

Implications For a Gesture Design Tool

Allan Christian Long, Jr., James A. Landay, Lawrence A. Rowe

Electrical Engineering and Computer Science Department

University of California at Berkeley

Berkeley, CA 94720-1776 USA

{allanl,landay,larry}@cs.berkeley.edu

+1 510 {643 7106, 643 3043, 642 5615}

<http://www.cs.berkeley.edu/~{allanl,landay,larry}>

ABSTRACT

Interest in pen-based user interfaces is growing rapidly. One potentially useful feature of pen-based user interfaces is gestures, that is, a mark or stroke that causes a command to execute. Unfortunately, it is difficult to design gestures that are easy 1) for computers to recognize and 2) for humans to learn and remember. To investigate these problems, we built a prototype tool for designing gesture sets. An experiment was then performed to gain insight into the gesture design process and to evaluate the tool. The experiment confirmed that gesture design is very difficult and suggested several ways in which current tools can be improved. The most important of these improvements is to make the tools more active and provide more guidance for designers. This paper describes the gesture design tool, the experiment, and its results.

Keywords

pen-based user interface, PDA, user study, gesture, UI design

INTRODUCTION

This work explores the process of gesture design with the goal of improving gestures for pen-based user interfaces. By observing gesture design in an experimental setting we have discovered certain useful strategies for this task and some detrimental strategies. Our observations also suggest new directions for future gesture design tools. For example, our results indicate that gesture design tools should be much more active participants in the design process than they have been heretofore. Only by improving gesture design tools can we fully realize the advantages of pen-based computing.

Pen and paper has been an important, widely used technology for centuries. It is versatile and can easily express text, numbers, tables, diagrams, and equations [12]. Many authors list the benefits pen-based computer interfaces could enjoy on desktop and portable computing devices [1, 4, 5, 12, 13, 21]. In particular, commands issued with pens (i.e., *gestures*) are desirable because they are faster (because command and operand are specified in one stroke [2]), commonly used, and iconic, which makes them easier to remember than textual commands [13].

Recently, more and more computer users have adopted pen-based computers. Approximately three million hand-held computers were sold in 1997 and sales are expected to reach 13 million by 2001 [6]. The use of pen-based input for desktop systems is also growing as prices for tablets and integrated display tablets fall. As pen-based devices proliferate, pen-based user interfaces become increasingly important.

Gestures are useful on displays ranging from the very small, where screen space is at a premium, to the very large, where controls can be more than arm's reach away [17]. We performed a survey of PDA users [11] which showed that users think gestures are powerful, efficient, and convenient. Users also want applications to support more gestures and some users want to be able to define their own gestures. However, the survey also revealed that gestures are hard for users to remember and difficult for the computer to recognize.

A disadvantage of gestures is that individual gestures and sets of gestures are difficult to design. We focus on two goals for gesture design:

1. *Gestures should be reliably recognized by the computer.*
2. *Gestures should be easy for people to learn and remember.*

An added complication is that gestures in a set affect the recognition, learnability, and memorability of one another so that individual gestures cannot be designed in isolation.

A great deal of interface research has dealt with WIMP (windows, icons, menu, and pointing) GUIs (graphical user interfaces). There is also a substantial body of work on pen-based interfaces, but many hard problems remain. To a first approximation, pens can be used to replace mice, but what is easy with a mouse is not necessarily easy with a pen, and vice versa. For any given task, the ideal pen-based UI will not be limited to techniques developed for GUIs, but will incorporate pen-specific techniques that take advantage of the unique capabilities of pens.

In our work we have decided to concentrate on gestures in the spirit of copy editing [10, 19] rather than marking menus [20], because we believe that traditional marks are more useful in some circumstances. For example, they can specify operands at the same time as the operation, and they can be iconic.

At first, it might seem that the solution to the gesture set design problem is to invent one gesture set and use it in all applications. We believe standard gestures will be developed for common operations (e.g., cut, copy, paste). However, pens (and gestures) will be used for a wide variety of application areas, and each area will have unique operations that will require specialized gestures. For

example, a user will want different gestures for a text markup application than for an architectural CAD application. It is also possible that the best gestures for small devices will be different than the best gestures for large ones. Thus, our solution is to develop tools to support pen-based UI designers in the tasks of inventing gesture sets and augmenting existing sets with new gestures. We want to empower users who do not have experience with recognition or with psychology to create good gesture sets.

As a first step, we designed and implemented a prototype tool for gesture set design, called *gdt*. We conducted an experiment to evaluate *gdt* and to investigate the process of gesture set design. We found that while *gdt* improved the process of gesture set design, the task is still difficult and there is a great deal more a tool can do to aid the designer.

This paper presents *gdt*, an experiment on gesture design, and the results and implications of that experiment. We begin with an overview of the PDA user survey [11], the recognition algorithm used in *gdt*, and a typical recognizer training program. Next, we describe *gdt*. This is followed by a description of the experiment. Then, we present the experimental results and analysis. The next section discusses implications of the experiment for a gesture design tool. The last section gives some conclusions.

BACKGROUND

This section describes some background information that motivated our design of *gdt* and the experiment. The first subsection summarizes a survey of PDA users. The second briefly describes the recognition algorithm we used in our work. The last section describes our experiences with an existing tool for training a gesture recognizer.

PDA User Survey

In the summer of 1997, a survey of ninety-nine PalmPilot and forty-two Apple Newton users was conducted to determine how they viewed gestures and how they used their PDAs (Personal Digital Assistants) [11]. The study found the following:

- Users think gestures are powerful, efficient, and convenient.
- Users want more gestures (i.e., more gesture interfaces to operations).
- Users want to define their own gestures, providing a macro-like capability.
- Users are dissatisfied with the recognition accuracy of gestures. They do not want to sacrifice recognition for the sake of more gestures.

Users want more gestures and the ability to design their own, but designing good gestures is difficult. With current tools, designing gestures that can be recognized by the computer requires knowledge of recognition technology. Similarly, designing gestures that will be easy to learn and remember requires knowledge of psychology. Designers need a tool that encapsulates this knowledge to allow a wider range of people to design good gesture sets.

Gesture Recognition

There are different types of gesture recognition algorithms. Two common ones are neural network- and feature-based. Neural network recognizers generally have a higher recognition rate, but they require hundreds if not thousands of training examples. This requirement virtually prohibits

iterative design, so instead we chose to use the Rubine feature-based recognition algorithm [19], which:

- requires only a small number of training examples (15-20) per gesture class.
- is freely available.
- is easy to implement.
- has a reference implementation available (from Amulet [16]).
- other research systems have successfully used [3, 8].

A gesture set consists of gesture classes, each of which is a single type of gesture (such as a circle to perform the select operation). A class is defined by a collection of training examples. The goal of a recognizer is to correctly decide to which class in the set a drawn gesture belongs. A feature-based algorithm does this classification by measuring properties (called *features*) such as initial angle and total length, and computing statistically which class it is most like. Rubine's algorithm computes eleven different features about each gesture.

Before the recognizer can classify gestures, the designer must first train it by giving it example gestures for each class. During recognition, the feature values of the unclassified gesture are statistically compared with the feature values of the training examples to determine which class the new gesture is most like. For more details about the algorithm, see [19].

Experiences with *Agate*

The Garnet [15] and Amulet [16] toolkits include an implementation of Rubine's algorithm and a training tool for the recognizer called *Agate* [9]. Before beginning our present work, we used these toolkits and *Agate* to understand how they could be improved.

Although *Agate* is a fine recognizer training tool, it was not intended to be a gesture design tool. *Agate* allows the

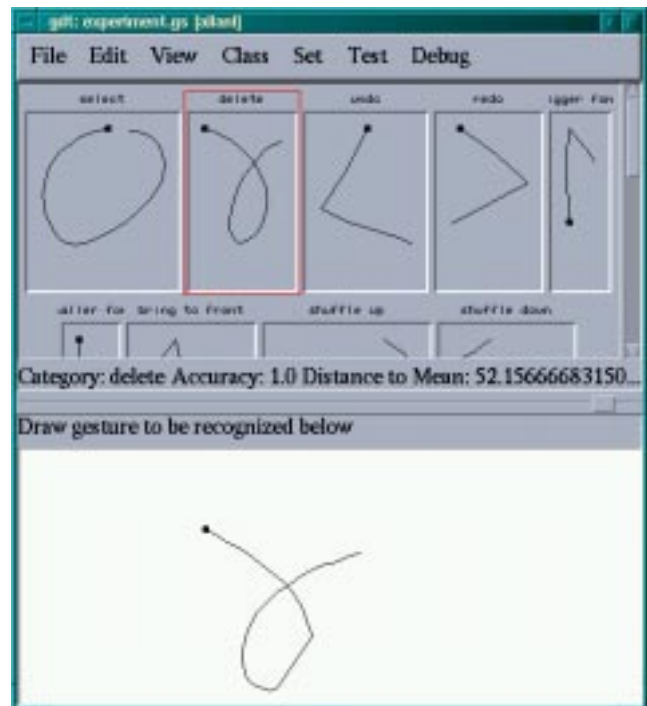


Figure 1: *gdt* main window.

designer to enter examples to be recognized, so a designer can discover that a recognition problem exists. Unfortunately, *Agate* provides no support for discovering why a recognition problem exists or how to fix it. As a first step toward solving this problem, we decided to build a new tool that would expose some of the information about the recognition process. We believed that by showing designers some of the details underlying the recognition, they could more easily determine why recognition problems exist and how to fix them.

GESTURE DESIGN TOOL

We believe pen-based UI designers need a tool to help them design gestures. We built a prototype gesture design tool (*gdt*) that is loosely based on *Agate*. This section first gives an overview of the differences between *gdt* and *Agate* and then describes the different parts of the *gdt* UI in detail.

The significant improvement of *gdt* over *Agate* are visualizations intended to help designers discover and fix recognition problems. Other enhancements include: multiple windows for viewing more information at once; cut, copy, and paste of training examples and gesture classes; and the ability to save and reuse individual classes. The remainder of this section describes *gdt* in more detail.

gdt allows designers to enter and edit training examples, train the recognizer, and recognize individual examples. Figure 1 shows the *gdt* main window with a gesture set loaded. Each gesture class is shown as a name and an exemplar gesture (currently, the first training example for the class). In this example, the only fully visible classes are select, delete, undo, redo, and bigger font (in the top part of the window). The user has drawn a gesture to be recognized, which is shown in the white region at the bottom of the window. The recognition result can be seen across the middle of the window. The example shows that the gesture was correctly recognized as delete and gives some additional information about how well it was recognized. From the main window the user can, among other things, see all gesture classes in the set, open a gesture class to examine its training examples, call up visualizations of the gesture set, and enter gestures to be recognized.

gdt allows the designer to examine training examples for a class and enter new ones. Figure 2 shows some training examples for delete. The individual examples can be deleted, cut, copied, and pasted. New examples can be added by drawing them in the white area at the bottom.

Unlike *Agate*, *gdt* also provides visualizations to aid the designer in discovering and fixing computer recognition problems. One visualization, called the distance matrix (shown in Figure 3), highlights gesture classes that the recognizer has difficulty differentiating. Distances are a weighted Euclidean distance (specifically, Mahalanobis distance [7]) in the recognizer's feature space. This visualization is helpful because gesture classes that are close to each other in feature space are more likely to be confused with one another by the algorithm. The slider on the right side of the window may be used to grey out distances above a threshold so that the user can more easily find similar gestures. For example, one can see in the figure that the most similar classes are smaller font and undo because the distance between them is the smallest.

Another visualization provided by *gdt* is the classification matrix (shown in Figure 4), which tallies how the training examples are recognized. The row and column names list gesture classes. Each cell contains the percentage of training

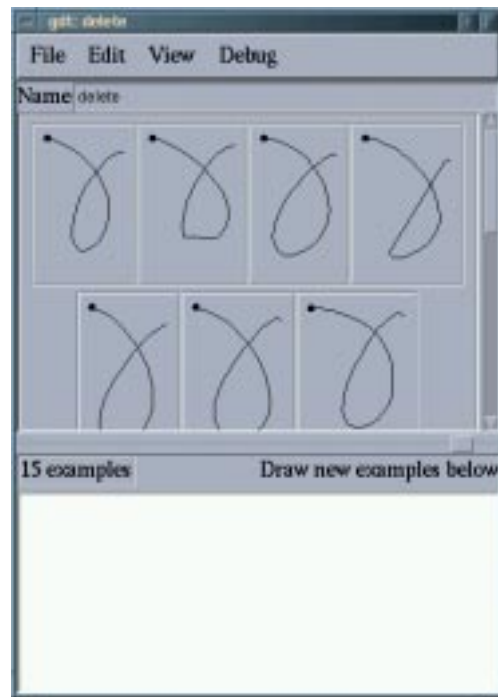


Figure 2: *gdt* class window.

examples of the class specified by the row name that were recognized as the class specified by the column name. To make interesting entries stand out, misrecognized entries are colored red (dark shaded in this paper) and diagonal entries that are less than 100% are colored yellow (light shaded in this paper). In this example, six percent of select examples were misrecognized as delete and six percent of smaller font examples were misrecognized as undo. A misrecognized training example is either a poorly entered example or a sign that the gesture class to which it belongs is too similar to another class.

The third visualization provided by *gdt* is a graphical display of raw feature values for all training examples. We thought that such a display might help designers to determine why classes were similar and how to change them to make them less similar. Unfortunately, the visualization was too complicated and so it was not used in the experiment (described below).

Distance Matrix						
Table						
Class	select	delete	undo	redo	bigger font s	Threshold
select	0.0	11	13	13	16	
delete	11	0.0	16	11	14	
undo	13	16	0.0	7.4	18	
redo	13	11	7.4	0.0	17	
bigger f...	16	14	18	17	0.0	
smaller ...	13	14	6.5	7.4	17	
bring to...	15	17	9.3	12	19	
shuffle ...	13	13	16	13	18	
shuffle ...	15	18	12	14	19	
zoom in	14	10	19	18	12	
zoom out	16	12	18	14	19	
send to ...	11	11	14	11	17	
rotate c...	12	9.9	15	14	9.9	
rotate c...	14	16	15	14	17	
select all	8.2	14	13	13	20	

Figure 3: *gdt* distance matrix. Larger distances are grayed out.

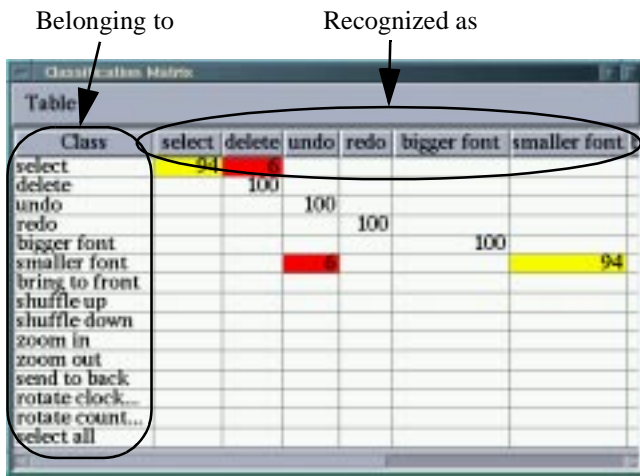


Figure 4: *gdt* classification matrix.

In addition, *gdt* has a test procedure in which it asks the user to draw gestures from the current set. The tool tallies how well the test gestures are recognized. In a single test run, *gdt* displays each class name and exemplar five times in a random order and asks the user to draw it each time. After the test, *gdt* displays the overall recognition rate, or test score, and shows how the entered gestures were recognized in a format identical to the classification matrix (Figure 4). Also, after the test *gdt* allows the user to examine the gestures drawn during the test (this feature was not available during the experiment, which is described below).

gdt Limitations

There is a tension in designing a gesture design tool in terms of the extent to which it should be recognizer-independent versus recognizer-dependent. The benefits of recognizer-independence are obvious: the tool can be run with any recognition technology at design time and a different technology at run-time. On the other hand, by using recognizer-dependent features, the tool may be able to offer better advice, but at the cost of non-portability. In our design of *gdt*, we decided to include some of both types of features.

The Rubine algorithm is good for prototyping gesture sets, but designers may want to use different recognition technology in a final product. Some features of *gdt* will apply to many types of recognizers, while others are specific to the Rubine algorithm. Recognizer-independent features are: entry and editing of training examples (Figure 2), the classification matrix visualization (Figure 4), and the test mode. Conversely, the distance matrix (Figure 3) and feature visualizations may not apply to other recognizers. Recognizer-independent features we plan to add to *gdt* are described below in the Discussion section.

gdt was implemented entirely in Java. During the experiment, it was run on a 200MHz Pentium Pro with 64MB RAM using the Visual Café Java run-time environment. Of all the opinions we solicited about the system in the post-experiment questionnaire, system speed was ranked the lowest. We suspect the system did not have enough main memory and so *gdt* often stalled while swapping.¹

1. Interestingly, people who had used PDAs were more forgiving of system sluggishness and unreliability than those who had not.

EXPERIMENT

We ran an experiment to evaluate *gdt* and, more importantly, to gain insight into the process of designing gestures. We formulated several hypotheses:

- Participants could use *gdt* to improve their gesture sets.
- The visualizations *gdt* provided would aid designers.
- The performance of participants with art or design background would differ from that of participants with technical or computer science backgrounds.
- PDA users and non-PDA users would perform differently.

We recruited two types of participants: technical (mostly computer science undergraduates) and artistic (architects and artists). We paid each participant \$25 for participating. We ran ten pilot participants and ten in the experiment proper.

The experiment was performed in a small lab where participants could work without distraction. A video camera recorded participant utterances and facial expressions. The computer screen was videotaped using a scan converter. All computer interaction except for the post-experimental questionnaire was done on a Wacom display tablet using a stylus. The experimenter was present in the room during the experiment, observing the participant and taking notes. The experimenter was allowed to answer questions if the answer was contained in the materials given to the participant.

Experimental Procedure

This section describes the different steps of the experiment. The total time for each participant ranged from 1.5 to 2.5 hours. All participants were asked to sign a consent form based on the standard consent form provided by the Berkeley campus Committee for the Protection of Human Subjects.

Demonstration

Participants were shown a demonstration of *gdt*. They were shown how to enter gestures to be recognized, how to examine gesture classes and training examples, and the distance matrix and classification matrix visualizations.

Tutorial

Next, participants were given a printed tutorial about *gdt* that gave a simple description of the Rubine algorithm and showed how to perform the tasks necessary to do the experiment. The tutorial also described the distance and classification matrix visualizations.

Practice Task

To allow participants to familiarize themselves with *gdt*, we asked them to perform a practice task. This task was their first opportunity to actually use the tool. In this task, they were given a gesture set containing one gesture class and asked to add two new gesture classes of fifteen examples each with specified names and shapes. After adding them, participants were to draw each of the two new gestures five times to be recognized.

Baseline Task

We wanted to compare recognition rates from the experimental task across participants, but recognition rates will vary across participants (e.g., due to being neat or sloppy). To account for this variance, we measured the recognition rate of a standard gesture set for each participant. The gesture set used was the same one used for

the experimental task, which had fifteen gesture classes, each of which already had fifteen training examples.

Since users were not familiar with the *gdt* test procedure, we thought a single test would be unreliable. We asked participants to perform the test twice with the experimental set.

A drawback of the Rubine algorithm is that gesture drawn by one person (such as the participant) may not be recognized well if the gesture set was trained by a different person (such as the experimenter). We wanted to know whether participants could improve their recognition rate by adding their own examples to the preexisting gesture set. So, next participants added five examples of their own to each gesture class in the initial experimental set and did another test. The resulting gesture set we term the *baseline gesture set*. We recorded the recognition rate for this third baseline test and used it as the *target recognition rate* for the experimental task.

Experimental Task

The experimental task was to invent gestures for ten new operations and add these gestures to the baseline gesture set. Participants were told to design gestures that were recognizable by the computer and were easy for people to learn and remember. As an incentive, we offered \$100 to the creator of the best gesture set, as judged by the experimenters. Most participants followed this general strategy:

1. Think of one or more new gesture classes.
2. Enter training examples for the class.
3. Informally enter examples of the new class(es) to see if they are recognized.
4. Look at the new class(es) statistics in the classification matrix and/or distance matrix.
5. Possibly modify the class(es) just entered.
6. Repeat until all new classes are entered.

After entering all the classes, each participant ran a test in *gdt*. If a participant did not reach the target recognition rate, the experimenter asked the participant to try to improve the recognition rate.

Participants were asked to continue to work until they had either achieved a recognition rate equal to or better than the target rate or until they had spent ninety minutes on the experimental task. Participants were not told that there was a time limit until five minutes before the end, at which time they were asked to finish up and take the recognition test (again).

Post-experiment Questionnaire

After the experiment, participants were led to a second computer (to avoid negative effects suggested by [18]) where they used a web browser (with traditional WIMP interface) to fill out a questionnaire. The questionnaire asked for three basic types of information: 1. opinions about various aspects of *gdt* and the experiment, 2. PDA usage, and 3. general demographics.

RESULTS AND ANALYSIS

This section describes and analyzes the results of the experiment. First, we will discuss evidence for or against our proposed hypotheses. Then we will discuss general results related to the gesture design process.

Hypotheses

One of the most important questions we wanted to answer was whether participants could use *gdt* to improve their gesture sets. We measured improvement as the difference between the best recognition rate achieved during the experimental task and the recognition rate of the first test done during the experimental task, called the *initial test*. We found that on average participants improved their gesture sets by 4.0% (from 91.4% to 95.4%) and that this difference was statistically significant ($p < 0.006$, 2 tailed t-test).

We were also interested in whether the distance matrix, classification matrix, or test result visualizations would be helpful in designing gesture sets. Six participants used the classification matrix. Eight used the distance matrix. Seven looked at the test results. For each visualization, including the test results, we compared the performance of those who used them and those who did not. Surprisingly, usage of none of the three visualizations had a significant effect.

We hypothesized that artistic and technical participants might perform differently in this experiment. However, no significant difference was detected in the performance of these two groups.

Among other things, we asked participants on the post-experiment questionnaire to rate their user interface design experience and if and for how long they had used a PDA. As a metric of gesture set goodness, we measured the average pairwise distance between all gesture classes in the final gesture set of each participant (because classes that are farther apart are less likely to be misrecognized). We found that average pairwise distance correlated both with UI design experience and with length of PDA usage (correlation coefficients 0.67 and 0.97). In other words, participants who had designed UIs or used PDAs designed objectively better gesture sets.

Gesture Design Process

Although participants were able to use *gdt* to improve their gesture sets, it was not an easy task. This section discusses problems participants encountered and strategies they used in designing gestures.

1. Finding and fixing recognition problems. Participants had difficulty finding and fixing recognition problems. On the post-experiment questionnaire, using a scale of 1 (difficult) to 9 (easy), finding recognition problems was ranked 5.8 and fixing them was ranked 4.6. The average best recognition rate was 95.4%, which we do not believe is good enough for commercial applications.¹ Much of this was likely due to a lack of understanding of the recognizer, which many participants expressed verbally.

2. Adding new classes. We also found that adding new gesture classes caused a statistically significant drop of 2.4% in the recognition rate of the preexisting gestures ($p < 0.041$, 2 tailed t-test). Most participants did not seem aware that this problem might occur. Many participants thought a low recognition rate was a problem with how they drew the gestures during the test.

3. New similar class. One way new classes were observed to cause a problem is by being too similar to one or more existing classes. Sometimes the participant noticed this problem by informally testing the recognition (i.e., just drawing in the main window and seeing how it was

1. [1] reports that users found a 98% recognition rate inadequate.

recognized) or with the distance matrix. However, not all participants watched for this problem.

4. Outlier feature values. Another way new classes were seen to cause recognition problems is by having feature values that were significantly different than the values of many old classes. The outlier values caused the values for old classes, which were close together by comparison, to clump together. Unfortunately, these features were important for disambiguating the old classes, and so by adding the new classes the old ones became harder to correctly recognize.

5. Drawing gestures backwards. Since several features used in the Rubine recognizer depend on the starting point, it is important for users to be consistent about the placement of the starting point and the initial direction. Unfortunately, some participants drew test gestures backwards (i.e., starting at the end and going to the beginning), either because they had not learned the gesture well enough or because the start and end point of the gesture were too close together, and it was unclear which direction was the correct one.

6. Radical changes. Participants also varied by what strategy they used to try to solve recognition problems. When they discovered that two classes were confused with one another, some participants made a small change in one of the two classes. Other participants made a dramatic change to one of the problem classes. One of the metrics for success in the experimental task was how much the recognition rate improved from the beginning of the experimental task to the best recognition rate achieved during the experimental task. The improvement in recognition rate of participants who made radical changes was lower than the improvement of those who did not make radical changes (1.4% vs. 6.6%), and this difference was significant ($p < 0.006$, 2 tailed t-test).

7. Over-testing. When faced with a test score lower than the target, some participants elected to take the test again, because they thought they had been sloppy when entering the gesture. They thought if they were neater they would do better. Sometimes this strategy succeeded and other times it did not.

8. Limited test support. Participants in the experiment relied heavily on the test procedure. At present, the tool has only rudimentary support for testing how well a gesture set is recognized. The only test results available were the count of how many gestures of each class were recognized and the overall recognition rate.

9. Multiple gestures for one operation. Several participants wanted to experiment with different gestures for the same operation. For example, a participant wanted to experiment with several gestures for the “pen” operation and so made three classes with three different gestures: pen, pen 2, and pen 3. Unfortunately, the alternative classes affect the recognition of one another, which is undesirable since the final set will contain at most one of them.

We learned a great deal about the gesture design process from this experiment. Based on its results, we think that a tool like *gdt* is valuable, but it falls short of an ideal gesture design tool in many ways. The next section discusses implications of the experiment for building a better gesture design tool.

DISCUSSION

This section discusses results from the experiment and what features a better gesture design tool might have. The first subsection discusses our experimental hypotheses. The second discusses implications of our experiment for building a gesture design tool to better support the task. The third describes other features that may lead to a better gesture design tool.

Hypotheses

This subsection discusses why the experimental hypotheses were validated or refuted.

Participants could use *gdt* to improve their gesture sets. The confirmation of this hypothesis did not surprise us, but we were surprised that participants were only able to reach a 95.4% recognition rate. We believe that this low performance is because the participants did not understand how the recognizer worked and the tool was not very sophisticated.

What we did not expect was that none of the visualizations provided by *gdt* would have a statistically significant effect on the performance of participants. We anticipated that the distance matrix, in particular, would be useful to participants in finding what we expected to be the most common cause of recognition problems: two (or more) gesture classes too close together. We believe that it was not useful because it was too abstract and because users did not understand how the recognizer worked well enough. Higher level information is probably more appropriate to provide to users. That is, rather than show the user n^2 distance numbers whose units are completely abstract, tell the user, “Class A and class B are too similar.”

We also expected that the classification matrix would be useful because we expected some training examples in every set to be misrecognized. In fact, often no training examples were misrecognized.

Although a fair number of participants consulted the distance matrix and classification matrix, the majority focused much more on the test results and seemed to base many decisions about where recognition problems were on it. We believe this did not improve their performance because the test results are determined not only by the quality of the gesture set, but by the goodness of the test examples themselves. From looking at the test results, participants could not tell what the reason for an unsatisfactory test was.

It is also somewhat surprising that we detected no significant difference in the performance of participants with an art/design background and those with a technical/computer science background. We believe that the creativity and design experience of the first group was balanced by the comprehension of the recognition technology of the second group. We would like to make it possible for artists and designers to design gesture sets, in the same way that modern tools allow them to design UIs. Unfortunately, the current state of gesture design tools (e.g., *gdt*) is such that technical expertise and background are significant.

As expected, we did find that performance correlated with participants’ self-ranked UI design experience. We believe this is due to experience with the design, prototype, evaluate cycle. Also as expected, PDA usage correlated with performance, which is likely due to familiarity with gestures.

Gesture Design Tool Implications

Both the experiment and our own experiences with gesture design and *gdt* have given us ideas about what features a gesture design tool should have, which are discussed in this subsection. We describe lessons we learned from the experiment and what they imply for future gesture design tools. Next, we discuss the features that arose from our own experiences.

Experimental Lessons

The single biggest lesson we drew from the experiment was that users do not understand recognition, so the tool must take an active role in warning them about recognition problems and in guiding them to solutions to those problems. First, we list some of the ways a gesture design tool could be more *active*. Following are some other features that address problems participants encountered in the experiment.

Active Feedback:

- **Warn about radical changes.** The most statistically significant result from the experiment is that it is detrimental to make radical changes to a gesture class. The tool should therefore monitor changes to gesture classes, and when a class is modified, measure how far the modified class is from the old class. If the new class is farther from the old class than some threshold, it should warn the designer that radical changes are often harmful. This addresses problem 6.
- **Warn about classes that are too similar.** The tool should examine each new class to see if it is too similar to an existing one. If so, the designer should be warned and told which feature to change (and in what direction) to make the new class less similar. This addresses problems 1, 2, and 3.
- **Warn about classes with outlying feature values.** The tool should examine each new class to see if the values of one or more features are outliers with respect to other classes in the set. If so, the user should be warned and told which feature(s) to change and in which direction. This addresses problems 1, 2, and 4.
- **Warn about drawing backwards.** The tool should reverse test gestures and recognize them, too. If the backwards gesture was correctly recognized and the forward (i.e., normal) one was not, it should warn the user at the end of the test that some gestures may have been drawn backwards and suggest that those classes be made direction-independent. This addresses problem 5.
- **Warn about over-testing.** If a large number of tests are performed successively without changes to the gesture set, the tool should suggest that the designer modify the set to try to get better recognition instead of retesting. This addresses problem 7.

As well as the active feedback mechanisms, the experiment also suggested a few other capabilities that would enhance a gesture design tool.

One such capability is better support for testing gesture sets. The testing feature was very popular in the experiment. A gesture design tool should make it easy to create a set of test

gestures that are to be recognized against another gesture set. These test gesture sets should be as easy to edit, load, and save as the training gesture sets. This enhancement addresses problem 8.

Another desirable capability is the ability to be enable or disable individual gesture classes without deleting and re-entering them. In particular, it would greatly ameliorate problems 8 and 9.

In addition, the experiment suggested the idea of allowing gesture examples and classes to be dragged and dropped between classes and sets, respectively. This capability would solve problem 9 (via drag-and-drop between the main set and a “scratch” set).

We also learned from the experiment that the lack of understanding how the recognizer worked greatly hindered participants both in discovering and in fixing recognition problems. We want designers to be able to make gesture sets without being experts on gesture set design. Unfortunately, this knowledge is required to successfully use current tools.

One feature that would aid designers in understanding the recognition process is to graphically explain the features. For example, superimposing a graphic representation of each feature on top of a gesture would help designers understand what the features are and thus how to change one or more classes to make them less similar. For example, Figure 5 shows a graphic representation of the initial angle feature. The lighter dotted lines are guidelines and the darker dotted lines are the value of the feature. (They would be drawn in different colors on-screen.)

Additional Capabilities

Besides the results that arose directly out of the experiment, we have ideas from our own experience about features that would be useful in a gesture design tool. We describe those features below.

Another capability we think would be useful is assistance in making a class size- or rotation-independent. If the user indicated that a class should be size-independent, for example, the tool could generate training examples of different sizes by transforming existing examples. This feature could be extended to other types of independence besides size and rotation, such as drawing direction.

A weakness of the Rubine algorithm is that it works best if the values of each feature within each gesture class are normally distributed. That is, for a given class and feature,

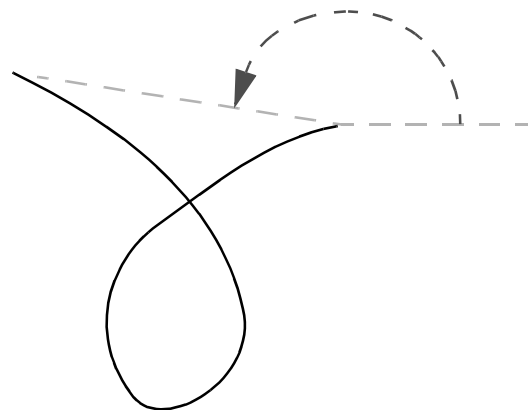


Figure 5: Initial angle visualization.

the distribution of a feature's value across training examples in that class should be approximately normal (ideally, the degenerate case of all values being identical or nearly so). If a feature that is important for disambiguating two classes is not normally distributed in some third class, the recognition of the first two classes might suffer. For example, one might have a gesture set whose classes are disambiguated based on size. If there is a square class is added that is intended to match squares of two very different sizes, then some of its examples will be large and others small, which will make its size feature bimodally distributed. This non-normal distribution may hinder the other classes from being disambiguated based on size.

This problem could be solved by breaking the square class into a "big square" class and a "small square" class. The tool could notice the need for such a split and do it automatically (or after designer confirmation). At application run-time, the recognition system could combine the split classes into one so they would appear to be one class to the application. Such a split may be necessary after automatically generating examples to make a class size- or rotation-independent, as discussed above.

Other recognition systems, especially voice recognition systems, show the n-best recognized results instead of just the best recognized one [14]. This feature would be useful for a gesture design tool as well.

All of the preceding features deal with computer recognition. We also believe that the ability of the user to learn and remember gestures is very important. We are currently researching how to estimate these for gesture sets. In the immediate future, we are planning experiments to investigate how people judge gesture similarity.

CONCLUSIONS

It is difficult to design gesture sets that are well-recognized. There are many pitfalls of which to be wary, and many of them are all but invisible to those unfamiliar with recognition technology. It was very difficult for the participants in our experiment to attain a good recognition rate for their gesture set, and we believe this was due in large part to difficulty in understanding the recognizer.

Pen-based user interfaces are suitable for a wide variety of applications, and those applications will all need different gesture sets. It is important for the usability of pen-based UIs that designers are able to design gesture sets that are easy for the computer to recognize and for humans to learn and remember. To perform this difficult task, designers will require significantly better gesture design tools than are currently available.

The contribution of this work is as a first step in exploring the process of gesture set design. We have shown certain strategies for gesture design to be useful and others to be detrimental. We have provided directions to explore for future gesture design tools.

REFERENCES

- [1] Briggs, R., Dennis, A., Beck, B., and Nunamaker, Jr., J. Whither the pen-based interface? *Journal of Management Information Systems*, 9(3):71–90, 1992-1993.
- [2] Buxton, W. *There's More to Interaction Than Meets the Eye: Some Issues in Manual Input*, pages 319–337. Lawrence Erlbaum Associates, Hillsdale, N.J., 1986.
- [3] Chatty, S. and Lecoanet, P. Pen computing for air traffic control. In *Human Factors in Computing Systems (SIGCHI Proceedings)*, pages 87–94. ACM, Addison-Wesley, Apr 1996.
- [4] Frankish, C., Hull, R., and Morgan, P. Recognition accuracy and user acceptance of pen interfaces. In *Human Factors in Computing Systems (SIGCHI Proceedings)*, pages 503–510. ACM, Addison-Wesley, Apr 1995.
- [5] Hanne, K.-H. and Bullinger, H.-J. *Multimedia Interface Design*, chapter 8, pages 127–138. ACM Press, 1992.
- [6] Hickman, A. Power in your pocket. *PC Magazine Online*, May 1998. <http://www.zdnet.com/pcmag/news/trends/t980514a.htm>.
- [7] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*, volume 3 of *Oxford Statistical Science Series*. Oxford University Press, New York, NY, 1988.
- [8] Landay, J. and Myers, B. Interactive sketching for the early stages of user interface design. In *Human Factors in Computing Systems (SIGCHI Proceedings)*, pages 43–50. ACM, Addison-Wesley, Apr 1995.
- [9] Landay, J. A. and Myers, B. A. Extending an existing user interface toolkit to support gesture recognition. In Ashlund, S. et al., editors, *Adjunct Proceedings of INTERCHI '93: Human Factors in Computing Systems*, pages 24–29. ACM, Addison Wesley, Apr 1993.
- [10] Lipscomb, J. A trainable gesture recognizer. *Pattern Recognition*, 24(9):895–907, Sep 1991.
- [11] Long, Jr., A. C., Landay, J. A., and Rowe, L. A. PDA and gesture use in practice: Insights for designers of pen-based user interfaces. Technical Report CSD-97-976, U.C. Berkeley, 1997. Available at <http://bmrc.berkeley.edu/papers/1997/142/142.html>.
- [12] Meyer, A. Pen computing. *SIGCHI Bulletin*, 27(3):46–90, Jul 1995.
- [13] Morrel-Samuels, P. Clarifying the distinction between lexical and gestural commands. *International Journal of Man-Machine Studies*, 32:581–590, 1990.
- [14] Murray, A., Frankish, C., and Jones, D. *Interactive Speech Technology: Human factors issues in the application of speech input/output to computers*, chapter 15, pages 137–144. Taylor & Francis, 1993.
- [15] Myers, B. A. et al. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11), Nov 1990.
- [16] Myers, B. A. et al. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, Jun 1997.
- [17] Pier, K. and Landay, J. A. Issues for location-independent interfaces. Technical Report ISTL92-4, Xerox Palo Alto Research Center, Dec 1992.
- [18] Reeves, B. and Nass, C. *The media equation: how people treat computers, television, and new media like real people and places*. Center for the Study of Language and Information; Cambridge University Press, Stanford, Calif.: Cambridge [England]; New York, 1996.
- [19] Rubine, D. Specifying gestures by example. In *Computer Graphics*, pages 329–337. ACM SIGGRAPH, Addison Wesley, Jul 1991.
- [20] Tapia, M. and Kurtenbach, G. Some design refinements and principles on the appearance and behavior of marking menus. In *Proceedings of the ACM Symposium on User Interface and Software Technology (UIST)*, pages 189–195. ACM, Nov 1995.
- [21] Wolf, C., Rhyne, J., and Ellozy, H. The paper-like interface. In Salvendy, G. and Smith, M., editors, *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*, volume 12B of *Advances in Human Factors/Ergonomics*, pages 494–501. Elsevier, Sep 1989.