xstroke: Full-screen Gesture Recognition for X

Carl D. Worth
Information Sciences Institute
University of Southern California
Arlington, VA, 22203
cworth@isi.edu

Abstract

Gesture recognition is a common method of text input on handheld and other pen-based computing devices. Xstroke is a full-screen gesture recognition program for the X Window System.

The touchscreen of a typical pen-based device is divided into two regions, a primary region for application display and interaction, and a secondary region for gesture input. Full-screen gesture recognition improves on the typical implementation by sharing the entire screen for both purposes. Applications benefit as more screen real estate is available. Recognition performance improves due to increased information from larger gestures. Usability increases as less time is lost switching attention between two separate screen regions.

Full-screen gesture recognition presents several user-interface design challenges. Conventional GUI interaction is pointer driven. This makes it difficult for systems with single-button pointer devices to distinguish between GUI interaction and character input. Several solutions to this problem are examined.

The simple feature-based recognition engine at the heart of Xstroke has proven capable of resolving gesture sets of 100 different gestures with up to 95% accuracy. Xstroke is freely available and has become the predominant gesture recognition system for X-based handheld devices.

1 Introduction

Gestures, or marks entered with a stylus or mouse to invoke commands, have become an increasingly relevant aspect of user interfaces over the past few years. Gestures are an efficient means of command input since a gesture can simultaneously indicate both the operator and operand for a command. On the desktop, gestures have long been present in specialized fields such as CAD, but have recently begun to appear in general-purpose applications including text editors [3] and web browsers [13, 8].

Outside the desktop arena, gestures are perhaps more compelling since many handheld and other mobile computing devices provide a stylus as a primary input device. With these devices, software is required to allow the user to efficiently input text using the stylus. Gestural text input is possible if the gesture recognition system supports a gesture set large enough to cover every desired character. It is also desirable if the recognition system is capable of recognizing gestures that are similar to character shapes in their natural forms.

Xstroke is a gesture recognition system designed to add gestures to the desktop and to provide efficient character input for pen-based computers running the X Window System. The X Window System is the standard graphical user for UNIX and UNIX-like operating systems. The XFree86 project [15] distributes an implementation of the X Window System including a tiny X server known as kdrive [14] which is suitable for use on handheld computers.

xstroke has several distinguishing features:

Full-screen recognition xstroke allows gestures to be entered anywhere on the screen, directly on top of the program to receive the gesture command. The current stroke is drawn in translucent "ink" above other programs.

Extensible recognition xstroke includes a simple feature-based recognition engine, and can be extended with more sophisticated recognition engines through dynamically loaded libraries.

Adaptive slant correction xstroke automatically adapts its recognition to account for changes in the slant or rotation of input gestures.

Configurable alphabet xstroke includes a default gesture set designed to allow efficient use of xstroke as a keyboard replacement. The gesture→action mapping can be customized and custom gestures may be added to the gesture set.

1.1 Related Work

Gesture recognition can be added to user interface environments at several different layers including direct implementation within an application, recognition within a library or toolkit, or recognition through an external program. Each strategy has different implications on the effort needed to integrate the recognition support into

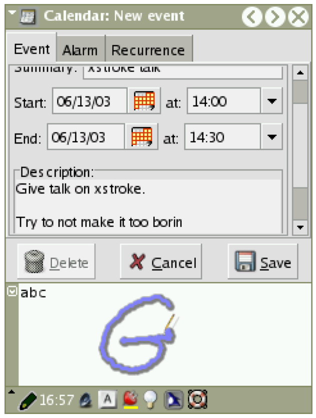

Figure 1: xstroke in window mode

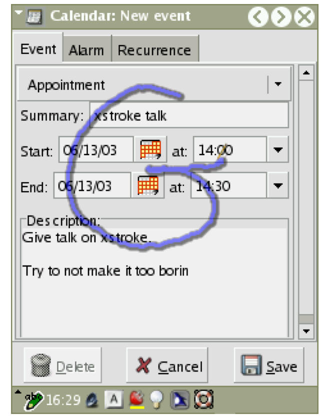

Figure 2: xstroke in full-screen mode

applications.

Open source/free software programs directly supporting gestures include emacs/xemacs [3] and mozilla [8]. In both of these programs, the recognition engine is included as an integrated part of the application and is not available to other programs.

If the recognition engine is placed within a library, it can be used by multiple applications. James Kempf developed a generic library interface [11] for handwriting recognition engines, which has not achieved widespread use. Mark Willey's recognizer, libstroke [16], is an example of a specific recognizer implemented as a library. Incorporating gesture support at this level requires modifying the application to pass gestures to the recognition library and act on the recognized results. libstroke has been used in modified versions of ggv, gEDA, and the FVMW window manager.

Several efforts at providing system-level gesture support implement recognition within the user-interface toolkit. This is the approach taken in SATIN [10] with a Java toolkit; Artkit [9], a custom toolkit supporting gestures; and in a modified version of the Garnet toolkit [12]. The benefit of toolkit integration is that the gestures become available to all applications using the toolkit, without any gesture-specific code required in the application. Naturally, applications designed without using the toolkit do not get the gesture support. Neither of the popular modern open source/free software toolkits, (KDE [5] and GNOME [6]), offer gesture support.

The final approach for integrating gesture recognition with the user interface is as a peer application. The benefit of this approach is that gesture support can be provided to applications without modification to those programs. Existing programs using this approach include xscribble and wayV [4].

The author's experience, (and frustration), with xscribble led directly to this work. Recognition performance was poor and there were no readily available documentation on how to re-train the recognizer. These frustrations led directly to the fact that all of the gesture definitions for xstroke are in human readable form and can be modified online with no need for external tools beyond the ability to edit a text file.

The author was unaware of wayV until after xstroke had been written. Like xstroke, wayV is a full-screen gesture recognition program for the X Window System. Gesture recognition is distinguished from conventional pointer interaction by the use of a modifier key or a specific pointer button. Configurable gesture actions include program execution and generation of keypress events.

2 Full-screen Recognition

Initial versions of xstroke required gestures to be entered into a window belonging to xstroke. Figure 1 shows xstroke being used in this way to enter text into a calendar application. Entering gestures into a separate window has several drawbacks:

- On small displays, the recognition window steals precious screen real estate from the active application. Typical handheld computers that run xstroke have a display size of 320×240 pixels. Application developers are hard-pressed to get all the information they would like into those pixels without one fourth of them being used for input and unavailable for output.
- Restricting gesture recognition to a window places a maximize size on the gestures that can be input. Smaller gestures make it harder for the recognition engine to extract the desired features.
- Gestures in a separate window cannot achieve their full expressive power. A gesture in a window can indicate an operation by its shape, but cannot indicate the operand for the operation by its position.

Current versions of xstroke allow gestures to be entered anywhere on the screen as shown in Figure 2. This has benefits that are the opposite of each of the drawbacks listed above: the entire display is available for active applications, gestures may be entered as large as desired leading to more accurate recognition, and the position of gestures may be used to indicate operands for gesture commands.

2.1 Resolving Pointer Ambiguity

Full-screen gesture input is a compelling feature, but it also provides some significant challenges in the user interface design. The main problem is that full-screen gesture recognition introduces an ambiguity in the handling of pointer motion events. Namely, does a sequence of pointer motion events indicate a gesture for recognition, or is intended for direct manipulation of graphical elements of the user interface, (ie. standard mouse interaction)? xstroke implements several complementary mechanisms for resolving this ambiguity.

2.1.1 Reserved Button

If multiple pointer buttons are available, xstroke can be configured to perform recognition only when a specific button is pressed. If a button can be dedicated to xstroke this approach is sufficient and can be quite satisfactory. However, for many handheld systems, the only pointer device is a simple touchscreen which acts as a single-button pointer device. In that case, other mechanisms are necessary for managing when recognition occurs.

2.1.2 Manual Toggling

xstroke provides a small control window with a button that can be used to toggle recognition. When recognition is enabled, xstroke intercepts all pointer motion between button-down and button-up and performs recognition on the resulting gesture. When recognition is disabled, all pointer events are handled as if xstroke was not present. This toggle capability provides the necessary mechanism to allow the user to control the handling of pointer events, but it is far from convenient. Moving the pointer back and forth from the active application where gestures are being performed to the control window is wasted user time. This is especially costly when the display is large, (causing the control window to be far from the active application), or when toggling recognition is frequent, (such as the need to continually scroll a document while writing large amounts of text).

2.1.3 Automatic Toggling

The author regards manual toggling of recognition as a feature of last resort. Gestures are powerful enough that full-screen recognition should always be enabled. Instead of requiring the user to turn recognition off to use a button or a scrollbar, the system should "know" that recognition is not desired on those elements and should automatically disable recognition when appropriate.

For the case of buttons, a simple solution proves quite effective. In the default configuration, xstroke recognizes a single pen tap as a gesture associated with an action to synthesize a button press event. Because of this, GUI buttons can be used directly without having to disable recognition.

The only remaining problem then is how to automatically disable recognition for GUI elements that require pointer motion while the button is pressed, (ie. click-and-drag events), for example, a scrollbar. The problem was solved for buttons by recognizing a gesture, then synthesizing pointer events. That approach does not work here since the gesture is not recognized until the pen is lifted, while the user will expect the scrollbar to respond while the button remains pressed.

We experimented with automatically disabling recognition based on a heuristic for determining if the window under the stroke "expected" recognition or not. The heuristic is to examine the event mask of the window at which the gesture begins to see if it has selected for KeyPress events. The rationale is that most gestures are used to send synthetic keypress events, so if a window has not selected for keypress events, then recognition can be disabled and pointer events can be passed through directly. This idea had some success. With xterm, xstroke could be used for recognition within the terminal window and for operating the scrollbar. With GTK+ applications, scrollbars and menus could be used without manually disabling recognition.

But, at the same time, the complex window hierarchies within some application cause this heuristic to fail, sometimes with disastrous results. For example, with Qt applications, the scrollbars and menus are still not usable without manually disabling recognition. This failure mode is acceptable.

However, with the window hierarchy of rxvt, for example, the child window does not select for KeyPress events, (instead a parent window does). In this case the xstroke heuristic disables recognition for the entire window, and xstroke cannot be used to input text into rxvt. A similar problem happens with some GTK+ dialog boxes with multiple text boxes. Recognition is disabled for the majority of the dialog box, except for strokes that actually began inside one of the text boxes.

These failure modes are so bad, and the heuristic fails often enough that it was decided that attempting context-sensitive enabling/disabling of stroke recognition based on heuristics is infeasible.

Context-sensitive recognition would still be very powerful—if it could be done reliably. One possibility is that a convention could be developed so that windows could explicitly state whether recognition should be enabled on each window or not. It is not anticipated that many applications will be modified to provide these hints. Rather, it is expected that toolkits will provide these hints for those classes of windows which require them, (such as scrollbars). A more general solution that would also be very desirable is if toolkits would provide some indication of the widget-class of each window, (eg. menu, button, scrollbar). Xt implements a feature like this with the _MIT_OBJ_CLASS name, but most modern toolkits provide no such indication.

With a few carefully placed (and standardized) hints in place within the toolkits, the pen should just do the right thing—allowing full-screen recognition, but also enabling scrollbars, menus, and other GUI widgets to still be manipulated.

2.1.4 Tap-and-hold

Finally, there is one more class of window that still needs to be discussed. Some windows want recognition events, but also act on click-and-drag pointer events. The most common example is a text widget which could use gesture recognition for text input, but also allows click-and-drag to select text. A simple hint controlling whether recognition should occur or not is not sufficient here. For this case, xstroke has another toggle mechanism that is considerably faster than the global toggle button in the control window. If a gesture begins with a tap-and-hold motion, (that is, if there is little or no pointer motion within a short timeout period following a button-down event), then xstroke will disable stroke recognition for a single gesture.

With this mechanism in place, the text widget becomes quite usable. In the most frequent usage, gestures are used on the text widget to input text. Then, when an occasional text selection is required, a quick tap-and-hold, (by default .75 seconds and naturally configurable), is all that is needed to make the pointer select text rather than recognize a gesture. xstroke also changes the pointer cursor, (from a pencil for recognition to the text selection bar), to indicate that the timeout has expired and the user can begin the selection operation.

3 Recognition

Within xstroke, gestures are recognized with a simple feature-based engine. An input gesture is captured as a sequence of time-stamped pen positions. It is then converted into a sequence of discrete feature values with the goal of selecting the correct target gesture.

The gesture feature vector is then compared with each configured gesture class. When the input gesture matches a gesture class exactly for each feature, that gesture class is returned as the recognized class.

The currently implemented feature recognizers are quite limited. But, the recognizer interface is well-defined so that it can easily be extended. New recognizers can be added as dynamically loaded modules for ease in experimenting. The following sections describe in detail the feature recognizers currently available in xstroke.

3.1 Grid Recognizer

The primary feature implemented in xstroke is the grid feature. This feature was inspired by the algorithm used in libstroke[16] though xstroke features a custom implementation with some improvements. The grid feature is based on a 3×3 grid numbered from 1 to 9. The grid is centered on the gesture and is generally scaled independently in the X and Y dimensions to fit the bounding box of the gesture as shown in Figure 3.

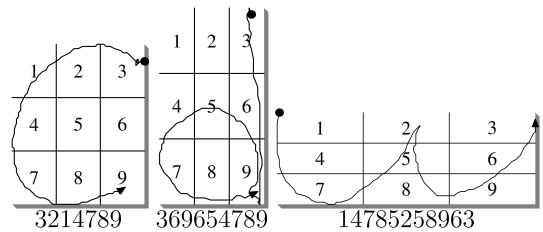

Figure 3: Grid fit to 3 different gestures

The automatic fitting of the grid allows for consistent recognition in spite of gesture variation in scale or in stretching in one dimension or the other. However, for some strokes that are very wide or very tall, the deformation introduced by scaling the grid will have a negative impact on recognition. For example, this grid scaling would introduce an ambiguity between a gesture consisting of a $45°$ line and a gesture of a very nearly horizontal line. To avoid these errors, the following rule is applied. Whenever the bounding box of a gesture is more than 4 times larger in one dimension than the other, the grid is fit to the smallest square containing the gesture. This is demonstrated in Figure 4.

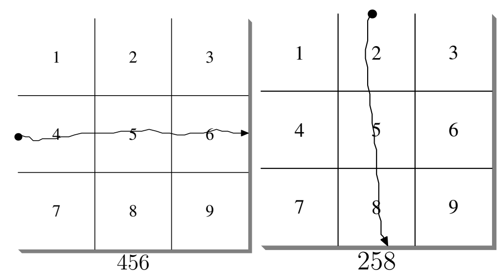

Figure 4: Grids constrained to be square for wide and tall gestures

Once the grid has been fit to the gesture, the feature value is determined by simply traversing the gesture points in time order and recording the number of each grid cell as the gesture passes through it. For example, the horizontal line in Figure 4 has a value of `456` while the 'C'-shaped stroke of Figure 3 has a value of `3214789`.

There are two details in the extraction of the grid cell sequence worth noting. First, depending on the sampling rate of the input device and the speed of the gesture input, there may be gaps between samples large enough to cause a cell to be missed. To avoid this problem xstroke uses linear interpolation between every sample point, (using a Bresenham algorithm), and examines the cell of every interpolated point.

Second, the gesture may pass through a cell so briefly that perhaps that cell value should be considered noise and discarded. Libstroke [16] attempts to solve this problem by disregarding cells which are passed through by some small percentage of the total number of sample points. Early experiments using libstroke within xstroke demonstrated two problems with this approach. First, discarding cells can result in an "impossible" digit sequence with the gesture jumping from one cell to a non-adjacent cell. Second, as gestures become more complex, the static threshold results in a greater number of cells being discarded. With extremely long gestures, an empty digit sequence may be generated.

In order to avoid discarding useful information describing the shape of the gesture, xstroke does not attempt to filter the grid digit sequence. As a result xstroke must be able to match sequences with more noise, (and hopefully more information), than those provided by libstroke. The sequence matching technique used in xstroke is described below.

3.2 Regex Sequence Matching

After generating a digit sequence for a gesture, libstroke, (as well as early versions of xstroke), uses exact string comparison to match the sequence with configured gestures. This approach can be problematic. Consider a gesture that is a diagonal line from the upper-right to the lower-left, (this is the default gesture in xstroke for the Return key). The ideal digit sequence for this gesture would be `357`, but since the ideal stroke passes near cell intersections in the grid, significant variation in the digit sequence can result in practice. Figure 5 demonstrates two of the many possible digit sequences that might result.

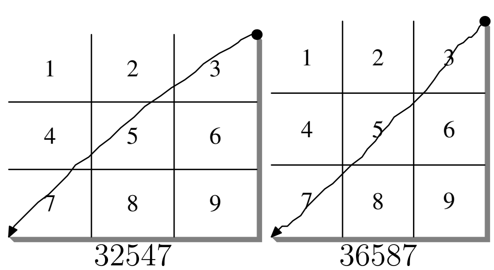

Figure 5: Similar strokes with different digit sequences

In order to reliably recognize this gesture with exact string matching, the recognizer would have to be configured with all likely variations of the digit sequence: `357`, `3257`, `3657`, `3547`, `3587`, `32547`, `32587`, `36547`, `36587`.

For this very simple gesture class, it is necessary to store 9 separate strings in the recognizer! This gets tedious very quickly, especially since an increase in stroke complexity yields an exponential increase in the number of possible sequences, (generally 3^N possible sequences for a stroke passing near N cell intersections). Clearly, this is not a scalable approach.

The solution in xstroke is that the grid feature value for a class of gestures is represented by a regular expression. For the case above, the default xstroke configuration contains the following specification: `Return = grid("3[26]?5[48]?7")` This regular expression provides an efficient way to store the class of 9 different sequences. The matching of a gesture sequence with a class is also efficient as regular expression matching is generally provided by a highly optimized system library.

This same technique can be used to encode robust gesture classes for sophisticated shapes. As a rather extreme example, the following regular expression represents a robust class of 'B' shapes for xstroke: `b=grid("([12]*[45][78]|[12][45]+[78]?)?[78]*[4]*(1?[2][369]+|1[25][369]*)([369]+[25]+8?[147]?[258]*[369]+|[25]*8?[147]+[258]+[369]*)([369]*[58][74]+|[369]+[58][74]*)").` This single expression describes the three ideal variations of the 'B' shape shown in Figure 6 along with the myriad sequences that may result when these gestures are input.

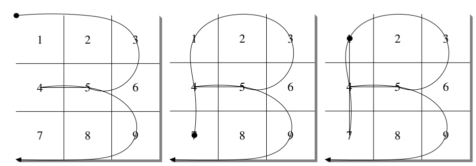

Figure 6: Three ideal 'B' shapes

The expression above can be daunting to read, (let alone attempt to understand). Its existence demonstrates that xstroke did not achieve all of its original goals — it had been hoped that the xstroke configuration file would always be easy to read and edit.

But the complex 'B'-shape expression also proves an interesting result. The author originally chose the grid recognizer since it would be so easy to implement. It was never expected that it would be able to handle anything beyond the most simple gestures. But, with expressions such as the one above, this simple recognizer is capable of distinguishing some rather sophisticated gestures. The grid recognizer has been so successful that investigation into other recognizers has proceeded slowly since the grid recognizer is often good enough.

3.3 Anchor Recognizer

xstroke contains a special-purpose recognizer known as the anchor recognizer. This recognizer cannot distinguish much of the shape of a gesture, but is used to "anchor" a gesture in absolute position and scale. For example, this allows xstroke to distinguish a gesture made in the middle of a touchscreen from a gesture of a similar shape that begins and ends at the screen edges. The anchor recognizer is based on the grid recognizer but has two important differences.

First, for the anchor recognizer the 3×3 grid has cells of non-uniform size. The center cell is enlarged to fill most of the grid area (roughly 80as very small rectangles positioned in each corner. The second difference is that the anchor grid is not scaled to fit the bounding box of the gesture but is scaled to cover the entire gesture area.

Figure 7 shows two gestures of similar shape; the grid recognizer returns the same digit sequence for each gesture. However the absolute position of the gestures is different and this is distinguished by the anchor recognizer. In practice, users of xstroke have found these global edge-to-edge gestures useful for strokes useful for invoking various operations. For example, a gesture from the bottom of the screen to the top might launch a program to display the currently configured xstroke gesture, or a diagonal slash from one corner of the screen to the other might close the current application. Using anchor, the gesture shapes for each letter in the alphabet can be reused for new operations besides their standard use for character input. For example, an edge-to-edge 'e' shape might be used to launch an email program.

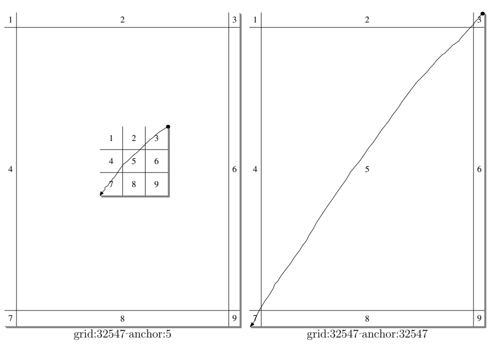

Figure 7: Anchor recognizer distinguishes absolute position and scale

3.4 Adaptive Slant Correction

There is a significant amount of variation in gesture orientation from one user to the next. In addition to the natural differences of slant due to different writing styles, there is added slant when xstroke is used on a handheld computer. Typically, one hand holds the computer while the other hand holds the pen. Due to the physical arrangement of human shoulders and arms, the most natural way to hold the computer in this way does not lend itself to precisely upright character entry. [XXX: It might be interesting to measure the natural slant in various users simply holding the device in a comfortable way].

Uncorrected slant leads to misrecognition as many desirable strokes differ only in orientation. Initial testing of xstroke without slant correction showed some users unable to achieve acceptable recognition rates. The need for slant correction has been shown in various other character recognition systems as well [7].

To correct for this, xstroke allows an OrientationCorrection action to be associated with any gesture. The OrientationCorrection action accepts a numeric argument giving the ideal orientation of the gesture that was recognized. For example, the left-to-right gesture used for Space in the default alphabet has an OrientationCorrection of 0 degrees while the right-to-left gesture used for BackSpace has an OrientationCorrection of 180 degrees. All of the simple one-line strokes have similar OrientationCorrection actions associated with them.

Whenever xstroke correctly recognizes a gesture with an OrientationCorrection action, it measures the actual angle from the first point of the stroke to the final point. It subtracts from this angle the ideal angle giving in the OrientationCorrection action to computes the expected slant at which the user is holding the device. xstroke uses an average of the last 5 values computed in this way as an estimate for the current gesture orientation. Then, as each gesture is performed, before the gesture is passed to the recognition engine, it is rotated by the current orientation estimate to compensate for any slant.

This system has proved quite effective at helping to alleviate problems with recognition from holding the device at various angles. This system gets great benefit from the fact that the Space gesture, the most common character in text, provides orientation information. Also, in the presence of recognition errors due to misestimated orientation, the Backspace gesture will often be performed to correct the errors, and will provide the needed information for orientation correction as well.

It may seem strange that the orientation correction actions only occur after characters are correctly recognized. If the current orientation estimate is incorrect, how will the system correctly recognize a gesture in order to get an OrientationCorrection action to improve the estimate? From casual observance of users, it appears that when users struggle with repeated recognition problems, they naturally write more carefully and more upright. Then, as xstroke adapts to their preferred orientation, the user can drift into a more comfortable position for holding the device.

4 Customization

Like many successful open systems, xstroke is designed to be modified and extended. The goal has been to design a system with sane defaults in which all policy decisions can me customized by the user. Several aspects of the user-interface can be configured such as which button triggers recognition and whether recognition is limited to a single window or is available full-screen. xstroke also allows the customization of the gesture set and extension of the set of feature recognizers. These aspects are discussed in detail below.

4.1 Default Gesture Set

The current release of xstroke contains pre-defined gestures for all printable ASCII characters, (eg. letters a-z, digits 0-9, and punctuation). It also includes additional gestures necessary to function as a full keyboard replacement Space, BackSpace, Return, Tab, Escape, Shift, Control, Meta, etc.

The default gesture set is available in two forms. In the "Basic Set" , (as illustrated in Figures 8-11), the gestures for letters, numbers, and punctuation are each made available as independent gesture sets called modes. There are also additional mode-changing gestures to change the current mode either for a single stroke, (ie. a single punctuation character), or until the next mode change, (ie. to input a large set of numbers). The most commonly used gestures such as Space, BackSpace, Return are available in a "Global Mode" that is always active.

The basic gesture set does include several variations for many letters. These are intended to accommodate multiple writing styles and to improve efficiency. For example, the vertical bar on may letter shapes, (eg. B, D, R), is optional which allows faster gesture entry.

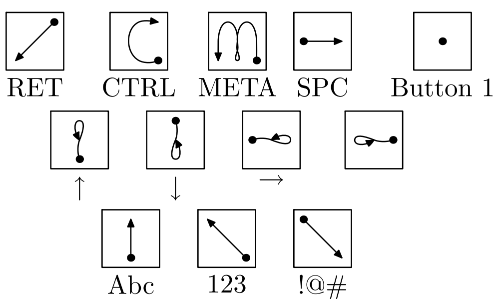

Figure 8: Global Gestures

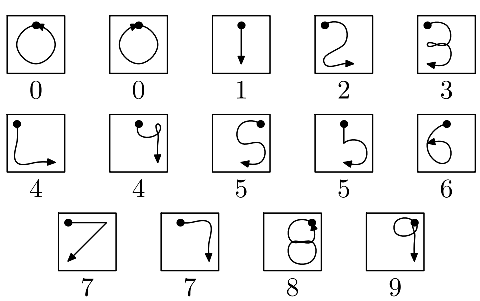

Figure 10: Basic Gestures: Numbers

The second form, the "Advanced Set", does not require separate modes. A single gesture set contains gestures for all letters, numbers, and punctuation. This began as an experiment to determine if a single gesture set could include enough independent gestures to act as a keyboard replacement. The author was pleasantly surprised to find that the simple grid recognizer is capable of distinguishing such a large set of gestures.

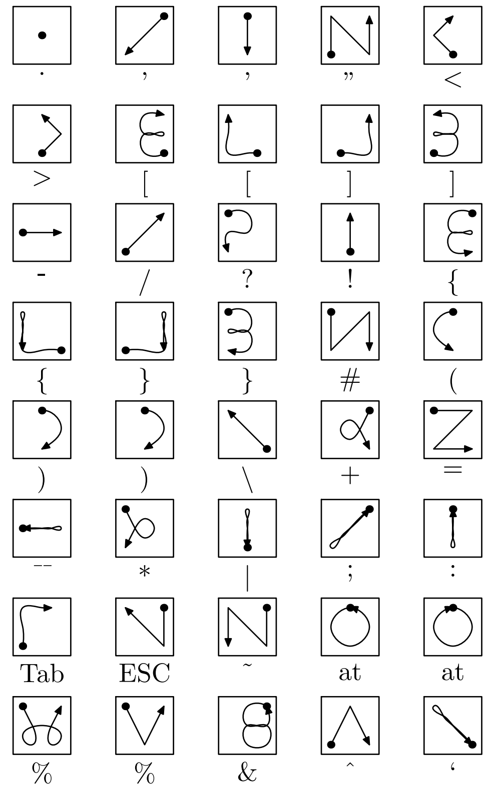

Figure 11: Basic Gestures: Punctuation

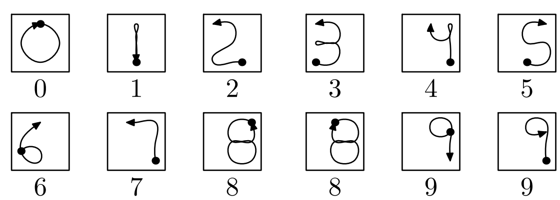

Figure 12: Advanced Gestures: Numbers in letter mode

Cramming every key on the keyboard into a single gesture set requires some creativity to avoid several clashes of similar shapes (eg. 'O' vs. '0', 'S' vs. '5', etc.). The "advanced" gesture set solves these problems by including gestures for each number that are drawn in the opposite direction, (eg. the gesture for the letter 'O' is drawn in a counter-clockwise direction while that for the number '0' is drawn clockwise, 'S' is drawn from top to bottom while '5' is drawn from bottom to top, etc.).

The advanced gesture set is a strict superset of the basic set, so the advanced set is enabled by default in the current system. The only reason to distinguish between the two sets is that the documentation for the basic set is simpler and more suitable for new users.

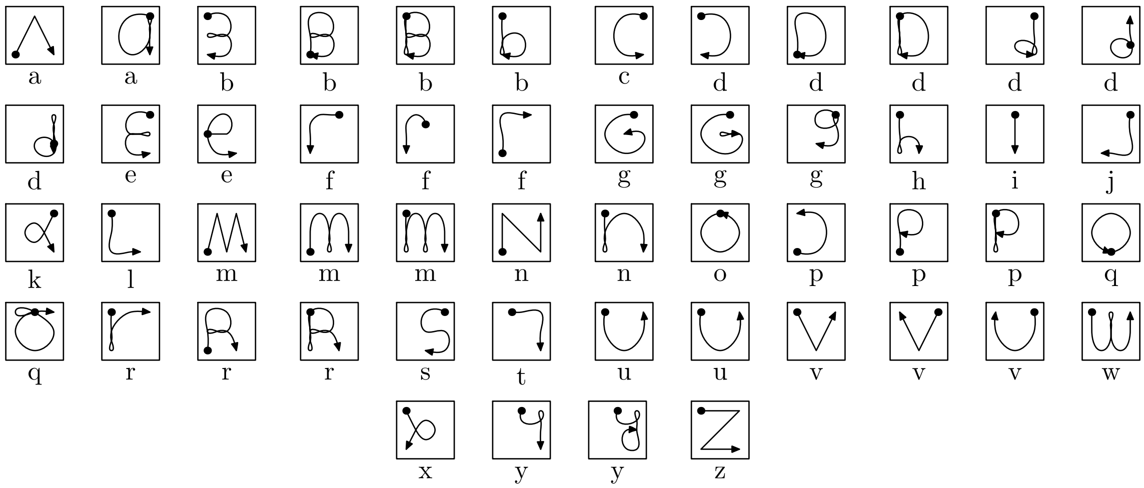

Figure 9: Basic Gestures: Letters

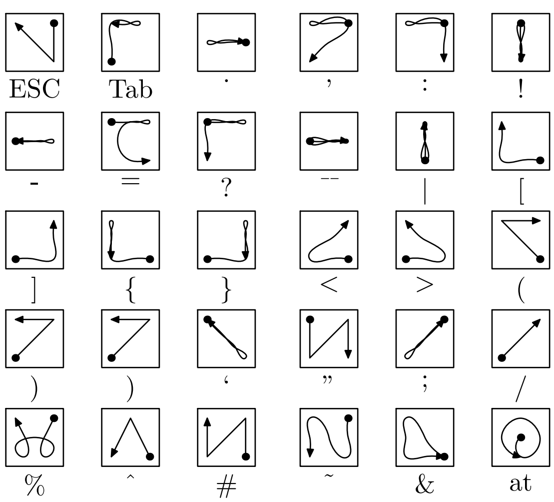

Figure 13: Advanced Gestures: Punctuation in letter mode

4.2 Customizing Gestures

The gesture set for xstroke is contained in a plaintext configuration file to make it easy to customize gestures and their associated actions. One common customization desired by many users is the ability to input accented characters. A simple approach is to change the gestures for standard punctuation symbols into their “dead key” equivalents. This enables multi-stroke combinations to be used to input accented characters in the exact same manner as on a keyboard.

There is a significant potential problem caused by allowing the end-user to customize the actual gesture shapes, (as opposed to the gesture actions). Namely, good gesture set design is a difficult task.

One aspect of the difficulty of gesture design is that visual similarity of gestures can make it hard for users to remember gestures correctly [1]. The author does not claim to be an expert in gesture set design nor does he claim understanding of human perception of similarity. It is hoped that exposing customizable gestures will allow users to choose gestures that are easy to remember for themselves individually. In fact, a survey of PDA users found that in spite of problems with gesture memorability, users actually want to be able to design new gestures [2].

A second difficulty in gesture set design is the need to understand the gesture recognition engine in order to be able to determine if two different strokes can be reliably distinguished. An intentional design decision of the feature recognizers within xstroke can help with this problem. For all xstroke recognizers, the features were designed so that feature values would be meaningful and useful to humans.

This allows the possibility for the recognition system to reason about ambiguities in the gesture set and report them in a way that is meaningful to the user. This has already proven useful in improving several misclassification problems in xstroke, (such as the difficult distinction between 'v' and 'u').

4.3 Extending the Recognizer

The emphasis on human involvement in the training process is not meant to suggest that automated techniques should not be used. It would be very convenient for users to be able to specify new gestures by example, and xstroke could benefit greatly from machine learning techniques to adapt to the user. However, limitations may exist within a feature set that no amount of machine learning can overcome. This is one reason xstroke has a well-defined mechanism for loading additional feature recognizers through dynamic libraries.

As a concrete example, consider Figure 14 which shows two 'h'-shaped gesture and an 'L'-shaped gesture. The second 'h' is misclassified as an 'L' since the hump is not larger than 1/3 the height of the gesture. This is one of the most common current misclassification problems experienced by the author with xstroke.

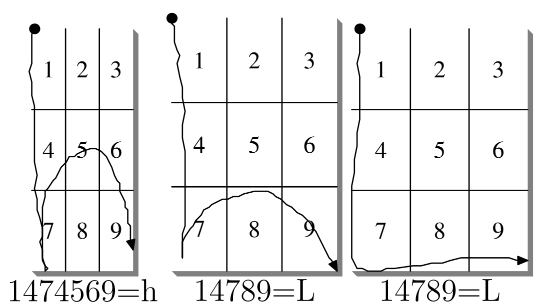

Figure 14: Misclassification of 'h' as 'L'

This misclassification could not be fixed through retraining. If a user retrained the misclassified 'h' as an 'L' all subsequent 'L'-shapes would be misclassified as 'h'. This would not be obvious to a user dealing exclusively at the level of gestures. But when looking at the grid feature values the problem is immediately apparent, and it is obvious that no quick fix is available since two distinct shapes both have a feature value of `14789`.

The only solution to this problem is to introduce a new feature which could distinguish between these shapes. xstroke currently has an experimental direction recognizer intended to address this problem. Just as the grid recognizer creates a short string based on quantizing 2D position into 9 values, the direction recognizer creates a string based on quantizing the stroke direction at each point into 8 directions. With the addition of this recognizer, the 'h' and 'L' strokes could then be easily distinguished:

$$\begin{aligned} \mathrm{h} &= \mathrm{grid}(\texttt{14789}) \cap \mathrm{direction}(\downarrow\nearrow\rightarrow\searrow) \\ \mathrm{L} &= \mathrm{grid}(\texttt{14789}) \cap \mathrm{direction}(\downarrow\rightarrow) \end{aligned}$$

5 Evaluation

Evaluating a new recognition system such as xstroke is a difficult task. It would be ideal to have access to a large database of multi-user gesture input data. Some large databases of gesture and handwriting samples are available commercially, but the data could not be redistributed, so those databases are not feasible for an open source project such as xstroke. It would be an interesting project to assemble an open corpus of stroke data, (which is especially feasible since in recent years there has been a great increase in the number of users of open source software that have access to touchscreen devices).

The custom gesture set in xstroke also complicates testing since it may be difficult to separate measurements of the performance of the recognizer from measurements of the level of expertise that the users have with the custom gesture set.

The author must certainly be considered an expert user having designed the gesture set. The author's own gestures were also the primary source of data for the manual training of the recognizers. The author typically achieves successful recognition of 95% of gestures input. Of the unsuccessfully recognized gestures roughly half are misclassified and half are rejected.

xstroke has supplanted xscribble as the standard character recognition program in the X11-based distributions from handhelds.org. There is anecdotal evidence to suggest that some users find xstroke to have more reliable recognition than character recognition software on popular commercial PDAs.

This paper has introduced several issues such as the user-interface issues of Section 2 and the advanced gesture set of Section 4.1. It would be very interesting to see a usability study investigating these issues, but such a study is beyond the scope of the current work.

6 Future Work

As discussed in Section 1.1 there are several approaches that can be used for integrating gesture recognition into a user-interface. The current mechanism of xstroke as an independent application requires the least modification to other programs, but also fails to provide tight integration with applications. It would be extremely useful to preserve the capabilities of the current xstroke interface but to split the actual recognition engine into a library that could also be used by other applications and toolkits.

Tighter integration with other applications would allow those applications to act on aspects of the gesture itself, rather than just the synthetic event triggered by recognition. For example, a scribble gesture might be used to indicate the deletion of the text covered by the gesture.

A significant limitation of the current user-interface is that only single-stroke gestures are collected for recognition. There is no inherent limitation in the recognition engine that would prevent it from recognizing multi-stroke gestures. This capability would allow much more natural letter forms to be used.

It would be very beneficial to have new tools for the configuration, training, and user adaptation of the recognizer. The extensible feature recognition system allows further research into more sophisticated recognizers. It would be very useful to implement a machine learning algorithm within xstroke.

7 Availability

xstroke is free software distributed under the terms of the GNU General Public license (GPL). It is available from http://www.xstroke.org.

8 Acknowledgements

The most sincere thanks to my wife without whose patience this work would have never been completed.

Many thanks to Keith Packard for patiently explaining details of the X Window System, devising the technique for rendering the translucent, shadowed brush, and for suggesting the idea behind adaptive orientation correction.

The author owes a tremendous amount of sushi to Keith Packard and Bart Massey for tireless efforts at improving the presentation of this paper.

9 Disclaimer

Portions of this effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0529. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

References

[1] Jr. A. Chris Long, James A. Landay, Lawrence A. Rowe, and Joseph Michiels. Visual similarity of pen gestures. In *Proceedings of the CHI 2000 conference on Human factors in computing systems*, pages 360–367. ACM Press, 2000.

[2] Jr. Allan Christian Long, James A. Landay, and Lawrence A. Rowe. PDA and gesture use in practice: Insights for designers of pen-based user interfaces. Technical report, U.C. Berkeley, UCB//CSD-97-976, 1997. URL `http://bmrc.berkeley.edu/research/publications/1997/142/142.html`.

[3] David Bakhash. Strokes package for xemacs, 1997. URL `http://www.mit.edu/people/cadet/strokes-help.html`.

[4] Mike Bennett. wayV description (web document). URL `http://www.stressbunny.com/wayv/`.

[5] Kalle Dalheimer. KDE: The highway ahead. In *Linux Journal*, number 58. Feb 1999.

[6] Miguel de Icaza. The GNOME project. In *Linux Journal*, number 58. Feb 1999.

[7] Yimei Ding, Fumitaka Kimura, and Yasuji Miyake. Slant estimation for handwritten words by directionally refined chain code. In *Proceedings of the Seventh International Workshop on Frontiers in Handwriting Recognition*, pages 53–61, Sep 2000.

[8] Andy Edmonds and Pavol Vaskovic. Optimoz mouse gestures, 2001. URL `http://optimoz.mozdev.org/gestures/`.

[9] Tyson R. Henry, Scott E. Hudson, and Gary L. Newell. Integrating gesture and snapping into a user interface toolkit. In *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology*, pages 112–122. ACM Press, 1990.

[10] Jason I. Hong and James A. Landay. Satin: a toolkit for informal ink-based applications. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 63–72. ACM Press, 2000.

[11] James Kempf. Integrating handwriting recognition into unix. In *Proceedings of the USENIX Summer 1993 Technical Conference*, Jun 1993.

[12] James A. Landay and Brad A. Myers. Extending an existing user interface toolkit to support gesture recognition. In *INTERACT '93 and CHI '93 conference companion on Human factors in computing systems*, pages 91–92. ACM Press, 1993.

[13] Michael Moyle and Andy Cockburn. Gesture navigation: An alternative 'back' for the future. In *Extended Abstracts of ACM CHI'2002 Conference on Human Factors in Computing Systems*, pages 822–823. ACM Press, Apr 2002.

[14] Keith Packard. Xkdrive manual. URL `http://www.xfree86.org/current/Xkdrive.1.html`.

[15] The XFree86 Project. Xfree86 (web document). URL `http://www.xfree86.org`.

[16] Mark Willey. Design and implementation of a stroke interface library. URL `http://www.etla.net/libstroke/`.