EXHIBIT 4.20

operators *left-of*, *above*, and *surrounds* to describe the relationships between strokes of Chinese characters.

A major problem with linguistic recognizers is the necessity of supplying a grammar for each pattern class. This usually represents considerably more effort than simply supplying examples for each class. While some research has been done on automatically deriving grammars from examples, this research appears not to be sufficiently advanced to be of use in a gesture recognition system. Also, linguistic systems are best for patterns with substantial internal structure, while gestures tend to be atomic (but not always [75]).

Connectionism.

Pattern recognition based on neural nets has received much research attention recently [65, 104, 132, 134]. A neural net is a configuration of simple processing elements, each of which is a super-simplified version of a neuron. A number of methods exist for training a neural network pattern recognizer from examples. Almost any of the different kinds of features listed above could serve as input to a neural net, though best results would likely be achieved with vectors of quantitative features. Also, some statistical discrimination functions may be implemented as simple neural networks.

Neural nets have been applied successfully to the recognition of line drawings [55, 82], characters [47], and DataGlove gestures [34]. Unfortunately, they tend to require a large amount of processing power, especially to train. It now appears likely that neural networks will, in the future, be a popular method for gesture recognition. The chief advantage is that neural nets, like template-based approaches, are able to take the raw sensor data as input. A neural network can learn to extract interesting features for use in classification. The disadvantage is that many labeled examples (often thousands) are needed in training.

The statistical classification method discussed in this dissertation may be considered a one-level neural network. It has the advantage over multilayer neural networks, in that it may be trained quickly using relatively few examples per class (typically 15). Rapid training time is important in a system that is used for prototyping gesture-based systems, since it allows the system designer to easily experiment with different sets of gestures for a given application.

Ad hoc methods.

If the set of patterns to be recognized is simple enough, a classifier may be programmed by hand. Indeed, this was the case in many of the gesture-based systems mentioned in Section 2.2. Even so, having to program a recognizer by hand can be difficult and makes the gesture set difficult to modify. The author believes that the difficulty of creating recognizers is one major reason why more gesture-based systems have not been built, and why there is a dearth of experiments which study the effect of varying the gestures in those systems which have been built. The major goal of this dissertation is to make the building of gesture-based systems easy by making recognizers specifiable by example, and incorporating them into an easy-to-use direct manipulation framework.

2.4 Direct Manipulation Architectures

A direct manipulation system is one in which the user manipulates graphical representations of objects in the task domain directly, usually with a mouse or other pointing device. In the words of Shneiderman [117],

> the central ideas seemed to be visibility of the object of interest; rapid, reversible, incremental actions; and replacement of complex command language syntax by direct manipulation of the object of interest–hence the term "direct manipulation."

As examples, he mentions display editors, Visicalc, video games, computer-aided design, and driving an automobile, among others.

For many application domains, the direct manipulation paradigm results in programs which are easy to learn and use. Of course there are tasks for which direct manipulation is not appropriate, due to the fact that the abstract nature of the task domain is not easily mapped onto concrete graphical objects [58]. For example, direct manipulation systems for the abstract task of programming have been rather difficult to design, though much progress has been made [98].

It is not intended here to debate the merits and drawbacks of direct manipulation systems. Instead, it is merely noted that direct manipulation has become an increasingly important and popular style of user interface. Furthermore, all existing gesture-based systems may be considered direct-manipulation systems. The reason is that graphical objects on the screen are natural targets of gesture commands, and updating those objects is an intuitive way of feeding back to the user the effect of his gesturing. In this section, existing approaches for constructing direct manipulation systems are reviewed. In Chapters 6 and 7 it is shown how some of these approaches may be extended to incorporate gestural input.

While direct manipulation systems are easy to use, they are among the most difficult kinds of interface to construct. Thus, there is a great interest in software tools for creating such interfaces. Myers [96] gives an excellent overview of the various tools which have been proposed for this purpose. Here, it is sufficient to divide user-interface software tools into three levels.

The lowest software level potentially seen by the direct manipulation system programmer is usually the *window manager*. Example window managers include X [113], News [127], Sun Windows [126], and Display Postscript [102]; see Myers [94] for an overview. For current purposes, it is sufficient to consider the window manager as providing a set of routines (*i.e.* a programming interface) for both output (textual and graphical) and input (keyboard and mouse or other device). Programming direct manipulation interfaces at the window manager level is a usually avoided, since a large amount of work will likely need to be redone for each application (*e.g.* menus will have to be implemented for each). Building from scratch this way will probably result in different and inconsistent interfaces for each application, making the total system difficult to recall and use.

The next software level is the *user interface toolkit*. Toolkits come in two forms: non-object-oriented and object-oriented. A toolkit provides a set of procedures or objects for constructing menus, scroll bars, and other standard interaction techniques. Most of the toolkits come totally disassembled, and it is up to the programmer to decide how to use the components. Some toolkits, notably MacApp [115] and GWUIMS [118], come partially assembled, making it easier for the programmer to customize the structure to fit the application. For this reason, some authors have referred to these systems as User Interface Management Systems, though here they are grouped with the other toolkits.

A non-object-oriented toolkit is simply a set of procedures for creating and manipulating the interaction techniques. This saves the programmer the effort involved in programming these interaction techniques directly, and has the added benefit that all systems created using a single toolkit will look and act similarly. One problem with non-object-oriented toolkits is that they usually do not give much support for the programmer who wishes to create new interaction techniques. Such a programmer typically cannot reuse any existing code and thus finds himself bogged down with many low-level details of input and screen management.

Instead of procedures, object-oriented toolkits provide a class (an object type) for each of the standard interaction techniques. To use one of the interaction techniques in an interface, the programmer creates an instance of the appropriate class. By using the inheritance mechanism of the object-oriented programming language, the programmer can create new classes which behave like existing classes except for modifications specified by the programmer. This subclassing gives the programmer a method of customizing each interaction technique for the particular application. It also provides assistance to the programmer wishing to create new interaction techniques–he can almost always subclass an existing class, which is usually much easier than programming the new technique from scratch. One problem with object-oriented toolkits is their complexity; often the programmer needs to be familiar with a large part of the class hierarchy before he can understand the functionality of a single class.

User Interface Management Systems (UIMSs) form the software level above toolkits [96]. UIMSs are systems which provide a method for specifying some aspect of the user interface that is at a higher level than simply using the base programming language. For example, the RAPID/USE system [135] uses state transition diagrams to specify the structure of user input, the Syngraph system [64] uses context-free grammers similarly, and the Cousin system [51] uses a declarative language. Such systems encourage or enforce a strict separation between the user interface specification and the application code. While having modularity advantages, it is becoming increasingly apparent that such a separation may not be appropriate for direct manipulation interfaces [110].

UIMSs which employ *direct graphical specification* of interface components are becoming increasingly popular. In these systems, the UIMS is itself a direct manipulation system. The user interface designer thus uses direct manipulation to specify the components of the direct manipulation interface he himself desires to build. The NeXT Interface Builder [102] and the Andrew Development Environment Workbench (ADEW) [100] allow the placement and properties of existing interface components to be specified via direct manipulation. However, new interface components must be programmed in the object-oriented toolkit provided. In addition to the direct manipulation of existing interface components, Lapidary [93] and Peridot [90] enable new interface components to be created by direct graphical specification.

UIMSs are generally built on top of user interface toolkits. The UIMSs that support the construction of direct manipulation interfaces, such as the ones which use direct graphical specification, tend to be built upon object-oriented toolkits. Since object-oriented toolkits are currently the preferred vehicle for the creation of direct manipulation systems, this dissertation concentrates upon the problem of integrating gesture into such toolkits. In preparation for this, the architectures of several existing object-oriented toolkits are now reviewed.

2.4.1 Object-oriented Toolkits

The object-oriented approach is often used for the construction of direct manipulation systems. Using object-oriented programming techniques, graphical objects on the screen can be made to correspond quite naturally with software objects internal to the system. The ways in which a graphic object can be manipulated correspond to the messages to which the corresponding software object responds. It is assumed that the reader of this dissertation is familiar with the concepts of object-oriented programming. Cox [27, 28], Stefik and Bobrow [121], Horn [56], Goldberg and Robson [44], and Schmucker [115] all present excellent overviews of the topic.

The Smalltalk-80 system [44] was the first object-oriented system that ran on a personal computer with a mouse and bitmapped display. From this system emerged the Model-View-Controller (MVC) paradigm for developing direct manipulation interfaces. Though MVC literature is only now beginning to appear in print [70, 63, 68], the MVC paradigm has directly influenced every object-oriented user interface architecture since its creation. For this reason, the review of object-oriented architectures for direct manipulation systems begins with a discussion of the use of the MVC paradigm in the Smalltalk-80 system.

The terms "model," "view," and "controller" refer to three different kinds of objects which play a role in the representation of single graphic object in a direct manipulation interface. A *model* is an object containing application specific data. Model objects encapsulate the data and computation of the task domain, and generally make no reference to the user interface.

A *view* object is responsible for displaying application data. Usually, a view is associated with a single model, and communicates with the model in order to acquire the application data that it will render on the screen. A single model may have multiple views, each potentially displaying different aspects of the model. Views implement the "look" of a user interface.

A *controller* object handles user interaction (*i.e.* input). Depending on the input, the controller may communicate directly with a model, a view, or both. A controller object is generally paired with a view object, where the controller handles input to a model and the view handles output. Internally, the controller and view objects typically contain pointers to each other and the associated model, and thus may directly send messages to each other and the model. Controllers implement the "feel" of a user interface.

When the application programmer codes a model object, for modularity purposes he does not generally include references to any particular view(s). The result is a separation between the application (the models) and the user interface (the views and controllers). There does however need to be some connection from a model to a view–otherwise how can the view be notified when the state of the model changes? This connection is accomplished in a modular fashion through the use of *dependencies*.

Dependencies work as follows: Any object may register itself as a dependent of any other object. Typically, a view object, when first created, registers as a dependent of a model object. Generally, there is a list of dependents associated with an object; in this way multiple views may be dependent on a single model. When an object that potentially has dependents changes its state, it sends itself the message `[self changed]`. Each dependent `d` of the object will then get sent the message `[d` `update]`, informing it that an object upon which it is dependent has changed. Thus, dependencies allow a model to communicate to its views the fact that it has changed, without referring to the views explicitly.

Many views display rectangular regions on the screen. A view may have subviews, each of which typically results in an object displayed within the rectangular region of the parent view. The subviews may themselves have subviews, and so on recursively, giving rise to the *view hierarchy*. Typically, a subview's display is clipped so as to wholly appear within the rectangular region of its parent. A subview generally occludes part of its parent's view.

A common criticism of the MVC paradigm is that two objects (the view and controller) are needed to implement the user interface for a model where one would suffice. This, the argument goes, is not only inefficient, but also not modular. Why implement the look and feel separately when in practice they always go together?

The reply to this criticism states that it is useful (often or occasionally) to control look and feel separately [68]. Knolle discusses the usefulness of a single view having several interchangeable controllers; implementing different user abilities (*i.e.* beginning, intermediate, and advanced) with different controllers, and having the system adapt to the user's ability at runtime is one example. While Knolle's examples may not be very persuasive, there is an important application of separating views from controllers, namely, the ability to handle multiple input devices. Chapters 6 and 7 explore further the benefits accrued from the separation of views and controllers.

Nonetheless, there is a simplicity to be had by combining views and controllers into a single object, giving rise to object-oriented toolkits based on the Data-View (DV) paradigm. Though the terminology varies, MacApp [115, 114], the Andrew Toolkit [105], the NeWS Development environment [108], and InterViews [79] all use the DV paradigm. In this paradigm, data objects contain application specific data (and thus are identical to MVC models) while view objects combine the functionality of MVC view and controller objects. In DV systems, the look and feel of an object are very tightly coupled, and detailed assumptions about the input hardware (*e.g.* a three button mouse) get built into every view.

Object-oriented toolkits also vary in the method by which they determine which controller objects get informed of a particular input event, and also in the details of that communication. Typically, input events (such as mouse clicks) are passed down the view hierarchy, with a view querying its subviews (and so on recursively) to see if one of them wishes to handle the event before deciding to handle the event itself. Many variations on this scheme are possible.

Controllers may be written to have methods for messages such as `leftButtonDown`. This style, while convenient for the programmer, has the effect of wiring in details of the input hardware all throughout the system [115, 68]. The NeXT AppKit [102], passes input events to the controller object in a more general form. This is generalized even further in Chapters 6 and 7.

Controllers are a very general mechanism for handling input. Garnet [92], a modern MVC-based system, takes a different approach, called *interactors* [95, 91]. The key insight behind interactors is that there are only several different kinds of interactive behavior, and a (parameterizable) interactor can be built for each. The user-interface designer then needs only to choose the appropriate interactor for each interaction technique he creates.

Gestural input is not currently handled by the existing interactors. It would be interesting to see if the interactor concept in Garnet is general enough to handle a gesture interactor. Unfortunately, the author was largely unaware of the Garnet project at the time he began the research described in Chapters 6 and 7. Had it been otherwise, a rather different method for incorporating gestures into direct manipulation systems than the one described here might have been created.

The Artkit system [52] has a considerably more general input mechanism than the MVC systems discussed thus far. Like the GRANDMA system discussed in this dissertation, Artkit integrates gesture into an object-oriented toolkit. Though developed simultaneously and independently, Artkit and GRANDMA have startlingly similar input architectures. The two systems will be compared in more detail in chapter 6.

Chapter 3

Statistical Single-Path Gesture Recognition

3.1 Overview

This chapter address the problem of recognizing single-path gestures. A single-path gesture is one that can be input with a single pointer, such as a mouse, stylus, or single-finger touch pad. It is further assumed that the start and end of the input gesture are clearly delineated. When gesturing with a mouse, the start of a gesture might be indicated by the pressing of a mouse button, and the end by the release of the button. Similarly, contact of the stylus with the tablet or of a finger with the touch screen could be used to delineate the endpoints of a gesture.

Baecker and Buxton[5] warn against using a mouse as a gestural input device for ergonomic reasons. For the research described in this chapter, the author has chosen to ignore that warning. The mouse was the only pointing device readily available when the work began. Furthermore, it was the only pointing device that is widely available–an important consideration as it allows others to utilize the present work. In addition, it is probably the case that any trainable recognizer that works well given mouse input could be made to work even better on devices more suitable for gesturing, such as a stylus and tablet.

The particular mouse used is labeled DEC Model VS10X-EA, revision A3. It has three buttons on top, and a metal trackball coming out of the bottom. Moving the mouse on a flat surface causes its trackball to roll. Inside the mouse, the trackball motion is mechanically divided into x and y components, and the mouse sends a pulse to the computer each time one of its components changes by a certain amount. The windowing software on the host implements mouse acceleration, meaning that the faster the mouse is moved a given distance, the farther the mouse cursor will travel on the screen. The metal mouseball was rolled on a Formica table, resulting it what might be termed a "hostile" system for studying gestural input.

All the work described in this chapter was developed on a Digital Equipment Corporation MicroVAX II.[1] The software was written in C [66] and runs on top of the MACH operating system

[1]MicroVAX is trademark of Digital Equipment Corporation.

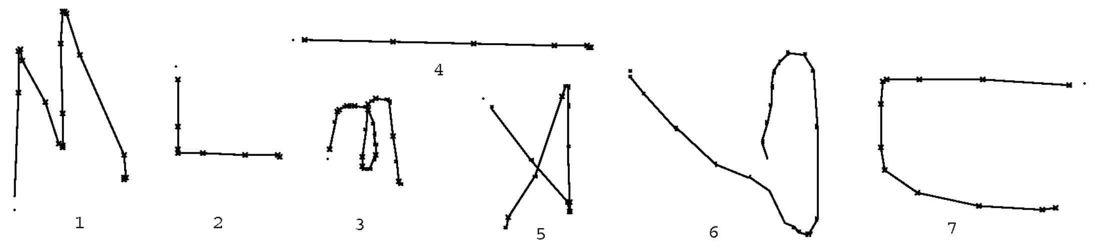

Figure 3.1: Some example gestures
The period indicates the start of the gesture. The actual mouse points that make up the gestures are indicated as well.

[131], which is UNIX[2] 4.3 BSD compatible. X10 [113] was the window system used, though there is a layer of software designed to make the code easy to port to other window systems.

3.2 Single-path Gestures

The gestures considered in this chapter consist of the two-dimensional path of a single point over time. Each gesture is represented as an array g of P time-stamped sample points:

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

The points are time stamped (the t_p) since the typical interface to many gestural input devices, particularly mice, does not deliver input points at regular intervals. In this dissertation, only two-dimensional gestures are considered, but the methods described may be generalized to the three-dimensional case.

When an input point is very close to the previous input point, it is ignored. This simple preprocessing of the input results in features that are much more reliable, since much of the jiggle, especially at the end of a gesture, is eliminated. The result is a large increase in recognition accuracy.

For the particular mouse used for the majority of this work, "very close" meant within three pixels. This threshold was empirically determined to produce an optimal recognition rate on a number of gesture sets.

Similar, but more complicated preprocessing was done by Leedham, *et. al.*, in their Pittman's shorthand recognition system[77]. The difference in preprocessing in Leedham's system and the current work stems largely from the difference in input devices (Leedham used an instrumented pen), indicating that preprocessing should be done on a per-input-device basis.

Figure 3.1 shows some example gestures used in the GDP drawing editor. The first point (g_0) in each gesture is indicated by a period. Each subsequent point (g_p) is connected by a line segment to the previous point (g_{p-1}). The time stamps are not shown in the figure.

The gesture recognition problem is stated as follows: There is a set of C gesture classes, numbered 0 through $C-1$. The classes may be specified by description, or, as is done in the present

[2]UNIX is a trademark of Bell Laboratories.

work, by example gestures for each class. Given an input gesture g, the problem is to determine the class to which g belongs (*i.e.* the class whose members are most like g). Some classifiers have a reject option: if g is sufficiently different so as not to belong to any of the gesture classes, it should be rejected.

3.3 Features

Statistical gesture recognition is done in two steps. First, a set of features is extracted from the input gesture. This set is represented as a feature vector, $\mathbf{f} = [f_1, \ldots, f_F]'$. (Here and throughout, the prime denotes vector transpose.) The feature vector is then classified as one of the possible gesture classes.

The set of features used was chosen according to the following criteria:

The number of features should be small. In the present scheme, the amount of time it takes to classify a gesture given the feature vector is proportional to the product of the size of the feature vector (*i.e.* the number of features) and the number of different gesture classes. Thus, for efficiency reasons, the number of features should be kept as small as possible while still being able to distinctly represent the different classes.

Each feature should be calculated efficiently. It is essential that the calculation of the feature vector itself not be too expensive: the amount of time to update the value of a feature when an input point g_p is received should be bounded by a constant. In particular, features that require all previous points to be examined for each new input point are disallowed. In this manner, very large gestures (those consisting of many points) are recognized as efficiently as smaller gestures.

In practice, this incremental calculation of features is often achieved by computing auxiliary features not used in classification. For example, if one feature is the average x value of the input points, an auxiliary feature consisting of the sum of the x values might be computed. This would require constant time (one addition) per input point. When the feature vector is needed (for classification) the average x value feature is computed in constant time by dividing the above sum by the number of input points.

Each feature should have a meaningful interpretation. Unlike simple handwriting systems, the gesture-based systems built here use the features not only for classification, but also for parametric information. For example, a drawing program might use the initial angle of a gesture to orient a newly created rectangle. While it is possible to extract such gestural attributes independent of classification, it is potentially less efficient to do so.

Meaningful features also provide useful information to the designer of a set of gesture classes for a particular application. By understanding the set of features, the designer has a better idea of what kind of gestures the system can and cannot distinguish; she is thus more likely to design gestures that can be classified accurately.

Individual features should have Gaussian-like distributions. The classifier described in this chapter is optimal when, among other things, within a given class each feature has a Gaussian distribution. This is because a class is essentially represented by its mean feature vector, and classification of an example takes place, to a first approximation, by determining the class whose mean feature vector is closest to the example's. Classification may suffer if a given feature in a given class has, for example, a bimodal distribution, whereby it tends toward one of two different values.

This requirement is satisfied when the feature is *stable*, meaning a small change in the input gesture results in a small change in the value of the feature. In general, this rules out features that are small integers, since presumably some small change in a gesture will cause a discrete unit step in the feature. When possible, features that depend on thresholds should also be avoided for similar reasons. Ideally, a feature is a real-valued continuous function of the input points.

Note that the input preprocessing is essentially a thresholding operation, and does have the effect that a seemingly small change in the gesture can cause big changes in the feature vector. However, eliminating this preprocessing would allow the noise inherent in the input device to seriously affect certain features. Thus, thresholding should not be ruled out per-se, but the tradeoffs must be considered. Another alternative is to use multiple thresholds to achieve a kind of multiscale representation of the input, thus avoiding problems inherent in using a single threshold [80].

The particular set of features used here evolved over the creation of two classifiers, the first being for a subset of GDP gestures, the second being a recognizer of upper-case letters, as handwritten by the author. In the current version of the recognition program, thirteen features are employed. Figure 3.2 depicts graphically the values used in the feature calculation.

The features are:

Cosine and sine of initial angle with respect to the X axis:

$$f_1 = \cos\alpha = (x_2 - x_0)/d$$
$$f_2 = \sin\alpha = (y_2 - y_0)/d$$

where $d = \sqrt{(x_2 - x_0)^2 + (y_2 - y_0)^2}$.

Length of the bounding box diagonal:

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

where $x_{max}, x_{min}, y_{max}, y_{min}$ are the maximum and minimum values for x_p and y_p respectively.

Angle of the bounding box:

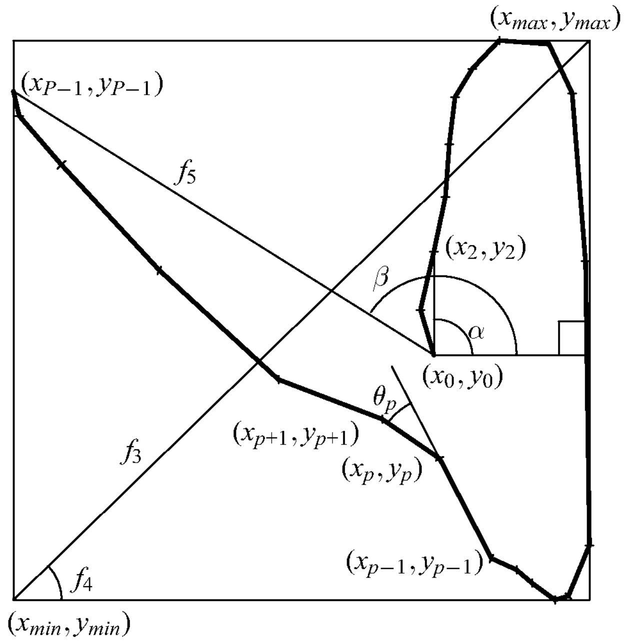

Figure 3.2: Feature calculation

Gesture 6 of figure 3.1 is shown with its relevant lengths and angles labeled with the intermediate variables used to compute features or the features themselves where possible.

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

Distance between first and last point:

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

Cosine and sine of angle between first and last point:

$$f_6 = \cos\beta = (x_{P-1} - x_0)/f_5$$
$$f_7 = \sin\beta = (y_{P-1} - y_0)/f_5$$

Total gesture length:

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

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

Total angle traversed (derived from the dot and cross product definitions[73]):

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

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

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

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

Maximum speed (squared):

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

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

Path duration:

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

Features f_{12} and f_{13} allow the gesture recognition to be based on temporal factors; thus gestures have a dynamic component and are not simply static pictures.

Some features (f_1, f_2, f_6, and f_7) are sines or cosines of angles, while others (f_5, f_{10}, f_{11}, f_{12}) depend on angles directly and thus require inverse trigonometric functions to compute. A four-quadrant arctangent is needed to compute θ_p; the arctangent function must take the numerator and denominator as separate parameters, returning an angle between $-\pi$ and π. For efficient recognition, it would be desirable to use just a single feature to represent an angle, rather than both the sine and cosine. However, the recognition algorithm requires that each feature have approximately a Gaussian distribution; this poses a problem when a small change in a gesture causes a large change in angle measurement due to the discontinuity when near $\pm\pi$. This mattered for initial angle, and the angle between the start and end point of the gesture, so each of these angles is represented by its sine and cosine. The bounding box angle is always between 0 and $\pi/2$ so there was no discontinuity problem for it.

For features dependent on θ_i, the angle between three successive input points, the discontinuity only occurs when the gesture stroke turns back upon itself. In practice, likely due to the few gestures used which have such changes, the recognition process has not been significantly hampered by the potential discontinuity (but see Section 9.1.1). The feature f_9 is a measure of the total angle traversed; in a gesture consisting of two clockwise loops, this feature might have a value near 4π. If the gesture was a clockwise loop followed by a counterclockwise loop, f_9 would be close to zero. The feature f_{10} accumulates the absolute value of instantaneous angle; in both loop gestures, its value would be near 4π. The feature f_{11} is a measure of the "sharpness" of gesture.

Figure 3.3 shows the value of some features as a function of p, the input point, for gestures 1 and 2 of figure 3.1. Note in particular how the value for f_{11} (the sharpness) increases at the angles of the gesture. The feature values at the last (rightmost) input point are the ones that are used to classify the gesture. The intent of the graph is to show how the features change with each new input point.

All the features can be computed incrementally, with a constant amount of work being done for each new input point. By utilizing table lookup for the square root and inverse trig functions, the amount of computation per input point can be made quite small.

A number of features were tried and found not to be as good as the features used. For example, instead of the sharpness metric f_{11}, initially a count of the number of times θ_p exceeded a certain threshold was used. The idea was to count sharp angles. While this worked fairly well, the more continuous measure of sharpness was found to give much better results. In general, features that are discrete counts do not work as well as continuous features that attempt to quantify the same phenomena. The reason for this is probably that continuous features more closely satisfy the normality criterion. In other words, an error or deviation in a discrete count tends to be much more significant than an error or deviation in continuous metric.

Appendix A shows the C code for incrementally calculating the feature vector of a gesture.

3.4 Gesture Classification

Given the feature vector $\mathbf{x}$ 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 $w_i^{\hat{c}}$ for $0 \le i \le F$, where F is the number of features, currently 13. (Per-class variables will be written using superscripts with hats to indicate the class. These are not and should not to be confused with exponentiation.) The evaluation functions are calculated as follows:

$$v^{\hat{c}} = w_0^{\hat{c}} + \sum_{i=1}^{F} w_i^{\hat{c}} x_i \qquad 0 \le c < C \tag{3.1}$$

The value $v^{\hat{c}}$ is the evaluation of class c. The classifier simply determines the c for which $v^{\hat{c}}$ is a maximum; this c is the classification of the gesture g. The possibility of rejecting g is discussed in Section 3.6.

Practitioners of pattern recognition will recognize this classifier as the classic linear discriminator [35, 30, 62, 74]. With the correct choice of weights $w_i^{\hat{c}}$, the linear discriminator is known to be optimal when (1) within a class the feature vectors have a multivariate normal distribution, and (2)

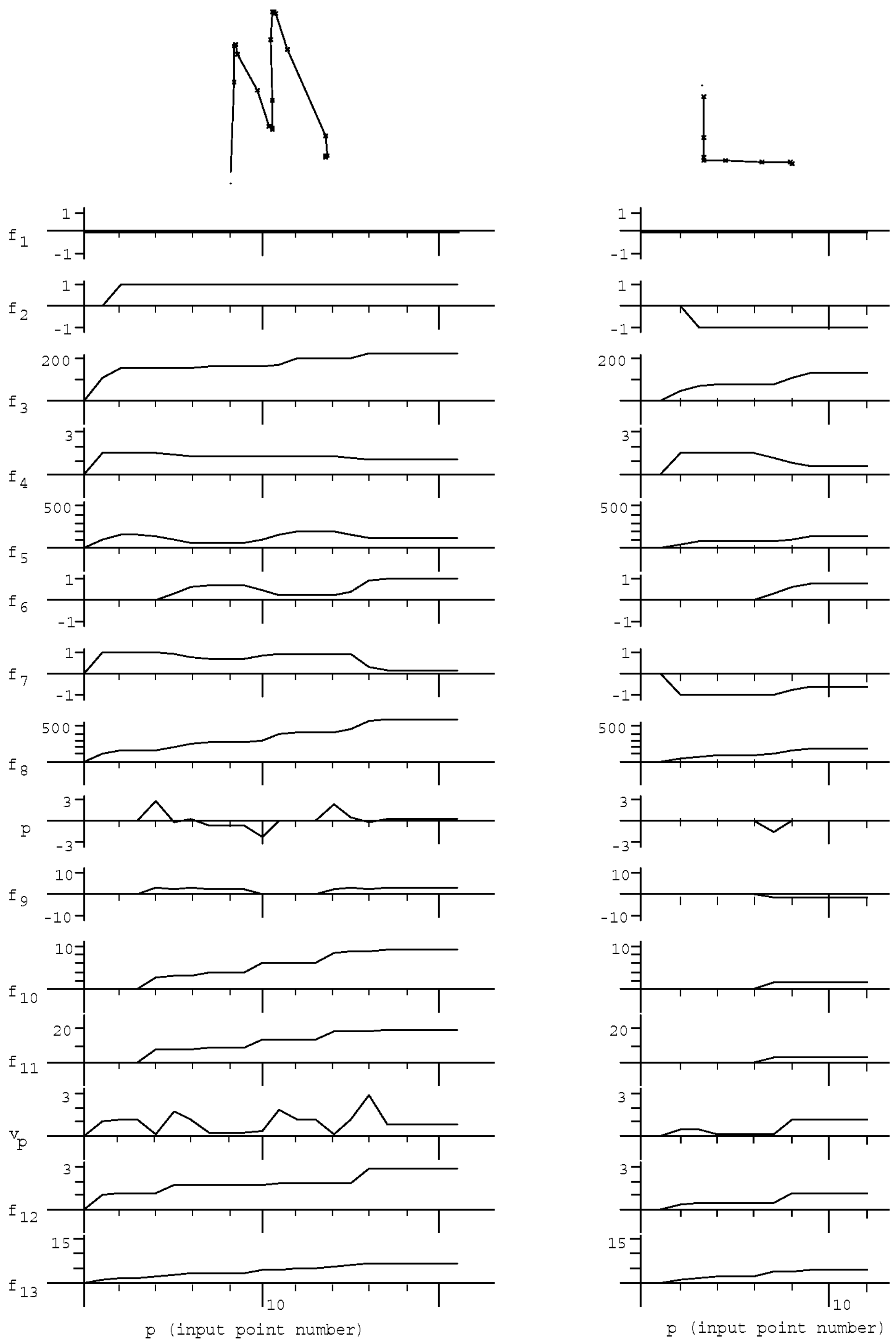

Figure 3.3: Feature vector computation

These graphs show how feature vectors change with each new input point. The left graphs refer to features of gesture 1 of 3.1 (an "M"), the right graphs to gesture 2 (an "L"). The final values of the features ($p = 21$ for gesture 1, $p = 12$ for gesture 2) are the ones used for classification. The instantaneous angle θ_p and velocity v_p have been included in the figure, although they are not part of the feature vector.

the per class feature covariance matrices are equal. (Exactly what this means is discussed in the next section. Other continuous distributions for which linear discriminant functions are optimal are investigated by Cooper [26].) These conditions do not hold for most sets of gesture classes given the feature set described; thus weights calculated assuming these conditions will not be optimal, even among linear classifiers (and even the optimal linear classifier can be outperformed by some non-linear classifiers if the above conditions are not satisfied). However, given the above set of features, linear discriminators computed as if the conditions are valid have been found to perform quite acceptably in practice.

3.5 Classifier Training

Once the decision has been made to use linear discriminators, the only problem that remains is the determination of the weights from example gestures of each class. This is known as the training problem.

Two methods for computing the weights were tried. The first was the multiclass perceptron training procedure described in Sklansky and Wassel[119]. The hope was that this method, which does not depend on the aforementioned conditions to choose weights, might perform better than methods that did. In this method, an initial guess of the weights was made, which are then used to classify the first example. For each class whose evaluation function scored higher than the correct class, each weight is reduced by an amount proportional to the corresponding feature of the example, while the correct class has its weights increased by the same amount. This is similar to back-propagation learning procedures in neural nets [34]. In this manner, all the examples are tried, multiple times if desired.

This method has the advantage of being simple, as well as needing very few example gestures to achieve reasonable results. However, the behavior of the classifier depends on the order in which the examples are presented for training, and good values for the initial weights and the constant of proportionality are difficult to determine in advance but have a large effect on the success and training efficiency of the method. The number of iterations of the examples is another variable whose optimum value is difficult to determine. Perhaps the most serious problem is that a single bad example might seriously corrupt the classifier.

Eventually, the perceptron training method was abandoned in favor of the plug-in estimation method. The plug-in estimation method usually performs approximately equally to the best perceptron-trained classifiers, and has none of the vagueness associated with perceptron training. In this method, the means of the features for each class are estimated from the example gestures, as is the common feature covariance matrix (all classes are assumed to have the same one). The estimates are then used to approximate the linear weights that would be optimal assuming the aforementioned conditions were true.

3.5.1 Deriving the linear classifier

The derivation of the plug-in classifier is given in detail in James [62]. James' explanation of the derivation is particularly good, though unfortunately the derivation itself is riddled with typos and other errors. Krzanowski [74] gives a similar derivation (with no errors), as well as a good general description of multivariate analysis. The derivation is summarized here for convenience.

Consider the class of "L" gestures, drawn starting from the top-left. One example of this class is gesture 2 in figure 3.1. It is easy to generate many more examples of this class. Each one gives rise to a feature vector, considered to be a column vector of F real numbers $(f_0, \ldots, f_{F-1})'$.

Let f be the random vector (*i.e.* a vector of random variables) representing the feature vectors of a given class of gestures, say "L" gestures. Assume (for now) that f has a *multivariate normal* distribution. The multivariate normal distribution is a generalization to vectors of the normal distribution for a single variable. A single variable (univariate) normal distribution is specified by its mean value and variance. Analogously, a multivariable normal distribution is specified by its mean vector, $\overline{\mu}$, and covariance matrix, Σ. In a multivariate normal distribution, each vector element (feature) has a univariate normal distribution, and the mean vector is simply a vector consisting of the means of the individual features. The variance of the features form the diagonal of the covariance matrix; the off-diagonal elements represent correlations between features.

The univariate normal distribution has a density function which is the familiar bell-shaped curve. The analog in the two variable (bivariate) case is a three-dimensional bell shape. In this case, the lines of equal probability (cross sections of the bell) are concentric ellipses. The axes of the ellipses are parallel to the feature axes if and only if the variables are uncorrelated. By analogy, in the higher dimensional cases, the distribution has a hyper-bell shape, and the equiprobability hypersurfaces are ellipsoids.

A more in-depth discussion of the properties of the multivariate normal distribution would take us too far afield here. The reader unfamiliar with the subject is asked to rely on the analogy with the univariate case, or to refer to a good text, such as Krzanowski [74].

The multivariate normal probability density function is the multivariate analog to the bell-shaped curve. It is written here as a conditional probability density, *i.e.* the density of the probability of getting vector $\mathbf{x}$ given $\mathbf{x}$ comes from multivariate distribution L with F variables, mean $\overline{\mu}$, and covariance matrix Σ.

$$p(\mathbf{x} \mid L) = (2\pi)^{-F/2} |\Sigma|^{-1/2} e^{-\frac{1}{2}(\mathbf{x}-\overline{\mu})' \Sigma^{-1} (\mathbf{x}-\overline{\mu})} \tag{3.2}$$

Note that this expression involves both the determinant and the inverse of the covariance matrix. The interested reader should verify that it reduces to the standard bell-shaped curve in the univariate case ($F = 1, \Sigma = [\sigma^2]$).

In the univariate case, to determine the probability that the value of a random variable will lie within a given interval, simply integrate the probability density function over that interval. Analogously in the multivariate case, given an interval for each of the variables (*i.e.* a hypervolume) perform a multiple integral, integrating each variable over its interval to determine the probability a random vector is within the hypervolume.

All this is preparation of the derivation of the linear classifier. Assume an example feature vector $\mathbf{x}$ to be classified is given. Let $C^{\hat{c}}$ denote the event that a random feature vector $\mathbf{X}$ is in class c, and $\mathbf{x}$, when used as an event, denote the event that the random feature vector $\mathbf{X}$ has value $\mathbf{x}$. We are interested in $P(C^{\hat{c}} \mid \mathbf{x})$, the probability that the particular feature vector $\mathbf{x}$ is in group $C^{\hat{c}}$. A reasonable classification rule is to assign $\mathbf{x}$ to the class i whose probability $P(C^{\hat{\imath}} \mid \mathbf{x})$ is greater than that of the other classes, *i.e.* $P(C^{\hat{\imath}} \mid \mathbf{x}) > P(C^{\hat{\jmath}} \mid \mathbf{x})$ for all $j \neq i$. This rule, which assigns the example to the class with the highest conditional probability, is known as *Bayes' rule*.

The problem is thus to determine $P(C^{\hat{c}} \mid \mathbf{x})$ for all classes c. Bayes' theorem tells us

$$P(C^{\hat{c}} \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid C^{\hat{c}})P(C^{\hat{c}})}{\sum_{\text{all } k} P(\mathbf{x} \mid C^{\hat{k}})P(C^{\hat{k}})} \tag{3.3}$$

Substituting, the assignment rule now becomes: assign $\mathbf{x}$ to class i if $P(\mathbf{x} \mid C^{\hat{\imath}})P(C^{\hat{\imath}}) > P(\mathbf{x} \mid C^{\hat{\jmath}})P(C^{\hat{\jmath}})$ for all $j \neq i$.

The terms of the form $P(C^{\hat{c}})$ are the *a priori* probabilities that a random example vector is in class c. In a gesture recognition system, these prior probabilities would depend on the frequency that each gesture command is likely to be used in an application. Lacking any better information, let us assume that all gestures are equally likely, resulting in the rule: assign $\mathbf{x}$ to class i if $P(\mathbf{x} \mid C^{\hat{\imath}}) > P(\mathbf{x} \mid C^{\hat{\jmath}})$ for all $j \neq i$.

A conditional probability of the form $P(\mathbf{x} \mid C^{\hat{c}})$ is known as the *likelihood* of $C^{\hat{c}}$ with respect to $\mathbf{x}$ [30]; assuming equal priors essentially replaces Bayes' rule with one that gives the maximum likelihood.

Assume now that each $C^{\hat{c}}$ is multivariate normal, with mean vector $\overline{\mu}^{\hat{c}}$, and covariance matrix $\Sigma_{\hat{c}}$. Substituting the multivariate normal density functions (equation 3.2) for the probabilities gives the assignment rule: assign $\mathbf{x}$ to class i if, for all $j \neq i$,

$$(2\pi)^{-F/2}|\Sigma_{\hat{\imath}}|^{-1/2}e^{-\frac{1}{2}(\mathbf{x}-\overline{\mu}^{\hat{\imath}})'\Sigma_{\hat{\imath}}^{-1}(\mathbf{x}-\overline{\mu}^{\hat{\imath}})} > (2\pi)^{-F/2}|\Sigma_{\hat{\jmath}}|^{-1/2}e^{-\frac{1}{2}(\mathbf{x}-\overline{\mu}^{\hat{\jmath}})'\Sigma_{\hat{\jmath}}^{-1}(\mathbf{x}-\overline{\mu}^{\hat{\jmath}})}$$

Taking the natural log of both sides, canceling, and multiplying through by -1 (thus reversing the inequality) gives the rule: assign $\mathbf{x}$ to class i if, for all $j \neq i$,

$$d^{\hat{\imath}}(\mathbf{x}) < d^{\hat{\jmath}}(\mathbf{x}), \text{ where } d^{\hat{c}}(\mathbf{x}) = \ln|\Sigma_{\hat{c}}| + (\mathbf{x} - \overline{\mu}^{\hat{c}})'\Sigma_{\hat{c}}^{-1}(\mathbf{x} - \overline{\mu}^{\hat{c}}) \tag{3.4}$$

$d^{\hat{c}}(\mathbf{x})$ is the discrimination function for class c applied to $\mathbf{x}$. This is *quadratic* discrimination, since $d^{\hat{c}}(\mathbf{x})$ is quadratic in elements of $\mathbf{x}$ (the features). The discriminant computation involves the weighted sum of the pairwise products of features, as well as terms linear in the features, and a constant term.

Making the further assumption that all the per-class covariances matrices are equal, *i.e.* $\Sigma_{\hat{\imath}} = \Sigma_{\hat{\jmath}} = \Sigma$, the assignment rule takes the form: assign $\mathbf{x}$ to class i if, for all $j \neq i$,

$$\ln|\Sigma| + (\mathbf{x} - \overline{\mu}^{\hat{\imath}})'\Sigma^{-1}(\mathbf{x} - \overline{\mu}^{\hat{\imath}}) < \ln|\Sigma| + (\mathbf{x} - \overline{\mu}^{\hat{\jmath}})'\Sigma^{-1}(\mathbf{x} - \overline{\mu}^{\hat{\jmath}}).$$

Distributing the subtractions and multiplying through by $-\frac{1}{2}$ gives the rule: assign $\mathbf{x}$ to class i if, for all $j \neq i$,

$$\nu^{\hat{\imath}}(\mathbf{x}) > \nu^{\hat{\jmath}}(\mathbf{x}), \text{ where } \nu^{\hat{c}}(\mathbf{x}) = (\overline{\mu}^{\hat{c}})'\Sigma^{-1}\mathbf{x} - \tfrac{1}{2}(\overline{\mu}^{\hat{c}})'\Sigma^{-1}\overline{\mu}^{\hat{c}} \tag{3.5}$$

Note that the discrimination functions $\nu^{\hat{c}}(\mathbf{x})$ are linear in the features (*i.e.* the elements of $\mathbf{x}$), the weights being $(\overline{\mu}^{\hat{c}})'\Sigma^{-1}$ and the constant term being $-\frac{1}{2}(\overline{\mu}^{\hat{\jmath}})'\Sigma^{-1}\overline{\mu}^{\hat{\jmath}}$.

Comparing equations 3.5 and 3.1 it is seen that to have the optimum classifier (given the assumptions) we take

$$w_j^{\hat{c}} = \sum_{i=1}^{F} \Sigma_{ij}^{-1} \overline{\mu}_i^{\hat{c}} \qquad 1 \leq j \leq F$$

and

$$w_0^{\hat{c}} = -\frac{1}{2} \sum_{i=1}^{F} w_i^{\hat{c}} \overline{\mu}_i^{\hat{c}}$$

for all classes c. It is not possible to know the $\overline{\mu}^{\hat{c}}$ and Σ; these must be estimated from the examples as described in the next section. The result will be that the $w_i^{\hat{c}}$ will be estimates of the optimal weights.

The possibility of a tie for the largest discriminant has thus far neglected. If $\nu^{\hat{\imath}}(\mathbf{x}) = \nu^{\hat{k}}(\mathbf{x}) > \nu^{\hat{\jmath}}(\mathbf{x})$ for all $j \neq i$ and $j \neq k$, it is clear that the classifier may arbitrarily choose i or k as the class of $\mathbf{x}$. However, this is a prime case for rejecting the gesture $\mathbf{x}$ altogether, since it is ambiguous. This kind of rejection is generalized in Section 3.6.

3.5.2 Estimating the parameters

The linear classifier just derived is optimal (given all the assumptions) in the sense that it maximizes the probability of correct classification. However, the parameters needed to operate the classifier, namely the per-class mean vectors $\overline{\mu}^{\hat{c}}$ and the common covariance matrix Σ, are not known *a priori*. They must be estimated from the training examples. The simplest approach is to use the plug-in estimates for these statistics. Since the equations that follow actually need to be programmed, the matrix notation is discarded in favor of writing the sums out explicitly in terms of the components.

Let $f_{ei}^{\hat{c}}$ 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 plug-in estimate of $\overline{\mu}^{\hat{c}}$, the mean feature vector per class, is denoted $\overline{f}_i^{\hat{c}}$. It is simply the average of the features in the class:

$$\overline{f}_i^{\hat{c}} = \frac{1}{E^{\hat{c}}} \sum_{e=0}^{E^{\hat{c}}-1} f_{ei}^{\hat{c}}$$

$s_{ij}^{\hat{c}}$ is the plug-in estimate of $\Sigma_{\hat{c}}$, the feature covariance matrix for class c:

$$s_{ij}^{\hat{c}} = \sum_{e=0}^{E^{\hat{c}}-1} (f_{ei}^{\hat{c}} - \overline{f}_i^{\hat{c}})(f_{ej}^{\hat{c}} - \overline{f}_j^{\hat{c}})$$

(For convenience in the next step, the usual $1/(E^{\hat{c}} - 1)$ factor has not been included in $s_{ij}^{\hat{c}}$.) The $s_{ij}^{\hat{c}}$ are averaged to give s_{ij}, an estimate of the common covariance matrix Σ.

$$s_{ij} = \frac{\sum_{c=0}^{C-1} s_{ij}^{\hat{c}}}{-C + \sum_{c=0}^{C-1} E^{\hat{c}}} \tag{3.6}$$

APLNDC00022025

The plug-in estimate of the common covariance matrix s_{ij} is then inverted, the result of which is denoted $(s^{-1})_{ij}$.

The $v^{\hat{c}}$ are estimates of the optimal evaluation functions $\nu^{\hat{c}}(\mathbf{x})$. The weights $w_j^{\hat{c}}$ are computed from the estimates as follows:

$$w_j^{\hat{c}} = \sum_{i=1}^{F} (s^{-1})_{ij} \overline{f}_i^{\hat{c}} \qquad 1 \le j \le F$$

and

$$w_0^{\hat{c}} = -\frac{1}{2} \sum_{i=1}^{F} w_i^{\hat{c}} \overline{f}_i^{\hat{c}}$$

As mentioned before, it is assumed that all gesture classes are equally likely to occur. The constant terms $w_0^{\hat{c}}$ may be adjusted if the *a priori* probabilities of each gesture class are known in advance, though the author has not found this to be necessary for good results. If the derivation of the classifier is carried out without assuming equal probabilities, the net result is, for each class, to add $\ln P(C^{\hat{c}})$ to $w_0^{\hat{c}}$. A similar correction may be made to the constant terms if differing per-class costs for misclassification must be taken into account [74].

Estimating the covariance matrix involves estimating its $F(F+1)/2$ elements. The matrix will be singular if, for example, less than approximately F examples are used in its computation. Or, a given feature may have zero variance in every class. In these cases, the classifier is underconstrained. Rather then give up (which seems an inappropriate response when underconstrained) an attempt is made to fix a singular covariance matrix. First, any zero diagonal element is replaced by a small positive number. If the matrix is still singular, then a search is made to eliminate unnecessary features.

The search starts with an empty set of features. At each iteration, a feature i is added to the set, and a covariance matrix based only on the features in the set is constructed (by taking the singular $F \times F$ covariance matrix and using only the rows and columns of those features in the set). If the constructed matrix is singular, feature i is removed from the set, otherwise i is kept. Each feature is tried in turn. The result is a covariance matrix (and its inverse) of dimensionality smaller than $F \times F$. The inverse covariance matrix is expanded to size $F \times F$ by adding rows and columns of zeros for each feature not used. The resulting matrix is used to compute the weights.

Appendix A shows C code for training classifiers and classifying feature vectors.

3.6 Rejection

Given an input gesture g, the classification algorithm calculates the evaluation $v^{\hat{c}}$, for each class c. The class k whose evaluation $v_{\hat{k}}$ is larger than all other $v^{\hat{c}}$ is presumed to be the class of g. However, there are two cases that might cause us to doubt the correctness of the classifier. The gesture g may be *ambiguous*, in that it is similar to the gestures of more than one class. Also, g may be an *outlier*, different from any of the expected gesture classes.

It would be desirable to get an estimate of how sure the classifier is that the input gesture is unambiguously in class i. Intuitively, one might expect that if some $v^{\hat{m}}$, $m \neq i$, is close to $v^{\hat{\imath}}$, then the classifier is unsure of its classification, since it almost picked m instead of i. This intuition is borne out in the expression for the probability that the feature vector $\mathbf{x}$ is in class $\hat{\imath}$. Again assuming normal features, equal covariances, and equal prior probabilities, substitute the multivariate normal density function (equation 3.2) into Bayes' Theorem (equation 3.3).

$$P(\hat{\imath} \mid \mathbf{x}) = \frac{e^{-\frac{1}{2}(\mathbf{x}-\overline{\mu}^{\hat{\imath}})'\Sigma^{-1}(\mathbf{x}-\overline{\mu}^{\hat{\imath}})}}{\sum_{j=0}^{C-1} e^{-\frac{1}{2}(x-\overline{\mu}^{\hat{\jmath}})'\Sigma^{-1}(x-\overline{\mu}^{\hat{\jmath}})}}$$

The common factor $(2\pi)^{-F/2}|\Sigma|^{-1/2}$ has been canceled from the numerator and denominator. We may further factor out and cancel $e^{-\frac{1}{2}\mathbf{x}'\Sigma^{-1}\mathbf{x}}$ and substitute equation 3.5, yielding

$$P(\hat{\imath} \mid \mathbf{x}) = \frac{e^{\nu^{\hat{\imath}}(\mathbf{x})}}{\sum_{j=0}^{C-1} e^{\nu^{\hat{\jmath}}(\mathbf{x})}}$$

Substituting the estimates $v^{\hat{c}}$ for the $\nu^{\hat{c}}(\mathbf{x})$ and incorporating the numerator into the denominator yields an estimate for the probability that i is the correct class for $\mathbf{x}$:

$$\tilde{P}(\hat{\imath} \mid \mathbf{x}) = \frac{1}{\sum_{j=0}^{C-1} e^{(v^{\hat{\jmath}}-v^{\hat{\imath}})}}$$

This value is computed after recognition and compared to a threshold T_P. If below the threshold, instead of accepting g as being in class i, g is rejected. The effect of varying T_P will be evaluated in Chapter 9. There is a tradeoff between wanting to reject as many ambiguous gestures as possible and not wanting to reject unambiguous gestures. Empirically, $T_P = 0.95$ has been found to be a reasonable value for a number of gesture sets (see Section 9.1.2).

The expression for $\tilde{P}(\hat{\imath} \mid \mathbf{x})$ bears out the intuition that if two or more classes evaluate to near the same result the gesture is ambiguous. In such cases the denominator will be significantly larger than unity. Note that the denominator is always at least unity due to the $j = i$ term in the sum. Also note that all the other terms the exponents $(v^{\hat{\jmath}} - v^{\hat{\imath}})$ for $j \neq i$ will always be negative, because $\mathbf{x}$ has been classified as class i by virtue of the fact that $v^{\hat{\imath}} > v^{\hat{\jmath}}$ for $j \neq i$.

$\tilde{P}(\hat{\imath} \mid \mathbf{x})$ may be computed efficiently by using table-lookup for the exponentiation. The table need not be very extensive, since any time $v^{\hat{\jmath}} - v^{\hat{\imath}}$ is sufficiently negative (less than -6, say) the term is negligible. In practice this will be the case for almost all j.

A linear classifier will give no indication if g is an outlier. Indeed, most outliers will be considered unambiguous by the above measure of $\tilde{P}$. To test if g is an outlier, a separate metric is needed to compare g to the typical gesture of class k. An approximation to the Mahalanobis distance [74] works well for this purpose.

Given a gesture with feature vector $\mathbf{x}$, the Mahalanobis distance between $\mathbf{x}$ and class i is defined as

$$\delta^2 = (\mathbf{x} - \overline{\mu}^{\hat{\imath}})'\Sigma^{-1}(\mathbf{x} - \overline{\mu}^{\hat{\imath}})$$

Note that δ^2 is used in the exponent of the multivariate normal probability density function (equation 3.2). It plays the role that $((x-\mu)/\sigma)^2$ plays in the univariate normal distribution: the Mahalanobis distance δ^2 essentially measures the (square of the) number of standard deviations that $\mathbf{x}$ is away from the mean $\overline{\mu}^{\hat{\imath}}$.

If Σ^{-1} happens to be the identity matrix, the Mahalanobis distance is equivalent to the Euclidean distance. In general, the Mahalanobis distance normalizes the effects of different scales for the different features, since these presumably show up as different magnitudes for the variances s_{ii}, the diagonal elements of the common covariance matrix. The Mahalanobis distance also normalizes away the effect of correlations between pairs of features, the off-diagonal elements of the covariance matrix.

As always, it is only possible to approximate the Mahalanobis distance between a feature vector $\mathbf{x}$ and a class i. Substituting the plug-in estimators for the population statistics and writing out the matrix multiplications explicitly gives

$$d^2 = \sum_{j=1}^{F}\sum_{k=1}^{F} s_{jk}^{-1}(x_j - \overline{f}_j^{\hat{\imath}})(x_k - \overline{f}_k^{\hat{\imath}}).$$

In order to reject outliers, compute d^2, an approximation of the Mahalanobis distance from the feature vector $\mathbf{x}$ to its computed class i. If the distance is greater than a certain threshold T_{d^2} the gesture is rejected. Section 9.1.2 evaluates various settings of T_{d^2}; here it is noted that setting $T_{d^2} = \frac{1}{2}F^2$ is a good compromise between accepting obvious outliers and rejecting reasonable gestures.

Now that the underlying mechanism of rejection has been explained, the question arises as to whether it is desirable to do rejections at all. The answer depends upon the application. In applications with easy to use undo and abort facilities, the reject option should probably be turned off completely. This is because in either failure mode (rejection or misclassification) the user will have to redo the gesture (probably about the same amount of work in both cases) and turning on rejection merely increases the number of gestures that will have to be redone.

In applications in which it is deemed desirable to do rejection, the question arises as to how the interface should behave when a gesture is rejected. The system may prompt the user with an error message, possibly listing the top possibilities for the class (judging from the discriminant functions) and asking the user to pick. Or, the system may choose to ignore the gesture and any subsequent input until the user indicates the end of the interaction. The proper response presumably depends on the application.

3.7 Discussion

One goal of the present research was to enable the implementor of a gesture-based system to produce gesture recognizers without the need to resort to hand-coding. The original plan was to try a number of pattern recognition techniques of increasing complexity until one powerful enough to recognize gestures was found. The author was pleasantly surprised when the first technique he tried, linear discrimination, produced accurate and efficient classifiers.

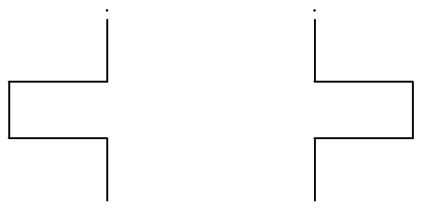

Figure 3.4: Two different gestures with identical feature vectors

The efficiency of linear recognition is a great asset: gestures are recognized virtually instantaneously, and the system scales well. The incremental feature calculation, with each new input point resulting in a bounded (and small) amount of computation, is also essential for efficiency, enabling the system to handle large gestures as efficiently as small ones.

3.7.1 The features

The particular feature set reported on here has worked fine discriminating between the gestures used in three sample applications: a simple drawing program, the uppercase letters in the alphabet, and a simple score editor. Tests using the gesture set of the score editor application are the most significant, since the recognizer was developed and tested on the other two. Chapter 9 studies the effect of training set size and number of classes on the performance of the recognizer. A classifier which recognizes thirty gestures classes had a recognition rate of 96.8% when trained with 100 examples per class, and a rate of 95.6% when trained with 10 examples per class. The misclassifications were largely beyond the control of the recognizer: there were problems using the mouse as a gesturing device and problems using a user process in a non-real-time system (UNIX) to collect the data.

It would be desirable to somehow show that the feature set was adequate for representing differences between all gestures likely to be encountered in practice. The measurements in Chapter 9 show good results on a number of different gestures sets, but are by no means a proof of the adequacy of the features. However, the mapping from gestures (sequences of points) to feature vectors is not one-to-one. In fact, it can easily be demonstrated that there are apparently different gestures that give rise to the same feature vector. Figure 3.4 shows one such pair of gestures. Since none of the features in the feature set depend on the order in which the angles in the gesture are encountered, and the two gestures are alike in every other respect, they have identical feature vectors. Obviously, any classifier based on the current feature set will find it impossible to distinguish between these gestures.

Of course, this particular deficiency of the feature set can be fixed by adding a feature that does depend on the order of the angles. Even then, it would be possible to generate two gestures which have the same angles in the same order, which differ, say, in the segment lengths between the angles, but nontheless give rise to the same feature vector. A new feature could then be added to handle this case, but it seems that there is still no way of being sure that there do not exist two different gestures giving rise to the same feature vector.

Nonetheless, adding features is a good way to deal with gesture sets containing ambiguous classes. Eventually, the number of features might grow to the point such that the recognizer performs inefficiently; if this happens, one of the algorithms that chooses a good subset of features could be applied [62, 103]. (Though not done in the present work, the contribution of individual features for a given classifier can be found using the statistical techniques of principle components analysis and analysis of variance [74].) However, given the good coverage that can be had with 13 features, 20 features would make it extremely unlikely that grossly different gestures with similar feature vectors would be encountered in practice. Since recognition time is proportional to the number of features, it is clear that a 20 feature recognizer does not entail a significant processing burden on modern hardware, even for large (40 class) gesture sets. There still may be good reason to employ fewer features when possible; for example, to reduce the number of training examples required.

The problem of detecting when a classifier has been trained on ambiguous classes is of great practical significance, since it determines if the classifier will perform poorly. One method is to run the training examples through the classifier, noting how many are classified incorrectly. Unfortunately, this may fail to find ambiguous classes since the classifier is naturally biased toward recognizing its training examples correctly. An alternative is to compute the pairwise Mahalanobis distance between the class means; potentially ambiguous classes will be near each other.

3.7.2 Training considerations

There is a potential problem in the training of classifiers, even when the intended classes are unambiguous. The problem arises when, within a class, the training examples do not have sufficient variability in the features that are irrelevant to the recognition of that class.

For example, consider distinguishing between two classes: (1) a rightward horizontal segment and (2) an upward vertical segment. Suppose all the training examples of the rightward segment class are short, and all those of the upward segment class are long. If the resulting classifier is asked to classify a long rightward segment, there is a significant probability of misclassification.

This is not surprising. Given the training examples, there was no way for classifier to know that being a rightward segment was the important feature of class (1), but that the length of the segment was irrelevant. The same training examples could just as well have been used to indicate that all elements of class (1) are short segments.

The problem is that, by not varying the length of the training examples, the trainer does not give the system significant information to produce the desired classifier. It is not clear what can be done about this problem, except perhaps to impress upon the people doing the training that they need to vary the irrelevant features of a class.

3.7.3 The covariance matrix

An important problem of linear recognition comes from the assumption that the covariance matrices for each class are identical. Consider a classifier meant to distinguish between three gestures classes named **C**, **U**, and **I** (figure 3.5). Examples of class **C** all look like the letter "C", and examples of class **U** all look like the letter "U." Assume that example **C** and **U** gestures are drawn similarly

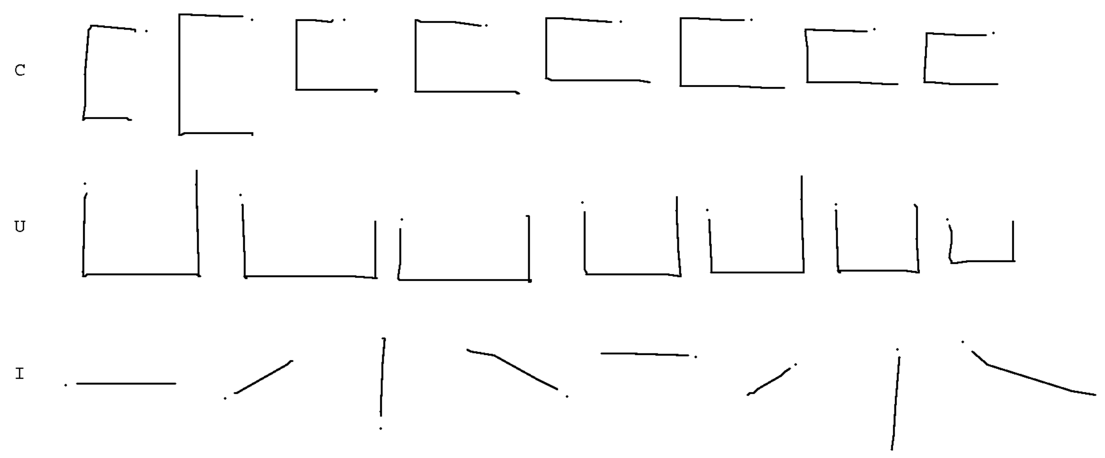

Figure 3.5: A potentially troublesome gesture set
This figure contains examples of three classes: **C**, **U**, *and* **I**. **I** *varies in orientation while* **C** *and* **U** *depend upon orientation to be distinguished. Theoretically, there should be a problem recognizing gestures in this set with the current algorithm, but in practice this has been shown not to be the case.*

except for the initial orientation. Examples of class **I**, however, are strokes which may occur in any initial orientation.

The point of this set of gesture classes is that initial orientation is essential for distinguishing between **C** and **U** gestures, but must be ignored in the case of **I** gestures. This information is contained in the per-class covariance matrices $s_{ij}^{\mathbf{C}}$, $s_{ij}^{\mathbf{U}}$, and $s_{ij}^{\mathbf{I}}$. In particular, consider the variance of the feature f_1, which, for each class c, is proportional to $s_{11}^{\hat{c}}$. Since the initial angle is almost the same for each example C gesture, $s_{11}^{\mathbf{C}}$ will be close to zero. Similarly, $s_{11}^{\mathbf{U}}$ will also be close to zero. However, since the examples of class **I** have different orientations, $s_{11}^{\mathbf{I}}$ will be significantly non-zero.

Unfortunately, the information on the variance of f_1 is lost when the per-class covariance matrix estimates $s_{ij}^{\hat{c}}$ are averaged to give an estimate of the common covariance matrix s_{ij} (equation 3.6). Initially, it was suspected this would cause a problem resulting in significantly lowered recognition rates, but in practice the effect has not been too noticeable. The classifier has no problem distinguishing between the above gestures correctly.

A more extensive test where some gestures vary in size and orientation while others depend on size and orientation to be recognized is presented in Section 9.1.4. The recognition rates achieved show the classifier has no special difficulty handling such gesture sets. Had there been a real problem, the plan was to experiment with improving the linear classifier, say by a few iterations of the perceptron training method [119]. Had this not worked, using a quadratic discriminator (equation 3.4) was another possible area of exploration.

3.8 Conclusion

This chapter discussed how linear statistical pattern recognition techniques can be successfully applied to the problem of classifying single-path gestures. By using these techniques, implementors of gesture-based systems no longer have to write application-specific gesture-recognition code. It is hoped that by making gesture recognizers easier to create and maintain, the promising field of gesture-based systems will be more widely explored in the future.

Chapter 4

Eager Recognition

4.1 Introduction

In Chapter 3, an algorithm for classifying single-path gestures was presented. The algorithm assumes that the entire input gesture is known, *i.e.* that the start and end of the gesture are clearly delineated. For some applications, this restriction is not a problem. For others, however, the need to indicate the end of the gesture makes the user interface more awkward than it need be.

Consider the use of mouse gestures in the GDP drawing editor (Section 1.1). To create a rectangle, the user presses a mouse button at one corner of the rectangle, enters the "L" gesture, stops (while still holding the button), waits for the rectangle to appear, and then positions the other corner. It would be much more natural if the user did not have to stop; *i.e.* if the system recognized the rectangle gesture *while the user was making it*, and then created the rectangle, allowing the user to drag the corner. What began as a gesture changes to a rubberbanding interaction with no explicit signal or timeout.

Another example, mentioned previously, is the manipulation of the image of a knob on the screen. Let us suppose that the knob responds to two gestures: it may be turned or it may be tapped. It would be awkward if the user, in order to turn the knob, needed to first begin to turn the knob (entering the turn gesture), then stop turning it (asking the system to recognize the turn gesture), and then continue turning the knob, now getting feedback from the system (the image of the knob now rotates). It would be better if the system, as soon as enough of the user's gesture has been seen so as to unambiguously indicate her intention of turning the knob, begins to turn the knob.

The author has coined the term *eager recognition* for the recognition of gestures as soon as they are unambiguous. Henry et. al. [52] mention that Artkit, a system similar to GRANDMA, can be used to build applications that perform eager recognition of mouse gestures. There is currently no information published as to how gesture recognition or eager recognition is implemented using Artkit. GloveTalk [34] does something similar in the recognition of DataGlove gestures. GloveTalk attempts to use the deceleration of the hand to indicate that the gesture in progress should be recognized. It utilizes four neural networks: the first recognizes the deceleration, the last three classify the gesture when indicated to do so by the first.

Eager recognition is the automatic recognition of the end of a gesture. For many applications, it

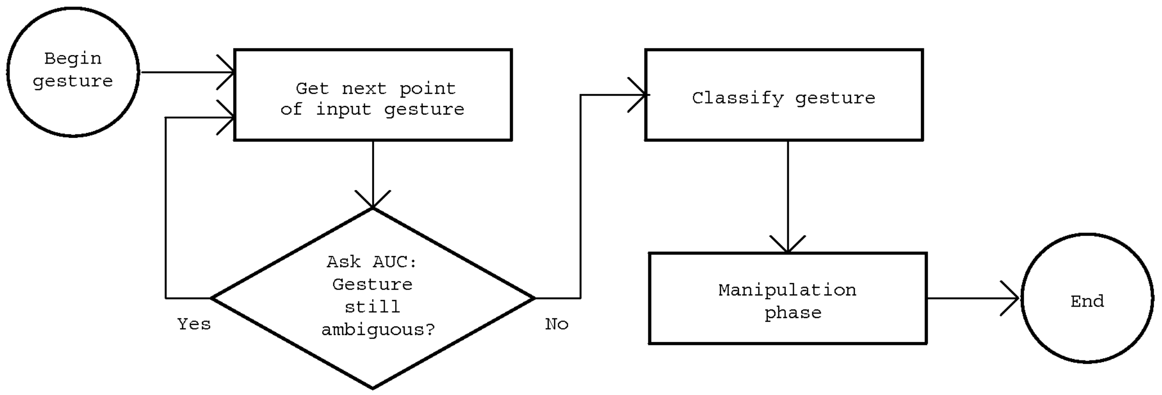
Begin gesture
Get next point of input gesture
Ask AUC: Gesture still ambiguous?
Yes
No
Classify gesture
Manipulation phase
End

Figure 4.1: Eager recognition overview
Eager recognition works by collecting points until the gesture is unambiguous, at which point the gesture is classified by the techniques of the previous chapter and the manipulation phase is entered. The determination as to whether the gesture seen so far is ambiguous is done by the AUC, i.e. *the* ambiguous/unambiguous *classifier.*

is not a problem to indicate the start of a gesture explicitly, by pressing a mouse button for example. In the present work, no attempt is made to solve the problem of determining the start of a gesture. Recognizing the start of a gesture automatically is especially important for gesture-based systems that use input devices without any explicit signaling capability (*e.g.* the Polhemus sensor or the DataGlove). For such a device, sudden changes in speed or direction might be used to indicate the start of a gesture. More complex techniques for determining the start of a gesture are outside the scope of this dissertation.

There has been some work on the automatic recognition of the start of gestures. Jackson and Roske-Hofstrand's system [61] recognizes the start of a circling gesture without an explicit indication. In GloveTalk, the user is always gesturing; thus the end of one gesture indicates the start of another. Also related is the automatic segmentation of characters in handwriting systems [125, 13], especially the online recognition of cursive writing [53].

4.2 An Overview of the Algorithm

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?" (figure 4.1). The insight here is to view this as a classification problem: classify a given gesture in progress (called a *subgesture* below) as an ambiguous or unambiguous gesture prefix. This is essentially the approach taken independently in GloveTalk. Here, the recognition techniques developed in the previous chapter are used to build the ambiguous/unambiguous classifier (AUC).

Two main problems need to be solved with this approach. First, training data is needed to train the AUC. Second, the AUC must be powerful enough to accurately discriminate between ambiguous and unambiguous subgestures.

In GloveTalk, the training data problem was solved by explicitly labeling snapshots of a gesture in progress. Each gesture was made up of an average of 47 snapshots (samples of the DataGlove and Polhemus sensors). For each of 638 gestures, the snapshot indicating the time at which the system should recognize the gesture had to be indicated. This is clearly a significant amount of work for the trainer of the system.

In order to avoid such tedious tasks, the present system constructs training examples for the AUC from the gestures used to train the main gesture recognizer. The system considers each subgesture of each example gesture, labels it either ambiguous or not, and uses the labeled subgestures as training data. It seems there is a chicken-and-egg problem here: in order to create the training data, the system needs to perform the very task for which it is trying to create a classifier. However, during the creation of the training data, the system has access to a crucial piece of information that makes the problem tractable: to determine if a given subgesture is ambiguous the system can examine the entire gesture from which the subgesture came.

Once the training data has been created, a classifier must be constructed. In GloveTalk this presented no particular difficulty, for two reasons. There, the classifier was trained to recognize decelerations that, as indicated by the sensor data, were similar between different gesture classes. Also, neural networks with hidden layers are better suited for recognizing classes with non-Gaussian distributions.

In the present system, the training data for the AUC consists of two sets: unambiguous subgestures and ambiguous subgestures. 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 GDP the unambiguous delete subgestures are very different from the unambiguous pack gestures, etc., so there will be a multimodal distribution of feature vectors in the unambiguous set. Similarly, the distribution of feature vectors in the ambiguous set will also likely be non-Gaussian. Thus, a linear discriminator of the form developed in the previous chapter will surely 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.

The details of the creation of the training data and the construction of the classifier are now presented. First a failed attempt at the algorithm is considered, during which the aforementioned problems were uncovered. Then a working version of the algorithm is presented.

4.3 Incomplete Subgestures

As in the last chapter, we are given a set of C gesture classes, and a number of examples of each class, $g_e^{\hat{c}}$, $0 \leq c < C$, $0 \leq e < E^{\hat{c}}$, where $E^{\hat{c}}$ is the number of examples of class c. The algorithm described in this chapter produces a function $\mathcal{D}$ which when given a subgesture returns a boolean indicating whether the subgesture is unambiguous with respect to the C gesture classes. When the function indicates that the subgesture is unambiguous, the recognition algorithm described in the previous chapter is used to classify the gesture.

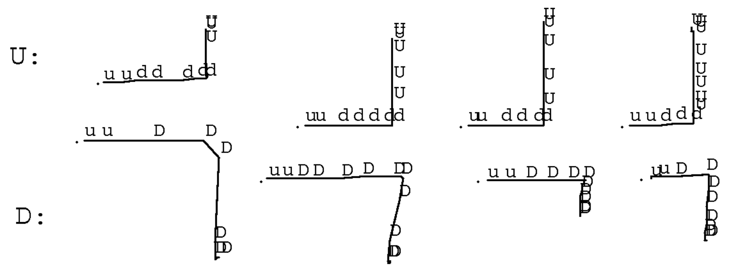

Figure 4.2: Incomplete and complete subgestures of U and D
The character indicates the classification (by the full classifier) of each subgesture. Uppercase characters indicate complete subgestures, meaning that the subgesture and all larger subgestures are correctly classified. Note that along the horizontal segment (where the subgestures are ambiguous) some subgestures are complete while others are not.

The classification algorithm of the previous chapter showed how, given a gesture g, to calculate a feature vector $\mathbf{x}$. A linear discriminator was then used to classify $\mathbf{x}$ as a class c. For much of this chapter, the classifier can be considered to be a function $\mathcal{C}$: $c = \mathcal{C}(g)$. In other words, $\mathcal{C}(g)$ is the class of g as computed by the classifier of Chapter 3.

The function $\mathcal{C}$ was produced from the statistics of the example gestures of each class c, $g_e^{\hat{c}}$. The algorithms described in this chapter work best if only the example gestures that are in fact classified correctly by the computed classifier are used. Thus, in this chapter it is assumed that $\mathcal{C}(g_e^{\hat{c}}) = c$ for all example gestures $g_e^{\hat{c}}$. In practice this is achieved by ignoring those very few training examples that are incorrectly classified by $\mathcal{C}$.

Denote the number of input 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 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" will be used when it is necessary to distinguish the full gesture g from its proper subgestures $g[i]$ for $i < |g|$. The term "full classifier" will be used to refer to $\mathcal{C}$, the classifier for full gestures.

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 $j, i \leq j < |g|, \mathcal{C}(g[j]) = \mathcal{C}(g)$. 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 4.2 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 of the subgesture which ends at the point. An upper case label indicates a complete subgesture, lower case 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

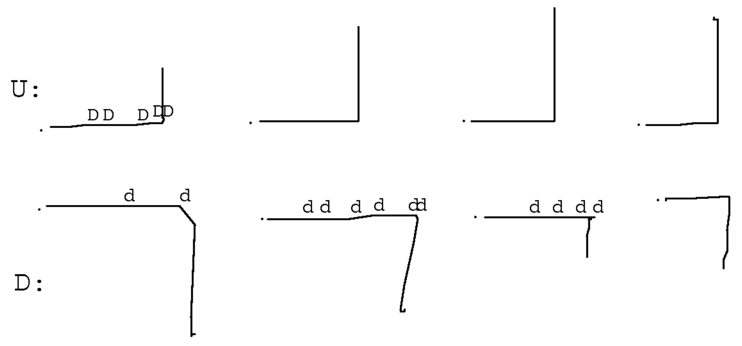

Figure 4.3: A first attempt at determining the ambiguity of subgestures
A two-class classifier was built to distinguish incomplete and complete subgestures, with the hope that those classified as complete are unambiguous and those classified as incomplete are ambiguous. The characters indicate where the resultant classifier differed from its training examples. The horizontal segment of the D *gestures were classified as incomplete (a fortuitous error), but the horizontal segment of the first* U *gesture was classified as complete. The latter is a grave mistake as the gestures are ambiguous along the horizontal segment and it would be premature for the full classifier to attempt to recognize the gesture at such points.*

the D examples).

4.4 A First Attempt

For eager recognition, subgestures that are unambiguous must be recognized as the gesture is being made. As stated above, the approach is to build an AUC, *i.e.* a classifier which distinguishes between ambiguous and unambiguous subgestures. Notice that the set of incomplete and complete subgestures approximate the set of ambiguous and unambiguous subgestures, respectively. The author's first, rather naive attempt at eager recognition was to partition the subgestures of all the example gestures into two classes, incomplete and complete. A linear classifier was then produced using the method described in Chapter 3. This classifier attempts to discriminate between complete and incomplete subgestures. The function $\mathcal{D}(g)$ then simply returns **false** whenever the above classifier reports that g is incomplete, and **true** whenever the classifier claims g is complete.

Figure 4.3 shows the output of the computed classifier for examples of U and D. Points corresponding to subgestures are labeled only when the classifier has made an error, in the sense that the classification does not agree with the training data (shown in figure 4.2). The worst possible error is for the classifier to indicate a complete gesture which happens to still be incomplete, which occurred along the right stroke of the first U gesture.

This approach to eager recognition was not very successful. That it is inadequate was indicated even more strongly by its numerous errors when tried on an example containing six gesture classes. It does however contain the germ of a good idea: that statistical classification may be used to determine if a gesture is ambiguous. A detailed examination of the problems of this attempt is instructive, and leads to a working eager recognition algorithm.

This first attempt at eager recognition has a number of problems:

- The distinction between incomplete and complete subgestures does not exactly correspond with the distinction between ambiguous and unambiguous subgestures. In the U and D example, subgestures consisting only of points along the right stroke are complete for gestures which eventually turn out to be D, and incomplete for gestures that turn out to be U. Yet, these subgestures have essentially identical features. Training a classifier on such conflicting data is bound to give poor results. In the example, as long as the right stroke is in progress the gesture is ambiguous. That it happens to be a complete D gesture is an artifact of the classifier $\mathcal{C}$ (it happens to choose D given only a right stroke).

- All the subgestures of examples were placed in one of only two categories: complete or incomplete. In the case of multiple gesture classes, within each of the two categories the subgestures are likely to form further clusters. For example, the complete U subgestures will cluster together, and be apart from the complete D subgestures. When more gesture classes are used, even more clustering will occur. Thus, the distribution of the complete subgestures is not likely to be normal. Furthermore, it is likely that incomplete subgestures will be more similar to complete gestures of the same class than to incomplete subgestures of other classes. (A similar remark holds for complete subgestures.) It is thus not likely that a linear discriminator will give good results separating complete and incomplete subgestures.

- The classifier, once computed, may make errors. The most severe error is reporting that a gesture is complete when it is in fact still ambiguous. The final classifier must be tuned to avoid such errors, even at the cost of making the recognition process less eager than it otherwise might be.

4.5 Constructing the Recognizer

Based on consideration of the above problems, a four step approach was adopted for the construction of classifiers able to distinguish unambiguous from ambiguous gestures.

Compute complete and incomplete sets.

Partition the example subgestures into $2C$ sets. These sets are named I-c and C-c for each gesture class c. A complete subgesture $g[i]$ is placed in the class C-c, where $c = \mathcal{C}(g[i]) = \mathcal{C}(g)$. An incomplete subgesture $g[i]$ is placed in the class I-c, where $c = \mathcal{C}(g[i])$ (and it is likely that $c \neq \mathcal{C}(g)$). 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.

Figure 4.4 shows pseudocode to perform this step. Figure 4.2, already seen, shows the result of this step, with the subgestures in class I-D labeled d, class I-U labeled u, class C-D labeled D, and class C-U labeled U. The practice of labeling incomplete subgestures with lowercase

```
for $c := 0$ to $C - 1$ { /* initialize the 2C sets */
          $incomplete_c := \emptyset$ /* This is the set I-c */
          $complete_c := \emptyset$ /* This is the set C-c */
}
for $c := 0$ to $C - 1$ { /* every class c */
          for $e := 0$ to $E^{\hat{c}} - 1$ { /* every training example in c */
                    $p := |g_e^{\hat{c}}|$ /* subgestures, largest to smallest */
                    while $p > 0 \wedge \mathcal{C}(g_e^{\hat{c}}[p]) = \mathcal{C}(g_e^{\hat{c}})$ {
                              $complete_{\mathcal{C}(g_e^{\hat{c}}[p])} := complete_{\mathcal{C}(g_e^{\hat{c}}[p])} \cup \{g_e^{\hat{c}}[p]\}$
                              $p := p - 1$
                    }
                    /* Once a subgesture is misrecognized by the full classifier, */
                    /* it and its subgestures are all incomplete. */
                    while $p > 0$ {
                              $incomplete_{\mathcal{C}(g_e^{\hat{c}}[p])} := incomplete_{\mathcal{C}(g_e^{\hat{c}}[p])} \cup \{g_e^{\hat{c}}[p]\}$
                              $p := p - 1$
                    }
          }
}
```

Figure 4.4: Step 1: Computing complete and incomplete sets

letters and complete subgestures with uppercase letters will be continued throughout the chapter.

Move accidentally complete elements.

Measure the distance of each subgesture $g[i]$ in each complete set to the mean of each incomplete set. If $g[i]$ is sufficiently close to one of the incomplete sets, it is removed from its complete set, and placed in the close 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 4.5 shows pseudocode to perform this operation.

Quantifying exactly what is meant by “sufficiently close” turned out to be rather difficult. Using the Mahalanobis distance as a metric turns out not to work well if applied naively. The problem is that it depends on the estimated average covariance matrix, which in turn depends upon the covariance matrix of the individual classes. However, some of the classes are malformed, which is why this step of moving accidentally complete elements is necessary in the first place. For example, the C-D class has accidentally complete subgestures in it, so its covariance matrix will indicate large standard deviations in a number of features (total angle, in this case). The effect of using the inverse of this covariance matrix to measure distance is that large differences between such features will map to small distances. Unfortunately, it is these very features that are needed to decide which subgestures are accidentally complete.

Alternatives exist. The average covariance matrix of the full gesture set (which does not include any subgestures) might be used. It would also be possible to use only the average covariance matrix of the incomplete classes. Or an attempt might be made to scale away the effect of different sized units of the features, and then apply a Euclidean metric. Or, the entire regrouping problem might be approached from a different direction, for example by applying a clustering algorithm to the training data [74]. The first alternative, using the average covariance matrix of the full gesture set (the same one used in the creation of the gesture classifier of Chapter 3) was chosen, since that matrix was easily available, and seems to work.

Once the metric has been chosen (Mahalanobis distance using the covariance matrix of the full gesture set), deciding when to move a subgesture from a complete class to an incomplete class is still difficult. The first method tried was to measure the distance of the subgesture to its current (complete) class, *i.e.* its distance from the mean of its class. The subgesture was moved to the closest incomplete class if that distance was less than the distance to its current class. This resulted in too few moves, as the mean of the complete class was biased since it was computed using some accidentally complete subgestures.

Instead, a threshold is computed, and if the distance of the complete subgesture to an incomplete class is below that threshold, the subgesture is moved. A fixed threshold does not work well, so the threshold 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, F^2, are not included in the minimum calculation to avoid trouble when an incomplete subgesture looked 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 90% of that minimum.

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

Figure 4.6 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.

Build the AUC, a classifier which attempts to discriminate between the partition sets.

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 algorithm in the previous chapter 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

```
**for** $c := 0$ **to** $C - 1$ {
    $\forall\, g \in complete_c$ /* each complete subgesture */{
        $m := 0$ /* m is the class of the incomplete set closest to g */
        **for** $i := 1$ **to** $C - 1$ {
            **if** distance($g$, $incomplete_i$) $<$ distance($g$, $incomplete_m$)
                $m := i$
        }
        **if** distance($g$, $incomplete_m$) $<$ $threshold$ {
            $complete_c := complete_c - \{g\}$
            $incomplete_c := incomplete_c \cup \{g\}$
        }
    }
}
```

Figure 4.5: Step 2: Moving accidentally complete subgestures
The distance function and threshold value are described in the text. Though not apparent from the above code, the distance function to an incomplete set does not change when elements are added to the set.

in one of the complete classes, in which case $\mathcal{D}(s)$ = **true**. Figure 4.7 shows pseudocode for building this classifier.

Evaluate and tweak the classifier.

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, w_{i0}, are incremented by a small amount, $\ln(M)$, where M is the relative cost of two kinds of misclassification. A reasonable value is $M = 5$, *i.e.* misclassifications as unambiguous are five times more costly than misclassifications as ambiguous. The effect is to bias the classifier so that it believes that ambiguous gestures are five times more likely than unambiguous gestures, so 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 4.9 shows the classification by the final classifier of the subgestures in the example. A larger example of eager recognizers is presented in section 9.2.

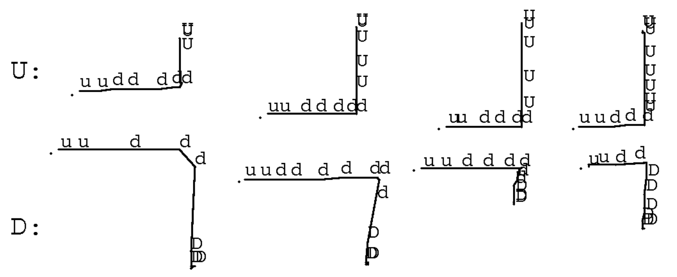

Figure 4.6: Accidentally complete subgestures have been moved

Comparing this to figure 4.2 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.*

```
$s$ := sNewClassifier()
for $c := 0$ to $C - 1$ {
        $\forall\, g \in complete_c$
                sAddExample($s, g$, "C − "$c$)
        $\forall\, g \in incomplete_c$
                sAddExample($s, g$, "I − "$c$)
}
sDoneAdding($s$)
```

Figure 4.7: Step 3: Building the AUC

The functions called to build a classifier are `sNewClassifier()`, *which returns a new classifier object,* `sAddExample`, *which adds an example of a class, and* `sDoneAdding`. *called to generate the per-class evaluation functions after all examples have been added. These functions are described in detail in appendix A. The notation* `"C-"`*c indicates the generation a class name by concatenating the string* `"C-"` *with the value of c.*

4.6 Discussion

The algorithm just described will determine whether a given subgesture is ambiguous with respect to a set of full gestures. Presumably, as soon as it is decided that the subgesture is unambiguous it will be passed to the full classifier, which will recognize it, and then up to the application level of the system, which will react accordingly.

How well this eager recognition works depends on a number of things, 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 no benefit trying to use eager recognition on Buxton's note gestures [21] (figure 2.4). 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. Designing a set of gestures for a given application that is both intuitive and amenable to eager recognition is in general a hard problem.

```
/* Add a small constant to the constant term of the evaluation function for */
/* each incomplete class in order to bias the classifier toward erring conservatively. */
for $i := 0$ to $C - 1$
    sIncrementConstantTerm($s$, "I-"$i$, $\ln(M)$)
/* Make sure that no ambiguous training example is ever classified as complete. */
for $i := 0$ to $C - 1$
    $\forall\, g \in incomplete_i$
        while $\exists\, c \mid$ sClassify($s, g$) = "C − "$c$
            sIncrementConstantTerm($s$, "C − "$c$, $-\epsilon$)
```

Figure 4.8: Step 4: Tweaking the classifier

First, a small constant is added to the constant term of every incomplete class (the ambiguous subgestures), to bias the classifier toward being conservative, rather than eager. Then every ambiguous subgestures is classified, and if any is accidentally classified as complete, the constant term of the evaluation function for that complete class is adjusted to avoid this.

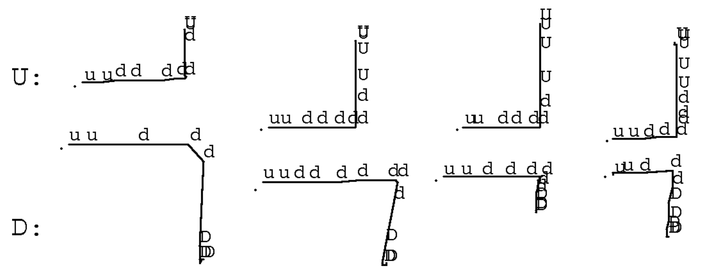

Figure 4.9: Classification of subgestures of U and D

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

The training of the eager recognizer is between one and two orders of magnitude more costly than the training of the corresponding classifier for full gestures. This is largely due to the number of training examples: each full gesture example typically gives rise to ten or twenty subgestures. The amount of processing per training example is also large. In addition to computing the feature vector of each training example, a number of passes must be made over the training data: first to classify the subgestures as incomplete or complete, then to move the accidentally complete subgestures, again to build the AUC, and again to ensure the AUC is not over-eager. While a full classifier takes less than a second to train, the eager recognizer might take a substantial portion of a minute, making it less satisfying to experiment with interactively. As will be seen (Chapter 7), a full classifier may be trained the first time a user gestures at a display object. One possibility would be to use the full classifier (with no eagerness) while training the AUC in the background, activating eager recognition when ready.

The running time for the eager recognizer is also more costly than the full classifier, though not prohibitively so. A feature vector needs to be calculated for every input point; this eliminates any benefit that using auxiliary features (Section 3.3) might have bought. Of course, the AUC needs to be run at every data point; this takes about $2CF$ multiply-adds (since the AUC has $2C$ classes). Since input points do not usually come faster than one every 30 milliseconds, and $2CF$ is typically at most 1000, this computational load is not usually a problem for today's typical workstation class machine. In the current system, the multiply-adds are done in floating point, though this is probably not necessary for the recognition to work well.

One slight defect of the algorithm used to construct the AUC is that it relies totally upon the full classifier. In particular, a subgesture will never be considered unambiguous unless it is classified correctly by the full classifier. To see where this might be suboptimal, consider a full classifier that recognizes two classes, GDP's single segment line gesture and three segment delete gesture. The full classifier would likely classify any subgesture that is the initial segment of a delete as a line. It *may* also classify some two segment subgestures of delete as line gestures, even though the presence of two segments implies the gesture is unambiguously delete. The resulting eager recognizer will then not be as eager as possible, in that it will not classify the input gesture as unambiguously delete immediately after the second segment of the gesture is begun.

Two classifiers are used for eager recognition: the AUC, which decides when a subgesture is unambiguous, and the full classifier, which classifies the unambiguous subgesture. It may seem odd to use two classifiers given the implementation of the AUC, in which a subgesture is not only classified as unambiguous, but unambiguously in a given class (*i.e.* classified as C-c for some c). Why not just return a classification of c without bothering to query the full classifier? There are two main reasons. First, the full classifier, having only C classes to discriminate between, will perform better than the AUC and its $2C$ classes. Second, the final tweaking step of the AUC adjusts constant terms to assure that ambiguous gestures are never classified as unambiguous, but makes no attempt to assure that when classified as unambiguously c, c is the correct class. The adjustment of the constant terms typically degrades the AUC in the sense that it makes it more likely that c will be incorrect.

It is likely that within a decade it will be practical for neural networks to be used for gesture recognition. When this occurs, the part of this chapter concerned with building a $2C$ class linear classifier will be obsolete, since a two-class neural network could presumably do the same job. However, the part of the chapter which shows how to construct training examples for the classifier from the full gestures will still be useful, since it eliminates the hand labeling that otherwise might be necessary.

4.7 Conclusion

An eager recognizer is able to classify a gesture as soon as enough of the gesture has been seen to conclude that the gesture is unambiguous. This chapter presents an algorithm for the automatic construction of eager recognizers for single-path gestures from examples of the full gestures. It is hoped that such an algorithm will make gesture-based systems more natural to use.

Chapter 5

Multi-Path Gesture Recognition

Chapters 3 and 4 discussed the recognition of single-path gestures such as those made with a mouse or stylus. This chapter addresses the problem of recognizing multi-path gestures, *e.g.* those made using an input device, such as the DataGlove, capable of tracking the paths of multiple fingertips. It is assumed that the start and end of the multi-path gesture are known. Eager recognition of multi-path gestures has been left for future work.

The particular input device used to test the ideas in this chapter is the Sensor Frame. The Sensor Frame, as discussed in Section 2.1, is a frame which is mounted on a CRT display. The particular Sensor Frame used was mounted on the display of a Silicon Graphics IRIS Personal Workstation. The Frame detects the XY positions of up to three fingertips in a plane approximately one half inch in front of the display.

By defining the problem as "multiple-path gesture recognition", it is quite natural to attempt to apply algorithms for single-path gesture recognition (*e.g.* those developed in Chapter 3). Indeed, the recognition algorithm described in this chapter combines information culled from a number of single-path classifiers, and a "global feature" classifier in order to classify a multiple-path gesture. Before the particular algorithm is discussed, the issue of mapping the raw data returned from the particular input sensors into a form suitable for processing by the recognition algorithm must be addressed. For the Sensor Frame, this processing consisted of two stages, path tracking and path sorting.

5.1 Path Tracking

The Sensor Frame, as it currently operates, delivers the X and Y coordinates of all fingers in its plane of view each time it is polled, at a maximum rate of 30 snapshots per second. No other information is supplied; in particular the correspondence between fingers in the current and the previous snapshots is not communicated. For example, when the previous snapshot indicated one finger and the current snapshot two, it is left to the host program to determine which of the two fingers (if any) is the same finger as the previously seen one, and which has just entered the field of view. Similarly, if both the previous and current snapshots indicate two fingers, the host program must determine which finger in the current snapshot is the same as the first finger in the previous snapshot, and so on. This

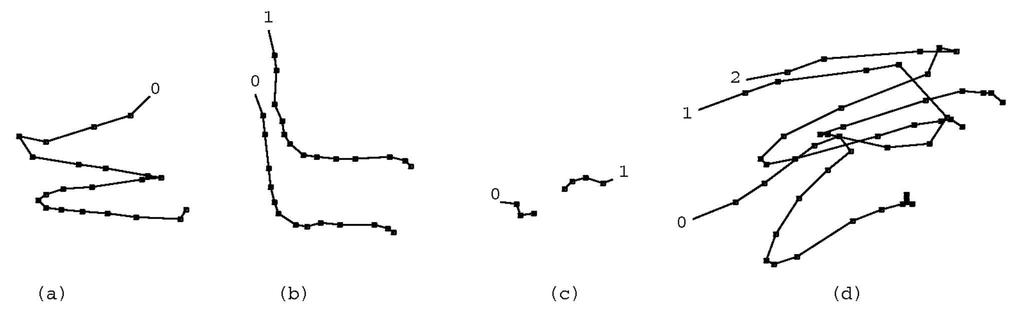

Figure 5.1: Some multi-path gestures

Shown are some MDP gestures made with a Sensor Frame. The start of each path is labeled with a path index, indicating the path's position in a canonical ordering. Gesture (a) is MDP's edit *gesture, an "E" made with a single finger. Gesture (b), parallel "L"s, is two finger* parallelogram *gesture, (c) is MDP's two finger* pinch *gesture (used for moving objects), and (d) is MDP's three finger* undo *gesture, three parallel "Z"s. The finger motions were smooth, and some noise due to the Sensor Frame's position detection can be seen in the examples.*

problem is known as *path tracking*, since it groups the raw input data into a number of paths which exist over time, each path having a definite beginning and end.

The path tracking algorithm used is quite straightforward. When a snapshot is first read, a triangular distance matrix, containing the Euclidean distance squared between each finger in the current snapshot and each in the previous, is computed. Then, for each possible mapping between current and previous fingers, an error metric, consisting of the sum of the squared distances between corresponding fingers, is calculated. The mapping with the smallest error metric is then chosen.

For efficiency, for each possible number of fingers in the previous snapshot and the current snapshot, a list of all the possible mappings are precomputed. Since the Sensor Frame detects from zero to three fingers, only 16 lists are needed. When the symmetry between the previous and current snapshots is considered, only eight lists are needed.

The low level tracking software labels each finger position with a *path identifier*. When there are no fingers in the Sensor Frame's field of view, the `next_path_identifier` variable is set to zero. A finger in the current snapshot which was not in the previous snapshot (as indicated by the chosen mapping) has its path identifier set to the value of `next_path_identifier` which is then incremented. It is thus possible for a single finger to generate multiple paths, since it will be assigned a new path identifier each time it leaves and reenters the field of view of the Sensor Frame, and those identifiers will increase as long as another finger remains in the field of view of the Frame.

The simple tracking algorithm described here was found to work very well. The anticipated problem of mistracking when finger paths crossed did not arrive very often in practice. (This was partly because all gestures were made with the fingers of a single hand, making it awkward for finger paths to cross.) Enhancements, such as using the velocity and the acceleration of each finger in the previous snapshot to predict where it is expected in the current snapshot, were not needed. Examples of the tracking algorithm in operation are shown in figure 5.1. In the figure, the start of each path is labeled with its path index (as defined in the following section), and the points in the path are connected by line segments. Figure 5.1d shows an uncommon case where the path tracking algorithm failed, causing paths 1 and 2 to be switched.

5.2 Path Sorting

The multi-path recognition algorithm, to be described below, works by classifying the first path in the gesture, then the second, and so on, then combining the results to classify the entire gesture. It would be possible to use a single classifier to classify all the paths; this option is discussed in Section 5.7. However, since classifiers tend to work better with fewer classes, it makes sense to create multiple classifiers, one for the first path of the gesture, one for the second, and so on. This however raises the question of which path in the gesture is the first path, which is the second, etc. This is the *path sorting* problem, and the result of this sorting assigns a number to each path called its *path index*.

The most important feature of a path sorting technique is consistency. Between similar multi-path gestures, it is essential that corresponding paths have the same index. Note that the path identifiers, discussed in the previous section, are not adequate for this purpose, since they are assigned in the order that the paths first appear. Consider, for example, a "pinching" gesture, in which the thumb and forefinger of the right hand are held apart horizontally and then brought together, the thumb moving right while the forefinger moves left. Using the Sensor Frame, the thumb path might be assigned path identifier zero in one pinching gesture, since it entered the view plane of the Frame first, but assigned path identifier one in another pinching gesture since in this case it entered the view plane a fraction of a second after the forefinger. In order for multi-path gesture recognition using of multiple classifiers to give good results, it is necessary that the all thumb motions be sent to the same classifier for training and recognition, thus using path identifiers as path indices would not give good results.

For multi-path input devices which are actually attached to the hand or body, such as the DataGlove, there is no problem determining which path corresponds to which finger. Thus, it would be possible to build one classifier for thumb paths, another for forefinger paths, etc. The characteristics of the device are such that the question of path sorting does not arise.

However, the Sensor Frame (and multifinger tablets) cannot tell which of the fingers is the thumb, which is the forefinger, and so on. Thus there is no *a priori* solution to the path sorting. The solution adopted here was to impose an ordering relation between paths. The consistency property is required of this ordering relation: the ordering of corresponding paths in similar gestures must be the same.

The primary ordering criterion used was the path starting time. However, to avoid the aforementioned timing problem, two paths which start within 200 milliseconds are considered simultaneous, and the secondary ordering criteria is used. A path which starts more than 200 msec before another path will be considered "less than" the other path, and show up before the other path in the sorting.

The secondary ordering criterion is the initial x coordinate. There is a window of 150 Sensor Frame length units (about one inch) within which two paths will be considered to start at the same x coordinate, causing the tertiary ordering criterion to be applied. Outside this window, the path with

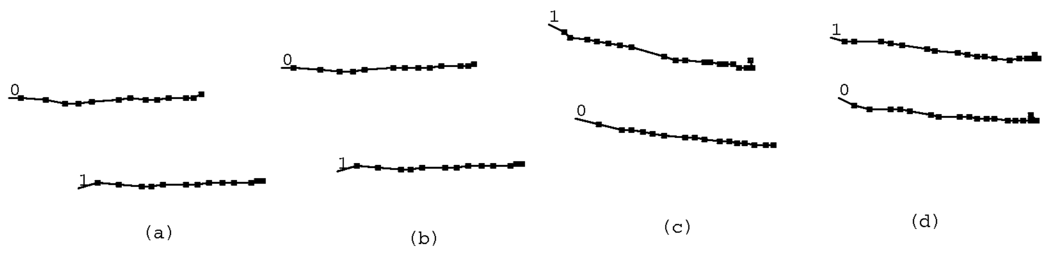

Figure 5.2: Inconsistencies in path sorting
The intention of the path sorting is that corresponding paths in two similar gestures should have the same path index. Here are four similar gestures for which this does not hold: between (b) and (c) the path sorting has changed.

the smaller initial x coordinate will appear before the other path in the sorting (assuming apparent simultaneity).

The tertiary ordering criterion is the initial y coordinate. Again, a window of 150 Sensor Frame length units is applied. Outside this window, the path whose y coordinate is less will appear earlier in the path ordering. Finally, if both the initial x and y coordinate differ by less than 150 units, the coordinate whose difference is the largest is used for ordering, and the path whose coordinate is smaller appears earlier in the path ordering.

Figure 5.2 shows the sorting for some multi-path gestures by labeling the start of each path with its index. Note that the consistency criteria is not maintained between panels (b) and (c), since the "corresponding" paths in the two gestures have different indices. The order of the paths in (b) was determined by the secondary ordering criterion (since the paths began almost simultaneously), while the ordering in (c) was determined by the tertiary ordering criterion (since the paths began simultaneously and had close x coordinates). Generally, *any* set of ordering rules which depend solely on the initial point of each path can be made to generate inconsistent sortings.

In practice, the possibility of inconsistencies has not been much of a problem. The ordering rules are set up so as to be stable for near-vertical and near horizontal finger configurations; they become unstable when the angle between (the initial points of) two fingers causes the 150 unit threshold to be crossed.[1] Knowing this makes it easy to design gesture sets with consistent path orderings. A more robust solution might be to compute a path ordering relation based on the actual gestures used to train the system.

As stated above, some multiple finger sensing devices, such as the DataGlove, do not require any path sorting. To use the DataGlove as input to the multi-path gesture recognizer described below, one approach that could be taken is to compute the paths (in three-space over time) of each fingertip, using the measured angles of the various hand joints. This will result in five sorted paths (one for each finger) which would be suitable as input into the multi-path recognition algorithm. (Of course, the lack of explicit signaling in the DataGlove still leaves the problem of determining the start and

[1]In retrospect, the 150 unit windows make the sorting more complicated then it need be. Using the coordinate whose difference is the largest (for simultaneous paths) makes the algorithm more predictable: it will become inconsistent when the initial points of two paths form an angle close to $-45°$ from the horizontal.

end of the gesture.)

5.3 Multi-path Recognition

Like the single path recognizers described in Chapter 3, the multi-path recognizer is trained by specifying a number of examples for each gesture class. The recognizer consists of a number of single-path classifiers, and a global feature classifier. These classifiers all use the statistical classification algorithm developed in Chapter 3. The differences are mainly in the sets of features used, as described in Section 5.5.

Each single-path classifier discriminates between gestures of a particular sorting index. Thus, there is a classifier for the first path of a gesture, another for the second path, and another for the third path. (The current implementation ignores all paths beyond the third, although it takes the actual number of paths into account.) When a multi-path gesture is presented to the system for classification, the paths are sorted (as described above) and the first path is classified using the first path classifier, and so on, resulting in a sequence of single-path classes.

The sequence of path classes which results is then submitted to a *decision tree*. The root node of the tree has slots pointing to subnodes for each possible class returned by the first path classifier. The subnode corresponding to the class of the first path is chosen. This node has slots pointing to subnodes for each possible class returned by the second path classifier. Some of these slots may be null, indicating that there is no expected gesture whose first and second path classes are the ones computed. In this case the gesture is rejected. Otherwise, the subnode corresponding to the class of the second path is chosen. The process is repeated for the third path class, if any.

Once the entire sequence of path classes is considered there are three possibilities. If the sequence was unexpected, the multi-path gesture is rejected since no node corresponding to this sequence exists in the decision tree. If the node does exist, the multi-path classification may be unambiguous, meaning only one multi-class gesture corresponds to this particular sequence of single-path classes. Or, there may be a number of multi-path gestures which correspond to this sequence of path classes. In this case, a global feature vector (one which encompasses information about all paths) is computed, and then classified by the global feature classifier. This class is used to choose a further subnode in the decision tree, which will result in the multi-path gesture either being classified individually or rejected. The intent is that, if needed, the global feature class is essentially appended to a sequence of path classes; some care is thus necessary to insure that the global feature classes are not confused with path classes.

Figure 5.3 shows an example of the use of a decision tree to classify multi-path gestures. The multi-path classifier recognizes four classes. Each class is composed of two paths. There are only two possible classes for the first path (path 0), since classes P, Q, and S all have similar first paths. Similarly, Q and S have similar second paths, so there are only three distinct possibilities for path 1. Since Q and S have identical path components, the global classifier is used to discriminate between these two, adding another level in the decision tree. The classification of the example input is indicated by dotted lines.

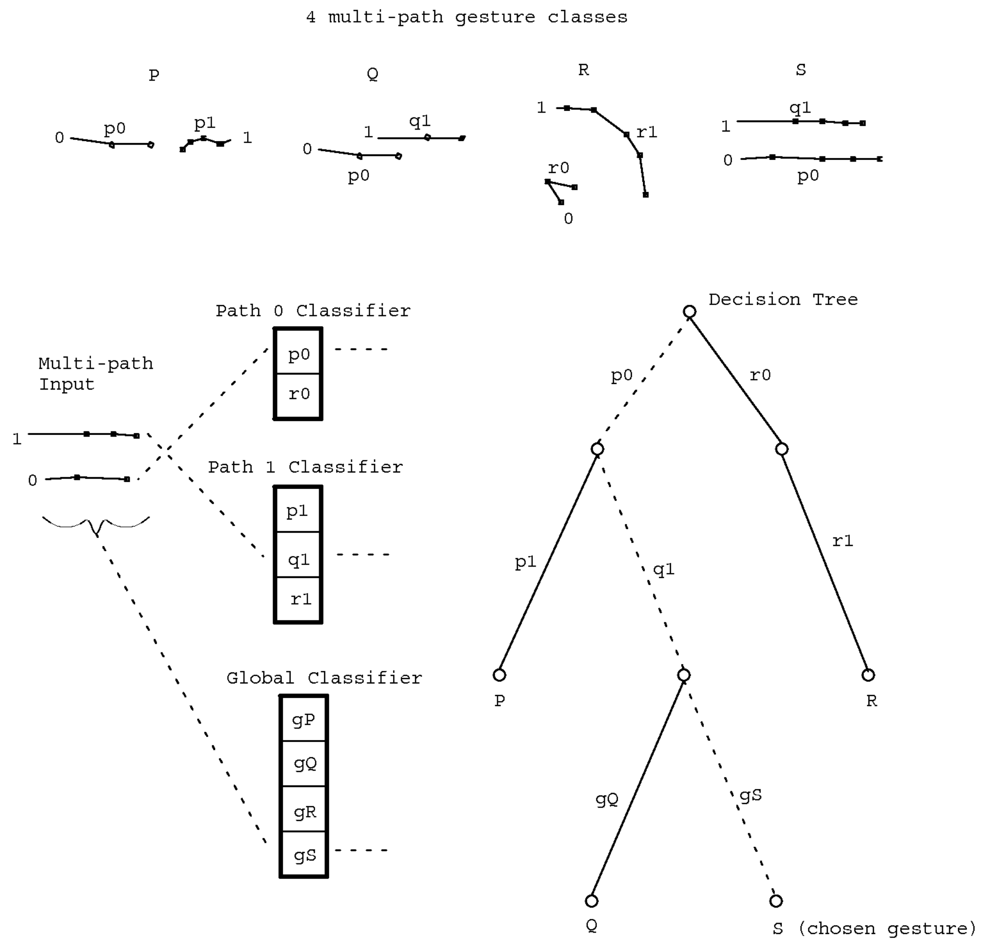

Figure 5.3: Classifying multi-path gestures

At the top are examples of four two-path gestures expected by this classifier, and at the left a two-path gesture to be classified. Path 0 of this gesture is classified (by the path 0 classifier) as path p0*, and path 1 as* q1*. These path classifications are used to traverse the decision tree, as shown by the dotted lines. The tree node reached is ambiguous (having children* Q *and* S*) so global features are used to resolve the discrepancy, and the gesture is recognized as class* S*.*

5.4 Training a Multi-path Classifier

The training algorithm for a multi-path classifier uses examples of each multi-path gesture class (typically ten to twenty examples of each class) to create a classifier. The creation of a multi-path classifier consists of the creation of a global classifier, a number of path classifiers, and a decision tree.

5.4.1 Creating the statistical classifiers

The path classifiers and the global classifiers are created using the statistical algorithm described in Chapter 3. The paths of each example are sorted, the paths for a given sorting index in each class forming a class used to train the path classifier for that index.

For example, consider training a multi-path classifier to discriminate between two multi-path gesture classes, A and B, each consisting of two paths. Gesture class A consists of two path classes, A_1 and A_2, the subscript indicating the sorting indices of the paths. Similarly, class B consists of path classes B_1 and B_2. The first path in all the A examples form the class A_1, and so on. The examples are used to train path classifier 1 to discriminate between A_1 and B_1, and path classifier 2 to discriminate between A_2 and B_2. The global features of A and B are used to create the global classifier, nominally able to discriminate between two classes of global features, A_G and B_G.

Within a given sorting index, it is quite possible and legitimate for paths from different gesture classes to be indistinguishable. For example, path classes A_1 and B_1 may both be straight right strokes. (Presumably A and B are distinguishable by their second paths or global features.) In this case it is likely that examples of class A_1 will be misclassified as B_1 or vice versa. It is desirable to remove these ambiguities from the path classifier by combining all classes which could be mistaken for each other into a single class.

A number of approaches could be taken for detecting and removing ambiguities from a statistical classifier. One possible approach would be to compute the Mahalanobis distance between each pair of classes, merging those below a given threshold. Another approach involves applying a clustering algorithm [74] to all the examples, merging those classes whose members are just as likely to cluster with examples from other classes as their own. A third approach is to actually evaluate the actual performance of a classifier which attempts to distinguish between possibly ambiguous classes; the misclassifications of the classifier then indicate which classes are to be merged. The latter approach was the one pursued here.

A naive approach for evaluating the performance of a classifier would be to construct the classifier using a set of examples, and then testing the performance of the classifier on those very same examples. This approach obviously underestimates the ambiguities of the classes since the classifier will be biased toward correctly classifying its training examples [62]. Instead, a classifier is constructed using only a small number of the examples (typically five per class) and then uses the remaining examples to evaluate the constructed classifiers. Misclassifications of the examples then indicate classes which are ambiguous and should be merged. In practice, thresholds must be established so that a single or very small percentage of misclassifications does not cause a merger.

Mathematically, combining classes is a simple operation. The mean vector of the combined class is computed as the average of the mean vectors of the component classes, each weighted by the relative number of examples in the class. A similar operation computes a composite average covariance matrix from the covariance matrices of the classes being combined.

The above algorithm, which removes ambiguities by combining classes, is applied to each path classifier as well as the global classifier. It remains now only to construct the decision tree for the multi-path classifier.

5.4.2 Creating the decision tree

A decision tree node has two fields: `mclass`, a pointer to a multi-path gesture class, and `next`, a pointer to an array of pointers to its subnodes. To construct the decision tree, a root node is allocated. Then, during the *class phase*, each multi-path gesture class is considered in turn. For each, a sequence of path classes (in sort index order), with its global feature class appended, is constructed. Nodes are created in the decision tree in such a way that by following the sequence a leaf node whose `mclass` value is the current multi-path gesture class is reached. This creates a decision tree which will correctly classify all multi-class gesture whose component paths and global features are correctly classified.

Next, during the *example phase*, each example gesture is considered in turn. The paths are sorted and classified, as are the global features. A sequence is constructed and the class of the gesture is added to the decision tree at the location corresponding to this sequence as before. Normally, the paths and global features of the gesture will have been classified correctly, so there would already be a node in the tree corresponding to this sequence. However, if one of the paths or the global feature vector of the gesture was classified incorrectly, a new node may be created in the decision tree, and thus the same classification mistake in the future will still result in a correct classification for the gesture.

When attempting to add a class using a sequence whose components are misclassifications, it is possible that the decision tree node reached already has a non-null `mclass` field referring to a different multi-path gesture class than the one whose example is currently being considered. This is a *conflict* and is resolved by ignoring the current example (though a warning message is printed). Ignoring all but the first instance of a sequence insures that the sequences generated during the class phase will take precedence over those generated during the example phase. Of course, a conflict occurring during the class phase indicates a serious problem, namely a pair of gesture classes between which the multi-path classifier is unable to discriminate.

During decision tree construction, nodes that have only one global feature class entry with a subnode have their `mclass` value set to the same gesture class as the `mclass` value of that subnode. In other words, sequences that can be classified without referring to their global feature class are marked as such. This avoids the extra work (and potential for error) of global feature classification.

5.5 Path Features and Global Features

The classification of the individual paths and of the global features of a multi-path gesture are central to the multi-path gesture recognition algorithm discussed thus far. This section describes the particular feature vectors used in more detail.

The classification algorithm used to classify paths and global features is the statistical algorithm discussed in Chapter 3, thus the criteria for feature selection discussed in section 3.3 must be addressed. In particular, only features with Gaussian-like distributions that can be calculated incrementally are considered.

The path features include all the features mentioned in Chapter 3. One additional feature was added: the starting time of the path relative to the starting time of the gesture. Thus, for example, a gesture consisting of two fingers, one above the other, which enter the field of view of the Sensor Frame simultaneously and move right in parallel can be distinguished from a gesture in which a single finger enters the field first, and while it is moving right a second finger is brought into the viewfield and moves right. In particular, the classifier (for the second sorting index) would be able to discriminate between a path which begins at the start of the gesture and one which begins later. The path start time is also used for path sorting, as described in section 5.2.

The main purpose of the global feature vector is to discriminate between multi-path gesture classes whose corresponding individual component paths are indistinguishable. For example, two gestures both consisting of two fingers moving right, one having the fingers oriented vertically, the other horizontally. Or, one having the fingers about one half inch apart, the other two inches apart.

The global features are the duration of the entire gesture, the length of the bounding box diagonal, the bounding box diagonal angle (always between 0 and $\pi/2$ so there are no wrap-around problems), the length, sine and cosine between the first point of the first path and the first point of the last path (referring to the path sorting order), and the length, sine, and cosine between the first point of the first path and the last point of the last path.

Another multi-path gesture attribute, which may be considered a global feature, is the actual number of paths in the gesture. The number of paths was not included in the above list, since it is not included in the vector input to the statistical classifier. Instead, it is required that all the gestures of a given class have the same number of paths. The number of paths must match exactly for a gesture to be classified as a given class. This restriction has an additional advantage, in that knowing exactly the number of paths simplifies specifying the semantics of the gesture (see Section 8.3.2).

The global features, crude as they might appear, in most cases enable effective discrimination between gesture classes which cannot be classified solely on the basis of their constituent paths.

5.6 A Further Improvement

As mentioned, the multi-path classifier has a path classifier for each sorting index. The path classifier for the first path needs to distinguish between all the gestures consisting only of a single path, as well as the first path in those gestures having two or more paths. Similarly, the second path classifier must discriminate not only between the second path of the two-path gestures, but also the second path of the three path gestures, and so on. This places an unnecessary burden on the path classifiers. Since gesture classes with different numbers of paths will never be confused, there is no need to have a path classifier able to discriminate between their constituent paths. This observation leads to a further improvement in the multi-path recognizer.

The improvement is instead of having a single multi-path recognizer for discriminating between multi-path gestures with differing numbers of paths, to have one multi-path gesture recognizer, as described above, for each possible number of paths. There is a multi-path recognizer for gestures consisting of only one path, another for two-path gestures, and so on, up until the maximum number of paths expected. Each path classifier now deals only with those paths with a given sorting index from those gestures with a given number of paths. The result is that many of the path classifiers have fewer paths to deal with, and improve their recognition ability accordingly.

Of course, for input devices in which the number of paths is fixed, such as the DataGlove, this improvement does not apply.

5.7 An Alternate Approach: Path Clustering

The multi-path gesture recognition implementation for the Sensor Frame relies heavily on path sorting. Path sorting is used to decide which paths are submitted to which classifiers, as well as in the global feature calculation. Errors in the path sorting (*i.e.* similar gestures having their corresponding paths end up in different places in the path ordering) are a potential source of misclassifications. Thus, it was thought that a multi-path recognition method that avoided path sorting might potentially be more accurate.

5.7.1 Global features without path sorting

The first step was to create a global feature set which did not rely on path sorting. As usual, a major design criterion was that a small change in a gesture should result in a small change in its global features. Thus, features which depend largely upon the precise order that paths begin cannot be used, since two paths which start almost simultaneously may appear in either order. However, such features can be weighted by the difference in starting times between successive paths, and thus vary smoothly as paths change order. Another approach which avoids the problem is to create global features which depend on, say, every pair of paths; these too would be immune to the problems of path sorting.

The global features are based on the previous global features discussed. However, for each feature which relied on path ordering there, two features were used here. The first was the previous feature weighted by path start time differences. For example, one feature is the length from the first point of the first path to the first point of the last path, multiplied by the difference between the start times of the first and second path, and again multiplied by the difference between the start times of the last and next to last path. The second was the sum of the feature between every pair path, such as the sum of the length between the start points of every pair of paths. For the sine and cosine features, the sum of the absolute values was used.

5.7.2 Multi-path recognition using one single-path classifier

Path sorting allows there to be a number of different path classifiers, one for the first path, one for the second, and so on. To avoid path sorting, a single classifier is used to classify all paths. Referring to the example in Section 5.4, a single classifier would be used to distinguish between A_1, A_2, B_1, and B_2.

Once all the paths in a gesture are classified, the class information needs to be combined to produce a classification for the gesture as a whole. As before, a decision tree is used. However, since path sorting has been eliminated, there is now no apparent order of the classes which will make up the sequence submitted to the decision tree. To remedy this, each path class is assigned an arbitrary distinct integer during training. The path class sequence is sorted according to this integer ranking (the global feature classification remains last in the sequence) and then the decision tree is examined. The net result is that each node in the decision tree corresponds to a set (rather than a sequence) of path classifications. (Actually, as will be explained later, each node corresponds to a multiset.)

In essence, the recognition algorithm is very simple: the lone path classifier determines the classes of all the paths in the gesture; this set of path classes, together with the global feature class, determines the class of the gesture. Unfortunately, this explanation glosses over a serious conceptual difficulty: In order to train the path classifier, known instances of each path class are required. But, without path sorting, how is it possible to know which of the two paths in an instance of gesture class A is A_1 and which is A_2? One of the paths of the first A example can arbitrarily be called A_1. Once this is done, which of the paths in each of the other examples of class A are in A_1?

Once asked, the answer to this question is straightforward. The path in the second instance of A which is similar to the path previously called A_1 should also be called A_1. If a gesture class has N paths, the goal is to divide the set of paths used in all the training examples of the class into N groups, each group containing exactly one path from each example. Ideally, the paths forming a group are similar to each other, or, in other words, they correspond to one another.

Note that path sorting produces exactly this set of groups. Within all the examples of a given gesture class, all paths with the same sorting index form a group. However, if the purpose of the endeavor is to build a multi-path recognizer which does not use path sorting, it seems inappropriate to resort to it during the training phase. Errors in sorting the example paths would get built into the path classifier, likely nullifying any beneficial effects of avoiding path sorting during recognition.

Another way to proceed is by analogy. Within a given gesture class, the paths in one example are compared to those of another example, and the corresponding paths are identified. The comparisons could conceivably be based on the feature of the path as well as the location and timing of the path. This approach was not tried, though in retrospect it seems the simplest and most likely to work well.

5.7.3 Clustering

Instead, the grouping of similar paths was attempted. The definition of similarity here only refers to the feature vector of the path. In particular, the relative location of the paths to one another was ignored. To group similar paths together solely on the basis of their feature vectors, a statistical procedure known as *hierarchical cluster analysis* [74] was applied.

The first step in cluster analysis is to create a triangular matrix containing the distance between every pair of samples, in this case the samples being every path of every example of a given class. The distance was computed by first normalizing each feature by dividing by the standard deviation. (The typical normalization step of first subtracting out the feature mean was omitted since it has no effect on the difference between two instances of a feature.) The distance between each pair of

APLNDC00022056

example path feature vectors was then calculated as the sum of the squared differences between the normalized features.

From this matrix, the clustering algorithm produces a cluster tree, or *dendrogram*. A dendrogram is a binary tree with an additional linear ordering on the interior nodes. The clustering algorithm initially considers each individual sample to be in a group (cluster) of its own, the distance matrix giving distances between every pair of groups. The two most similar groups, *i.e.* the pair corresponding to the smallest entry in the matrix, are combined into a single group, and a node representing the new group is created in the dendrogram, the subnodes of which refer to the two constituent groups. The distance matrix is then updated, replacing the two rows and columns of the constituent groups with a single row and column representing the composite group.

The distance of the composite group to each other group is calculated as a function of the distances of the two constituents to the other group. Many such combining functions are possible; the particular one used here is the *group average* method, which computes the distance of the newly formed group to another group as the average (weighted by group size) of the two constituent groups to the other group. After the matrix is updated, the process is repeated: the smallest matrix element is found, the two corresponding groups combined, and the matrix updated. This continues until there is only one group left, representing the entire sample set. The order of node creation gives the linear order on the dendrogram nodes, nodes created early having subnodes whose groups are more similar than nodes created later.

Figure 5.4 shows the dendrogram for the paths of 10 3-path clasp gestures, where the thumb moves slightly right while the index and middle fingers move left. The leaves of the dendrogram are labeled with the numbers of the paths of the examples. Notice how all the right strokes cluster together (one per example), as do all the left strokes (two per example).

Using the dendrogram, the original samples can be broken into an arbitrary (between one and the number of samples) number of groups. To get N groups, one simply discards the top $N-1$ nodes of the dendrogram. For example, to get two groups, the root node is discarded, and the two groups are represented by the two branches of the root node.

Turning back now to the problem of finding corresponding paths in examples of the same multi-path gesture class, the first step is to compute the dendrogram of all the paths in all examples of the gesture. The dendrogram is then traversed in a bottom-up (post-order) fashion, and at each node a histogram that indicates the count of the number of paths for each example is computed. The computation is straightforward: for each leaf node (*i.e.* for each path) the count is zero for all examples except the one the path came from; for each interior node, each element of the histogram is the sum of the corresponding elements of the subnode's histogram.

Ideally, there will be nodes in the tree whose histogram indicates that all the paths below this node come from different examples, and that each example is represented exactly once. In practice, however, things do not work out this nicely. First, errors in the clustering sometimes group two paths from the same example together before grouping one path from every example. This case is easily handled by setting a threshold, *e.g.* by accepting nodes in which paths from all but two examples appear exactly once in the cluster.

The second difficulty is more fundamental. It is possible that two or more paths in a single gesture are quite similar (remember that relative path location is being ignored). This is actually

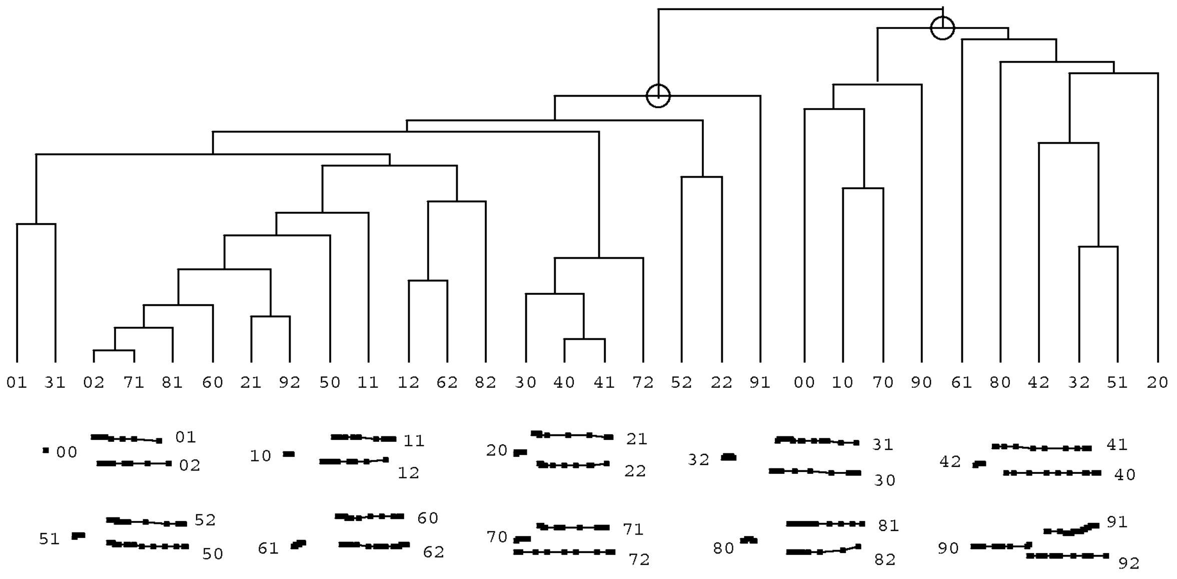

Figure 5.4: Path Clusters

This shows the result of clustering applied to the thirty paths of the ten three-path clasp *gestures shown. Each* clasp *gesture has a short, rightward moving path and two similar, long leftward moving paths. The hierarchical clustering algorithm groups similar paths (or groups of paths) together. The height of an interior node indicates the similarity of its groups, lower nodes being more similar. Note that the right subtree of the root contains 10 paths, one from each multi-path gesture. It is thus termed a good cluster (indicated by a circle on the graph), and its constituent paths correspond. The left subtree containing 20 paths, two from each gesture, is also a good cluster. Had one of its descendants been another good cluster (containing approximately 10 paths, one from each gesture), it would have been concluded that all three paths of the* clasp *gesture are different, with the corresponding paths given by the good clusters. As it happened, no descendant of the left subtree was a good cluster, so it is concluded that two of the paths within the* clasp *gesture are similar, and will thus be treated as examples of one single-path class.*

common for Sensor Frame gestures that are performed by moving the elbow and shoulder while keeping the wrist and fingers rigid. For these paths, it is just as likely that the two paths of the same example be grouped together as it is that corresponding paths of different examples be grouped together. Thus, instead of a histogram that shows one path from each example, ideally there will be a node with a histogram containing two paths per example. This is the case in figure 5.4.

Call a node which has a histogram indicating an equal (or almost equal) number of paths from each example a *good cluster*. The search for good clusters proceeds top down. The root node is surely a good cluster; *e.g.* given examples from a three path gesture class, the root node histogram will indicate three paths from each example gesture. If no descendants of the root node are good clusters, that indicates that all the paths of the gesture are similar. However, if there are good clusters below the root (with fewer examples per path than the root), that indicates that not all the paths of the gesture are similar to each other. In the three path example, if, say, one subnode of the root node was a good cluster with one path per example, those paths form a distinct path class, different than the other path classes in the gesture class. The other subnode of the root will also be a good cluster, with two paths per example. If there does not exist a descendant of that node which is a good cluster with one path per example, that indicates that the two gesture paths classes are similar. Otherwise, good clusters below that node (there will likely be two) indicate that each path class in the gesture class is different. The cluster analysis, somewhat like the path sorting, indicates which paths in each example of a given gesture class correspond. (Good clusters are indicated by circles in figure 5.4.)

Occasionally, there are *stragglers*, paths which are not in any of the good clusters identified by the analysis. An attempt is made to put the stragglers in an appropriate group. If an example contains a single straggler it can easily be placed in the group which is lacking an example from this class. If an example contains more than one straggler, they are currently ignored. If desired, a path classifier to discriminate between the good clusters could be created and then used to classify the stragglers. This was not done in the current implementation since there was never a significant number of stragglers.

Once the path classes in each gesture class have been identified using the clustering technique, a path classifier is trained which can distinguish between every path class of every gesture class. Note that it is possible for a path class to be formed from two or more paths from each example of a single gesture class, if the cluster analysis indicated the two paths were similar. If analogy techniques were used to separate such a class into multiple “one path per example” classes, the resulting classifier would ambiguously classify such paths. In any case, ambiguities are still possible since different gesture classes may have similar gesture paths. As in Section 5.4, the ambiguities are removed from the classifier by combining ambiguous classes into a single class. Each (now unambiguous) class which is recognized by the path classifier is numbered so as to establish a canonical order for sorting path class sequences during training and recognition.

5.7.4 Creating the decision tree

After the single-path and global classifiers have been trained, the decision tree must be constructed. As before, in the class phase, for each multi-path gesture class, the (now unambiguous) classes of each constituent path are enumerated. Since two paths in a single gesture class may be similar, this enumeration of classes may list a single class more than once, and thus may be considered a multiset. The list of classes is sequenced into canonical order, the global feature class appended, and the resulting sequence is used to add the multi-path class to the decision tree. As before, a conflict, due to the fact that two different gesture classes have the same multiset of path classes, is fatal.

Next comes the example phase. The paths of each example gesture are classified by the single path classifier, and the resulting sequence (in canonical order with the global feature class appended) is used to add the class of the example to the decision tree. Usually no work needs to be done, as the same sequence has already been used to add this class (usually in the class phase). However, if one of the paths in the sequence has been misclassified, adding it to the decision tree can improve recognition, since this misclassification may occur again in the future. Conflicts here are not fatal, but are simply ignored on the assumption that the sequences added in the class phase are more important than those added in the example phase.

5.8 Discussion

Two multi-path gesture recognition algorithms have been described, which are referred to as the "path sorting" and the "path clustering" methods. In situations where there is no uncertainty as to the path index information (*e.g.* a DataGlove, since the sensors are attached to the hand) then the path-sorting method is certainly superior. However, with input devices such as the Sensor Frame, the path sorting has to be done heuristically, which increases the likelihood of recognition error.

The path-clustering method avoids path sorting and its associated errors. However, other sources of misclassification are introduced. One single-path classifier is used to discriminate between all the path classes in the system, so will have to recognize a large number of classes. Since the error rate of a classifier increases with the number of classes, the path classifier in a path-clustering algorithm will never perform as well as those in a path-sorting algorithm. A second source of error is in the clustering itself; errors there cause errors in the classifier training data, which cause the performance of the path classifier to degrade. One way around this is to cluster the paths by hand rather than by having a computer perform it automatically. This needed to be done with some gesture classes from the Sensor Frame, which, because of glitches in the tracking hardware, could not be clustered reliably.

In practice, the path-sorting method always performed better. The poor performance of the path-clustering method was generally due to the noisy Sensor Frame data. It is however difficult to reach a general conclusion, as all the gesture sets upon which the methods were tested were designed with the path sorting algorithm in mind. It it easy to design a set of gestures that would perform poorly using sorted paths. One possibility for future work is to have a parameterizable algorithm for sorting paths, and choose the parameters based on the gesture set.

The Sensor Frame itself was a significant source of classification errors. Sometimes, the knuckles of fingers curled so as not to be sensed would inadvertently break the sensing plane, causing extra paths in the gesture (which would typically then be rejected). Also, three fingers in the sensing plane can easily occlude each other with respect to the sensors, making it difficult for the Sensor Frame to determine each finger's location. The Sensor Frame hardware usually knew from recent history that there were indeed three fingers present, and did its best to estimate the positions of each. However, the resulting data often had glitches that degraded classification, sometimes by confusing the tracking algorithm. It is likely that additional preprocessing of the paths before recognition would improve accuracy. Also, the Sensor Frame itself is still under development, and it is possible that such glitches will be eliminated by the hardware in the future.

Another area for future work is to apply the single-path eager recognition work described in Chapter 4 to the eager recognition of multi-path gestures. Presumably this is simply a matter of eagerly recognizing each path, and combining the results using the decision tree. How well this works remains to be seen.

It would also be possible to apply the multi-path algorithm to the recognition of multi-stroke gestures. The path sorting in this case would simply be the order that the strokes arrive. To date, this has not been tried.

5.9 Conclusion

In this chapter, two methods for multi-path gesture recognition were discussed and compared. Each classifies the paths of the gesture individually, uses a decision tree to combine the results, and uses global features to resolve any lingering ambiguities. The first method, path sorting, builds a separate classifier for each path in a multi-path gesture. In order to determine which path to submit to which classifier, either the physical input device needs to be able to tell which finger corresponds to which path, or a path sorting algorithm numbers the paths. The second method, path clustering, avoids path sorting (which has an arbitrary component) by using one classifier to classify all the paths in a gesture.

In general, the path sorting method proved superior. However, when the details of the path sorting algorithm are known it is possible to design a set of gestures which will be poorly recognized due to errors in the path sorting. That same knowledge can also be used to design gesture sets that will not run into path sorting problems.

Chapter 6

An Architecture for Direct Manipulation

This chapter describes the GRANDMA system. GRANDMA stands for "Gesture Recognizers Automated in a Novel Direct Manipulation Architecture." This chapter concentrates solely on the architecture of the system, without reference to gesture recognition. The design and implementation of gesture recognizers in GRANDMA is the subject of the next chapter.

GRANDMA is an object-oriented toolkit similar to those discussed in Section 2.4.1. Like those toolkits, is it based on the model-view-controller (MVC) paradigm. GRANDMA also borrows ideas from event-based user-interface systems such as Squeak [23], ALGAE [36], and Sassafras [54].

GRANDMA is implemented in Objective C [28] on a DEC MicroVax-II running UNIX and the X10 window system.

6.1 Motivation

Building an object-oriented user interface toolkit is a rather large task, not to be undertaken lightly. Furthermore, such toolkits are only peripherally related to the topic at hand, namely gesture-based systems. Thus, the decision to create GRANDMA requires some justification.

A single idea motivated the author to use object-oriented toolkits to construct gesture-based systems: gestures should be associated with objects on the screen. Just as an object's class determines the messages it understands, the author believed the class could and should be used to determine which gestures an object understands. The ideas of inheritance and overriding then naturally apply to gestures. The analogy of gestures and messages is the central idea of the "systems" portion of the current work.

It would have been desirable to integrate gestures into an existing object oriented toolkit, rather than build one from scratch. However, at the time the work began, the only such toolkits available were Smalltalk-80's MVC [70] and the Pascal-based MacApp [115], neither of which ran on the UNIX/C environment available to (and preferred by) the author. Thus, the author created GRANDMA.

The existing object-oriented user interface systems tend to have very low-level input models, with device dependencies spread throughout the system. For example, some systems require views to respond to messages such as `middleButtonDown` [28]; others use event structures that can only represent input from a fixed small set of devices [102]. In general, the output models of existing systems seem to have received much more attention than the input models. One goal of GRANDMA was to investigate new architectures for input processing.

6.2 Architectural Overview

Figure 6.1 shows a general overview of the architecture of the GRANDMA system. In order to introduce the architecture to the reader, the response to a typical input event is traced. But first, a brief description of the system components is in order.

GRANDMA is based on the Model-View-Controller (MVC) paradigm. Models are application objects. They are concerned only with the semantics of the application, and not with the user interface. Views are concerned with displaying the state of models. When a model changes, it is the responsibility of the model's view(s) to relay that change to the user. Controllers are objects which handle input. In GRANDMA, controllers take the form of *event handlers*.[1] A single passive event handler may be associated with many view objects; when input is first initiated toward a view, one of the view's passive event handlers may activate (a copy of) itself to handle further input.

6.2.1 An example: pressing a switch

Consider a display consisting of several toggle switches. Each toggle switch has a model, which is likely to be an object containing a boolean variable. The model has messages to set and retrieve the value of the variable, which are used by the view to display the state of the toggle switch, and by the event handler to change the state of the toggle.

When the mouse cursor is moved over one of the switches and, say, the left mouse button is pressed, the window manager informs GRANDMA, which raises an input `Pick` event. The event is an object which groups together all the information about the event: the fact that it was a mouse event, which button was pressed, and, most significantly, the coordinates of the mouse cursor.

Raising an event causes the *active event handler* list to be searched for a handler for this event. In turn, each event handler on the list is asked if it wishes to handle the event. Assuming none of the other handlers will be interested in the event, the last handler in the list, called the `XYEventHandler`, handles the event. This is what happens in the case of pressing the toggle switch.

The `XYEventHandler` is able to process any event at a location (*i.e.* events with X-Y coordinates). The handler first searches the *view database* and constructs a list of views which are "under" the event, in other words, views that are at the given event location. The search is simple: each view has a rectangular region in which it is included; if the event location is in the rectangle, the view is added to the list. In the switch example, the list of views consists of the indicated toggle switch view followed by the view representing the window in which the toggle switch is drawn.

[1]The distinction between controllers and event handlers is in the way each interacts with the underlying layer that generates input events. Once activated, controllers loop, continually calling the input layer for all input events until the interaction completes. In other words, controllers take control, forcing the user to complete one interaction before initiating the next. In contrast, event handlers are essentially called by the input layer whenever input occurs. It is thus possible to interact simultaneously with multiple event handlers, for example via multiple devices.

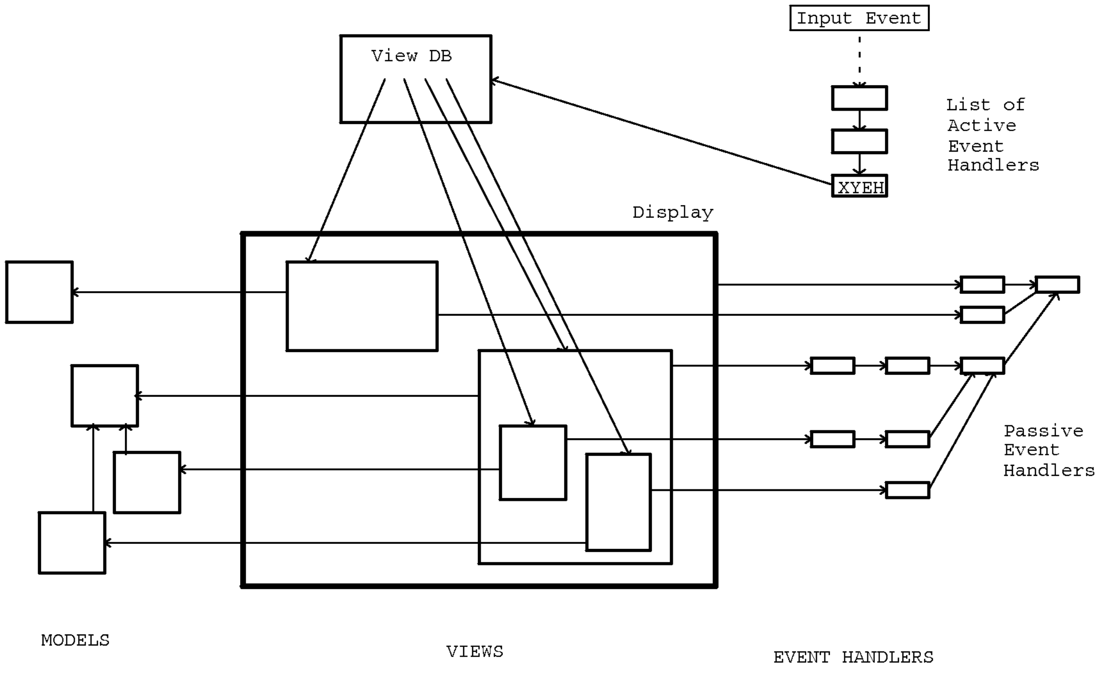

Figure 6.1: GRANDMA's Architecture

*In GRANDMA, user actions cause events to be raised (*i.e. *pressing a mouse button raises a* `Pick` *event). Each handler on the active event handler list is asked, in order, if it wishes to handle the event. The* `XYEventHandler`*, last on the list, is asked only if none of the previous active handlers have consumed the event. For an event with a screen location (i.e. a mouse event), the* `XYEventHandler` *uses the view database to determine the views at the given screen location, and asks each view (from front to back) if it wishes to handle the event. To answer, a view consults its list of passive event handlers, some associated with the view itself, others associated with the view's class and superclasses, to see if one of those is interested in the event. If so, that passive handler may activate itself, typically by placing a copy of itself at the front of the active event handler list. This enables subsequent events to be handled efficiently, short-circuiting the elaborate search for a handler initiated by the* `XYEventHandler`*. An event handler only consumes events in which it is interested, allowing other events to propagate to other event handlers.*

The views are then queried starting with the foreground view. First, a view is asked if the event location is indeed over the view; this gives an opportunity for a non-rectangular view to respond only to events directly over it. If the event is indeed over the view, the view is then asked if it wishes to handle the input event. The search proceeds until a view wishes to handle the event, or all the views under the event have declined. In the example, the toggle switch view handles the event, which would then not be propagated to the window view.

A view does not respond directly to a query as to whether it will handle an input event. Instead, that request is passed to the view's *passive event handlers*. Associated with each view is a list of event handlers that handle input for the view; a single passive event handler is often shared among many views in the system. The passive event handlers are each asked about the input in turn; the search stops when one decides to handle the input. In the example, the toggle switch has a toggle switch event handler first on its list of passive handlers that would handle the `Pick` event.

A passive event handler that has decided to handle an event may *activate* a copy or instance of itself, *i.e.* place the copy or instance in the active event handler list. Or, it may not, choosing to do all the work associated with the event when it gets the event. For example, a toggle switch may either change state immediately when the mouse button is pressed over the switch, or it may simply highlight itself, changing state only if the button is released over the switch. In the former case, there is no need to activate an event handler; the passive handler itself can change the state of the switch.

In the latter case, the passive handler activates a copy of itself which first highlights the switch, and then monitors subsequent input to watch if the cursor remains over the view. If the cursor moves away from the view, the active event handler will turn off the highlighting of the switch, and may (depending on the kind of interaction wanted) deactivate itself. Finally, if the mouse button is released over the switch, the active event handler will, through the view, toggle the state of the switch (and associated model), and then deactivate itself.

As noted above, active handlers are asked about events before the view database is searched and any passive handlers queried. Thus, in the switch example, subsequent mouse movements made while the button is held down, or the release of the mouse button, will be handled very efficiently since the active handler is at the head of the active event handler list.

6.2.2 Tools

The *tool* is one component of GRANDMA's architecture not mentioned in the above example. A tool is an object that raises events, and it is through such events that tools operate on views (and thus models) in the system. An event handler may be considered the mechanism through which a tool operates upon a view. The interaction is by no means unidirectional: some event handlers cause views to operate upon tools as well. In addition to operating on views directly, event handlers may themselves raise events, as will be seen.

Every event has an associated tool which typically refers to the device that generated the event. For example, a system with two mice would have two `MouseTool` objects, and the appropriate one would be used to identify which mouse caused a given `Pick` event. When asked to handle an event, an active handler typically checks that the event's tool is the same one that caused the handler to be activated in the first place. In this manner, the active event handler ignores events not intended for it.

Tools are also involved when one device emulates another. For example, a Sensor Frame may emulate a mouse by having an active handler that consumes events whose tool is a `SensorFrame` object, raising events whose tool is a `MouseTool` in response. That `MouseTool` does not correspond to a real mouse; rather, it allows the Sensor Frame to masquerade as a mouse.

Tools do not necessarily refer to hardware devices. *Virtual tools* are software objects (typically views) that act like input hardware in that they may generate events. For example, file views (icons) would be virtual tools when implementing a Macintosh-like Finder in GRANDMA. Dragging a file view would cause events to be raised in which the tool was the file view. A passive handler associated with folder (directory) views would be programmed to activate whenever an event whose tool is a file view is dragged over a folder. Thus, in GRANDMA the same mechanism is used when the mouse cursor is dragged over views as when the mouse is used to drag one view over other views.

The typical case, in which a tool has a semantic action which operates upon views that the tool is dropped upon, is handled gracefully in GRANDMA. Associated with every view is the passive `GenericToolOnViewEventHandler`. When a tool is dragged over a view which responds to the tool's action, the `GenericToolOnViewEventHandler` associated with the view activates itself, highlighting the view. Dropping the tool on the view causes the action to occur.[2] Thus, semantic feedback is easy to achieve using virtual tools (see section 6.7.7).

This concludes the brief overview of the GRANDMA architecture. A discussion of the details of the GRANDMA system now follows. A reader wishing to avoid the details may proceed directly to section 6.8, which summarizes the main points while comparing GRANDMA to some existing systems.

6.3 Objective-C Notation

As mentioned, GRANDMA is written in Objective C [28], a language which augments C with object-oriented programming constructs. In this part of the dissertation, program fragments will be written in Objective C.

In Objective C, variables and functions whose values are objects are all declared type `id`, as in

```
id aSet;
```

Variables of type `id` are really pointers, and can refer to any Objective C object, or have the value `nil`. Like all pointers, such variables need to be initialized before they refer to any object:

```
aSet = [Set new]; /* create a Set object */
```

The expression `[o messagename]` is used to send the message `messagename` to the object referred to by `o`. This object is termed the *receiver*, and `messagename` the *selector*. A message send is similar to a function call, and returns a value whose type depends on the selector.

Objective C comes supplied with a number of *factory* objects, also known as *classes*. `Set` is an example of a factory object, and like most factory objects, responds to the message `new` with a

[2]The related case, in which a tool is dragged over a view that acts upon the tool (*e.g.* the trash can), is handled by the `BucketEventHandler`.

newly allocated instance of itself.

Messages may also have parameters, as in

```
        id aRect = [Rectangle origin:10:10 corner:20:30];
        [aSet add:aRect];
```

The message selector is the concatenation of all the parameter labels (`origin::corner::` in the first case, `add:` in the second). In all cases, there is one parameter after each colon.

A factory's fields and methods are declared as in the following example:

```
    = Rect:Object { int x1,y1,x2,y2; }
    + origin:(int)_x1 :(int):_y1 corner:(int)_x2 :(int)_y2 {
        self = [self new];
        x1 = _x1, y1 = _y1, x2 = _x2; y2 = _y2;
        return self;
    }
    − shiftby:(int)x :(int)y
        { x1 += x; y1 += y; x2 += x; y2 += y; return self; }
```

This declares the factory `Rect` to be a subclass of the factory `Object`, the root of the class hierarchy. Note that the factory declaration begins with the "=" token. A method declared with "+" defines a message which is sent directly to a factory object; such methods often allocate and return an instance of the factory. A method declared with "`-`" defines a message that is sent directly to instances of the class. The variable `self` is accessible in all method declarations; it refers to the object to which the message was sent (the receiver). When `self` is set to an instance of the object class being defined, the fields in the object can be referenced directly. Thus, as in the `origin::corner::` method, the first step of a factory method is often to reassign `self` to be an instance of the factory, then to initialize the fields of the instance. The usage `[self new]` rather than `[Rect new]` allows the method to work even when applied to a subclass of `Rect` (since in that case `self` would refer to the factory object of the subclass). When the types of methods and arguments are left unspecified, they are assumed to be `id`, and typically methods return `self` when they have nothing better to return (rather than `void`, *i.e.* not returning anything).

When describing a method of a class, the fields and other methods are often omitted, as in

```
    = Rect ...
    − (int)area { return abs( (x2−x1)∗(y2−y1) ); }
```

In Objective C, messages selectors are first class objects, which can be assigned and passed as parameters and then later sent to objects. The construct `@selector(`*message-selector*`)` returns an object of type `SEL`, which is a runtime representation of the message selector:

```
        id aRect = [Rect origin:10:5 corner:40:35];
        SEL op = flag ?   @selector(area) :  @selector(height);
        printf("%d\n", [aRect perform:op]);
```

The rectangle `aRect` will be sent the `area` or `height` message depending on the state of `flag`. The `perform:` message sends the message indicated by the passed `SEL` to an object. Variants of the form `perform:with:with:` allow additional parameters to be sent as well.

The first class nature of message selectors distinguishes Objective C from more static object-oriented languages, notably C++. As they are analogous to pointers to functions in C, `SEL` values may be considered "pointers" to messages. Objective C includes functions for converting between `SEL` values and strings, and a method for inquiring at runtime whether an object responds to an arbitrary message selector. As will be seen, these Objective C features are often used in the GRANDMA implementation.

In the interest of simplicity, debugging code and memory management code have been removed from most of the code fragments shown below, though they are of course needed in practice. Also, as the code is explained in the text, many of the comments have been removed for brevity.

6.4 The Two Hierarchies

Thus far, two important hierarchies in object-oriented user interface toolkits have been hinted at, and it seems prudent to forestall confusion by further discussing them here. The first one is known as the *class hierarchy*. The class hierarchy is the tree of subclass/superclass relationships that one has in a single-inheritance system such as Objective C. In Objective C, the class `Object` is at the root of the class hierarchy; in GRANDMA classes like `Model`, `View`, and `EventHandler` are subclasses of `Object`; each of these has subclasses of its own (*e.g.* `ButtonView` is a subclass of `View`), and so on. The entire tree is referred to as the class hierarchy, and particular subtrees are referred to by qualifying this term with a class name. In particular, the `View` class hierarchy is the tree with the class `View` at the root, with the subclasses of `View` subnodes of the root, and so on.

The second hierarchy is referred to as the *view hierarchy* or *view tree*. A `View` object typically controls a rectangular region of the display window. The view may have *subviews* which control subareas of the parent view's rectangle. For example, a dialogue box view may have as subviews some radio buttons. Subviews are usually more to the foreground than their parent views; in other words, a subview usually obscures part of its parent's view. Of course, subviews themselves might have subviews, and so on, the entire structure being known as the view tree. In GRANDMA, the root of the view tree is a view corresponding to a particular window on the display; a program with multiple windows will have a view tree for each. It is important not to confuse the view hierarchy with the `View` class hierarchy; the former refers to the superview/subview relation, the latter to the superclass/subclass relation.

6.5 Models

Being a Model/View/Controller-based system, naturally the three most important classes in GRANDMA are `Model`, `View`, and `EventHandler` (the latter being GRANDMA's term for controller). The discussion of GRANDMA is divided into three sections, one for each of these classes. Class `Model` is considered first.

Models are objects which contain application-specific data. Model objects encapsulate the data and computation of the task domain. The MVC paradigm specifies that the methods of models should not contain any user-interface specific code. However, a model will typically respond to messages inquiring about its state. In this manner, a view object may gain information about the model in order to display a representation of the model.

In a number of MVC-like systems, there is no specific class named "`Model`" [28]. Instead, any object may act as a model. However, in GRANDMA, as in Smalltalk-80[70], there is a single class named `Model`, which is subclassed to implement application objects. This has the obvious disadvantage that already existing classes cannot directly serve as models. The advantage is the ease of implementation, and the ability to easily distinguish models from other objects.

One of the tenets of the MVC paradigm is that `Model` objects are independent of their views. The intent is that the user interface of the application should be able to be changed without modifying the application semantics. The effect of this desire for modularity is that a `Model` subclass is written without reference to its views.

However, when the state of a model changes, a mechanism is needed to inform the views of the model to update the display accordingly. The way this is accomplished is for each model to have a list of dependents. Objects, such as views, that wish to be informed when a model changes state register themselves as dependents of the model. By convention, a `Model` object sends itself the `modified` message when it changes; this results in all its dependents getting sent the `modelModified` message, at which time they can act accordingly.

The heart of the implementation of the `Model` class in GRANDMA is simple and instructive:

```
= Model :  Object { id dependents; }
- addDependent:d {
     if(dependents == nil) dependents = [OrdCltn new];
     [dependents add:d];
     return self;
}
- removeDependent:d {
     if(dependents != nil) [dependents remove:d];
     return self;
}
- modified {
     if(dependents != nil) /* send all dependents modelModified */
        [dependents elementsPerform:@selector(modelModified)];
     return self;
}
```

Thus, a `Model` is a subclass of `Object` with one additional field, `dependents`. When a `Model` is first created, its `dependents` field is automatically set to `nil`. The first time a dependent is added (by sending the message `addDependent:`), the `dependents` field is set to a new instance of `OrdCltn`, a class for representing lists of objects. The dependent is then added to the list; it can later be removed by the `removeDependent` message.

`Model` is an *abstract* class; it is not intended to be instantiated directly, but instead only be subclassed. A simple example of a `Model` might be boolean variable (whose view might be a toggle switch):

```
= Boolean :  Model { BOOL state; }
- (BOOL)getState { return state; }
```

```
    – setState:(BOOL)_state
        { state = _state; return [self modified]; }
    – toggle { state = !state; return [self modified]; }
```

Notice that whenever a `Boolean` object's state changes, it sends itself the `modified` message, which results in all of its dependents getting sent the `modelModified` message.

6.6 Views

The abstract class `View`, as mentioned, handles the display of `Models`. It is easily the most complex class in the GRANDMA system; it is over 800 lines of code, and it currently implements 10 factory methods and 67 instance methods (not including those inherited from `Object`). For brevity, most of the methods will not be mentioned, or are only mentioned in passing.

Views have a number of instance variables (fields):

```
    = View :  Object {
        id    model;
        id    parent, children;
        id    picture, highlight;
        short xloc, yloc;
        id    box;
        int   state;
    }
```

The `model` variable is the view's connection with its model. Some views have no model; in this case `model` will be `nil`. The fields `parent` and `children` implement the view tree, `parent` being the superview of the view, `children` being a list (`OrdCltn`) of the subviews of this view. The fields `picture` and `highlight` refer to the graphics used to draw and highlight the view, respectively. The graphics are drawn with respect to the origin specified by `(xloc, yloc)`, and are constrained to be within the `Rectangle` object `box`. The `state` field is a set of bits indicating both the current state of the view (set by the GRANDMA system) and the desired state of the view (controllable by the view user).

To illustrate some of `View`'s methods, here is a toggle switch view whose model is the class `Boolean` described above.

```
    = SwitchView:  View  { }
```

To create a toggle switch view:

```
        id aBoolean = [Boolean new];
        id aSwitchView = [SwitchView createViewOf:aBoolean];
```

The `createViewOf:` method of class `View` allocates a new view object (in this case an instance of `SwitchView`), sets the `model` instance variable, and, to add itself to the model's dependents, does `[model addDependent:self]`.

The graphics for the switch are implemented as:

```
    = SwitchView ...
    – updatePicture {
        id p = [self VbeginPicture];
        [p rectangle 0:0 :10:10];
        if([model getState])
            [p rectangle 2:2 :8:8];
        [self VendPicture];
        return self;
    }
```

The intention is to draw an empty rectangle 10 by 10 pixels in size for a switch whose model's state is FALSE, but put a smaller rectangle within the switch when the model's state is TRUE.

View's VbeginPicture and VendPicture methods deal with the picture instance variable. (The V prefix in the method names is a convention indicating that these messages are intended only to be sent by subclasses of View.) VbeginPicture creates or initializes the HangingPicture object which it returns. The graphics are then directed at the picture, which is in essence a display list of graphics commands. Note how the model's state is queried using the model instance variable inherited from class View. This is done for efficiency purposes; a more modular way to accomplish the same thing would be if([[self model] getState]).

The method updatePicture gets called indirectly from View's modelModified method:

```
    = View ...
    - modelModified {
        [self update];
        if(state & V_NOTIFY_CHILDREN)  /* propagate modelModified to kids */
            [children elementsPerform:@selector(modelModified)];
        return self;
    }
    - update { return [self updatePicture]; }
```

The state bit V_NOTIFY_CHILDREN is settable by the creator of a view; it determines whether modelModified messages will be propagated to subviews. Often when this bit is turned off, the subclass of View overrides the update method in order to propagate modelModified only to certain of its subviews. (For example, a view whose model is a list might have a subview for each element in the list displayed left to right, and when one element is deleted from the list the view could arrange that only the subviews to the right of the deleted one be redrawn.) In the more typical case, the subclass only implements the updatePicture method which redraws the view to reflect the state of the model.

For the switch to be displayed, it needs to be a subview (or a descendant) of a WallView. Class WallView is the abstraction of a window on the display. An instance of WallView is created for each window a program requires, as in:

```
        id aWallView = [WallView name:"gdp"];
        [aWallView addSubView:[aSwitchView at:50:30]];
```

This fragment creates a window named "gdp." The string "gdp" is looked up in a database (in this case, the .Xdefaults file as administered by the X window system) to determine the initial size and location of the window. The switch is added as a subview to the wall view, and displayed at coordinates (50,30) in the newly created window.

This ends the discussion of the major methods of class `View`. As the need arises, additional methods will be discussed. It is ironic how in this dissertation, largely concerned with input, so much effort was expended on output. The initial intention was to keep the output code as simple as possible while still being usable. Unfortunately, thousands of lines of code were required to get to that point.

6.7 Event Handlers

In GRANDMA, the analogue of MVC controllers are event handlers. When input occurs, it is represented as an *event* which is *raised*. Raising an event results in a search for an active event handler that will handle the event. For many events, the last handler in the active list is a catch-all handler whose function is to search for any views at the event's location. Each such view is asked if it wishes to handle the event; the view then asks each of its passive event handlers if it wants to handle the event. As mentioned, a single passive event handler may be associated with many different views. A passive event handler may activate a copy or instance of itself in response to input.

Warning to readers: due to this dissertation's focus on input, this is necessarily a very long section.

6.7.1 Events

Before event handlers can be discussed in detail, it is helpful to make concrete exactly what is meant by "event." All events are instances of some subclass of `Event`:

```
= Event :  Object { id instigator; }
– instigator { return instigator; }
– instigator:_instigator
     { instigator = _instigator; return self; }
```

The instigator of an event is the object posting the event. All window manager events are instigated by an instance of class `Wall`.[3]

Figure 6.2 shows the `Event` class hierarchy. (Like `instigator` in class `Event`, each instance variable shown has a method to set and a method to retrieve its value.) The most important subclass is `WallEvent`, which is an event associated with a window, and thus usually raised by (the GRANDMA interface to) the window manager. A `KeyEvent` is generated when a character is typed by the user. A `RefreshEvent` is generated when the window manager requests that a particular window be redrawn.

The subclasses of the abstract class `DragEvent`, when raised by the window manager, indicate a mouse event. In these cases, the `tool` field is an instance of `GenericMouseTool` or one of its subclasses. When a mouse button is pressed, a `PickEvent` is generated. The field `loc` is a

[3]The instigator is mostly used for tracing and debugging. Occasionally, it is used for a quick check by an active event handler that wishes to insure it is only handling events raised by the same object that raised the event which activated the handler in the first place. Most active handlers do not bother with this check, being content to simply check that the tool (rather than the instigator) is the same.

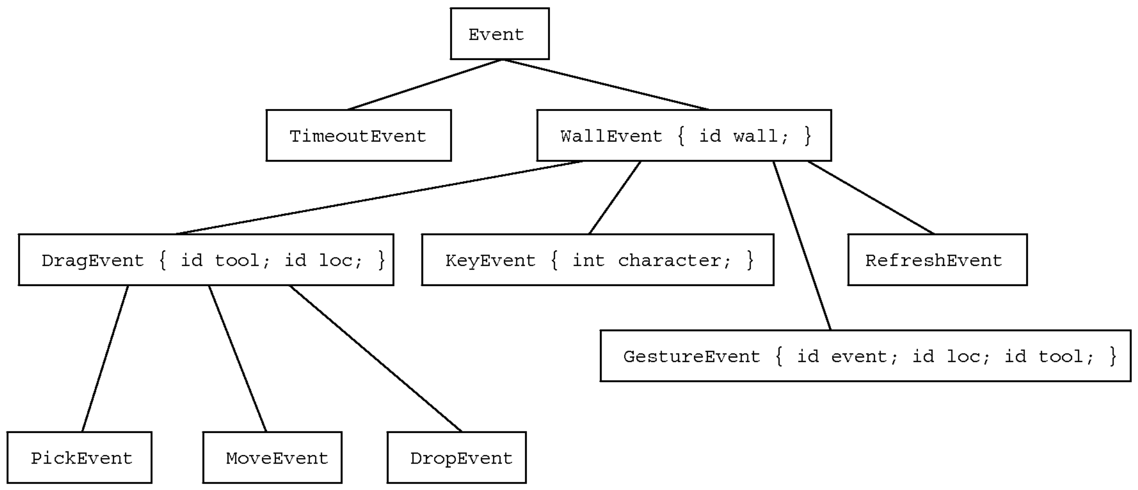
Event
TimeoutEvent
WallEvent { id wall; }
DragEvent { id tool; id loc; }
KeyEvent { int character; }
RefreshEvent
GestureEvent { id event; id loc; id tool; }
PickEvent
MoveEvent
DropEvent

Figure 6.2: The `Event` Hierarchy

`Point` object, indicating the location of the mouse cursor.[4] The mouse object referred to by the `tool` field indicates which button has been pressed. When the mouse is moved (currently only when a mouse button is pressed), a `MoveEvent` is generated. When the mouse button is released, a `DropEvent` is generated.

The classes `GestureEvent` and `TimeoutEvent` will be discussed in Chapter 7.

6.7.2 Raising an Event

A `WallView` object represents the root of the view tree of a given window. Associated with each `WallView` object is a `Wall` object which actually implements the interface between GRANDMA and the window manager. Also associated with each `WallView` object (*i.e.* each window) is an `EventHandlerList` object.

```
= WallView :  View { id handlers; id viewdatabase; id wall; }
+ name:(STR)name {
    self = [self createViewOf:nil];
    wall = [Wall create:name wallview:self];
    handlers = [EventHandlerList new];
    viewdatabase = [Xydb new];
    [handlers add:[XyEventHandler wallview:self]];
    return self;
}
- raise:event { return [handlers raise:event]; }
- viewdatabase { return viewdatabase; }
```

[4]In retrospect it probably would have been wiser either to always represent points and rectangles as C structures, or as separate coordinates, instead of using `Point` and `Rectangle` objects and their associated overhead.

```
= Wall :  Object (GRANDMA,Geometry)
    { Win win; id pictures; id wallview; }
- raise:event {
    if([event isKindOf:RefreshEvent])
        { [self redraw]; return; }
    return [wallview raise:event];
}
```

Events are raised within a particular window using the `raise` message. `Redraw` events are handled within the wall; since each wall maintains a list of `Picture` objects currently hung on it, redraw is easily accomplished. The `Redraw` special case is really just old code; it would be simple to replace this code with a redraw event handler. All other events are passed from the `Wall` to the `WallView` to the `EventHandlerList`:

```
= EventHandlerList :  OrdCltn { }
- raise:e { int i;
    for(i = [self lastOffset]; i >= 0; i--)
        if( [[self at:i] event:e] )
            break;
    return self;
}
```

An `EventHandlerList` is just an `OrdCltn`, thus `add:` and `remove:` messages can be sent to it to add or remove active event handlers. The `add:` message adds handlers to the end of the list; `raise` iterates through the list backwards, asking each element of the list in order if it wishes to handle the event. Thus, handlers activated most recently are asked about events before those activated earlier. (It is possible to install an active event handler at an arbitrary position in the `EventHandlerList` by using some of `OrdCltn`'s other methods, but this has never been needed in GRANDMA.) Note that the first thing a `WallView` object does when created is activate an `XyEventHandler`; this handler, since it is first in the list, will be tried only after the other handlers have declined to process the event.

6.7.3 Active Event Handlers

Every active event handler must respond to the `event:` message, returning a boolean value indicating whether it has handled the event.

```
= EventHandler :  Object { }
- (BOOL)event:e
    { return (BOOL)[self subclassResponsibility]; }
```

The `event:` method here is a placeholder for the actual method, which would be implemented differently in each subclass of `EventHandler`. The `subclassResponsibility` method is inherited from `Object`. The method simply prints an error message stating that the subclass of the receiver should have implemented the method.

Note that the `event:` message sent to the active event handlers has no reference to any views. When the event handler is first activated, it generally stores the view and tool which caused its activation[5]; it can then refer to these to decide whether to handle an event. When handling an event, the active event handler typically sends the view messages, if only to find out the model to which the view refers.

As previously mentioned, the last active event handler tried is the `XyEventHandler`. This event handler is rather atypical in that it never exists in a passive state.

```
= XyEventHandler :  EventHandler { id wallview; }
+ wallview:_wallview { self = [self new];
        wallview = _wallview; return self; }
– (BOOL)event:e { id views, seq, v, tool;
    if(!  [e respondsTo:@selector(loc) ]) return NO;
    views = [[wallview viewdatabase] at:[e loc]];
    tool = [e tool];
    for(seq = [views eachElement]; v = [seq next]; )
        if(v != tool && [v event:e]) return YES;
    return NO;
}
```

An `XyEventHandler` is instantiated and activated when a `WallView` is created (see Section 6.7.2). The `WallView` is recorded in the handler so that it can access the current database of views (those views in the `View` subtree of the `WallView`). (In retrospect, it would have been more efficient for the `XyEventHandler` to store a handle to the database directly, rather than always asking the `WallView` for it.)

When an `XyEventHandler` is asked to handle an event (via the `event:` message) it first checks to see if that event responds to the message `loc`. Currently, only (subclasses of) `DragEvents` respond to `loc`, but that could conceivably change in the future so the handler is written as generally as possible. This points to one of the major benefits of Objective C; one can inquire as to whether an object responds to a message before attempting to send it the message. Another example of this will be seen in Section 6.7.7. Since the `XyEventHandler` is going to look up views at the location of an event, it obviously cannot deal with events without locations, so returns `NO` (the Objective C term for `FALSE` or `0`) in this case.

The view database is then consulted, returning all the views whose bounding box contains the given point. The views returned are sorted from foremost to most background, *i.e.* according to their depth in the view tree, deepest first. In this order, each view is queried as to whether it wishes to handle the event, stopping when a view says `YES`. (The enigmatic test `v != tool` will be explained in section 6.7.7; suffice it to say here that in the typical case, `tool` is a kind of `GenericMouseTool` and thus can never be equal to a `View`.)

If no view is found that wishes to handle the event, the `XyEventHandler` returns `NO`. Since this handler is the last active event handler to be tried, when it says `NO`, the event is ignored. If desired, it is a simple matter to activate a catchall handler (to be tried after the `XyEventHandler`), the purpose of which is to handle all events, printing a message to the effect that events are being ignored.

[5]As shown in section 6.7.5, passive event handlers are asked to handle events via the `event:view:` message, one parameter being the event (from which the handler gets the tool), and the other is the view.

Another example event handler is given in Section 6.7.6; more will be said about active event handlers then.

6.7.4 The View Database

The function of the view database is to determine the set of views at a given location in a window. In many object-oriented UI toolkits, this function has been combined with event propagation, in that events propagate down the view tree [105] (or a corresponding controller tree [70, 63]) directly. The idea for a separate view database comes from GWUIMS[118]. By separating out the view database into its own data structure, efficient algorithms for looking up views at a given point, such as Bentley's dual range trees [7], may be applied. Unfortunately, this optimization was never completed, and in retrospect having to keep the view database synchronized with the view hierarchy was more effort than it was worth.

```
= Xydb :  Set { }
– enter:object at:rectangle {
    return [self replace:  [Xydbe object:object
                                    at:rectangle
                    depth:[object depth]]];
}

depthcmp(o1, o2) id *o1, *o2;
    { return [*o2 depth] – [*o1 depth]; }

– at:aPoint {
        id seq, e, array[MAXAT], result = [OrdCltn new]; int n;
        for(n = 0, seq = [self eachElement];
        (e = [seq next]) != nil; )
                if([e contains:aPoint]) array[n++] = e;
        qsort(array, n, sizeof(id), depthcmp);
        for(i = 0; i < n; i++) [result add:[array[i] object]];
        return result;
}

= Xydbe :  Rectangle { id object; unsigned depth; }
+ object:o at:rect depth:(unsigned)d {
        self = [self new] object = o; depth = d;
        return [[self origin:[rect origin]] corner:[rect corner]];
}
– object { return object; }
– (unsigned)depth { return depth; }
– (unsigned) hash { return [object hash]; }
– (BOOL)isEqual:o { object == o–>object; }
```

An Xydb is a set of Xydbe objects ("e" for "element"), each of which is a rectangle, an associated object (always a kind of View in GRANDMA), and a depth. View objects which move or grow must be sure to register their new locations in the view database for the wall on which they lie. This is currently done automatically in the _sync method of class View which is responsible for updating the display when a View changes. The hash and isEqual: methods are used by Set; here they define two Xydbe objects to be equal when their respective object fields are equal.

6.7.5 The Passive Event Handler Search Continues

Each View object has a list of passive handlers associated with it. The association is often implicit: passive handlers can be associated with the view directly, or with the class of the view, or any of the superclasses of the view's class. For example, the GenericToolOnViewEventHandler is directly associated with class View; it thus appears on every view's list of passive event handlers.

```
= View ...
- (BOOL)event:e { id seq, h;
        if(!  [self isOver:[e loc]]) return NO;
        for(seq = [self eachHandler]; h = [seq next];)
                if([h event:e view:self]) return YES;
    return NO;
}
- eachHandler { id r = [OrdCltn new]; id class;
    [r addContentsOf:[self passivehandlers]];
    for(class = [self class];
        class != Object; class = [class superClass])
        [r addContentsOf:[class passivehandlers]];
    return [r eachElement];
}
+ passivehandlers
    { return [UIprop getvalue:self propstr:"handlers"]; }
- passivehandlers
    { return [UIprop getvalue:self propstr:"handlers"]; }
```

When a view is asked if it wishes to handle an event, it firsts asks if the event's location is indeed over the view. The implementation of the isOver: method in class View simply returns YES. Non-rectangular subclasses of view (*e.g.* LineDrawingView, see Section 8.1) override this method.

Assuming the event location is over the view, each passive event handler associated with the view is sent the event:view: message, which asks if the passive handler wishes to handle the event. The search stops as soon as one of the handlers says YES or all the handlers have been tried.

The method eachHandler returns an ordered sequence of handlers associated with a view. The sequence is the concatenation of the handlers directly associated with the view object, those directly associated with the view's class, those associated with the view's superclass, and so on, up to and including those associated with class View. The associations themselves are stored in a global property list. The passive event handler is associated with a view object or class under the `"handlers"` property.

Herein lies another advantage of Objective C. An object's superclasses may be traversed at runtime, in this case enabling the simulation of inheritance of passive event handlers. This effect would be difficult to achieve had it not been possible to access the class hierarchy at runtime.

6.7.6 Passive Event Handlers

A passive event handler returns `YES` to the `event:view:` message if it wishes to handle the event directed at the given view. As a side effect, the passive event handler may activate (a copy or instance of) itself to handle additional input without incurring the cost of the search for a passive handler again.

In Objective C, classes are themselves first class objects in the system, known as factory objects. A factory object that is a subclass of `EventHandler` may play the role of a passive event handler.[6] To activate such a handler, the factory would instantiate itself and place the new instance on the active event list.

```
= EventHandler ...
+ (BOOL)event:e view:v
    { return (BOOL)[self subclassResponsibility]; }
```

As an example, consider the following handler for the toggle switch discussed earlier:

```
= ToggleSwitchEventHandler :  EventHandler { id view, tool; }
+ (BOOL) event:e view:v {
    if( !  [e isKindOf:PickEvent] ) return NO;
    if( !  [[e tool] isKindOf:MouseTool] ) return NO;
    self = [self new]; view = v; tool = [e tool];
    [[view wallview] activate:self];
    [view highlight];
    return YES;
}
- (BOOL)event:e {
    BOOL isOver;
    if( ![e isKindOf:DragEvent] || [e tool] != tool )
        return NO;
    isOver = [view pointInIboxAndOver:[e loc]];
    if(!isOver || [e isKindOf:DropEvent]) {
        [view unhighlight];
        [[view wallview] deactivate:self];
        if(!isOver) [[view model] toggle];
    }
```

[6] However, using factory objects for passive event handlers is restrictive, as there is only one instance of the factory object for a given class. This makes customization of a factory passive event handler difficult. Section 6.7.8 explains how regular (non-factory) objects may be used as passive event handlers.

```
        return YES;
    }
```

Assuming this event handler is associated with a `SwitchView`, when the mouse is pressed over such a view the handler's `event:view:` method is called, which instantiates and then activates this handler, and then highlights the view. Other events, such as typing a character or moving a mouse (with the button already pressed) over the view, will be ignored by this passive handler. Most handlers for mouse events, including this one, only respond to tools of kind `MouseTool`, where `MouseTool` is a subclass of `GenericMouseTool`. The reason for this is explained in Section 6.7.7.

Once the handler is activated, it gets first priority at all incoming events. The beginning of the `event:` method insures that it only responds to mouse events generated by the same mouse tool that initially caused the handler to be activated. For valid events, the handler checks if the location of the event (*i.e.* the mouse cursor) is over the view using `View`'s `pointInIboxAndOver:` method. Note that during passive event dispatch, the more efficient `isOver:` method was used, since by that point, the event location was already known to be in the bounding box of the view. The `pointInIboxAndOver` does both the bounding box check and the `isOver:` method, since active event handlers see events before it is determined which views they are over.

If the mouse is no longer over the switch, or the mouse button has been released, the highlighting of the view is turned off, and the handler deactivated. In the case where the mouse is over the view when the button was released, `[[view model] toggle]` is executed. The clause `[view model]` returns the model associated with the switch, presumably of class `Boolean`, which gets sent the `toggle` message. This will of course result in the switch's picture getting changed to reflect the model's new state.

In any case, by returning `YES` the active event handler indicates it has handled the event, so there will be no attempt to propagate it further.

Typically, the `ToggleSwitchEventHandler` would get associated with the `SwitchView` as follows:

```
    = SwitchView ...
    + initialize
        { return [self sethandler:ToggleSwitchEventHandler]; }
```

The `initialize` factory method is invoked for every class in the program (which has such a method) by the Objective C runtime system when the program is first started. In this case, the `sethandler` factory method would create a list (`OrdCltn`) containing the single element `ToggleSwitchEventHandler` and associate it with the class `SwitchView` under the `"handlers"` property.

Note that some simple changes to the `ToggleSwitchEventHandler` could radically alter the behavior of the switch. For example, if `[[view model] toggle]` is also executed when the switch is first pressed (*i.e.* in the `event:view:` method), the switch becomes a momentary pushbutton rather than a toggle switch. Similarly, by changing the initial check to `[e isKindOf:DragEvent]`, once the mouse moves off the switch (thus deactivating the handler), moving the mouse back on the switch with the button still pressed (or onto another instance of the switch) would (re)activate the handler. If the handler is changed only to deactivate when a

`DropEvent` is raised, the button now grabs the mouse, meaning no other objects would receive mouse events as long as the button is pressed. It is clear that many different behaviors are possible simply by changing the event handler.

While GRANDMA easily allows much flexibility in programming the behavior of individual widgets, interaction techniques that control multiple widgets in tandem are more difficult to program. For example, radio buttons (in which clicking one of a set of buttons causes it to be turned on and the rest of the set to be turned off) might be implemented by having the individual buttons to be subviews of a new parent view, and a new handler for the parent view could take care of the mutual exclusion. (Alternatively, the parent view could handle the mutual exclusion by providing a method for the individual buttons to call when pressed; in this case the parent necessarily provides the radio button interface to the rest of the program.)

6.7.7 Semantic Feedback

Semantic feedback is a response to a user's input which requires specialized information about the application objects [96]. For example, in the Macintosh Finder [2], dragging a file icon over a folder icon causes the folder icon to highlight, since dropping the file icon in the folder icon will cause the file to be moved to the folder. Dragging a file icon over another file icon causes no such highlighting, since dropping a file on another file has no effect. The highlighting is thus semantic feedback.

GRANDMA has a general mechanism for implementing (views of) objects which react when (views of) other objects are dropped on them, highlighting themselves whenever such objects are dragged over them. Such views are called *buckets* in GRANDMA. Any view may be made into a bucket simply by associating it with a passive `BucketEventHandler` (which expects the view to respond to the `actsUpon:` and `actUpon::` messages discussed below). Once a view has a `BucketEventHandler`, the semantic feedback described above will happen automatically.

Whereas a bucket is a view which causes an action when another view is dropped in it (*e.g.* the Macintosh trash can is a bucket), a `Tool` is an object which causes an action when it is dropped on a view (a "delete cursor" is thus a tool). As mentioned above, a tool corresponds to a physical input device (*e.g.* `GenericMouseTool`), but it is also possible for a view to be a tool. In the latter case, the view is referred to as a *virtual tool*.

Buckets and tools are quite similar, the main difference being that in buckets the action is associated with stationary views, while in tools the action is associated with the view being dragged. The implementation of tools is considered next. The similar implementation of buckets will not be described.

```
= Tool : Object { }
– (SEL)action { return (SEL) 0; }
– actionParameter { return nil; }
– (BOOL)actsUpon:v { return [v respondsTo:[self action]]; }
– actUpon:v event:e {
    [v perform:[self action]
          with:[self actionParameter]
          with:e
          with:self];
```

```
        return self;
    }
```

Every tool responds to the `actsUpon:` and `actUpon::` messages. In the default implementation above, a tool has an action (which is the runtime encoding of a message selector) and an action parameter (an arbitrary object). For example, one way to create a tool for deleting objects is

```
    = DeleteTool : Tool { }
    - (SEL)action { return @selector(delete); }
```

The `actsUpon:` method checks to see if the view passed as a parameter responds to the action of the tool, in this case `delete`. The `actUpon::` method actually performs the action, passing the action parameter, the event, and the tool itself as additional parameters (which are ignored in the `delete` case).

The `GenericToolOnViewEventHandler` is associated with every view via the `View` class:

```
    = View ...
    + initialize
        { [self sethandler:GenericToolOnViewEventHandler]; }

    = GenericToolOnViewEventHandler : EventHandler
        { id tool, view; }
    + (BOOL) event:e view:v {
        if( ! [e isKindOf:DragEvent]) return NO;
        if( ! [[e tool] actsUpon:v]) return NO;
        self = [self new];
        tool = [e tool]; view = v; [view highlight];
        [[view wallview] activate:self];
        return YES;
    }
    - (BOOL)event:e {
        if( ! [e isKindOf:DragEvent]) return NO;
        if( [e tool] != tool) return NO;
        if( [view pointInIboxAndOver:[e loc]] ) {
            if([e isKindOf:DropEvent]) {
                [view unhighlight];
                [[view wallview] deactivate:self];
                [tool actUpon:view event:e];
            }
            return YES;
        }
        [view unhighlight]; [[view wallview] deactivate:self];
        return NO;
    }
```

Passively, `GenericToolOnViewEventHandler` operates by simply checking if the tool over the view acts upon the view. If so, the view is highlighted (the semantic feedback) and the handler activates an instantiation of itself. Subsequent events will be checked by the activated handler to see if they are made by the same tool. If so, and if the tool is still over the view, the event is handled, and if it is a `DropEvent` then the tool will act upon the view. If the tool has moved off the view, the highlighting is turned off, and the handler deactivates itself and returns `NO` so that other handlers may handle this event.

The test `v != tool` in the `XYEventHandler` (see Section 6.7.3) prevents a view that is a virtual tool from ever attempting to operate upon itself.

6.7.8 Generic Event Handlers

If you have been following the story so far, you know that all the event handlers shown have the passive handler implemented by a factory (class) object which responds to `event:view:` messages. When necessary, such a passive handler activates an instantiation of itself. The drawback of having factory objects as passive event handlers is that they cannot be changed at runtime. For example, the `ToggleSwitchEventHandler` only passively responds to `PickEvents`. If one wanted to make a `ToggleSwitchEventHandler` that passively responded to any `DragEvent`, one could either change the implementation of `ToggleSwitchEventHandler` (thus affecting the behavior of every toggle switch view), or one could subclass `ToggleSwitchEventHandler`. Doing the latter, it would be necessary to duplicate much of the `event:view:` method, or change `ToggleSwitchEventHandler` by putting the `event:view:` method in another method, so that it can be used by subclasses. In any case, changing a simple item (the kind of event a handler passively responds to) is more difficult than it need be.

In order to make event handlers more parameterizable, the passive event handlers should be regular objects (*i.e.* not factory objects). In response to this problem, most event handlers are subclasses of `GenericEventHandler`.

```
= GenericEventHandler : EventHandler {
    BOOL shouldActivate;
    id startp, handlep, stopp;
    id view, wall, tool, env;
}
+ passive { return [self new]; }

– shouldActivate { shouldActivate = YES; return self; }

– startp:_startp { startp = _startp; return self; }
– startp { return startp; }
– (BOOL)evalstart:env { return [[startp eval:env] asBOOL]; }

– stopp:_stopp { stopp = _stopp; return self; }
– stopp { return stopp; }
– (BOOL)evalstop:env { return [[stopp eval:env] asBOOL]; }
```

```
− handlep:_handlep { handlep = _handlep; return self; }
− handlep { return handlep; }
− (BOOL)evalhandle:env
    { return [[handlep eval:env] asBOOL]; }

− (BOOL) event:e view:v {
    env = [[[Env new] str:"event" value:e]
              str:"view" value:v];
    if([self evalstart:env])
        { [self startOnView:v]; return YES; }
    return NO;
}
− startOnView:v event:e {
    if(shouldActivate)
        self = [self copy], [[view wallview] activate:self];
    view = v; wall = [view wallview]; tool = [e tool];
    [self passiveHandler:e];
    return self;
}
− (BOOL)event:e {
    if(tool != nil  &&  [e tool] != tool) return NO;
    env = [[[Env new] str:"event" value:e]
              str:"view" value:view];
    if([self evalstop:env])
        [self activeTerminator:e], [wall deactivate:self];
    else if([self evalhandle:env])
        [self activeHandler:e];
    else return NO;
    return YES;
}
− passiveHandler:e { return self; }
− activeHandler:e { return self; }
− activeTerminator:e { return self; }
```

A new passive handler is created by sending a kind of `GenericEventHandler` the `passive` message. A generic event handler object has settable predicates `startp`, `handlep`, and `stopp`. These predicates are expression objects, essentially runtime representations of almost arbitrary Objective C expressions. (The Objective C interpreter built into GRANDMA is discussed in section 7.7.3.) By convention, these predicates are evaluated in an environment where `event` is bound to the event under consideration and `view` is bound to a view at the location of the event. Of course, the result of evaluating a predicate is a boolean value.

The `passive` method is typically overridden by subclasses of `GenericEventHandler` in order to provide default values for `startp`, `handlep`, and `stopp`. The predicate `startp` controls what events the passive handler reacts to. The class `EventExpr` allows easy specification of simple predicates, *e.g.* the call

```
        [self startp:[[[EventExpr new] eventkind:PickEvent]
                                        toolkind:MouseTool]];
```

sets the start predicate to check that the event is a kind of `PickEvent` and that the tool is a `MouseTool`. This results in the same passive event check that was hard-coded into the factory `ToggleSwitchEventHandler`, but now such a check may be easily modified at runtime.

The message `shouldActivate` tells the passive event handler to activate itself whenever its `startp` predicate is satisfied. Note that it is a clone of the handler that is activated, due to the statement `self = [self copy];` it is thus possible for a single passive event handler to activate multiple instances of itself simultaneously. The active handler responds to any message which satisfies its `handlep` or `donep` predicates. In the latter case, the active event handler is deactivated.

When the `startp`, `handlep`, or `donep` predicates are satisfied, the generic event handler sends itself the `passiveHandler:`, `activeHandler:` or `activeTerminator:` message, respectively. The main work of subclasses of `GenericEventHandler` are done in these methods.

The `startOnView:event:` allows a passive handler to be activated externally (*i.e.* instead of the typical way of having its `startp` satisfied in the `event:view:` method). In this case, the `event` parameter is usually `nil`. For example, an application that wishes to force the user to type some text into a dialogue box before proceeding might activate a text handler in this manner.

The purpose of generic event handlers in GRANDMA is similar to that of *interactors* in Garnet [95, 91] and *pluggable views* in Smalltalk-80 [70]. Since GRANDMA comes with a number of generally useful generic event handlers, application programmers often need not write their own. Instead, they may customize one of the generic handlers by setting up the parameters to suit their purposes. The only parameters every generic event handler has in common are the predicates, and indeed these are the ones most often modified. GRANDMA has a subsystem which allows these parameters to be modified at runtime by the user.[7]

6.7.9 The Drag Handler

As an example of a generic event handler, consider the `DragHandler`. When associated with a view, the `DragHandler` allows the view to be moved (dragged) with the mouse. If desired, moving the view will result in new events being raised. This allows the view to be used as tool, as discussed in section 6.7.7. Also parameterizable are whether the view is moved using absolute or relative coordinates, whether the view is copied and then the copy is moved, and the messages that are sent to actually move the view. Reasonable defaults are supplied for all parameters.

```
    = DragHandler : GenericEventHandler {
        BOOL    copyview, genevents, relative;
        SEL     whenmoved, whendone;
```

[7]Typically, it would be the interface designer, rather than the end user, who would use this facility.

```
    BOOL    deactivate;
    int     savedx, savedy;
}

+ passive {
    self = [super passive];
    [self shouldActivate];
    [self startp:[[[EventExpr new] eventkind:DragEvent]
                                   toolkind:MouseTool]];
    [self handlep:[[EventExpr new] eventkind:DragEvent]];
     [self stopp:[[EventExpr new] eventkind:DropEvent]];
    copyview = NO; genevents = YES; relative = NO;
    whenmoved = @selector(at::);  whendone = (SEL) 0;
    return self;
}
```

/∗ changing default parameters: ∗/

/∗ copyviewON causes the view to be copied and then the copy to be dragged ∗/

```
- copyviewON { copyview = YES; return self; }
```

/∗ genEventsOFF makes the handler not raise any events ∗/

```
- genEventsOFF { genevents = NO; return self; }
```

/∗ relativeON makes the handler send the move:: message, passing relative coordinates (deltas from the current position) ∗/

```
- relativeON { relative = YES;
           whenmoved = @selector(move::); }
```

/∗ whendone: sets the message sent on the event that terminates the drag ∗/

```
- whendone:(SEL)sel { whendone = sel; return self; }
```

/∗ whenmoved: sets the message sent for every point in the drag ∗/

```
- whenmoved:(SEL)sel { whenmoved = sel; return self; }

- passiveHandler:e {
    id l = [e loc];
    if(relative) savedx = [l x], savedy = [l y];
    else savedx = [view xloc]-[l x],
         savedy = [view yloc]-[l y];
    if(copyview) view = [view viewcopy];
    [view flash];
```