Assignment 3 – Symbol Recognizer
CAP5937

Due: 10/08/07 11:59pm

The focus of this third assignment is to learn the intricacies of creating a machine learning-based symbol recognizer. This is actually the first part of a two part assignment where you will be creating a simple pen-based calculator. In this assignment you are going to create a symbol recognition engine based on Rubine’s 1991 SIGGRAPH paper, “Specifying Gestures by Example”.

Requirements

Your symbol recognizer must be able to recognize the following symbols:

0,1,2,3,4,5,6,7,8,9,+,-,*,t,a,n,s,c,i, and the square root symbol.

You should also be able to use your scribble gesture to erase symbols.

You will also need to perform an experiment to evaluate your recognizer’s accuracy. The experiment will explore how the number of training samples used per symbol affects recognition. I would suggest testing your recognizer with 5, 10, 15, and 20 samples per symbol. For the test itself, I would write each symbol 5 or 10 times which should give you a good accuracy number. Please put the results of your experiment in the README file.

Strategy

To implement your symbol recognizer, read the Rubine paper. It is fairly straightforward once you understand the mathematics.

Things you should keep in mind.

1. You need to find a way to invoke the recognizer. You can have it run in real time or in batch mode (for ex. lassoing the symbol or symbols and taping to invoke the recognizer).

2. Regardless of the invocation method, you will need some form of ink segmentation since you must be able to detect when a symbol has 2 or more strokes. Simple line segment intersection should suffice here since it is relatively easy to determine if you have a multi-stroke symbol in our alphabet.

3. Rubine’s algorithm uses matrices. So make use of the Matrix library found on the course webpage.

4. Rubine’s algorithm is designed to deal with only single stroke symbols. To recognize multi-stroke symbols, simply compute the features for each stroke and take the average.

5. You will need to show recognition results to the user. A simple text box is fine but if you want to be more elaborate feel free to do so.

Deliverables

You must submit a zip file containing your source and any relevant files needed to compile and run your application. Also include a README file describing what works and what does not in your application, the results of your accuracy experiment, any known bugs, and any problems you encountered. Please include a file I can open in your application that has all of the symbols written down in ink. This will show me how you wrote your symbols for testing purposes. To submit, you can email me your zip file.

Grading

Grading will be loosely based on the following:

80% correct functionality
20% documentation

Specifying Gestures by Example

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

Abstract

Gesture-based interfaces offer an alternative to traditional keyboard, menu, and direct manipulation interfaces. The ability to specify objects, an operation, and additional parameters with a single intuitive gesture appeals to both novice and experienced users. Unfortunately, gesture-based interfaces have not been extensively researched, partly because they are difficult to create. This paper describes GRANDMA, a toolkit for rapidly adding gestures to direct manipulation interfaces. The trainable single-stroke gesture recognizer used by GRANDMA is also described.

Keywords — gesture, interaction techniques, user interface toolkits, statistical pattern recognition

1 Introduction

Gesture, as the term is used here, refers to hand markings, entered with a stylus or mouse, that indicate scope and commands [18]. Buxton gives the example of a proofreader's mark for moving text [1]. A single stroke indicates the operation (move text), the operand (the text to be moved), and additional parameters (the new location of the text). The intuitiveness and power of this gesture hints at the great potential of gestural interfaces for improving input from people to machines, historically the bottleneck in human-computer interaction. Additional motivation for gestural input is given by Rhyne [18] and Buxton [1].

A variety of gesture-based applications have been created. Coleman implemented a text editor based on proofreader's marks [3]. Minsky built a gestural interface to the LOGO programming language [13]. A group at IBM constructed a spreadsheet application that combines gesture and handwriting [18]. Buxton's group produced a musical score editor that uses gestures for entering notes [2] and more recently a graphical editor [9]. In these gesture-based applications (and many others) the module that distinguishes between the gestures expected by the system, known as the *gesture recognizer*, is hand coded. This code is usually complicated, making the systems (and the set of gestures accepted) difficult to create, maintain, and modify.

Creating hand-coded recognizers is difficult. This is one reason why gestural input has not received greater attention. This paper describes how gesture recognizers may be created automatically from example gestures, removing the need for hand coding. The recognition technology is incorporated into GRANDMA (Gesture Recognizers Automated in a Novel Direct Manipulation Architecture), a toolkit that enables an implementor to create gestural interfaces for applications with direct manipulation ("click-and-drag") interfaces. In the current work, such applications must themselves be built using GRANDMA. Hopefully, this paper will stimulate the integration of gesture recognition into other user interface construction tools.

Very few tools have been built to aid development of gesture-based applications. Artkit [7] provides architectural support for gestural interfaces, but no support for creating recognizers. Existing trainable character recognizers, such as those built from neural networks [6] or dictionary lookup [15], have significant shortcomings when applied to gestures, due to the different requirements gesture recognition places on a recognizer. In response, Lipscomb [11] has built a trainable recognizer specialized toward gestures, as has this author.

The recognition technology described here produces a small, fast, and accurate recognizers. Each recognizer is rapidly trained from a small number of examples of each gesture. Some gestures may vary in size and/or orientation while others depend on size and/or orientation for discrimination. Dynamic attributes (left-to-right or right-to-left, fast or slow) may be considered in classification. The gestural attributes used for classification are generally meaningful, and may be used as parameters to application routines.

The remainder of the paper describes various facets of GRANDMA. GDP, a gesture-based drawing program built using GRANDMA, is used as an example. First GDP's

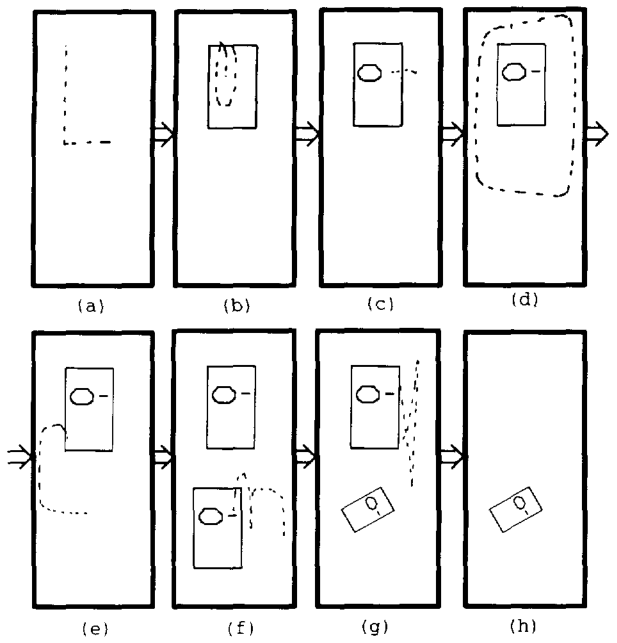


Figure 1: GDP, a gesture-based drawing program. *The figure shows a sequence of windows in a GDP session. Gestures are illustrated with dotted lines, and the resulting graphics with solid lines. The effect of each gesture is shown in the panel which follows it; for example panel (a) shows a* rectangle *gesture, and panel (b) shows the created rectangle.*

operation is sketched from the user's point of view. Next, the gesture designer's use of GRANDMA to add gestures to a click-and-drag version of GDP is described. The details of the single-stroke gesture recognition and training algorithms are then covered. This is followed by a brief discussion of two extensions of the algorithms, eager recognition (in which a gesture is recognized as soon as enough of it has been seen to do so unambiguously) and multi-finger gesture recognition. The paper concludes with an eye toward future work. A more detailed treatment of the topics covered in this paper may be found in the author's dissertation [20].

2 GDP, an Example Gesture-based Application

Figure 1 shows some snapshots of GDP in action. When first started, GDP presents the user with a blank window. Panel **(a)** shows the screen as a rectangle gesture is being entered. The user begins the gesture by positioning the mouse cursor and pressing a mouse button. The user then draws the gesture by moving the mouse. The inking, shown with dotted lines in the figure, disappears as soon as the gesture is recognized.

The end of the gesture is indicated in one of two ways. If the user simply releases the mouse button immediately after drawing "L," a rectangle is created, one corner of which is at the start of the gesture (where the button was first pressed), and the opposite corner is at the end of the gesture (where the button was released). Another way to end the gesture is to stop moving the mouse for a given amount of time (0.2 seconds by default), while still pressing the mouse button. In this case, a rectangle is created with one corner at the start of the gesture, and the opposite corner at the mouse's location when the timeout occurs. As long as the button is held, that corner is dragged by the mouse, enabling the size and shape of the rectangle to be determined interactively.

Panel **(b)** of Figure 1 shows the created rectangle and an ellipse gesture, whose starting point is the center of the new ellipse. After recognition the ellipse's size and eccentricity may be interactively determined by dragging.

Panel **(c)** shows the created ellipse, and a line gesture. As expected, the start of the gesture determines one endpoint of the line, and the mouse position after the gesture has been recognized determines the other endpoint, allowing the line to be rubberbanded.

Panel **(d)** shows all three shapes being encircled by a pack gesture. This gesture groups all the objects that it encloses into a single composite object, which can then be manipulated as a unit.

Panel **(e)** shows a copy gesture: the composite object is copied and the copy is then dragged by the mouse.

Panel **(f)** shows the rotate-scale gesture. The object is made to rotate around the starting point of the gesture; a point on the object is dragged by the mouse allowing the user to interactively determine the size and orientation of the object.

Panel **(g)** shows the delete gesture, essentially an "X" drawn with a single stroke. In GDP, the start of the gesture (rather than its self-intersection point) determines the object to be deleted.

Each GDP gesture corresponds to a high-level operation. The class of the gesture determines the operation; attributes of the gesture determine the operands (scope) as well as any additional parameters. For example, the delete gesture specifies the object to be deleted, the pack gesture specifies the objects to be grouped, and the line gesture specifies the endpoints of the line. Note how gesturing and direct-manipulation are combined in a new two-phase interaction technique: when the gesture collection phase ends, gesture classification occurs, and the manipulation phase begins.

The gestures used in GDP are all single strokes. This is an intentional limitation of GRANDMA, and a marked departure from multi-stroke gesture-based systems. The single-stroke restriction avoids the segmentation problem of multi-stroke character recognition [21], allowing shorter timeouts to be used. Also, the emphasis on single strokes has led to the new two-phase interaction technique as well as to eager recognition (both of which are potentially applicable to multi-stroke gestures). Finally, with single-stroke gestures an entire command coincides with a single physical tensing and relaxing of the user, a property thought to contribute positively to the usability of user interfaces [1].

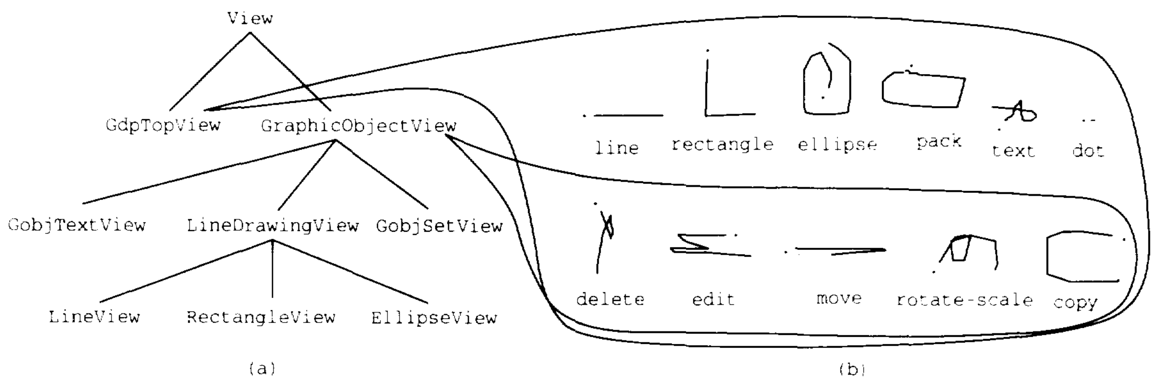


Figure 2: GDP view classes and associated gesture sets (a period marks the first point of each gesture).

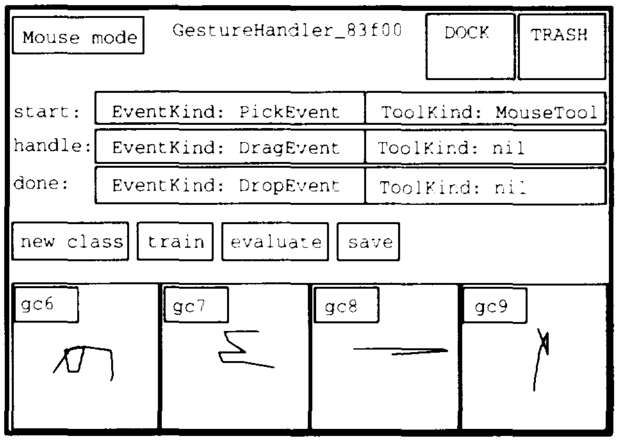
One obvious disadvantage is that many intuitive symbols (e.g. "X" and "—>") are ruled out.

3 Design GDP's Gestures with GRANDMA

Given a click-and-drag interface to an application, the gesture designer modifies the way input is handled, leaving the output mechanisms untouched. Both the click-and-drag interface and the application must be built using the object-oriented toolkit GRANDMA. Figure 2a shows GDP's view class hierarchy, the heart of its output mechanism. The gesture designer must first determine which of the view classes are to have associated gestures, and then design a set of intuitive gestures for them. Figure 2b shows the sets of gestures associated with GDP's `GdpTopView` and `GraphicObjectView` classes. A `GdpTopView` object refers to the window in which GDP runs. A `GraphicObjectView` object is either a line, rectangle, ellipse, or text object, or a set of these.

GRANDMA is a Model/View/Controller-like system [8]. In GRANDMA, a single input event handler (a "controller" in MVC terms) may be associated with a view class, and thus shared between all instances of the class (including instances of subclasses). This adds flexibility while eliminating a major overhead of Smalltalk MVC, where one controller object is associated with each view object that expects input.

The gesture designer adds gestures to GDP's initial click-and-drag interface at runtime. First, a new gesture handler is created and associated with the `GraphicObjectView` class, easily done using GRANDMA. Figure 3 shows the gesture handler window after four gestures have been created (using the "new class" button), and Figure 4 shows the window in which seven examples of the delete gesture have been entered. Empirical evidence suggests that 15 training examples per gesture class is adequate (see Section 4.5). These 15 examples should reflect any desired variance in size and/or orientation of the gesture.



start:	EventKind: PickEvent	ToolKind: MouseTool
handle:	EventKind: DragEvent	ToolKind: nil
done:	EventKind: DropEvent	ToolKind: nil

Figure 3: Manipulating gesture handlers at runtime. *This window allows gestures to be added to or deleted from the set of gestures recognized by a particular view class.*

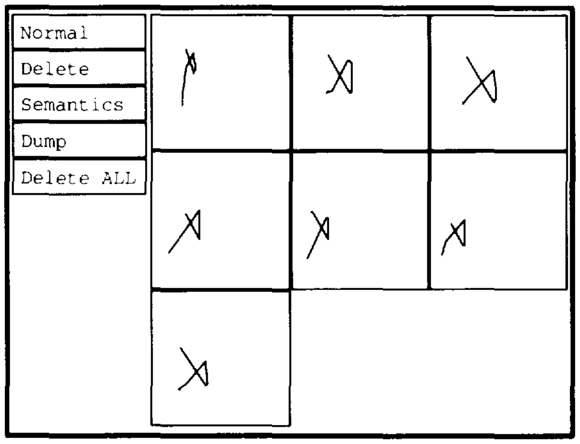
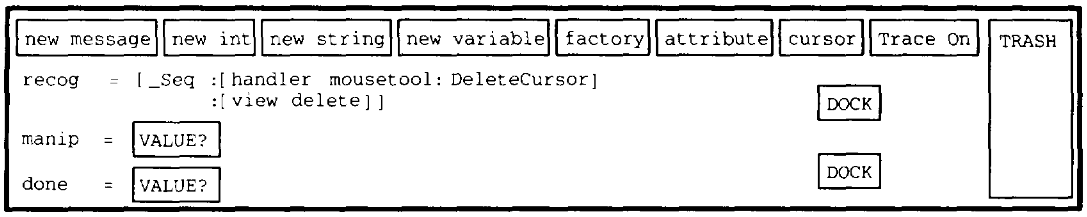


Figure 4: Entering examples of the delete gesture. *In this window, training examples of a gesture class may be added or deleted. The "Delete ALL" button deletes all the gesture's examples, making it easy to try out various forms of a gesture.*



```
recog  = [_Seq :[handler mousetool: DeleteCursor]
               :[view delete]]
manip  = VALUE?
done   = VALUE?
```

Figure 5: Editing the semantics of the **delete** gesture.

The "Semantics" button is used to initiate editing of the semantics of each gesture in the handler's set. Clicking on the button brings up a structured editing and browsing interface to a simple Objective-C [4] interpreter (Figure 5). The designer enters an expression for each of the three semantic components: `recog` is evaluated when the gesture is recognized (i.e. when the mouse stops moving), `manip` is evaluated on subsequent mouse points, and `done` is evaluated when the mouse button is released. The **delete** semantics shown in the figure simply change the mouse cursor to a delete cursor (providing feedback to the user), and then delete the view at which the gesture was aimed. The designer may now immediately try out the **delete** gesture, as in Figure 1g.

The designer repeats the process to create a gesture handler for the set of gestures associated with class `GdpTopView`, the view that refers to the window in which GDP runs. This handler recognizes the **line**, **rectangle**, and **ellipse** gestures (which create graphic objects), the **pack** gesture (which creates a set out of the enclosed graphic objects), the **dot** gesture (which repeats the last command), the **text** gesture (which allows text to be entered from the keyboard), and the **delete**, **edit**, **move**, **rotate-scale**, and **copy** gestures (which are also handled by `GraphicObjectView`'s gesture handler but when made at a `GdpTopView` simply change the cursor without operating directly on a graphic object).

The attributes of the gesture may be used in the gesture semantics. For example, the semantics of the **line** gesture are:

```
 recog =
    [Seq :[handler mousetool:LineCursor]
         :[[view createLine]
            setEndpoint:0
            x:<startX> y:<startY>]];
manip = [recog setEndpoint:1
          x:<currentX> y:<currentY>];
done = nil;
```

The semantic expressions execute in a rich environment. For example, `view` is bound to the view at which the gesture was directed (in this case a `GdpTopView`) and `handler` is bound to the current gesture handler. Note that `Seq` executes its arguments sequentially, returning the last value, in this case the newly created line. The last value is bound to `recog` for later use in the `manip` expression.

The example shows how the gesture attributes, shown in angle brackets, are useful in the semantic expressions. The attributes `<startX>` and `<startY>`, the coordinates of the first point in the gesture, determine one endpoint of the line, while `<currentX>` and `<currentY>`, the mouse coordinates, determine the other endpoint.

Other gestural attributes are useful in gesture semantics. For example, the length of the **line** gesture can be used to control the line thickness. The initial angle of the **rectangle** gesture can determine the orientation of the rectangle. The attribute `<enclosed>`, which contains the list of views enclosed by the gesture, is used, for example, by the **pack** gesture (Figure 1d).

When a gesture is made over multiple gesture-handling views, the union of the set of gestures recognized by each handler is used, with priority given to the topmost view. For example, any gesture made at a GDP `GraphicObjectView` is necessarily made over the `GdpTopView`. A **delete** gesture would be handled by the `GraphicObjectView` while a **line** gesture would be handled by the `GdpTopView`. Set union also occurs when gestures are (conceptually) inherited via the view class hierarchy. For example, the gesture designer might create a new gesture handler for the `GobjSetView` class containing an **unpack** gesture. The set of gestures recognized by `GobjSetViews` would then consist of the **unpack** gesture as well as the five gestures already handled by `GraphicObjectView`.

Space limitations preclude an explanation of how GRANDMA's object-oriented user interface toolkit is used to construct applications and their click-and-drag interfaces. Also omitted is a discussion of GRANDMA's internals. The interested reader is referred to the author's dissertation[20].

4 Statistical Single-Stroke Gesture Recognition

This section discusses the low-level recognition of two-dimensional, single-stroke gestures. Both the classification and the training algorithms are short and self-contained, making them useful for those wishing to include trainable gesture recognition in their interfaces.

For the present, it is assumed that the start and end of the input gesture are clearly delineated. As mentioned previously, the start of the gesture is typically indicated by the pressing of a mouse button, while the end is indicated by releasing the button or ceasing to move the mouse.

Each gesture is an array g of P time-stamped sample points:

$$g_p = (x_p, y_p, t_p) \qquad 0 \le p < P$$

Some simple preprocessing is done to eliminate jiggle: an input point within 3 pixels of the previous input point is discarded.

The gesture recognition problem is stated as follows: There is a set of C gesture classes, numbered 0 through $C-1$. Each class is specified by example gestures. Given an input gesture g, determine the class to which g belongs (*i.e.* the class whose members are most like g).

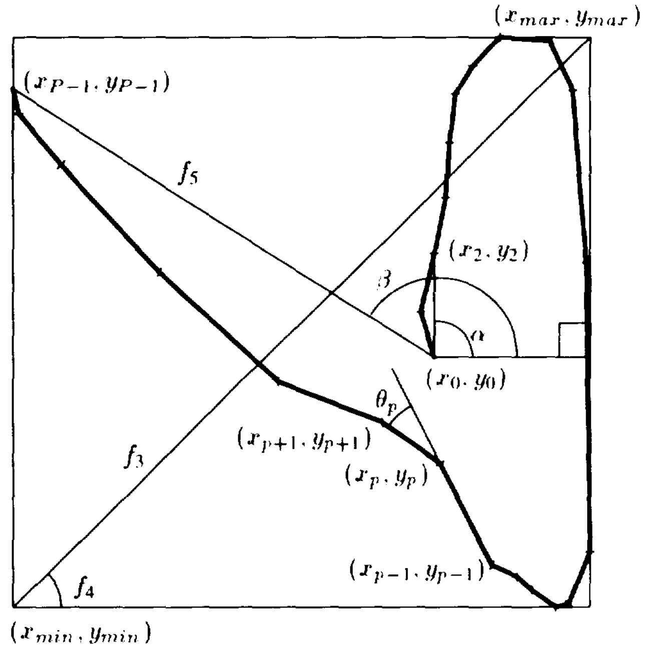
Statistical gesture recognition is done in two steps. First, a vector of features, $\mathbf{f} = [f_1, \ldots, f_F]$, is extracted from the input gesture. Then, the feature vector is classified as one of the C possible gestures via a linear machine.

4.1 The Features

Features were chosen according to the following criteria. Each feature should be incrementally computable in constant time per input point, which allows arbitrarily large gestures to be handled as efficiently as small ones. Since the classification algorithm performs poorly when a class has a feature with a multimodal distribution, a small change in the input should result in a correspondingly small change in each feature. Each feature should be meaningful so that is can be used in gesture semantics as well as for recognition. Finally, there should be enough features to provide differentiation between all gestures that might reasonably be expected, but, for efficiency reasons, there should not be too many.

Figure 6 shows the actual features used, both geometrically and algebraically. The features are the cosine (f_1) and the sine (f_2) of the initial angle of the gesture, the length (f_3) and the angle (f_4) of the bounding box diagonal, the distance (f_5) between the first and the last point, the cosine (f_6) and the sine (f_7) of the angle between the first and last point, the total gesture length (f_8), the total angle traversed (f_9), the sum of the absolute value of the angle at each mouse point (f_{10}), the sum of the squared value of those angles (f_{11}), the maximum speed (squared) of the gesture (f_{12}), and the duration of the gesture (f_{13}).

An angle's cosine and sine are used as features rather than the angle itself to avoid a discontinuity as the angle passes through 2π and wraps to 0. The "sharpness" feature, f_{11}, is needed to distinguish between smooth gestures and those with sharp angles, e.g. "U" and "V." Features f_{12} and f_{13} add a dynamic component so that gestures are not simply static pictures. Some applications may wish to disable these two features. The initial angle features, f_1 and f_2, are computed from the first and third mouse point because the result is generally less noisy than when computed from the first two points.



$$f_1 = \cos\alpha = (x_2 - x_0)/\sqrt{(x_2-x_0)^2 + (y_2-y_0)^2}$$

$$f_2 = \sin\alpha = (y_2 - y_0)/\sqrt{(x_2-x_0)^2 + (y_2-y_0)^2}$$

$$f_3 = \sqrt{(x_{max} - x_{min})^2 + (y_{max} - y_{min})^2}$$

$$f_4 = \arctan \frac{y_{max} - y_{min}}{x_{max} - x_{min}}$$

$$f_5 = \sqrt{(x_{P-1} - x_0)^2 + (y_{P-1} - y_0)^2}$$

$$f_6 = \cos\beta = (x_{P-1} - x_0)/f_5$$

$$f_7 = \sin\beta = (y_{P-1} - y_0)/f_5$$

Let $\Delta x_p = x_{p+1} - x_p \qquad \Delta y_p = y_{p+1} - y_p$

$$f_8 = \sum_{p=0}^{P-2} \sqrt{\Delta x_p^2 + \Delta y_p^2}$$

Let $\theta_p = \arctan \dfrac{\Delta x_p \Delta y_{p-1} - \Delta x_{p-1} \Delta y_p}{\Delta x_p \Delta x_{p-1} + \Delta y_p \Delta y_{p-1}}$

$$f_9 = \sum_{p=1}^{P-2} \theta_p$$

$$f_{10} = \sum_{p=1}^{P-2} |\theta_p|$$

$$f_{11} = \sum_{p=1}^{P-2} \theta_p^2$$

Let $\Delta t_p = t_{p+1} - t_p$

$$f_{12} = \max_{p=0}^{P-2} \frac{\Delta x_p^2 + \Delta y_p^2}{\Delta t_p^2}$$

$$f_{13} = t_{P-1} - t_0$$

Figure 6: Features used to identify strokes

The aforementioned feature set was empirically determined by the author to work well on a number of different gesture sets. Unfortunately, there are cases in which the features fail to distinguish between obviously different gestures (e.g. because the features take no account of the ordering of angles in a gesture). In such cases an additional feature may be added to discriminate between the thus far indistinguishable gestures. The extensibility of the feature set is a potential advantage that this statistical gesture recognition algorithm has over most known methods for online character recognition [21].

4.2 Gesture Classification

Given the feature vector **f** computed for an input gesture g, the classification algorithm is quite simple and efficient. Associated with each gesture class is a linear evaluation function over the features. Gesture class c has weights $w_{\hat{c}i}$ for $0 \leq i \leq F$, where F is the number of features, currently 13. (Per-gesture-class variables are written with hatted subscripts to indicate their class.) The evaluations, $v_{\hat{c}}$, are calculated as follows:

$$v_{\hat{c}} = w_{\hat{c}0} + \sum_{i=1}^{F} w_{\hat{c}i} f_i \qquad 0 \leq c < C \qquad (1)$$

The classification of gesture g is simply the c which maximizes $v_{\hat{c}}$. The possibility of rejecting g is discussed in section 4.4.

4.3 Training

Practitioners of pattern recognition will recognize this as the classic linear discriminator [5]. The training problem is to determine the weights $w_{\hat{c}i}$ from the example gestures. Iterative techniques were avoided to get efficient training. Instead, a well-known closed formula is used. The formula is known to produce optimal classifiers under certain rather strict normality assumptions on the per-class distributions of feature vectors. Even though these assumptions generally do not hold in practice, the formula still produces good classifiers.

Let $f_{\hat{c}ei}$ be the i^{th} feature of the e^{th} example of gesture class c, $0 \leq e < E_{\hat{c}}$, where $E_{\hat{c}}$ is the number of training examples of class c. The sample estimate of the mean feature vector per class, $\overline{\mathbf{f}}_{\hat{c}}$, is simply the average of the features in the class:

$$\overline{f}_{\hat{c}i} = \frac{1}{E_{\hat{c}}} \sum_{e=0}^{E_{\hat{c}}-1} f_{\hat{c}ei}$$

The sample estimate of the covariance matrix of class c, $\Sigma_{\hat{c}ij}$, is computed as:

$$\Sigma_{\hat{c}ij} = \sum_{e=0}^{E_{\hat{c}}-1} (f_{\hat{c}ei} - \overline{f}_{\hat{c}i})(f_{\hat{c}ej} - \overline{f}_{\hat{c}j})$$

(For convenience in the next step, the usual $1/(E_{\hat{c}} - 1)$ factor has not been included in $\Sigma_{\hat{c}ij}$.) The $\Sigma_{\hat{c}ij}$ are averaged to yield $\mathbf{\Sigma}_{ij}$, an estimate of the common covariance matrix.

$$\mathbf{\Sigma}_{ij} = \frac{\sum_{c=0}^{C-1} \Sigma_{\hat{c}ij}}{-C + \sum_{c=0}^{C-1} E_{\hat{c}}} \qquad (2)$$

The sample estimate of the common covariance matrix $\mathbf{\Sigma}_{ij}$ is then inverted. The result is denoted $(\mathbf{\Sigma}^{-1})_{ij}$. The weights $w_{\hat{c}j}$ are computed from the estimates as follows:

$$w_{\hat{c}j} = \sum_{i=1}^{F} (\mathbf{\Sigma}^{-1})_{ij} \overline{f}_{\hat{c}i} \qquad 1 \leq j \leq F$$

$$w_{\hat{c}0} = -\frac{1}{2} \sum_{i=1}^{F} w_{\hat{c}i} \overline{f}_{\hat{c}i}$$

A singular matrix can usually be handled by discarding a subset of the features.

4.4 Rejection

A linear classifier will always classify a gesture g as one of the C gesture classes. This section presents methods for rejecting ambiguous gestures and outliers.

Intuitively, if there is a near tie for the maximum per-class evaluation function $v_{\hat{i}}$ the gesture is ambiguous. Given a gesture g with feature vector **f** classified as class i (i.e. $v_{\hat{i}} > v_{\hat{j}}$ for all $j \neq i$)

$$\tilde{P}(\hat{i} \mid g) = \frac{1}{\sum_{j=0}^{C-1} e^{(v_{\hat{j}} - v_{\hat{i}})}}$$

is an estimate of the probability that g was classified correctly. Rejecting gestures in which $\tilde{P}(\hat{i} \mid g) < 0.95$ works well in practice.

The Mahalanobis distance [5] can be used to determine number of standard deviations a gesture g is away from the mean of its chosen class i.

$$\delta^2 = \sum_{j=1}^{F} \sum_{k=1}^{F} (\mathbf{\Sigma}^{-1})_{jk} (f_j - \overline{f}_{\hat{i}j})(f_k - \overline{f}_{\hat{i}k})$$

Rejecting gestures for which $\delta^2 > \frac{1}{2}F^2$ eliminates the obvious outliers. Unfortunately, this thresholding also tends to reject many seemingly good gestures, making it less than ideal.

Generally, a gesture-based system will ignore a rejected gesture, and the user can immediately try the gesture again. In contrast, the effect of a misclassified gesture will typically be undone before the gesture is retried. If undo is quick

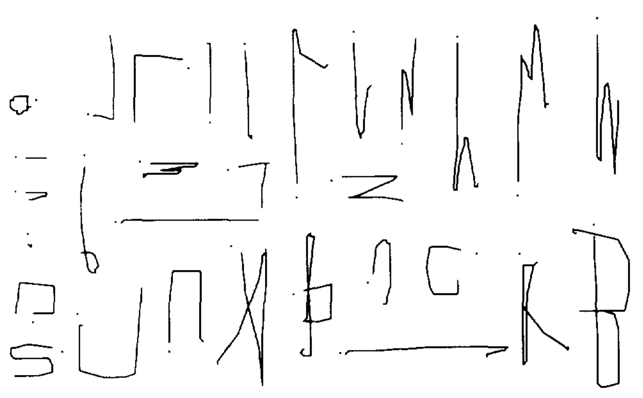


Figure 7: GSCORE gesture set used for evaluation (a period marks the start of each gesture).

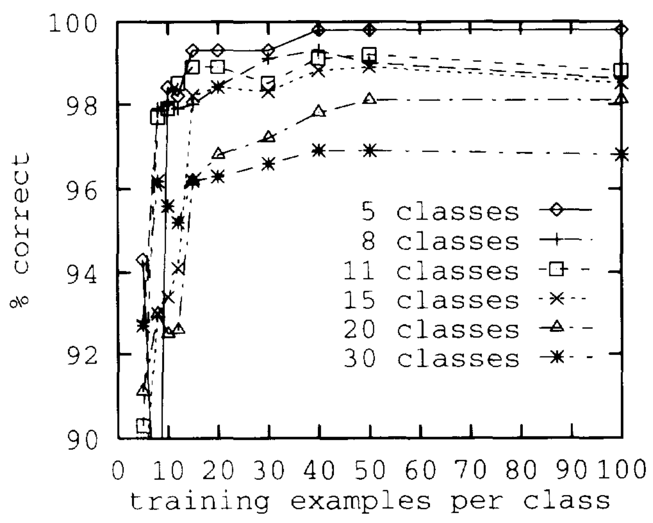


Figure 8: Recognition rate vs. training set size.

and easy, the time spent retrying the gesture will dominate. Since rejection increases the number of gestures that need to be redone (because inevitably some gestures that would have been classified correctly will be rejected), rejection should be disabled in applications with good undo facilities. Of course in applications without undo, rejection is preferable to misclassification and should be enabled.

4.5 Evaluation

Despite their simplicity, classifiers trained using this algorithm usually perform quite well in practice. Performance has been evaluated on 10 different gesture sets. Figure 8 shows some typical results for the gesture set shown in Figure 7. The gestures are from GSCORE, an editor for musical scores. The plot shows the recognition rate as a function of the number of training examples per class for various subsets of the GSCORE gestures. In the cases where 15 or fewer gesture classes are recognized by a classifier trained with 15 or more examples per class, at least 98% of the test gestures are classified correctly. The 30 class classifier trained with 40 examples per class has a 97% recognition rate. Recognition dropped to 96% when given only 15 training examples per class. Many of the misclassifications can be attributed to poor mouse tracking.

Figure 9 shows the recognition rate for five gesture sets. Each set was trained on fifteen examples per class and evaluated on 50 test gestures per class. In all cases the author was the gesturer; preliminary evaluations on other subjects show comparable performance.

On a DEC MicroVAX II, the classifier spends 0.2 milliseconds per mouse point calculating the feature vector, and then 0.3 msec per class to do the classification (8 msec to choose between 30 classes). Training time is 4 msec per training example, 80 msec to compute and invert the covariance matrix, and 5 msec per class to compute the weights. The per-mouse point and per-gesture calculations are done incrementally as the gesture is entered and thus never noticed by the user. Performance improves by a factor of 12 on a DEC PMAX-3100. In short, the classification time is negligible and the training is fast enough to be done in response to user input, such as the first time a gesture is made over a particular view class.

5 Extensions

Versions of GDP utilizing eager recognition and multi-finger recognition have been built by the author to demonstrate the feasibility of the concepts. Unfortunately, space limitations preclude a thorough discussion. For more details, the reader is again referred to [20].

5.1 Eager Recognition

Eager recognition refers to the recognition of gestures as soon as they are unambiguous. The author's approach [19, 20] uses the basic stroke recognition algorithm to classify subgestures (gestures in progress) as ambiguous or unambiguous. Note that classification occurs on every mouse point.

In GDP, a user presses a mouse button, enters the "L" gesture, stops and waits for a rectangle to appear (while still holding the button), and then manipulates one of the rectangle's corners. Eager recognition eliminates the stop: the system recognizes the rectangle gesture *while the user is making it*, and then creates the rectangle, allowing the user to drag the corner. What begins as a gesture changes into a rubberbanding interaction with no explicit indication from the user.

5.2 Multi-finger recognition

Recognizing gestures made with multiple fingers simultaneously has recently become of interest due to the availability of new input devices, including multi-finger touch pads [10],

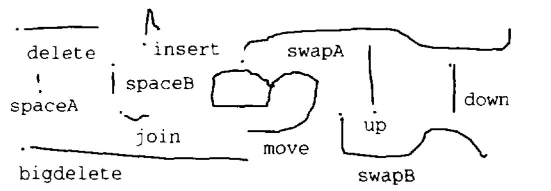
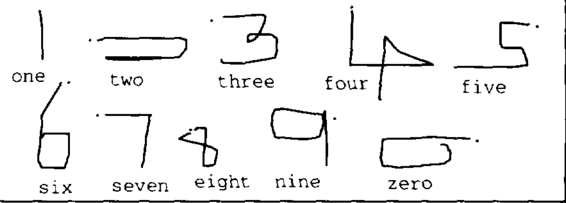
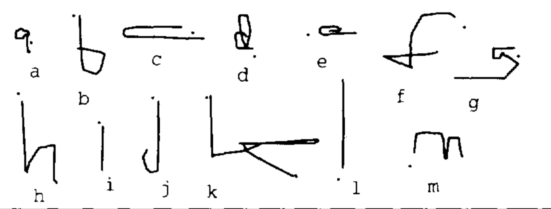
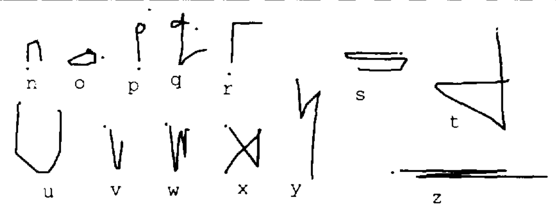
Set Name	Gesture Classes	Number of Classes	Recognition Rate
Coleman	 delete insert swapA spaceA spaceB join down up move bigdelete swapB	11	100.0%
Digits	 one two three four five six seven eight nine zero	10	98.5%
Let:a-m	 a b c d e f g h i j k l m	13	99.2%
Let:n-z	 n o p q r s t u v w x y z	13	98.4%
Letters	Union of Let:a-m and Let:n-z	26	97.1%

Figure 9: Recognition rates for various gesture sets.

the VPL DataGlove [22], and the Sensor Frame [12]. By treating the multi-finger input as multi-path data (e.g. the paths of the fingertips) the single-stroke recognition algorithm may be applied to each of the paths individually and the results combined to classify the multi-path gesture. A decision tree is used to combine the single-stroke classifications, and a set of global features is used to discriminate between different multi-path gestures whose corresponding paths are indistinguishable.

Note that the stroke recognition cannot immediately be applied to DataGlove finger paths, because the DataGlove has no way of indicating the start of a gesture, and also because the paths are three dimensional. This is one area for future work.

6 Conclusion and Future Directions

This paper described GRANDMA, a tool that dramatically reduces the effort involved in creating a gesture-based interface to an application. Starting with an application with a traditional direct manipulation interface, GRANDMA lets the designer specify gestures by example, associate those gestures with views in the interface, and specify the effect each gesture has on its associated views through a simple programming interface. Since the attributes of the gesture are available for use as parameters to application routines, a single gesture can be very powerful.

Some parameters of application commands are best determined at the time the gesture is recognized; others require subsequent manipulation and feedback to determine. This is the motivation behind the two-phase interaction technique that combines gesturing and direct manipulation. After recognition the user can manipulate additional parameters as long as the mouse button remains pressed. Eager recognition smooths the transition from gesturing to manipulation.

The foundation of this work is a new algorithm for recognizing single-stroke gestures specified by example. The combination of a meaningful, extensible feature set and well-understood statistical pattern recognition techniques appears to be flexible enough to evolve beyond two-dimensional single-stroke gesture recognition into the gesture recognizers of the future. The recognition technology is in no way dependent on the GRANDMA toolkit and its integration into other systems is strongly encouraged.

Based on the experience with GRANDMA, gestures are now being integrated into the NeXT Application Kit [16], the Andrew Toolkit [17], and Garnet [14]. This should allow gestural interfaces to be added to existing applications, enabling further use and study of this promising input technique.

Acknowledgements

I wish to thank CMU's School of Computer Science and CMU's Information Technology Center for their support of this work. I am also grateful to Roger Dannenberg and some anonymous reviewers for their helpful criticism of an earlier draft of this paper. Special thanks goes to Klaus Gross, whose detailed comments greatly improved this paper.

References

[1] BUXTON, W. Chunking and phrasing and the design of human-computer dialogues. In *Information Processing 86* (North Holland, 1986), Elsevier Science Publishers B.V.

[2] BUXTON, W., SNIDERMAN, R., REEVES, W., PATEL, S., AND BAECKER, R. The evolution of the SSSP score-editing tools. In *Foundations of Computer Music*, C. Roads and J. Strawn, Eds. MIT Press, Cambridge, Mass., 1985, ch. 22, pp. 387–392.

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

[4] COX, B. J. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[5] DUDA, R., AND HART, P. *Pattern Classification and Scene Analysis*. Wiley Interscience, 1973.

[6] GUYON, I., ALBRECHT, P., CUN, Y. L., DENKER, J., AND HUBBARD, W. Design of a neural network character recognizer for a touch terminal. *Pattern Recognition* (forthcoming).

[7] HENRY, T., HUDSON, S., AND NEWELL, G. Integrating gesture and snapping into a user interface toolkit. In *UIST '90* (1990), ACM, pp. 112–122.

[8] KRASNER, G. E., AND POPE, S. T. A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming 1*, 3 (Aug. 1988), 26–49.

[9] KURTENBACH, G., AND BUXTON, W. GEdit: A test bed for editing by contiguous gestures. To be published in SIGCHI Bulletin, 1991.

[10] LEE, S., BUXTON, W., AND SMITH, K. A multi-touch three dimensional touch tablet. In *Proceedings of CHI'85 Conference on Human Factors in Computing Systems* (1985), ACM, pp. 21–25.

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

[12] MCAVINNEY, P. Telltale gestures. *Byte 15*, 7 (July 1990), 237–240.

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

[14] MYERS, B. A., GIUSE, D., DANNENBERG, R. B., ZANDEN, B. V., KOSBIE, D., PERVIN, E., MICKISH, A., AND MARCHAL, P. Comprehensive support for graphical, highly-interactive user interfaces: The Garnet user interface development environment. *IEEE Computer 23*, 11 (Nov 1990).

[15] NEWMAN, W., AND SPROULL, R. *Principles of Interactive Computer Graphics*. McGraw-Hill, 1979.

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

[17] PALAY, A., HANSEN, W., KAZAR, M., SHERMAN, M., WADLOW, M., NEUENDORFFER, T., STERN, Z., BADER, M., AND PETERS, T. The Andrew toolkit: An overview. In *Proceedings of the USENIX Technical Conference* (Dallas, February 1988), pp. 11-23.

[18] RHYNE, J. R., AND WOLF, C. G. Gestural interfaces for information processing applications. Tech. Rep. RC12179, IBM T.J. Watson Research Center, IBM Corporation, P.O. Box 218, Yorktown Heights, NY 10598, Sept. 1986.

[19] RUBINE, D. Integrating gesture recognition and direct manipulation. In *Proceedings of the Summer '91 USENIX Technical Conference* (1991).

[20] RUBINE, D. *The Automatic Recognition of Gestures*. PhD thesis, School of Computer Science, Carnegie Mellon University, forthcoming, 1991.

[21] SUEN, C., BERTHOD, M., AND MORI, S. Automatic recognition of handprinted characters: The state of the art. *Proceedings of the IEEE 68*, 4 (April 1980), 469–487.

[22] ZIMMERMAN, T., LANIER, J., BLANCHARD, C., BRYSON, S., AND HARVILL, Y. A hand gesture interface device. *CHI+GI* (1987), 189–192.