OctoPocus: A Dynamic Guide for Learning Gesture-Based Command Sets

Olivier Bau & Wendy E. Mackay
In|Situ|, INRIA, LRI
Building 490, Univ. Paris-Sud, Orsay Cedex, France
bau@lri.fr, mackay@lri.fr

ABSTRACT
We describe *OctoPocus*, an example of a *dynamic guide* that combines on-screen feedforward and feedback to help users learn, execute and remember gesture sets. OctoPocus can be applied to a wide range of single-stroke gestures and recognition algorithms and helps users progress smoothly from novice to expert performance. We provide an analysis of the design space and describe the results of two experiments that show that OctoPocus is significantly faster and improves learning of arbitrary gestures, compared to conventional Help menus. It can also be adapted to a mark-based gesture set, significantly improving input time compared to a two-level, four-item Hierarchical Marking menu.

ACM Classification: D.2.2 [Software Engineering]: Design Tools & Techniques, User interfaces, H.1.2 [Models & Principles]: User/Machine Systems Human factors, H.5.3

General terms: Design, Human Factors, Experimentation

Keywords: Dynamic Guides, Feedback, Feedforward, Gesture recognition, Mouse input, OctoPocus, Pen input

INTRODUCTION

Gesture-based interfaces allow users to interact with the objects of interest on the screen, providing expert users in particular with a direct and efficient form of interaction. Unlike buttons and pull-down menus, which force the user to move the mouse or pen to the command's location on the screen, gestures can be performed directly from the current cursor position. Gestures may be arbitrary, such as strokes in different directions, or mnemonic, so that the gesture's shape corresponds to the command's meaning, e.g., drawing a "c" to invoke a copy command. The former are simple and quick to execute; the latter are easier to remember.

Some researchers have explored gesture recognition in specific applications, including flick gestures in web browsers [27], interfaces for air traffic control [8] and drawing applications [17]. However most graphical user interfaces continue to use standard buttons and pull-down menus. Technically, gesture recognition demands robust and accurate algorithms that reliably differentiate among noisy inputs and correctly identify specified commands. For users, gesture-based interfaces demand extra learning: they must *recall* which gesture is associated with which command whereas users of button and menu-based interfaces need only *recognize* the correct command. Although gesture-based interaction is ultimately more efficient for experts, novices need extra support to learn and to correct errors.

How can we improve gesture-based interfaces to make them more efficient and accessible to users, independent of their expertise? Some research focuses on improving gesture recognition algorithms, e.g., Rubine's example-based recognition [29] and the more recent $1 Recognizer [31]. Others focus on helping to design effective gesture sets, e.g., Cao et al.'s [11] pen-gesture model. We are interested in the complementary problem: helping users to *learn*, *execute* and *remember* new gesture sets. Our goal is to help users move from novice- to expert-level performance, obtaining the benefits of gesture-based interaction independently of the gesture set or recognition algorithm used.

This paper proposes a design space for classifying existing feedforward and feedback mechanisms in gesture-recognition systems. We describe the concept of *dynamic guides,* which address an empty area in this design space, and our development of *OctoPocus*, which provides novices with continuous feedforward and feedback while drawing a gesture. We describe two experiments that compare OctoPocus to a standard Help menu and a Hierarchical Marking menu, respectively, and conclude with a discussion and directions for future work.

RELATED WORK

This section first reviews technical strategies for improving pen- and mouse-based gesture recognition systems. We then review literature that focuses on the user's perspective i.e. on-screen strategies to help users during gesture input.

Technical perspective

Gesture recognition algorithms include Rubine's popular gesture classifier [29], which requires initial training by drawing sample gestures, and the *$1 Recognizer* [31], which provides a simple and efficient algorithm intended to support rapid prototyping of gesture-based interfaces. Other algorithms use symbol fragmentation [14] or turning angle representation [15] to recognize shapes. Another way to improve gesture recognition is through the gesture sets themselves. Some researchers create general tools for designing gesture-based interfaces, e.g., SATIN [13]. Others develop specific gesture sets that emphasize easy text input, such as, Graffiti [5] and the Unistroke [10] alphabets. Still others focus on quantitative models for predicting perceived similarities among gestures [23,24] or models of human performance to analyze single-stroke pen gestures [7]. While these are all important, our focus here is on the complementary problem of improving gesture-based interfaces from the user's perspective.

User's perspective

Existing systems provide two basic mechanisms for learning gesture sets and reducing errors: *Feedforward* mechanisms provide information about a gesture's shape and its association with a particular command, prior to the execution or completion of the gesture. *Feedback* mechanisms provide low-level information about the recognition process, either during or after the execution of the gesture.

Feedforward mechanisms are relatively common in commercial gesture-based interfaces. Some provide a physical help card or pop-up 'cheat sheet' to illustrate the gestures and associated commands. The disadvantage is that users must divide their attention between the current gesture and the cheat sheet. Also, displaying a complete set of gestures and associated commands takes a large amount of screen space and risks occluding large sections of the screen.

Kurtenbach et al. [20] combined *crib-sheets* and *contextual animation* to help users learn which gestures are currently available. A pop-up cheat sheet displays the relevant subset of the gesture vocabulary available depending on context, with whole gestures next to their corresponding commands. They provide "*animated, annotated demonstrations"* to demonstrate each gesture and help users visualize how gestures should be performed.

Avrahami et al. [4] Paper PDA is a paper-electronic interface with templates that guide simple, single-stroke input. The approach does not scale to on-screen interaction, because the templates include the whole gesture set and would occlude major sections of the screen. Hover Widgets [11] offer an alternative, providing a guide based on a dedicated algorithm that recognizes simple gestures. The user's pen 'hovers' over a graphics tablet to obtain a feed-forward display of all possible gestures. On-screen "tunnels" indicate where to move to invoke commands. Crossing the border of one tunnel resets it, though others may not. The visual complexity increases when full gestures overlap.

Feedback mechanisms depend upon the recognition algorithm and can only occur after the user has begun making a gesture. Feedback may consist of displaying the recognized command or provide incremental information as to the current state of the recognition algorithm. Mankoff et al. [26] focus on post-input *mediation,* in which the recognizer reveals how it interpreted the input. They survey existing error-correction techniques and identify two mediation strategies: repetition and choice. They also describe *OOPS,* a "toolkit that supports resolution of input ambiguity through mediation". They use Igarashi's interactive beautification technique [16] as an example of post-input mediation, since it shows users how the gesture was interpreted and lets them choose among "perfect" alternatives.

Other feedback approaches provide recognition results during input. Most focus on *shape beautification* i.e. modifying the user's hand-drawn input to illustrate a perfect instance of a given gesture class. Fluid Sketches [2] morph gesture input into simple shapes, such as circles. Users must draw a significant portion of the final gesture before obtaining useful feedback and, although Fluid Sketches have been integrated into a graph-drawing application [3], they remain limited to simple gesture sets. Li et al's [21] Incremental Intention Extraction provides feedback that can be seen as a discrete version of Fluid Sketches. If a part of a user's drawing is recognized as, for example, a clockwise elliptic arc, it is "beautified" during input. Agar and Novins' [1] *polygon beautification* algorithm uses drawing speed to detect corners, providing recognition information only about the segments between corners.

Some gesture-set design tools provide gradual information about recognition state, which helps gesture-set designers discover and fix recognition problems. For example, Long et al.'s *GDT class window* [23] displays the state of Rubine's algorithm after a given test input, indicating not only whether a particular gesture was recognized but also a numeric value quantifying how well it was recognized. This dynamic approach makes the recognition process more transparent and helps the user to input gestures correctly.

Combining Feedforward and Feedback Mechanisms. Kurtenbach's [19] Marking menus extend Pie menus [6] to combine feedforward and feedback and provide a smooth transition between novice and expert use. Users flick the pen or mouse in a particular compass direction to indicate a command. Marking menus take advantage of novice users' hesitation when they are unsure of a gesture or command. After a "press and wait" gesture, a circular feedforward display appears around the current mouse cursor, showing each available command and its associated direction. Highlighting the current selected item during input gives feedback on how a user's input is being interpreted. This approach offers an excellent compromise between learning and efficient use: Novices pause to take advantage of the feedforward display. As they become expert, they move more quickly, no longer needing the Marking menu, and thus significantly increase overall performance.

Marking menus are three times faster than ordinary pull-down menus [18] and have been adapted for text entry [28] and multiple command entry [12]. Zhao and his colleagues have further improved them by increasing menu breadth with zone and polygon menus [33] and have improved the efficiency of Hierarchical Marking menus by converting zigzags into single strokes [32]. Their key limitation relates to the gesture set itself. Kurtenbach notes that "the mark set is not particularly expressive" and that the Marking menu is not adapted to complex gestures. One of our goals is to take advantage of the Marking menu approach for helping users and extend it to encompass any type of single-stroke gesture for a wide range of recognition algorithms.

DESIGN SPACE

This section examines representative examples of feedforward and feedback mechanisms. We identify underlying dimensions that form a design space, independent of the design of gesture sets or of gesture recognition algorithms.

Feedforward Mechanisms

We can classify the feedforward systems in the literature along two dimensions (Fig. 1):

Level of detail: from a minimal hint to a portion of the gesture to the whole gesture, and

Update rate: from only once prior to execution to discrete intervals to continuously during execution.

Marking menus are updated once, if the user hesitates, and display low-detail hints to indicate possible directions. This is sufficient for guiding mark-based gestures, although not suitable for more complex gestures. *Hierarchical Marking menus* update these low-detail hints in discrete steps as the user progresses through the menu hierarchy. The *Paper PDA*, *Contextual animation* and on-screen *cheat sheets* display high detail, i.e. whole gestures from the command vocabulary, but update them only once. *Hover widgets* display whole gesture templates and continuously update them to prevent the user's cursor from crossing the tunnel boundary.

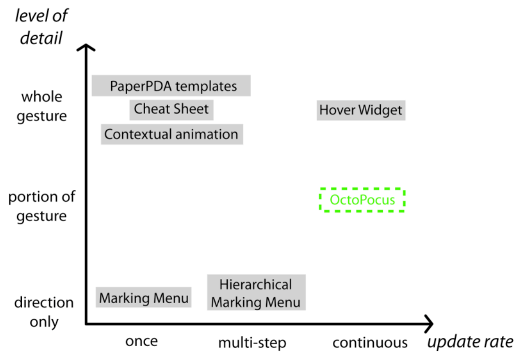

Fig. 1: Six feedforward approaches for guiding gesture input

Feedback Mechanisms

Unlike feedforward systems that are independent of a user's gesture, feedback systems are closely tied to the gesture recognition process and typically involve three phases:

1. *Acquire raw recognition data from the recognizer.* Most algorithms output gesture-input recognition rates according to each gesture class of a given vocabulary. Depending upon the recognition algorithm, those rates may produce classifications, distances or probabilities. The resulting values may be binary (recognized or not) or real (percentages). A data sheet is produced for a given input, with all gesture classes and corresponding recognition rates.
2. *Choose raw, converted or filtered recognition data.* A pre-determined threshold may be used to convert a raw percentage into a binary 'recognized or not' message. Similarly, a filter may select which command subset remains available for the user.
3. *Represent feedback to users.* Feedback may take different forms, e.g., highlighting the label of a recognized command or displaying the probability of recognizing each of several commands.

Recognition systems can display feedback to the user based on information generated during any of these phases. We identified four dimensions for classifying feedback systems, each associated with one recognition process phase:

Recognition value: The recognition rate of a particular gesture may be expressed as a binary or real value.

Filtering: Feedback may include any portion of the gesture set: one gesture, a subset or all gestures.

Update rate: Feedback may be updated at different rates during input: once after input, in multiple discrete steps or continuously.

Representation: Recognition may be visualized differently, from highlighting a command name to beautifying current input by replacing it with the recognized gesture class' template.

Fig. 2 classifies four systems, *Fluid sketches, Interactive beautification*, *GDT* and *Marking menus* which were chosen to represent the range of different feedback systems. Three systems, *Marking menus*, *Interactive beautification* and *Fluid Sketches*, generate a binary **recognition value**, expressed as 'recognized or not'. This is simple and easy to interpret, but offers limited visibility as to the algorithm's recognition state. *GDT's class windows* provide real values about Rubine's training and recognition states, giving gesture-set designers precise feedback about how well a gesture is recognized within a particular gesture class.

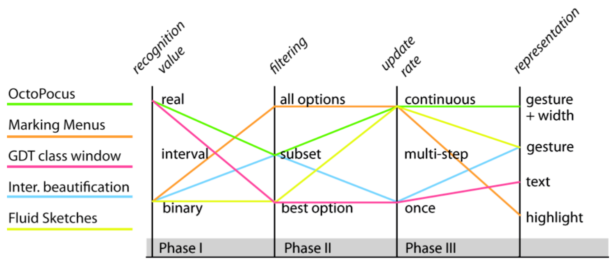

Fig. 2: Feedback options during the gesture recognition process

Filtering permits display of subsets of possibly recognized gestures, allowing flexibility in the face of ambiguity and error. Strategies range from *Marking menus*, which display all possible commands, to *Interactive beautification*, which displays the subset of most likely commands, to *Fluid sketches*, which show only the best recognized command. *GDT*'s class window feedback does not filter, because the user focuses on only one gesture class.

The **update rate** ranges from just once for *Interactive beautification* and *GDT* to continuous display during input for *Marking Menus* and *Fluid sketche*s which reflect the evolution of current input recognition states.

Each system offers a different form of **representation**. *Interactive beautification* and *Fluid sketches* emphasize perfect gesture templates, whereas *Marking menus* highlight the recognized command's label and *GDT's class window* displays a real value of the recognition rate. Graphical representations that remain close to the actual gestures help users focus on the actual form of the gestures.

Dynamic Guides

If we examine these two sets of dimensions together, several empty spaces are apparent. Feedforward systems provide either very limited or complete final information; none provide intermediate information showing what is left to draw among the remaining subset of possible gestures. Similarly, feedback systems provide either binary recognition values, expressed as complete gestures, or percentage numeric values. None reveal a graphical representation of the recognizer's state during execution of the gesture.

Hierarchical marking menus are a good example of dynamic feedforward for a simple gesture set. GDT's class window is a good example of gradual feedback. The challenge is how to continuously issue dynamic feedforward and gradual feedback *during* input. We propose *dynamic guides* as a general approach for providing users with continuously updated information: feedforward about the user's current set of options and feedback about how well the current gesture has been recognized. The next section describes the design and implementation of *OctoPocus*, a dynamic guide that accommodates both incremental and non-incremental recognition algorithms and can be adapted to a wide range of gesture sets, from simple, direction-only marks to arbitrarily complex gestures.

OCTOPOCUS

Like Marking menus, OctoPocus appears after a "press and wait gesture" of approximately 250ms. However, for OctoPocus, both feedforward and feedback are continuously updated as the gesture progresses. Novice users may display a map of all possible gestures and commands, centered around the current cursor position, to help them learn the associations between gestures and commands. After the user selects and begins to make a gesture, less likely gesture guide paths become thinner and disappear. OctoPocus reveals each gesture's ideal future path as well as how the current gesture has been interpreted by the recognizer. Because OctoPocus appears only if the user hesitates, experts can execute commands very efficiently, but can slow down at any time to see which gestures and commands remain.

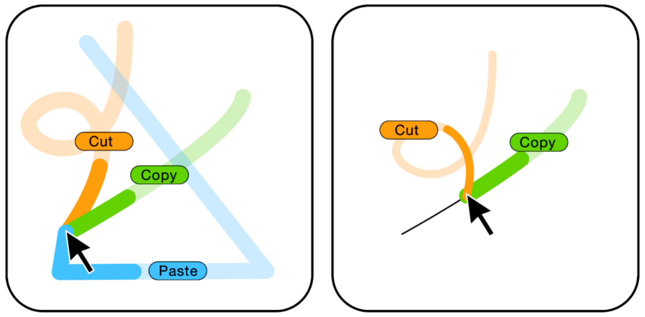

Fig. 3: OctoPocus displays three gestures and commands. Tracing ***copy*** causes ***paste*** to disappear and ***cut*** to get thinner.

Fig. 3 shows how OctoPocus appears to a novice user who is learning gestures associated with *copy*, *cut* and *paste*. As the user begins to follow the *copy* guide path, the *paste* path quickly disappears and the *cut* path becomes progressively thinner, indicating that it is less likely to be recognized. If the user returns to the starting point of the gesture without releasing the mouse button, OctoPocus resets itself.

Feedforward: OctoPocus uses templates to represent each gesture class. These may be extracted automatically from a set of examples or defined when the gesture vocabulary is designed. Each class requires a 'perfect' gesture example that, if drawn by the user, will lead to perfect recognition. All templates begin with a prefix that shares the same starting point radiating from the cursor. After a "press and wait" gesture, prefixes of each template appear around the cursor and serve as guides for drawing the gesture correctly.

Fig. 4 shows how a single gesture template evolves. The prefix proceeds from the cursor and is rendered in solid blue. The associated command always appears at the end of the prefix; the rest of the path is translucent. As the user moves along the path, the prefix moves accordingly. Subtracting the current input length generates a sub-template that represents the remaining path. The prefix length remains constant until the path length is less than its prefix, at which point only the remaining path is highlighted.

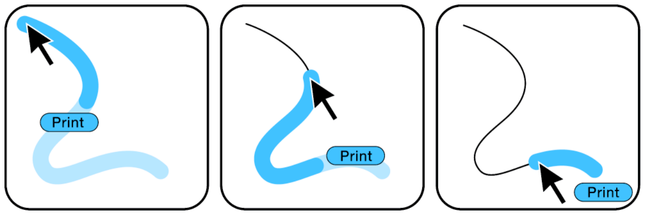

Fig. 4: The prefix serves as a guide for drawing the gesture

Feedback: In addition to highlighting the currently recognized command, OctoPocus varies the template path's thickness to indicate the evolving state of the recognition process. This requires continuous or discrete values from the recognition algorithm. We have adapted OctoPocus to work with both non-incremental and incremental recognition algorithms, using a version of Rubine's algorithm and an incremental turning angle representation, respectively.

Classification algorithms are able to define the closest gesture class for any given input. To reduce inconsistencies, they use a distance measure that indicates how well the input maps to a specific gesture class. Our Rubine-based implementation of OctoPocus uses the Mahalanobis distance for gradual feedback. Our turning-angle representation implementation classifies gestures by computing only distance.

Based on these continuous distance values, we compute the *consumable error rate* for each vocabulary class. Note that other continuous values could be used, depending upon the recognition algorithm. The *consumable error rate* corresponds to the remaining amount of possible user error, before the input can no longer be recognized as a member of a given gesture class. We map the consumable error rate onto the thickness of the guide. Figure 5 shows the evolution of path thickness during input. Initially, all paths have the same thickness. The path being followed retains this thickness, since no errors have been made. Other paths that are not being followed become progressively thinner as error accumulates and then the path finally disappears.

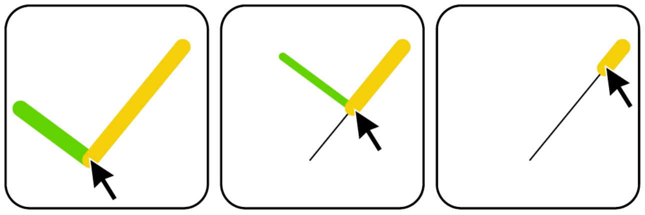
Fig. 5: Path thickness is mapped to the current gesture recognition state, becoming thinner and then disappearing as recognition becomes unlikely and then impossible.

We use part of the template to complete the gesture and then compare it to the corresponding class. This allows us to compute the consumable error rate for incomplete input when using non-incremental recognition algorithms. Assuming T is the template or 'perfect' gesture for class C, our algorithm proceeds as follows for each template:

1. Subtract the prefix of the length of user's input from the full template T, resulting in a sub-template subT.
2. Concatenate the current user's input with sub-template subT. The resulting shape perfT is the user's completed input together with a perfect drawing for a given class.
3. Use the recognizer to compute the distance between the resulting shape perfT and the gesture class C associated with the template.
4. Compute the difference between the computed current value and a given threshold. This gives gives user's room for error before reaching the distance threshold i.e. before the input no longer resembles an element of class C, from the recognizer's perspective.

This algorithm allows users to adapt their gestures in real time with respect to the evolving probability of error. Of course, for very short gestures, users may not have time to incorporate this gradual feedback during input. However, for long gestures or when input is repeatedly misrecognized, the user may slow down and examine which section of the gesture decreases the error rate.

Note that OctoPocus can be adapted to use both scale-dependent and scale-independent algorithms. With the latter, expert-mode gestures (performed *without* OctoPocus) may be executed at any scale. However, for novice-mode gestures (performed *with* OctoPocus), the current version of OctoPocus uses a single, standard-size template for each gesture, scaled appropriately for the current screen.

Managing Visual Complexity: Dynamic guides demand a trade-off between *visual search* and *exploration*. For example, Hierarchical Marking menus were designed to increase how many items a Marking menu can handle. At any one time, users have a less complex visual display but must progress through one or more levels of a hierarchy. In contrast, OctoPocus risks creating visual complexity by displaying many items at the same time, but does not require the user to explore multiple levels.

To reduce OctoPocus' visual complexity, we highlight only the initial prefix of each gesture and the corresponding command. Rendering the ends of each path translucent helps the user focus on the next portion of the gesture to be drawn while still providing an overview of the whole gesture. We also reduce the thickness of unlikely commands and quickly eliminate impossible alternatives. The worst case of visual complexity occurs when OctoPocus appears at the very beginning of an input.. Yet even here, the user can ignore the paths and focus on the desired command label. Once the user finds it and starts tracing the related gesture, improbable gesture paths quickly disappear.

Fig. 6 shows the rapid decrease in visual complexity due to the number of gestures surrounding the cursor, for an arbitrary 16-command gesture set. In this example, an average of six of the 16 templates remain around the cursor after an input length of approximately 50 pixels. The mean length of gestures in this example is 533 pixels.

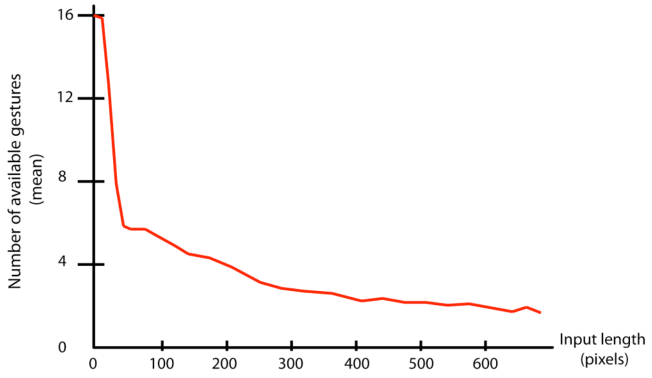

Fig. 6: Reducing visual complexity through rapid elimination of paths that are unlikely to be recognized.

We can create gesture sets that let OctoPocus take advantage of this by designing gestures that share a common prefix and clustering them. For example, S, C and O all begin with the same initial arc segment and form a pseudo-hierarchical structure. Users begin by drawing the initial arc and then complete the gesture differently to indicate the desired letter. This approach also helps when using a touch screen. We can reduce occlusion from the user's hand by creating gestures that fan out from a common prefix, grouping them to appear in the most visible part of the screen.

EXPERIMENTS

We conducted two experiments to determine whether the dynamic feedforward and feedback provided by OctoPocus helps users to learn and execute gesture sets. Experiments that test learning are notoriously difficult to control and are often conducted with a between-subjects design. However this increases inter-subject variability, due to individual differences in learning patterns, and makes results more difficult to compare. We chose instead to use a within-subjects design that explicitly avoids training effects for subsequent conditions. This requires thorough counter-balancing of all trials and blocks, across subjects, and also ensuring that one condition does not influence the next.

Experiment 1 was influenced by the observation that few real-world systems use complex shapes as gesture sets, despite many potential applications. They might simply be too difficult to learn and execute: Not only must the user learn the association between a complex gesture and its command, but he or she must also master the details of drawing the shape to improve recognizer accuracy. If so, providing continuous feedforward and feedback facilitates learning and execution of complex gesture sets, gives interface designers more options. Experiment 1 compares a traditional, static Help menu, with the dynamic feedforward provided by OctoPocus. We chose arbitrary (non-mnemonic) gestures to ensure the difficulty of the task.

Experiment 2 examined whether OctoPocus could help even with gesture sets that have already been shown to be simple and efficient, such as the marks used with Hierarchical marking menus. Does continuous rather than discrete feedback continuous aid performance? Is it faster to learn? If so, then OctoPocus could be used as a single, efficient technique for a wide range of gesture sets. We compare Hierarchical marking menu (discrete) feedback with OctoPocus's continuous feedback, adapted to recognize the same two-stroke gesture set used in a two-level Hierarchical marking menu with four items at each level.

Both experiments use 16-item gesture sets because they are difficult to learn and exceed the limits of short-term memory. Each guide condition tests eight of each set of 16 gesture-command pairs. This reduces overall run-time for participants and the eight non-tested gestures serve as a secondary control, allowing us to compare learning with no guide to learning with each type of guide. We also assess whether simple exposure to gesture-command alternatives increases the user's likelihood of learning non-trained commands.

Procedure

Both experiments use a one-factor within-subjects design and follow the procedure outlined in Fig. 7. Techniques and gesture sets are counter-balanced across participants. An initial practice session prior to each experiment lets participants learn about and practice the guides they will use in the experimental conditions. None of the specific commands used in the experimental conditions appeared in these practice sessions.

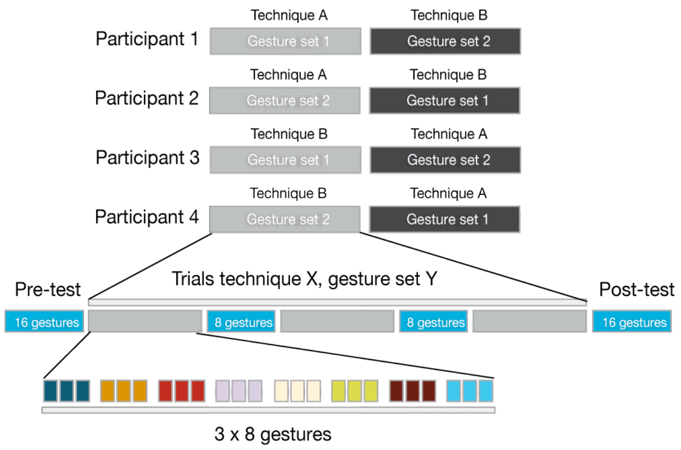

Fig. 7: Pre-test / post-test design for both experiments, with trial counterbalancing strategy.

Pre-test: We use a *pre-test/post-test* experimental design, which is standard for learning experiments. Prior to any training during the experimental conditions, we present participants with a block of 16 command names and ask them to generate a gesture for each. They are told to just guess, since they are not expected to know the gestures at this point. This provides the baseline level of performance and error rates, and forms the basis for comparison of each successive post-test. Note that since the associations between gestures and commands are arbitrary, we expect initial performance to be at chance, near 0%.

Training Condition: Each training condition teaches a new command set using one of two on-line guides. Participants are exposed to 16 gestures and 16 associated commands of which we test performance on eight. All conditions are counterbalanced for order across participants and guide type. Each training condition consists of three blocks of 24 trials, followed by a post-test.

Each trial begins by displaying a square on the screen. When the participant clicks within this square, a command name appears and the clock starts. If the participant chooses to *not* use any form of assistance, the trial is in ***expert mode***. If the participant chooses to obtain assistance, which varies according to the specific condition, the trial is in ***novice mode***. When the participant releases the mouse button after input, the command that corresponds to the drawn gesture appears and the trial clock ends. If correct, this command corresponds to that already on the screen. If incorrect, either another command recognized by the system appears or else the phrase "not recognized".

Each block consists of 24 trials (eight commands with three replications of each command) and tests the same eight commands, of the possible 16 in the command set. Within a block, each command is presented three times in a row. We ask participants to learn each command and perform the associated gesture as quickly and as accurately as possible, trying to improve performance each time.

Post-test: After each block of 24 trials, participants are asked to perform each of the eight commands they have learned, without any guidance. We display their overall score at the end of the test. The final posttest is identical to the pre-test i.e. participants perform all 16 gestures without any guidance. This allows us to also assess whether participants learn commands for which there was no guide at all.

Debriefing: After completing all sessions with both on-line guides, we ask participants which type of guide they prefer and how they judge their own levels of performance with respect to learning, errors and speed.

Data collection: We capture all generated gestures as well as start and stop times. We classify trials as in *expert mode* i.e. without assistance or in *novice mode* i.e. the participant used one of the on-line guides: OctoPocus, Help Menu or Hierarchical Marking Menu. For trials in novice mode, our time measure includes 1. time from trial start to the time the participant chose assistance and 2. time from when the guide appears to the time when the gesture is complete.

Trials are classified as either *correct*: the participant performed the requested gesture and our system recognized it, or *incorrect*: the participant performed a gesture that the system recognized, but it was not the command requested.

Data analysis: Within-subjects repeated measures designs expose participants to equivalent conditions, counterbalanced for order. We performed an ANOVA and accounted for *repeated* measures by treating subject (participant) as a random variable. We also used JMP's REML function to account for *replicated* measures, when participants are exposed to multiple instances of each condition.

EXPERIMENT 1

Experiment 1 compared OctoPocus to a standard Help menu. Our hypothesis was that the combination of dynamic feedforward and feedback provided by OctoPocus would lead to better gesture learning and reduce input errors, without requiring extra input time.

Participants: 16 (14 men and 2 women) with medium- to expert-level computer experience. All were right-handed.

Apparatus: The experiment was conducted on an Intel 2Ghz laptop, using a mouse for input and a 15'' display. The system ran on MacOs X; all software was implemented in Java 1.4. We used Rubine's algorithm and the authors trained the gesture classes. Final training of the classifiers was adjusted based on the results of four pilot experiments. Our pilot studies allowed us to adjust the gestures to ensure similar input times and errors and we distributed gesture complexity evenly across the two techniques.

Test items: We created a difficult 16-item gesture-command set. OctoPocus was available for eight gestures and a traditional Help menu was available for eight other gestures. Fig. 8 shows the two guides at the first moment of novice mode. We randomly linked gestures to city names and verified that there were no obvious relations between them, e.g., a round shape for Oslo.

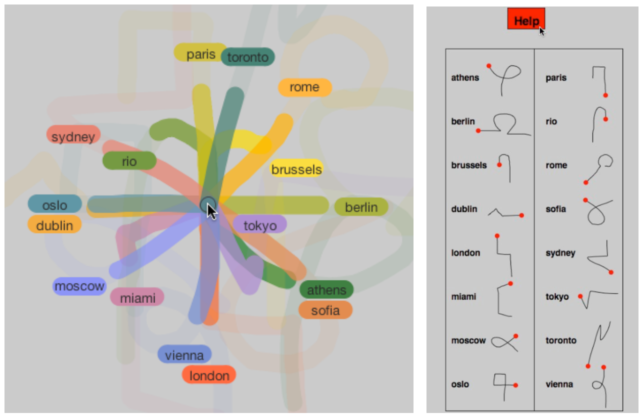

Fig. 8: OctoPocus and a standard Help menu, each showing 16 arbitrary gesture-command pairs

The Help menu pops up when the user presses the Help button. It provides an alphabetical "cheat sheet" with each command and its associated gesture; a red dot shows the starting position. On release, the Help menu disappears. Note that this is more accessible than in most real applications, which hide help menus within a menu hierarchy. We did not implement Kurtenbach et al.'s 'crib sheet' because, with 16 gestures, it would occlude too much of the screen. In novice mode, it takes 250ms to display OctoPocus.

Experiment 1 compares two strategies: *learn first, then do* (Help menu) and *learn while doing* (OctoPocus). We were interested in whether the dynamic nature of OctoPocus improves learning of a set of arbitrary gesture-based commands, compared to a conventional Help menu.

Results

For trials in *novice mode* i.e. trials in which the participant chose to use one of the guides, the overall input time was significantly faster for OctoPocus, 5.7 sec., than the Help Menu, 5.9 sec., F(1,79)=5.27; p=0.024. We define overall input time as the time from the start of the trial until the gesture is complete. Some of this time is due to the difference in mean time to access the Help Menu, mean = 1.3 sec versus OctoPocus, mean = 1.1 sec, which are significantly different from each other, F(1,13)=10.03; p=0.007). We define input time as the time from when a guide of either type appears on the screen until the gesture is complete. We might expect the time for OctoPocus to be significantly longer, since the user must slow down to follow the desired path instead of just glancing at the Help menu and executing the gesture. However, we found no significant differences between OctoPocus and the Help Menu, (F(1,13)=0.55 ; p=0.47).

Overall error rates were low, with no significant differences obtained between OctoPocus, 5%, and Help Menu, 7%. However, if we look at the overall error during training sessions, including those in expert mode when users did not choose additional help, we find significantly *fewer errors* for gestures trained with OctoPocus, 4%, compared to the Help menu, 8%, F(1,79)=7.30; p=0.008.

Participants used OctoPocus significantly more often, 40%, than the Help Menu, 32%, F (1,15)=14.40; p=0.002. This is consistent with comments from several participants who said OctoPocus was 'less risky' than the Help Menu. When in doubt, they could hesitate and get a quick hint from OctoPocus, without having to decide to move the mouse to go the Help menu. For 'expert' trials in which the user did not seek help, OctoPocus, 4%, resulted in significantly fewer errors than the Help menu, 7%, F(1,79)=6.81; p=0.010.

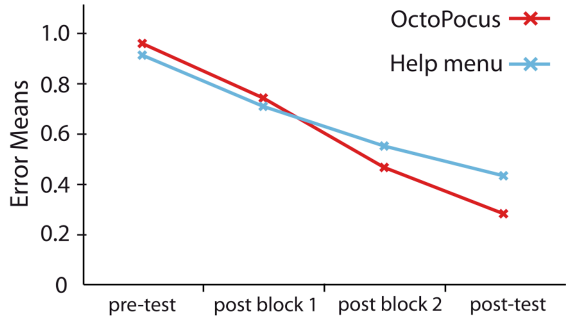

Fig. 9: Learning curve: Evolution of mean error over time. Pre-test: both almost 100% error. Post-test: significantly fewer errors for OctoPocus (27%) than Help Menu (43%).

The series of post-tests after each training block tested the participants' ability to perform each gesture without any guidance (Fig. 8). Users performed significantly better with OctoPocus, both in terms of the learning curve, F(3,105)=3.00; p=0.033 and final post-test performance F(1,15)=7.98; p=0.012 (Fig. 9). In terms of final expertise, OctoPocus had a significantly lower error rate, 27 %, compared to the Help menu, 43% F(1,15)=7.97 ; p=0.012. Thirteen participants preferred OctoPocus' learning-by-drawing approach and 15 said they preferred OctoPocus because it is more "playful".

EXPERIMENT 2

Experiment 2 compared OctoPocus to a Hierarchical Marking menu for learning a standard mark-based gesture set, with two levels of hierarchy (Fig. 10). We wanted to test whether OctoPocus could take advantage of the efficiency already demonstrated with mark-based gesture sets. Our hypothesis is that the combination of dynamic feedforward and feedback provided by OctoPocus would lead to better gesture learning and reduce input errors, without requiring extra input time.

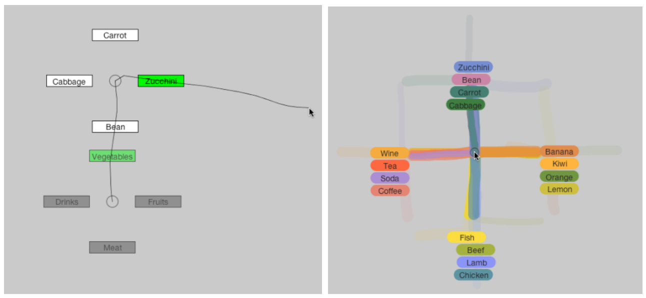

Fig. 10 : Hierarchical Marking Menu and OctoPocus, adapted to two-stroke marks.

Participants: 12 (9 men and 3 women) with medium- to expert-level computer experience. All were right-handed.

Apparatus: The experiment was conducted on an Intel 2Ghz desktop computer with a 23'' display and a 21" Wacom tablet in absolute mode with a pen for input. The system ran on MacOs X; all software was implemented in Java 1.4. We used the original discrimination algorithm based on gesture directions and adapted OctoPocus to Rubine's algorithm. The authors trained the gesture set, which we adjusted based on the results of four pilot experiments. These allowed us to ensure that angle-based recognition was equivalent to recognition based on Rubine's algorithm, when applied to the same set of 16 gestures.

Test items: We used the set of 16 gestures that correspond to a two-level Hierarchical Marking Menu, with four items at each level. Level one contained four food categories; level two contained four specific foods for each general category. This was designed to slightly favor Hierarchical Marking menus at early learning stages, since users need not explore all 16 items and have, at worst, one chance in four of choosing the right gesture, compared to one chance in 16 for OctoPocus.

We trained eight gestures from each set of 16 gestures and associated commands varying only the style of guide: Hierarchical Marking menu and OctoPocus. We ensured that no marks had an unintended symbolic link, e.g., *L* for *Lamb*.

Experiment 2 compared two strategies: *multiple discrete steps* (Hierarchical Marking Menus) and *dynamic guide* (OctoPocus). The former requires *exploration*, stepping through fewer options in two steps whereas the latter requires *visual search*, with all options simultaneously available in a progressively simpler display.

Results

Input time in *novice mode* (Fig. 11) was significantly faster for OctoPocus, mean = 2.6 sec, than for Hierarchical Marking menu, mean = 3.1 sec, F(1,59)=11.84; p=0.001. The mean difference between the two input times is 555 ms, which is higher than the 250ms wait time needed to make the second-level Hierarchical Marking menu appear.

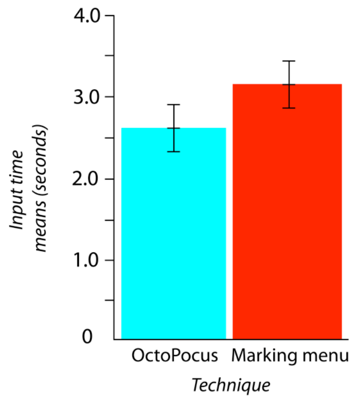

Fig. 11: Differences in input time when in novice mode

In novice mode, no significant differences obtain in error rates between OctoPocus, 4% and Hierarchical Marking menus, 2%, F(1,59)=3.75; p=0.058. Similarly, the evolution of error rates over the series of post-tests does not show a significant difference F(3,77)=1.75, p=0.162.

DISCUSSION

Experiment 1 shows that, for a set of 16 complex gestures, users learn better with OctoPocus, without incurring an additional cost in time, compared to a traditional Help menu. OctoPocus also requires significantly less access time than the Help menu. In real-world applications, Help menus are often buried in a menu hierarchy that would take even more time. Users also preferred the "in situ" learning of gestures with OctoPocus, compared to simply reading a Help Menu list.

Experiment 2 shows that OctoPocus can be adapted to diverse gesture sets, including marks. Given a 16-item mark-based gesture set, with two levels and four items at each level, OctoPocus incurs no additional learning cost, nor does it result in additional errors. However, it does improve input time for users while they are in *novice mode.* We plan to test OctoPocus with other marking-based gestures sets.

Learning: OctoPocus helps users to *learn* gesture-command sets by showing the path of each possible gesture and its associated command. This highlights a key advantage of OctoPocus over other help systems; i.e. it can be applied to complex gesture sets that are highly mnemonic and specifically designed for a particular application.

Execution: OctoPocus also improves how users *execute* their gestures, thus improving subsequent recognition. Because gesture recognition algorithms are opaque, users often have trouble identifying which specific aspects of a particular gesture are deemed necessary by the recognizer. Users tend to focus on the shape and iconic meaning of a gesture, whereas recognizers focus on quantitative characteristics of the gesture. For example, some execution errors occur when the recognition algorithm attends to the direction in which a gesture is executed but the user focuses only on the final result. If the user moves clockwise to draw a circle and the recognizer requires a counterclockwise movement, the system will generate an error and the user will be confused. OctoPocus, like some cheat sheets and other guides, addresses this problem by showing the direction in which to draw each stroke as it radiates from the current cursor position.

OctoPocus also shows the subsequent optimal path for a 'perfect' gesture, from the recognizer's perspective. Other errors may result if the recognizer focuses on the number of direction changes and the user focuses only on the overall shape. Thus the system may recognize a circle and a square as the same, whereas, to the user, they are distinct gestures. OctoPocus addresses this by varying the thickness of the guide path to indicate the current level of error in the gesture as drawn.

Remembering: OctoPocus also helps users to *remember* previously learned gestures by providing a quick hint of the correct path at any point during the execution process. Similarly, OctoPocus helps users to remember gesture command-associations by displaying the command label in the middle of each possible gesture. This helps users to understand errors that result from mis-remembering gesture-command associations. For example, if a command executed in expert mode is recognized, but the 'wrong' command label is displayed at the end, the user can find out why by entering novice mode and checking to see which gesture would have led to the desired command. OctoPocus is especially valuable for intermittent users of gesture-based applications. If the user already has a rough idea of a gesture, she can dramatically reduce visual complexity by beginning the gesture and then hesitating, so that OctoPocus only reveals the few possible commands that remain.

FUTURE WORK

The main limitation of OctoPocus is the level of visual complexity that users face if they enter novice mode without having drawn any portion of a gesture. As explained earlier, we address this by making the ends of paths translucent and rapidly eliminating impossible alternatives. However, we also plan to explore other variations of dynamic guides that address this problem differently. We would like to explore the practical limits of OctoPocus with respect to both gesture set size and gesture type. We chose a 16-item set for our experiments because it exceeds short-term memory limits and poses a sufficiently difficult learning task but we need to explore how well it scales.

Our algorithm for completing the user's gesture with the template is not always optimal, since early parts of the gesture may introduce error and another shape that is not part of the template may in fact produce a better recognition result. We plan to improve our current algorithm for more optimal completion of the user's input and explore the use of scale-independent templates. We also plan to develop a hierarchical version of OctoPocus, building upon Rubine's concept of eager recognition. Our goal is to combine different instances of OctoPocus to create different types of gestures and thus invoke commands within a hierarchy.

CONCLUSION

This paper addresses the problem of helping users to learn, execute and remember gesture-based command sets. We examined existing feedforward and feedback systems that provide on-screen guidance in gesture-based interfaces and classified them along six dimensions in a design space. We then introduced the concept of *dynamic guides*, which combine dynamic feedforward and feedback to directly guide novice user's performance, without penalizing expert users. We describe *OctoPocus*, a dynamic guide that continuously updates the state of the recognition algorithm by gradually modifying the thickness of possible gesture paths, based on its 'consumable error rate'. We have shown that users can better learn, execute and remember gesture sets if we reveal, during input, what is normally an opaque process i.e. the current state of recognition, and represent gestures in a graphical form that shows the optimal path for the remaining alternatives.

The notion of dynamic guides suggests a more general approach that can be applied to a variety recognition-based input systems. Instead of creating autonomous recognizers that simply provide feedback about the final result, we can explicitly create a dynamic collaboration between the human user and the recognition system. If the recognizer provides the user with continuously updated feedback about the current state of the recognition algorithm, combined with feedforward about the remaining possible options, users can increase their understanding how best to interact with the system while simultaneously modifying their behavior to reduce errors and improve overall recognition accuracy.

Applications such as Eisenstein & Mackay's Object Tracker [9] use the same basic principle to improve computer vision recognition of human movement gestures. However, we believe that this notion of a continuous, interactive guide, in which the user receives dynamic feedback during any type of recognizable performance from the computer-based recognizer can be extended to a wide variety of recognition-based systems, including voice recognition and ubiquitous computing applications. We plan to extend the current design space to classify a wider range of feedforward and feedback systems and use it as a generative technique for creating additional dynamic guides.

ACKNOWLEDGMENTS

Our thanks to the study participants and Michel Beaudouin-Lafon, Emmanuel Pietriga, Sofiane Guedanna, Aurélien Tabard and Olivier Chapuis for comments on earlier drafts.

REFERENCES

1. Agar, P. & Novins, K. (2003) Polygon Recognition in Sketch-Based Interfaces with Immediate and Continuous Feedback. In *Proc. GRAPHITE'03, Computer Graphics and Interactive Techniques*. pp. 147-150.
2. Arvo, J. & Novins, K. (2000) Fluid sketches: continuous recognition and morphing of simple hand-drawn shapes. In *Proc. UIST'00 ACM User interface Software & Technology*, pp. 73-80.
3. Arvo, J. & Novins, K. (2006) Fluid sketching of directed graphs. In *Proc. Australasian User Interface Conf.* Vol. **50**, Australian Computer Soc., pp. 81-86.
4. Avrahami, D., Hudson, S., Hudson, S., Moran, T., & Williams, B. (2001) Guided gesture support in the paper PDA. In *Proc. UIST'01 ACM User interface Software & Technology*, pp.197-198.
5. Blinkenstorfer, C. (1995) Graffiti. *Pen Computing*, p. 30-31.
6. Callahan, J., Hopkins, D., Weiser, M. & Shneiderman, B. (1988) An empirical comparison of pie vs. linear menus. In *Proc. CHI'88 ACM Human Factors in Computing Systems*. pp. 95-100.
7. Cao, X. & Zhai, S. (2007) Modeling human performance of pen stroke gestures. In *Proc. CHI'07 ACM Human Factors in Computing Systems*. pp. 495-1504.
8. Chatty, S. & Lecolinet, E. (1996) Pen Computing for Air Traffic Control. In *Proc. CHI'96 ACM Human Factors in Computing Systems*. pp. 87-94.
9. Eisenstein, J. & Mackay, W. (2006) Interacting with Communication Appliances: An evaluation of two computer vision-based selection techniques. In *CHI'06 ACM Human Factors in Computing Systems*. pp. 1111-1114.
10. Goldberg, D. & Richardson, C. (1993) Touch-typing with a stylus. In *Proc. CHI'93 ACM Human Factors in Computing Systems.* pp. 80-87.
11. Grossman, T., Hinckley, K., Baudisch, P., Agrawala, M., & Balakrishnan, R. (2006) Hover widgets: Using the tracking state to extend the capabilities of pen-operated devices. In *Proc. CHI'06 ACM Human Factors in Computing Systems*. pp. 861-870.
12. Guimbretiere, F. & Winograd, T. (2000): FlowMenu: Combining Command, Text, and Data Entry. In *Proc. UIST'00 ACM User interface Software & Technology*, pp. 213-216.
13. Hong, J. & Landay, J. (2000) SATIN: A toolkit for informal ink-based applications. In *Proc. UIST'00 ACM User Interface Software & Technolog*y, pp. 63-72.
14. Hse (2004) Hse, H., Shilman, M. & Newton, A.. (2004). Robust Sketched Symbol Fragmentation using Templates. In *Proc. IUI'04 Intelligent User Interfaces*. pp. 156-160
15. Iannizzotto, G. & Vita, L. (1999) A Multiscale Turning Angle Representation of Object Shapes for Image Retrieval. In *Proc. Visual information & Information Systems*. Vol. **1614**, Springer-Verlag, pp. 609-616.
16. Igarashi, T., Matsuoka, S., Kawachiya, S., & Tanaka, H. (1997) Interactive beautification: a technique for rapid geometric design. In *Proc. UIST '97 ACM User Interface Software & Technology*. pp. 105-114.
17. Igarashi, T., Kawachiya, S., Tanaka, H., and Matsuoka, S. (1998) Pegasus: a drawing system for rapid geometric design. In *Proc. CHI'98 ACM Human Factors in Computing Systems*. pp. 24-25.
18. Kurtenbach, G. & Buxton, W. (1993) The limits of expert performance using hierarchic marking menus. In *Proc. CHI '93 ACM Human Factors in Computing Systems*. p. 482-487.
19. Kurtenbach, G. (1993) The Design and Evaluation of Marking Menus, *Ph. D. Thesis*, Dept. of Computer Science, University of Toronto.
20. Kurtenbach, G. & Moran, T. (1994) Contextual Animation of Gestural Commands. In *Proc. Eurographics Computer Graphics Forum.* Vol. **13**(5), 305-314.
21. Li, J., Zhang, X., Ao, X., and Dai, G. (2005) Sketch recognition with continuous feedback based on incremental intention extraction. In *Proc. IUI'05 Intelligent User Interfaces*. pp. 145-150.
22. Liao, C., Guimbretière, F. & Hinckley, K. (2005) PapierCraft: A Command System for Interactive Paper. In *Proc. UIST'05 ACM User interface Software & Technology*, pp. 241 – 244.
23. Long, A., Landay, J. & Rowe, L. (1999) Implications for a gesture design tool. In *Proc. CHI '99 ACM Human Factors in Computing Systems*. pp. 40-47.
24. Long, A., Landay, J., Rowe, L. & Michiels, J. (2000) Visual Similarity of Pen Gestures. In *Proc. CHI'00 ACM Human Factors in Computing Systems*. pp. 360-367.
25. Mackay, W., Fayard, A., Frobert, L. & Médini, L. (1998) Reinventing the Familiar: Exploring an Augmented Reality Design Space for Air Traffic Control. In *Proc. CHI'98 ACM Human Factors in Computing Systems*. pp. 558-565.
26. Mankoff, J., Hudson, S. E., & Abowd, G. D. (2000) Interaction techniques for ambiguity resolution in recognition-based interfaces. In *Proc. UIST '00 ACM User Interface Software & Technology*. pp. 11-20.
27. Moyle, M. & Cockburn, A. (2003) The design and evaluation of a flick gesture for 'back' and 'forward' in web browsers. In *Proc. Australasian User Interface Conf.* Vol. **18**, Australian Computer Soc., pp. 39-46.
28. Perlin, K. (1998) Quikwriting: Continuous Stylus-based Text Entry. In *Proc. UIST '98 ACM User Interface Software & Technology*. pp. 215-216.
29. Rubine, D. (1991) Specifying gestures by example. In *Proc. ACM SIGGRAPH Computer Graphics.* **24**(4):329-337.
30. Rubine, D. (1992) Combining gestures and direct manipulation. In *Proc. CHI '92 ACM Human Factors in Computing Systems*. pp. 659-660.
31. Wobbrock, J., Wilson, A. & Li, Y. (2007) Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *Proc. UIST '07 ACM User Interface Software & Technology*. pp. 159-168.
32. Zhao, S. & Balakrishnan, R. (2004) Simple vs. compound mark hierarchical marking menus. In *Proc. UIST '0. ACM User Interface Software & Technology*. pp. 33-42.
33. Zhao, S., Agrawala, M. & Hinckley, K. (2006) Zone and polygon menus: using relative position to increase the breadth of multi-stroke marking menus. In *Proc. CHI '06 ACM Human Factors in Computing Systems*. pp. 1077-1086.