Resources & Eval.).
3. H. Shimodaira et al. On-line overlaid handwriting recognition based on substroke HMMs. In Intl. Conf. Document Analysis and Recognition (ICDAR), pp. 1043–1047, 2003.
4. Y. Zou et al. Overlapped handwriting input on mobile phones. Intl. Conf. Document Analysis & Recognition (ICDAR 2011), pp. 369–73.
5. Y. Freund and R. Schapire. A decision-theoretic general-ization of on-line learning and an application to boosting. In Computational Learning Theory, LNCS Vol. 904, pp. 23–37. Springer, 1995.
6. Cocoa Box Design LLC. http://www.cocoabox.com/, 2010.
7. Gee Whiz Stuff LLC. http://www.geewhizstuff.com/, 2012.
8. I. MacKenzie and R. Soukoreff. Phrase sets for evaluating text entry techniques. CHI 2003 Extended Abstracts, pp. 754–755.
9. L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. Proc. IEEE, 77(2):257–286, 1989.
10. A. Ritter, C. Cherry, and B. Dolan. Unsupervised modeling of Twitter conversations. In Human Language Technologies, HLT '10, pp. 172–180. ACL, 2010.
11. S. Saponas, C. Harrison, and H. Benko. PocketTouch: through-fabric capacitive touch input. UIST 2011, pp. 303–308.
12. H. Tinwala and I. S. MacKenzie. Eyes-free text entry on a touchscreen phone. Toronto Intl Conf Sci & Tech (TIC-STH), pp. 83–88, 2009.
13. J. LaViola and R. Zeleznik. MathPad^2: A System for the Creation and Exploration of Mathematical Sketches. SIGGRAPH 2004, p. 432-40.