HAL open science

Using Strokes as Command Shortcuts: Cognitive Benefits and Toolkit Support

Caroline Appert, Shumin Zhai

To cite this version:

Caroline Appert, Shumin Zhai. Using Strokes as Command Shortcuts: Cognitive Benefits and Toolkit Support. International conference on Human factors in computing systems, 2010, Boston, MA, United States. pp.2289-2298, 10.1145/1518701.1519052 . inria-00538372

HAL Id: inria-00538372

https://inria.hal.science/inria-00538372

Submitted on 23 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Using Strokes as Command Shortcuts: Cognitive Benefits and Toolkit Support

Caroline Appert[1,2]
[2]LRI - Université Paris-Sud
Orsay, France
appert@lri.fr

Shumin Zhai[1]
[1]IBM Almaden Research Center
San Jose, CA, USA
zhai@us.ibm.com

ABSTRACT
This paper investigates using stroke gestures as shortcuts to menu selection. We first experimentally measured the performance and ease of learning of stroke shortcuts in comparison to keyboard shortcuts when there is no mnemonic link between the shortcut and the command. While both types of shortcuts had the same level of performance with enough practice, stroke shortcuts had substantial cognitive advantages in learning and recall. With the same amount of practice, users could successfully recall more shortcuts and make fewer errors with stroke shortcuts than with keyboard shortcuts. The second half of the paper focuses on UI development support and articulates guidelines for toolkits to implement stroke shortcuts in a wide range of software applications. We illustrate how to apply these guidelines by introducing the *Stroke Shortcuts Toolkit* (SST) which is a library for adding stroke shortcuts to Java Swing applications with just a few lines of code.

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: User Interfaces—*Graphical user interfaces*

Author Keywords
Gesture, Stroke, Shortcuts, Toolkit

INTRODUCTION
Invoking a command in a graphical user interface can usually be done through three different means: finding and clicking its label in a *menu*, finding and clicking its *icon* (e.g. toolbar buttons) or recalling and activating a *shortcut*. The most common type of shortcuts is typing a sequence of keys, known as keyboard shortcuts or hotkeys. Gesturing strokes is an alternative or complementary type of shortcuts that is also used but only in a few products. For example, the Opera[1] and the Firefox[2] web browsers have a set of strokes to open a new window, close a tab, etc. However, while the UI community has a longstanding interest in strokes as an interaction medium [24, 23, 7] and developed several research GUI prototypes, e.g. [17, 25], our understanding in this subject is still rather limited. In particular, there is a lack of basic cognitive and motor performance measurements of stroke gestures as command shortcuts in comparison to the standard keyboard shortcuts.

Traditional keyboard shortcuts have their limits in many situations. First, studies show that users often have difficulty to transition from menu selection to keyboard shortcuts [14]. Second, keyboard shortcuts may not be convenient to use, particularly for a growing number of non-traditional computing and communication devices. For example the iPhone and the pen-based Tablet PCs either have no keyboard at all or require the user to manipulate the screen to make the keyboard accessible. Enabling the users to efficiently trigger a command with a stroke gesture would overcome some of these problems, complement our current interaction vocabulary and enhance user experience.

In this paper we formulate and investigate the following hypothesis: stroke shortcuts may have a cognitive advantage in that they are easier to memorize than keyboard shortcuts. To better support recall, designers should make the shortcuts as analogous or mnemonic to the command name or meaning as possible (See related work on icons [19]). However *arbitrary mappings* are unavoidable since many concepts in the digital world do not offer a direct metaphor to the physical world. Interestingly, because strokes are spatial and iconic, which makes richer and deeper processing possible in human memory [3] even if the mapping is arbitrary, we hypothesize that stroke shortcuts could have cognitive (learning and recall) advantages over keyboard shortcuts.

We test this hypothesis from a user behavior perspective. Complementarily from a system design and development perspective, we articulate a set of guidelines for developing easy to use stroke shortcuts toolkits. As a first step in this direction, we present the *Stroke Shortcuts Toolkit* (SST) that integrates stroke shortcuts in a widely used development environment, the Java programming language and its Swing toolkit. Combining both types of contributions, we hope to encourage broader and faster adoption of stroke shortcuts in real world applications.

[1]http://www.opera.com/products/desktop/mouse/
[2]http://www.mousegestures.org/

ACM, 2009. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *CHI 2009* (April 4 - 9, 2009, Boston, Massachusetts, USA) http://doi.acm.org/10.1145/1518701.1519052

RELATED WORK

Strokes and commands

The best known work on using strokes to activate commands is probably the marking menus designed by Kurtenbach and Buxton [11]. A marking menu is a circular menu displayed after a delay so expert users who have already learned the menu layout can stroke ahead without the visual feedback. Extensions to marking menus include simple mark hierarchical marking menus [27] and the Hotbox [12].

Kurtenbach and colleagues also proposed a technique that can handle a larger vocabulary of gestures in the Tivoli system [13]. In that system, if the user does not know which gestures are available or how to gesture a command, he presses down the pen and waits for a *crib-sheet* to display the commands and their corresponding gesture strokes available in the current context. The Fluid Inking [25] system proposes a similar approach: to discover the available strokes, users invoke a marking menu in which an item is composed of a command name and the corresponding stroke description (in words), such as "Select (Lasso)". Neither crib-sheets in Tivoli nor the augmented marking menus in Fluid Inking have been experimentally evaluated.

Command strokes (CSs) proposed by Kristensson and Zhai [10] took another approach. CSs are based on the ShapeWriter text input system [26, 9]. With ShapeWriter, instead of tapping a sequence of soft keys the user draws a stroke that approximately links the letters of the intended word on a soft keyboard. To invoke a command, the user shape writes the name, or a part of the name, of a command prefixed by the special *Command* key. With CSs, users were able to invoke a command 1.6 times faster than selecting an item in a pull-down menu. Obviously, CSs require the presence of a soft keyboard which takes some valuable screen space.

Shortcuts and memorization

Most studies on command input focus on the execution phase and bypass the command recall phase by using experimental tasks where the stimuli and the responses are congruent and direct. For example, participants select an item in a marking menu in response to a given direction (e.g., N, W, E, S) in [11]. The same is true in [5, 8] where participants selected a color swatch (in a toolglass, flow menu or a palette) in response to a colored dot. In a study that did involve indirect mapping between the stimulus and the response, Odell and colleagues [18] compared toolglasses, marking menus and keyboard shortcuts to invoke a set of three commands (oval, rectangle, line). In particular, they compared two sets of keyboard shortcuts. One used the first letter of each command ('O', 'R' and 'L') while the second used three abstract numeric keys ('1', '2' and '3'). The latter assignment was the most efficient on average in their study.

Grossman et al. [4] recently conducted what is possibly the most comprehensive study to date on learning arbitrary associations between commands and keyboard shortcuts. In their task, the stimulus was a graphical icon of a familiar object and the action was a keyboard shortcut composed of one modifier key and an alphabetic key which was not a letter contained in the object name depicted by the icon. They explored a number of display methods to accelerate user learning of keyboard shortcuts but found them ineffective except for two rather forceful ones: one augments a menu command selection with the speech audio of the corresponding hot keys; the other simply disables the menu selection ability (rendering the menu a crib sheet of shortcuts) forcing the user to rehearse the keyboard shortcuts.

Despite these and other related works, using strokes as shortcuts to commands still requires investigations. First, researchers have never measured users' ability to learn associations between a stroke and a command therefore the understanding of strokes as commands is rather limited. Second, practitioners do not have the right tools to easily implement stroke-based commands and integrate them into mainstream products. The following sections address these understanding and practical aspects respectively.

STROKE SHORTCUTS VS KEYBOARD SHORTCUTS

In this section we evaluate the performance of stroke shortcuts relative to that of keyboard shortcuts. This comparison to keyboard shortcuts is not meant to be a competition, but rather to use keyboard shortcuts as a baseline control condition. Since the use of shortcuts largely depend on their ease of learning, we focus our study on learning aspects involved in both types of shortcuts. We also limit our study to the general case of arbitrary mappings between the commands and the shortcuts, namely mappings without direct mnemonic association in either condition. This decision was based on several considerations. First, a learning experiment takes time to do well even when it is focused. Second, the special cases of mnemonic mapping, which should be maximized in actual design, is rather limited in number. For example the usual way of making a keyboard shortcut mnemonic is to use the first letter of the command name. However this rule makes interface developers quickly run into conflicts: in fact the small set of five common commands {Cut, Copy, Paste, Save and Print} already exhibits two conflicts. Also, for non-English speakers, the same command may have different names in different languages yet it has the same keyboard shortcut (which is probably a reasonable design choice for consistency). Third, stroke shortcuts can always be made as mnemonics as keyboard shortcuts by choosing letter-shaped strokes. Learning required in that case is probably limited.

Participants

Fourteen adults, two females and twelve males, 26 to 44 years old (mean = 31.8, SD = 4.7), participated in our experiment. They were rewarded with a lunch coupon.

Apparatus

The apparatus was a 1.5GHz Pentium M ThinkPad Tablet PC with a 13-inch tablet digitizer screen at 1024×768 resolution. The experiment window was set in full-screen mode. Participants used the stylus to stroke gestures and could hold the tablet at any time if it felt more comfortable. The set of strokes was designed by the experimenter and the stroke recognizer was based on Rubine's algorithm [20] trained with a set of 15 examples per stroke input by the experimenter.

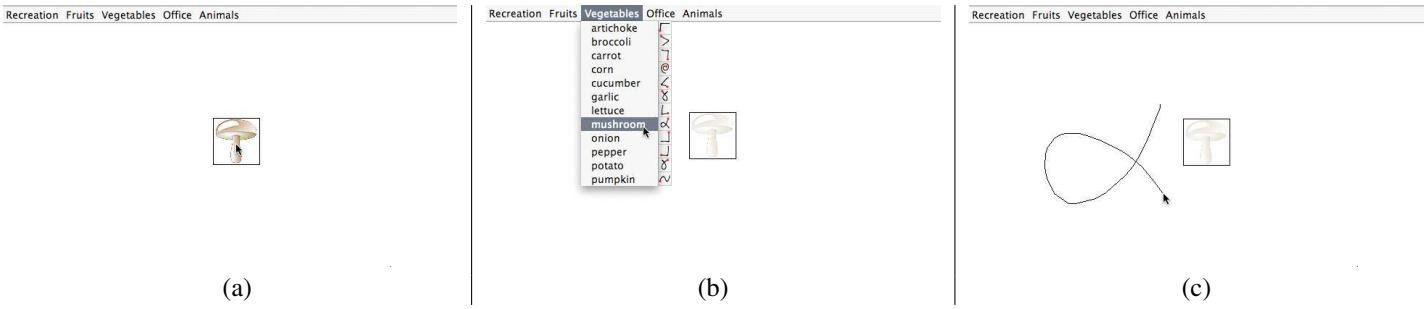

Figure 1. The task used in the experiment: (a) a command stimulus appears as an icon, the participant clicks on it (this makes the icon become semi-transparent) (b) the participant invokes the command through a menu, (c) or through a shortcut (a stroke shortcut in this case)

ICON	Keys	Stroke	ICON	Keys	Stroke
	Shift+W			Ctrl+W	
	Shift+D			Ctrl+D	
...	...	...	...	...	...

Figure 2. An excerpt of the mappings used in the experiment.

m_1	(Karate,12) ; (Pumpkin,12) ; (Hockey,6) ; (Mushroom,6) ; ... ; (Keyboard,2) ; (Garlic,2) ; (Dinosaur,1) ; (Pineapple,1)
m_2	(Karate,6) ; (Pumpkin,6) ; (Hockey,4) ; (Mushroom,4) ; ... ; (Keyboard,1) ; (Garlic,1) ; (Dinosaur,12) ; (Pineapple,12)
...	

Figure 3. Examples of frequency assignments used in the experiment.

Task

We modeled our experimental task after Grossman et al. [4] which was the most recent and most complete study to date on learning keyboard shortcuts. The task required the participants to activate a set of commands that were accessible through both menus and shortcuts. Once a command stimulus (i.e. a graphical icon, as in [4]) was displayed in the center of the screen, the participant was asked to first click on the icon (Figure 1-(a)) and then execute a corresponding command as quickly as possible through either menu selection (Figure 1-(b)) or a shortcut activation (by drawing a stroke or pressing hot keys, depending on the experimental condition) (Figure 1-(c)). The click on the icon at the beginning of each trial prevented the participant from keeping the mouse cursor in the menu area to only interact with menu items. Both types of shortcuts were displayed *on-line* beside the corresponding menu items. The participant was explicitly told to learn as many shortcuts as possible. In case he did not know or remember a shortcut, he can use the menu to directly select the command or look at the shortcut.

The keyboard shortcuts were assigned in accordance to the rule used in [4]: they were composed of a sequence of a modifier key followed by an alphabetic key that was not the first or last letter of the name of the object. To reflect a necessary difficulty in practical keyboard assignments, the same alphabetic key preceded by two different modifier keys (Ctrl or Shift) constituted two different commands. To reduce a potential bias, we reproduced this potential pair confusion in stroke shortcuts as well: the same shaped stroke with two different orientations activated two different commands. Table 2 shows a sample of the icons and the two types of shortcut we tested. To minimize the influence of the participants' personal experience, commands tested were not those in common software applications but rather objects and activities of everyday life organized into five menus (categories): Animals, Fruits, Office, Recreation and Vegetables. Each menu contained 12 menu items resulting in a total of 60 items. In order to have enough trial repetitions, the participants had to activate a subset of 14 commands during the experiment. Note that the rest of the 60 items were also assigned shortcuts and served as distracters both to the participants and to the stroke recognizer.

To reflect the fact that some commands are invoked more frequently than others in real applications, we assigned different frequencies to different commands for each participant, as in [4]. The fourteen frequencies, defined as the number of occurrence per block of trials, were (12, 12, 6, 6, 4, 4, 3, 3, 2, 2, 2, 2, 1, 1). We used 7 frequency assignments (m_1, ..., m_7), balanced across the 14 commands (Figure 3), and assigned each mapping to a group of two participants. The 7 different mappings we used ensured that we collected the same total number of measures per command in the overall experiment.

Design

Participants had to complete 12 blocks of 60 trials organized into two sessions on two consecutive days. Presentation order for commands within a block was randomized while respecting the assigned frequencies. Participants had to perform 10 blocks on the first day and two blocks on the second day. In the first two blocks on the first day (*warm-up*), the only way of invoking a command was through menu selection so participants could become familiar with the menu layout and the experimental task ($Shortcut$ = $None$). In the 8 other blocks (*test*) on the first day commands could be invoked through either menu selection or shortcuts. These blocks were divided into two sets: in 4 blocks the shortcuts were keyboard-based ($Shortcut$ = $Keyboard$) and in the other 4 they were stroke-based ($Shortcut$ = $Stroke$). Within a group of two participants assigned to the same frequency mapping m_i, one experienced the test blocks in the order $Keyboard$ - $Stroke$ while the other the order $Stroke$ - $Keyboard$. For the 11th and 12th blocks on the second

day (*re-test*) both types of shortcuts were available and the participants were told to use what was most convenient for each trial ($Shortcut$ = $Both$).

Before starting the first session, the experimenter distributed instructions explaining the task and asking the participants to learn as many shortcuts as they could in order to complete the study as quickly as possible. Participants were not told what would be in the second session so that they would not consciously rehearse shortcuts during the break between the two days. On the second day, they were told to complete the last two blocks as quickly as possible by using the method of their choice for each trial (*re-test* blocks). Throughout the experiment the participants could rest not only between blocks but after every 20 trials within a block. At the end of the experiment, they were given a questionnaire about their background (if and how much they used keyboard shortcuts and if they had already used a gesture-based interface) and their preference between the two types of shortcuts based on their experience in the study. The final part of the questionnaire was a table organized into three columns "Icon/command", "Keyboard shortcut" and "Stroke shortcut", similar to Figure 2, but with only cells of the first column filled. The participants had to write down the two types of shortcuts as they recalled them for every icon they saw during the experiment. They also had to indicate a confidence level between 0 (don't remember at all) and 1 (totally confident) to each shortcut.

Hypothesis

As mentioned in the Introduction, we hypothesize that an arbitrary association between a command and a shortcut is more learnable when this shortcut is a stroke rather than a combination of keys. This hypothesis is based on two arguments, one in favor of strokes and one against key combinations:

- It has been previously postulated in the literature that strokes (also known as gestures or marks) have various possible advantages including being iconic [17]. The fact that contemporary software applications widely use icons indicates that many users are able to build arbitrary mappings between commands and icons. For example, a compass icon is used for launching the Safari browser, a curved arrow is for reversing the last action (undo) and a floppy disk, now an obsolete concept, is used as an icon for saving the current file on the hard drive. More theoretically, human memory research suggests that deeper or more levels of encoding and processing help memory [3]. The spatial and iconic information in a stroke may better enable users to imagine (encode) an association between the stroke and its corresponding commands. For example, when an upward straight stroke was arbitrarily assigned to the object "bat", the user may make up the association of a bat flying upwards.

- Letters are special symbols which are strongly linked to words in which they appear so it can be very difficult to link a letter to a command name that does not start with this letter (such as Ctrl+V for paste).

Results

We used three measures in our analyses:

- $Time$, the total time interval (in ms) from the command icon being presented to the completion of the correct command. Note that this was the total duration including both recall and execution time.
- $Errors$, the number of times the participant entered a wrong shortcut before typing or stroking the correct one.
- $Recall$, a binary measure which is equal to 1 when the participant was able to activate the right command with a shortcut without opening the menu and without any error, 0 otherwise.

The main results lie in the measures collected for the *test* blocks in which $Shortcut$=$Stroke$ and $Shortcut$=$Keyboard$ were balanced and compared. Variance analysis on the $Time$, $Error$ and $Recall$ data showed that the interaction effect of $Presentation\ Order \times Shortcut$ was not significant, confirming that the counterbalancing strategy for minimizing presentation order effect was successful. We also verified that the participants followed the instructions and indeed used the shortcuts instead of relying solely on menu selection. Across the 8 blocks they used shortcuts in 96% of the trials for $Shortcut = Stroke$ and in 88.5% of the trials for $Shortcut = Keyboard$, indicating that the participants switched from menu selection to stroke shortcuts more often or earlier than to keyboard shortcuts. This measure already suggests that stroke shortcuts were easier to learn.

Our hypothesis was also supported by the three main measures from the 8 *test* blocks. First, on average the trials in the $Stroke$ condition were completed faster than the trials in the $Keyboard$ condition ($F_{1,13} = 36$, $p < .0001$). Second, the participants had significantly better recall scores with stroke shortcuts than with keyboard shortcuts ($F_{1,13} = 32$, $p < .0001$). Third, the participants made significantly fewer errors with stroke shortcuts than with keyboard shortcuts ($F_{1,13} = 23$, $p < .0003$)[3]. Figure 4 summarizes these results.

To compare the learning speed for each type of shortcut, we plotted the mean $Time$ and $Recall$ performances as a function of the number of times an item was tested from the beginning of the experiment (Figure 5). The results also supported our hypothesis: $Time$ decreased faster with stroke shortcuts than with keyboard shortcuts; $Recall$ accuracy increased faster with stroke shortcuts than with keyboard shortcuts. Note that the performance difference between the two types of shortcuts is primarily cognitive (learning and recalling the shortcuts). With enough practice, when the user performance is more likely to be limited by motor execution (around the 25th exposure in this experiment), the difference in both time and recall between the two types of shortcuts became indistinguishable.

Data collected in the *re-test* blocks on the second day allowed us to evaluate users' memory retention of the short-

[3]Note that this result is actually even more favorable to stroke shortcuts since some participants reported that some of their errors were due to a lack of accuracy in the stroke recognizer.

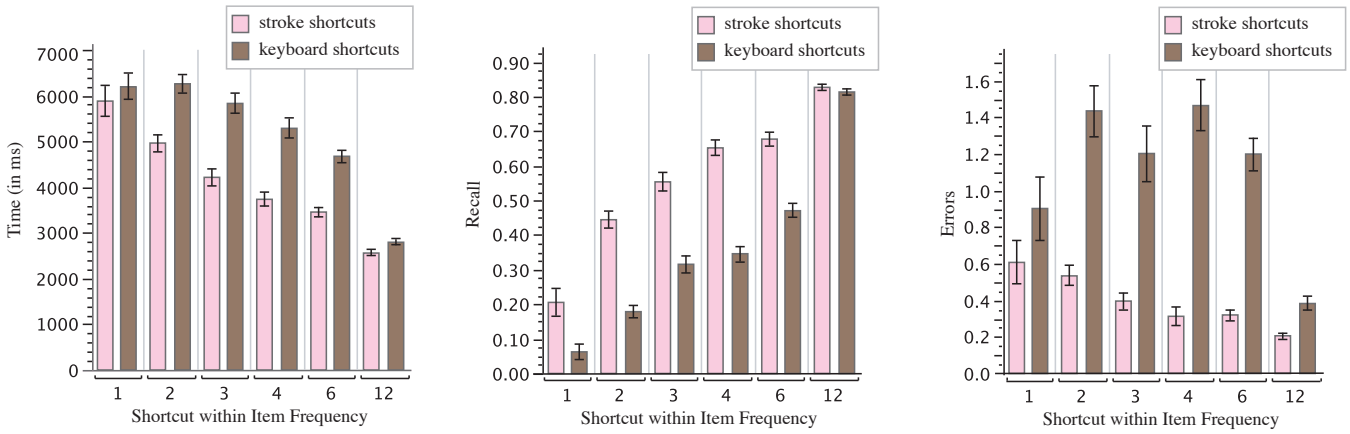

Figure 4. $Time$, $Recall$ **and** $Error$ **by** $Shortcut \times Frequency$

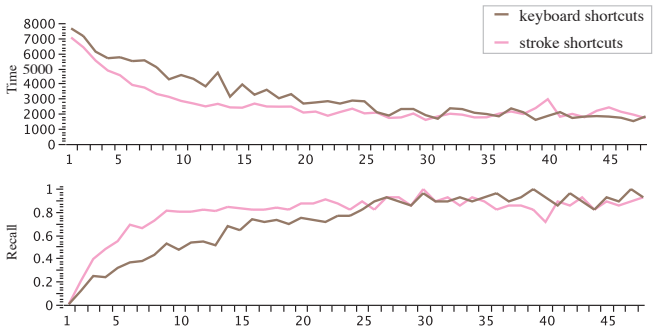

Figure 5. Time and Recall performance according to the number of times a command has been seen

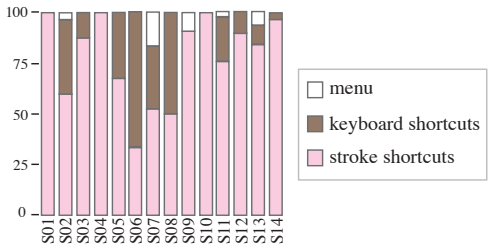

Figure 6. Percentage of use of each technique in *retest* on the second day (by participant)

cuts learned and to see which type of shortcuts they preferred. Figure 6 shows each individual participants percentage of use for each technique ($Keyboard$, $Stroke$ and $Menu$). Although varied by individual, on average significantly more stroke shortcuts than keyboard shortcuts were used ($F_{1,13} = 43$, $p < .0001$). The overall mean percentages of use for the three techniques were: 77.7 % $Stroke$, 20.3 % $Keyboard$, 2 % $Menu$.

Finally, answers to the post hoc questionnaire showed that all of the participants had intensive previous experience with keyboard shortcuts in their everyday activity (about 15-20 different shortcuts) and that none of them had ever used strokes. Despite this experience bias in favor of keyboard shortcuts, the answers to the final question where they had to fill the table revealed that they had learned stroke shortcuts better than keyboard shortcuts in this study. On average 11.6 stroke shortcuts and 4 keyboard shortcuts were correctly answered. The participants' confidence level was also higher with stroke shortcuts (11.7/14 on average; 14 means complete confidence on all commands tested) than with keyboard shortcuts (4.2/14 on average).

The participants' open remarks confirmed some of the analyses that led to our hypothesis. Strokes gave them richer clues to make up an association (more levels of processing) between a command and its arbitrarily assigned stroke: "I thought of this stroke as fish because the shape's stroke makes me think about a basin" or "I associated this stroke with a jump and I see karate as a sport where people jump". Interestingly, no two people mentioned the same trick to associate a stroke with a command.

In summary, although the purpose of stroke shortcuts is not to replace or compete against either menu selection or keyboard shortcuts, the experiment clearly shows that stroke shortcuts can be as efficient as or more advantageous than keyboard shortcuts. After enough practice, the total trial completion times including both recall and execution were indistinguishable between the two types of shortcuts. However with the same fixed amount of practice, the participants successfully recalled more shortcuts and made fewer errors in the $Stroke$ condition than in the $Keyboard$ condition. On the second day the participants chose to use stroke shortcuts significantly more often than keyboard shortcuts, and correctly recalled about 3 times as many stroke shortcuts as keyboard shortcuts.

STROKE SHORTCUTS AND UI DEVELOPMENT

The study we conducted suggests that stroke gestures can be used as command shortcuts that are as effective as, or even more effective than, keyboard shortcuts. However, implementing stroke shortcuts in real applications is more challenging than implementing keyboard shortcuts because commonly used graphical toolkits do not support stroke input. In order to encourage the adoption of stroke shortcuts in a wide

range of applications, we articulate a set of guidelines for stroke shortcuts development based on an analysis of previous literature and our own experience. We then introduce *Stroke Shortcuts Toolkit* (SST), an extension to Java Swing that we have developed to support stroke shortcuts.

Guidelines to make stroke shortcuts easy to implement

(1) template-based recognition algorithm

Several tools for implementing stroke recognition already exist. For example, Satin [7] is a Java toolkit that uses a special component, a *Sheet*, on which strokes can be drawn and sent to a recognizer. Satin's recognizer is built on Rubine's training-based recognition algorithm [20]. To accurately train the different features representing a stroke in the algorithm (e.g. size, orientation, speed), enough examples (about 15) must be given for each stroke and these examples must reflect the variance along these feature dimensions. Either the interface designer or the end user has to enter these examples. On the one hand, it is difficult for the designer to foresee the stroke variations that can occur among all users [4]. On the other hand, if the training task is left to the end user, another set of difficulties arises: when and how should the interface ask the user to enter these examples? Users tend to be reluctant to invest time and effort upfront to train or adjust software before using it. A third approach is to train the recognizer with examples from a large standardized stroke corpus. However, without a firmly established user community and stroke standard, such a corpus is difficult to collect.

While training-based recognition handles different styles and habits in natural handwriting fairly well, it may not be necessary with novel stroke gestures that can be explicitly defined with unique templates. In fact the work of ShapeWriter has shown that template-based recognition can handle thousands of stroke gestures if multiple channels of information are appropriately integrated ([9]). More recently, Wobbrock et al [22] formally evaluated template matching methods (with and without elasticity [21]) in comparison to Rubine's algorithm for recognizing strokes similar to those used in this paper. In their favored method, the $1 recognizer, each template is represented by a set of equally spaced points, scaled to a given bounding box and rotated to an *indicative* angle (i.e. the angle formed between the first point and the centroid of the template). When a stroke is entered, it is resampled, scaled and rotated to its indicative angle so its distance to each template can be computed by summing the distances between pairs of corresponding points. Their experiment results show that such a simple template matching approach in fact has better performance than Rubine's algorithm. By eliminating training issues while still being accurate, ***a template-based algorithm is the best choice to implement stroke shortcuts***.

(2) Simplify the task of designing a set of strokes

In [15], Long et al. studied the task of designing a set of strokes for Rubine's recognition algorithm. Participants were asked to obtain the best recognition accuracy they could. Results showed that it is a very difficult task and no one participant was able to go beyond the 95.4% recognition rate. A typical problem they observed is that participants tend to add strokes that are too similar to those already defined. This shows that ***designers' imagination must be stimulated by providing them with a design space*** for defining a set of strokes for the commands of the application they want to enhance.

Most of the other problems Long et al. identify in the task of defining a set of strokes are specifically dependent on Rubine's algorithm [15]. They concluded that participants (including computer science students) were not able to get a high recognition rate because they do not understand the principles of the algorithm. It is very difficult to get a mental model of how Rubine's algorithm works: it represents a gesture as a set of features and not as a series of points and uses a covariance matrix that evolves each time an example or a new stroke class is added with the potential unpredictable consequence of degrading the recognition accuracy between the old stroke classes. The study in [15] suggests that ***the underlying mechanisms in the recognition engine must be transparent to the interface designers***.The simple shape matching algorithm used in the $1 recognizer is probably better from that perspective. However, the rotation independence property can be hard to anticipate since the notion of indicative angle is not straightforward. This rotation step can also be a limitation: for example, the rotationa-independent recognizer cannot distinguish among lines in different directions which are convenient for invoking reciprocal commands (e.g. "previous" and "next" in a web browser). Furthermore, Long et al. [16] showed that stroke initial angle and angle formed by first and last stroke points are important to perceive two strokes as different while the rotation independence limits variations that can be expressed along these two dimensions. Thus the most comprehensive and permissive recognition algorithm is probably the one used in the $1 recognizer without the rotation independence which in fact was also the algorithm used in the shape channel of ShapeWriter ([9]).

(3) Make stroke shortcuts visible to end users

A well-known and important drawback of using strokes to activate commands is that these strokes are not self-revealing [13, 6, 1]. In other words, as opposed to buttons and menus, the user cannot guess which stroke-based commands are available and which stroke triggers which command. Often novel features of an interface are unused not because they are difficult to use, but because the users are not aware of them. Therefore ***interfaces should offer visual clues to available strokes*** to make end users able to discover and learn their effect.

(4) Integrate stroke shortcuts in graphical toolkits

Because interface developers are not willing to change their development environment or rewrite their existing applications, ***interface toolkits should support the implementation of stroke shortcuts***. Of course, the implementation capabilities should be high-level enough to minimize developer pro-

[4]This was a challenge that we faced during the experiment presented in the previous section in which the Rubine's recognizer was trained by the experimenter.

Figure 7. A simple Java Swing interface for a music player.

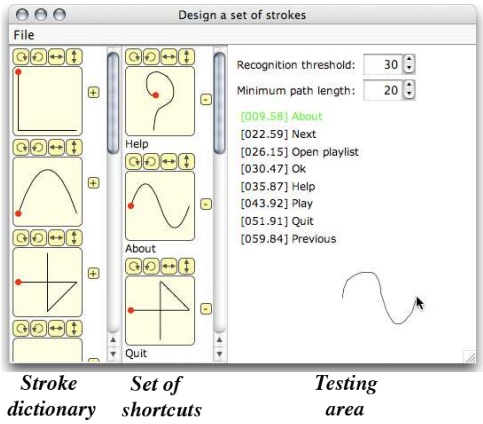

Figure 8. The *Design Shortcuts* application

gramming effort. As a baseline, developers typically need to only add one line of code per command to implement a keyboard shortcut. Implementing a stroke shortcut should not involve much more programming effort.

SST: stroke shortcuts in Java Swing

In this section, we present SST[5], a Java Swing extension to simplify the addition of stroke shortcuts to any Swing application. To illustrate, lets consider that we want to add stroke shortcuts to the music player window shown on Figure 7 and built with the instruction:

```
SimplePlayer player = new SimplePlayer();
```

To define the mappings between the commands and their shortcut strokes, the developer can invoke the *Design Shortcuts* graphical design environment shown on Figure 8. Launching this environment on the application windows for which he wants to map commands with stroke shortcuts requires the single line:

*// Launch Design Shortcuts environment on the main player window and its **About** dialog shown on Figure 10*

```
1  new DesignShortcuts(player, player.about);
```

The *Design Shortcuts* interface (Figure 8) is divided into three areas: the *stroke dictionary* (left panel), the *set of shortcuts* (middle panel) and the *testing area* (right panel). To define a new shortcut, the developer clicks on the '+' button displayed on the right of a stroke in the *dictionary*. This pops up the list of commands found in the attached windows. He can either (i) pick one of these commands in the list or (ii) type a *new command* name. Callbacks for these new commands are handled through the use of Java listeners as explained later in this section. At any time, the developer can test the recognition accuracy by drawing in the testing area. As soon as a stroke ends, the application displays the list of the distances between the input stroke and each template, the recognized stroke being the template with the shortest distance.

The *stroke dictionary* contains an initial set of 9 predefined strokes for the developer to choose from. With these predefined strokes, the developer can already define a large set of shortcuts by combining several of these strokes and/or applying geometrical transformations to them. One can use the *transformation buttons* displayed on top of each stroke to rotate or mirror (horizontally or vertically) a stroke before adding it to the set of shortcuts (using the '+' button displayed to the right of the stroke). In the example shown in Figure 8, the developer has used the same shape for **Ok** and **Play**: the orientation of the **Ok** shortcut suggests a check mark while the **Play** shortcut suggests the symbol usually dedicated to the play command in many music players. Selecting several strokes before pressing one of the '+' buttons will build a new stroke that is the concatenation of the selected strokes. For example, the stroke for the **About** command has been defined by concatenating the predefined "arch" stroke with a mirrored copy of it[6]. Once added to the set of shortcuts, the transformation buttons remain displayed for further modifications. If needed, the '-' button displayed to the right of each stroke allows the shortcut to be removed.

Compared with starting with a "blank page", providing a set of primitive strokes and a set of operations on these strokes opens a structured design space that can be systematically explored. However, there is no reason to constrain the developer to this set of primitives. Developers can expand the stroke dictionary with additional custom strokes: a "Free stroke" button at the bottom of the list of primitives opens a separate frame for drawing a stroke to be added to the dictionary. This is how the developer has defined a question mark stroke for the **Help** command in our example (Fig. 8).

Once designed, the set of shortcuts can be saved as a file ("player.strokes" here) and enabled on a given Swing interface through a few lines of code. In our music player example, only 10 lines of code (Figure 9) are needed to accomplish this. SST connects the shortcuts to a Swing GUI using a central object, the *stroke shortcuts manager* (`m`, line 1). This object is in charge of integrating stroke shortcuts to the Java Swing toolkit and is used to register:

- the mappings (stroke to command name) (line 2),
- the windows that contain commands that can be invoked through these shortcuts (lines 3 and 4) and

[5]SST is an open source project containing about 3000 lines of code and is available online: http://code.google.com/p/strokesshortcuts/.

[6]A pop up menu allows to duplicate any stroke in the dictionary.

```
1  StrokeShortcuts m = new StrokeShortcuts(
             player, player.dialogAbout);
2  m.addShortcuts("player.strokes",
             MENU_PREVIEW, TOOLTIP_PREVIEW);

3  m.setCriterion(player.playlist, new Criterion(){
4     public boolean startStroke(MouseEvent event) {
5        return event.getButton() == MouseEvent.BUTTON3;
6     }
7  });
8  m.disableStrokes(player.sliderSong);
9  m.disableStrokes(player.sliderVolume);

10 m.enableStrokesSheet();
```

Figure 9. Complete code to add keyboard shortcuts to the music player interface.

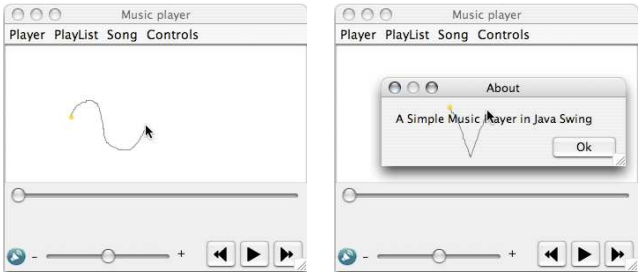

Figure 10. Strokes in different windows.

- the "strokable" components, i.e. the Swing widgets on which strokes can be drawn (lines 5 to 13, detailed below).

In SST, a stroke is defined as a series of points sent by an analog input device (a mouse or a digital pen) that starts with a press event and ends with a release event. Each stroke occurring on a "strokable" component is entered into the recognizer to get the name of the command that is then invoked through the Java accessibility interface. In our example, line 1 registers both the main frame and the **About** dialog as "strokable" components so the user can draw on any of the two windows as illustrated in Figure 10. By default, all children components of a "strokable" component are also "strokable". Since press, drag and release are events that may already be used by standard widgets, SST allows the developer (i) to associate a criterion on the mouse press event that specifies when the stroke recognition must be enabled or (ii) to disable stroke recognition on specific components. For example, no criterion is required when the user wants to stroke on the interface background while one is required for a list box on which a drag is already an action dedicated to selecting items in the list. In this latter case, the developer can decide to accept only strokes drawn when the right mouse button is pressed (lines 3 to 7). Finally, he disables stroke recognition on the sliders for adjusting the playing point in a song and the volume (lines 8 and 9).

By default, in SST, a stroke leaves a visible ink trail (by means of the transparent overlay available in the window containing the component). At the end of a stroke, its ink trail is either smoothly morphed into the template it matches (in case of recognition, as ShapeWriter does [9]) or flashes red (if it is not recognized). Note that the ink is morphed into a template scaled to the same size as the stroke to minimize visual change. Also, the morphing animation stops as soon as the user starts a new stroke so expert users can enter strokes in rapid succession. The morphing animation (or beautification) not only provides a feedback of recognition result but also helps novice users learn the correct stroke shape and discourages expert users from departing too much from the ideal shape. If the transparent overlay is already used for another purpose, stroke ink can be disabled and a different feedback mechanism can be implemented. One or several stroke listeners can be attached to the shortcut manager which will be notified each time the user begins a stroke, adds a point to a stroke or ends a stroke. When ending a stroke, the event can be of one of the three types: *recognized shortcut*, *recognized stroke* or *non recognized stroke*. In all cases, the current input stroke can be retrieved from the received event so that it can easily be used for other interactions. For example, a non recognized stroke could be used for drawing in a graphical editor.

To address the *visibility* problem (i.e. users do not have a way to discover the available strokes and their meaning), SST offers three types of visual clues to make the user discover and learn the mappings: *Tooltip*, *Menu preview* and a *Strokes Sheet*. The first two types of visual clues are turned on or off by the parameters of `addShortcuts` (line 2 on Figure 9). If *Tooltip* is turned on, any graphical component that provides a shortcut will display it in a tooltip that pops up when the mouse cursor dwells over this component. If this component already has a tooltip associated with it, the existing tooltip is augmented with the stroke illustration while preserving its original text. Similarly, if the *Menu preview* is turned on, any menu item that is invokable by a stroke displays a preview of this stroke beside its label as is usually done with keyboard shortcuts. Finally, a *Strokes Sheet* can be enabled in the stroke shortcuts manager (line 10, Figure 9). A strokes sheet is an independent window that displays the list of shortcuts and the name of their associated commands found in the current opened windows. The behavior of the strokes sheet has been inspired by the Tivoli system in [13]: this sheet pops up each time the user pauses during a stroke (at the beginning or at any moment while stroking) and remains visible until the user enters a shortcut that is successfully recognized or closes the sheet.

In this section, we showed how SST allows a developer to add stroke shortcuts to a Java Swing interface in only 10 lines of code without having to modify the basic code for the music player. The 3000 lines of code in SST offload the developer not only from having to implement or train any recognizer but also from developing visual displays (morphing feedback, tool tips or crib sheet). We have also shown how the *Design Shortcuts* environment helps developers to map a set of strokes by offering a structured design space.

Recognition accuracy in SST

Since the recognizer implemented in SST skips the rotation step of the $1 recognizer evaluated in [22], one may wonder to what extent it affects recognition accuracy. Although we have not observed a noticeable degradation during our informal tests, we decided to conduct a controlled experiment

that measures the recognition accuracy of our simple matching algorithm on the set of strokes shown in Figure 11. We chose to use a set of 16 strokes since the answers to the post hoc questionnaire of Experiment 1 revealed that our participants use roughly 15 different keyboard shortcuts in their everyday use of computers.

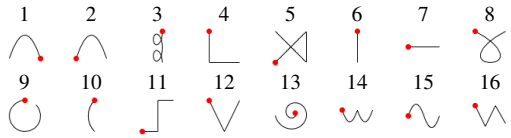

Figure 11. The 16 strokes used in our experiment.

This experiment involved 6 of the participants that had already served in our first experiment and the same apparatus. The task was very simple: one stroke was displayed on the screen and the participant was told to reproduce it as fast as possible and as accurately as possible. As soon as the participant started to draw, the sample stroke disappeared to avoid turning the reproduction task into a copy task. At any time (except during stroking), the participant could have the stroke displayed again by pressing the space bar and start the task again. We implemented this possibility to avoid the situation where the recognition failed and the participant had forgotten what stroke to produce. A trial ended only if the right stroke was recognized.

The experiment had two $Input\ device$ conditions: Pen and $Mouse$. We included a regular mouse for two reasons. First, since the mouse is less dexterous than a pen in articulating shapes (drawing one's signature with a mouse vs. a pen shows the difference), the mouse condition would add more stress to the recognizer. Second, it is also practically useful to know if stroke shortcuts can be used with a mouse. In the experiment the participants had to perform 11 blocks in each condition that were grouped to avoid successive changes of input device. Each block consisted of 16 trials, one per $Stroke$, presented in a random order. The presentation order of input devices was counterbalanced between participants so 3 participants started in the $Mouse$ condition while the 3 others started in the Pen condition. In each condition, the first block was a practice block.

We measured the $Stroking\ time$, i.e. the time between the press and the release event when drawing the right stroke, and the number of $Recognition\ errors$. Analysis of variance revealed a significant effect of both $Input\ device$ ($F_{1,5} = 34$, $p < 0.002$) and $Stroke$ ($F_{15,75} = 25$, $p < 0.0001$) on $Stroking\ time$. Users were faster with a pen (394 ms on average) than with a mouse (704 ms on average). Also, the $Stroking\ Time$ increased with the complexity of the stroke, supporting the results reported in [2]. More surprisingly, we observed a significant interaction effect of $Stroke \times Input\ device$ on $Stroking\ time$ ($F_{15,75} = 5$, $p < 0.0001$): differences between input devices seem to increase with the complexity of the stroke (particular more curves). All these results are illustrated on Figure 12.

Analysis of variance also revealed a significant effect of $Input\ device$ ($F_{1,5} = 9$, $p < 0.03$) on $Recognition\ errors$. In the $Mouse$ condition, the participants made 7.4% errors on the first attempt at each stroke sample presented. Among these, they succeeded 73% of the time with the second attempt, 10% with the third attempt, and 17% with the subsequent attempts. In the Pen condition, only 3% of the trials failed with the first attempt, of which 76% were corrected with the second attempt, 7% with the third attempt, and 17% with the subsequent attempts. There was also a significant main effect of $Stroke$ ($F_{15,75} = 3$, $p < 0.001$) on $Recognition\ errors$: the error rates drastically changed when removing the 3 more complex strokes from our data: less than 0.001% of the trials in the $Mouse$ condition and 0% of the trials in the Pen condition on the first attempt. Finally, for $Recognition\ errors$, the interaction effect $Stroke \times Input\ device$ was not significant. Overall this study shows that the recognizer used in the *StrokeShortcuts* library is accurate. Although users' stroke articulation speed was considerably slower with a mouse than with a pen, the shape-matching based recognition algorithm could accurately recognize mouse strokes as well.

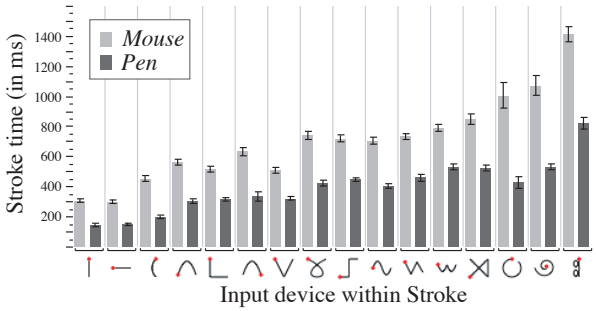

Figure 12. Stroke time by $Stroke \times Input\ Device$

CONCLUSION

Menu selection has been, and will likely continue to be, the basic and dominant way of activating commands in human-computer interaction. Ubiquitous in modern software applications, keyboard shortcuts provide a faster alternative to frequently used commands. The investigation presented in this paper encourages the use of stroke gesture as shortcuts for touch screen-based devices without a physical keyboard. The conceptual and empirical study in the first part of the article shows that stroke shortcuts can be as effective as keyboard shortcuts in eventual performance, but have cognitive advantages in learning and recall. With the same amount of practice, about three times as many stroke shortcuts were learned as keyboard shortcuts. Following a set of development guidelines articulated in the second half of the paper, we have shown a simple way to implement stroke shortcuts in Java Swing by providing developers with SST. Requiring no training by the developer or the end user, the built-in shape matching-based recognizer in SST can yield high accuracy for simple strokes, even if the strokes are articulated with a mouse. SST offers a structured yet open design environment and simplifies the implementation of strokes' visibility in applications.

Integrating stroke shortcuts in the Java / Swing platform-independent framework is a first step, we now plan to develop extensions to other frameworks like Objective C / Cocoa or C# to cover most of the applications developed for touch screen devices ranging from Apple's iPhone to HP's TouchSmart desktop PCs.

ACKNOWLEDGEMENTS

We thank Michel Beaudouin-Lafon and Alison Sue for helping to improve this article, our participants for having served in our studies and Pierre Dragicevic for sharing with us the set of icons used in [4].

REFERENCES

1. O. Bau and W. E. Mackay. Octopocus: a dynamic guide for learning gesture-based command sets. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, 37–46, New York, NY, USA, 2008. ACM.
2. X. Cao and S. Zhai. Modeling human performance of pen stroke gestures. In *CHI '07: Proc. ACM Conference on Human factors in computing systems*, 1495–1504, 2007.
3. F. Craik and R. Lockhart. Levels of processing: A framework for memory research. *Journal of Verbal Learning and Verbal Behavior*, 11(6):671–684, 1972.
4. T. Grossman, P. Dragicevic, and R. Balakrishnan. Strategies for accelerating on-line learning of hotkeys. In *CHI '07: Proc. ACM Conference on Human factors in computing systems*, 1591–1600, 2007.
5. F. Guimbretière, A. Martin, and T. Winograd. Benefits of merging command selection and direct manipulation. *ACM Trans. Comput.-Hum. Interact.*, 12(3):460–476, 2005.
6. K. Hinckley, S. Zhao, R. Sarin, P. Baudisch, E. Cutrell, M. Shilman, and D. Tan. Inkseine: In situ search for active note taking. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, 251–260, New York, NY, USA, 2007. ACM.
7. J. Hong and J. Landay. Satin: a toolkit for informal ink-based applications. In *UIST '00: Proc. ACM Symposium on User interface software and technology*, 63–72, 2000.
8. P. Kabbash, W. Buxton, and A. Sellen. Two-handed input in a compound task. In *CHI '94: Proc. ACM Conference on Human factors in computing systems*, 417–423, 1994.
9. P.-O. Kristensson and S. Zhai. Shark2: a large vocabulary shorthand writing system for pen-based computers. In *UIST '04: Proc. ACM symposium on User interface software and technology*, 43–52, 2004.
10. P.-O. Kristensson and S. Zhai. Command strokes with and without preview: using pen gestures on keyboard for command selection. In *CHI '07: Proc. ACM Conference on Human factors in computing systems*, 1137–1146, 2007.
11. G. Kurtenbach and W. Buxton. User learning and performance with marking menus. In *CHI '94: Proc. ACM Conference on Human factors in computing systems*, 258–264, 1994.
12. G. Kurtenbach, G. Fitzmaurice, R. Owen, and T. Baudel. The hotbox: efficient access to a large number of menu-items. In *CHI '99: Proc. ACM Conference on Human factors in computing systems*, 231–237, 1999.
13. G. Kurtenbach and T. Moran. Contextual animation of gestural commands. *Eurographics Computer Graphics Forum*, 13(5):305–314, 1994.
14. D. Lane, H. Napier, S. Peres, and A. Sandor. Hidden Costs of Graphical User Interfaces: Failure to Make the Transition from Menus and Icon Toolbars to Keyboard Shortcuts. *International Journal of Human-Computer Interaction*, 18(1):133–144, 2005.
15. A. C. J. Long, J. A. Landay, and L. A. Rowe. Implications for a gesture design tool. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, 40–47, New York, NY, USA, 1999. ACM.
16. A. C. J. Long, J. A. Landay, L. A. Rowe, and J. Michiels. Visual similarity of pen gestures. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, 360–367, New York, NY, USA, 2000. ACM.
17. P. Morrel-Samuels. Clarifying the distinction between lexical and gestural commands. *Int. J. Man-Mach. Stud.*, 32(5):581–590, 1990.
18. D. Odell, R. Davis, A. Smith, and P. Wright. Toolglasses, marking menus, and hotkeys: a comparison of one and two-handed command selection techniques. In *GI '04: Proc. of Graphics Interface*, 17–24, 2004.
19. Y. Rogers. Evaluating the meaningfulness of icon sets to represent command operations. In *Proc. Conference of the British Computer Society*, 586–603, 1986.
20. D. Rubine. Specifying gestures by example. *SIGGRAPH Comput. Graph.*, 25(4):329–337, 1991.
21. C. Tappert. Cursive script recognition by elastic matching. *IBM Journal of Research Development*, 26(6):765–771, 1982.
22. J. Wobbrock, A. Wilson, and Y. Li. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *UIST '07: Proc. ACM Symposium on User interface software and technology*, 159–168, 2007.
23. C. Wolf. Can people use gesture commands? *ACM SIGCHI Bulletin*, 18(2):73–74, 1986.
24. C. Wolf, J. Rhyne, and H. Ellozy. The paper-like interface. In *Proc. International conference on human-computer interaction on Designing and using human-computer interfaces and knowledge based systems*, 494–501, 1989.
25. R. Zeleznik and T. Miller. Fluid inking: augmenting the medium of free-form inking with gestures. In *GI '06: Proc. of Graphics Interface*, 155–162, 2006.
26. S. Zhai and P.-O. Kristensson. Shorthand writing on stylus keyboard. In *CHI '03: Proc. ACM Conference on Human factors in computing systems*, 97–104, 2003.
27. S. Zhao and R. Balakrishnan. Simple vs. compound mark hierarchical marking menus. In *UIST '04: Proc. ACM Symposium on User interface software and technology*, 33–42, 2004.