CMU-ITC-91 100

TO BE PUBLISHED IN SUMMER 91 USENIX TECHNICAL PROCEEDINGS

Integrating Gesture Recognition and Direct Manipulation

Dean Rubine
Information Technology Center
Carnegie Mellon University
Dean.Rubine@cs.cmu.edu

Abstract

A gesture-based interface is one in which the user specifies commands by simple drawings, typically made with a mouse or stylus. A single intuitive gesture can simultaneously specify objects, an operation, and additional parameters, making gestures more powerful than the "clicks" and "drags" of traditional direct-manipulation interfaces. However, a problem with most gesture-based systems is that an entire gesture must be entered and the interaction completed before the system responds. Such a system makes it awkward to use gestures for operations that require continuous feedback.

GRANDMA, a tool for building gesture-based applications, overcomes this shortcoming through two methods of integrating gesturing and direct manipulation. First, GRANDMA allows views that respond to gesture and views that respond to clicks and drags (e.g. widgets) to coexist in the same interface. More interestingly, GRANDMA supports a new two-phase interaction technique, in which a gesture collection phase is immediately followed by a manipulation phase. In its simplest form, the phase transition is indicated by keeping the mouse still while continuing to hold the button down. Alternatively, the phase transition can occur once enough of the gesture has been seen to recognize it unambiguously. The latter method, called *eager recognition*, results in a smooth and natural interaction. In addition to describing how GRANDMA supports the integration of gesture and direct manipulation, this paper presents an algorithm for creating eager recognizers from example gestures.

1. Introduction

Gestures are hand-drawn strokes that are used to command computers. The canonical example is a proofreader's mark used for editing text [2, 4] shown in figure 1.

Gesture-based systems are similar to handwriting systems [26], in that both rely on pattern recognition for interpreting drawn symbols. Unlike a handwritten character, a single gesture indicates an operation, its operands, and additional parameters. For example, the gesture in figure 1 might be translated "move these characters to this location," with the referents clearly indicated by the gesture.

Generally, the end of the gesture must be indicated before the gesture is classified and command execution commences. In almost every gesture-based system to date, the gesture ends when the user

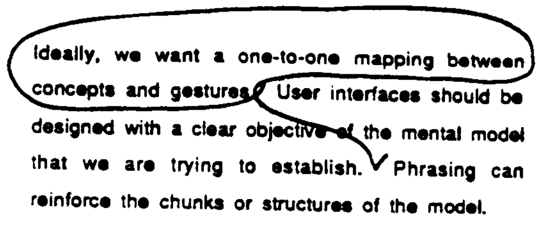

Ideally, we want a one-to-one mapping between concepts and gestures. User interfaces should be designed with a clear objective of the mental model that we are trying to establish. Phrasing can reinforce the chunks or structures of the model.

Figure 1: A move text gesture (from Buxton [2])

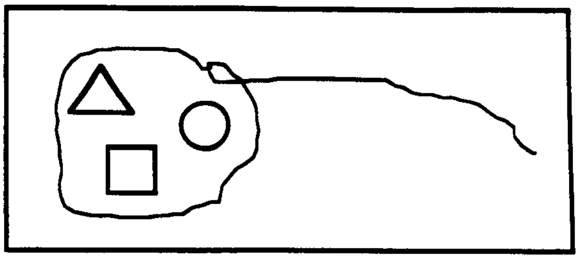


Figure 2: Moving objects in GEdit (from Kurtenbach and Buxton [14])

relaxes physically, e.g. by releasing the mouse button or lifting the stylus from the tablet. (Examples include IBM's Paper-Like Interface [11, 21], Coleman's proofreader's marks [4], MCC's HITS [10], and Buxton's char-rec note input tool [3].) The physical tension and relaxation of making a gesture correlates pleasantly with the mental tension and relaxation involved in performing a primitive task in the application [2]. However, since command execution begins only after the interaction ends, there is no opportunity for semantic feedback from the application *during* the interaction.

Referring to the move text gesture, Kurtenbach and Buxton [14] claim that application feedback is unnecessary during the interaction. They do admit that in the case of a drawing editor (figure 2) the lack of feedback hampers precise positioning. In the text case, they are probably correct that actually moving the text during the interaction is undesirable. What is desirable, I claim, is feedback in the form of a text cursor, dragged by the mouse but snapping [1] to legal destinations for the text. Such a cursor confirms that the gesture was indeed recognized correctly, and allows the user to be sure of the text's destination before committing to the operation by releasing the mouse button.

Direct manipulation [25] is an accepted paradigm for providing application feedback during mouse interactions. The goal of the present work is to combine gesturing with direct manipulation. One way this might be done is via modes: after a gesture is recognized, the following mouse interaction is interpreted as a manipulation rather than another gesture. A better way[1] is to interpret mouse interactions as gestures when they begin on certain objects, and otherwise as direct manipulation. This is done in GEdit [14]: a mouse press on a shape causes it to be dragged, while a mouse press over the background window is interpreted as gesture. An alternative would be to use one mouse button for gesturing and another for direct manipulation. While these techniques work, they may result in a primitive application task in the user's mental model ("create a rectangle of a given size and position") being serialized into multiple interactions ("create a rectangle" then "manipulate its size", then "manipulate its position"). This serialization undoes the correlation between physical and mental tension.

This paper advocates integrating gesture and direct manipulation in a single, two-phase interaction. The intent is to retain the intuitiveness of each interaction style, while combining their power.

[1]Modes, the "global variable" of user interfaces, are generally frowned upon.

The first phase of the interaction is *collection*, during which the points of the gesture are collected. Then the end of the gesture is indicated, the gesture classified, and the *manipulation* phase is entered. The classification of the gesture determines the operation to be performed. The operand and some parameters may also be determined at classification time. During the manipulation phase, additional parameters may be determined interactively, in the presence of application feedback.

GRANDMA (Gesture Recognizers Automated in a Novel Direct Manipulation Architecture) is a system I built for creating gesture-based applications [22, 23]. Written in Objective-C [5], GRANDMA runs on a DEC MicroVax II under MACH [27] and X10 [24]. In GRANDMA, the time of the transition from collection to manipulation is determined in one of three ways:

1. when the mouse button is released (in which case the manipulation phase is omitted),
2. by a timeout indicating that the user has not moved the mouse for 200 milliseconds, or
3. when enough of the gesture has been seen to unambiguously classify it.

The last alternative, termed *eager recognition*, results in smooth and graceful interactions. Citing my dissertation, Henry *et. al.* [9] describes hand-coded eager recognizers for a particular application.

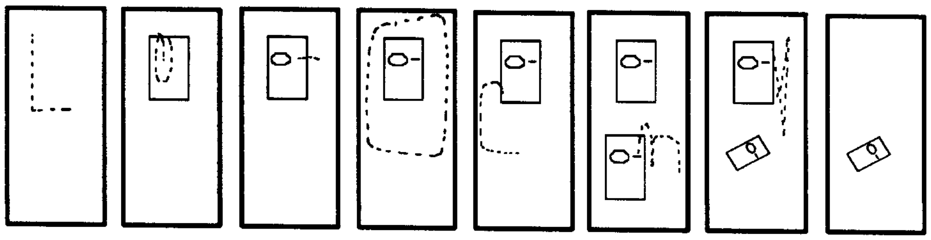
The present work focuses on *trainable* recognition, in which gesture recognizers, both eager and not, are built from example gestures. The discussion begins with GDP, a gesture-based drawing program which combines gesture and direct manipulation. The next section describes an algorithm for constructing eager recognizers from training examples. Performance measurements of the algorithm are covered in the following section. A concluding section summarizes the work and presents some future directions.

2. GDP: A gesture-based drawing program

GDP is a gesture-based drawing program based on (the non-gesture-based program) DP [7]. GDP is capable of producing drawings made with lines, rectangles, ellipses, and text. This section sketches GDP's operation from the user's perspective.

Figure 3 shows the effect of a sequence of GDP gestures. (Eager recognition has been turned off, so full gestures are shown.) The user presses the mouse button and enters the rectangle gesture and then stops, holding the button down. The gesture is recognized, and a rectangle is created with one endpoint at the start of the gesture, another endpoint at the current mouse location. The latter endpoint can then be dragged by the mouse: this enables the rectangle's size to be determined interactively.

The line and ellipse gestures work similarly. The group gesture generates a composite object out of the enclosed objects; additional objects may be added to the group by touching them during the manipulation phase. The copy gesture replicates an object, allowing it to be positioned during manipulation. The move gesture, not shown, works analogously. The initial point of the rotate-scale gesture determines the center of rotation; the final point (i.e. the mouse position when the gesture is recognized) determines a point (not necessarily on the object) that will be dragged around to interactively manipulate the object's size and orientation. The delete gesture deletes the object



Gesture:	Rectangle	Ellipse	Line	Group	Copy	Rotate-scale	Delete
Determined at recognition time:	Corner 1	Center	Endpt 1	Enclosed objects to group	Object to copy	Object Center of rotation Drag point	Object to delete
Determined by manipulation:	Corner 2	Size Eccentricity	Endpt 2	Touch other objects to add	Location of copy	Size Orientation	Touch additional objects to delete

Figure 3: Some GDP gestures and parameters (adapted from [22])
Gestures are shown with dotted lines. The effect of each gesture is shown in the panel to its right. Under each panel are listed those parameters that are determined at the time the gesture is recognized, and those that may be manipulated in the presence of application feedback.

at the gesture start. During the manipulation phase, any additional objects touched by the mouse cursor are also deleted.

In a modified version of GDP, the initial angle of the rectangle gesture determines the orientation of the rectangle (with respect to the horizontal). For this to work, the rectangle gesture was trained in multiple orientations. In the version shown here, only the "L" orientation was used in training. Also in the modified version, the length of the line gesture determines the thickness of the line. To keep things simple, the modified version was not used either to generate the figure or in the remainder of this paper. It is mentioned here to illustrate how gestural attributes may be mapped to application parameters.

Not shown is an edit gesture (which looks like "Σ"). This gesture brings up control points on an object. The control points do not themselves respond to gesture, but can be dragged around directly (scaling the object accordingly). This illustrates that systems built with GRANDMA can combine gesture and direct manipulation in the same interface.

Note that each gesture used in GDP is a single stroke. This is a limitation of GRANDMA's gesture recognition algorithm. Supporting only single stroke gestures reinforces the correlation between physical and mental tension, allows the use of short timeouts, and simplifies both non-eager gesture recognition and eager recognition. The major drawback is that many common marks (e.g. "X" and "—>") cannot be used as gestures by GRANDMA. A number of techniques exist for adapting single-stroke recognizers to multiple stroke recognition [8, 15], so perhaps GRANDMA's recognizer will extended this way in the future.

3. Support for gesture and direct manipulation in GRANDMA

The following summary of GRANDMA's architecture is intended to be sufficient for explaining how gesture and direct manipulation are integrated. Much detail has been glossed over; the interested reader is referred to [23] for the full story.

GRANDMA is a Model/View/Controller-like system [13]. In GRANDMA, models are application objects, views are objects responsible for displaying models, and event handlers deal with input directed at views. GRANDMA generalizes MVC by allowing a list of event handlers (rather than a single controller) to be associated with a view. Event handlers may be associated with view classes as well, and are inherited. Associating a handler with an entire class greatly improves efficiency, as a single handler is automatically shared by many objects.

3.1. Gesture and direct-manipulation in the same interface

Each class of event handler implements a particular kind of interaction technique. For example, the drag handler handles drag interactions, enabling entire objects (or parts of objects) to be dragged by the mouse. A gesture handler contains a classifier for a set of gestures, and handles both the collection and manipulation phases of the two-phase interaction. Thus, it is straightforward in GRANDMA to have some views respond to gesture while other respond to direct manipulation: simply associate gesture handlers with the former's class and drag handlers (or other direct-manipulation style handlers) with the latter's. Similarly, views of different classes may respond to different sets of gestures by associating each view class with a different gesture handler.

A single view (or view class) may respond to both gesture and direct manipulation (say, via different mouse buttons) by associating multiple handlers with the view. Each handler has a predicate that it uses to decide which events it will handle. It is simple to arrange for a handler to deal only with particular types of events (e.g. mouse down, mouse moved, mouse up) or only with events generated by a particular mouse button. The handlers associated with a particular view are queried in order whenever input is initiated at the view; any input ignored by one handler is propagated to the next.

3.2. Gesture and direct-manipulation in a two-phase interaction

As mentioned above, the gesture handler implements the two-phase interaction technique. Each instance of a gesture handler recognizes its own set of gestures, and can have its own semantics associated with each gesture. The handler is responsible for collecting and inking the gesture, determining when the phase transition occurs, classifying the gesture, and executing the gesture's semantics.

The gesture semantics consist of three expressions: `recog`, evaluated when the gesture is recognized (i.e. at the phase transition), `manip`, evaluated for each mouse point that arrives during the manipulation phase, and `done`, evaluated when the interaction ends (i.e. the mouse button is released). For example, the semantics of GDP's rectangle gesture are:

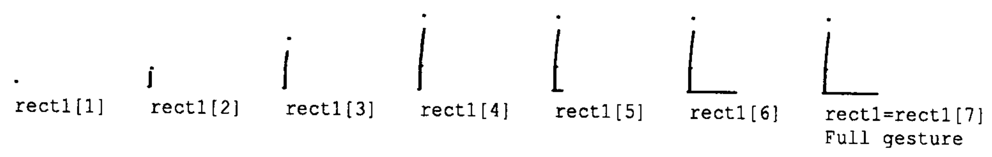


Figure 4: A full rectangle gesture and all its subgestures.

```
recog = [[view createRect]
             setEndpoint:0 x:<startX> y:<startY>];
manip = [recog setEndpoint:1 x:<currentX> y:<currentY>];
done = nil;
```

The syntax is that of Objective-C messages. The expressions are evaluated by a simple Objective-C message interpreter built into GRANDMA. During evaluation, the values of many gestural attributes are lazily bound to variables in the environment, and are thus available for use as parameters in application routines. In the above example, `view` refers to the object at which the gesture is directed, in this case the window in which GDP runs. This view is sent to message `createRect`, which returns a newly created rectangle. The attributes `<startX>` and `<startY>` refer to the initial point of the gesture; the newly created rectangle is sent a message making this point one corner of the rectangle. The rectangle is conveniently stored in the variable `recog` for use in the `manip` semantics. In response to each mouse point during the manipulation phase, the `manip` semantics makes the other corner of the rectangle `<currentX>`, `<currentY>`, thus implementing the interactive "rubberbanding" of the rectangle. The `done` expression is null in this case, as the processing was done by `manip`. There are many other attributes available to the semantics writer; see [22, 23] for details.

This section described how, using GRANDMA, gestures and direct manipulation may be combined in an interface by associating different handlers with different views (or with the same view but having different predicates). The two-phase interaction technique, in which gesture and direct-manipulation are combined in a single interaction, is implemented by the gesture handler class. Each gesture handler knows how to collect gestures, classify them as elements of the gesture set expected by the handler, and execute the corresponding gesture semantics.

4. Eager Recognition

As has been seen, gestures may be combined with direct manipulation to create a powerful two-phase interaction technique. This section focuses on how the transition between the phases may be made without any explicit indication from the user. Thus far the paper has concentrated (1) on the description of the two-phase interaction technique, (2) on GRANDMA, a system which supports the technique, and (3) on the use of the technique in GDP, an example application. The treatment in this section is at a lower level: here we are concerned with the pattern recognition technology that is used to implement eager recognition, a particular flavor of the two-phase interaction technique.

4.1. Gestures, subgestures, and full gestures

A gesture is defined as a sequence of points. Denote the number of points in a gesture g as $|g|$, and the particular points as $g_p = (x_p, y_p, t_p)$, $0 \leq p < |g|$. The triple (x, y, t) represents a two-dimensional mouse point (x, y) that arrived at time t. (The actual content of the points turns out to be largely irrelevant for the eager recognition algorithm presented below, and is only given here for concreteness.)

The i^{th} *subgesture* of g, denoted $g[i]$, is defined as a gesture consisting of the first i points of g. Thus, $g[i]_p = g_p$ and $|g[i]| = i$. The subgesture $g[i]$ is simply a prefix of g, and is undefined when $i > |g|$. The term *full gesture* is used when it is necessary to distinguish the full gesture g from its proper subgestures $g[i]$ for $i < |g|$ (see figure 4).

4.2. Statistical single-stroke gesture recognition

We are given a set of C gesture classes, and a number of (full) example gestures of each class, $g_{\hat{c}e}$, $0 \leq c < C$, $0 \leq e < E_{\hat{c}}$, where $E_{\hat{c}}$ is the number of training examples of class c. In GDP, $C = 11$ (the classes are line, rectangle, ellipse, group, text, delete, edit, move, rotate-scale, copy, and dot) and typically we train with 15 examples of each class, i.e. $E_{\hat{c}} = 15$.

The (non-eager) gesture recognition problem is stated as follows: given an input gesture g, determine the class c to which g belongs, i.e. the class whose training examples $g_{\hat{c}e}$ are most like g. (Some of the vagueness here can be eliminated by assuming each class c has a certain probability distribution over the space of gestures, that the training examples of each class were drawn according to that class's distribution, and that the recognition problem is to choose the class c whose distribution, as revealed by its training examples, is the most likely to have produced g.) A classifier $\mathcal{C}$ is a function that attempts to map g to its class c: $c = \mathcal{C}(g)$. As $\mathcal{C}$ is trained on the full gestures $g_{\hat{c}e}$, it is referred to here as a *full* classifier.

The field of pattern recognition in general [6], and on-line handwriting recognition in particular [26], offers many suggestions on how to compute $\mathcal{C}$ given training examples $g_{\hat{c}e}$. Popular methods include the Ledeen recognizer [18] and connectionist models (i.e. neural networks) [8, 10]. Curiously, many gesture researchers [3, 9, 12, 14, 17] choose to hand-code $\mathcal{C}$ for their particular application, rather than attempt to create it from training examples. Lipscomb [15] presents a method tailored to the demands of gesture recognition (rather than handwriting), as does the current author[22, 23].

My method of classifying single-stroke gestures, called *statistical gesture recognition*, works by representing a gesture g by a vector of (currently twelve) features $\mathbf{f}$. Each feature has the property that it can be updated in constant time per mouse point, thus arbitrarily large gestures can be handled. Classification is done via linear discrimination: each class has a linear evaluation function (including a constant term) that is applied to the features, and the class with the maximum evaluation is chosen as the value of $\mathcal{C}(g)$. Training is also efficient, as there is a closed form expression (optimal given some normality assumptions on the distribution of the feature vectors of a class) for determining the evaluation functions from the training data.

There are two more properties of the single-stroke classifier that are exploited by the eager recognition algorithm below. The first is the ability to handle differing costs of misclassification:

simply by adjusting the constant terms of the evaluation functions, it is possible to bias the classifier away from certain classes. This is useful when mistakenly choosing a certain class is a grave error, while mistakenly *not* choosing that class is a minor inconvenience. The other property used below is a side effect of computing a classifier. Theoretically, the computed classifier works by creating a distance metric (the Mahalanobis distance[6]), and the chosen class of a feature vector is simply the class whose mean is closest to the given feature vector under this metric. As will be seen, the distance metric is also used in the construction of eager recognizers.

4.3. The Ambiguous/Unambiguous Classifier

In order to implement eager recognition, a module is needed that can answer the question "has enough of the gesture being entered been seen so that it may be unambiguously classified?" If the gesture seen so far is considered to be a subgesture $g[i]$ of some full gesture g that we have yet to see, we can ask the question this way: "are we reasonably sure that $\mathcal{C}(g[i]) = \mathcal{C}(g)$?" The goal is to design a function $\mathcal{D}$ (for "done") that answers this question: $\mathcal{D}(g[i]) =$ `false` if $g[i]$ is ambiguous (i.e. there might be two full gestures of different classes both of which have $g[i]$ as a subgesture), and $\mathcal{D}(g[i]) =$ `true` if $g[i]$ is unambiguous (meaning all full gestures that might have $g[i]$ as a subgesture are of the same class).

Given $\mathcal{D}$, eager recognition works as follows: Each time a new mouse point arrives it is appended to the gesture being collected, and $\mathcal{D}$ is applied to this gesture. As long as $\mathcal{D}$ returns `false` we iterate and collect the next point. Once $\mathcal{D}$ return `true` the collected gesture is passed to $\mathcal{C}$ whose result is return and the manipulation phase entered.

The problem of eager recognition is thus to produce $\mathcal{D}$ from the given training examples. The insight here is to view this as a classification problem: classify a given subgesture as an ambiguous or unambiguous gesture prefix. The recognition techniques developed for single-path recognition (and discussed in the previous section) are used to build the ambiguous/unambiguous classifier (AUC). $\mathcal{D}$ returns `true` if and only if the AUC classifies the subgesture as unambiguous.

4.4. Complete and incomplete subgestures

Once the idea of using the AUC to generate $\mathcal{D}$ is accepted, it is necessary to produce data to train the AUC. Since the purpose of the AUC is to classify subgestures as ambiguous or unambiguous, the training data must be subgestures that are labeled as ambiguous or unambiguous.

We may use the full classifier $\mathcal{C}$ to generate a first approximation to these two sets (ambiguous and unambiguous). For each example gesture of class c, $g = g_e^{\hat{c}}$, some subgestures $g[i]$ will be classified correctly by the full classifier $\mathcal{C}$, while others likely will not. A subgesture $g[i]$ is termed *complete* with respect to gesture g, if, for all $j, i \le j < |g|, \mathcal{C}(g[j]) = \mathcal{C}(g)$. The remaining subgestures of g are *incomplete*. A complete subgesture is one which is classified correctly by the full classifier, and all larger subgestures (of the same gesture) are also classified correctly.

Figure 5 shows examples of two gestures classes, U and D. Both start with a horizontal segment, but U gestures end with an upward segment, while D gestures end with a downward segment. In this simple example, it is clear that the subgestures which include only the horizontal segment are ambiguous, but subgestures which include the corner are unambiguous. In the figure, each point in the gesture is labeled with a character indicating the classification by C of the subgesture which ends at the point. An uppercase label indicates a complete subgesture, lowercase an incomplete subgesture. Notice that incomplete subgestures are all ambiguous, all unambiguous subgestures are complete, but there are complete subgestures that are ambiguous (along the horizontal segment of the D examples). These subgestures are termed *accidentally* complete since they happened to be classified correctly even though they are ambiguous.

It turns out that even if it is possible to completely determine which subgestures are ambiguous and which are not, using the training methods referred to in section 4.2 to produce a single-stroke recognizer to discriminate between the two classes ambiguous and unambiguous does not work very well. This is because the training methods assume that the distribution of feature vectors within a class is approximately multivariate Gaussian. The distribution of feature vectors within the set of unambiguous subgestures will likely be wildly non-Gaussian, since the member subgestures are drawn from many different gesture classes. For example, in the figure the unambiguous U subgestures are very different than the unambiguous D gestures, so there will be a bimodal distribution of feature vectors in the unambiguous set. Thus, a linear discriminator will not be adequate to discriminate between two classes ambiguous and unambiguous subgestures. What must be done is to turn this two-class problem (ambiguous or unambiguous) into a multi-class problem. This is done by breaking up the ambiguous subgestures into multiple classes, each of which has an approximately normal distribution. The unambiguous subgestures must be similarly partitioned.

To do this, instead of partitioning the example subgestures into just two sets (complete and incomplete), they are partioned into $2C$ sets. These sets are named C-c and I-c for each gesture class c. A complete subgesture $g[i]$ is placed in the class C-c, where $c = C(g[i]) = C(g)$. An incomplete subgesture $g[i]$ is placed in the class I-c, where $c = C(g[i])$ (and it is likely that $c \neq C(g)$). The sets I-c are termed incomplete sets, and the sets C-c, complete sets. Note that the class in each set's name refers to the full classifier's classification of the set's elements. In the case of incomplete subgestures, this is likely not the class of the example gesture of which the subgesture is a prefix. In figure 5 each lowercase letter names a set of incomplete subgestures, while each uppercase letter names a set of complete subgestures.

4.5. Moving Accidentally Complete Subgestures

The next step in generating the training data is to move any accidentally complete subgestures into incomplete classes. Intuitively, it is possible to identify accidentally complete subgestures because they will be similar to some incomplete subgestures (for example, in figure 5 the subgestures along the horizontal segment are all similar even though some are complete and others are incomplete.) A threshold applied to the Mahalanobis distance metric mentioned in section 4.2 may be used to test for this similarity.

To do so, the distance of each subgesture $g[i]$ in each complete set to the mean of each incomplete set is measured. If $g[i]$ is sufficiently close to one of the incomplete sets, it is removed from its complete set, and placed in the closest incomplete set. In this manner, an example subgesture that was accidentally considered complete (such as a right stroke of a D gesture) is grouped together with the other incomplete right strokes (class I-D in this case).

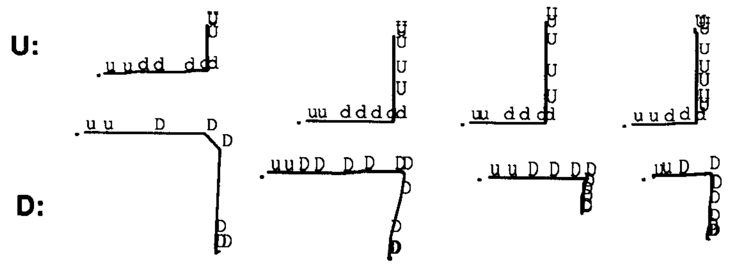


Figure 5: Incomplete and complete subgestures of U and D

The character indicates the classification (by the full classifier) of each subgesture. Uppercase characters indicate complete subgestures, meaning that the subgesture and all larger subgestures are correctly classified. Note that along the horizontal segment (where the subgestures are ambiguous) some subgestures are complete while others are not.

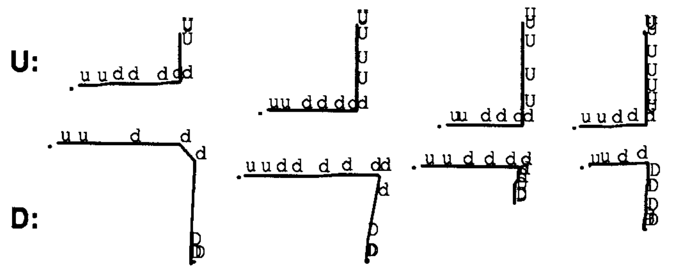


Figure 6: Accidentally complete subgestures have been moved

Comparing this to figure 5 it can be seen that the subgestures along the horizontal segment of the D *gestures have been made incomplete. Unlike before, after this step all ambiguous subgestures are incomplete.*

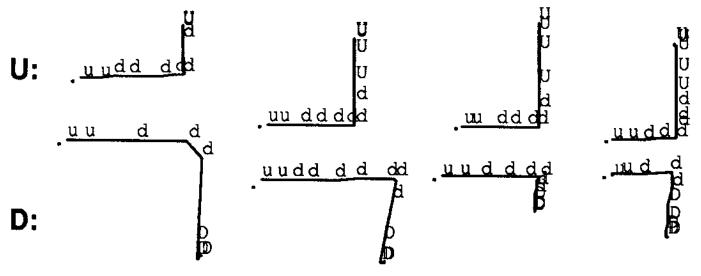


Figure 7: Classification of subgestures of U and D

This shows the results of running the AUC on the training examples. As can been seen, the AUC performs conservatively, never indicating that a subgestures is unambiguous when it is not, but sometimes indicating ambiguity of an unambiguous subgesture.

Quantifying exactly what is meant by "sufficiently close" turns out to be rather difficult. The threshold on the distance metric is computed as follows: The distance of the mean of each full gesture class to the mean of each incomplete subgesture class is computed, and the minimum found. However, distances less than another threshold are not included in the minimum calculation to avoid trouble when an incomplete subgesture looks like a full gesture of a different class. (This is the case if, in addition to U and D, there is a third gesture class consisting simply of a right stroke.) The threshold used is 50% of that minimum.

The complete subgestures of a full gesture are tested for accidental completeness from largest (the full gesture) to smallest. Once a subgesture is determined to be accidentally complete, it and the remaining (smaller) complete subgestures are moved to the appropriate incomplete classes.

Figure 6 shows the classes of the subgestures in the example after the accidentally complete subgestures have been moved. Note that now the incomplete subgestures (lowercase labels) are all ambiguous.

4.6. Create and tweak the AUC

Now that there is training data containing C complete classes (indicating unambiguous subgestures), and C incomplete classes (indicating ambiguous subgestures), it is a simple matter to run the single-stroke training algorithm (section 4.2) to create a classifier to discriminate between these $2C$ classes. This classifier will be used to compute the function $\mathcal{D}$ as follows: if this classifier places a subgesture s in any incomplete class, $\mathcal{D}(s)$ = `false`, otherwise the s is judged to be in one of the complete classes, in which case $\mathcal{D}(s)$ = `true`.

It is very important that subgestures not be judged unambiguous wrongly. This is a case where the cost of misclassification is unequal between classes: a subgesture erroneously classified ambiguous will merely cause the recognition not to be as eager as it could be, whereas a subgesture erroneously classified unambiguous will very likely result in the gesture recognizer misclassifying the gesture (since it has not seen enough of it to classify it unambiguously). To avoid this, the constant terms of the evaluation function of the incomplete classes i are incremented by a small amount. The increment is chosen to bias the classifier so that it believes that ambiguous gestures are five times more likely than unambiguous gestures. In this way, it is much more likely to choose an ambiguous class when unsure.

Each incomplete subgesture is then tested on the new classifier. Any time such a subgesture is classified as belonging to a complete set (a serious mistake), the constant term of the evaluation function corresponding to the complete set is adjusted automatically (by just enough plus a little more) to keep this from happening.

Figure 7 shows the classification by the final classifier of the subgestures in the example. A larger example of eager recognizers is presented in the next section.

4.7. Summarizing the eager recognition training algorithm

While the details are fairly involved, the idea behind the eager recognition technology is straightforward. The basic problem is to determine if enough of a gesture has been seen to classify it

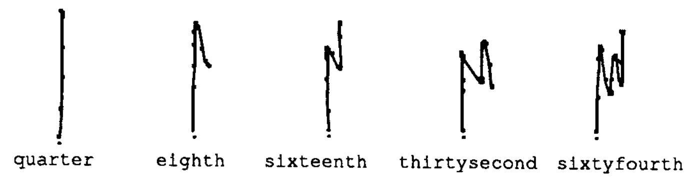


Figure 8: A set of gestures not amenable to eager recognition
Because all but the last gesture is approximately a subgesture of the one to its right, these gestures would always be considered ambiguous by the eager recognizer, and thus would never be eagerly recognized. The period indicates the first point of each gesture.

unambiguously. This determination is itself a classification problem to which the same trainable gesture recognition technology may be applied. The main hurdle is to produce data to train this classifier to discriminate between ambiguous and unambiguous subgestures. This is done by running the full classifier on every subgesture of the original training examples. Any subgesture classified differently than the full gesture from which it arose is considered ambiguous; also ambiguous are those subgestures that happen to be classified the same as their full gestures but are similar to subgestures already considered to be ambiguous. For safety, after being trained to discriminate between ambiguous and unambiguous subgestures, the new classifier is conservatively biased toward classifying subgestures as ambiguous.

5. Evaluating Eager Recognition

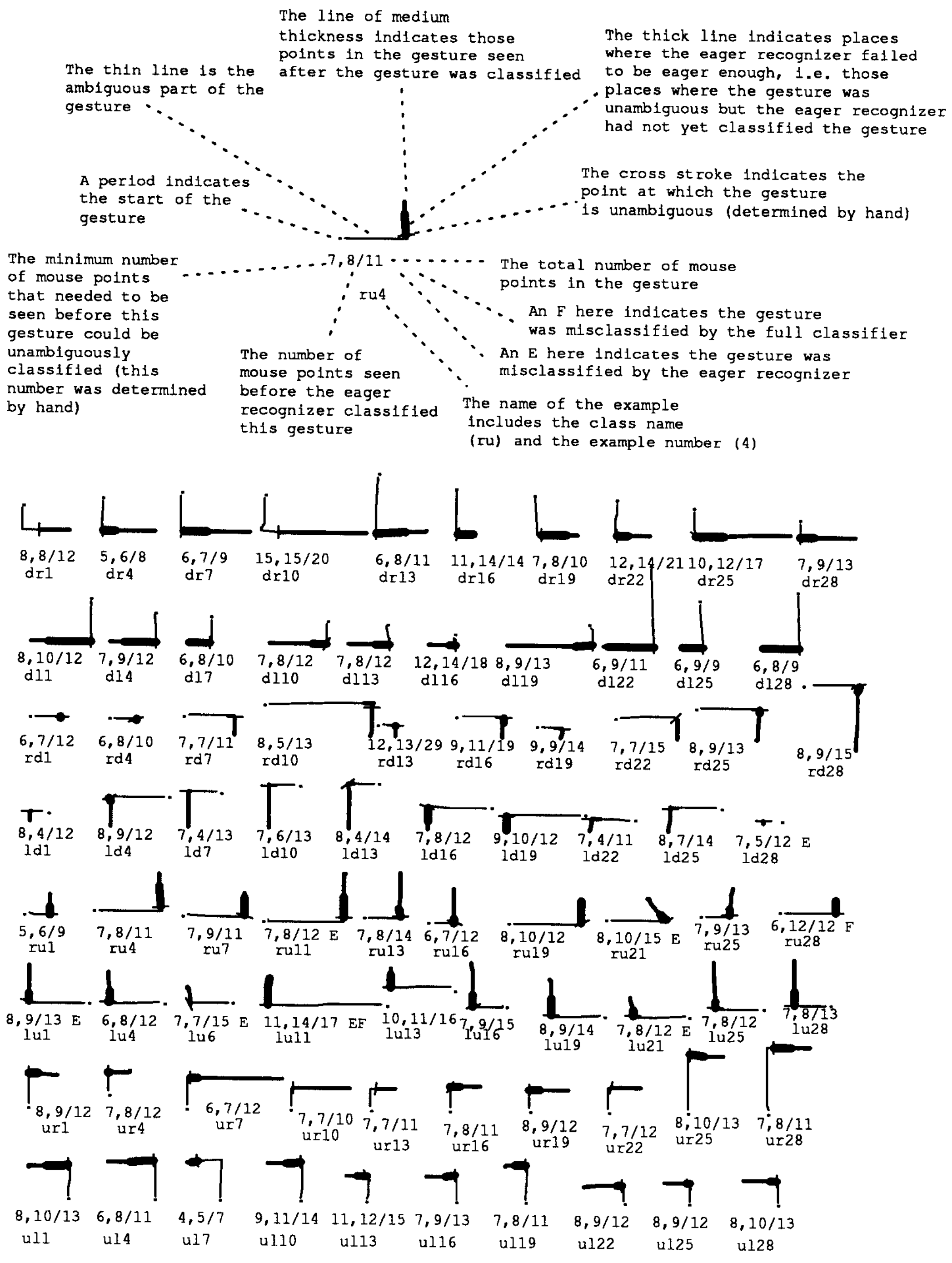
How well the eager recognition algorithm works depends on a number of factors, the most critical being the gesture set itself. It is very easy to design a gesture set that does not lend itself well to eager recognition; for example, there would be almost no benefit trying to use eager recognition on Buxton's note gestures [3] (figure 8). This is because the note gestures for longer notes are subgestures of the note gestures for shorter notes, and thus would always be considered ambiguous by the eager recognizer.

In order to determine how well the eager recognition algorithm works, an eager recognizer was created to classify the eight gestures classes shown in 9. Each class named for the direction of its two segments, e.g. "ur" means "up, right." Each of these gestures is ambiguous along its initial segment, and becomes unambiguous once the corner is turned and the second segment begun.

The eager recognizer was trained with ten examples of each of the eight classes, and tested on thirty examples of each class. The figure shows ten of the thirty test examples for each class, and includes all the examples that were misclassified.

Two comparisons are of interest for the gesture set: the eager recognition rate versus the recognition rate of the full classifier, and the eagerness of the recognizer versus the maximum possible eagerness. The eager recognizer classified 97.0% of the gestures correctly, compared to 99.2% correct for the full classifier. Most of the eager recognizer's errors were due to a corner looping 270 degrees rather than being a sharp 90 degrees, so it appeared to the eager recognizer the second stroke was going in the opposite direction than intended. In the figure "E" indicates a gesture misclassified by the eager recognizer, and "F" indicates a misclassification by the full classifier.

KEY



The line of medium thickness indicates those points in the gesture seen after the gesture was classified
The thin line is the ambiguous part of the gesture
The thick line indicates places where the eager recognizer failed to be eager enough, i.e. those places where the gesture was unambiguous but the eager recognizer had not yet classified the gesture
A period indicates the start of the gesture
The cross stroke indicates the point at which the gesture is unambiguous (determined by hand)
The minimum number of mouse points that needed to be seen before this gesture could be unambiguously classified (this number was determined by hand)
7,8/11
ru4
The total number of mouse points in the gesture
An F here indicates the gesture was misclassified by the full classifier
An E here indicates the gesture was misclassified by the eager recognizer
The number of mouse points seen before the eager recognizer classified this gesture
The name of the example includes the class name (ru) and the example number (4)
8,8/12 dr1
5,6/8 dr4
6,7/9 dr7
15,15/20 dr10
6,8/11 dr13
11,14/14 dr16
7,8/10 dr19
12,14/21 dr22
10,12/17 dr25
7,9/13 dr28
8,10/12 dl1
7,9/12 dl4
6,8/10 dl7
7,8/12 dl10
7,8/12 dl13
12,14/18 dl16
8,9/13 dl19
6,9/11 dl22
6,9/9 dl25
6,8/9 dl28
6,7/12 rd1
6,8/10 rd4
7,7/11 rd7
8,5/13 rd10
12,13/29 rd13
9,11/19 rd16
9,9/14 rd19
7,7/15 rd22
8,9/13 rd25
8,9/15 rd28
8,4/12 ld1
8,9/12 ld4
7,4/13 ld7
7,6/13 ld10
8,4/14 ld13
7,8/12 ld16
9,10/12 ld19
7,4/11 ld22
8,7/14 ld25
7,5/12 E ld28
5,6/9 ru1
7,8/11 ru4
7,9/11 ru7
7,8/12 E ru11
7,8/14 ru13
6,7/12 ru16
8,10/12 ru19
8,10/15 E ru21
7,9/13 ru25
6,12/12 F ru28
8,9/13 E lu1
6,8/12 lu4
7,7/15 E lu6
11,14/17 EF lu11
10,11/16 lu13
7,9/15 lu16
8,9/14 lu19
7,8/12 E lu21
7,8/12 lu25
7,8/13 lu28
8,9/12 ur1
7,8/12 ur4
6,7/12 ur7
7,7/10 ur10
7,7/11 ur13
7,8/11 ur16
8,9/12 ur19
7,7/12 ur22
8,10/13 ur25
7,8/11 ur28
8,10/13 ul1
6,8/11 ul4
4,5/7 ul7
9,11/14 ul10
11,12/15 ul13
7,9/13 ul16
7,8/11 ul19
8,9/12 ul22
8,9/12 ul25
8,10/13 ul28

Figure 9: The performance of the eager recognizer on easily understood data

On the average, the eager recognizer examined 67.9% of the mouse points of each gesture before deciding the gesture was unambiguous. By hand I determined for each gesture the number of mouse points from the start through the corner turn, and concluded that on the average 59.4% of the mouse points of each gesture needed to be seen before the gesture could be unambiguously classified. The parts of each gesture at which unambiguous classification could have occurred but did not are indicated in the figure by thick lines.

Figure 10 shows the performance of the eager recognizer on GDP gestures. The eager recognizer was trained with 10 examples of each of 11 gesture classes, and tested on 30 examples of each class, five of which are shown in the figure. The GDP gesture set was slightly altered to increase eagerness: the group gesture was trained clockwise because when it was counterclockwise it prevented the copy gesture from ever being eagerly recognized. For the GDP gestures, the full classifier had a 99.7% correct recognition rate as compared with 93.5% for the eager recognizer. On the average 60.5% of each gesture was examined by the eager recognizer before classification occurred. For this set no attempt was made to determine the minimum average gesture percentage that needed to seen for unambiguous classification.

From these tests we can conclude that the trainable eager recognition algorithm performs acceptably but there is plenty of room for improvement, both in the recognition rate and the amount of eagerness.

Computationally, eager recognition is quite tractable on modest hardware. A fixed amount of computation needs to occur on each mouse point: first the feature vector must be updated (taking 0.5 msec on a DEC MicroVAX II), and then the vector must be classified by the AUC (taking 0.27 msec per class, or 6 msec in the case of GDP).

6. Conclusion

In this paper I have shown how gesturing and direct manipulation can be combined in a two-phase interaction technique that exhibits the best qualities of both. These include the ability to specify an operation, the operands, and additional parameters with a single, intuitive stroke, with some of those parameters being manipulated directly in the presence of application feedback. The technique of eager recognition allows a smooth transition between the gesture collection and the direct manipulation phases of the interaction. An algorithm for creating eager recognizers from example gestures was presented and evaluated.

There is an unexpected benefit of combining gesture and direct manipulation in a single interaction: gesture classification is often simpler and more accurate. Consider the "move text" gesture in figure 1. Selecting the text to move is a circling gesture that will not vary too much each time the gesture is made. However, after the text is selected the gesture continues and the destination of the text is indicated by the "tail" of the gesture. The size and shape of this tail will vary greatly with each instance of the "move text" gesture. This variation makes the gesture difficult to recognize in general, especially when using a trainable recognizer. Perhaps this is why many researchers hand code their classifiers. In any case, in a two-phase interaction the tail is no longer part of the gesture, but instead part of the manipulation. Trainable recognition techniques will be much more successful on the remaining prefix.

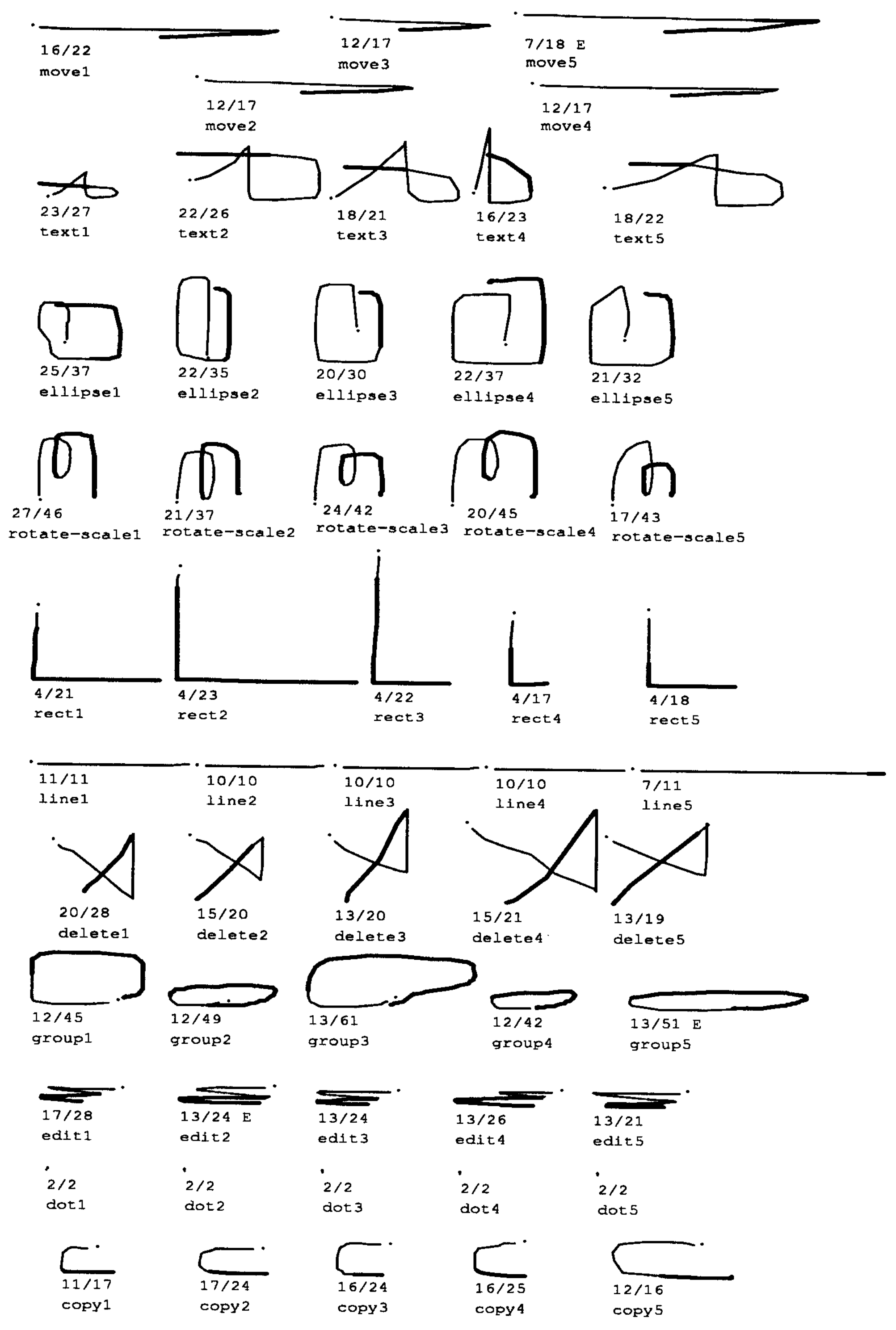


Figure 10: The performance of the eager recognizer on GDP gestures

The transitions from thin to thick lines indicates where eager recognition occurred.

The two-phase interaction technique is also applicable to multi-path gestures. Using the Sensor Frame [16] as an input device, I have implemented a drawing program based on multiple finger gestures. The results have been quite encouraging. For example, the translate-rotate-scale gesture is made with two fingers, which during the manipulation phase allow for simultaneous rotation, translation, and scaling of graphic objects. Even some single finger gestures allow additional fingers to be brought into the field of view during manipulation, thus allowing additional parameters (such as color and thickness) to be specified interactively.

In the future, I plan to incorporate gestures (and the two-phase interaction) into some existing object-oriented user-interface toolkits, notably the Andrew Toolkit[20] and the NeXT Application Kit[19]. Other extensions including handling multi-stroke gestures, and integrating gesture recognition with the handwriting recognition used on the notebook computers now beginning to appear. Further work is needed to utilize devices, such as the DataGlove[28], which have no explicit signaling with which to indicate the start of a gesture.

Acknowledgements

I gratefully acknowledge CMU's School of Computer Science for supporting this work as part of my dissertation effort, and CMU's Information Technology Center for supporting me in the creation of this paper. Thanks also to Roger Dannenberg for his helpful comments.

References

[1] Eric Bier and Maureen Stone. Snap-dragging. *Computer Graphics*, 20(4):233–240, 1986.

[2] W. Buxton. There's more to interaction than meets the eye: Some issues in manual input. In D.A. Norman and S.W. Draper, editors, *User Centered Systems Design: New Perspectives on Human-Computer Interaction*, pages 319–337. Lawrence Erlbaum Associates, Hillsdale, N.J., 1986.

[3] W. Buxton, R. Sniderman, W. Reeves, S. Patel, and R. Baecker. The evolution of the SSSP score-editing tools. In Curtis Roads and John Strawn, editors, *Foundations of Computer Music*, chapter 22, pages 387–392. MIT Press, Cambridge, Mass., 1985.

[4] Michael L. Coleman. Text editing on a graphic display device using hand-drawn proofreader's symbols. In M. Faiman and J. Nievergelt, editors, *Pertinent Concepts in Computer Graphics, Proceedings of the Second University of Illinois Conference on Computer Graphics*, pages 283–290. University of Illinois Press, Urbana, Chicago, London, 1969.

[5] Brad J. Cox. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1986.

[6] R. Duda and P. Hart. *Pattern Classification and Scene Analysis.* Wiley Interscience, 1973.

[7] Dario Giuse. DP command set. Technical Report CMU-RI-TR-82-11, Carnegie Mellon University Robotics Institute, October 1982.

[8] I. Guyon, P. Albrecht, Y. Le Cun, J. Denker, and W. Hubbard. Design of a neural network character recognizer for a touch terminal. *Pattern Recognition*, forthcoming.

[9] T.R. Henry, S.E. Hudson, and G.L. Newell. Integrating gesture and snapping into a user interface toolkit. In *UIST '90*, pages 112–122. ACM, 1990.

[10] J. Hollan, E. Rich, W. Hill, D. Wroblewski, W. Wilner, K. Wittenberg, J. Grudin, and Members of the Human Interface Laboratory. An introduction to HITS: Human interface tool suite. Technical Report ACA-HI-406-88, Microelectronics and Computer Technology Corporation, Austin, Texas, 1988.

[11] IBM. The Paper-Like Interface. In *CHI '89 Techical Video Program: Interface Technology*, volume Issue 47 of *SIGGRAPH Video Review*. ACM, 1989.

[12] Joonki Kim. Gesture recognition by feature analysis. Technical Report RC12472, IBM Research, December 1986.

[13] Glenn E. Krasner and Stephen T. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, August 1988.

[14] G. Kurtenbach and W. Buxton. GEdit: A test bed for editing by contiguous gestures. to be published in SIGCHI Bulletin, 1990.

[15] James S. Lipscomb. A trainable gesture recognizer. *Pattern Recognition*, 1991. Also available as IBM Tech Report RC 16448 (#73078).

[16] P. McAvinney. Telltale gestures. *Byte*, 15(7):237–240, July 1990.

[17] Margaret R. Minsky. Manipulating simulated objects with real-world gestures using a force and position sensitive screen. *Computer Graphics*, 18(3):195–203, July 1984.

[18] W.M. Newman and R.F. Sproull. *Principles of Interactive Computer Graphics*. McGraw-Hill, 1979.

[19] NeXT. *The NeXT System Reference Manual*. NeXT, Inc., 1989.

[20] A.J. Palay, W.J. Hansen, M.L. Kazar, M. Sherman, M.G. Wadlow, T.P. Neuendorffer, Z. Stern, M. Bader, and T. Peters. The Andrew toolkit: An overview. In *Proceedings of the USENIX Technical Conference*, pages 11–23, Dallas, February 1988.

[21] James R. Rhyne and Catherine G. Wolf. Gestural interfaces for information processing applications. Technical Report RC12179, IBM T.J. Watson Research Center, IBM Corporation, P.O. Box 218, Yorktown Heights, NY 10598, September 1986.

[22] Dean Rubine. Specifying gestures by example. In *SIGGRAPH 91*. ACM, 1991.

[23] Dean Rubine. *The Automatic Recognition of Gestures*. PhD thesis, School of Computer Science, Carnegie Mellon University, forthcoming, 1991.

[24] R.W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2), April 1986.

[25] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, pages 57–62, August 1983.

[26] C.C. Tappert. On-line handwriting recognition - a survey. Technical Report RC 14045, IBM T.J. Watson Research Center, August 1987.

[27] A. Tevanian. MACH: A basis for future UNIX development. Technical Report CMU-CS-87-139, Carnegie Mellon University Computer Science Dept., Pittsburgh, PA, 1987.

[28] T. Zimmerman, J. Lanier, C. Blanchard, S. Bryson, and Y. Harvill. A hand gesture interface device. *CHI+GI*, pages 189–192, 1987.