The Automatic Recognition of Gestures

Dean Harris Rubine

December, 1991

CMU–CS–91–202

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science at Carnegie Mellon University.

Thesis Committee:

Roger B. Dannenberg, Advisor
Dario Giuse
Brad Myers
William A. S. Buxton, University of Toronto

Abstract

Gesture-based interfaces, in which the user specifies commands by simple freehand drawings, offer an alternative to traditional keyboard, menu, and direct manipulation interfaces. The ability to specify objects, an operation, and additional parameters with a single intuitive gesture makes gesture-based systems appealing to both novice and experienced users.

Unfortunately, the difficulty in building gesture-based systems has prevented such systems from being adequately explored. This dissertation presents work that attempts to alleviate two of the major difficulties: the construction of gesture classifiers and the integration of gestures into direct-manipulation interfaces. Three example gesture-based applications were built to demonstrate this work.

Gesture-based systems require classifiers to distinguish between the possible gestures a user may enter. In the past, classifiers have often been hand-coded for each new application, making them difficult to build, change, and maintain. This dissertation applies elementary statistical pattern recognition techniques to produce gesture classifiers that are trained by example, greatly simplifying their creation and maintenance. Both single-path gestures (drawn with a mouse or stylus) and multiple-path gestures (consisting of the simultaneous paths of multiple fingers) may be classified. On a 1 MIPS workstation, a 30-class single-path recognizer takes 175 milliseconds to train (once the examples have been entered), and classification takes 9 milliseconds, typically achieving 97% accuracy. A method for classifying a gesture as soon as it is unambiguous is also presented.

This dissertation also describes GRANDMA, a toolkit for building gesture-based applications based on Smalltalk's Model/View/Controller paradigm. Using GRANDMA, one associates sets of gesture classes with individual views or entire view classes. A gesture class can be specified at runtime by entering a few examples of the class, typically 15. The semantics of a gesture class can be specified at runtime via a simple programming interface. Besides allowing for easy experimentation with gesture-based interfaces, GRANDMA sports a novel input architecture, capable of supporting multiple input devices and multi-threaded dialogues. The notion of virtual tools and semantic feedback are shown to arise naturally from GRANDMA's approach.

Acknowledgments

First and foremost, I wish to express my enormous gratitude to my advisor, Roger Dannenberg. Roger was always there when I needed him, never failing to come up with a fresh idea. In retrospect, I should have availed myself more than I did. In his own work, Roger always addresses fundamental problems, and his solutions are always simple and elegant. I try to follow Roger's example in my own work, usually falling far short. Roger, thank you for your insight and your example. Sorry for taking so long.

I was incredibly lucky that Brad Myers showed up at CMU while I was working on this research. His seminar on user interface software gave me the knowledge and breadth I needed to approach the problem of software architectures for gesture-based systems. Furthermore, his extensive comments on drafts of this document improved it immensely. Much of the merit in this work is due to him. Thank you, Brad. I am also grateful to Bill Buxton and Dario Giuse, both of whom provided valuable criticism and excellent suggestions during the course of this work.

It was Paul McAvinney's influence that led me to my thesis topic; had I never met him, mine would have been a dissertation on compiler technology. Paul is an inexhaustible source of ideas, and this thesis is really the *second* idea of Paul's that I've spent multiple years pursuing. Exploring Paul's ideas could easily be the life's work of hundreds of researchers. Thanks, Paul, you madman you.

My wife Ruth Sample deserves much of the credit for the existence of this dissertation. She supported me immeasurably, fed me and clothed me, made me laugh, motivated me to finish, and lovingly tolerated me the whole time. Honey, I love you. Thanks for everything.

I could not have done it with the love and support of my parents, Shirley and Stanley, my brother Scott, my uncle Donald, and my grandma Bertha. For years they encouraged me to be a doctor, and they were not the least bit dismayed when they found out the kind of doctor I wanted to be. They hardly even balked when "just another year" turned out to be six. Thanks, folks, you're the best. I love you all very much.

My friends Dale Amon, Josh Bloch, Blaine Burks, Paul Crumley, Ken Goldberg, Klaus Gross, Gary Keim, Charlie Krueger, Kenny Nail, Eric Nyberg, Barak Pearlmutter, Todd Rockoff, Tom Neuendorffer, Marie-Helene Serra, Ellen Siegal, Kathy Swedlow, Paul Vranesevic, Peter Velikonja, and Brad White all helped me in innumerable ways, from technical assistance to making life worth living. Peter and Klaus deserve special thanks for all the time and aid they've given me over the years. Also, Mark Maimone and John Howard provided valuable criticism which helped me prepare for my oral examination. I am grateful to you all.

I wish to also thank my dog Dismal, who was present at my feet during much of the design, implementation, and writing efforts, and who concurs on all opinions. Dismal, however, strongly objects to this dissertation's focus on *human* gesture.

I also wish to acknowledge the excellent environment that CMU Computer Science provides; none of this work would have been possible without their support. In particular, I'd like to thank Nico Habermann and the faculty for supporting my work for so long, and my dear friends Sharon Burks, Sylvia Berry, Edith Colmer, and Cathy Copetas.

Contents

List of Figures

List of Tables

to Grandma Bertha

Chapter 1

Introduction

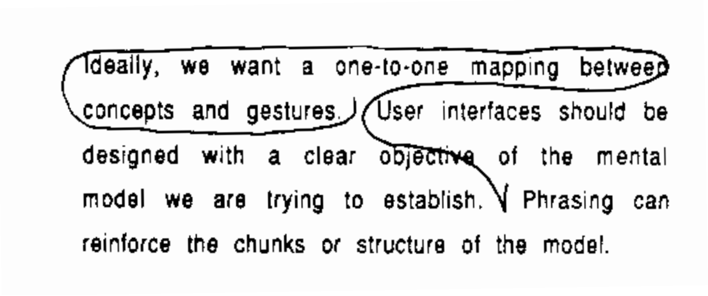
People naturally use hand motions to communicate with other people. This dissertation explores the use of human gestures to communicate with computers.

Random House [122] defines "gesture" as "the movement of the body, head, arms, hands, or face that is expressive of an idea, opinion, emotion, etc." This is a rather general definition, which characterizes well what is generally thought of as gesture. It might eventually be possible through computer vision for machines to interpret gestures, as defined above, in real time. Currently such an approach is well beyond the state of the art in computer science.

Because of this, the term "gesture" usually has a restricted connotation when used in the context of human-computer interaction. There, gesture refers to hand markings, entered with a stylus or mouse, which function to indicate scope and commands [109]. Buxton [14] gives a fine example, reproduced here as figure 1.1. In this dissertation, such gestures are referred to as *single-path* gestures.

Recently, input devices able to track the paths of multiple fingers have come into use. The Sensor Frame [84] and the DataGlove [32, 130] are two examples. The human-computer interaction community has naturally extended the use of the term "gesture" to refer to hand motions used to indicate commands and scope, entered via such multiple finger input devices. These are referred to here as *multi-path* gestures.

Rather than defining gesture more precisely at this point, the following section describes an



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

Figure 1.1: Proofreader's Gesture (from Buxton [15])

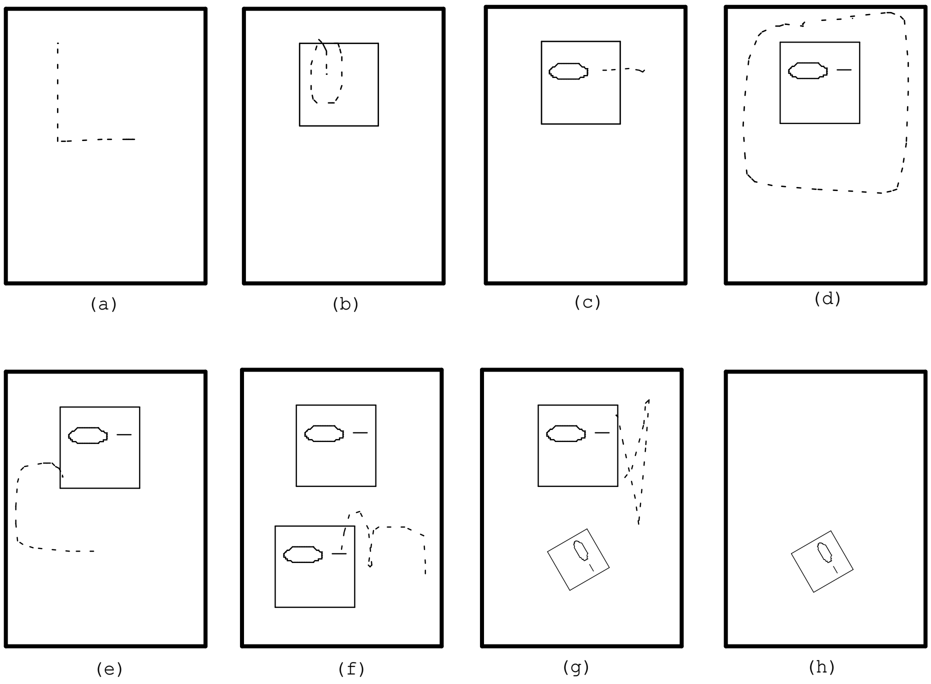


Figure 1.2: GDP, a gesture-based drawing program

example application with a gestural interface. A more technical definition of gesture will be presented in section 1.6.

1.1 An Example Gesture-based Application

GRANDMA is a toolkit used to create gesture-based systems. It was built by the author and is described in detail in the pages that follow. GRANDMA was used to create GDP, a gesture-based drawing editor loosely based on DP [42]. GDP provides for the creation and manipulation of lines, rectangles, ellipses, and text. In this section, GDP is used as an example gesture-based system. GDP's operation is presented first, followed by a description of how GRANDMA was used to create GDP's gestural interface.

1.1.1 GDP from the user's perspective

GDP's operation from a user's point of view will now be described. (GDP's design and implementation is presented in detail in Section 8.1.) The intent is to give the reader a concrete example of a gesture-based system before embarking on a general discussion of such systems. Furthermore, the description of GDP serves to illustrates many of GRANDMA's capabilities. A new interaction technique, which combines gesture and direct manipulation in a single interaction, is also introduced in the description.

Figure 1.2 shows some snapshots of GDP in action. When first started, GDP presents the user with a blank window. Panel (a) shows the rectangle gesture being entered. This gesture is drawn like an "L."[1] The user begins the gesture by positioning the mouse cursor and pressing a mouse button. The user then draws the gesture by moving the mouse.

The gesture is shown on the screen as is being entered. This technique is called *inking* [109], and provides valuable feedback to the user. In the figure, inking is shown with dotted lines so that the gesture may be distinguished from the objects in the drawing. In GDP, the inking is done with solid lines, and disappears as soon as the gesture has been recognized.

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

Panel (b) of figure 1.2 shows the rectangle that has been created and the ellipse gesture. This gesture creates an ellipse with its center at the start of the gesture. A point on the ellipse tracks the mouse after the gesture has been recognized; this gives the user interactive control over the size and eccentricity of the ellipse.

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

Panel (d) shows all three shapes being encircled by a pack gesture. This gesture packs (groups) all the objects which it encloses into a single composite object, which can then be manipulated as a unit. Panel (e) shows a copy gesture being made; the composite object is copied and the copy is dragged by the mouse.

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

Panel (g) shows the delete gesture, essentially an "X" drawn with a single stroke. The object at the gesture start is deleted, as shown in panel (h).

This brief description of GDP illustrates a number of features of gesture-based systems. Perhaps the most striking feature is that each gesture corresponds to a high-level operation. The class of the gesture determines the operation; attributes of the gesture determine its scope (the operands) and any additional parameters. For example, the delete gesture specifies the object to be deleted, the pack gesture specifies the objects to be combined, and the line gesture specifies the endpoints of the line.

---

[1] It is often convenient to describe single-path gestures as if they were handwritten letters. This is not meant to imply that gesture-based systems can only recognize alphabetic symbols, or even that they usually recognize alphabetic symbols. The many ways in which gesture-based systems are distinct from handwriting-recognition systems will be enumerated in section 1.8.

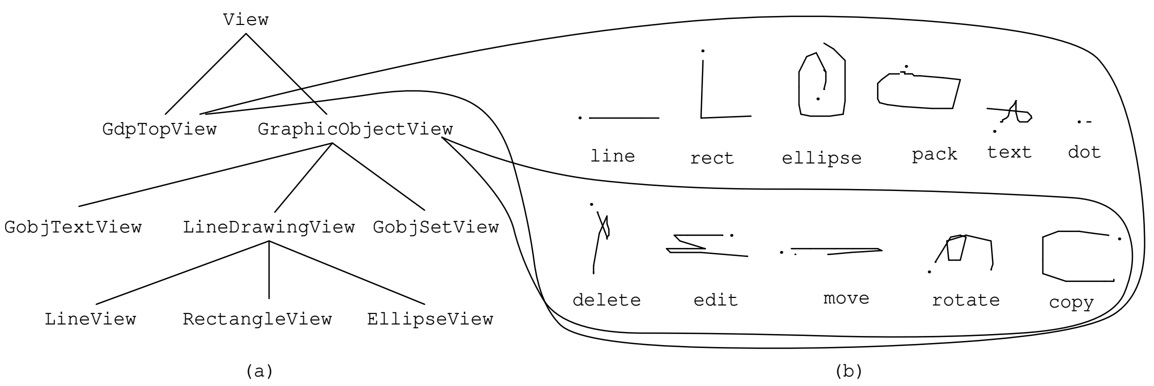


Figure 1.3: GDP's `View` class hierarchy and associated gestures

A period indicates the first point of each gesture.

It is possible to control more than positional parameters with gestural attributes. For example, one version of GDP uses the length (in pixels) of the line gesture to control the thickness of the new line.

Note how gesturing and direct manipulation are combined in a new two-phase interaction technique. The first phase, the collection of the gesture, ends when the user stops moving the mouse while holding the button. At that time, the gesture is recognized and a number of parameters to the application command are determined. After recognition, a manipulation phase is entered during which the user can control additional parameters interactively.

In addition to its gestural interface, GDP provides a more traditional click-and-drag interface. This is mainly used to compare the two styles of interface, and is further discussed in Section 8.1. The gestural interface is grafted on top of the click-and-drag interface, as will be explained next.

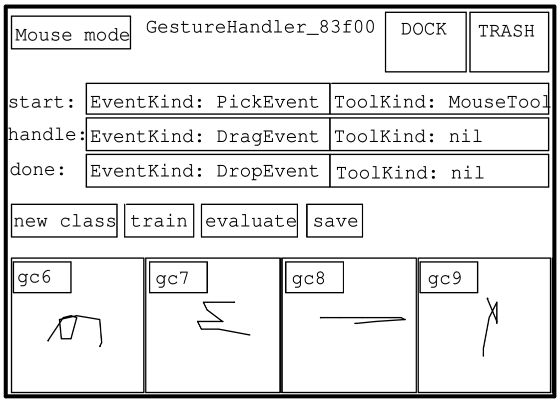
1.1.2 Using GRANDMA to Design GDP's Gestures

In the current work, the *gesture designer* creates a gestural interface to an application out of an existing click-and-drag interface to the application. Both the click-and-drag interface and the application are built using the object-oriented toolkit GRANDMA. The gesture designer only modifies the way input is handled, leaving the output mechanisms untouched.

A system built using GRANDMA utilizes the object-oriented programming paradigm to represent windows and the graphics objects displayed in windows. For example, figure 1.3a shows GDP's `View` class hierarchy.[2] This hierarchy shows the relationship of the classes concerned with output. The task of the gesture designer is to determine which of these classes are to have associated gestures, and for each such view class, to design a set of gestures that intuitively expresses the allowable operations on the view. Figure 1.2b shows the sets of gestures associated with GDP's `GraphicObjectView` and `GdpTopView` classes. The `GraphicObjectView` collectively

---

[2] For expositional purposes, the hierarchy shown is a simplified version of the actual hierarchy. Some of the details that follow have also been simplified. Section 8.1 tells the truth in gory detail.



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

Figure 1.4: Manipulating gesture handlers at runtime

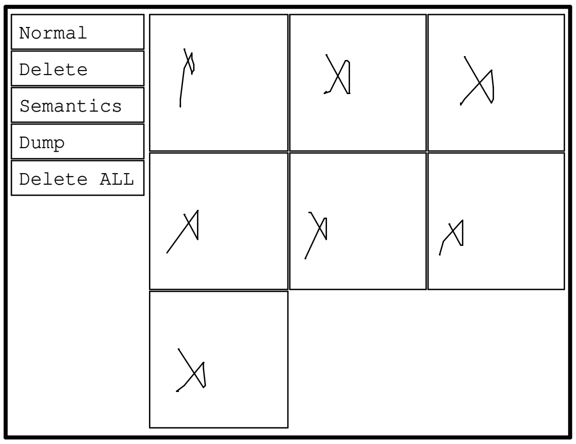


Figure 1.5: Adding examples of the delete gesture

refers to the line, rectangle, and ellipse shapes, while `GdpTopView` represents the window in which GDP runs.

GRANDMA is a Model/View/Controller-like system [70]. In GRANDMA, an input event handler (a "controller" in MVC terms) may be associated with a view class, and thus shared between all instances of the class (including instances of subclasses). This adds flexibility while eliminating a major overhead of Smalltalk MVC, where one or more controller objects are associated with each view object that expects input.

The gesture designer adds gestures to GDP's initial click-and-drag interface at runtime. The first step is to create a new gesture handler and associate it the `GraphicObjectView` class, easily done using GRANDMA. Figure 1.4 shows the gesture handler window after a number of gestures have been created (using the "new class" button), and figure 1.5 shows the window in which examples of the delete gesture have been entered. Fifteen examples of each gesture class typically suffice. If a gesture is to vary in size and/or orientation, the examples should reflect that.

Clicking on the "Semantics" button brings up a window that the designer uses to specify the semantics of each gesture in the handler's set. The window is a structured editing and browsing interface to a simple Objective-C [28] interpreter, and the designer enters three expressions: `recog`, evaluated when the gesture is first recognized; `manip`, evaluated on subsequent mouse points; and `done`, evaluated when the mouse button is released. In this case, the delete semantics simply change the mouse cursor to a delete cursor, providing feedback to the user, and then delete the view at which the gesture was aimed. The expressions entered are[3]

---

[3] Objective C syntax is used throughout. `[view delete]` sends the `delete` message to the object referred to by the variable `view`. `[handler mousetool:DeleteCursor]` sends the `mousetool:` message to the object referred to by the variable `handler` passing the value of the variable `DeleteCursor` as an argument. See Section 6.3 for more information on Objective C notation.

```
recog = [_Seq :[handler mousetool:DeleteCursor]
              :[view delete]];
manip = nil;
done = nil;
```

The designer may now immediately try out the delete gesture, as in figure 1.2g.

The designer repeats the process to create a gesture handler for the set of gestures associated with class `GdpTopView`, the view that refers to the window in which GDP runs. This handler deals with the gestures that create graphic objects, the pack gesture (which creates a set out of the enclosed graphic objects), the dot gesture (which repeats the last command), and the gestures also handled by `GraphicObjectView`'s gesture handler (which when made at a `GdpTopView` change the cursor without operating directly on a graphic object).

The attributes of the gesture are directly available for use in the gesture semantics. For example, the semantics of the line gesture are:

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

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

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

Many other gesture attributes are useful in semantics. The line semantics could be augmented to control the thickness of the line from the maximum speed or total path length of the gesture. The rectangle semantics could use the initial angle of the rectangle gesture to determine the orientation of the rectangle. The attribute `<enclosed>` is especially noteworthy: it contains a list of views enclosed by the gesture and is used, for example, by the pack gesture (figure 1.2d). When convenient, the semantics can simulate input to the click-and-drag interface, rather than communicating directly with application objects or their views, as shown above.

When the first point of a gesture is over more than one gesture-handling view, the union of the set of gestures recognized by each handler is used, with priority given to the foremost views. For example, any gesture made at a GDP `GraphicObjectView` is necessarily made over the `GdpTopView`. A delete gesture made at a graphic object would be handled by the `GraphicObjectView` while a line gesture at the same place would be handled by the `GdpTopView`. Set union also occurs when gestures are (conceptually) inherited via the view class hierarchy. For example, the gesture designer might create a new gesture handler for the `GobjSetView` class containing an unpack gesture. The set of gestures recognized by

`GobjSetViews` would then consist of the unpack gesture as well as the five gestures handled by `GraphicObjectView`.

1.2 Glossary

This section defines and clarifies some terms that will be used throughout the dissertation. It may safely be skipped and referred back to as needed. Some of the terms (click, drag) have their common usage in the human-computer interaction community, while others (pick, move, drop) are given technical definitions solely for use here.

class In this dissertation, "class" is used in two ways. "Gesture class" refers to a set of gestures all of which are intended to be treated the same, for example, the class of delete gestures. (In this dissertation, the names of gesture classes will be shown in sans serif typeface.) The job of a gesture recognizer is, given an example gesture, to determine its class (see also "gesture"). "Class" is also used in the object-oriented sense, referring to the type (loosely speaking) of a software object. It should be clear from context which of these meanings is intended.

click A click consists of positioning the mouse cursor and then pressing and releasing a mouse button, with no intervening mouse motion. In the Macintosh, a click is generally used to select an object on the screen.

click-and-drag A click-and-drag interface is a direct-manipulation interface in which objects on the screen are operated upon using mouse clicks, drags, and sometimes double-clicks.

direct manipulation A direct-manipulation interface is one in which the user manipulates a graphic representation of the underlying data by pointing at and/or moving them with an appropriate device, such as a mouse with buttons.

double-click A double-click is two clicks in rapid succession.

drag A drag consists of locating the mouse cursor and pressing the mouse button, moving the mouse cursor while holding the mouse button, and then releasing the mouse button. Drag interactions are used in click-and-drag interfaces to, for example, move objects around on the screen.

drop The final part of a drag (or click) interaction in which the mouse button is released.

eager recognition A kind of gesture recognition in which gestures are often recognized without the end of the gesture having to be explicitly signaled. Ideally, an eager recognizer will recognize a gesture as soon as enough of it has been seen to determine its class unambiguously.

gesture Essentially a freehand drawing used to indicate a command and all its parameters. Depending on context, the term maybe used to refer to an example gesture or a class of gestures, *e.g.* "a delete gesture" means an example gesture belonging to the class of delete gestures. Usually "gesture" refers to the part of the interaction up until the input is recognized as one of a number of possible gesture classes, but sometimes the entire interaction (which includes a manipulation phase after recognition) is referred to as a gesture.

move The component of drag interaction during which the mouse is moved while a mouse button is held down. It is the presence of a move that distinguishes a click from a drag.

multi-path A multi-path gesture is one made with an input device that allows more than one position to be indicated simultaneously (multiple pointers). One may make multi-path gestures with a Sensor Frame, a multiple-finger touch pad, or a DataGlove, to name a few such devices.

off-line Considering an algorithm to be a sequence of operations, an off-line algorithm is one which examines subsequent operations before producing output for the current operation.

on-line An on-line algorithm is one in which the output of an operation is produced before any subsequent operations are read.

pick The initial part of a drag (or click) interaction consisting of positioning the mouse cursor at the desired location and pressing a mouse button.

press refers to the pressing of a mouse button.

real-time A real-time algorithm is an on-line algorithm in which each operation is processed in time bounded by a constant.

release refers to the releasing of a mouse button.

segment A segment is an approximately linear portion of a stroke. For example, the letter "L" is two segments, one vertical and one horizontal.

single-path A single-path gesture is one drawn by an input device, such as a mouse or stylus, capable of specifying only a single point over time. A single-path gesture may consist of multiple strokes (like the character "X").

single-stroke A single-stroke gesture is a single-path gesture that is one stroke. Thus drawing "L" is a single-stroke gesture, while "X" is not. In this dissertation the only single-path gestures considered are single-stroke gestures.

stroke A stroke is an unbroken curve made by a single movement of a pen, stylus, mouse, or other instrument. Generally, strokes begin and end with explicit user actions (*e.g.* , pen down/pen up, mouse button down/mouse button up).

1.3 Summary of Contributions

This dissertation makes contributions in four areas: new interaction techniques, new algorithms for gesture recognition, a new way of integrating gestures into user interfaces, and a new architecture for input in object-oriented toolkits.

The first new interaction technique is the two-phase combination of single-stroke gesture collection followed by direct manipulation, mentioned previously. In the GDP example discussed above, the boundary between the two phases is an interval of motionlessness. Eager recognition, the second new interaction technique, eliminates this interval by recognizing the single-stroke gesture and entering the manipulation phase as soon as enough of the gesture has been seen to do so unambiguously, making the entire interaction very smooth. A third new interaction technique is the two-phase interaction applied to multi-path gestures: after a multi-path gesture has been recognized, individual paths (*i.e.* fingers, possibly including additional fingers not involved in making the recognized gesture) may be assigned to manipulate independent application parameters simultaneously.

The second contribution is a new trainable, single-stroke recognition algorithm tailored for recognizing gestures. The classification is based on meaningful features, which in addition to being useful for recognition are also suitable for passing to application routines. The particular set of features used has been shown to be suitable for many different gesture sets, and is easily extensible. When restricted to features that can be updated incrementally in constant time per input point, arbitrarily large gestures may be handled. The single-stroke recognition algorithm has been extended to do eager recognition (eager recognizers are automatically generated from example gestures), and also to multi-path gesture recognition.

Third, a new paradigm for creating gestural interfaces is also propounded. As seen in the example, starting from a click-and-drag implementation of an interface, gestures are associated with classes of views (display objects), with the set of gestures recognized at a particular screen location dynamically determined by the set of overlapping views at the location, and by inheritance up the class hierarchy of each such view. The classification and attributes of gestures map directly to application operations and parameters. The creation, deletion, and manipulation of gesture handlers, gesture classes, gesture examples, and gesture semantics all occur at runtime, enabling quick and easy experimentation with gestural interfaces.

Fourth, GRANDMA, as an object-oriented user interface toolkit, makes some contributions to the area of input handling. Event handler objects are associated with particular views or entire view classes. A single event handler may be shared between many different objects, eliminating a major overhead of MVC systems. Multiple event handlers may be associated with a single object, enabling the object to support multiple interaction techniques simultaneously, including the use of multiple input devices. Furthermore, a single mechanism handles both mouse tools (*e.g.* a delete cursor that deletes clicked-upon objects) and virtual tools (*e.g.* a delete icon that is dragged around and dropped upon objects to delete them). Additionally, GRANDMA provides support for semantic feedback, and enables the runtime creation and manipulation of event handlers.

1.4 Motivation for Gestures

At this point, the reader should have a good idea of the scope of the work to be presented in this dissertation. Stepping back, this section begins a general discussion of gestures by examining the motivation for using and studying gesture-based interfaces. Much of the discussion is based on that of Buxton [14].

Computers get faster, bitmapped displays produce ever increasing information rates, speech and

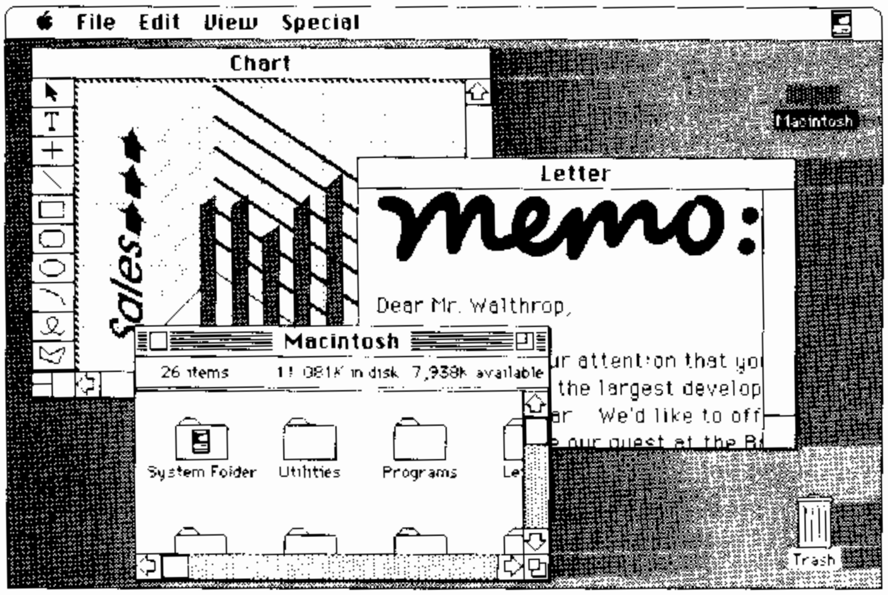


Figure 1.6: Macintosh Finder, MacDraw, and MacWrite (from Apple [2])

music can be generated in real-time, yet input just seems to plod along with little or no improvement. This is regrettable because, in Paul McAvinney's words [84], most of the useful information in the world resides in humans, not computers. Most people who interact with computers spend most of their time entering information [22]. Due to this input bottleneck, the total time to do many tasks would hardly improve even if computers became infinitely fast. Thus, improvements in input technology are a major factor in improving the productivity of computer users in general.

Of course, progress has been made. Input has progressed from batch data entry, to interactive line editors, to two-dimensional screen editors, to mouse-based systems with bitmapped displays. Pointing with a mouse has proved a useful interaction technique in many applications. "Click and drag" interfaces, where the user directly manipulates graphic objects on the screen with a mouse, are often very intuitive to use. Because of this, direct manipulation interfaces have become commonplace, despite being rather difficult to build.

Consider the Macintosh [2], generally regarded as having a good direct-manipulation interface. As shown in figure 1.6, the screen has on it a number of graphic objects, including file icons, folder icons, sliders, buttons, and pull-down menu names. Each one is generally a rectangular region, which may be clicked, sometimes double-clicked, and sometimes dragged. The Macintosh Finder, which may be used to access all Macintosh applications and documents, is almost entirely controlled via these three interaction techniques.[4]

The click and double-click interactions have a single object (or location) as parameter. The drag

---

[4] Obviously this discussion ignores keyboard entry of text and commands.

interaction has two parameters: an object or location where the mouse button is first pressed, and another object or location at the release point. Having only these three interaction techniques is one reason the Macintosh is simple to operate. There is, however, a cost: both the application and the user must express all operations in terms of these three interaction techniques.

An application that provides more than three operations on any given object (as many do) has several design alternatives. The first, exemplified by the Finder, relies heavily on *selection*. In the Finder, a click interaction selects an object, a double-click opens an object (the meaning of which depends upon the object's type), and a drag moves an object (the meaning of which is also object-type specific). Opening an object by a double-click is a means for invoking the most common operation on the object, *e.g.* opening a MacWrite document starts the MacWrite application on the document. Dragging is used for adjusting sliders (such as those which scroll windows), changing window size or position, moving files between folders, and selecting menu items.

All other operations are done in at least two steps: first the object to be operated upon is selected, and then the desired operation is chosen from a menu. For example, to print an object, one selects it (click) then chooses "Print" from the appropriate menu (drag); to move some text, one selects it (drag), chooses "Cut" (drag), selects an insertion point (click), and chooses "Paste" (drag). The cost of only having three interaction techniques is that some operations are necessarily performed via a sequence of interactions. The user must adjust her mental model so that she thinks in terms of the component operations.

An alternative to the selection-based click-and-drag approach is one based on modes. Consider MacDraw [2], a drawing program. The user is presented with a palette offering choices such as line, text, rectangles, circles, and so on. Clicking on the "line" icon puts the program into line-drawing mode. The next drag operation in the drawing window cause lines to be drawn. In MacDraw, after the drag operation the program reverts back to selection mode. DP, the program upon which GDP is based, is similar except that it remains in its current mode until it is explicitly changed. Mistakes occur when the user believes he is in one mode but is actually in another. The claim that direct manipulation interfaces derive their power from being modeless is not really true. Good direct manipulation interfaces simply make the modes very visible, which helps to alleviate the problems of modal interfaces.

By mandating the sole use of click, double-click, and drag interactions, the Macintosh interface paradigm necessarily causes conceptually primitive tasks to be divided into a sequence of primitive interactions. The intent of gestural interfaces is to avoid this division, by packing the basic interaction with all the parameters necessary to complete the entire transaction. Ideally, each primitive task in the user's model of the application is executed with a single gesture. Such interfaces would have less modeness than the current so-called modeless interfaces.

The Macintosh discussion in the previous section is somewhat oversimplified. Many applications allow variations on the basic interaction techniques; for example "shift-click" (holding the shift key while clicking the mouse) adds an object to the current set of selected objects. Other computer systems allow different mouse buttons to indicated different operations. There is a tradeoff between having a small number and a large number of (consistently applied) interaction techniques. The former results in a system whose primitive operations are easy to learn, perform, and recall, but a single natural chunk may be divided into a sequence of operations. In the latter case, the primitive operations are harder to learn (because there are more of them), but each one can potentially implement an entire natural chunk.

The motivation for gestural interfaces may also apply to interfaces which combine modalities (*e.g.* speech and pointing). As with gestures, one potential benefit of multi-modal interfaces is that different modalities allow many parameters to be specified simultaneously, thus eliminating the need for modes. The "Put-That-There" system is one example [12].

1.5 Primitive Interactions

The discussion thus far has been vague as to what exactly may be considered a "primitive" interaction technique. The Macintosh has three: click, double-click, and drag. It is interesting to ask what criteria can be used for judging the "primitiveness" of proposed interaction techniques.

Buxton [14] suggests physical tension as a criterion. The user, starting from a relaxed state, begins a primitive interaction by tensing some muscles. The interaction is over when the user again relaxes those muscles. Buxton cites evidence that "such periods of tension are accompanied by a heightened state of attentiveness and improved performance." The three Macintosh interaction techniques all satisfy this concept of primitive interaction. (Presumably the user remains tense during a double-click because the time between clicks is short.)

Buxton likens the primitive interaction to a musical phrase. Each consists of a period of tension followed by a return to a state where a new phrase may be introduced. In human-computer interaction, such a phrase is used to accomplish a chunk of a task. The goal is to make each of these chunks a primitive task in the user's model of the application domain. This is what a gesture-based interface attempts to do.

1.6 The Anatomy of a Gesture

In this section a technical definition of gesture is developed, and the syntactic and semantic properties of gestures are then discussed. The dictionary definition of gesture, "expressive motion," has already been seen. How can the notion of gesture in a form suitable for sensing and processing by machine be captured?

1.6.1 Gestural motion

The motion aspect of gesture is formalized as follows: a gesture consists of the paths of multiple points over time. The points in question are (conceptually) affixed to the parts of the body which perform the gesture. For hand gestures, the points tracked might include the fingertips, knuckles, palm, and wrist of each hand. Over the course of a gesture, each point traces a path in space. Assuming enough points (attached to the body in appropriate places), these paths contain the essence of the gestural motion. A computer with appropriate hardware can rapidly sample positions along the paths, thus conveniently capturing the gesture.

The idea of gesture as the motion of multiple points over time is a generalization of pointing. Pointing may be considered the simplest gesture: it specifies a single position at an instance of time. This is generalized to allow for the movement of the point over time, *i.e.* a path. A further generalization admits multiple paths, *i.e.* the movement of multiple points over time.[5]

Current gesture-sensing hardware limits both the number of points which may be tracked simultaneously and the dimensionality of the space in which the points travel. Gestures limited to the motion of a single point are referred to here as *single-path* gestures. Most previous gestural research has focused upon gestures made with a stylus and tablet, mouse, or single-finger touch pad. The gestures which may be made with such devices are two-dimensional, single-path gestures.

An additional feature of existing hardware is that the points are not tracked at all times. For example, a touch pad can only determine finger position when the finger is touching the pad. Thus, the path of the point will have a beginning (when the finger first makes contact) and an end (when the finger is lifted). This apparent limitation of certain gesture-sensing hardware may be used to delineate the start and possibly the end of each gesture, a necessary function in gesture-based systems. Mouse buttons may be used to similar effect.[6]

In all the work reported here, a gesture (including the manipulation phase after recognition) is always a primitive interaction. A gesture begins with the user going from a relaxed state to one of muscular tension, and ends when the user again relaxes. It is further assumed that the tension or relaxation of the user is directly indicated by some aspect of the sensing hardware. For mouse gestures, the user is considered in a state of tension if and only if a mouse button is pressed. Thus, in the current work a double-click is not considered a gesture. This is certainly a limitation, but one that could be removed, for example by having a minimum time that the button needs to be released before the user is considered to have relaxed. This added complication has not been explored here.

The space in which the points of the gesture move is typically physical space, and thus a path is represented by a set of points (x, y, z, t) consisting of three spatial Cartesian coordinates and time. However, there are devices which measure non-spatial gestural parameters; hence, gestures consisting of paths through a space where at least some of the coordinates are not lengths are possible. For example, some touch pads can sense force, and for this hardware a gesture path might consist of a set of points (x, y, f, t), f being the force measurement at time t.

The formalization of gesture as multiple paths is just one among many possible representations. It is a good representation because it coincides nicely with most of the existing gesture-sensing hardware, and it is a useful form for efficient processing. The multiple-snapshot representation, in which each snapshot gives the position of multiple points at a single instant, is another possibility, and in some sense may be considered the dual of multiple paths. Such a representation might be more suitable for gestural data derived from hardware (such as video cameras) which are not considered in this dissertation.

1.6.2 Gestural meaning

In addition to the physical aspect of a gesture, there is the content or meaning of the gesture to consider. Generally speaking, a gesture contains two kinds of information: *categorical* and

---

[5] A configuration of multiple points at a single instance of time may be termed *posture*. Posture recognition is commonly used with the DataGlove.

[6] Buxton [17] presents a model of the discrete signaling capabilities of various pointing devices and a list of the signaling requirements for common interaction techniques.

parametric. Consider the different motions between people meaning “come here” (beckoning gestures), “stop” (prohibiting gestures), and “keep going” (encouragement gestures). These are different categories, or *classes*, of gestures. Within each class, a gesture also can indicate parametric data. For example, a parameter of the beckoning gesture is the urgency of the request: “hurry up” or “take your time.” In general, the category of the gesture must be determined before the parameters can be interpreted.

Parametric information itself comes in two forms. The first is the kind of information that can be culled at the time the gesture is classified. For example, the position, size and orientation of the gesture fall into this category. The second kind of parametric gestural information is manipulation information. After the gesture is recognized, the user can use this kind of parametric information to continuously communicate information. An example would be the directional information communicated by the gestures of a person helping a driver to back up a truck. An example from GDP (see Section 1.1) is the rubberbanding of a line after it is created, where the user continuously manipulates one endpoint.

The term “gesture” as used here does not exactly correspond to what is normally thought of as gesture. Many gestures cannot currently be processed by machine due to limitations of existing gesture-sensing hardware. Also, consider what might be referred to as “direct-manipulation gestures.” A person turning a knob would not normally be considered to be gesturing. However, a similar motion used to manipulate the graphic image of a knob drawn on a computer display is considered to be a gesture. Actually, the difference here is more illusory than real: a person might make the knob-turning gesture at another person, in effect asking the latter to turn the knob. The intent here is simply to point out the very broad class of motions considered herein to be gesture.

While the notion of gesture developed here is very general (multiple paths), in practice, machine gestures have hitherto almost always been limited to finger and/or hand motions. Furthermore, the paths have largely been restricted to two dimensions. The concentration on two-dimensional hand gesturing is a result of the available gesture-sensing hardware. Of course, such hardware was built because it was believed that hand and fingers are capable of accurate and diverse gesturing, yet more amenable to practical detection than facial or other body motions. With the appearance of new input devices, three (or more) dimensional gesturing, as well as the use of parts of the body other than the hand, are becoming possible. Nonetheless, this dissertation concentrates largely on two-dimensional hand gestures, assuming that by viewing gesture simply as multiple paths, the work described may be applied to non-hand gestures, or generalized to apply to gestures in three or more dimensions.

1.7 Gesture-based systems

A gesture-based interface, as the term is used here, is one in which the user specifies commands by gesturing. Typically, gesturing consists of drawing or other freehand motions. Excluded from the class of gesture-based interfaces are those in which input is done solely via keyboard, menu, or click-and-drag interactions. In other words, while pointing is in some sense the most basic gesture, those interfaces in which pointing is the only form of gesture are not considered here to be gesture-based interfaces. A gesture-based system is a program (or set of programs) with which the user interacts via a gesture-based interface.

In all but the simplest gesture-based systems, the user may enter a gesture belonging to one of several different gesture categories or classes; the different classes refer to different commands to the system. An important component of gesture-based systems is the gesture *recognizer* or *classifier*, the module whose job is to classify the user's gesture as the first step toward inferring its meaning. This dissertation addresses the implementation of gesture recognizers, and their incorporation into gesture-based systems.

1.7.1 The four states of interaction

User interaction with the gesture-based systems considered in this dissertation may be described using the following four state model. The states–WAIT, COLLECT, MANIPULATE, EXECUTE–usually occur in sequence for each interaction.

- The WAIT state is the quiescent state of the system. The system is waiting for the user to initiate a gesture.

- The COLLECT state is entered when the user begins to gesture. While in this state, the system collects gestural data from the input hardware in anticipation of classifying the gesture. For most gesturing hardware, an explicit start action (such as pressing a mouse button) indicates the beginning of each gesture, and thus causes the system to enter this state.

- The MANIPULATE state is entered once the gesture is classified. This occurs in one of three ways:

    1. The end of the gesture is indicated explicitly, *e.g.* by releasing the mouse button;
    2. the end of the gesture is indicated implicitly, *e.g.* by a timeout which indicates the user has not moved the mouse for, say, 200 milliseconds; or
    3. the system initiates classification because it believes it has now seen enough of the gesture to classify it unambiguously (eager recognition).

  When the MANIPULATE state is entered, the system should provide feedback to the user as to the classification of the gesture and update any screen objects accordingly. While in this state, the user can further manipulate the screen objects with his motions.

- The EXECUTE state is entered when the user has completed his role in the interaction, and has indicated such (*e.g.* by releasing the mouse button). At this point the system performs any final actions as implied by the user's gesture. Ideally, this state lasts only a very short time, after which the display is updated to reflect the current state of the system, and the system reverts back to the WAIT state.

This model is sufficient to describe most current systems which use pointing devices. (For simplicity, keyboard input is ignored.) Depending on the system, the COLLECT or MANIPULATE state may be omitted from the cycle. A handwriting interface will usually omit the MANIPULATE state, classifying the collected characters and executing the resulting command. Conversely, a direct-manipulation system will omit the COLLECT state (and the attendant classification). The GDP example described above has both COLLECT and MANIPULATE phases. The result is the new two-phase interaction technique mentioned earlier.

1.8 A Comparison with Handwriting Systems

In this section, the frequently asked question, "how do gesture-based systems differ from handwriting systems?" is addressed.

Handwriting systems may broadly be grouped into two classes: on-line and off-line. On-line handwriting recognition simply means characters are recognized as they are drawn. Usually, the characters are drawn with a stylus on a tablet, thus the recognition process takes as input a list of successive points or line segments. The problem is thus considerably different than off-line handwriting recognition, in which the characters are first drawn on paper, and then optically scanned and represented as two-dimensional rasters. Suen, Berthod, and Mori review the literature of both on-line and off-line handwriting systems [125], while Tappert, Suen, and Wakaha [129] give a recent review of on-line handwriting systems. The intention here is to contrast gesture-based systems with on-line handwriting recognition systems, as these are the most closely related.

Gesture-based systems have much in common with systems which employ on-line handwriting recognition for input. Both use freehand drawing as the primary means of user input, and both depend on recognizers to interpret that input. However, there are some important differences between the two classes of systems, differences that illustrate the merits of gesture-based systems:

- Gestures may be motions in two, three, or more dimensions, whereas handwriting systems are necessarily two-dimensional. Similarly, single-path and multiple-path gestures are both possible, whereas handwriting is always a single path.

- The alphabet used in a handwriting system is generally well-known and fixed, and users will generally have lifelong experience writing that alphabet. With gestures, it is less likely that users will have preconceptions or extensive experience.

- In addition to the command itself, a single gesture can specify parameters to the command. The proofreader's gesture (figure 1.1) discussed above, is an excellent example. Another example, also due to Buxton [21], and used in GSCORE (Section 8.2), is a musical score editor, in which a single stroke indicates the location, pitch, and duration of a note to be added to the score.

- As stated, a command and all its parameters may be specified with a single gesture. The physical relaxation of the user when she completes a gesture reinforces the conceptual completion of a command [14].

- Gestures of a given class may vary in both size and orientation. Typical handwriting recognizers expect the characters to be of a particular size and oriented in the usual manner (though successful systems will necessarily be able to cope with at least small variations in size and orientation). However, some gesture commands may use the size and orientation to specify parameters; gesture recognizers must be able to recognize such gestures in whatever size and orientation they occur. Kim [67] discusses augmenting a handwriting recognition system so as to allow it to recognize some gestures independently of their size and orientation. Chapter 3 discusses the approach taken here toward the same end.

- Gestures can have a dynamic component. Handwriting systems usually view the input character as a static picture. In a gesture-based system, the same stroke may have different meanings if drawn left-to-right, right-to-left, quickly, or slowly. Gesture recognizers may use such directional and temporal information in the recognition process.

In summary, gestures may potentially deal in dimensions other than the two commonly used in handwriting, be drawn from unusual alphabets, specify entire commands, vary in size and orientation, and have a dynamic component. Thus, while ideas from on-line handwriting recognition algorithms may be used for gesture recognition, handwriting recognizers generally rely on assumptions that make them inadequate for gesture recognition. The ideal gesture recognition algorithm should be adaptable to new gestures, dimensions, additional features, and variations in size and orientation, and should produce parametric information in addition to a classification. Unfortunately, the price for this generality is the likelyhood that a gesture recognizer, when used for handwriting recognition, will be less accurate than a recognizer built and tuned specifically for handwriting recognition.

1.9 Motivation for this Research

In spite of the potential advantages of gesture-based systems, only a handful have been built. Examples include Button Box [86], editing using proofreader's symbols [25], the Char-rec note-input tool [21], and a spreadsheet application built at IBM [109]. These and other gesture-based systems are discussed in section 2.2. Gesture recognition in most existing systems has been done by writing code to recognize the particular set of gestures used by the system. This code is usually complicated, making the systems (and the set of gestures accepted) difficult to create, maintain, and modify. These difficulties are the reasons more gesture-based systems have not been built.

One goal of the present work is to eliminate hand-coding as the way to create gesture recognizers. Instead, gesture classes are specified by giving examples of gestures in the class. From these examples, recognizers are automatically constructed. If a particular gesture class is to be recognized in any size or orientation, its examples of the class should reflect that. Similarly, by making all of the examples of a given class the same size or orientation, the system learns that gestures in this class must appear in the same size or orientation as the examples. The first half of this dissertation is concerned with the automatic construction of gesture recognizers.

Even given gesture recognition, it is still difficult to build direct-manipulation systems which incorporate gestures. This is the motivation for the second half of this dissertation, which describes GRANDMA–Gesture Recognizers Automated in a Novel Direct Manipulation Architecture.

1.10 Criteria for Gesture-based Systems

The goal of this research was to produce tools which aid in the construction of gesture-based systems. The efficacy of the tools may be judged by how well the tools and resulting gesture-based systems satisfy the following criteria.

1.10.1 Meaningful gestures must be specifiable

A meaningful gesture may be rather complex, involving simultaneous motions of a number of points. These complex gestures must be easily specifiable. Two methods of specification are possible: specification by example, and specification by description. In the former, each application has a training session in which examples of the different gestures are submitted to the system. The result of the training is a representation for all gestures that the system must recognize, and this representation is used to drive the actual gesture recognizer that will run as part of the application. In the latter method of specification, a description of each gesture is written in a gesture description language, which is a formal language in which the "syntax" of each gesture is specified. For example, a set of gestures may be specified by a context-free grammar, in which the terminals represent primitive motions (e.g. "straight line segment") and gestures are non-terminals composed of terminals and other non-terminals.

All else being equal, the author considers specification by example to be superior to specification by description. In order to specify gestures by description, it will be necessary for the specifier to learn a description language. Conversely, in order to specify by example, the specifier need only be able to gesture. Given a system in which gestures are specified by example, the possibility arises for end users to train the system directly, either to replace the existing gestures with ones more to their liking, or to have the system improve its recognition accuracy by adapting to the particular idiosyncrasies of a given user's gestures.

One potential drawback of specification by example is the difficulty in specifying the allowable variations between gestures of a given class. In a description language, it can be made straightforward to declare that gestures of a given class may be of any size or of any orientation. The same information might be conveyed to a specify-by-example system by having multiple examples of a single class vary in size or orientation. The system would then have to *infer* that the size or orientation of a given gesture class was irrelevant to the classification of the gesture. Also, training classifiers may take longer, and recognition may be less accurate, when using examples as specifications, though this is by no means necessarily so. Similar issues arise in demonstrational interfaces [97].

1.10.2 Accurate recognition

An important characterization of a gesture recognition system will be the frequency with which gestures fail to be recognized or are recognized incorrectly. Obviously it is desirable that these numbers be made as small as possible. Questions pertaining to the amount of inaccuracy acceptable to people are difficult to answer objectively. There will likely be tradeoffs between the complexity of gestures, the number of different gestures to be disambiguated, the time needed for recognition, and the accuracy of recognition.

In speech recognition there is the problem that the accuracy of recognition decreases as the user population grows. However the analogous problem in gesture recognition is not as easy to gauge. Different people speak the same words differently due to inevitable differences in anatomy and upbringing. The way a person says a word is largely determined before she encounters a speech recognition system. By contrast, most people have few preconceptions of the way to gesture at a machine. People will most likely be able to adapt themselves to gesturing in ways the machine understands. The recognition system may similarly adapt to each user's gestures. It would be interesting, though outside the scope of this dissertation, to study the fraction of incorrectly recognized gestures as a function of a person's experience with the system.

1.10.3 Evaluation of accuracy

It should be possible for a gesture-based system to monitor its own performance with respect to accuracy of recognition. This is not necessarily easy, since in general it is impossible to know which gesture the user had intended to make. A good gesture-based system should incorporate some method by which the user can easily inform the system when a gesture has been classified incorrectly. Ideally, this method should be integrated with the undo or abort features of the systems. (Lerner [78] gives an alternative in which subsequent user actions are monitored to determine when the user is satisfied with the results of system heuristics.)

1.10.4 Efficient recognition

The goal of this work is to enable the construction of applications that use gestures as input, the idea being that gesture input will enhance human/computer interaction. Speed of recognition is very important–a slow system would be frustrating to use and hinder rather than enhance interaction

Speed is a very important factor in the success or failure of user interfaces in general. Baecker and Buxton [5] state that one of the chief determinants of user satisfaction with interactive computer systems is response time. Poor performance in a direct-manipulation system is particularly bad, as any noticeable delay destroys the feeling of directness. Rapid recognition is essential to the success of gesture as a medium for human-computer interaction, even if achieving it means sacrificing certain features or, perhaps, a limited amount of recognition accuracy.

1.10.5 On-line/real-time recognition

When possible, the recognition system should attempt to match partial inputs with possible gestures. It may also be desirable to inform the user as soon as possible when the input does not seem to match any possible gesture. An on-line/real-time matching algorithm has these desirable properties. The gesture recognition algorithms discussed in Chapters 3, 4, and 5 all do a small, bounded amount of work given each new input point, and are thus all on-line/real-time algorithms.

1.10.6 General quantitative application interface

An application must specify what happens when a gesture is recognized. This will often take the form of a callback to an application-specific routine. There is an opportunity here to relay the parametric data contained in the gesture to the application. This includes the parametric data which can be derived when the gesture is first recognized, as well as any manipulation data which follows.

1.10.7 Immediate feedback

In certain applications, it is desirable that the application be informed immediately once a gesture is recognized but before it is completed. An example is the turning of a knob: once the system recognizes that the user is gesturing to turn a knob it can monitor the exact details of a gesture, relaying quantitative data to the application. The application can respond by immediately and continuously varying the parameter which the knob controls (for example the volume of a musical instrument).

1.10.8 Context restrictions

A gesture sensing system should be able, within a single application, to sense different sets of gestures in different contexts. An example of a context is a particular area of the display screen. Different areas could respond to different sets of gestures. The set of gestures to which the application responds should also be variable over time–the application program entering a new mode could potentially cause a different set of gestures to be sensed.

The idea of contexts is closely related to the idea of using gestures to manipulate graphic objects. Associated with each picture of an object on the screen will be an area of the screen within which gestures refer to the object. A good gesture recognition system should allow the application program to make this association explicit.

1.10.9 Efficient training

An ideal system would allow the user to experiment with different gesture classes, and also adapt to the user's gestures to improve recognition accuracy. It would be desirable if the system responded immediately to any changes in the gesture specifications; a system that took several hours to retrain itself would not be a good platform for experimentation.

1.10.10 Good handling of misclassifications

Misclassifications of gestures are a fact of life in gesture-based systems. A typical system might have a recognition rate of 95% or 99%. This means one out of twenty or one out of one hundred gestures will be misunderstood. A gesture-based system should be prepared to deal with the possibility of misclassification, typically by providing easy access to abort and undo facilities.

1.10.11 Device independence

Certain assumptions about the form of the input data are necessary if gesture systems are to be built. As previously stated, the assumption made here is that the input device will supply position as a function of time for each input "path" (or supply data from which it is convenient to calculate such positions). (A path may be thought of as a continuous curve drawn by a single finger.) This form of data is supplied by the Sensor Frame, and (at least for the single finger case) a mouse and a clock can be made to supply similar data. The recognition systems should do their recognition based on the position versus time data; in this way other input devices may also benefit from this research.

1.10.12 Device utilization

Each particular brand of input hardware used for gesture sensing will have characteristics that other brands of hardware will not have. It would be unfortunate not to take advantage of all the special features of the hardware. For example, the Sensor Frame can compute finger angle and finger velocity.[7] While for device independence it may be desirable that the gesture matching not depend on the value of these inputs, there should be some facility for passing these parameters to the application specific code, if the application so desires. Baecker [4] states the case strongly: "Although portability is facilitated by device-independence, interactivity and usability are enhanced by *device dependence*."

1.11 Outline

The following chapter describes previous related work in gesture-based systems. This is divided into four sections: Section 2.1 discusses various hardware devices suitable for gestural input. Section 2.2 discusses existing gesture-based systems. Section 2.3 reviews the various approaches to pattern recognition in order to determine their potential for gesture recognition. Section 2.4 examines existing software systems and toolkits that are used to build direct-manipulation interfaces. Ideas from such systems will be generalized in order to incorporate gesture recognition into such systems.

Everything after Chapter 2 focuses on various aspects of the gesture-based interface creation tool built by the author. Such a tool makes it easy to 1) specify and create classifiers, and 2) associate gestures classes and their meanings with graphic objects. The former goal is addressed in Chapters 3, 4, and 5, the latter in 6 and 7.

The discussion of the implementation of gesture recognition begins in Chapter 3. Here the problem of classifying single-path, two-dimensional gestures is tackled. This chapter assumes that the start and end of the gesture are known, and uses statistical pattern recognition to derive efficient gesture classifiers. The training of such classifiers from example gestures is also covered.

Chapter 3 shows how to classify single-path gestures; Chapter 4 shows *when*. This chapter addresses the problem of recognizing gestures while they are being made, without any explicit indication of the end of the gesture. The approach taken is to define and construct another classifier. This classifier is intended solely to discriminate between ambiguous and unambiguous subgestures.

Chapter 5 extends the statistical approach to the recognition of multiple-path gestures. This is useful for utilizing devices that can sense the positions of multiple fingers simultaneously, in particular the Sensor Frame.

Chapter 6 presents the architecture of an object-oriented toolkit for the construction of direct-manipulation systems. Like many other systems, this architecture is based on the Model-View-

---

[7] This describes the Sensor Frame as originally envisioned. The hardware is capable of producing a few bits of finger velocity and angle information, although to date this has not been attempted.

Controller paradigm. Compared to previous toolkits, the input model is considerably generalized in preparation for the incorporation of gesture recognition into a direct-manipulation system. The notion of virtual tools, through which input may be generated by software objects in the same manner as by hardware input devices, is introduced. Semantic feedback will be shown to arise naturally from this approach.

Chapter 7 shows how gesture recognizers are incorporated into the direct-manipulation architecture presented in Chapter 6. A gesture handler may be associated with a particular view of an object on the screen, or at any level in the view hierarchy. In this manner, different objects will respond to different sets of gestures. The communication of parametric data from gesture handler to application is also examined.

Chapter 8 discusses three gesture-based systems built using these techniques: GDP, GSCORE, and MDP. The first two, GDP and GSCORE, use mouse gestures. GDP, as already mentioned, is the drawing editor based on DP. GSCORE is a musical score editor, based on Buxton's SSSP work [21]. MDP is also a drawing editor, but it operates using multi-path gestures made with a Sensor Frame. The design and implementation of each system is discussed, and the gestures for each shown.

Chapter 9 evaluates a number of aspects of this work. The particular recognition algorithms are tested for recognition accuracy. Measurements of the performance of the gesture classifiers used in the applications is presented. Then, an informal user study assessing the utility of gesture-based systems is discussed.

Finally, Chapter 10 concludes this dissertation. The contributions of this dissertation are discussed, as are the directions for future work.

1.12 What Is Not Covered

This dissertation attempts to cover many topics relevant to gesture-based systems, though by no means all of them. In particular, the issues involved in the ergonomics and suitability of gesture-based systems applied to various task domains have not been studied. It is the opinion of the author that such issues can only be studied after the tools have been made available which allow easy creation of and experimentation with such systems. The intent of the current work is to provide such tools. Future research is needed to determine how to use the tools to create the most usable gesture-based systems possible.

Of course, choices have had to be made in the implementation of such tools. By avoiding the problem of determining which kind of gesture-based systems are best, the work opens itself to charges of possibly "throwing the baby out with the bath-water." The claim is that the general system produced is capable of implementing systems comparable to many existing gesture-based systems; the example applications implemented (see Chapter 8) support this claim. Furthermore, the places where restrictive choices have been made (*e.g.* two-dimensional gestures) have been indicated, and extensible and scalable methods (*e.g.* linear discrimination) have been used wherever possible.

There are two major limitations of the current work. The first is that single-path multi-stroke gestures (*e.g.* handwritten characters) are not handled. Most existing gesture-based systems use single-path multi-stroke gestures. The second limitation is that the start of a gesture must be explicitly indicated. This rules out (at least at first glance) using devices such as the DataGlove which lack buttons or other explicit signaling hardware. However, one result of the current work is that these apparent limitations give rise to certain advantages in gestural interfaces. For example, the limitations enforce Buxton's notion of tension and release mentioned above.

Gestural output, *i.e.* generating a gesture in response to a query, is also not covered. For an example of gestural output, ask the author why he has taken so long to complete this dissertation.

Chapter 2

Related Work

This chapter discusses previous work relevant to gesture recognition. This includes hardware devices suitable for gestural input, existing gesture-based systems, pattern recognition techniques, and software systems for building user interfaces.

Before delving into details, it is worth mentioning some general work that attempts to define gesture as a technique for interacting with computers. Morrel-Samuels [87] examines the distinction between gestural and lexical commands, and then further discusses problems and advantages of gestural commands. Wolf and Rhyne [140] integrate gesture into a general taxonomy of direct manipulation interactions. Rhyne and Wolf [109] discuss in general terms human-factors concerns of gestural interfaces, as well as hardware and software issues.

The use of gesture as an interaction technique is justified in a number of studies. Wolf [139] performed two experiments that showed gestural interfaces compare favorably to keyboard interfaces. Wolf [141] showed that many different people naturally use the same gestures in a text-editing context. Hauptmann [49] demonstrated a similar result for an image manipulation task, further showing that people prefer to combine gesture and speech rather than use either modality alone.

2.1 Input Devices

A number of input devices are suitable for providing input to a gesture recognizer. This section concentrates on those devices which provide the position of one or more points over time, or whose data is easily converted into that representation. The intention is to list the types of devices which can potentially be used for gesturing. The techniques developed in this dissertation can be applied, directly or with some generalization, to the devices mentioned.

A large variety of devices may be used as two-dimensional, single-path gesturing devices. Some graphical input devices, such as mice [33], tablets and styli, light pens, joysticks, trackballs, touch tablets, thumb-wheels, and single-finger touch screens [107, 124], have been in common use for years. Less common are foot controllers, knee controllers, eye trackers [12], and tongue-activated joysticks. Each may potentially be used for gestural input, though ergonomically some are better suited for gesturing than others. Baecker and Buxton [5], Buxton [14], and Buxton, Hill and Rowley [18] discuss the suitability of many of the above devices for various tasks. Buxton further points out that two different joysticks, for example, may have very different properties that must be considered with respect to the task.

For gesturing, as with pointing, it is useful for a device to have some signaling capability in addition to the pointer. For example, a mouse usually has one or more buttons, the pressing of which can be used to indicate the start of a gesture. Similarly, tablets usually indicate when the stylus makes or breaks contact with the tablet (though with a tablet it is not possible to carefully position the screen cursor before contact). If a device does not have this signaling capacity, it will be necessary to simulate it somehow. Exactly how this is done can have a large impact on whether or not the device will be suitable for gesturing.

The 3SPACE Isotrack system, developed by Polhemus Navigation Sciences Division of McDonnel Douglas Electronics Company [32], is a device which measures the position and orientation of a stylus or a one-inch cube using magnetic fields. The Polhemus sensor, as it is often called, is a full six-degree-of-freedom sensor, returning x, y, and z rectangular coordinates, as well as azimuth, altitude, and roll angles. It is potentially useful for single path gesturing in three positional dimensions. By considering the angular dimensions, 4, 5, or 6 dimensional gestures may be entered. It is also possible to use one of the angular dimensions for signaling purposes.

Bell Laboratories has produced prototypes of a clear plate capable of detecting the position and pressure of many fingers [10, 99]. The position information is two-dimensional, and there is a third dimension as well: finger pressure. The author has seen the device reliably track 10 fingers simultaneously. The pressure detection may be used for signaling purposes, or as a third dimension for gesturing. The inventor of the multi-finger touch plate has invented another device, the Radio Drum [11], which can sense the position of multiple antennae in three dimensions. To date, the antennae have been embedded in the tips of drum sticks (thus the name), but it would also be possible to make a glove containing the antenna which would make the device more suitable for detecting hand gestures.

The Sensor Frame [84] is a frame mounted on a workstation screen (figure 2.1). It consists of a light source (which frames the screen) and four optical sensors (one in each corner). The Sensor Frame computes the two-dimensional positions of up to three fingertips in a plane parallel to, and slightly above the screen. The net result is similar to a multi-finger touch screen. The author has used the Sensor Frame to verify the multi-finger recognition algorithm described in Chapter 5. The Sensor Cube [85] is a device similar to the Sensor Frame but capable of sensing finger positions in three dimensions. It is currently under construction. The VideoHarp [112, 111] is a musical instrument based on the same sensing technology, and is designed to capture parametric gestural data.

The DataGlove [32, 130] is a glove worn on the hand able to produce the positions of multiple fingers as well as other points on the hand in three dimensions. By itself it can only output relative positions. However, in combination with the Polhemus sensor, absolute finger positions can be computed. Such a device can translate gestures as complex as American Sign Language [123] into a multi-path form suitable for processing. The DataGlove, the similar Dexterous Hand Master from Exos, and the Power Glove from Mattel, are shown in figure 2.2.

The DataGlove comes with hardware which may be trained to recognize certain static configurations of the glove. For example, the DataGlove hardware might be trained to recognize a fist,

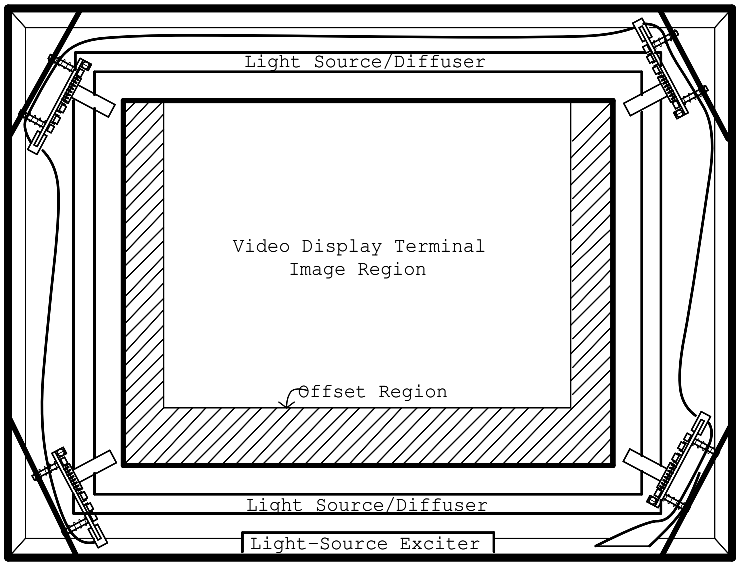


Figure 2.1: The Sensor Frame

The Sensor Frame is a frame mounted on a computer display consisting of a rectangular light source and four sensors, one in each corner. It is capable of detecting up to three fingers its field of view. (Drawing by Paul McAvinney.)

Figure 2.2: The DataGlove, Dexterous Hand Master, and PowerGlove (from Eglowstein [32])

The DataGlove, Dexterous Hand Master, and PowerGlove are three glove-like input devices capable of measuring the angles of various hand and finger joints.

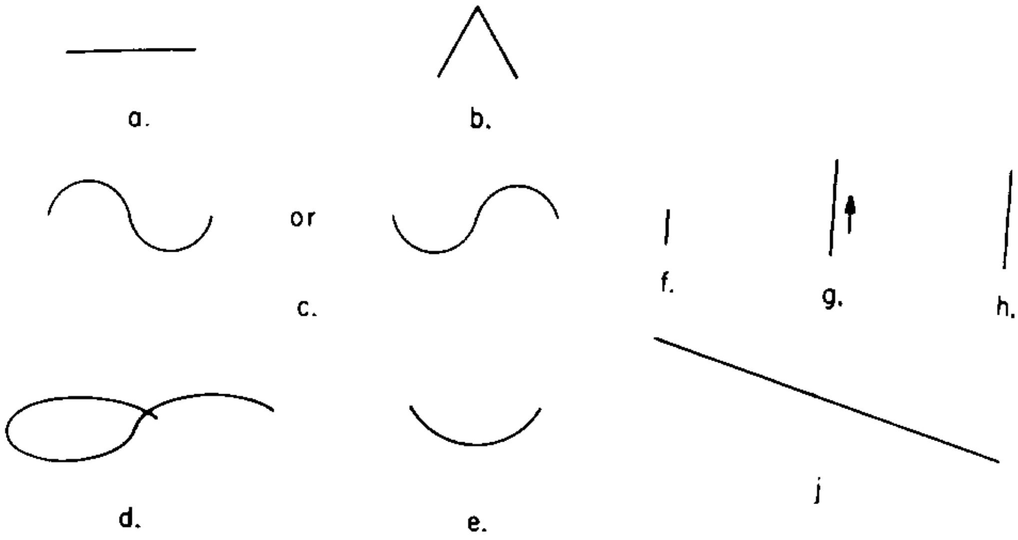


Figure 2.3: Proofreading symbols (from Coleman [25])

The operations intended by each are as follows: a) delete text (from a single line), b) insert text, c) swap text, d) move text, e) join (delete space), f) insert space, g) scroll up, h) scroll down, and i) delete multiple lines of text. Many of the marks convey additional parameters to the operation, e.g. *the text to be moved or deleted.*

signaling the host computer whenever a fist is made. These static hand positions are not considered to be gestures, since they do not involve motion. The glove hardware recognizes "posture" rather than gesture, the distinction being that posture is a static snapshot (a pose), while gesture involves motion over time. Nonetheless, it is a rather elegant way to add signaling capability to a device without buttons or switches.

The Videodesk [71, 72] is an input device based on a constrained form of video input. The Videodesk consists of a translucent tablecloth over a glass top. Under the desk is a light source, over the desk a video camera. The user's hands are placed over the desk. The tablecloth diffuses the light, the net effect being that the camera receives an image of the silhouette of the hands. Additional hardware is used to detect and track the user's fingertips.

Some researchers have investigated the attachment of point light sources to various points on the body or hand to get position information as a function of time. The output of a camera (or pair of cameras for three dimensional input) can be used as input to a gesture sensor.

2.2 Example Gesture-based Systems

This section describes a number of existing gesture-based systems that have been described in the literature. A system must both classify its gestural input and use information other than the class (*i.e.* parametric information) to be included in this survey. The order is roughly chronological.

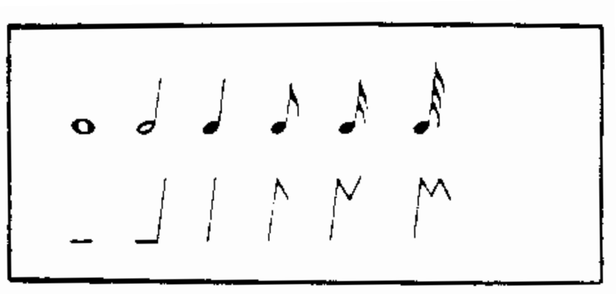


Figure 2.4: Note gestures (from Buxton [21])

A single gesture indicates note duration (from the shape of the stroke as shown) as well as pitch and starting time, both of which are determined from the position of the start of the gesture.

Coleman [25] has created a text editor which used hand-drawn proofreader's symbols to specify editing commands (figure 2.3). For example, a sideways "S" indicated that two sets of characters should be interchanged, the characters themselves being delimited by the two halves of the "S." The input device was a touch tablet, and the gesture classification was done by a hand-coded discrimination net (*i.e.* a loop-free flowchart).[1]

Buxton [21] has built a musical score editor with a small amount of gesture input using a mouse (figure 2.4). His system used simple gestures to indicate note durations and scoping operations. Buxton considered this system to be more a character recognition system than a gesture-based system, the characters being taken from an alphabet of musical symbols. Since information was derived not only from the classification of the characters, but their positions as well, the author considers this to be a gesture-based system in the true sense. Buxton's technique was later incorporated into Notewriter II, a commercial music scoring program. Lamb and Buckley [76] describe a gesture-based music editor usable by children.

Margaret Minsky [86] implemented a system called Button Box, which uses gestures for selection, movement, and path specification to provide a complete Logo programming environment (figure 2.5). Her input device was a clear plate mounted in front of a display. The device sensed the position and shear forces of a single finger touching the plate. Minksy proposed the use of multiple fingers for gesture input, but never experimented with an actual multiple-finger input device.

In Minsky's system, buttons for each Logo operation were displayed on the screen. Tapping a button caused it to execute; touching a button and dragging it caused it to be moved. The classification needed to distinguish between a touch and a tap was programmed by hand. There were buttons used for copying other buttons and for grouping sets of buttons together. A path could be drawn through a series of buttons–touching the end of a path caused its constituent buttons to execute sequentially.

VIDEOPLACE [72] is a system based on the Videodesk. As stated above, the silhouette of the user's hands are monitored. When a hand is placed in a pointing posture, the tip of the index finger

---

[1] Curiously, this research was done while Coleman was a graduate student at Carnegie Mellon. Coleman apparently never received a Ph.D. from CMU, and it would be twenty years before another CMU graduate student (me) would go near the topic of gesture recognition.

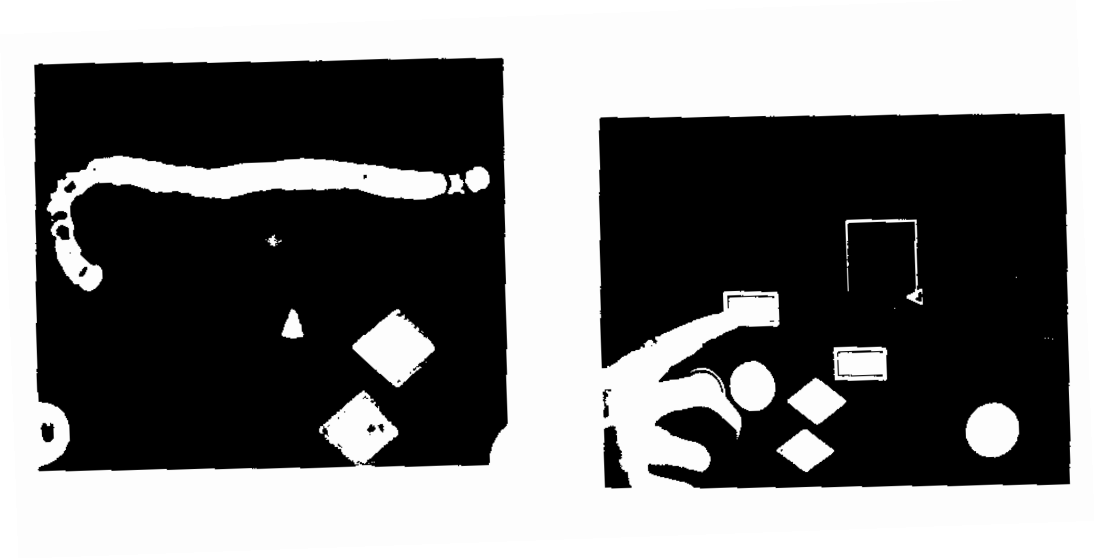
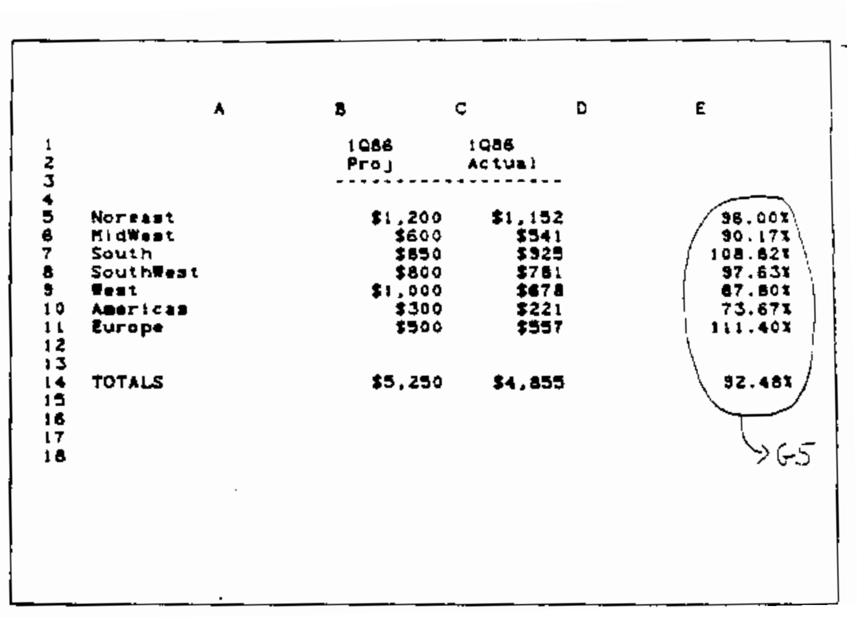


Figure 2.5: Button Box (from Minksy [86])

Tapping a displayed button causes it to execute its assigned function while touching a button and dragging it causes it to be moved.



	A	B	C	D	E
1		1Q86	1Q86		
2		Proj	Actual		
3		----------	----------		
4					
5	Noreast	$1,200	$1,152		96.00%
6	MidWest	$600	$541		90.17%
7	South	$850	$925		108.82%
8	SouthWest	$800	$781		97.63%
9	West	$1,000	$678		67.80%
10	Americas	$300	$221		73.67%
11	Europe	$500	$557		111.40%
12					
13					
14	TOTALS	$5,250	$4,855		92.48%
15					
16					
17					
18					

G5

Figure 2.6: A gesture-based spreadsheet (from Rhyne and Wolf [109])

The Paper-Like Interface project produces systems which combine gesture and handwriting. The input shown here selects a group of cells and requests they be moved to the cell beginning at location "G5."

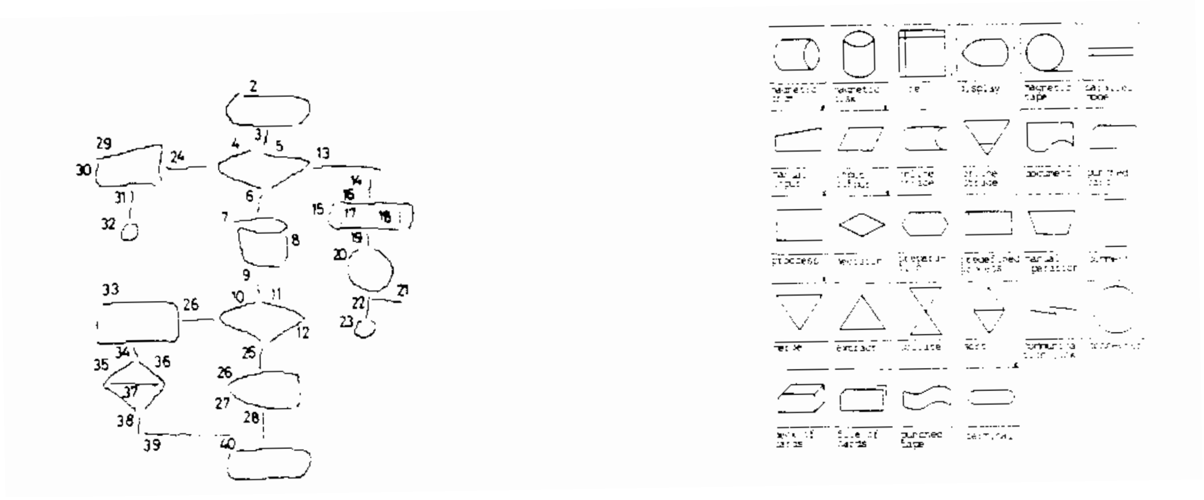


Figure 2.7: Recognizing flowchart symbols

Recognizing flowchart symbols (from Murase and Wakahara [89]). The system takes an entire freehand drawing of a flowchart (left) and recognizes the individual flowchart symbols (right), producing an internal representation of the flowchart (as nodes and edges) and a flowchart picture in which the freehand symbols are replaced by machine generated line-drawings drawn from the alphabet of symbols. This system shows a style of interface in which pattern recognition is used for something other than the detection of gestures or characters.

may be used for menu selection. After selection, the fingertips may be used to manipulated graphic objects, such as the controlling points of a spline curve.

A group at IBM doing research into gestural human-computer systems has produced a gesture-based spreadsheet application [109]. Somewhat similar to Coleman's editor, the user manipulates the spreadsheet by gesturing with a stylus on a tablet (figure 2.6). For example, deletion is done by drawing an "X" over a cell, selection by an "0", and moving selected cells by an arrow, the tip of which indicates the destination of the move. The application is interesting in that it combines handwriting recognition (isolated letters and numbers) with gesturing. For example, by using handwriting the user can enter numbers or text into a cell without using a keyboard. The portion of the recognizer which classifies letters, numbers, and gestures of a fixed size and orientation has (presumably) been trained by example using standard handwriting recognition techniques. However, the recognition of gestures which vary in size or orientation requires hand coding [67].

Murase and Wakahara [89] describe a system in which freehand-drawn flowcharts symbols are recognized by machine (figure 2.7). Tamura and Kawasaki [128] have a system which recognizes sign-language gestures from video input (figure 2.8).

HITS from MCC [55] and Artkit from the University of Arizona [52] are both systems that may be used to construct gesture-based interfaces. The author has seen a system built with HITS similar

Figure 2.8: Sign language recognition (from Tamura [128])
This system processes an image from a video camera in order to recognize a form of Japanese sign language.

to that of Murase; in it an entire control panel is drawn freehand, and then the freehand symbols are segmented, classified and replaced by icons. (Similar work is discussed by Martin, *et. al* [82], also from MCC.) Artkit has much in common with the GRANDMA system described in this dissertation, and will be mentioned again later (Sections 4.1 and 6.8). Artkit systems tend to be similar to those created using GRANDMA, in that gesture commands are executed as soon as they are entered.

Kurtenbach and Buxton [75] have implemented a drawing program based on single-stroke gestures (figure 2.9). They have used the program to study, among other things, issues of scope in gestural systems. To the present author, GEdit's most interesting attribute is the use of compound gestures, as shown in the figure. GEdit's gesture recognizer is hand-coded.

The Glove-talk system [34] uses a DataGlove to control a speech synthesizer (figure 2.10). Like Artkit and the work described in Chapter 4, Glove-talk performs eager recognition: a gesture is recognized and acted upon without its end being indicated explicitly. Weimer and Ganapathy [136] describe a system combining DataGlove gesture and speech recognition.

The use of the circling gesture as an alternative means of selection is considered in Jackson and Roske-Hofstrand [61]. In their system, the start of the circling gesture is detected automatically, *i.e.* the mouse buttons are not used. Circling is also used for selection in the JUNO system from Xerox Corporation [142].

A number of computer products offer a stylus and tablet as their sole or primary input device. These systems include GRID Systems Corp.'s GRIDPad [50], Active Book Company's new portable [43], Pencept Inc.'s computer [59], Scenario's DynaWriter, Toshiba's PenPC, Sony's Palmtop, Mometa's laptop, MicroSlate's Datalite, DFM System's TraveLite, Agilis Corp.'s system, and Go Corp.'s PenPoint system [81, 24]. While details of the interface of many of these systems are hard to find (many of these systems have not yet been released), the author suspects that many use gestures. For further reading, please see [16, 106, 31].

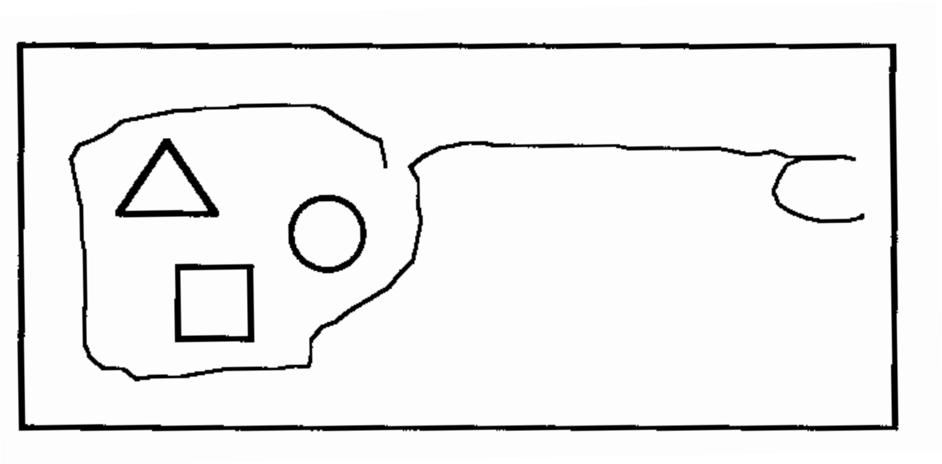


Figure 2.9: Copying a group of objects in GEdit (from Kurtenbach and Buxton [75])

Note the compound gesture: the initial closed curve does selection, and the final "C" indicates the data should be copied rather than moved.

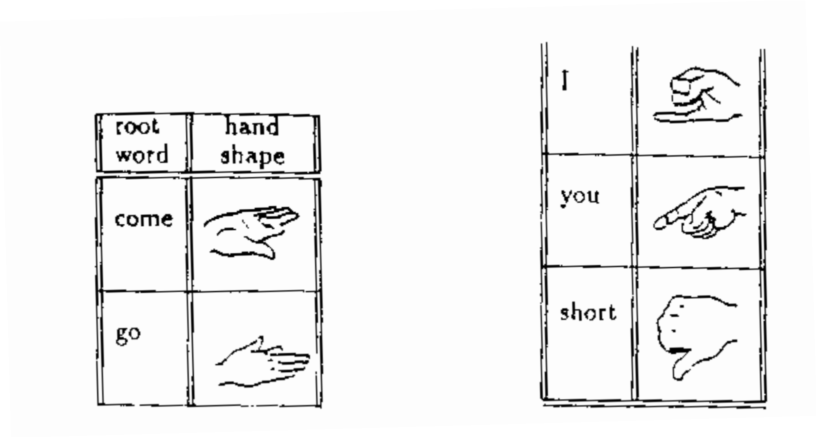
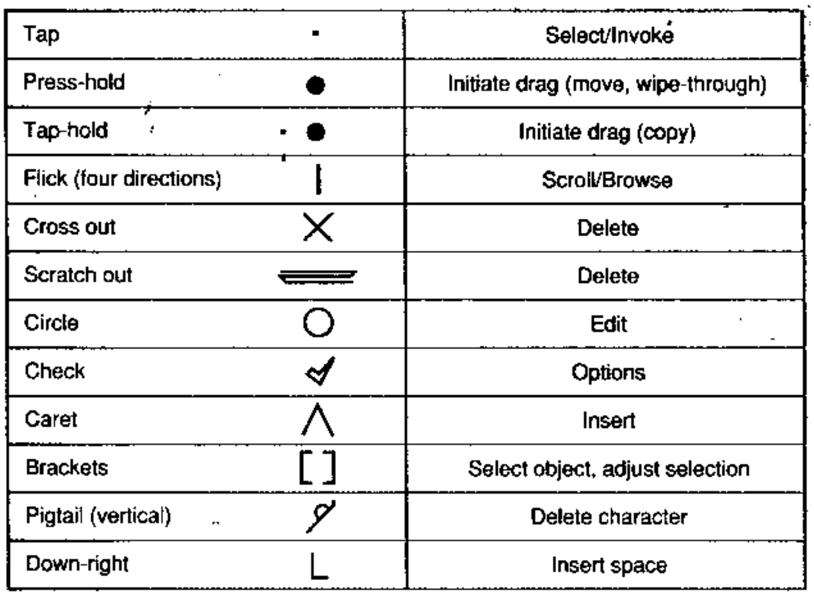


Figure 2.10: GloveTalk (from Fels and Hinton [34])

GloveTalk connects a DataGlove to a speech synthesizer through several neural networks. Gestures indicate root words (shown) and modifiers. Reversing the direction of the hand motion causes a word to be emitted from the synthesizer as well as indicating the start of the next gesture.



Tap	·	Select/Invoke
Press-hold	●	Initiate drag (move, wipe-through)
Tap-hold	· ●	Initiate drag (copy)
Flick (four directions)	|	Scroll/Browse
Cross out	✕	Delete
Scratch out	═	Delete
Circle	○	Edit
Check	✓	Options
Caret	∧	Insert
Brackets	[ ]	Select object, adjust selection
Pigtail (vertical)	∮	Delete character
Down-right	L	Insert space

Figure 2.11: Basic PenPoint gestures (from Carr [24])

Recently, prototypes of Go Corporation's PenPoint system have been demonstrated. Each consists of a notebook-sized computer with a flat display. The sole input device is a stylus, which is used for gestures and handwriting on the display itself. Figure 2.11 shows the basic gestures recognized; depending on the context, additional gestures and handwriting can also be recognized. As can be seen, PenPoint gestures may consist of multiple strokes. Although it seems that trainable recognition algorithms are used internally, at the present time the user cannot add any new gestures to the existing set. The hardware is able to sense pen *proximity* (how near the stylus is to the tablet), which is used to help detect the end of multi-stroke gestures and characters. PenPoint applications include a drawing program, a word processor, and a form-based data entry system.

Many of the above systems combine gesture and direct manipulation in the same interface. GEdit, for example, appears to treat mouse input as gestural when begun on the background window, but drags objects when mouse input begins on the object. Almost none combine gesture and direct manipulation in the same interaction. One exception, PenPoint, uses the dot gesture (touching the stylus to the tablet and then not moving until recognition has been indicated) to drag graphic objects. Button Box does something similar for dragging objects. Artkit [52] uses eager recognition, more or less crediting the idea to me.

2.3 Approaches for Gesture Classification

Fu [40] states that "the problem of pattern recognition usually denotes a discrimination or classification of a set of processes or events." Clearly gesture recognition, in which the input is considered to be an event to be classified as one of a particular set of gestures, is a problem of pattern recognition.

In this dissertation, known techniques of pattern recognition are applied to the problem of sensing gestures.

The general pattern recognition problem consists of two subproblems: pattern representation and decision making [40]. This implies that the architecture of the general pattern recognition consists of two main parts. First, the *representer* takes the raw pattern as input and outputs the internal representation of the pattern. Then, the *decider* takes as input the output of the representer, and outputs a classification (and/or a description) of the pattern.

This section reviews the pattern recognition work relevant to gesture recognition. In particular, the on-line recognition of handwritten characters is discussed whenever possible, since that is the closest solved problem to gesture recognition. For a good overview of handwriting systems in general, see Suen *et. al.* [125] or Tappert *et. al.* [129].

The review is divided into two parts: alternatives for representers and alternatives for deciders. Each alternative is briefly explained, usually by reference to an existing system which uses the approach. The advantages and disadvantages of the alternative are then discussed, particularly as they apply to single-path gesture recognition.

2.3.1 Alternatives for Representers

The representer module takes the raw data from the input device and transforms it into a form suitable for classification by the decider. In the case of single-path gestures, as with on-line handprint, the raw data consists of a sequence of points. The representer outputs *features* of the input pattern.

Representers may be grouped in terms of the kinds of features which they output. The major kinds of features are: templates, global transformations, zones, and geometric features. While a single representer may combine different kinds of features, representers are discussed here as if each only outputs one kind of feature. This will make clearer the differences between the kinds of features. Also, in practice most representers do depend largely on a single kind of feature.

Templates.

Templates are the simplest features to compute: they are simply the input data in its raw form. For a path, a template would simply consist of the sequence of points which make up the path. Recognition systems based on templates require the decider to do the difficult work; namely, matching the template of the input pattern to stored example templates for each class.

Templates have the obvious advantage that the features are simple to compute. One disadvantage is that the size of the feature data grows with the size of the input, making the features unsuitable as input to certain kinds of deciders. Also, template features are very sensitive to changes in the size, location, or orientation of the input, complicating classifiers which attempt to allow for variations of these within a given class. Examples of template systems are mentioned in the discussion of template matching below.

Global Transformations.

Some of the problems of template features are addressed by global transformations of the input data. The transformations are often mathematically defined so as to be invariant under *e.g.* rotation, translation, or scaling of the input data. For example, the Fourier transform will result in features invariant with respect to rotation of the input pattern [46]. Global transformations generally output a fixed number of features, often smaller than the input data.

A set of fixed features allows for a greater variety in the choice of deciders, and obviously the invariance properties allow for variations within a class. Unfortunately, there is no way to "turn off" these invariances in order to disallow intra-class variation. Also, the global transformations generally take as input a two-dimensional raster, making the technique awkward to use for path data (it would have to first be transformed into raster data). Furthermore, the computation of the transformation may be expensive, and the resulting features do not usually have a useful parametric interpretations (in the sense of Section 1.6.2), requiring a separate pass over the data to gather parametric information.

Zones.

Zoning is a simple way of deriving features from a path. Space is divided into a number of zones, and an input path is transformed into the sequence of zones which the path traverses [57]. One variation on this scheme incorporates the direction each zone is entered into the encoding [101]. As with templates, the number of features are not fixed; thus only certain deciders may be used. The major advantage of zoning schemes are their simplicity and efficiency.

If the recognition is to be size invariant, zoning schemes generally require the input to be normalized ahead of time. Making a zoning scheme rotationally invariant is more difficult. Such normalizations make it impossible to compute zones incrementally as the input data is received. Also, small changes to a pattern might cause zones to be missed entirely, resulting in misclassification. And again, the features do not usually hold any useful parametric information.

Geometric Features.

Geometric features are the most commonly used in handwriting recognition [125]. Some geometric features of a path (such as its total length, total angle, number of times it crosses itself, *etc.*) represent global properties of the path. Local properties, such as the sequence of basic strokes, may also be represented.

It is possible to use combinations of geometric features, each invariant under some transformations of the input pattern but not others. For example, the initial angle of a path may be a feature, and all other features might be invariant with respect to rotation of the input. In this fashion, classifiers may potentially be created which allow different variations on a per-class basis.

Geometric features often carry useful parametric information, *e.g.* the total path length, a geometric feature, is potentially a useful parameter. Also, geometric features can be fed to deciders which expect a fixed number of features (if only global geometric features are used), or to deciders which expect a sequence of features (if local features are used).

Geometric features tend to be more complex to compute than the other types of features listed. With care, however, the computation can be made efficient and incremental. For all these reasons, the current work concentrates on the use of global geometric features for the single-path gesture recognition in this dissertation (see Chapter 3).

2.3.2 Alternatives for Deciders

Given a vector or sequence of features output by a representer, it is the job of the decider to determine the class of the input pattern with those features. Seven general methods for deciders may be enumerated: template-matching, dictionary lookup, a discrimination net, statistical matching, linguistic matching, connectionism, and *ad hoc*. Some of the methods are suitable to only one kind of representer, while others are more generally applicable.

Template-matching.

A template-matching decider compares a given input template to one or more prototypical templates of each expected class. Typically, the decider is based on a function which measures the similarity (or dissimilarity) between pairs of templates. The input is classified as being a member of the same class as the prototype to which it is most similar. Usually there is a similarity threshold, below which the input will be rejected as belonging to none of the possible classes.

The similarity metric may be computed as a correlation function between the input and the prototype [69]. Dynamic programming techniques may be used to efficiently warp the input in order to better match up points in the input template to those in the prototype [133, 60, 9].

Template systems have the advantage that the prototypes are simply example templates, making the system easy to train. In order to accommodate large variations, for example in the orientation of a given gesture, a number of different prototypes of various orientation must be specified. Unfortunately, a large number of prototypes can make the use of template matching prohibitively expensive, since the input pattern must be compared to every template.

Lipscomb [80] presents a variation on template matching used for recognizing gestures. In his scheme, each training example is considered at different resolutions, giving rise to multiple templates per example. (The algorithm is thus similar to multiscale algorithms used in image processing [138].) Lipscomb has applied the multiscale technique to stroke data by using an angle filter, in which different resolutions correspond to different thresholds applied to the angles in the gestures. To represent a gesture at a given resolution, points are discarded so that the remaining angles are all below the threshold. To classify an input gesture, first its highest resolution representation is (conceptually) compared to each template (at every resolution). Successively lower resolutions of the input are tried in turn, until an exact match is found. Multiple matches are decided in favor of the template whose resolution is closest to the current resolution of the input.

Dictionary lookup.

When the input features are a sequence of tokens taken from a small alphabet, lookup techniques may be used. This is often how zoning features are classified [101]. The advantage is efficient recognition, since binary search (or similar algorithms) may be used to lookup patterns in the dictionary. Often some allowance is made for non-exact matches, since otherwise classification is sensitive to small changes in the input. Even with such allowances, dictionary systems are often brittle, due to the features employed (*e.g.* sequences of zones). Of course, a dictionary is initially created from example training input. It is also a simple matter to add new entries for rejected patterns; thus the dictionary system can adapt to a given user.

Discrimination nets.

A discrimination net (also called a decision tree) is basically a flowchart without loops. Each interior node contains a boolean condition on the features, and is connected to two other nodes (a "true" branch and a "false" branch). Each leaf node is labeled with a class name. A given feature set is classified by starting at the root note, evaluating each condition encountered and taking the appropriate branch, stopping and outputting the classification when a leaf node is reached.

Discrimination nets may be created by hand [25], or derived from example inputs [8]. They are more appropriate to classifying fixed-length feature vectors, rather than sequences of arbitrary length, and often result in accurate and efficient classifiers. However, discrimination nets trained by example tend to become unwieldy as the number of examples grows.

Statistical matching.

In statistical matching, the statistics of example feature vectors are used to derive classifiers. Typically, statistical matchers operate only on feature vectors, not sequences. Some typical statistics used are: average feature vector per class, per-class variances of the individual features, and per-class correlations within features. One method of statistical matching is to compute the distance of the input feature vector to the average feature vector of each class, choosing the class which is the closest. Another method uses the statistics to derive per-class discrimination functions over the features. Discrimination functions are like evaluation functions: each discrimination function is applied to the input feature vector, the class being determined by the largest result. Fisher [35] showed how to create discrimination functions which are simply linear combinations of the input features, and thus particularly efficient. Arakawa *et. al.*[3] used statistical classification of Fourier features for on-line handwriting recognition; Chapter 3 of the present work uses statistical classification of geometric features.

Some statistical classifiers, such as the Fisher classifier, make assumptions about the distributions of features within a class (such as multivariate normality); those tend to perform poorly when the assumptions are violated. Other classifiers [48] make no such assumptions, but instead attempt to estimate the form of the distribution from the training examples. Such classifiers tend to require many training examples before they function adequately. The former approach is adopted in the current work, with the feature set carefully chosen so as to not violate assumptions about the underlying distribution too drastically.

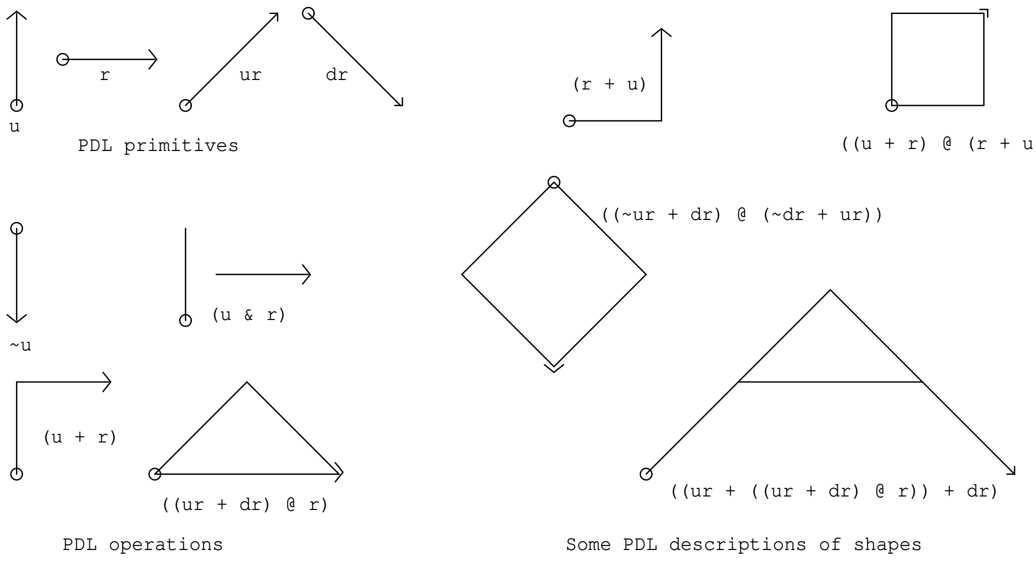


Figure 2.12: Shaw's Picture Description Language

PDL enables line drawings to be coded in string form, making it possible to apply textual parsing algorithms to the recognition of line drawings. The component line segments and combining operations are shown on the left; the right shows how the letter "A" can be described using these primitives.

Linguistic matching.

The linguistic approach attempts to apply automata and formal language theory to the problem of pattern recognition [37]. The representer outputs a sequence of tokens which is composed of a set of pattern primitives and composition operators representing the relation between the primitives. The decider has a grammar for each possible pattern class. It takes as input the sentence and attempts to parse it with respect to each pattern class grammar. Ideally, exactly one of the parses is successful and the pattern is classified thus. A useful side effect of the syntax analysis is the parse tree (or other parse trace) which reveals the internal structure of the pattern.

Linguistic recognizers may be classified based on the form of the representer output. If the output is a string then standard language recognition technology, such as regular expressions and context-free grammars, may be used to parse the input. An error-correcting parser may be used in order to robustly deal with errors in the input. Alternatively, the output of the representer may be a tree or graph, in which case the decider could use graph matching algorithms to do the parse.

The token sequence could come from a zoning representer, a representer based on local geometric properties, or from the output of a lower-level classifier. The latter is a hybrid approach–where, for example, statistical recognition is used to classify paths (or path segments), and linguistic recognition is used to classify based on the relationships between paths. This approach is similar to that taken by Fu in a number of applications [40, 39, 38].

Shaw's picture description language (PDL, see figure 2.12) has been used successfully to describe and classify line drawings [116, 40]. In another system, Stallings [120, 37] uses the composition 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.

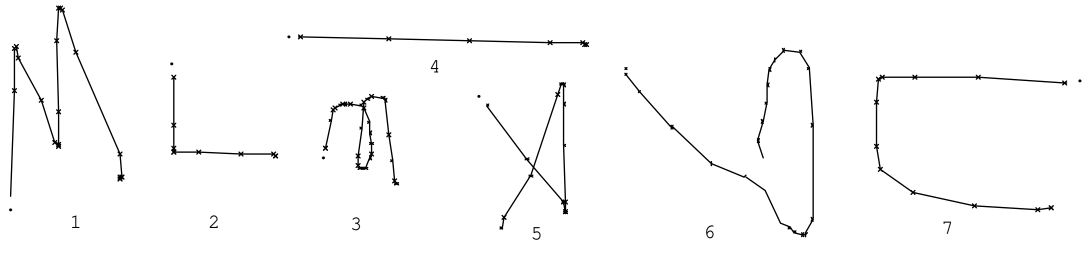


Figure 3.1: Some example gestures
The period indicates the start of the gesture. The actual mouse points that make up the gestures are indicated as well.

[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:

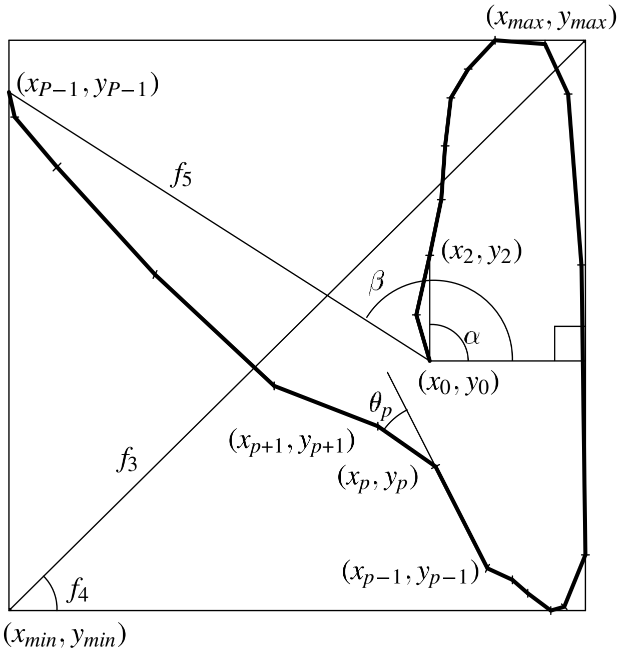


Figure 3.2: Feature calculation

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:

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

Total gesture length:

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

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

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 **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 $0 \leq i \leq F$, 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:

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

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)

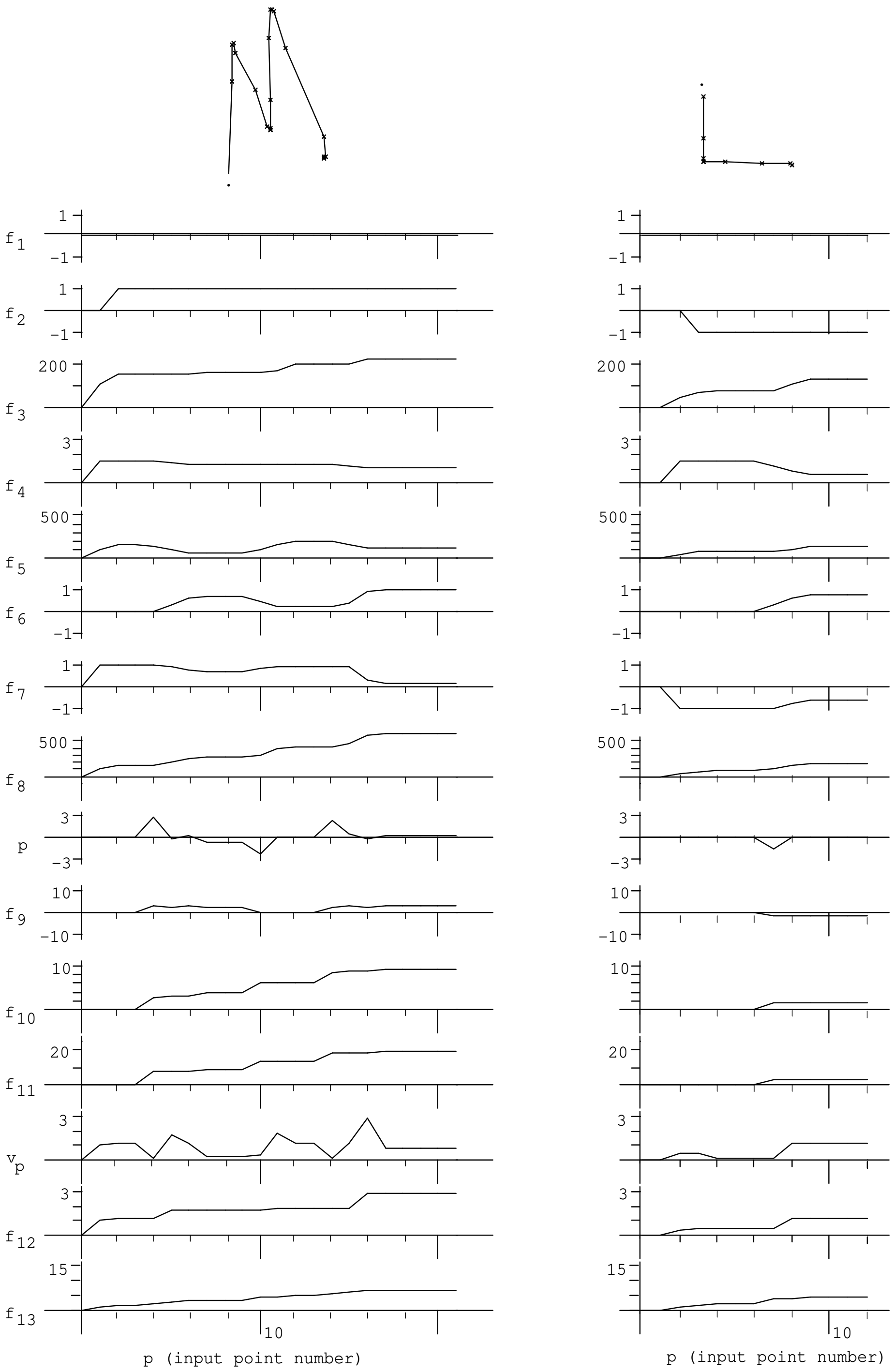


Figure 3.3: Feature vector computation

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

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

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

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

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, $0 \le e < E^{\hat c}$, 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.

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

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 **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 **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 **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.

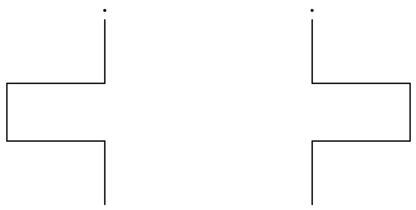


Figure 3.4: Two different gestures with identical feature vectors

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

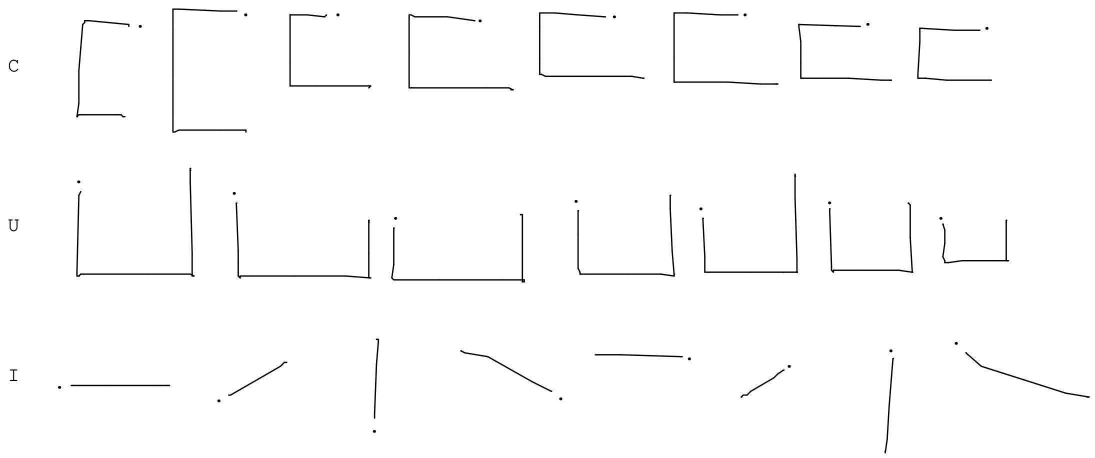


Figure 3.5: A potentially troublesome gesture set

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 $s^{\mathbf{C}}_{ij}$, $s^{\mathbf{U}}_{ij}$, and $s^{\mathbf{I}}_{ij}$. In particular, consider the variance of the feature f_1, which, for each class c, is proportional to $s^{\hat{c}}_{11}$. Since the initial angle is almost the same for each example C gesture, $s^{\mathbf{C}}_{11}$ will be close to zero. Similarly, $s^{\mathbf{U}}_{11}$ will also be close to zero. However, since the examples of class **I** have different orientations, $s^{\mathbf{I}}_{11}$ will be significantly non-zero.

Unfortunately, the information on the variance of f_1 is lost when the per-class covariance matrix estimates $s^{\hat{c}}_{ij}$ 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

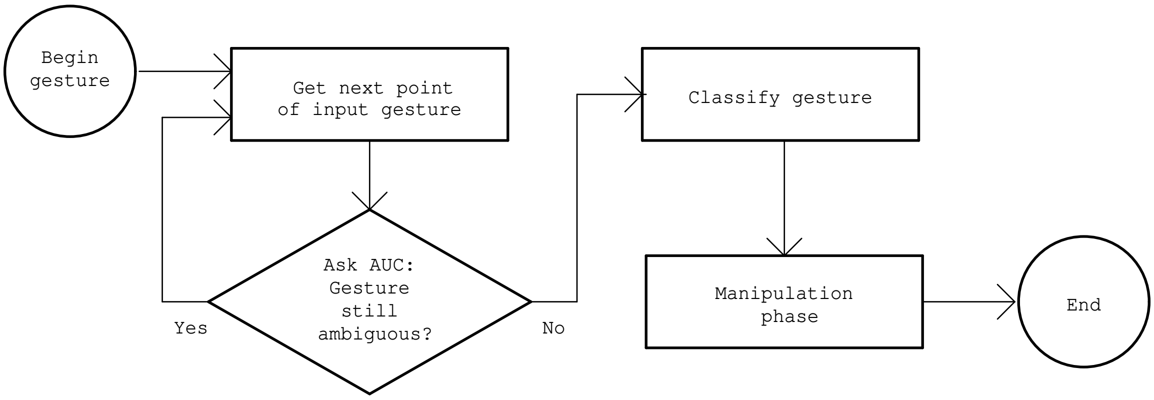


Figure 4.1: Eager recognition overview

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.

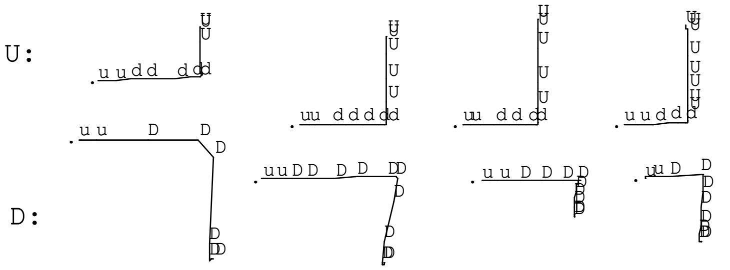


Figure 4.2: Incomplete and complete subgestures of U and D

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

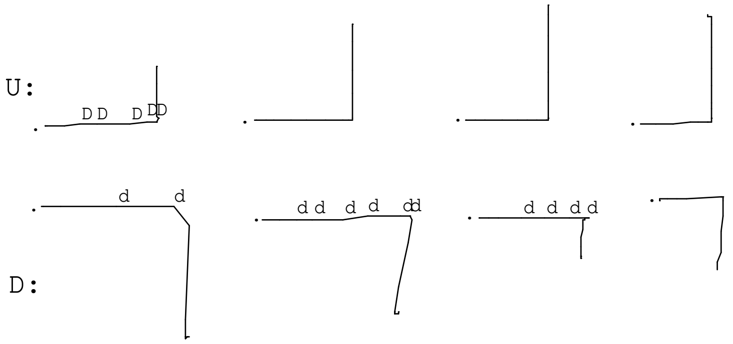


Figure 4.3: A first attempt at determining the ambiguity of subgestures

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.

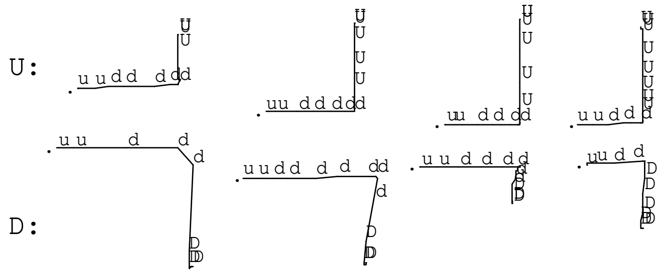


Figure 4.6: Accidentally complete subgestures have been moved

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 $\exists\ c$ | 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.

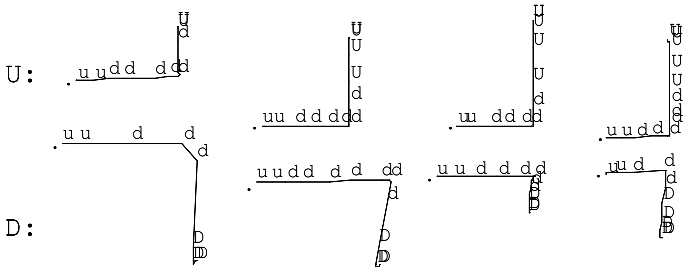


Figure 4.9: Classification of subgestures of U and D

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

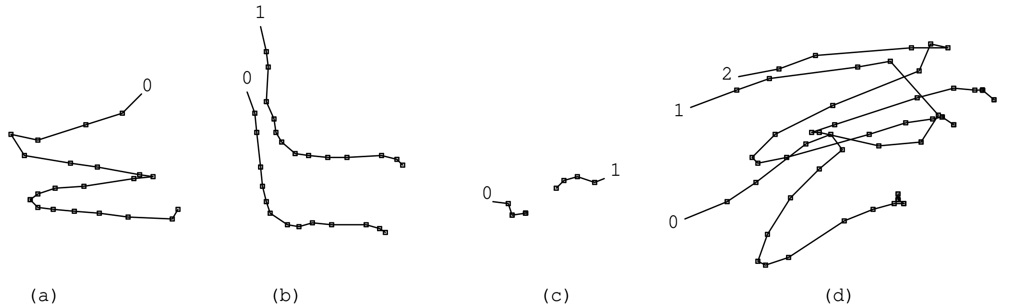


Figure 5.1: Some multi-path gestures

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

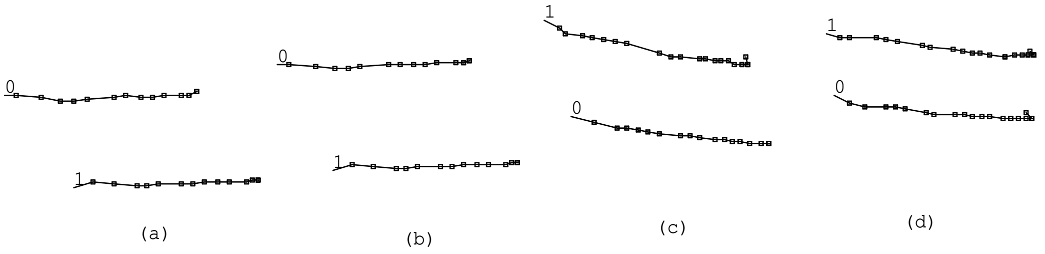


Figure 5.2: Inconsistencies in path sorting

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.

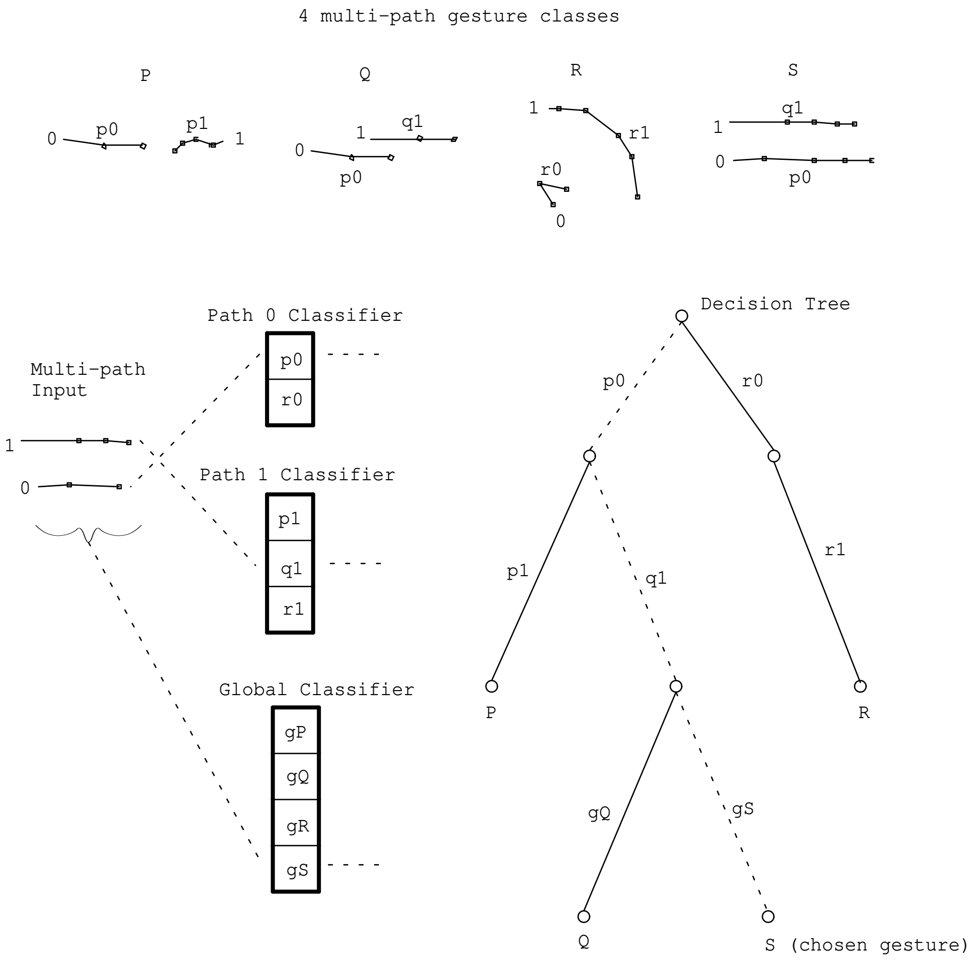


Figure 5.3: Classifying multi-path gestures

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 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

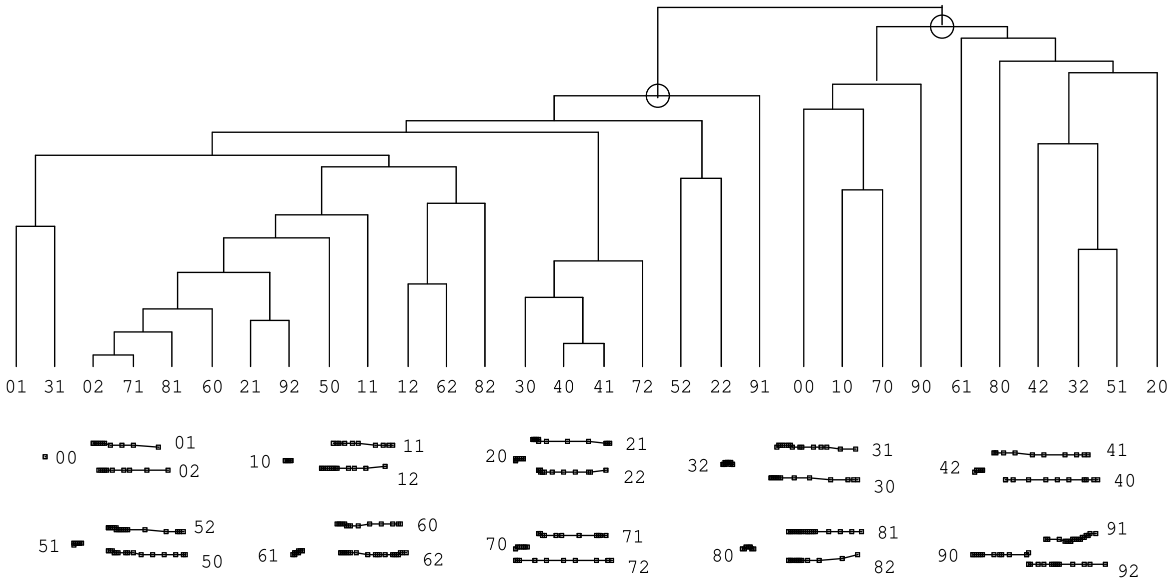


Figure 5.4: Path Clusters

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.

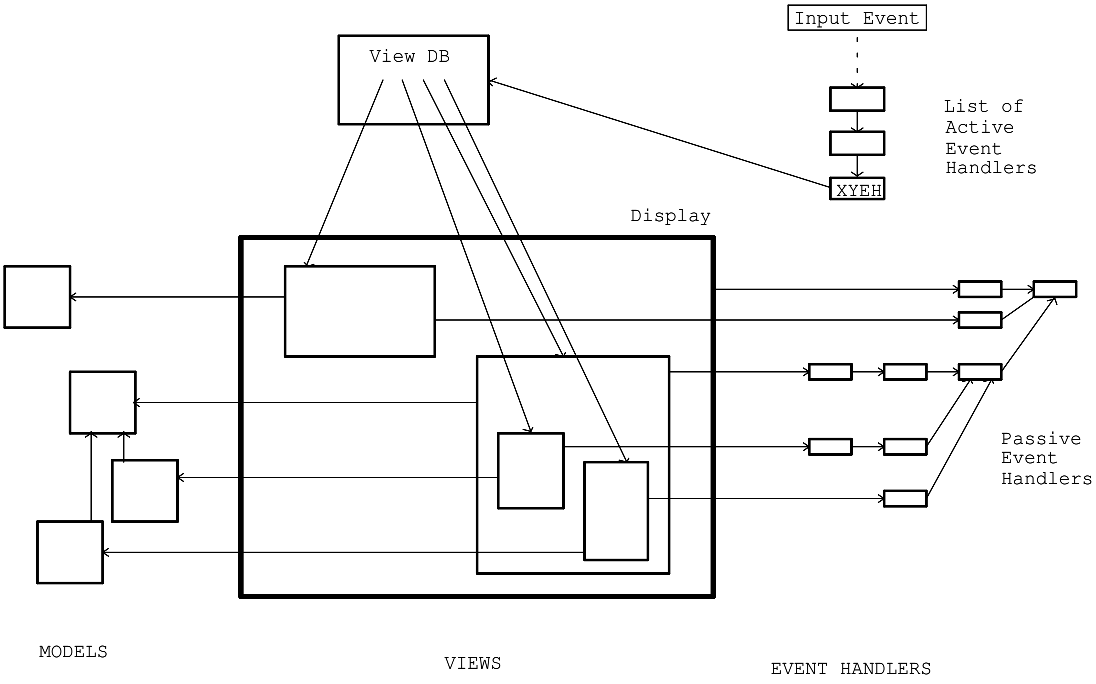


Figure 6.1: GRANDMA's Architecture

*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.

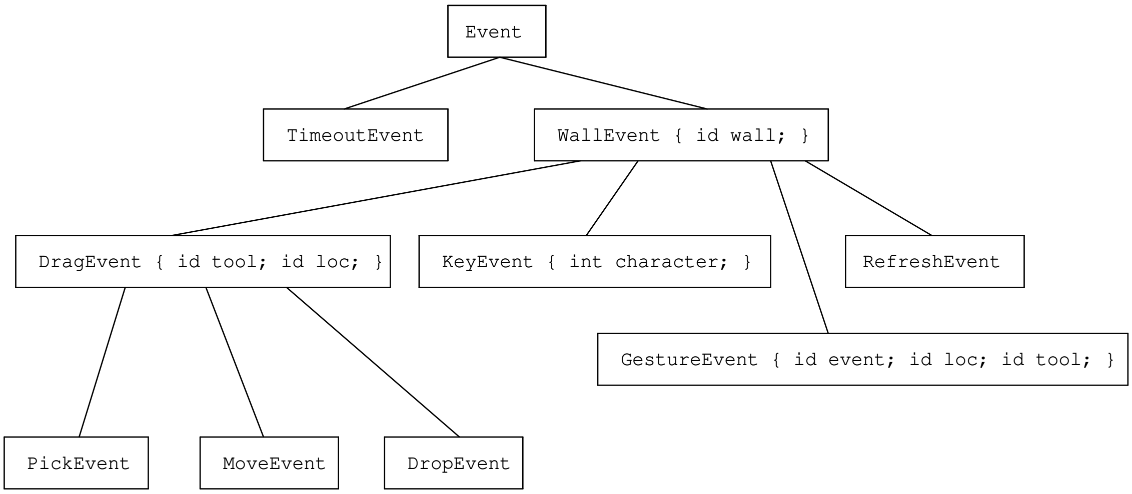


Figure 6.2: The `Event` Hierarchy

`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];
```

```
        return self;
    }

    - activeHandler:e {
        int x, y;
        if(relative) {
            x = [[e loc] x], y = [[e loc] y];
            [view perform:whenmoved
                    with:(x - savedx) with:(y - savedy)];
            savedx = x, savedy = y;
        }
        else {
            x = [[e loc] x] + savedx, y = [[e loc] y] + savedy;
            [view perform:whenmoved with:x with:y];
        }
        if(genevents)
             [wall raise:[[e class] tool:view loc:newloc
                  wall:wall instigator:self time:[e time]]];
        return self;
    }
    - activeTerminator:e {
        if(genevents)
             [wall raise:[[e class] tool:view loc:[e loc]
                  wall:wall instigator:self time:[e time]]];
        if(whendone) [view perform:whendone];
        return self;
    }
```

The `passive` factory method creates a `DragHandler` with instance variables set to the default parameters. Those parameters can be changed with the `startp:`, `handlep`, `stopp:`, `copyviewON`, `genEventsOFF`, `relativeON`, `whendone:`, and `whenmoved:` messages. (Please refer to the comments in the above code for a description of the function of these parameters.)

For example, a `DragHandler` might be associated with class `LabelView` as follows:

```
    = LabelView ...
    + initialize {
        [self sethandler:
          [[[DragHandler passive]
             startp:[[[EventExpr new]
                    eventkind:PickEvent] toolkind:MouseTool]]
            genEventsOFF]];
    }
```

Any `LabelView` can thus be dragged around with the mouse by clicking directly on it (since the start predicate was changed to `PickEvent`). A `LabelView` will not generate events as it is dragged since `genEventsOFF` was sent to the handler; thus `LabelViews` in general would not be used as tools or items that can be deposited in buckets. Of course, subclasses and instances of `LabelView` may have their own passive event handlers to override this behavior.

When a passive `DragHandler` gets an event that satisfies its start predicate, the `passiveHandler:` method is invoked. For a `DragHandler`, some location information is saved, the view is copied if need be, and the view is flashed (rapidly highlighted and unhighlighted) as user feedback.

Any subsequent event that satisfies the stop predicate will cause the `activeTerminator:` method to be invoked. Other events that satisfy the handle predicate will cause `activeHandler:` to be invoked. In `DragHandler`, `activeHandler:` first moves the view (typically by sending it the `at::` message with the new coordinates as arguments) then possibly raises a new event with the view playing the role of tool in the event. If the view is indeed a tool, raising this event might result in the `GenericToolOnView` handler being activated, as previously discussed.

Note that the event to be raised is created by first determining the class object (factory) of the passed event (given the default predicates, in this case the class will either be `MoveEvent` or `DropEvent`), and then asking the class to create a new event, which will thus be the same class as the passed event. Most of the new event attributes are copied verbatim from the old attributes; only the `tool` and `instigator` are changed. A more sophisticated `DragHandler` might also change the event location to be at some designated hot spot of the view being moved, rather than simply use the location of the passed event. For simplicity, this was not shown here.

The `activeTerminator:` method also possibly raises a new event, and possibly sends the view the message stored in the `whendone` variable. As an example, `whendone` might be set to `@selector(delete)` when `copyview` is set. When the mouse button is pressed over a view, a copy of the view is created. Moving the mouse drags the copy, and when the mouse button is finally released, the copy is deleted.

Creating a new drag handler and associating it with a view or view class is all that is required to make that view "draggable" (since every view inherits the `at::` message). As shown in the next chapter, GRANDMA has a facility for creating handlers and making the association at runtime.

6.8 Summary of GRANDMA

This concludes the detailed discussion of GRANDMA. As the discussion has concentrated on the features which distinguish GRANDMA from other MVC-like systems, much of the system has not been discussed. It should be mentioned that the facilities described are sufficiently powerful to build a number of useful view and controller classes. In particular, standard items such as popup views, menus, sliders, buttons, switches, text fields, and list views have all been implemented. Chapter 8 shows how some of these are used in applications.

GRANDMA's innovations come from its input model. Here is a summary of the main points of the input architecture:

1. Input events are full-blown objects. The `Event` hierarchy imposes structure on events without imposing device dependencies.

2. Raised events are propagated down an active event list.
3. Otherwise unhandled events with screen locations are automatically routed to views at those locations.
4. A view object may have any number of passive event handlers associated with itself, its class, or its superclass, *etc*. Events are automatically routed to the appropriate handler.
5. A passive event handler may be shared by many views, and can activate a copy of itself to deal with events aimed at any particular view.
6. Event handlers have predicates that describe the events to which they respond.
7. The generic event handler simplifies the creation of dynamically parameterizable event handlers.

Because of the input architecture, GRANDMA has a number of novel features. They are listed here, and compared to other systems when appropriate.

GRANDMA can support many different input devices simultaneously. Due to item 1 above, GRANDMA can support many different input devices in addition to just a single keyboard and mouse. Each device needs to integrate the set of event classes which it raises into GRANDMA's `Event` hierarchy. Much flexibility is possible; for example, a Sensor Frame device might raise a single `SensorFrameEvent` describing the current set of fingers in the plane of the frame, or separate `DragEvents` for each finger, the tool in this case being a `SensorFrameFingerTool`. Because of item 6, it is possible to write event handlers for any new device which comes along.

By contrast, most of the existing user interface toolkits have hard-wired limitations in the kinds of devices they support. For example, most systems (the NeXT AppKit [102], the Macintosh Toolbox [1], the X library [41]) have a fixed structure which describes input events, and cannot be easily altered. Some systems go so far as to advocate building device dependencies into the views themselves; for example, Hypertalk event handlers [45] are labeled with event descriptors such as `mouseUp` and Cox's system [28] has views that respond to messages like `rightButtonDown`. Similarly, systems with a single controller per view [70] cannot deal with input events from different devices. On the other hand, GWUIMS [118] seems to have a general object classification scheme for describing input events.

GRANDMA supports the emulation of one device with another. In GRANDMA, to get the most out of each device it is necessary to have event handlers which can respond to events from that device associated with every view that needs them. If those event handlers are not available, it is still possible to write an event handler that emulates one device by another. For example, an active handler might catch all `SensorFrameEvents` and raise `DragEvents` whose tool is a `Mousetool` in response. The rest of the program cannot tell that it is not getting real mouse data; it responds as if it is getting actual mouse input.

GRANDMA can handle multiple input threads simultaneously. Because passive handlers activate copies of themselves, even views that refer to the same handler can get input simultaneously. The input events are simply propagated down the active event handler list, and each active handler only handles the events it expects. In GRANDMA, a system that had two mice [19] would simply have two `MouseTool` objects, which could easily interleave events. Normally, a passive handler would only activate itself to receive input from a single tool (mouse, in this case), allowing input from the two mice to be handled independently (even when directed at the same view). It would also be possible to write an event handler that explicitly dealt with events from both mice, if that was desired.

Event-based systems, such as Sassafras [54] and Squeak [23], are also able to deal with multi-threaded dialogues. Indeed, it is GRANDMA's similarity to those systems which gives it a similar power. This is in contrast to systems such as Smalltalk [70] where, once a controller is activated it loops polling for events, and thus does not allow other controllers to receive events until it is deactivated.

GRANDMA provides virtual tools. Given the general structure of input events, there is no requirement for them only to be generated by the window manager. Event handlers can themselves raise other events. Many events have `tools` associated with them; for example, mouse events are associated with `MouseTools`. The tools may themselves be views or other objects. By responding to messages such as `action`, a tool makes known its effect on objects which it is dragged over. The `GenericToolOnView` handler, which is associated with the `View` class (and thus every view in the system) will handle the interaction when a tool which has a certain action is dragged over an object which accepts that action. The tools are virtual, in the sense that they do not correspond directly to any input hardware, and they may send arbitrary messages to views with which they interact.

GRANDMA supports semantic feedback. Handlers like `GenericToolOnView` can test at run-time if an arbitrary tool is able to operate upon an arbitrary view which it is dragged over, and if so highlight the view and/or tool. No special code is required in either the tool or the view to make this work. A tool and the views upon which it operates often make no reference to each other. The sole connection between the two is that one is able to send a message that the other is able to receive.

Of course, the default behavior may be easily overridden. A tool can make arbitrary enquiries into the view and its model in order to decide if it does indeed wish to operate upon the view.

Event handling in GRANDMA is both general and efficient. The generality comes from the event dispatch, where, if no other active handler handles an event, the `XYEventHandler` can query the views at the location of the event. The views consult their own list of passive event handlers, which potentially may handle many different kinds of events. There is space efficiency in that a single passive event handler may be shared by many views, eliminating the overhead of a controller object per view. There is time efficiency, in that once a passive handler handles an event, it may activate itself, after which it receives events immediately, without going through the elaborate dispatch of the `XYEventHandler`.

Artkit [52] has a priority list of dispatch agents that is similar to GRANDMA's active event handler list. Such agents receive low-level events (*e.g.* from the window manager), and attempt to translate them into higher level events to be received by interactor objects (which seem to be views). Interactor agents register the high-level events in which they are interested.

Artkit's architecture is so similar to GRANDMA's that it is difficult to precisely characterize the difference. The high-level events in Artkit play a role similar to both that of messages that a view may receive and events that a view's passive event handlers expect. In GRANDMA, the registering is implicit; because of the Objective-C runtime implementation, the messages understood by a given object need not be specified explicitly or limited to a small set. Instead, one object may ask another if it recognizes a given message before sending it.

Because of the translation from low-level to high-level events, it does not seem that Artkit can, for example, emulate one device with another. In particular, it does not seem possible to translate low-level events from one device into those of another. GRANDMA does not make a distinction between low-level and high-level events. Instead, GRANDMA distinguishes between events and messages; events are propagated down the active event handler list; when accepted by an event handler, the handler may raise new events and/or send messages to views or their models.

GRANDMA supports gestures. GRANDMA's general input mechanism had the major design goal of being able to support gestural input. As will be seen in the next chapter, the gestures are recognized by `GestureEventHandlers`; these collect mouse (or other) events, determine a set of gestures which they recognize depending on the views at the initial point of the gesture, and once recognized, can translate the gesture into messages to models or views, or into new events.

Artkit also handles gestural input, and, somewhat like GRANDMA, has gesture event handlers which capture low-level events and produce high-level events. The designers claim that Artkit, because of its object-oriented structure, can use a number of different gesture recognition algorithms, and thus tailor the recognizer to the application, or even bits of the application. The same is true for GRANDMA, of course, though the intention was that the algorithms described in the first half of the thesis are of sufficient generality and accuracy that other recognition algorithms are not typically required. Artkit's claim that *many* recognizers can be used seems like an excuse not to provide *any*. One of the driving forces behind the present work is the belief that gesture recognizers are sufficiently difficult to build that requiring application programmers to hand code such recognizers for each gesture set is a major reason that hardly any applications use gestures. Thus, it is necessary to provide a general, trainable recognizer in order for gesture-based interfaces to be explored. How such a recognizer is integrated into an object-oriented toolkit is the subject of the next chapter.

Of course, GRANDMA does have its disadvantages. Like other MVC systems, GRANDMA provides a multitude of classes, and the programmer needs to be familiar with most of them before he can decide how to best implement his particular task. The elaborate input architecture exacerbates the problem: a large number of possible combinations of views, event handlers, and tools must be considered by the programmer of a new interaction technique. Also, GRANDMA does nothing toward solving a common problem faced when using any MVC system: deciding what functionality goes into a view and what goes into a model. Another problem is that even though the protocol between event handlers and views is meant to be very general (the event handlers are initialized with arbitrary message selectors to use when communicating with the view), in practice the views are written with the intention that they will communicate with particular event handlers, so that it is not really right to claim that specifics of input have truly been factored out of views.

Chapter 7

Gesture Recognizers in GRANDMA

This chapter discusses how gesture recognition may be incorporated into systems for building direct manipulation interfaces. In particular, the design and implementation of gesture handlers in GRANDMA is shown. Even though the emphasis is on the GRANDMA system, the methods are intended to be generally applicable to any object-oriented user interface construction tool.

7.1 A Note on Terms

Before beginning the discussion, some explanation is needed to help avoid confusion between terms. As discussed in Section 6.4, it is important not to confuse the view hierarchy, which is the tree determined by the subview relationship, and the view class hierarchy, which is the tree determined by the subclass relationship. In GRANDMA, the view hierarchy has a `WallView` object (corresponding to an X window) at its root, while the view class hierarchy has the class `View` at its root.

Another potentially ambiguous term is "class." Usually, the term is used in the object-oriented sense, and refers to the type (loosely speaking) of the object. However, the term "gesture class" refers to the result of the gesture recognition process. In other words, a gesture recognizer (also known as a gesture classifier) discriminates between gesture classes. For example, consider a handwriting recognizer able to discriminate between the written digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. In this example, each digit represents a class; presumably, the recognizer was trained using a number of examples of each class.

To make matters more confusing, in GRANDMA there is a class (in the object-oriented sense) named `Gesture`; an object of this class represents a particular gesture instance, *i.e.* the list of points which make up a single gesture. There is also a class named `GestureClass`; objects of this class refer to individual gesture classes; for example, a digit recognizer would reference 10 different `GestureClass` objects.

Sometimes the term "gesture" is used to refer to an entire gesture class; other times it refers to a single instance of gesture. For example, when it is said a recognizer discriminates between a set of gestures, what is meant is that the recognizer discriminates between a set of gesture classes. Conversely, "the user enters a gesture" refers to a particular instance. In all cases which follow, the intent should be obvious from the context.

7.2 Gestures in MVC systems

As discussed in Chapters 2 and 6, object-oriented user interface systems typically consist of models (application objects), views (responsible for displaying the state of models on the screen), and controllers (responsible for responding to input by sending messages to views and models). Typical Model/View/Controller systems, such as that in Smalltalk[70], have a view object and controller object for each model object to displayed on the screen.

This section describes how gestures are integrated into GRANDMA, providing an example of how gestures might be integrated into other MVC-based systems.

7.2.1 Gestures and the View Class Hierarchy

Central to all the variations of object-oriented user interface tools is the `View` class. In all such systems, view objects handle the display of models. Since the notion of views is central to all object-oriented user interface tools, views provide a focal point for adding gestures to such tools.

Simply stated, the idea for integrating gestures into direct manipulation interfaces is this: *each view responds to a particular set of gestures*. Intuitively, it seems obvious that, for example, a switch should be controlled by a different set of gestures than a dial. The ability to simply and easily specify a set of gestures and their associated semantics, and to easily associate the set of gestures with particular views, was the primary design goal in adding gestures to GRANDMA.

Of course, it is unlikely that every view will respond to a distinct set of gestures. In general, the user will expect similar views to respond to similar sets of gestures. Fortunately, object-oriented user interfaces already have the concept of similarity built into the view class hierarchy. In particular, it usually makes the most sense for all view objects of the same class to respond to the same set of gestures. Similarly, it is intuitively appealing for a view subclass to respond to all the gestures of its parent class, while possibly responding to some new gestures specific to the subclass.

The above intuitions essentially apply the notions of class identity and inheritance[121] (in the object-oriented sense) to gestures. It is seen that gestures are analogous to messages. All objects of a given class respond to the same set of messages, just as they respond to the same set of gestures. An object in a subclass inherits methods from its superclass; similarly such an object should respond to all gestures to which its superclass responds. Continuing the analogy, a subclass may override existing methods or add new methods not understood by its superclass; similarly, a subclass may override (the interpretation of) existing gestures, or recognize additional gestures. Some object-oriented languages allow a subclass to disable certain messages understood by its superclass (though it is not common), and analogously, it is possible that a subclass may wish to disable a gesture class recognized by its superclass.

Given the close parallel between gesture classes and messages, one possible way to implement gesture semantics would be for each kind of view to implement a method for each gesture class it expects. Classifying an input gesture would result in its class's particular message to be sent to the view, which implements it as it sees fit. A subclass inherits the methods of its superclass, and may override some of these methods. Thus, in this scheme a subclass understands all the gestures that its superclass understands, but may change the interpretation of some of these gestures.

This close association of gestures and messages was not done in GRANDMA since it was felt to be too constricting. Since in Objective C all methods have to be specified at compile time, adding new gesture classes would require program recompilations. Since it is quite easy to add new gesture classes at runtime, it would be unfortunate if such additions required recompilations. One of the goals of GRANDMA is to permit the rapid exploration of different gestures sets and their semantics; forcing recompilations would make the whole system much more tedious to use for experimentation.

Instead, the solution adopted was to have a small interpreter built into GRANDMA. A piece of interpreted code is associated with each gesture class; this code is executed when the gesture is recognized. Since the code is interpreted, it is straightforward to add new code at the time a new class is specified, as well as to modify existing code, all at runtime. While at first glance building an interpreter into GRANDMA seems quite difficult and expensive, Objective C makes the task simple, as explained in Section 7.7.3.

7.2.2 Gestures and the View Tree

Consider a number of views being displayed in a window. In GRANDMA, as in many other systems, pressing a mouse button while pointing at a particular view (usually) directs input at that view. In other words, the view that gets input is usually determined at the time of the initial button press. Due to the view tree, views may overlap on the screen, and thus the initial mouse location may point at a number of views simultaneously. Typically the views are queried in order, from foremost to background, to determine which one gets to handle the input.

A similar approach may be taken for gestures. The first point of the gesture determines the views at which the gesture might be directed. However, determining which of the overlapping views is the target of the gesture is usually impossible when just the first point has been seen. What is usually desirable is that the entire gesture be collected before the determination is made.

Consider a simplification of GDP. The wall view, behind all other views, has a set of gestures for creating graphic objects. A straight stroke “-” gesture creates a line, and an “L” gesture creates a rectangle. The graphic object views respond to a different set of gestures; an “X” deletes a graphic object, while a “C” copies a graphic object. When a gesture is made over, say, an existing rectangle, it is not immediately clear whether it is directed at the rectangle itself or at the background. It depends on the gesture: an “X” is directed at the existing rectangle, an “L” at the wall view. Clearly the determination cannot be made when just the first point of the gesture has been seen.

Actually, this is not quite true. It is conceivable that the graphic object views could handle gestures themselves that normally would be directed at the wall view. There is some practical value in this. For example, creating a new graphic object over an existing one might include lining up the vertices of the two objects. However, while it is nice to have the option, in general it seems a bad idea to force each view to explicitly handle any gestures that might be directed at any views it covers.

Chapter 3 addressed the problem of classifying a gesture as one of a given set of gesture classes. It is seen here that this set of gestures is not necessarily the set associated with a single view, but instead is the union of gesture sets recognized by all views under the initial point. There are some technical difficulties involved in doing this. It would in general be quite inefficient to have to construct a classifier for every possible union of view gestures sets. However, it is necessary that classifiers be constructed for the unions which do occur. The current implementation dynamically constructs a classifier for a given set of gesture classes the first time the set appears; this classifier is then cached for future use.

It is possible that more than one view under the initial point responds to a given gesture class. In these cases, preference is given to the topmost view. The result is a kind of dynamic scoping. Similarly, the way a subclass can override a gesture class recognized by its superclass may be considered a kind of static scoping.

7.3 The GRANDMA Gesture Subsystem

In GRANDMA, gestural input is handled by objects of class `GestureEventHandler`. Class `GestureEventHandler`, a subclass of `GenericEventHandler`, is easily the most complex event handler in the GRANDMA system. In addition to the five hundred lines of code which directly implement its various methods, `GestureEventHandler` is the sole user of many other GRANDMA subsystems. These include the gesture classification subsystem, the interface which allows the user to modify gesture handlers (by, for example, adding new gesture classes) at runtime, the Objective C interpreter used for gesture semantics and its user interface, as well as some classes (*e.g.* `GestureEvent`, `TimeoutEvent`) used solely by the gesture handler.

Before getting into details, an overview of GRANDMA's various gesture-related components is presented. Figure 7.1 shows the relations between objects and classes associated with gestures in GRANDMA. The main focus is the `GestureEventHandler`. Like all event handlers, when activated it has a `view` object, which itself has a model and a wall view.[1] A `GestureEventHandler` uses the wall view to activate itself, raise `GestureEvents`, set up timeouts and their handlers, and draw the gesture as it is being made.

Associated with a gesture event handler is a set of `SemClass` objects. A `SemClass` object groups together a gesture class object (class `GestureClass`) with three expressions (subclasses of `Expr`). The `GestureClass` objects represent the particular gesture classes recognized directly by this event handler. The three expressions comprise the semantics associated with the gesture class by this event handler. The first expression is evaluated when the gesture is recognized, the second on each subsequent input event handled by the gesture handler after recognition (the manipulation phase, see Section 1.1), and the third when the manipulation phase ends.

Associated with each `GestureClass` object is a set of `Gesture` objects. These are the examples of gestures in the class and are used in the training of classifiers that recognize the class. A `GestureClass` object contains aggregate information about its examples, such as the estimated mean vector and covariance matrix of the examples' features, both of which are used in the construction of classifiers.

When a `GestureEventHandler` determines which gesture classes it must discriminate among (according to the rules described in the previous section), it asks the `Classifier` class

---

[1] Recall that a wall view is the root of the view tree and represents a window on the screen.

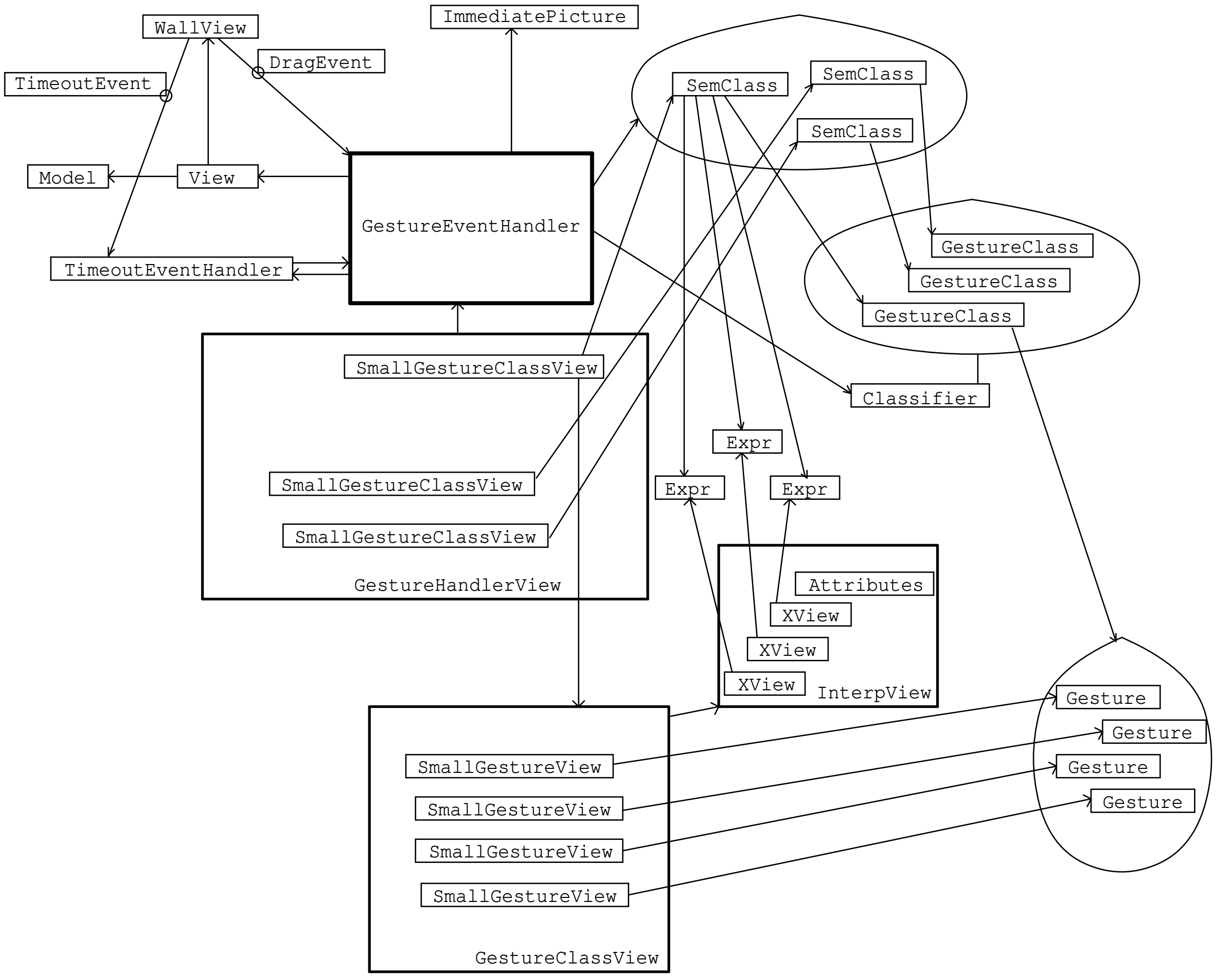


Figure 7.1: GRANDMA's gesture subsystem

A passive `GestureEventHandler` *is associated with a view or view class that expects gestural input. Once gestural input begins, the handler is activated and refers directly to the view at which the gesture was directed, as shown in the figure. The* `ImmediatePicture` *object is used for the inking of the gesture. The handler uses a timeout mechanism to indicate when to change from the collection to manipulation state. A* `SemClass` *object exists for each gesture expected by the handler, with each* `SemClass` *object associating a gesture class with its semantics. Each* `GestureClass` *object is described by a set of example* `Gestures`*, and there are view objects for each of the examples (*`SmallGestureView`*) as well as for the class as a whole (*`GestureClassView`*,* `SmallGestureClassView`*) which allow these to be displayed and edited. The gesture semantics are represented by* `Expr` *objects, and may be edited in the* `InterpView` *window.*

for a classifier object capable of doing this discrimination. Normally such a classifier will already exist; in this case, the existing classifier is simply returned. It is possible that one of the gesture classes in the set has changed; in this case the existing classifier has to be retrained (*i.e.* recalculated). Occasionally, this set of gesture classes has never been seen before; in this case a new classifier is created for this set, returned, and cached for future use.

The components related to the gesture event handler through `GestureHandlerView` are all concerned with enabling the user to see and alter various facets of the event handler. The predicates for starting, handling, and stopping the collection of gesture input may be altered by the user. In addition, gesture classes may be created, deleted, or copied from other gesture event handlers. The examples of a given class may be examined, and individual examples may be added or deleted. Finally, the semantics associated with a given gesture class may be altered through the interface to the Objective C interpreter.

7.4 Gesture Event Handlers

The details of the class `GestureEventHandler` are now described, beginning with its instance variables.

```
static BOOL masterSwitch = YES;
= GestureEventHandler :  GenericEventHandler {
    STR    name;
    id     gesture;
    id     picture;
    id     classes;
    id     env;
    int    timeval;
    id     timeouteh;
    short  lastx, lasty;
    id     sclass;
    struct gassoc { id sclass, view; } *gassoc;
    int    ngassocs;
    id     class_set;
    BOOL   manip_phase;
    BOOL   classify;
    BOOL   ignoring;
    id     mousetool;
}
```

The `masterSwitch`, settable via the `masterSwitch:` factory method, enables and disables all gesture handlers in an application. This provides a simple method for an application to provide two interfaces, one gesture-based, the other not. Every gesture handler will ignore all events when `masterSwitch` is `NO`. It will be as if the application had no gesture event handlers. Typically, the remaining event handlers would provide a more traditional click and drag interface to the application.

A particular handler can be turned off by setting its `ignoring` instance variable via the `ignore:` message. GRANDMA can thus be used to compare, say, two completely different gestural interfaces to a given application, switching between them at runtime by turning the appropriate handlers on and off.

The instance variable `name` is the name of the gesture handler. A handler is named so that it can be saved, along with its gesture classes, their semantics and examples, in a file. This is obviously necessary to avoid having the user enter examples of each gesture class each time an application is started. The name is passed to the `passive:` method which creates a passive gesture handler:

```
= GestureEventHandler ...
+ passive:(STR)_name {
    FILE *f;

    self = [super passive];
    classes = [OrdCltn new];
    [self instantiateON];
    [self startp:[[[EventExpr new] eventkind:PickEvent]
                              toolkind:MouseTool]];
    [self handlep:[[EventExpr new] eventkind:DragEvent]];
    [self stopp:[[EventExpr new] eventkind:DropEvent]];
    [self name:_name];
    timeval = DefaultTimeval;
    classify = YES;
    if((f = [self openfile:"r"]) != NULL) [self read:f];
    return self;
}
```

The typical gesture handler activates itself in response to mouse `PickEvents`, handles all subsequent mouse events, and deactivates itself when the mouse button is released. Of course, being a kind of generic event handler, this default behavior can be easily overridden, as was done to the `DragEventHandler` discussed in Section 6.7.9.

By default, the gesture event handler plans to classify any gestures directed at it (`classify = YES`). This is changed in those gesture event handlers that collect gestures for training other gesture event handlers.

The default `timeval` is 200, meaning 200 milliseconds, or two tenths of a second. This is the duration that mouse input must cease (the mouse must remain still) for the end of a gesture to be recognized. The user may change the default, thus affecting every gesture event handler. The timeout interval may also be changed on a per handler basis, a feature useful mainly for comparing the feel of different intervals.

When an event satisfies the handler's start predicate, the handler activates itself, and its `passiveHandler` is called.

```
= GestureEventHandler ...
- passiveHandler:e {
    gesture = [[Gesture new] newevent:e];
```

```
        picture = [ImmediatePicture create];
        [view _hang:picture at:0:0];
        lastx = [[e loc] x]; lasty = [[e loc] y];
        env = [Env new];
        [env str:"gesture" value:gesture];
        [env str:"startEvent" value:[e copy]];
        [env str:"currentEvent" value:[e copy]];
        [env str:"handler" value:self];
        manip_phase = NO;
        timeouteh = [[TimeoutEventHandler active]
               rec:self sel:@selector(timedout:)];
        [wall activate:timeouteh];
        [wall timeout:timeval];

        if(classify) {
            class_set = [Set new];
            gassoc = (struct gassoc *)
                   malloc(MAXCLASSES * sizeof(struct gassoc));
            ngassocs = 0;
            [[wall handlers]
               raise:[GestureEvent instigator:self event:e
                     env:[[Env new] str:"event" value:e]]];
        }
        return self;
    }
```

The passive handler allocates a new `Gesture` object which will be sent the input events as they arrive. The initial event is sent immediately.

The `picture` allows the gesture handler to ink the gesture on the display as it is being made. Class `ImmediatePicture` is used for pictures which are displayed as they are drawn, rather than the normal `HangingPicture` class which requires pictures to be completed before they can be drawn.

The `env` variable holds the environment in which the gesture semantics will be executed. Within this environment, the interpreter variables `gesture`, `startEvent`, `currentEvent`, and `handler` are bound appropriately (see Section 7.7.1).

The boolean `manip_phase` is true if and only if the entire gesture has been collected and the handler is now in the manipulation phase (see Section 1.1).

A `TimeoutEventHandler` is created and activated. When a `TimeoutEvent` is received by the handler, the handler will send an arbitrary message (with the timeout event as a parameter) to an arbitrary object. In the current case, the `timedout:` message is sent to the active `GestureEventHandler`. In retrospect, the general functionality of the `TimeoutEventHandler` is not needed here; the `GestureEventHandler` could itself easily receive and process `TimeoutEvents` directly, without the overhead of a `TimeoutEventHandler`.

The code `[wall timeout:timeval]` causes the wall to raise a `TimeoutEvent` if there has been no input to the wall in `timeval` milliseconds. A `timeval` of zero disables the raising of `TimeoutEvents`. As previously mentioned, a gesture is considered complete even if the mouse button is held down, as long as the mouse has not been moved in `timeval` milliseconds. The `TimeoutEvent` is used to implement this behavior.

If the gesture being collected is intended to be classified, the set of possible gesture classes must be constructed, and a `Set` object is allocated for this purpose. Recall from Section 7.2.2 that there may be multiple views at the location of the start gesture each of which accepts certain gestures. An array of `gassoc` structures is allocated to associate each of the possible gesture classes expected with its corresponding view. A `GestureEvent` is then raised, with the instigator being the current gesture handler, and having the current event as an additional field.

Raising the `GestureEvent` initiates the search for the possible gesture classes given the initial event. Recall from Sections 7.2.1 and 7.2.2 that each view under the initial point is considered from top to bottom, and for each view, the gestures associated directly with the view itself, and with its class and superclasses, are added in order. Note that this is exactly the same search sequence as that used to find passive event handlers for events that no active handler wants (see Section 6.7). The `GestureEvent`, handled by the same passive event handler mechanism, will thus be propagated to other `GestureEventHandlers` in the correct order. Each passive gesture handler that would have handled the initial event sends a message to the gesture handler which raised the `GestureEvent` indicating the set of gesture classes it recognizes and the view with which it is associated.

Note that only views under the first point of the gesture are queried. The case where a gesture is more naturally expressed by not beginning on the view at which it is targeted is not handled by GRANDMA. For example, it would be desirable for a knob turning gesture to go around the knob, rather than directly over it. In GRANMDA either the knob view area would have to be larger than the actual knob graphic to insure that the starting point of the gesture is over the knob view, or a background view that includes the knob as a subview must handle the knob-turning gesture. In the latter case, the gesture semantics are complicated because the background view needs to explicitly determine at which knob, if any, the gesture is directed. Henry et. al. [52] also notes the problem, and suggests that one gesture handler might hand off a gesture in progress to another handler if it determines that the initial point of the gesture was misleading, but exactly how such a determination would be made is unclear.

```
= GestureEventHandler ...
- (BOOL)event:e view:v {
    if((classify && masterSwitch==NO) || ignoring==YES)
        return NO;
    if( [e isKindOf:GestureEvent] ) {
        if(classify
        && [self evalstart:[[e env] str:"view" value:v] )
                [ [e instigator] classes:classes view:v ];
        return NO;
    }
```

```
        return [super event:e view:v];
    }
```

The `GestureEventHandler` overrides `GenericEventHandler`'s `event:view:` method to check directly for `GestureEvents`. (A check for `GestureEvents` could have been included in the default start predicate, but this would require programs which modify the start predicate to always include such a check, an unnecessary complication.) First the state of the `masterSwitch` and `ignoring` switches is checked, so that this handler will not operate if explicitly turned off. (The reason `classify` is checked is to allow gesture handlers which do not classify gestures, *i.e.* those used to collect gesture examples for training purposes, to operate even though gestures are disabled throughout the system.)

When a `GestureEvent` is seen, the handler checks that it indeed classifies gestures and that it would itself have handled the start event (see Section 6.7.8). The environment used for evaluating the start predicate is constructed so that `"event"` and `"view"` are bound to what they would have been had the handler actually been asked to handle the initial event. If the handler would have handled the event, the set of gesture classes associated with the handler, as well as the view, are passed to the handler which instigated the `GestureEvent`.

Note that no special case is needed for the handler which actually raised the `GestureEvent`. This handler will be the first to receive and respond to the `GestureEvent`, which it will then propagate to any other handlers. The propagation occurs simply because the `event:view:` method returns `NO`, as if it did not handle the event at all.

```
    = GestureEventHandler ...
    - classes:gesture_classes view:v {
        id c, seq = [gesture_classes eachElement];
        while( c = [seq next] ) {
            if([class_set addNTest:c]) {  /* added new element? */
                gassoc[ngassocs].sclass = c;
                gassoc[ngassocs].view = v;
                ngassocs++;
            }
        }
        return self;
    }
```

Each gesture handler that could have handled the initial event sends the gesture handler that did handle the initial event the `classes:view:` message. The latter handler then adds each gesture class to its `class_set`. If the gesture class was not previously there, it is associated with the passed view via the `gassoc` array. This membership test assures that when a given gesture class is expected by more than one view (at the initial point), the topmost view will be associated with the gesture class.

By the time the `GestureEvent` has finished propagating, the `class_set` variable of the instigator will have as elements the gesture classes (`SemClass` objects, actually) that are valid given the initial event. The `gassoc` variable of the instigator will associate each such gesture class with the view that will be affected if the gesture being entered turns out to be that class.

The search for the set of valid gesture classes may be relatively expensive, especially if there are a significant number of views under the initial event and each view has a number of event handlers associated with it. The substantial fraction of a second consumed by the search had an unfortunate interaction with the lower level window manager interface that resulted in an increase in recognition errors. When queried, the low-level window manager software returns only the latest mouse event, discarding any intermediate mouse events that occurred since it was last queried. The time interval between the first and second point of the gesture was often many times larger than the interval between subsequent pairs of points. More importantly, it was much larger than that of the first and second points of the gesture examples used to train the classifier. Details at the beginning of gestures would be lost, and some features, such as the initial angle, would be significantly different. The substantial delay in sampling the second point of the gesture thus caused the classifier performance to degrade.

There are a number of possible solutions to this problem. The window manager software could be set to not discard intermediate mouse events, thus resulting in similar data in the actual and training gestures. This would result in a large additional number of mouse events, and a corresponding increase in processing costs, making the system appear sluggish to the user if events could not be processed as fast as they arrived. Or, the search for gesture classes could be postponed until after the gesture was collected. This would result in a substantial delay after the gesture was collected, again making the system appear sluggish to the user. The solution finally adopted was to poll the window manager during the raising of `GestureEvents`. (In the interest of clarity, the code in `XyEventHandler` and `EventHandlerList` which did the polling was not shown.) After this modification, running `GestureEventHandlers` received input events at the same rate as the `GestureEventHandlers` used for training, improving recognition performance considerably.

The polling resulted in new mouse events being raised before the `GestureEvent` was finished being propagated. The result was a kind of pseudo-multi-threaded operation, with many of the typical problems which arise when concurrency is a possibility. `GestureEventHandlers` were complicated somewhat, since, for example, they had to explicitly deal with the possibility that the end of the gesture might be seen before the set of possible gesture classes was calculated. Also, the event handling methods for `GestureEventHandlers` had to be made reentrant. The complications have been omitted from the code shown here, since they tend to make the program much more difficult to understand.

The end of a gesture is indicated either by a timeout event (resulting in a `timedout:` message being sent to the `GestureEventHandler`), or by the stop predicate being satisfied (resulting in the `activeTerminator:` message being sent to the handler). The third alternative, eager recognition (Chapter 4), has not yet been integrated into the GRANDMA gesture handler, though it has been tested in non-GRANDMA applications (see Section 9.2).

```
= GestureEventHandler ...
- timedout:e { if( !  [self gesture:gesture] )
                        [self deactivate]; return nil; }

- activeTerminator:e {
    [env str:"currentEvent" value:[e copy]];
```

```
        if(! manip_phase) [self gesture:gesture];
        return self;
    }
```

Both methods result in the `gesture:` message being sent when the gesture has been completely collected. The `gesture:` message returns `nil` if the gesture has no semantics to be evaluated during the manipulation phase. This is checked by the `timedout:` method, and in this case the handler simply deactivates itself immediately. This is typically used by gesture classes whose recognition semantics change the mouse tool (*e.g.* a delete gesture that changes the mouse cursor to a delete tool); a timeout deactivates the gesture handler immediately, allowing the mouse to function as a tool as long as the mouse button is held.

The `GenericEventHandler` code arranges for the `deactivate` message to be sent immediately after the `activeTerminator:` message, so there is no need for the `activeTerminator:` method to explicitly send `deactivate`. The environment is changed so that the semantic expression evaluated in the `deactivate` method executes in the correct environment. The `gesture:` method is called if the handler is still in the gesture collection phase, *e.g.* if the gesture end was indicated by releasing the mouse button rather than a timeout.

```
    = GestureEventHandler ...
    - deactivate {
        id r;
        if(manip_phase && sclass)
            eval([sclass done_expr], env, TypeId, &r);
        return [super deactivate];
    }
```

The `gesture:` method sets the `sclass` field to the `SemClass` object of the recognized gesture. The *done expression*, the last of three semantic expressions, is evaluated immediately before the gesture handler is deactivated.

```
    = GestureEventHandler ...
    - (BOOL)event:e { return ignoring ? NO : [super event:e]; }

    - activeHandler:e {          /* new mouse point */
        [env str:"currentEvent" value:[e copy]];
        if( manip_phase) { id r;      /* in manipulation phase */
            if(sclass) eval([sclass manip_expr], env, TypeId, &r);
        }
        else {                         /* still in collection phase */
            int x = [e [loc x]], y = [e [loc y]];
            [gesture newevent:e];        /* update feature vector */
            [view updatePicture:
                   [picture line:lastx :lasty :x :y]]; /* ink */
            lastx = x; lasty = y;
        }
        return self;
```

```
}
```

Once activated, the `GestureEventHandler` functions just like any other `GenericEventHandler` except that it will not handle any events if its `ignoring` flag is set. The active event handler does different things depending on whether the gesture handler is in the collection phase or the manipulation phase. In the former case, the current event location is added to the gesture, and a line connecting the previous location to the current one is drawn on the display. In the latter case, the *manipulation expression* associated with the gesture (the second of the three semantic expressions) is evaluated.

```
= GestureEventHandler ...
- gesture:g { /* called when gesture collection phase in complete */
    double a, d;
    id r;
    id classifier;
    register struct gassoc *ga;
    id c, class;
    id curevent;

    manip_phase = YES;
    [wall timeout:0]; [wall deactivate:timeouteh];
    [view _unhang:picture];    /* erase inking */
    [picture discard]; picture = nil;

    /* inform interested views (only used in a training session) */
    if([view respondsTo:@selector(gesture:)])
        [view gesture:g];

    if(classify) {
        /* find a classifier for the set; create it if necessary */
        classifier = [Classifier lookupOrCreate:class_set];
        /* run the classifier on the feature vector of the collected gesture */
        class = [classifier classify:[g fv]
                            ambigprob:&a distance:&d];
        sclass = nil;
        if(class == nil || a < AmbigProb || d > MaxDist)
            return [self reject];        /* rejected */

        /* find the class of the gesture in the gassoc array */
        for(ga = gassoc; ga < &gassoc[ngassocs]; ga++)
            if([ga->sclass gclass] == class)
                break;
        if(ga == &gassoc[ngassocs])
            return [self error:"gassocs?"];
```

```
      /* the gassoc entry gives the both the view at which the gesture */
      /* is directed and the semantic expressions of the gesture */
      sclass = ga->sclass;
      [env str:"view" value:ga->view];
      [env str:"endEvent"
          value:curevent=[env atStr:"currentEvent"]];
      eval([sclass recog_expr], env, TypeId, &r);
      if((c = [sclass manip_expr]) != nil &&
         [c val] != nil)
            eval(c, env, TypeId, &r);
      else {   /* raise event */
         if(curevent) {
            ignoring = YES;
            if(mousetool) [curevent tool:mousetool];
            [wall raise:curevent];
         }
         if( (c = [sclass done_expr]) == nil
            || [c val] == nil)
               return nil;
      }
   }
   return self;
}
```

The `gesture:` method is called when the entire gesture has been collected. It sets the variable `manip_phase` to indicate the handler is now in the manipulation phase of the gestural input cycle, deactivates the timeout event handler, and erases the gesture from the display. If the view associated with the handler responds to `gesture:` it is sent that message, with the collected gesture as argument. This is the mechanism by which example gestures are collected during training: one handler collects the gesture, sends its view (typically a kind of `WallView` devoted to training) the example gesture, which adds it to the `GestureClass` being trained.

In the typical case, the gesture is to be classified. The `Classifier` factory method named `lookupOrCreate:` is called to find a gesture classifier which discriminated between elements of the `class_set`. If no such classifier is found, this method calculates one and caches it for future use. (This lookup and creation could possibly have been done in the pseudo-thread that was spawned during the first point of the gesture, but was not, since most of the time the lookup finds the classifier in the cache, and it was not worth the additional complication and loss of modularity to add polling to the classifier creation code.) The returned classifier is then used to classify the gesture. In addition to the class, the probability that the classification was ambiguous and the distance of the example gesture to the mean of the calculated class are returned. These are compared against thresholds to check for possible rejection of the gesture (see Section 3.6).

The elements of the `gassoc` array are searched to find the one whose gesture class is the class returned by the classifier. This determines both the semantics of the recognized gesture and the view at which the gesture was directed. The `sclass` field is set to the `SemClass` object associated with the recognized gesture, and then the *recognition expression*, the first of the three semantic expressions, is evaluated in an environment in which `"startEvent", "currentEvent", "endEvent"` and `"view"` are all appropriately bound.

If it exists, the manipulation expression is evaluated immediately after evaluating the recognition expression. If there is no manipulation expression, the current event is reraised on the assumption that its tool may wish to operate on a view. The `ignoring` flag is set so that the active handler does not attempt to handle the event it is about to raise. Furthermore, the semantics of the gesture may have changed the current mouse tool. If so, the tool field of the current event would be incorrect, and is changed to the new tool before the event is raised. In order for this to work, any gesture semantics that wish to change the current mouse tool must do so by sending the `mousetool:` message to the gesture handler instead of directly to the wallview.

```
= GestureEventHandler ...
- mousetool:_mousetool {
        mousetool = _mousetool;
        return [super mousetool:_mousetool];
}
```

The `gesture:` method returns `nil` if there are no manipulation or done semantics associated with the recognized gesture class. As seen, this is a signal for the handler to be deactivated immediately after the gesture is recognized.

7.5 Gesture Classification and Training

In this section the implementation of classes which support the gesture classification and training algorithms of Chapter 3 is discussed.

At the lowest level is the class `Gesture`. A `Gesture` object represents a single example of a gesture. These objects are created and manipulated by `GestureEventHandlers`, both during the normal gesture recognition that occurs when an application is being used, and during the specification of gesture classes when training classifiers.

7.5.1 Class `Gesture`

Internally, a gesture object is an array of points, each consisting of an x, y, and time coordinate. Another instance variable is the `GestureClass` object of this example gesture, which is non-`nil` if this example was specified during training. Intermediate values used in the calculation of the example's feature vector, as well as the feature vector itself, are also stored. Also, an arbitrary string of text may be associated with a `Gesture` object.

For brevity, detailed listing of the code for the `Gesture` class is avoided. The interesting part, namely the feature vector calculation, has already been specified in detail in Chapter 3 and C code is shown in Appendix A. Instead of listing more code here, an explanation of each message `Gesture` objects respond to is given.

A new gesture is allocated and initialized via `g = [Gesture new]`. Adding a point to a `Gesture` objects is done by sending it the `newevent` message: `[g newevent:e]`, which simply results in the call: `[g x:[[e loc] x] y:[[e loc] y] t:[e time]]`. The `x:y:t:` method adds the new point to the list of points, and incrementally calculates the various components of the feature vector (see Section 3.3). The call `[g fv]` returns the calculated feature vector. The methods `class:`, `class`, `text:`, and `text` respectively set and get the class and text instance variables.

A `Gesture` object can dump itself to a file via `[g save:f]` (given a file stream pointer `FILE *f`) and can also initialize itself from a file dump using `[g read:f]`. Using `save:`, a number of gesture objects may dump themselves sequentially into a single file, and could then be read back one at a time using `read:`. All examples of a given gesture class are stored in a single file via these methods.

The call `[g contains:x:y]` returns a boolean value indicating if the gesture `g`, when closed by connecting its last point to its first point, contains the point `(x,y)`. This is useful for testing, for example, if a given view has been encircled by the gesture, enabling the gesture to indicate the scope of a command. (The algorithm for testing if a point is within a given gesture is described at the end of section 7.7.3.)

7.5.2 Class `GestureClass`

The class `GestureClass` represents a gesture class. A gesture class is simply a set of example gestures, presumably alike, that are to be considered the same for the purposes of classification. The input to the gesture classifier training method is a set of `GestureClass` objects; the result of classifying a gesture is a `GestureClass` object.

```
= GestureClass:  NamedModel {
    id      examples;
    Vector  sum, average;
    Matrix  sumcov;
    int     state;
    STR     text;
}
```

`GestureClass` is a subclass of `NamedModel`, itself a subclass of `Model`. `GestureClass` is a model so that it can have views, enabling new gesture classes to be created and manipulated at runtime. Please do not confuse `GestureClass` with `GestureEventHandler` objects; a `GestureClass` serves only to represent a class of gestures, and itself handles no input. A `NamedModel` augments the capabilities of a `Model` by adding functions that facilitate reading and writing the model to a file. Also, models read this way are cached, so that a model asked to be input more than once is only read once. This is important for gesture class objects, since a single `GestureClass` object may be a constituent of many different classifiers, and it is necessary that every classifier recognizing a particular class refer to the same `GestureClass` object.

The `GestureClass` instance variable `examples` is a `Set` of examples which make up the class. The field `sum` is the vector that the sum of all feature vectors of every example in the class; `average` is `sum` divided by the number of examples. The covariance matrix for this class may be found by dividing the matrix `sumcov` by one less than the number of examples. The calculation of classifiers is slightly more efficient given `sumcov` matrices, rather than covariance matrices, as input (see Chapter 3). C code to calculate the `sumcov` matrices incrementally is shown in Appendix A.

The `state` instance variable is a set of bit fields indicating whether the `average` and `sumcov` variables are up to date. The `text` field allows an arbitrary text string to be associated with a gesture class.

The `addExample:` method adds a `Gesture` to the set of examples in the gesture class, incrementally updating the `sum` field. The `removeExample:` method deletes the passed `Gesture` from the class, updating `sum` accordingly. The `examples` method returns the set of examples of this class, `average` returns the estimated mean of the feature vector of all the examples in this class, `nexamples` returns the number of examples, and `sumcov` returns the unnormalized estimated covariance matrix.

7.5.3 Class `GestureSemClass`

```
= GestureSemClass:  NamedModel {
        id        gclass;
        id        recog, manip, done;
}
```

`GestureSemClass` objects are named models, enabling them to be referred to by name for reading or writing to disk, and for being automatically cached when read. The purpose of `GestureSemClass` objects is to associate a given gesture class with a set of semantics. It is necessary to have a separate class for this because a given `GestureClass` may have more than one set of semantics associated with it.

In addition to methods for setting and getting each field, there are methods for reading and writing `GestureSemClass` objects to disk. `GestureSemClass` uses Objective C's `Filer` class to read and write each of the three semantic expressions (`recog`, `manip`, and `done`). The availability of the `Filer` is another advantage of using Objective C [28]. In a typical interpreter, a substantial amount of coding would be required to read and write the intermediate tree form of the program to and from disk files. The `Filer`, which allows the writing to and from disk of any object (at least those having no C pointers besides strings and `id`s as instance variables), made it trivial to save interpreter expressions to disk.

Along with the semantics, the disk file of a `GestureSemClass` contains only the name of `gestureClass` object referred to by `gclass`. When reading in a `GestureSemClass`, the name is used to read in the associated `GestureClass`. Since `GestureClass` is a `NamedModel`, there will be only one `GestureClass` object for each distinct gesture class.

7.5.4 Class `Classifier`

The `Classifier` class encapsulates the basic gesture recognition capabilities in GRANDMA. Each `Classifier` object has a set (actually an `OrdCltn`) of gesture classes between which it discriminates. Each `Classifier` object contains the linear evaluation function for each class (as described in Chapter 3), and the inverse of the average covariance matrix, which is used to calculate the discrimination functions, as well as to calculate the Mahalanobis distance between two of the component gesture classes, or a given gesture example and one of the gesture classes.

```
= Classifier :  Object {
        id            gestureclasses;
        int           nclasses, nfeatures;
        Vector        cnst, *w;          /* discrimination functions */
        Matrix        invavgcov;
        int           hashvalue;
}
```

`[Classifier lookupOrCreate:classes]` returns a classifier which discriminates between the gesture classes in the passed collection `classes`. The method for `lookupOrCreate:` caches all classifier objects which it creates; thus, if it is subsequently passed a set of gesture classes which it has seen before, it returns the classifier for that set without having to recompute it. The search for an existing classifier for a given set of gestures is facilitated by the `hashvalue` instance variable, which is calculated by "XORing" together the object ids of the particular `GestureClass` objects in the set.

When necessary, the `lookupOrCreate:` method creates a new classifier object, initializes its `gestureclasses` instance variable and then sends itself the `train` message. The `train` method implements the training algorithm of chapter 3.

```
– train {
    register int i, j;
    int denom = 0;
    id c, seq;
    register Matrix s, avgcov;
    Vector avg;
    double det;

    /* eliminate any gesture classes with no examples */
    [self eliminateEmptyClasses];

    /* calculate the average covariance matrix from the (unnormalized)
       covariance matrices of the gesture classes. */
    avgcov = NewMatrix(nfeatures, nfeatures);
    ZeroMatrix(avgcov);
    for(seq = [gestureclasses eachElement];
        c = [[seq next] gclass]; ) {
            denom += [c nexamples] – 1;
```

```
            s = [c sumcov];
            for(i = 0; i < nfeatures; i++)
                for(j = i; j < nfeatures; j++)
                    avgcov[i][j] += s[i][j];
    }

    if(denom == 0) [self error:"no examples"];
    for(i = 0; i < nfeatures; i++)
        for(j = i; j < nfeatures; j++)
            avgcov[j][i] = (avgcov[i][j] /= denom);

    /* invert the average covariance matrix */
    invavgcov = NewMatrix(nfeatures, nfeatures);
    det = InvertMatrix(avgcov, invavgcov);
    if(det == 0.0)
        [self fixClassifier:avgcov];

    /* calculate the discriminationfunctions:
       w[i][j] is the weight on the jth feature of the ith class.
      cnst[i] is the constant termfor the ith class. */

    w = allocate(nclasses, Vector);
    cnst = NewVector(nclasses);
    for(i = 0; i < nclasses; i++) {
        avg = [[[gestureclasses at:i] gclass] average];
                                        /* w[i] = avg*invavgcov */
        w[i] = NewVector(nfeatures);
        VectorTimesMatrix(avg, invavgcov, w[i]);
        cnst[i] = −0.5 * InnerProduct(w[i], avg);
    }
}
```

The `eliminateEmptyClasses` method removes any gesture classes from the set which have no examples. The (estimated) average covariance matrix is then computed, and an attempt is made to invert it. If it is singular, the `fixClassifier:` method is called, which creates a usable inverse covariance matrix as described in Section 3.5.2. (C code for fixing the classifier is shown in Appendix A.)

Given the inverse covariance matrix, the discrimination functions for each class are calculated as specified in Section 3.5.2. The weights on the features for a given class are computed by multiplying the inverse average covariance matrix by the average feature vector of the class, while the constant term is computed as negative one-half of the weights applied to the class average. This constant computation gives optimal classifiers under the assumptions of that all classes are equally likely and the misclassifications between classes have equal cost (also assumed is multivariate normality and a common covariance matrix). The `Classifier` class provides a `class:incrconst:` method which allows the constant terms for a given class to be adjusted if the application so desires.

The call `[Classifier trainall:classes]` causes all `Classifier` objects whose set of gestures includes all the gestures in the set `classes` to be retrained (by sending them the `train:` message). This is useful whenever training examples are added or deleted, since all the classifiers depending on this class can then be recalculated at once. Generally a classifier may be retrained in less than a quarter second; Section 9.1.7 presents training times in detail.

Classifying a given example gesture is done by the `classify:ambigprob:distance:` method. This method is passed the feature vector of the example gesture, and evaluates the discrimination function for each class, choosing the maximum. If desired, the probability that the gesture is unambiguous, as well as the Mahalanobis distance of the example gesture from the its calculated class are also computed; this allow the callers of the classification method to implement rejection options if they so choose.

```
- classify:(Vector)fv
  ambigprob:(double *)ap distance:(double *)dp
{
    double maxdisc, disc[MAXCLASSES];
    register int i, maxclass;
    double denom, exp();
    id class;

    for(i = 0; i < nclasses; i++)
        disc[i] = InnerProduct(w[i], fv) + cnst[i];

    maxclass = 0;
    for(i = 1; i < nclasses; i++)
        if(disc[i] > disc[maxclass])
            maxclass = i;
    class = [[gestureclasses at:maxclass] gclass];

    if(ap) {     /* calculate probability of non-ambiguity */
        for(denom = 0, i = 0; i < nclasses; i++)
            denom += exp(disc[i] - disc[maxclass]);
        *ap = 1.0 / denom;
    }

    if(dp)       /* calculate distance to mean of chosen class */
        *dp = [class d2fv:fv sigmainv:invavgcov];

    return class;
}
```

`Classifier` objects respond to numerous messages not yet mentioned. The `evaluate` message causes the example gestures of each class to be classified, so that the recognition rate of the classifier may be estimated. Of course, the procedure of testing the classifier on the very examples it was trained upon results in an overoptimistic evaluation, but it nonetheless is useful. By sending the particular gesture classes and examples `text:` messages, the result of the evaluation is fed back to the user, who can then see which examples of each class were classified incorrectly. A high rate of misclassification usually points to an ambiguity, indicating a poor design of the set of gestures to be recognized. The ambiguity is typically fixed by modifying the gesture examples of one or more of the gesture classes. The incorrectly classified examples indicate to the gesture designer which gesture classes need to be revised.

`Classifier` objects also respond to messages which save and restore classifiers to files, as well as messages which cause the internal state of a classifier to be printed on the terminal for debugging purposes, and a matrix of the Mahalanobis distances between class pairs to be printed (so that the gesture designer can get a measure of how confusable the set of gestures is).

7.6 Manipulating Gesture Event Handlers at Runtime

One goal of this work was to provide a platform that allows experimentation with different gestural interfaces to a given application. To this end, GRANDMA was designed to allow gesture recognizers to be manipulated at runtime. Gesture classes may be added or deleted, training examples for each class may also be added or deleted, and the semantics of a gesture class (with respect to a particular handler) may all be specified at runtime. In addition, gestures as a whole, or particular gesture event handlers, may be turned on and off at runtime, allowing, for example, easy comparison between gesture-based and click-drag interfaces to the same application program. This section discusses the interface GRANDMA presents to the user that facilitates the manipulation of gesture handlers at runtime.

The `View` class implements the `editHandlers` method. When sent `editHandlers`, a view creates a new window (if one does not already exist) as shown in figure 7.2. The top row is a set of pull down menus. Each subsequent row lists the passive event handlers for the view, its class, its superclass, and so on up the class hierarchy until the `View` class. The event handlers are listed in the order that they are queried for events, from top to bottom, and within a row, from left to right.

The "Mouse mode" menu item controls which mouse cursor is currently active in the window. With the normal mouse (indicated by an arrow), the user is able to drag the individual event handler boxes so as to rearrange the order. (The other mode, "edit handler," will be discussed shortly.) A handler may also be dragged into the trash box, in which case it is removed from the list of handler associated with a view or view class. A handler may be dragged into the dock; anything in the dock will remain visible when the handler lists for a different view are accessed. A handler dragged into the dock reappears on its original list as well; thus the dock allows the same event handlers to be shared between different objects and between different classes.

The "create handler" menu item results in a pull-down menu of all classes which respond to the `passive` message. Thus, at runtime new handlers may be created and associated with any view object or class. For example, a drag handler may be created and attached to an object, which can

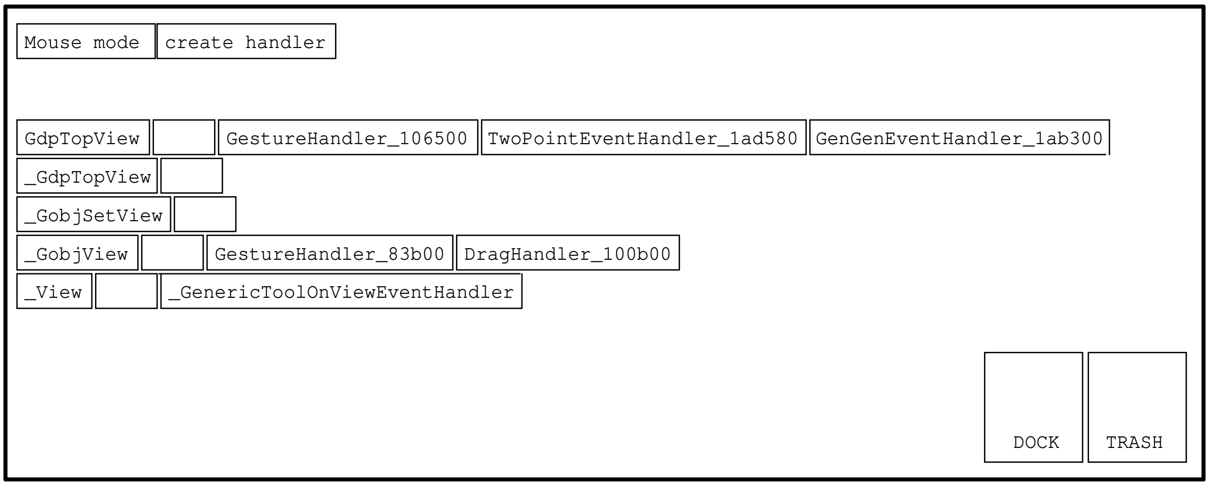


Figure 7.2: Passive Event Handler Lists

then be dragged around with the mouse. New gesture handlers may also be created this way.

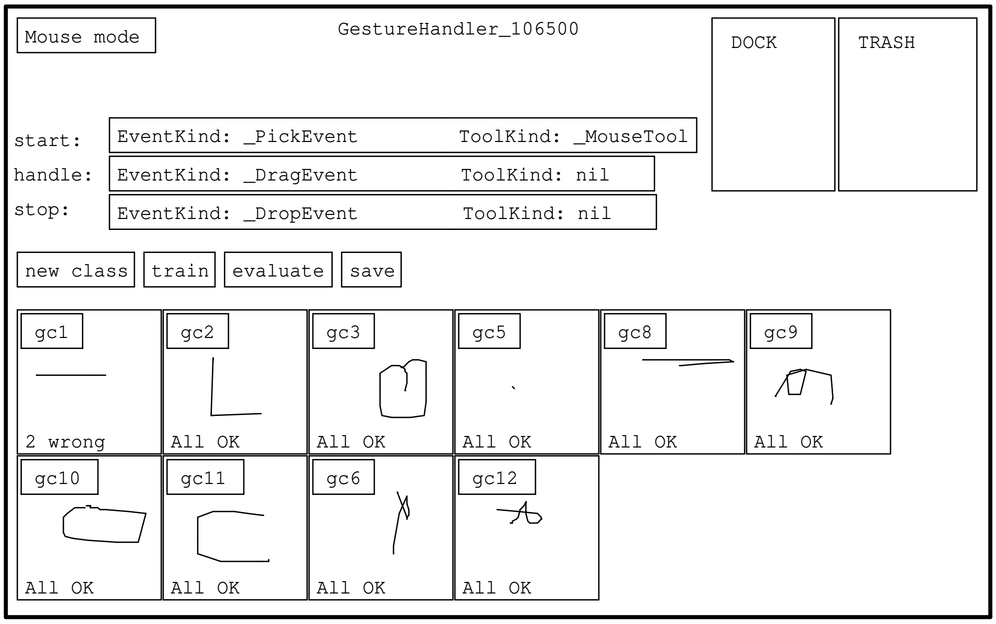
The other mouse cursor, “edit handler”, may be clicked upon any passive event handler. It results in a new window being created which shows the details of a particular edit handler. Figure 7.3 shows the window for a typical gesture handler.

At the top left of the window is the “Mouse mode” pull down menu, used in the unlikely event that one wishes to examine the handlers of any of the views in this window. To the right is the name of this event handler, constructed by concatenating the class of the handler with its internal address.

The next three rows show three `EventExpr` objects; these are the starting predicate, handling predicate and stopping predicate of the gesture handler. Each item in the predicate display is a button that shows a pop-up menu; it is thus a simple matter to change the predicates at runtime. For example, the start predicate may be changed from matching only `PickEvents` to matching all `DragEvents`. The kind of tool expected may also be changed at runtime, as well as attributes of the tool (*e.g.* a particular mouse button may be specified). If desired, the entire predicate expression may be replaced by a completely new expression. In all cases, the changes take effect immediately.

The window contents thus far discussed are common to all `GenericEventHandlers`. The following ones are particular to `GestureEventHandlers`. First there are a set of buttons (“new class”, “train”, “evaluate”, “save”). Below this are some squares, each representing a gesture class recognized by this handler. In each square is a miniaturized example gesture, some text associated with the class, and a small rectangle which names the class. The text typically shows the result of the evaluation of the particular gesture recognizer for this set of classes when run on the examples used to train it. The small rectangles may be dragged (copied) into the dock. Each such rectangle represents a particular gesture class. Any rectangles in the dock will remain there when another gesture handler is edited. Each then may be dragged into any gesture class square, where it replaces the existing class. Typically, a rectangle from the dock is dragged into empty class square (created by the “new class” button); this is the way multiple gesture handlers can recognize the same class.

Clicking on one of the gesture class squares (but not in the class name rectangle) brings up the



GestureHandler_106500

start: EventKind: _PickEvent ToolKind: _MouseTool
handle: EventKind: _DragEvent ToolKind: nil
stop: EventKind: _DropEvent ToolKind: nil

Figure 7.3: A Gesture Event Handler

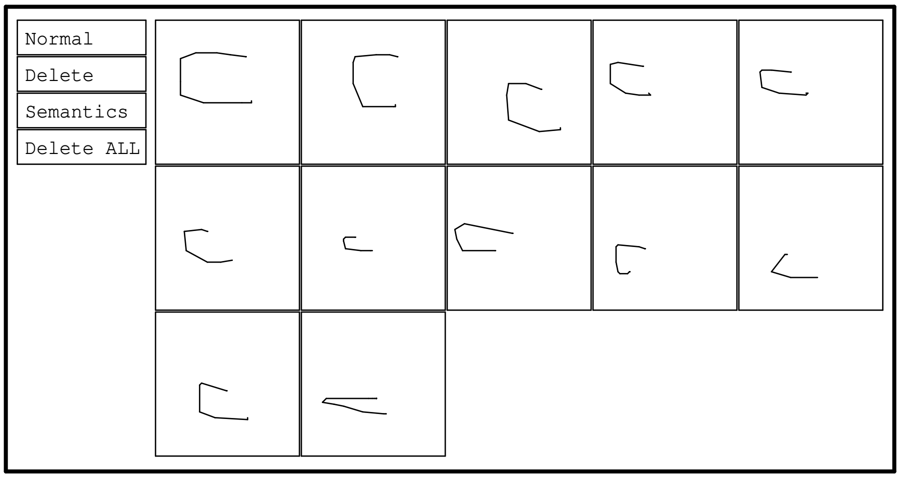


Figure 7.4: Window of examples of a gesture class

.

window of example gestures, as shown in Figure 7.4. Each square in this window contains a single, miniaturized example of a gesture in this class. These examples are used for training the classifier. A new example may be added simply by gesturing in this window. An example may be deleted by clicking the delete button on the left (which changes the mouse cursor to a delete cursor) and then clicking on the example. A user wishing to change a gesture to something more to his liking simply has to delete all the examples of the class (easily done using the "Delete ALL" button) and then enter new example gestures. The "train" button will cause a new classifier to be built, and the "evaluate" button will cause the examples to be run through the newly built classifier. Any incorrectly classified examples will be indicated by displaying the mistaken class name in the example square; the user can then examine the example to see if it was malformed or otherwise ambiguous.

The "semantics" button in the window of examples causes the semantics of the gesture class to be displayed. This is the subject of the next section.

7.7 Gesture Semantics

GRANDMA contains a simple Objective-C interpreter that allows the semantics of gestures to be specified at runtime. In GRANDMA, the semantics of a gesture are determined by three program fragments per gesture class (per handler). The first program fragment, labeled `recog`, is executed when the gesture is first recognized to be in a particular class. The second fragment, `manip`, is executed on every input event handled by the activated gesture handler after the gesture has been recognized. The third fragment, `done`, is executed just before the handler deactivates itself. The exact sequence of executions was described in detail in section 7.4; this section is concerned with the contents and specification of the program fragments themselves.

7.7.1 Gesture Semantics Code

As mentioned, the semantics of a gesture are defined by three expressions, `recog`, `manip`, and `done`. The kinds of expressions found in practice may be loosely grouped according to the level of the GRANDMA system that they access.

Some semantic expressions deal directly with models, *i.e.* directly with application objects. These are typically the easiest to code and understand. An example from the GSCORE application discussed in section 8.2 is the sharp gesture. GSCORE is an editor for musical scores. In GSCORE, making an "S" gesture over a note in the score causes the note to be "sharped", which is indicated in musical notation by placing the sharp sign "#" before the note. The class `Note` is a model in the GSCORE application, and one of its methods is `acc:` which sets the accidental of a note to one of DOUBLEFLAT, FLAT, NATURAL, SHARP, DOUBLESHARP, or NOACCIDENTAL.

The sharp gesture, performed by making an "S" over a `NoteView`, has the semantics:

```
recog = [ [view model] acc:SHARP ];
manip = nil;
done = nil;
```

In these semantics, the `Note` object (the model of the `NoteView` object) is directly sent the `acc:` message when the sharp gesture is recognized. The model then changes its internal state to reflect the new accidental, and then calls `[self modified]` which will eventually result in the display updated to add a sharp on the note.

Note that the semantic expressions are evaluated in a context in which certain names are assumed to be bound. In the above example, obviously `view` and `SHARP` must be bound to their correct values for the code to work. Section 7.4 described how the `GestureEventHandler` creates an environment where `view` is bound to the view at which the gesture is directed, `startEvent` is bound to the initial event of the gesture, `endEvent` is bound to the last event of the gesture (*i.e.* the event just before the gesture was classified), and `currentEvent` is bound to the most recent event, typically a `MoveEvent` during the manipulation phase. A particular application may globally bind application-specific symbols (such as `SHARP` in the above example) in order to facilitate the writing of semantic expressions.

Instead of dealing directly with the model, the semantics of a gesture may send messages directly to the view object. In the score editor, for example, the delete gesture (in the handler associated with a `ScoreEvent`) might have the semantics

```
recog = [view delete];
manip = nil;
done = nil;
```

(The actual semantics are slightly more complicated since they also change the mouse cursor; see Section 8.2 for details.) The `delete` method for the typical view just sends `delete` to its model, perhaps after doing some housekeeping.

The semantic expressions of a gesture are invoked from a `GestureEventHandler`, and the sending of messages to models and views seen so far is typical of many different kinds of event handlers. Another thing that event handlers often do (see in particular section 6.7.9 for a discussion of the `DragHandler`) is raise events of their own. There are many reasons a handler might wish to do this. A `DragHandler` raises events in order to make the view being dragged be considered a virtual tool. As mentioned previously, a handler might also raise events in order to simulate one input device with another. (For example, imagine a `SensorFrameMouseEmulator` which responds to `SensorFrameEvents`, raising `DragEvents` whose tool is the current `GenericMouseTool` so as to simulate a mouse with a Sensor Frame.) One of the main purposes of having an active event handler list and a list of passive events handlers associated with each view is to allow this kind of flexibility. In the Smalltalk MVC system, the pairing of a single controller with a view really constrains the view to deal only with a single kind of input, namely mouse input. In GRANDMA, a view can have a number of different event handlers, and thus may be able to deal with many different input devices and methods.

In GRANDMA, gesture-based applications are typically first written and debugged with a more traditional menu driven, click-and-drag, direct manipulation interface. Given that gestures are added on top of this existing structure, there is another level at which gesture semantics may be written. At this level, the gesture semantics emulate, for example, the mouse input that would give the appropriate behavior. In other words, the gesture is translated into a click-and-drag interaction which gives the desired result.

An example of this from the score editor is the placement of a note into a score. In the click-and-drag interface, adding a note to the score involves dragging a note of appropriate duration from a palette of notes to its desired location in a musical staff. This is implemented by having the `NoteView` be a virtual tool which sends a message to which `StaffView` objects respond. While the note is being dragged, a `DragHandler` raises an event whose tool is a `NoteView` which will be processed by the `GenericToolOnView` handler when the note is over the `StaffView`.

In the gesture-based interface, there is a gesture class for each possible note duration recognized by handler associated with the `StaffView` class. The semantics for the gesture which gives rise to an eighth note are

```
recog = [[[noteview8up viewcopy] at:startLoc]
                reraise:currentEvent];
manip = nil;
done = nil;
```

The symbol `noteview8up` is bound to the view of one of the notes in the palette; it is copied and moved to the starting location of the gesture. The `currentEvent` (either a `MoveEvent` or `DropEvent` which ended the gesture) is copied, its tool field is set to the copy of the note view, and the resulting event is raised. The moving of the note and the raising of a new event is exactly what a `DragHandler` does; the effect is to simulate the dragging of a note to a particular location. Note that the note is moved to `startLoc`, the starting point of the gesture, which necessarily is over a `StaffView` (otherwise this gesture handler would never have been invoked). Thus, the handlers for `StaffView` will handle the event, and use the location of the note view to determine the new note's pitch and location in the score.

It would have been possible in the semantics to simulate the mouse being clicked on the appropriate note in the palette and then being dragged onto the appropriate place in the staff. In this case, that was not done as it would be needlessly complex. The point is that, due to the flexibility of GRANDMA's input architecture, the writer of gesture semantics can address the system at many levels of abstraction, from simulated input to directly dealing with application objects.

The example semantics seen thus far have only had `recog` expressions, which are evaluated at recognition time. The following example, which implements the semantics of a gesture which creates a line and then allows the line to be rubberbanded, illustrates the use of `manip`:

```
recog = [[view createLine] endpoint0at:startLoc];
manip = [recog endpoint1at:currentLoc];
done = nil;
```

In this example, `view` is assumed to be a background view, typically a `WallView` of a drawing editor program (Section 8.1 discusses GDP, a gesture-based drawing editor). Sending it the `createLine` message results in a new line being created in the window, whose first endpoint is the start of the gesture. The other endpoint of the line moves with the mouse after the gesture has been recognized; this is the effect of the `manip` expression. Note the use of `recog` as a variable to hold the newly created line object. If desired, the semantics programmer may create other local variables to communicate between different (or even the same) semantic expressions.

7.7.2 The User Interface

GRANDMA allows the specification of gesture semantics to be done at runtime. In the current implementation, the semantics must be specified at runtime; there is no facility for hardwiring the semantic expressions of a given gesture into an application. Currently, the semantics of a gesture class are read in from a file (as are examples of the gesture class) each time an application is started. The semantics of a gesture may only be created or modified using the user interface facilities discussed in this section.

Gesture semantics are currently specified using a limited set of expressions. An expression may be a constant expression (integer or string), a variable reference, an assignment, or a message send. Each expression has its obvious effect: a constant evaluates to itself, a variable evaluates to its value in the current environment, an assignment evaluates to the evaluation of its right hand side (with the side effect of setting the variable on the left hand side), and a message send first evaluates the receiver expression and each argument expression, and then sends the specified message and resulting arguments to the receiver. The value of a message expression is the value that the receiver's method returns. For programming convenience, integer, string, and objects are converted as needed so that the types of the arguments and receiver of a message send match what is expected by the message selector.

Figure 7.5 shows the window activated when the "Semantics" button of a gesture class is pressed. At the top of the window are a row of buttons used in the creation of various kinds of expressions. They work as follows:

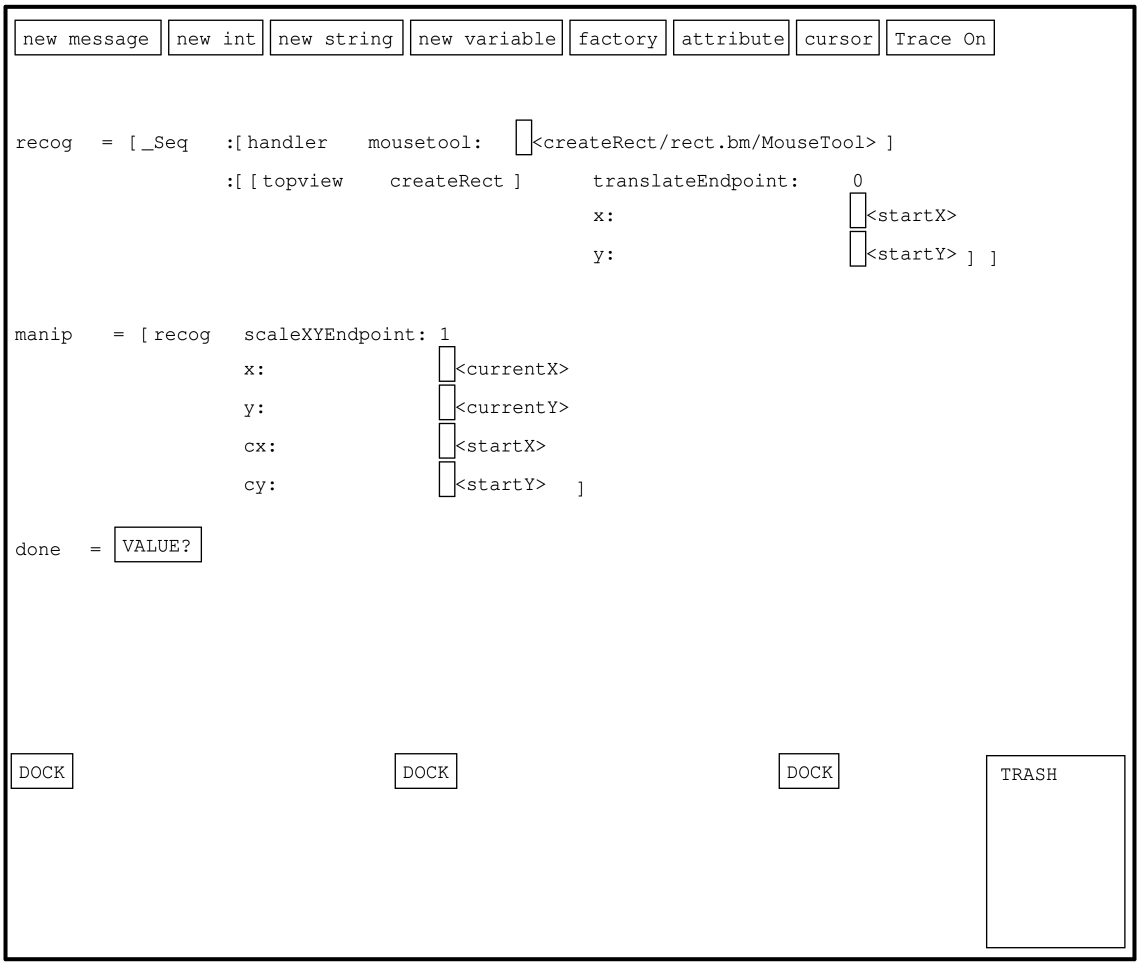
new message The new message button creates a template of a message send, with a slot for the receiver and the message selector. Any expression may then be dragged into the receiver ("REC?") slot. Clicking on the "SELECTOR?" box causes a dialogue box to be displayed (figure 7.6). Users can then browse through the class hierarchy until they find the message selector they desire, which can then be selected. The "+" and "-" buttons may be used to switch between factory and instance methods. The starting point in the browsing is set to the class of the receiver, when it can be determined. Once the selector has be okayed, the template changes to have a slot for each argument expected by the selector, as shown into figure 7.7. Any expression may then be dragged into the argument slots. In particular, gesture attributes (see below) are often used.

new int This button creates a box into which an integer may be typed.

new string This button creates a box into which a string may be typed.

new variable This button creates a template ($\boxed{} = \boxed{\text{VALUE?}}$) for assigning a variable into which the name of a variable may be typed. Any expression may then be dragged into the "VALUE?" slot. The entire assignment expression may be dragged around by the "=" sign. Attempting to drag the variable name on the left hand side actually copies the variable name before allowing it to be dragged; this resulting expression (simply the name of the variable) may be used anywhere the value of the variable is needed.

factory This button generates a constant expression which is the object identifier of an Objective C class (also known as a "factory"). Pressing the button pops up a browser which allows the user to walk through the class hierarchy to select the desired class.



```
recog   = [ _Seq   :[ handler    mousetool:   <createRect/rect.bm/MouseTool> ]
                   :[ [ topview    createRect ]      translateEndpoint:    0
                                                     x:                    <startX>
                                                     y:                    <startY> ] ]

manip    = [ recog   scaleXYEndpoint: 1
                     x:                <currentX>
                     y:                <currentY>
                     cx:               <startX>
                     cy:               <startY>   ]

done   =  VALUE?
```

Figure 7.5: The interpreter window for editing gesture semantics

Figure 7.6: An empty message and a selector browser

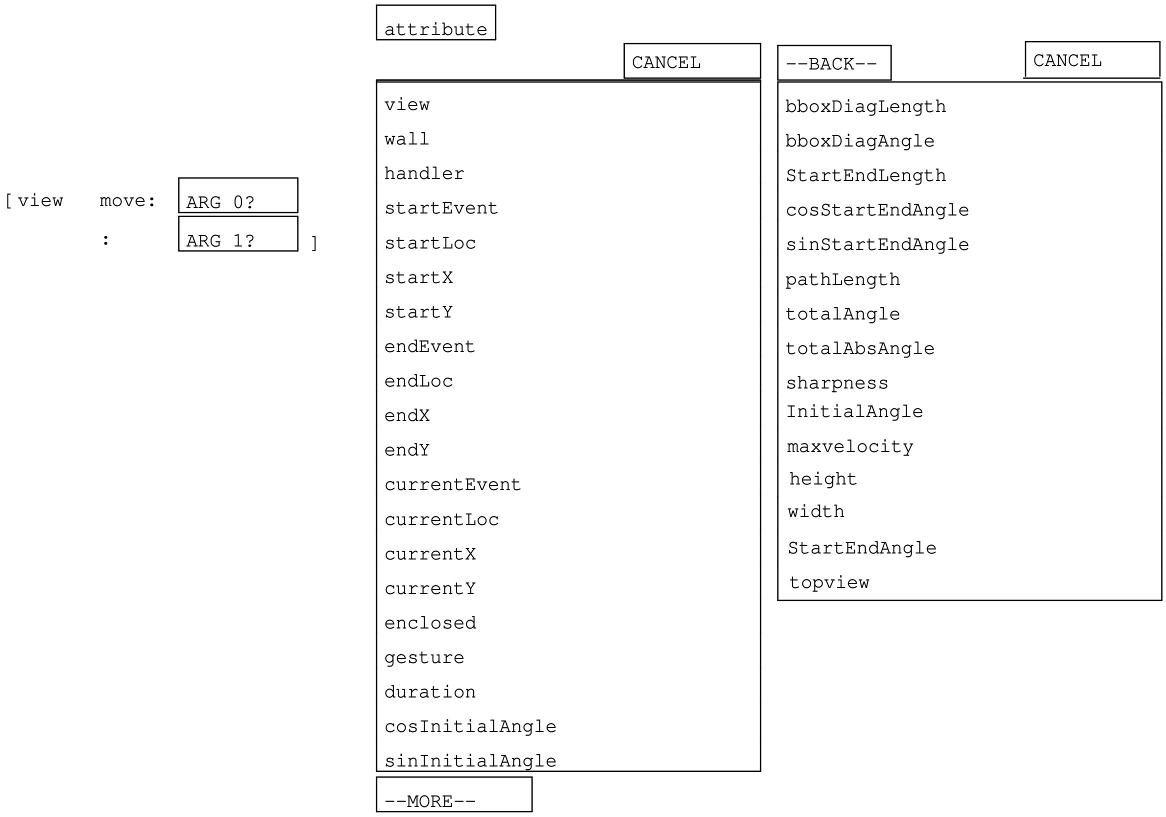


Figure 7.7: Attributes to use in gesture semantics

.

attribute Clicking this button generates a menu of useful subexpressions that are often used in gesture semantics. (Figure 7.7 shows both pages of attributes). The expressions are either variable names, or named messages. As expressions, named messages are distinguishable from variable names by the angle brackets and the small box before the name. Clicking in the box reveals the underlying expression to which the name refers. (Note the angle brackets and box are not shown in the list of attributes but appear once an attribute is selected. Figure 7.5 contains some examples of such attributes.)

Most attributes in the list refer to characteristics of the current gesture (*i.e.* the gesture which causes the semantics to be evaluated). Other attributes refer to the current view, wall, event handler, events, and set of objects enclosed by the gesture. Many examples of using attributes in gesture semantics are covered in the next chapter.

Having the attributes of a gesture available when writing the semantics of the gesture is the embodiment of one central idea of idea of this thesis. The idea is that the meaning of a gesture may depend not only upon its classification, but also on the features of the particular instance of the gesture. For example, in the drawing program it is a simple matter to tie the length of the line gesture to the thickness of the resulting line. This is in addition to using the starting point of the gesture as one endpoint of the line, another example of how gesture attributes are useful in gesture semantics.

cursor This button displays a menu of the available cursors. The cursors are almost always a kind of `GenericMouseTool`, and consists of an icon that has been read in from a file, and the message that the tool sends. The cursors are useful, for example, in semantic expressions that wish to provide some feedback to the user by changing the cursor after the gesture has been recognized.

Trace On This button turns on tracing of the interpreter evaluation loop, which prints the values of all expressions and subexpressions as they are evaluated. This helps the writer of gesture semantics to debug his code.

The middle mouse button brings up a menu of useful operations. “Normal” restores the cursor to the default cursor which drags expressions. “Copy” changes the cursor to the copy cursor, which when used to drag expressions causes them to be copied first. “Hide” hides the semantics window, which is so large that it typically obscures the application window. The various remaining editing commands are useful for examining the event handlers associated with various objects in the user interface, and are not really of general interest to the writer of gesture semantics. They would be of interest if one attempted to add a gestural interface to the interpreter itself.

An expression dragged into a “DOCK” slot remains there even when the gesture class is changed. The dock provides a useful mechanism for sharing code between different gesture classes, or between the same gesture class in different handlers. Any expression dragged into the trash is, of course, deleted.

The above-described interface to the semantics is usually slower to use than a more straightforward textual interface. A straightforward textual interface would require a parser but would still be simpler and better that the current click-and-drag interface. On the other hand, with the click-and-drag interface it is not possible to make a syntax error. The main reason such an interface was built was to exercise the facilities of the GRANDMA system. Before the project began the author suspected that a click-and-drag interface to a programming language would be awkward, and he was not surprised. He did, however, consider the possibility of building a gesture-based interface to the interpreter, one which might have been significantly more efficient to use than the current click-and drag interface. It should be possible at the present time to add a gesture-based interface to the interpreter without even recompiling, though to date the author has not made the attempt.

7.7.3 Interpreter Implementation

The interpreter internals are implemented in a most straightforward manner. The class `Expression` is a subclass of `Model` and has a subclass for each type of expression: `VarExpr`, `AssignExpr`, `MessageExpr`, and `ConstantExpr` (and some not discussed: `CharEventExpr`, `EventExpr`, and `FunctionExpr`). `AssignExpr` and `MessageExpr` objects have fields which hold their respective subexpressions, while `ConstantExpr` and `VarExpr` objects have fields which hold the constant object and name of the variable, respectively.

Expression Evaluation

All expressions are evaluated in an environment, which is simply an association of names with values (which are objects). Evaluating `VarExpr` objects is done by looking up the variable in an environment and returning its value; `AssignExpr` objects are evaluated by adding or modifying an environment so as to associate the named variable with its value. In addition to the environment that is passed whenever an expression is evaluated, there is a global environment. If a name is not found in the passed environment, it is then looked up in the global environment.

The interpreter has a number of types with which it can deal. Each type is represented by a subclass of class `Type`. An instance of one of these subclasses is a value of that type. The commonly used type classes are `TypeChar`, `TypeId`, `TypeInt`, `TypeShort`, `TypeSTR`, `TypeUnsigned`, and `TypeVoid`. The `TypeId` represents an arbitrary Objective-C object; the others represent their corresponding C type.

Consider the implementation of `TypeInt`:

```
= TypeInt :  Type { int _int; }
+ initialize { [super register:"int"];
               [super register:"long"]; }
+ set_int:(int)v { return [[super new] set_int:v]; }
+ (void *)fromObject:o result:(void *)r
        { *(int *)r = [o asInt]; return r; }
+ toObject:(void *)r { return [self set_int:*(int *)r]; }
- set_int:(int)v { _int = v; return self; }
- (int)asInt { return _int; }
- (short)asShort { return (short)_int; }
- (char)asChar { return (char)_int; }
- (unsigned)asUnsigned { return (unsigned)_int; }
```

```
- (STR)asString:(STR)s { sprintf(s, "%d", _int); return s; }
- (int)Plus:(int)b { return _int + b; }
- (int)Minus:(int)b { return _int - b; }
- (int)Times:(int)b { return _int * b; }
- (int)DividedBy:(int)b { return b == 0 ?
        [self error:"division by zero"], 0 :  _int / b; }
- (int)Mod:(int)b { return b == 0 ?
        [self error:"mod by zero"], 0 :  _int % b; }
- (int)Clip:(int)b :(int)c
        { return _int < b ?  b :  _int > c ?  c :  _int; }
- (int)Times:(int)b Plus:(int)c { return _int * b + c; }
```

The initialize method declares that this type represents the C types “int” and “long.” This information is used when reading in the files that the Objective-C compiler writes to describe the arguments and return types of message selectors. A sample line from one of these files is:

```
(id)at::,int,int;
```

This line says that the `at::` method (as implemented by `View`, for example) takes two integers as arguments, and returns an `id`, *i.e.* an object. (In Objective C, the type or signature of a selector such as `at::` must be the same in all classes that provide corresponding methods.) The interpreter reads this line and creates a `Selector` object which records the fact that `at::` expects its first argument to be `TypeInt`, its second argument to be `TypeInt`, and returns a `TypeId`. This `Selector` object is used when a `MessageExpr` whose selector is `at::` is evaluated; it assures that the arguments are converted to machine integers before the `at::` method is invoked.

The knowledge of how to do conversions is embodied in the `fromObject:result:` and `toObject:` methods. The intent is to freely convert between the values represented as machine integers, or characters, etc., and the values represented as objects. Given `int r; id anInt = TypeInt set_int:3];`, the call `[TypeInt fromObject:anInt result:&r]` sets `r` to 3. Conversely, `r = 4; anInt = [TypeInt toObject:&r];` sets `anInt` to a newly created object of class `TypeInt` whose `_int` field is 4.

Note that the ability to do arithmetic is embodied in `TypeInt`, as is the ability to convert between `TypeInts` and the other integer types (and string type).

Evaluating an expression node in a given environment is done by calling eval:

```
eval(expr, env, type, resultp)
id expr, env, type; void *resultp;
```

The `eval` function takes as argument an expression object, an environment object, a type object, and a pointer to a place to put the result. The eval function takes care of printing out tracing information, if necessary, and then simply sends `expr` the `eval:resultType:result:` message. Each expression class is responsible for knowing how to evaluate itself, and is able to convert its return value into the appropriate type.

The most interesting case is the evaluation of a `MessageExpr`:

```
= MessageExpr:  Expression {
        id      sel;            /* Selector object */
        id      rec;            /* (unevaluated) receiver object */
```

```
        id      arg[MAXARGS];       /* unevaluated arguments */
}

- (void*)eval:env resultType:rt result:(void *)r {
    id v;
    id _rec, _arg[5];
    int i;
    int nargs = [sel nargs];
    SEL _sel = [sel sel];
    id rettype = [sel rettype];

    eval(rec, env, TypeId, &_rec);
    for(i = 0; i < nargs; i++)
        eval(arg[i], env, [sel argtype:i], &_arg[i]);
    v = _msg(_rec, _sel, _arg[0], _arg[1],
             _arg[2], _arg[3], _arg[4]);
    if(rt == rettype) { /* no need to convert */
        *(id *)r = v;          /* hack, assumes id or equal size */
        return r;
    }
    return [rt fromObject:[rettype toObject:&v] result:r];
}
```

There is some pointer cheating going on here, as the arguments which are to be sent to the receiver object are stored in an array of `id`s, even though they are not necessarily objects. This relies on the fact that, at least on the hardware this code runs upon (a MicroVax II), pointers, long integers, short integers, and characters are all represented as four-byte values when passed to functions.

The `sel` variable is the `Selector` object, and is used to get the number and types of the arguments and the return value of this selector. First `eval` is called recursively to evaluate the receiver of the message; the result type is necessarily `TypeId` since a receiver of a message must be an Objective C object. Each of the argument expressions is evaluated, the result being stored in the `_arg` array. The type of the returned result is that which is expected for this argument in the message about to be sent. The function `_msg` is the low-level message sending function that lies at the heart of Objective C; it is passed a receiver, a selector, and any arguments, and returns the result of sending the message specified by the selector and the arguments to the specified receiver. This result is then converted to the correct type. If this message selector is already known to return the same type as desired, then no conversion is necessary, and the value is simply copied into the correct place. Otherwise, the returned value is first converted to an object (by invoking the `toObject:` method of the known return type) and then converted from an object to the desired return type (via the `fromObject:result:` method). In the typical case, either `rt` or `rettype` is `TypeId`, so one of the conversions to or from an object does no significant work.

The reason for passing the return type to `eval`, rather than having `eval` always return an object, and then converting returned objects to machine integers, characters, and strings when needed, is efficiency. In the current scheme, nested message expressions, where the inner expression returns, say, an integer which is the expected argument type of the outer expression, there is no overhead converting the intermediate result to an object and then immediately back to an integer.

Note that the automatic conversion to objects allows arithmetic to be done relatively painlessly. For example, to add 10 to the x coordinate of a view, use:

```
[[view xloc] Plus:10]
```

The `[view xloc]` returns a machine integer; since this is the intended receiver of the `Plus:` message it must be converted to a `TypeId`, *i.e.* an object, which in this case will be an instance of `TypeInt`. The `Plus:` method expects its argument to be a machine integer; since the interpreter will represent the constant 10 by a `TypeInt` object, it is converted to a machine integer (by calling `eval` with a result type argument of `TypeInt`). The `Plus:` method is then invoked, and it returns a machine integer, which may or may not be converted to a `TypeInt` object depending on the context in which the above program fragment is used.

The above example could be specified more efficiently in the gesture semantics as `[10 Plus:[view xloc]]`. In this case, all the conversions are avoided, since `10` is already represented as an object of `TypeInt`, and `Plus:` expects a machine integer as argument, which is exactly what is returned by `[view xloc]`.

One thing not shown in the above implementation is garbage collection. During expression evaluation, objects are freely being created and discarded, and it is important that the memory associated with them be released when they are discarded. The current implementation of the interpreter does not do this very well, since there is not much point given the lax attitude toward memory management throughout GRANDMA.

Interface Implementation

All the expression nodes are subclasses of `Model`, and each one has a corresponding subclass of `View` to display it on the screen. The expression views act as virtual tools; these tools act on empty argument and receiver slots, as well as the docks and the trash. Implementing the interpreter interface in GRANDMA was a good exercise of the GRANDMA facilities, but is not especially interesting so will not be covered in detail here.

Control Constructs

The only control construct currently implemented is `Seq`, which allows a list of expressions to be evaluated in order. `Seq`, it turns out, was implemented without any extra mechanism in the interpreter; all that was required was the creation of a `Seq` class, whose class methods simply returned their last argument:

```
= Seq:  Object (GRANDMA, Primitive) { }
+ :a1 { return a1; }
+ :a1:a2 { return a2; }
+ :a1:a2:a3 { return a3; }
+ :a1:a2:a3:a4 { return a4; }
+ :a1:a2:a3:a4:a5 { return a5; }
```

Since arguments are evaluated in order, this has the desired effect.

Other control constructs, such as `While` and `If`, have not been implemented, but could easily be implemented if the need arose. One simple implementation technique would to make `WhileExpr` and `IfExpr` both subclasses of `MessageExpr`, and then make `While` and `If` classes which have methods that have the right number of arguments. For simplicity, the normal message expression display code could be used to display `If` and `While` expressions; the only new code to be added would be new `eval:resultType:result:` methods in `WhileExpr` and `IfExpr` which have the desired effect.

Attributes and Cursors

An important consideration in allowing gesture semantics to be specified at runtime is exactly what the application programmer makes visible to the gesture semantics programmer. There are a number of means by which the application programmer can make a feature available to the semantics programmer; all of these hinge on making visible objects which can be the receivers of relevant messages.

The "Attributes" lists provides a way of giving the semantics writer easy access to application objects and features. This is done by creating expressions for each attribute. GRANDMA already supplies entries for all accessible gesture attributes and features.

As an illustrative example of how attributes are specified and implemented, consider the two attributes `handler` and `enclosed`. The `handler` attribute simply refers to the gesture handler that is currently executing. The `enclosed` attribute refers to the list of `View` objects enclosed by the current gesture. Selecting `enclosed` from the attribute list results in a named message; clicking on its box reveals that the message is `[handler enclosed]`.

Internally,

```
        handlerVar = [[VarExpr str:"handler"]
                              vclass:GestureEventHandler];
    /* The above statement adds "handler" to the list of attributes to be displayed
       in the interpreter window, and declared that its value is of type GestureEventHandler.
       Its value is actually set by the GestureEventHandler before any gesture
       semantics are evaluated.  */

        enclosedExpr = [[[[MessageExpr sel:@selector(enclosed)]
                               rec:handlerVar]
                              str:"enclosed"]
                             vclass:OrdCltn];
    /* The above statement adds "enclosed" to the attribute list.  When evaluated
       in gesture semantics, the "enclosed" attribute will result in
       [handler enclosed] being executed.  */
```

Both `handlerVar` and `enclosedExpr` are added to the list of interpreter attributes, and show up in the list as "handler" and "enclosed" respectively. Each of these expressions evaluates to an Objective C object; the `vclass:` message records the expected class of the object. The recorded class is used by the selector browser as a starting point when choosing a message to send to an attribute.

The “handler” attribute, being a `VarExpr`, is evaluated by looking up the string “handler” in the current environment. Section 7.4 described how the environment in which semantic expressions are evaluated is initialized so as the bind `handler` to the current event handler. Evaluating `enclosed` thus results in the `enclosed` message being sent to the current handler:

```
= GestureEventHandler ...
– enclosed { id o, e, seq; int xmin, ymin, xmax, ymax;
     [gesture xmin:&xmin ymin:&ymin xmax:&xmax ymax:&ymax];
     o = [[wall viewdatabase]
        partiallyInRect:xmin:ymin:xmax:ymax];
     for(seq = [o eachElement]; e = [seq next]; )
         if(!  [e isContainedIn:gesture] ) [o remove:e];
     return o;
}
```

The interpreter’s evaluation of the `enclosed` attribute thus results in a call to the above method. This method determines the bounding box of the current gesture, and consults the view database for a list of views contained within this bound. Each object is polled to see if it is enclosed by the gesture, and is removed from the list if it is not. The list is then returned.

The default implementation of `isContainedIn:`, in the `View` class, simply tests if each corner of the bounding box is enclosed within the gesture. This test may be overridden by non-rectangular views, or rectangular views that wish to ensure its each edge is entirely contained within the gesture.

```
= View ...
– (BOOL)isContainedIn:g {
     int x1, y1, x2, y2; [self _calc_new_box];
     x1 = [box left]; y1 = [box top];
     x2 = [box right]; y2 = [box bottom];
     return [g contains:x1:y1] && [g contains:x1:y2] &&
            [g contains:x2:y1] && [g contains:x2:y2];
}
```

The `Gesture` class implements the `contains::` message, which tests if a point is enclosed within the gesture. The current implementation first closes the gesture by conceptually connecting the ending point to the starting point, and then counts the number of times a line from the point to a known point outside the gesture crosses the gesture. An odd number of crossings indicates that the point is indeed enclosed by the gesture.

Other attributes work similarly, although their code tends to be much simpler than that of `enclosed`. In particular, there are attributes for each feature discussed in Section 3.3; the attributes are named messages implemented as `[[handler gesture] ifvi:N]`, where `N` is the corresponding index into the feature vector.

Cursors are added to the list of cursors available for use in semantic expressions simply by sending them the `public` message. The application programmer should create and make available any cursor that might prove useful to the semantics writer.

7.8 Conclusion

The gesture subsystem of GRANDMA consists of the gesture event handler, the low level gesture recognition modules, the user interface which allows the modification of gesture handlers, gesture examples, and gesture classes, and the interpreter for evaluating the semantics of gestures. Each of these parts has been discussed in detail. The next chapter demonstrates how GRANDMA is used to build gesture-based applications.

Chapter 8

Applications

This chapter discusses three gesture-based applications built by the author. The first, GDP, is a simple drawing editor based on the drawing program DP [42]. The second, GSCORE, is an editor for musical scores. The third, MDP, is an implementation of the GDP drawing editor that uses multi-finger gestures.

GDP and GSCORE are both written in Objective C, and run on a DEC MicroVAX II. They are both gesture-based applications built using the GRANDMA system, discussed in Chapters 6 and 7. As such, the gestures used are all single-path gestures drawn with a mouse. GRANDMA interfaces to the X10 window system [113] through the GDEV interface written by the author. GDEV runs on several different processors (MicroVAX II, SUN-2, IBM PC-RT), and several different window managers (X10, X11, Andrew). GRANDMA, however, only runs on the MicroVax, which for years was the only system available to the author that ran Objective C. It should be relatively straightforward to port GRANDMA to any UNIX-based environment that ran Objective-C, though to date this has not been done.

MDP is written in C (not Objective C), and runs on a Silicon Graphics IRIS 4D Personal Workstation. MDP responds to multiple-finger gestures input via the Sensor Frame. Unlike GDP and GSCORE, MDP is not built on top of GRANDMA. The reason for this is that the only functioning Sensor Frame is attached to the above-mentioned IRIS, for which no Objective C compiler exists. It would be desirable and interesting to integrate Sensor Frame input and multi-path gesture recognition into GRANDMA (see Section 10.2).

8.1 GDP

GDP, a gesture-based drawing program, is based on DP [42]. In DP there is always a *current mode*, which determines the meaning of mouse clicks in the drawing window. Single letter keyboard commands or a popup menu may be used to change the current mode. The current mode is displayed at the bottom of the drawing window, as are the actions of the three mouse buttons. For example, when the current mode is “line”, the left mouse button is used for drawing horizontal and vertical lines, the middle button for arbitrary lines, and the right button for lines which have no gravity. Some DP commands cause dialogue boxes to be displayed; this is useful for changing parameters such as the current thickness to use for lines, the current font to use for text, and so on.

With the gesture handlers turned off, GDP (loosely) emulates DP. The current mode is indicated by the cursor. For example, when the "line" cursor is displayed, clicking a mouse button in the drawing window causes a new line to be created and one endpoint to be fixed at the position of the mouse. As long as the mouse button is held down, the other end of the line follows any subsequent motion of the mouse, in a "rubberband" fashion. The user releases the mouse button when the second endpoint of the line is at the desired location.

Both DP and GDP support *sets*, whereby multiple graphic objects may be grouped together and subsequently function as a single object. Once created, a set is translated, rotated, copied, and deleted as a unit. A set may include one or more sets as components, allowing the hierarchical construction of drawings. In DP, there is the "pack" command, which creates a new set from a group of objects selected by the user, and the "unpack" command, whereby a selected set object is transformed back into its components. GDP functions similarly, though the selection method differs from DP.

GDP makes no attempt to emulate every aspect of DP. In particular, the various treatments of the different mouse buttons are not supported. These and other features were not implemented since doing so would be tangential to the purpose of the author, which was to demonstrate the use of gestures. As the unimplemented features present no conceptual problems for implementation in GRANDMA, the author chose not to expend the effort.

8.1.1 GDP's gestural interface

GDP's gesture-based operation has already been briefly described in Section 1.1. That description will be expanded upon, but not repeated, here.

Figures 1.2a, b, c, and d show the rectangle, ellipse, line, and pack gestures, all of which are directed at the GDP window, rather than at graphic objects. Also in this class is the text gesture, a cursive "t", and the dot gesture, entered by pressing the mouse button with no subsequent mouse motion. The text gesture causes a text cursor to be displayed at the initial point of the gesture. The user may then enter text via the keyboard. The dot gesture causes the last command (as indicated by the current mode) to be repeated. For example, after a delete gesture, a dot gesture over an existing object will cause that object to be deleted.

Figures 1.2e, f, and g show the copy, rotate, and delete gestures, all of which act directly on graphic objects. The move gesture, a simple arrow (figure 8.1), is similar. All of these gestures act upon the graphic object at the initial point of the gesture. These gestures are also recognized by the GDP window when not begun over a graphic object. In this case, the cursor is changed to indicate the corresponding mode, and the underlying DP interface takes over. In particular, dragging one of these cursors over a graphic object causes the corresponding operation to occur.

8.1.2 GDP Implementation

Since GDP was built on top of GRANDMA, the implementation followed the MVC paradigm. Figure 8.2 shows the position in the class hierarchy for the new classes defined in GDP.

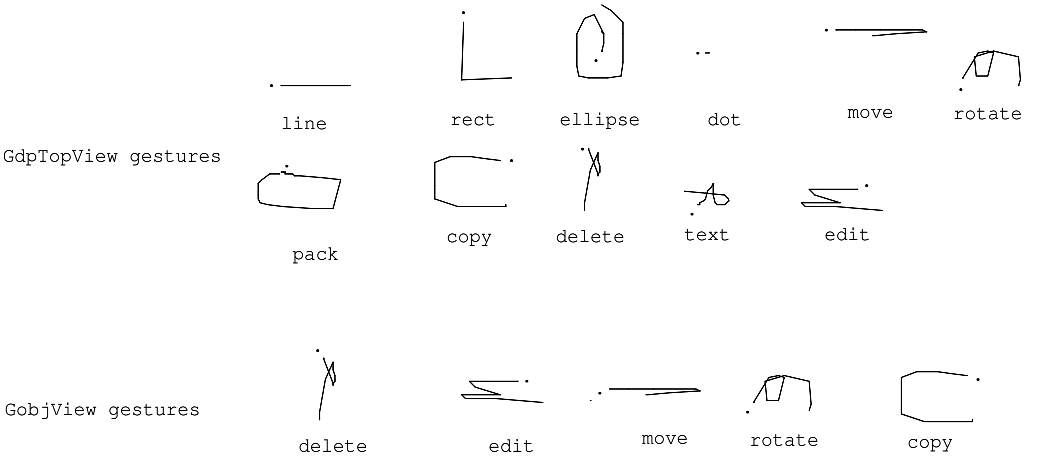


Figure 8.1: GDP gestures

As always, the period indicates the start of the gesture.

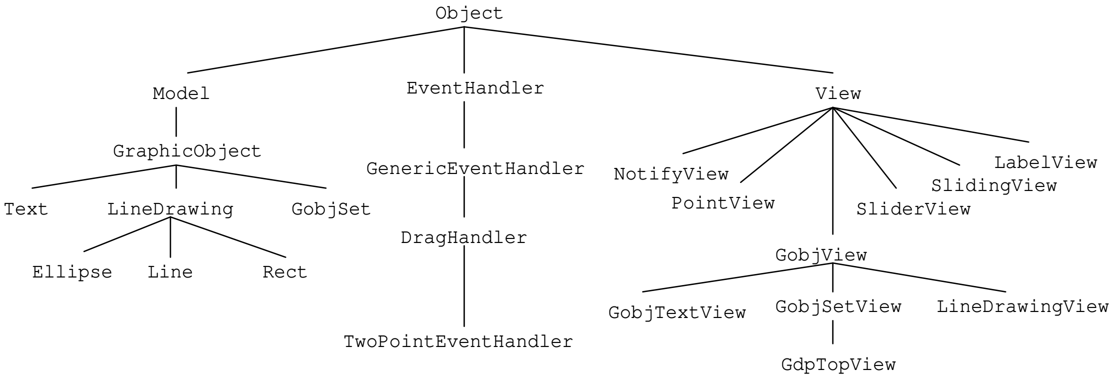


Figure 8.2: GDP's class hierarchy

8.1.3 Models

The implementation of GDP centers on the class `GraphicObject`, a subclass of `Model`. Each component of the drawing is a `GraphicObject`. The entire drawing is also implemented as a graphic object. `GraphicObjects` are either `Text` objects, `LineDrawing` objects (lines, rectangles, and ellipses), or `GobjSet` objects, which implement the set concept.

A `GraphicObject` has two instance variables: `parent`, the `GobjSet` object of which this object is a member, and `trans`, a transformation matrix [101] for mapping the object into the drawing. Every `GraphicObject` is a member of exactly one set, be it the set which represents the entire drawing (these are top level objects), or a member of a set which is itself part of the drawing.

`LineDrawing` objects have a single instance variable, `thickness`, that controls the thickness of the lines used in the line drawing. The three subclasses of `LineDrawing`, namely `Line`, `Rectangle`, and `Ellipse`, represent all graphics in the drawing. Associated with each `LineDrawing` subclass is a list of points which specify a sequence of line segments for drawing the object. The points in the list are normalized so that one significant point of the object lies on the origin and another significant point is at point (1,1). For `Lines`, one endpoint is at (0,0) and the other at (1,1). The point list for `Rectangles` specifies a square with corners at (0,0), (0,1), (1,1), and (1,0). The `Ellipse` is represented by 16 line segments that approximate a circle with center (0,0) and that passes through the point (1,1). The transformation matrix in each `LineDrawing` object is used to map the list of points in each `LineDrawing` object into drawing (window) coordinates.

A `GobjSet` object contains a `Set` of objects that make up the set. In order to display a set, the transformation matrix of the set is composed with (multiplied by) that of each of the constituent objects. This composition happens recursively, so that deeply nested objects are displayed correctly.

`Text` objects contain a font reference and text string to be displayed.

8.1.4 `Views`

Each of the immediate subclasses of `GraphicObject` has a corresponding subclass of `GobjView` associated with it. Each `LineDrawingView` object is responsible for displaying the `LineDrawing` object which is its model on the screen. Similarly, `GobjTextViews` display `Text` objects, and `GobjSetViews` display `GobjSets`.

All `GobjViews` respond to the `updatePicture` message in order to redraw their picture appropriately. A `LineDrawingView` simply asks its model for the lists of points (suitably transformed) which it proceeds to connect via lines. The model also provides the appropriate thickness of the lines as well. (Note that it is not necessary to provide view classes for the three subclasses of `LineDrawing` since all three classes are taken care of by `LineDrawingView`.)

`GobjTextViews` draw their models one character at a time in order to accommodate the transformation of the model. Transformations which have a unit scale factor (no shrinking or dilation) and no rotation component cause the text to be drawn horizontally, with the characters spacing determined by their widths in the current font. In the current implementation, scaling or rotation does not effect the character size or orientations (as X10 will not rotate or scale characters), but does effect the character positions.

`GobjSetViews` have the views of their model's component objects as subviews. Since the `update` method for `View` will automatically propagate `update` messages to subviews, no `updatePicture` method is required for `GobjSetView`.

The `GobjView` class overrides the `move::` method (of `View`). Recall from Section 6.6 that this method simply changes the location of the view, thus translating the view in two dimensions. This method is used, for example, by the drag handler (section 6.7.9) to cause views to move with the mouse cursor. The purpose of overriding the default method is so that dragging any `GobjView` causes its model to be changed so as to reflect the new coordinates of the object in the drawing. The model is changed by first sending it the message `getLocalTrans`, which returns the model's transformation matrix, then calling a function which modifies the matrix to reflect the additional translation, and then sending the model a `setLocalTrans:` message, which causes the new transformation matrix to be recorded in the model. Of course the model then sends itself the `modified` message which causes the model's view to redraw the model at its new location.

`GobjView` also implements the `delete` message, by first sending itself the `free` message (which, among other things, removes it from its parent's subview list), and then sending its model the `delete` message. `GobjView` also overrides the default `isOver:` and `isContainedIn:` methods (Sections 6.7.5 and 7.7.3) so that they always return `NO` for objects not at the top-level of the drawing. Each subclass of `GobjView` implements `isReallyOver:` and `isReallyContainedIn:`, which are invoked when the object is indeed top-level.

The outermost window is itself a view. It is an instance of `GdpTopView`, which is a subclass of `GdpSetView`. The `GdpTopView` representing the entire drawing.

8.1.5 Event Handlers

GDP required the addition of one new event handler, `TwoPointEventHandler`, which is of sufficient utility and generality to be incorporated into the standard set of GRANDMA event handlers. The purpose of the `TwoPointEventHandler` is to implement the typical "rubberbanding" interaction. For example, clicking the "line" cursor in the drawing window causes a new line to be created, one endpoint of which is constrained to be at the location of the click, the other endpoint of which stays attached to the cursor until the mouse button is released. A `TwoPointEventHandler` can be used to produce this behavior.

As a `GenericEventHandler`, a `TwoPointEventHandler` has a parameterizable starting predicate, handling predicate, and stopping predicate (Section 6.7.8). In order for a passive `TwoPointEventHandler` to be activated, the tool of the activating event must operate on the view to which the handler is attached (like a `GenericToolOnViewHandler`, section 6.7.7). If the tool operates on the view and the event satisfies the starting predicate, the handler is activated. When activated, the tool is allowed to operate on the view, and the operation is expected to return an object which is to be the receiver of subsequent messages. In the above example, the "line" tool operates upon the drawing window view (a `GdpTopView`) the result of which is a newly created `Line` object. The handler then sends the new object a message whose parameters are the starting event location coordinates. The actual message sent is a parameter to the passive event handler; in the example the message is `setEndpoint0::`. Each subsequent event handled results in the new object being sent another message containing the coordinates of the event (`setEndpoint1::` in the example).

8.1.6 Gestures in GDP

This section describes the addition of gestures to the implementation described above. The gesture handlers, gesture classes, example gestures, and gesture semantics were all added at runtime, allowing them to be tested immediately. I should admit that in several cases it was necessary to add some features directly to the existing C code and recompile. This was partly due to the fact that GRANDMA's gesture subsystem was being developed at the same time as this application, and partly due to the gesture semantics wanting to access models and views through methods other than ones already provided, for reasons such as readability and efficiency.

Figure 8.1 shows the gesture classes recognized by each of the two GDP gesture handlers. Note that the gestures expected by a `GobjView` are a subset of those expected by a `GdpTopView`. Allowing one gesture class to be recognized by multiple handlers allows the semantics of the gesture to depend upon the view at which it is directed.

Several gestures (line, rect, ellipse, and text) cause graphic objects to be created. These gestures are only recognized by the top level view, which covers the entire window, a `GdpTopView`. When, for example, a line gesture (a straight stroke) is made, a line is created, the first endpoint of which is at the gesture start, while the second endpoint tracks the mouse in a rubberband fashion.

The semantics for the line gesture are:

```
recog = [Seq :[handler mousetool:createLine_MouseTool]
             :[[topview createLine] translateEndpoint:0
                                     x:<startX> y:<startY>] ];
manip = [recog scaleXYEndpoint:1 x:<currentX> y:<currentY>
                                cx:<startX>  cy:<startY>];
```

(The `done` expression is assumed to be `nil`.) When the line gesture is recognized, the gesture handler is sent the `mousetool:` message, passing the `createLine_MouseTool` as a parameter. The handler sends a message to its view's wall, and the cursor shape changes. (Internally, the handler changes its `tool` instance variable to the new tool, as well.) Then, a line is created (via the `createLine` message sent to the top view), and the new line is sent a message which translates one endpoint to the starting point of the gesture. (The identifiers enclosed in angle brackets are gestural attributes, as discussed in Section 7.7.3.) The `::` message to `Seq`, which is used evaluate two expressions sequentially, returns its last parameter, in this case the newly created line, which is assigned to `recog`.

Upon each subsequent mouse input the `manip` expression is evaluated. It sends the new line (referred to through `recog`) a message to scale itself, keeping the "center" point (`startX`, `startY`) in the same location, mapping the other endpoint to (`currentX`, `currentY`).

The semantics for the rect and ellipse gestures are similar to those of line, the only difference being the resultant cursor shape and the creation message sent to `topview`. The start of the rectangle gesture controls one corner of the rectangle and subsequent mouse events control the other corner. The start of the ellipse gesture determines the center of the ellipse, and the scaling guarantees that the mouse manipulates a point on the ellipse. The rectangle is created so that its sides are parallel to the window. Similarly, the ellipse is created so that its axes are horizontal and vertical. Manipulations after any of the creation gestures is recognized never effect the orientation of the created object. With only a single mouse position for continuous control (two degrees of freedom) it is impossible to independently alter the orientation angle, size, and aspect ratio of the graphic object. The design choice was made to modify only the size and aspect ratio in the creation gesture; a rotate gesture may subsequently be used to modify the orientation angle.

It is still possible, however, to use other features of the gesture to control additional attributes of the graphic object. Changing the `recog` semantics of a line gesture to

```
recog = [Seq :[handler mousetool:<createLine>]
             :[[[topview createLine] translateEndpoint:0
                                     x:<startX> y:<startY>]
               thickness:[[pathLength DividedBy:40]
                                Clip:1 :9] ]  ];
```

causes the thickness of the line to be the length of the gesture divided by 40 and constrained to be between 1 and 9 (pixels) inclusive. The length of the gesture determines the thickness of the newly created line, which can subsequently be continuously manipulated into any length.

The dot gesture (where the user simply presses the mouse without moving it) has the null semantics. When it is recognized, the gesture handler turns itself off immediately, enabling events to propagate past it, and thus allowing whatever cursor is being displayed to be used as a tool. Thus GDP, like DP, has the notion of a current mode, accessible via the dot gesture.

The pack gesture has semantics:

```
recog = [Seq :[handler mousetool:pack_MouseTool]
             :[topview pack_list:<enclosed>]];
```

The attribute `<enclosed>` is an alias for `[handler enclosed]`. Recall from Section 7.7.3 that this message returns a list of objects enclosed by the gesture. This list is passed to the `topview`, which creates the set. As long as the mouse button is held down, the pack tool will cause the `pack` message to be sent to any object it touches; those objects will execute `[parent pack:self]` (the implementation of the `pack` method) to add themselves to the current set.

The copy, move, rotate, edit, and delete gestures simply bring up their corresponding cursors when aimed at the background (`GdpTopView`) view. They have more interesting semantics when associated with a `GobjView`. The copy gesture, for example, causes:

```
recog = [Seq :[handler mousetool:viewcopy_MouseTool]
             :copy = [[view viewcopy]
                          move:<endX> :<endY>]
             :lastX = <endX>
             :lastY = <endY>]
manip = [Seq :[copy move:[<currentX> Minus:lastX]
                        :[<currentY> Minus:lastY]]
             :lastX = <endX>
             :lastY = <endY>]
```

This illustrates that the gesture semantics can mimic the essential features of the `DragHandler` (Section 6.7.9). The semantics of the move gesture are almost identical, except that no copy is made. A simpler way to do this kind of thing (by reraising events) is shown when the semantics of the GSCORE program are discussed.

The delete gesture has semantics

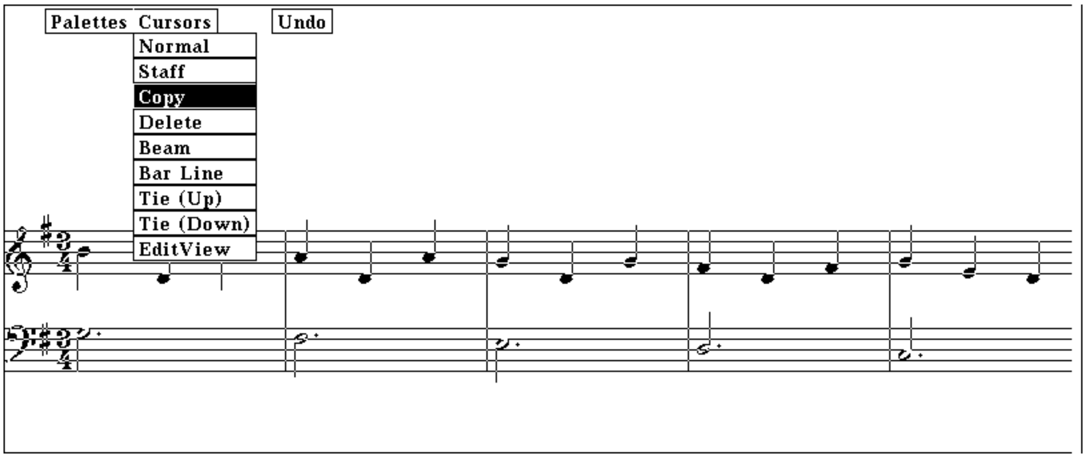


Figure 8.3: GSCORE's cursor menu

```
recog = [Seq :[handler mousetool:delete_Mousetool]
             :[view delete]];
```

The edit gesture semantics are similar.

The rotate gesture has semantics:

```
recog = nil;
manip = [Seq :[handler mousetool:rotate_MouseTool]
             :[view rotateAndScaleEndpoint:0
                    x:<currentX>
                    y:<currentY>
                    cx:<startX>
                    cy:<startY>]];
```

The `rotateAndScaleEndpoint:` message causes one point of the view to be mapped to the coordinate indicated by `x:` and `y:` which keeping the point indicated by `cx:` and `cy:` constant. This gesture always drags endpoint 0 of a graphic object. It would be better to be able to drag an arbitrary point, as is done by MDP, discussed later.

8.2 GSCORE

GSCORE is a gesture-based musical score editor. Its design is not based on any particular program, but its gesture set was influenced by the SSSP score-editing tools [18] and the Notewriter II score editor.

8.2.1 A brief description of the interface

GSCORE has two interfaces, one gesture-based, the other not. Figure 8.3 shows the non-gesture-based interface in action. Initially, a staff (the five lines) is presented to the user. The user may call

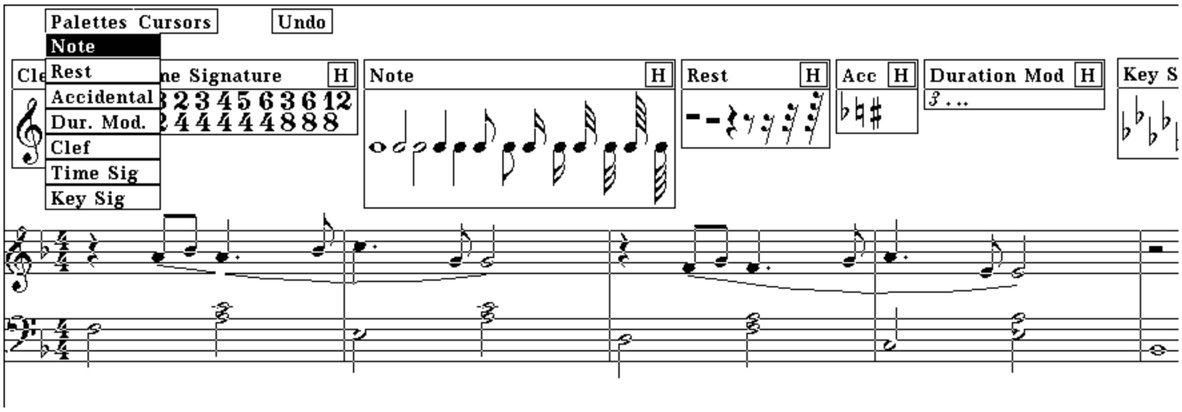


Figure 8.4: GSCORE's palette menu

up additional staves by accessing the staff tool in the "Cursors" menu (which is shown in the figure). In figure 8.4, the user has displayed a number of palettes from which he can drag musical symbols onto the staff. As can be seen, the user has already placed a number of symbols on the staff. The user has also used the down-tie tool to indicate two phrases and the beam tool to add beams so as to connect some notes.[1] Both tools work by clicking the mouse on a starting note, then touching other notes. The tie tool adds a tie between the initial note and the last one touched, while the beam tool beams together all the notes touched during the interaction.

Dragging a note onto the staff determines its starting time as follows: If a note is dragged to approximately the same x location as another note, the two are made to start at the same time (and are made into a chord). Otherwise, the note begins at the ending time of the note (or rest or barline) just before it. Other score objects are positioned like notes.

The palettes are accessed via the palette menu, shown in figure 8.4. The palettes themselves may be dragged around so as to be convenient for the user. The "H" button hides the palette; once hidden it must be retrieved from the menu.

The delete cursor deletes score events. When the mouse button is pressed, dragging the delete button over objects which may be deleted causes them to be highlighted. Releasing the button over such a highlighted object causes it to be deleted. Individual chord notes may be deleted by clicking on their note heads; an entire chord by clicking on its stem. When a beam is deleted, the notes revert to their unbeamed state.

The gestural interface provides an alternative to the palette interface. Figure 8.5 shows the three sets of gestures recognized by GSCORE objects. The largest set, associated with the staff, all result

---

[1] Note to readers unfamiliar with common music notation: A tie is a curved line connecting two adjacent notes of the same pitch. A tie indicates that the two connected notes are to be performed as a single note whose duration equals the sum of those of the connected notes. A curved line between adjacent differently pitched notes is a slur, performed by connecting the second note to the first with no intermediate breath or break. Between nonadjacent notes, the curved line is a phrase mark, which indicates a group of notes that makes up a musical phrase, as shown in figure 8.4. In GSCORE, the tie tool can be used to enter ties, slurs, and phrase marks. A beam is a thick line that connects the stems of adjacent notes (again see figure 8.4). By grouping multiple short notes together, beams serve to emphasize the metrical (rhythmic) structure of the music.

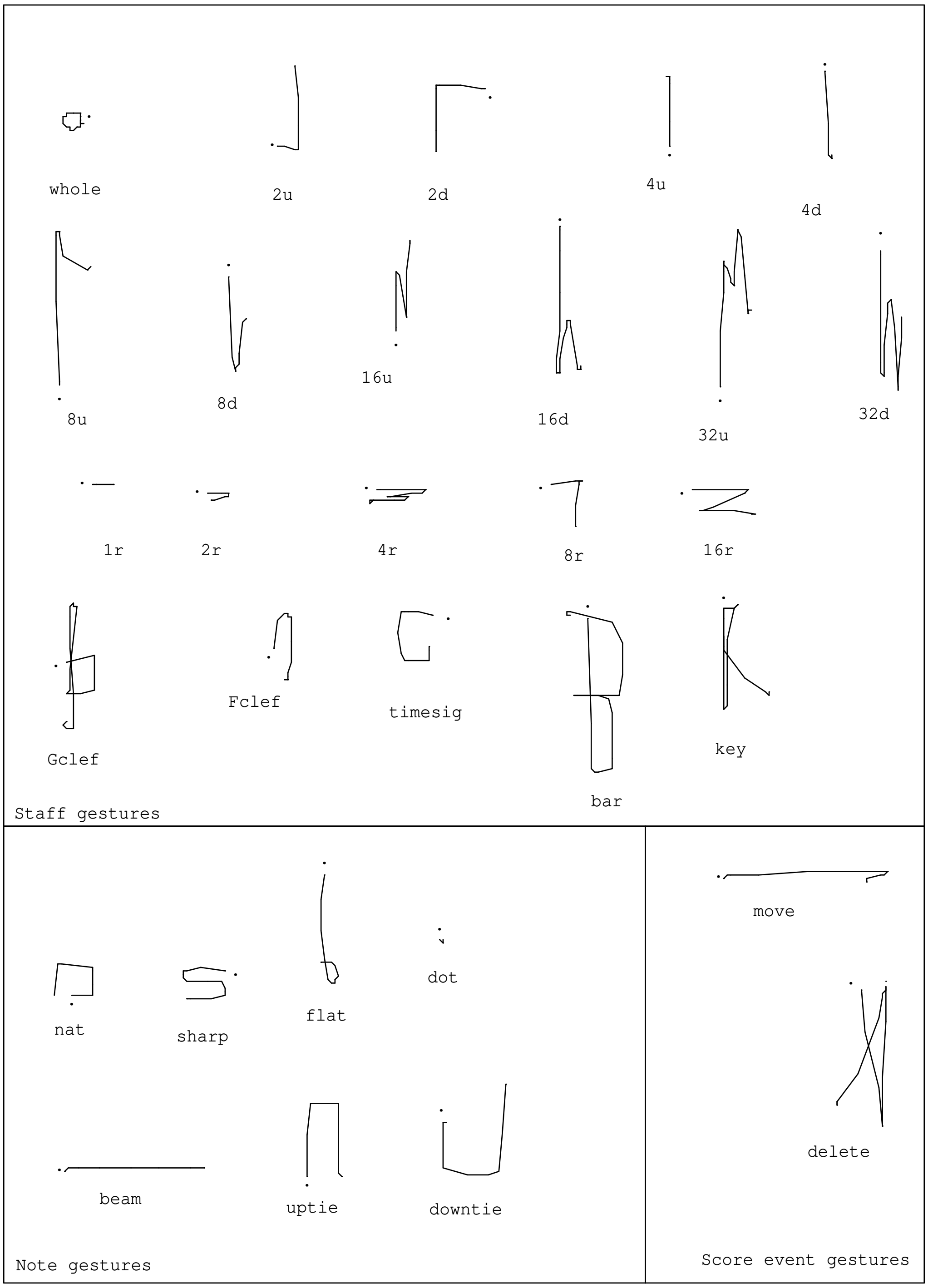


Figure 8.5: GSCORE gestures

in staff events being created. There are two gestures, move and delete, that operate upon existing score events. Seven additional gestures are for manipulating notes.

A gesture at a staff creates either a note, rest, clef, bar line, time signature, or key signature object. The object created will be placed on the staff at (or near) the initial point of the gesture. For notes, the x coordinate determines the starting time while the y coordinate determines the pitch class. The gesture class determines the actual note duration (whole note, half note, quarter note, eight note, sixteenth note, or thirtysecond note) and the direction of the stem.

Like note gestures, the remaining staff gestures use the initial x coordinate to determine the staff position of the created object. The five rest gestures generate rests of various durations. The two clef gestures generate the F and G clefs (C clefs may only be dragged from the palette). The timesig gesture generates a time signature. After the gesture is recognized, the user controls the numerator of the time signature by changes in the x coordinate of the mouse, and the denominator by changes in y. Similarly, after the key gesture is recognized, the user controls the number of sharps or flats by moving the mouse up or down. When a bar gesture is recognized, a bar line is placed in the staff, and the cursor changes to the bar cursor. While the mouse button is held, the newly created bar line extends to any staff touched by the mouse cursor.

The note-specific gestures all manipulate notes. Accidentals are placed on the note using the sharp, flat, and natural gestures. The beam gesture causes the notes to be beamed together. The note on which the beam gesture begins is one of the beamed notes; the beam is extended to other notes as they are touched after the gesture is recognized. The uptie and downtie gestures operate similarly. The dot gesture causes the duration of the note to be multipled by $\frac{3}{2}$, typically resulting in a dot being added to a note.

Since a note is a score event, and always exists on a staff, a gesture which begins on a note may either be note specific (*e.g.* sharp), score-event specific (*e.g.* delete), or directed at the staff (*e.g.* one of the note gestures). The first time a gesture is made at a note, the three gesture sets are unioned and a classifier created that can discriminate between each of them, as described in Section 7.2.

Figure 8.6 shows an example session with GSCORE.

8.2.2 Design and implementation

Figure 8.7 shows where the classes defined by GSCORE fit into GRANDMA's class hierarchy. In general, each model class created has a corresponding view class for displaying it. No new event handlers needed to be created for GSCORE; GRANDMA's existing ones proved adequate.

Generally useful views

Two new views of general utility, `PullDownRowView` and `PaletteView`, were implemented during the development of GSCORE. A `PullDownRowView` is a row of buttons, each of which activates a popup menu. It provides functionality similar to the Macintosh menu bar. A `PaletteView` implements a palette of objects, each of which is copied when dragged. `PaletteView` instantiates a single `DragHandler` (Section 6.7.9) that it associates with every object on a palette. The drag handler has been sent the message `copyviewON`, which gives the palette its functionality.

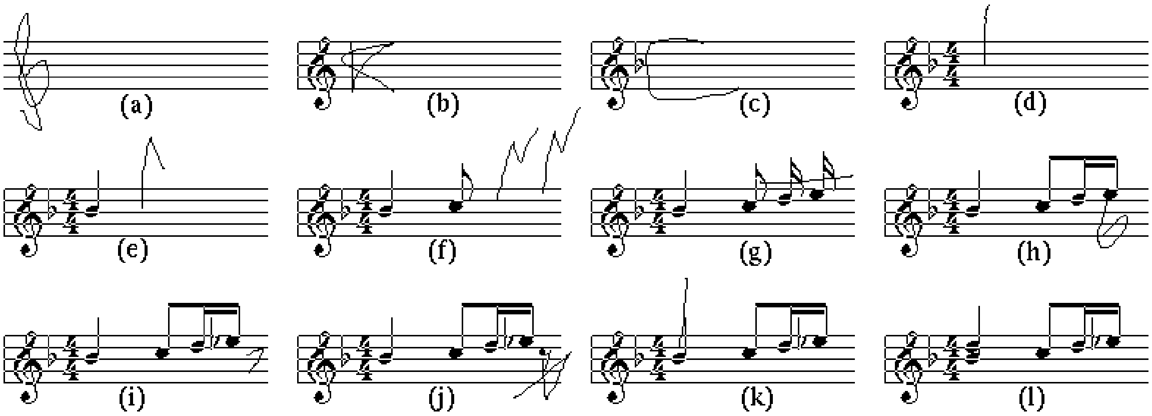


Figure 8.6: A GSCORE session

Panel (a) shows a blank staff upon which the Gclef *gesture has been entered. Panel (b) show the created treble clef, and a* key *(key signature) gesture. After recognition, the number of flats or sharps can be manipulated by the distance the mouse moves above the staff or below the staff, respectively. Panel (c) shows the created key signature (one flat), and a* timesig *(time signature) gesture. After recognition, the horizontal distance from the recognition point determines the numerator of the time signature, and the vertical distance determines the denominator. Panel (d) shows the resulting time signature, and the* 4u *(quarter note) gesture, a single vertical stroke. Since this is an upstroke, the note will have an upward stem. The initial point of the gesture determines both the pitch of the note (via vertical position) and the starting time of the note (via horizontal position). Panel (e) shows the created note, and the* 8u *(eighth note) gesture. Like the quarter note gesture, the gesture class determines the note's duration, and gestural attributes determines the note's stem direction, start time and pitch. Panel (f) shows two* 16u *(sixteenth note) gestures (combining two steps into one). Panel (g) shows a* beam *gesture. This gesture begins on a note, rather than the gestures mentioned thus far, which begin on a staff. After the gesture is recognized, the user touches other notes in order to beam them together. Panel (h) shows the beamed notes, and a* flat *gesture drawn on a note. Panel (i) shows the resulting flat sign added before the note, and an* 8r *(eighth rest) gesture drawn on the staff. Panel (j) shows the resulting rest, and a* delete *gesture beginning on the rest. Panel (k) shows a* 4u *(quarter note) gesture drawn over an existing quarter note (all symbols in GSCORE have rectangular input regions), the result being a chord, as shown in panel (l).*

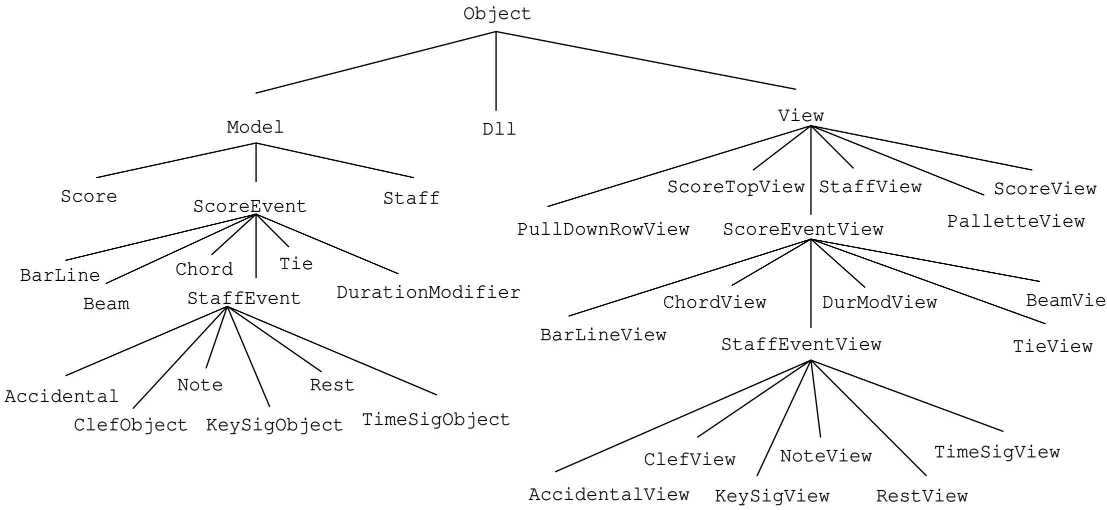


Figure 8.7: GSCORE's class hierarchy

Each palette can implement an arbitrary action when one of the dragged objects is dropped. For most palettes of score events (notes, rests, clefs, and so on), no special action is taken. The copied view becomes a subview of a `StaffView` when dragged onto a staff. However, accidentals and duration modifiers (dots and triplets) are tools which send messages to `NoteView` objects when dragged over them; the `NoteView` takes care of updating its state and creating any accidentals or duration modifiers it needs. The copies that are dragged from the palette thus never become part of the score, and so are automatically deleted when dropped.

GSCORE Models

With the exception of `PullDownRowView` and `PaletteView`, the new classes created during the implementation of GSCORE are specific to score editing. A `Score` object represents a musical score. It contains a list of `Staff` objects and a doubly-linked list (class `Dll`) of `ScoreEvent` objects. Each `ScoreEvent` has a `time` field indicating where in the score it begins; the doubly-linked list is maintained in time order.

The subclass `StaffEvent` includes all classes that can only be associated with a single staff. A `BarLine` is not a `StaffEvent` since it may connect more than one staff, and thus maintains a `Set` of staves in an instance variable. Similarly, a `Chord` may contain notes from different staves, as may a `Tie` and `Beam`. A `DurationModifier` is not attached directly to a `Staff`, but instead with a `Note` or `Beam`, so it is not a `StaffEvent` either.

The responsibility of mapping time to *x* coordinate in a staff rests mainly with the `Score` object. It has two methods `timeOf:` and `xposOf:` which map *x* coordinates to times, and times to *x* coordinates, respectively. `Score` has the method `addEvent:` for adding events to the list and `delete:` and `erase:` for deleting and erasing events. Erase is a kind of "soft" delete; the object is removed from the list of score events, but it is not deallocated or in any other way disturbed. A typical use would be to erase an object, change its `time` field, and then add it to the score, thus moving it in time.

Each `ScoreEvent` subclass implements the `tiebreaker` message; this orders score events that occur simultaneously. This is important for determining the position of score events; bar lines must come before clefs, which must come before key signatures, and so on. Besides determining the order events will appear on the staff, tiebreakers are important because they maintain a canonical ordering of score events which can be relied upon throughout the code.

Particular `ScoreEvent` classes have straightforward implementations. `Note` has instance variables that contain its pitch, raw duration (excluding duration modifiers), actual duration, stem direction, back pointers to any `Chord` or `Beam` that contain it, and pointers to `Accidental` and `DurationModifier` objects that apply to it. It has messages for setting most of those, and maintains consistency between dependent variables. Notes are able to delete themselves gracefully, first by removing themselves from any beams or chords in which they participate, and deleting any accidentals or duration modifiers attached to them, then finally deleting themselves from the score. Other score events behave similarly.

Sending a `ScoreEvent` the `time:` message, which changes its start time, results in its `Score` being informed. The score takes care to move the `ScoreEvent` to the correct place in its list of events. This is accomplished by first erasing the event from the score, and then adding it again.

While the internal representation of scores for use in editing is quite an interesting topic in its own right [20, 83, 88, 29] it is tangential to the main topic, gesture-based systems. The representation has now been described in enough detail so that the implementation of the user interface, as well as the gesture semantics, can be appreciated. These are now described.

GSCORE Views

As expected from the MVC paradigm, there is a `View` subclass corresponding to each of the `Models` discussed above. `ScoreView` provides a backdrop. Not surprisingly, instances of `StaffView` are subviews of `ScoreView`. Perhaps more surprisingly, all `ScoreEventView` objects are also subviews of `ScoreView`. For simplicity, the various `StaffEventView` classes are not subviews of the `StaffView` upon which they are drawn. This simplifies screen update, since the `ScoreView` need not traverse a nested structure to search for objects that need updating.

It is often necessary for a view to access related views; for example a `BeamView` needs to communicate with the `NoteView` or `ChordView` objects being beamed together. One alternative is for the views to keeps pointers to the related views in instance variables. This is very common in MVC-based systems: pointers between views explicitly mimic relations between the corresponding models. It is the task of the programmer to keep these pointers consistent as the model objects are added, deleted, or modified.

In one sense, this is one of the costs associated with the MVC paradigm. For reasons of modularity, MVC dictates that views and models be separate, and that models make no reference to their views (except indirectly, through a model's list of dependents). The benefit is that models may be written cleanly, and each may have multiple views. Unfortunately, the separation results in redundancy at best (since the structure is maintained as both pointers between models and pointers between views), and inconsistency at worse (since the two structures can get "out of sync"). Also, any changes to a model's relationship to other models requires parallel changes in the corresponding views. This duplication, noticed during the initial construction of GSCORE, seemed to be contrary to the ideals of object-oriented programming, where techniques such as inheritance are utilized to avoid duplication of effort.

GRANDMA attempts to address this problem of MVC in a general way. The problem is caused by the taboo which prevents a model from explicitly referencing its view(s). GRANDMA maintains this taboo, but provides a mechanism for inquiring as to the view of a given model. In order to retain the possibility of multiple views of a single model, the query is sent to a *context object*; within the context, a model has at most one view. The implementation requires that a context be a kind of `View` object:

```
View ...
− setModelOfView:v { /* associates v with [v model] */ }
− getViewOfModel:m { /* returns view associated with m */ }
```

The implementation is done using an association list per context: given a context, the message `setModelOfView:` associates a view with its model in the context. Objective C's association list object uses hashing internally, so `getViewOfModel:` typically operates in constant time independent of the number of associations. The result is a kind of inverted index, mapping models to views.

In GSCORE, only a single context is used (since there is only one view per model), which, for convenience, is the parent of all `ScoreEventView` objects, a `ScoreView`. The various subclasses of `ScoreEventView` no longer have to keep consistent a set of pointers to related objects. For example, a `BeamView` needs only to query its model for the list of `Note` and/or `Chord` models that it is to beam together; it can then ask each of those models `m` for its view via `[parent getViewOfModel:m]`. The instance variable `parent` here refers to the `ScoreView` of which the `BeamView` is a subview. Thus, the problem of keeping parallel structures consistent is eliminated. One drawback, however, is that it is now necessary to maintain the inverted index as views are created and deleted.

Now that the problem of how views access their related views has been solved, redisplaying a view is straightforward. Recall (Section 6.5) that when a model is modified, it sends itself the `modified` message, which results in all its dependents (in particular its view) getting the message `modelModified`. The default implementation of `modelModified` results in `updatePicture` being sent to the view and all of its subviews (Section 6.6). Normally, `updatePicture` is the method that is directly responsible for querying the model and updating the graphics. `ScoreEventView` overrides `updatePicture`, and the task of actually producing the graphics for a score event is relegated to a new method, `createPicture`, implemented by each of `ScoreEventView`'s subclasses. `ScoreEventView`'s `updatePicture` sends itself `createPicture`, but also does some additional work to be discussed shortly.

As an example, consider what happens when the pitch of a note is changed. When a `Note` is sent the `abspitch:` message, which changes its pitch, it updates its internal state and sends itself the `modified` message. (Changing the pitch might result in `Accidental` objects being added or deleted from the score, a possibility ignored for now.) This `Note`'s `NoteView` will get sent `createPicture`, and query its model (and the `Score` and `Staff` objects of the model) to determine the kind and position of the note head, as well as the stem direction, if needed. The proper note head is selected from the music font, and drawn on the staff (with ledger lines if necessary) at the determined location.

One reason for `ScoreEvent`'s `updatePicture` sending `createPicture` is to test in a single place the possibility that the view may have moved since the last time it was drawn. In particular, if the x coordinate of the right edge of the view's bounding box has changed, this is an indication that the score events after the this might have to be repositioned. If so, the `Score` object is sent a message to this effect, and takes care of changing the x position of any affected models. Another reason for the extra step in creating pictures is to stop a recursive message that attempts to create a picture currently being created, a possibility in certain cases.

Adding or deleting a `ScoreEvent` causes the `Score` object to send itself the `modified` message. Before doing so, it creates a record indicating exactly what was changed. When notified, its `ScoreView` object will request that record, creating or deleting `ScoreEventViews` as required. `ScoreView` uses an association list to associate view classes with model classes; it can thus send the `createViewOf:` message to the appropriate factory.

`ScoreEventViews` function as virtual tools, performing the action `scoreeventview:`. (This default is overridden by `AccView`, `DurModView`, `BarLineView`, and `TieView`, as these do not operate on `StaffViews`.) The only class that handles `scoreeventview:` messages is `StaffView`. A version of `GenericToolOnViewEventHandler` different than the one discussed in Section 6.7.7 is associated with class `ScoreEventView`. This version is a kind of `GenericEventHandler`, and thus more parameterizable than the one discussed earlier. The instance associated with `StaffViews` has its parameters set so that it performs its operation immediately (as soon as a tool is dragged over a view which accepts its action), rather than the normal behavior of providing immediate semantic feedback and performing the action when the tool is dropped on the view.

Thus, when a `ScoreEventView` whose action is `scoreeventview:` is dragged over a `StaffView`, the `StaffView` immediately gets sent the message `scoreeventview:`, with the tool (*i.e.* the `ScoreEventView`) as a parameter. The first step is to `erase:` the model of the `ScoreEventView` from the score, if possible. The `StaffView` then sends its model's `Score` the `timeOf:` message, with parameter the x coordinate of the `StaffEventView` being dragged. The time returned is made the time of the `ScoreEventView`'s model, which is then added to the score. When a subsequent drag event of the `ScoreEventView` results in the `scoreeventview:` message to be sent to the `StaffView`, the process is repeated again. Thus as the user drags around the `ScoreEventView`, the score is continuously updated, and the effect of the drag immediately reflected on the display.

Though they have different actions, `AccView`, `TieView`, `DurModView`, and `BarLineView` tools operate similarly to the other `ScoreEventViews`. Rather than explain their functionality in the non-gesture-based interface, the next section discusses the semantics of the gestural interface to GSCORE.

GSCORE's gesture semantics

The gesture semantics rely heavily on the palette interface described above. When the palettes are first created, every view placed in the palette is named and made accessible via the "Attributes" button in the gesture semantics window (see Sections 7.7.2 and 7.7.3). It is then a simple matter in the gesture semantics to simulate dragging a copy of the view onto the staff (see Section 7.7.1). For example, consider the semantics of the 8u gesture, which creates an eighth note with an up stem:

```
recog = [[[noteview8up viewcopy] at:<startLoc>]
             reRaise:<currentEvent>];
```

The name `noteview8up` refers to the view of the eighth note with the up stem placed in the palette during program initialization. That view is copied (which results in the model being copied as well), moved to the starting location of the gesture (another "Attribute"), and the `currentEvent` (another "Attribute") is reraised using this view as the tool and its location as the event location. This simulates the actions of the `DragHandler`, and since `startLoc` is guaranteed to be over the staff (otherwise these semantics would never have been executed) the effect is to place an eighth note into the score. Similar semantics (the only difference is the view being copied) are used for all other note gestures, as well as all rest gestures and clef gestures.

The semantics of the bar gesture is similar to that of the note gestures, the difference being that a mouse tool is used rather than a virtual (view) tool.

```
recog = [handler mousetool:
              [barlineEvent_MouseTool
                     reRaise:<currentEvent>
                     at:<startLoc>]];
```

The timesig gesture for creating time signatures is more interesting. After it is recognized, *x* and *y* of the mouse control the numerator and the denominator of the time signature, respectively:

```
recog = [Seq :sx = <currentX>
             :sy = <currentY>
             :[[[timesigview4_4 viewcopy] at:<startLoc>]
                     reRaise:<currentEvent>]]
manip = [[recog model]
             timesig:[[[<currentX> Minus:sx]
                          DividedBy:10] Clip :1 :100]
                   :[[[<currentY> Minus:sy]
                        DividedBy:10] Clip :1 :100]]
```

Note that the `recog` expression is similar to the others; a view from the palette is copied, moved to the staff, and used as a tool in the reraising of an event. The `manip` expression, in contrast, does not operate on the level of simulated drags. Instead, it accesses the model of the newly created `TimeSigView` directly, sending it the `timesig::` message which sets its numerator and denominator. The division by 10 means that the mouse has to move 10 pixels in order to change one unit. The `Clip::` message ensures the result will be between 1 and 100, inclusive. For musical purposes, it is probably better to only use powers of two for the denominator, but unfortunately no `toThe:` message has been implemented in `TypeInt` (though it would be simple to do).

The key signature gesture (**key**) works similarly, except that only the *y* coordinate of the mouse is used (to control the number of accidentals in the key signature):

```
recog = [Seq :sy = <currentY>
             :[[[keysigview1sharps viewcopy]
                   at:<startLoc>]
                     reRaise:<currentEvent>]]
manip = [[recog model]
             keysig:[[[sy Minus:<currentY>]
                         DividedBy:10] Clip:[0 Minus:6] :6]]
```

A positive value for key signature indicates the number of sharps, a negative one the (negation of the) number of flats. The awkward `[0 Minus:6]` is used because the author failed to allow the creation of negative numbers with the "new int" button.

The above gestures are recognized when made on the staff. The **delete** and **move** gestures are only recognized when they begin on `ScoreEventViews`. The semantics of the **delete** gesture are:

```
recog = [Seq :[handler mousetool:delete_MouseTool]
             :[view delete]];
```

This changes the cursor, and deletes the view that the gesture began on. The latter effect could also have been achieved using `reRaise:`, but the above code is simpler.

The **move** gesture simply restores the normal cursor and reraises it at the starting location of the gesture, relying on the fact that in the non-gesture-based interface, score events may be dragged with the mouse:

```
recog = [[handler mousetool:normal_MouseTool]
            reRaise:startEvent];
```

In addition to the gestures that apply to any `ScoreEventView`, `NoteView` recognizes a few of its own. The three gestures for adding accidentals to notes (**sharp**, **flat**, and **natural**) access the `Note` object directly. For example, the semantics of the **sharp** gesture are:

```
recog = [[view model] acc:SHARP];
```

The **beam** gesture changes the cursor to the beam cursor and simulates clicking the beam cursor on the `NoteView` at the initial point:

```
recog = [[handler mousetool:beamtool_MouseTool]
           reRaise:startEvent];
```

The tie gestures (**uptie** and **downtie**) could have been implemented similarly. Instead, a variation of the above semantics causes the mouse cursor to revert to the normal cursor when the mouse button is released after the gesture is over:

```
recog = [Seq :[handler mousetool:tieUpEvent_MouseTool]
             :[tieUpEvent_MouseTool reRaise:startEvent]]];

manip = :[tieUpEvent_MouseTool reRaise:currentEvent]]];

done = [Seq :[tieUpEvent_MouseTool reRaise:currentEvent]
            :[handler mousetool:normal_MouseTool]];
```

The dot gesture accesses the `Note`'s raw duration, multiplies it by $\frac{3}{2}$ and changes the duration to the result. The note will add the appropriate dot in the score when it receives its new duration

```
recog = [Seq :m = [view model]
             :[m dur:[[[m rawdur] Times:3] DividedBy:2]]];
manip = recog;
```

The `manip = recog` statement itself does nothing of itself, but by virtue of it being non-nil, the gesture handler does not relinquish control until the mouse button is released. Without this statement, the mouse cursor tool (whatever it happens to be) would operate on any view it was dragged across after the dot gesture was recognized.

8.3 MDP

MDP is gesture-based drawing program that takes multi-finger Sensor Frame gestures as input. Though primarily a demonstration of multi-path gesture recognition, MDP also shows how gestures can be incorporated cheaply and quickly into a non-object-oriented system. This is in contrast to GRANDMA, which, whatever its merits, requires a great deal of mechanism (an object=oriented user interface toolkit with appropriate hooks) before gestures can be incorporated.

The user interface to MDP is similar to that of GDP. The user makes gestures, which results in various geometric objects being created and manipulated. The main differences are due to the different input devices. In addition to classifying multiple finger gestures, MDP uses multiple fingers in the manipulation phase. This allows, for example, a graphic object to be rotated, translated, and scaled simultaneously.

Figure 8.8 shows an example MDP session. Note that how, once a gesture has been recognized, additional fingers may be brought in and out of the picture to manipulate various parameters. Multiple finger tracking imbues the two-phase interaction with even more power than the single-path two-phase interaction.

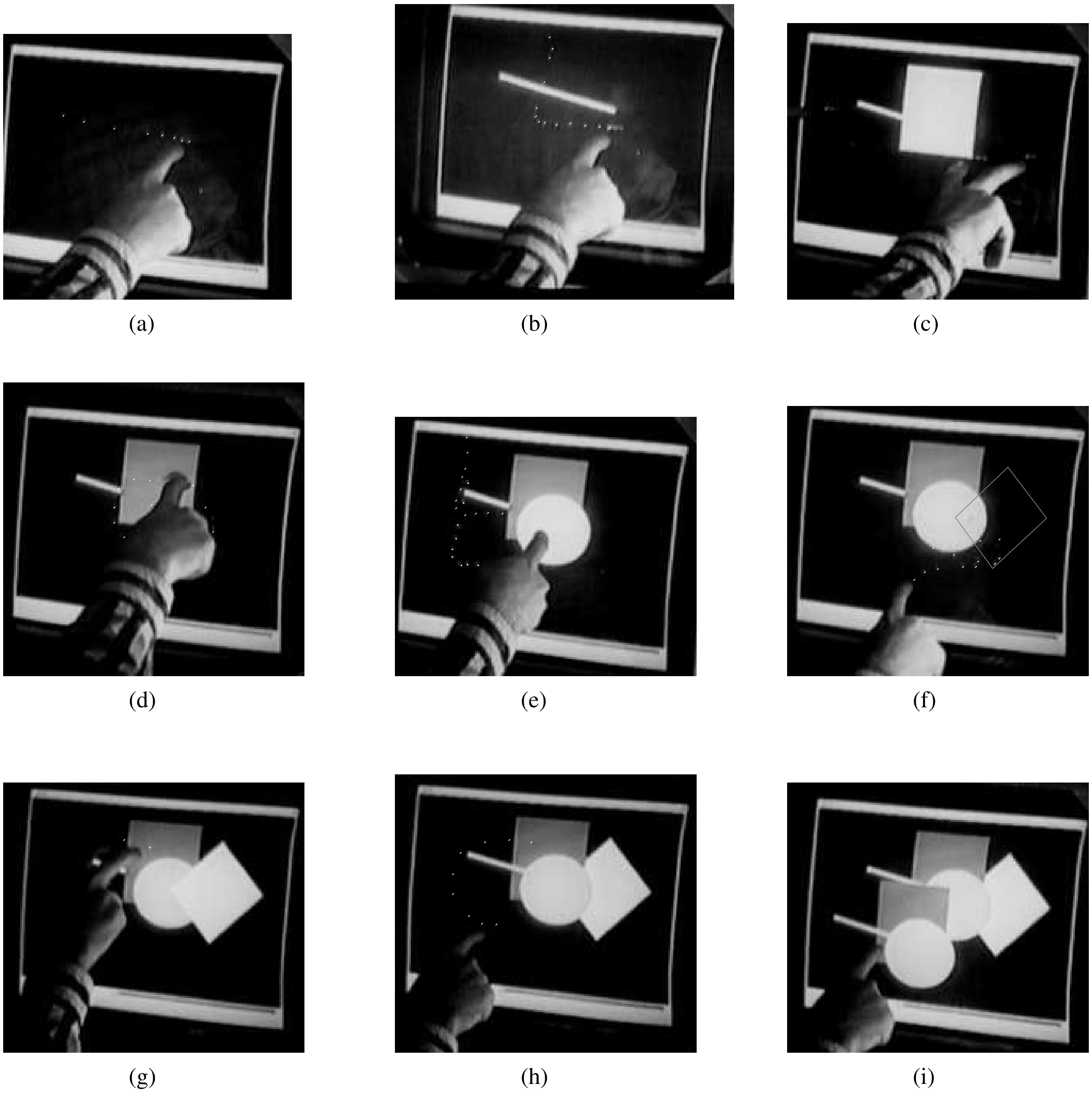
8.3.1 Internals

Figure 8.9 shows the internal architecture of MDP. The lines indicate the main data flow paths through the various modules.

Like the gesture-based systems built using GRANDMA, when MDP is first started, a set of gesture training examples is read from a file. These are used to train the multi-path classifier as described in Chapter 5. MDP itself provides no facility for creating or modifying the training examples. Instead, a separate program is used for this purpose.

The Sensor Frame is not integrated with the window manager on the IRIS, making the handling of its input more difficult than the handling of mouse input. In particular, coordinates returned by the Sensor Frame are absolute screen coordinates in an arbitrary scale, while the window manager generally expects window-relative coordinates to be used. Fortunately, the IRIS windowing system supports general coordinate transformations on a per-window basis, which MDP uses as follows.

When started, MDP creates a window on the screen, and reads an *alignment file* to determine the coordinate transformation for mapping window coordinates to screen coordinates that makes the

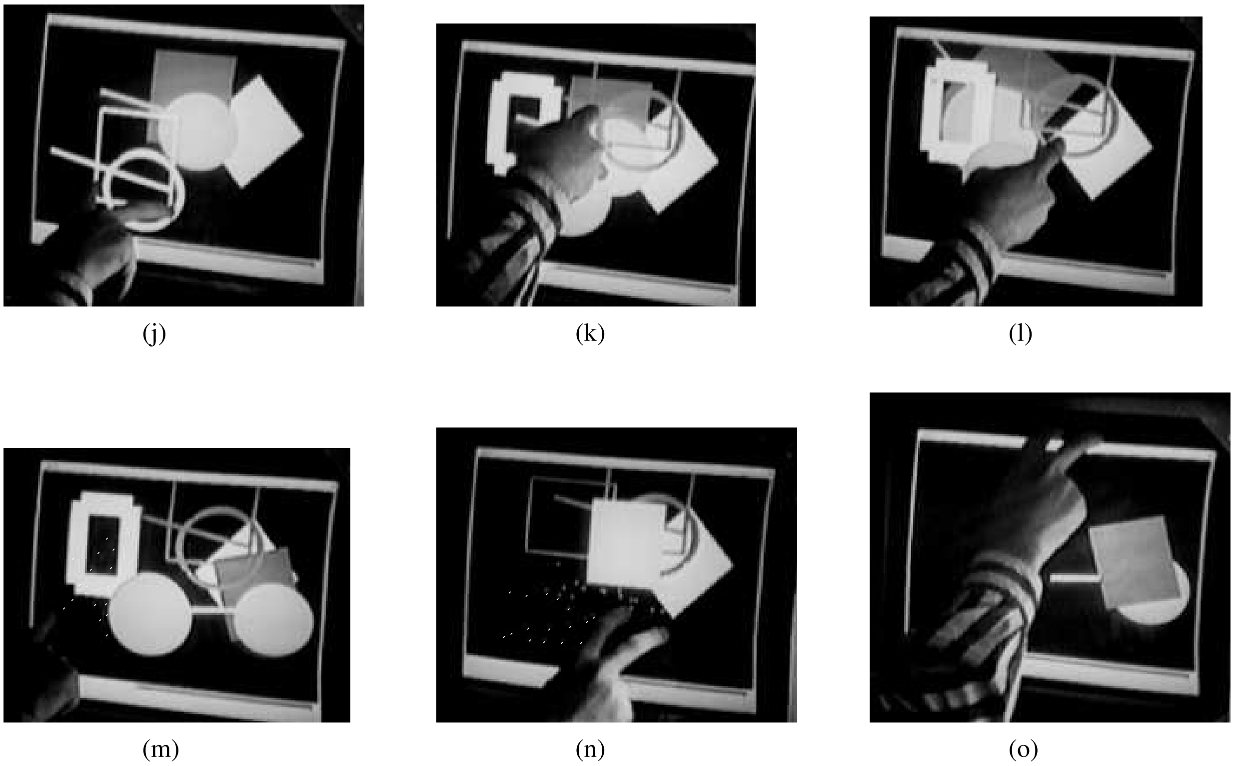

(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figure 8.8: An example MDP session

This figure consists of snapshots of a video of an MDP session. Some panels have been retouched to make the inking more apparent. Panel (a) shows the single finger line *gesture, which is essentially the same as GDP's* line *gesture. As in GDP, the start of the gesture gives one endpoint of the line, while the other endpoint is dragged by the gesturing finger after the gesture is recognized. Additional fingers may be used to control the line's color and thickness. Panel (b) shows the created line, and the* rectangle *gesture, again the same as GDP's. After the gesture is recognized, additional fingers may be brought into the sensing plane to control the rectangle's color, thickness, and filled property, as shown in panel (c). Panel (d) shows the* circle *gesture, which works analogously. Panel (e) shows the two finger* parallelogram *gesture. After the gesture is recognized, the two gesturing fingers control two corners of the parallegram. An additional finger*


(j) (k) (l)

(m) (n) (o)

Fig 8.8 (continued)

in the sensing plane will then control a third corner, allowing an arbitrary parallelogram to be entered. Panel (f) shows the edit color *gesture being made at the newly created parallelogram. After this gesture is recognized, the parallelogram's color and filled property may be dynamically manipulated. Panel (g) shows the three finger* pack *(group) gesture. During the* pack *interaction, all object touched by any of the fingers are grouped into a single set. Here, the line, rectangle, and circle are grouped together to make a cart. Panel (h) shows the* copy *gesture. After the gesture is recognized, the object indicated by the first point of the gesture (in this case, the cart) is dragged by the gesturing finger, as shown in panel (i). Additional fingers allow the color, edge thicknesses, and filled property of the copy to be manipulated, as shown in panel (j).* Circle *and* rectangle *gestures (both not shown) were then used to create some additional shapes. Panel (k) shows the two finger* rotate *gesture. After it is recognized, each of the two fingers become attached to their respective points where they first touched the designated object. By moving the fingers apart or together, rotating the hand, and moving the hand, the object may be simultaneously scaled, rotated, and translated as shown in panel (l). (The fingers are not touching the object due to the delay in getting the input data and refreshing the screen.) Panel (m) shows the* delete *gesture being used to delete a rectangle. Not shown are more deletion and creation gestures, leaving the drawing in the state shown in panel (n). Panel (n) showns the three finger* undo *gesture. Upon recognition, the most recent creation or deletion is undone. Moving the fingers up causes more and more operations to be undone, while moving the fingers down allows undone operations to be redone, interactively. Panel (o) shows a state during the interaction where many operations have been undone. In this implementation, creations and deletions are undoable, but position changes are not. This explains why, in panel (o), only the cart items remain (undo back to panel (e)), but those items are in the position they assumed in panel (m).*

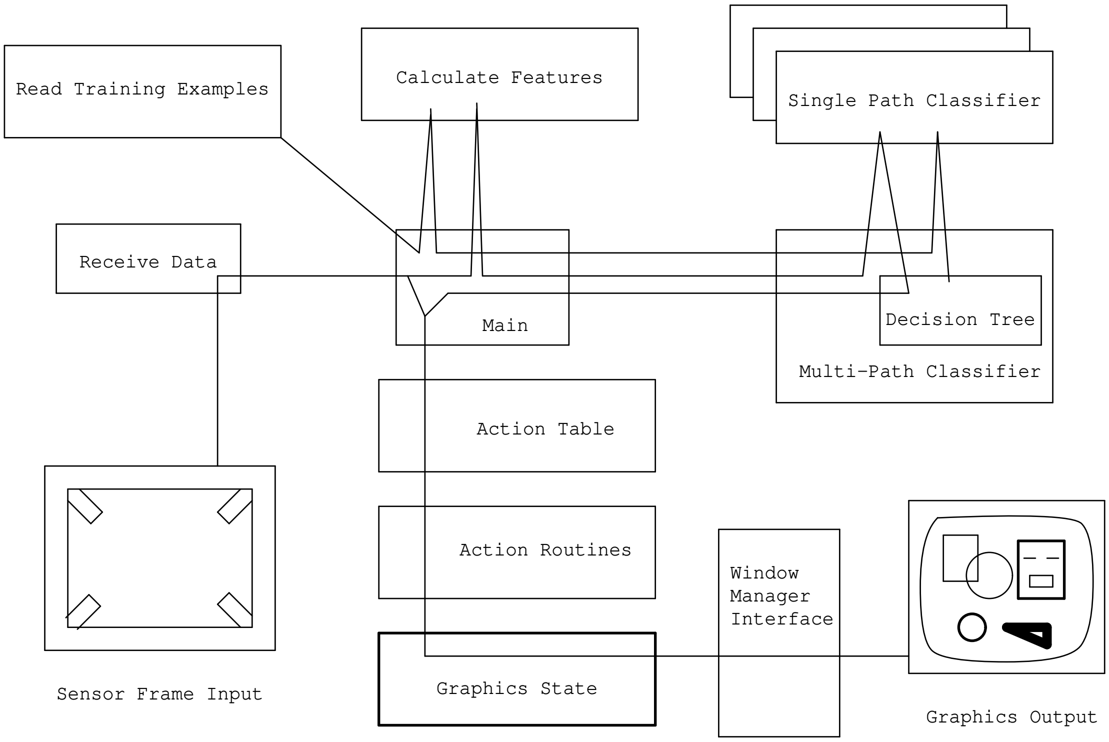


Figure 8.9: MDP internal structure

window coordinate system identical with the Sensor Frame coordinate system. If the given window size and position has not been seen before (as indicated by the alignment file) the user is forced to go through an alignment dialogue before proceeding (this also occurs when the window occurs moved or resized). Two dots are displayed, one in each corner of the window, and the user is asked to touch each dot. The data read are used to make window coordinates exactly match Sensor Frame coordinates. The transformation for window coordinates to screen coordinates is done by the IRIS software, and does not have to be considered by the rest of the program. The parameters are saved in the alignment file to avoid having to repeat the procedure each time MDP is started.[2]

Once initialized, the MDP begins to read data from the Sensor Frame. The current Sensor Frame software works by polling, and typically returns data at the rate of approximately 30 snapshots per second. The "Receive Data" module performs the path tracking (see section 5.1) and returns snapshot records consisting of the current time, number of fingers seen by the frame, and tuples (x, y, i) for each finger, (x, y) being the finger's location in the frame. The i is the path identifier, as determined by the path tracker. The intent is that a given value of i represents the same finger in successive snapshots.

Normally, MDP is in its WAIT state, where the polling indicates that there are no fingers in the plane of the frame. Once one or more fingers enter the field of view of the frame, the COLLECT state is entered. Each successive snapshot is passed to the "calculate features" module, which performs the incremental feature calculation. The COLLECT state ends when the user removes all fingers from the frame viewfield or stops moving for 150 milliseconds. (The timeout interval is settable by the user, but 150 milliseconds has been found to work well.) Unlike a mouse user, it is difficult for Sensor Frame users to hold their fingers perfectly still, so a threshold is used to decide when the user has not moved. In other words, the threshold determines the amount of movement allowable between successive snapshots that is to count as "not moving." This is done by comparing the threshold to the error metric calculated during the path tracking (sum of squared distances between corresponding points in successive snapshots).

Once the gesture has been collected, its feature vectors are passed to the multi-path classifier, which returns the gesture's class. Then the recognition action associated with the class is looked up in the action table and executed. As long as at least one finger remains in the field of view, the manipulation action of the class is executed.

Many of GRANDMA's ideas for specifying gesture semantics are used in MDP. Although MDP does not have a full-blown interpreter, there is a table specifying the recognition action and manipulation action for each class. While it would be possible for the tables to be constructed at runtime, currently the table is compiled into MDP. Each row in the entry for a class consists of a finger specification, the name of a C function to call to execute the row, and a constant argument to pass to the function. The finger specification determines which finger coordinates to pass as additional arguments to the function.

Consider the table entries for the MDP line gesture, similar to the GDP line gesture:

```
ACTION(_LINErecog)
    { ALWAYS,        BltnCreate,      (int)Line, },
```

---

[2] Moving or resizing the window often requires the alignment procedure to be repeated, a problem that would of course have to be fixed in a production version of the program.

```
    { START(0),      BltnSetPoint,    0, },
END_ACTION

ACTION(_LINEmanip)
    { CURRENT(0),    BltnSetPoint,    1, },
    { CURRENT(1),    BltnThickness,   0, },
    { CURRENT(2),    BltnColorFill,   0, },
END_ACTION
```

When a line gesture is recognized, the `_LINErecog` action is executed. Its first line results in the call `BltnCreate(Line)` being executed. The `ALWAYS` means that this row is not associated with any particular finger, thus no finger coordinates are passed to `BltnCreate`. The next line results in `BltnSetPoint(0,` x_{0s}, y_{0s}`)` being called, where (x_{0s}, y_{0s}) is the initial point of the first finger (finger 0) in the gesture.

For each snapshot after the line gesture has been recognized, the `_LINEmanip` action is executed. The first line causes `BltnSetPoint(1,` x_{0c}, y_{0c}`)` to be called, where (x_{0c}, y_{0c}) is the current location of the first finger (finger 0). The next line causes `BltnThickness(0,` x_{1c}, y_{1c}`)` to be called, (x_{1c}, y_{1c}) being the current location of the second finger. Similarly, the third line causes `BltnColorFill(0,` x_{2c}, y_{2c}`)` to be called.

If any of the fingers named in a line of the action are not actually in the field of view of the frame, that line is ignored. For example, the line gesture in MDP, as in GDP, is a single straight stroke. Immediately after recognition there will only be one finger seen by the frame, namely finger zero, so the lines beginning `CURRENT(1)` and `CURRENT(2)` will not be executed. If a second finger is now inserted into the viewfield, both the `CURRENT(0)` and `CURRENT(1)` lines will be executed every snapshot. If the initial finger is now removed, the `CURRENT(0)` line will no longer be executed, until another finger is placed in the viewfield.

The assignment of finger numbers is done as follows: when the gesture is first recognized, each finger is assigned its index in the path sorting (see Section 5.2). During the manipulation phase, when a finger is removed, its number is *freed*, but the numbers of the remaining fingers stay the same. When a finger enters, it is assigned the smallest free number.

The semantic routines (*e.g.* `BltnColorFill`) communicate with each other (and successive calls to themselves) via shared variables. All these functions are defined in a single file with the shared variables declared at the top. When there are no fingers in the viewfield, the call `BltnReset()` is made; its function is to initialize the shared variables. In MDP, all shared variables are initialized by `BltnReset();` from this it follows that the interface is *modeless*. Another system might have some state retained across calls to `BltnReset();` for example, the current selection might be maintained this way.

The `Bltn...` functions manipulate the drawing elements through a package of routines. The actual implementation of those routines is similar to the implementation of the GDP objects. Rather than go into detail, the underlying routines are summarized. MDP declares the following types:

```
typedef enum { Nothing, Line, Rect,
               Circle, SetOfObjects } Type;
```

```
typedef struct { /* ... */ } *Element;
typedef struct { /* ... */ } *Trans;
```

Assume the following declarations for expositional purposes:

```
Element e;          /* a graphic object */
Type    type;       /* the type of a graphic object */
int     x, y;       /* coordinates */
int     p;          /* a point number: 0, 1, or 2, 3 */
int     thickness, color;
BOOL    b;
Trans   tr;         /* a transformation matrix */
```

The `Element` is a pointer to a structure representing an element of a drawing, which is either a `Line`, `Rect`, `Circle` or `SetOfObjects`. The `Element` structure includes an array of points for those element types which need them. A `Line` has two points (the endpoints), a `Rect` has three points (representing three corners, thus a `Rect` is actually a parallelogram), and a `Circle` has two points (the center and a point on the circle). A `SetOfObjects` contains a list of component elements which make up a single composite element.

`Element StNewObj(type)` adds a new element of the passed `type` to the drawing, and returns a handle. Initially, all the points in the element are marked *uninitialized*. Any element with uninitialized points will not be drawn, with the exception of `Rect` objects, which will be drawn parallel to the axes if point 1 is uninitialized.

`StUpdatePoint(e, p, x, y)` changes point `p` of element `e` to be `(x,y)`. Returns FALSE iff `e` has no point `p`.

`StGetPoint(e, p, &x, &y)` sets `x` and `y` to point `p` of element `e`. Returns FALSE iff `e` has no point `p` or point `p` is uninitialized.

`StDelete(e)` deletes object `e` from the drawing.

`StFill(e, b)` makes object `e` filled if `b` is `TRUE`, otherwise makes `e` unfilled. This only applies to circles and rectangles, which will be only have their borders drawn if unfilled, otherwise will be “colored in.”

`StThickness(e, t)` sets the thickness of `e`’s borders to `t`. Only applies to circles, rectangles, and lines.

`StColor(e, color)` changes the color of `e` to `color`, which is an index into a standard color map. If `e` is a set, all members of `e` are changed.

`StTransform(e, tr)` applies the transformation `tr` to `e`. In general, `tr` can cause translations, rotations, and scalings in any combination.

`void StMove(e, x, y)` is a special case of `StTransform` which translates `e` by the vector `(x,y)`.

`StCopyElement(e)` adds an identical copy of `e` to the drawing, which is also returned. If `e` is a set, its elements will be recursively copied.

`StPick(x, y)` returns the element in the drawing at point `(x,y)`, or `NULL` if there is no element there. The topmost element at `(x,y)` is returned, where elements created later are considered to be on top of elements created earlier. The thickness and "filled-ness" of an element are considered when determining if an element is at `(x,y)`.

`StHighlight(e, b)` turns on highlighting of `e` if `b` is `TRUE`, off otherwise. Highlighting is currently implemented by blinking the object.

`StUnHighlightAll()` turns off highlighting on all objects in the drawing.

`void StRedraw()` draws the entire picture on the display. Double buffering is used to ensure smooth changes.

`StCheckpoint()` saves the current state of the drawing, which can be later restored via `StUndoMore`.

`StUndoMore(b)` changes the drawing to its previously checkpointed state (if `b` is `TRUE`). Each successive call to `StUndoMore(TRUE)` returns to a previous state of the picture until the state of the picture when the program was started in reached. `StUndoMore(FALSE)` performs a `redo`, undoing the effect of the last `StUndoMore(TRUE)`. Successive calls to `StUndoMore(FALSE)` will eventually restore a drawing to its latest checkpointed state.

`Trans AllocTran()` allocates a transformation, which is initialized to the identity transformation.

`SegmentTran(tr, x0,y0, x1,y1, X0,Y0, X1,Y1)` sets `tr` to a transformation consisting of a rotation, followed by a scaling, followed by translation, the net effect of which would be to map a line segment with endpoints `(x0,y0)` and `(x1,y1)` to one with endpoints `(X0,Y0)` and `(X1,Y1)`. Other transformation creation functions exist, but this is the only one used directly by the gesture semantics.

`JotC(color, x, y, text)` draws the passed text string on the screen in the passed color, at the point `(x,y)`. The text will be erased at the next call to `StRedraw`.

8.3.2 MDP gestures and their semantics

Now that the basic primitives used by MDP have been described, the actual gestures used, and their effect and implementation are discussed. Figure 8.10 shows typical examples of the MDP gestures used. Each is described in turn.

Line The line gesture creates a line with one endpoint being the start of the gesture, the other tracking finger 0 after the gesture has been recognized. Finger 1 (which must be brought in after the gesture has been recognized) controls the thickness of the line as follows: the point

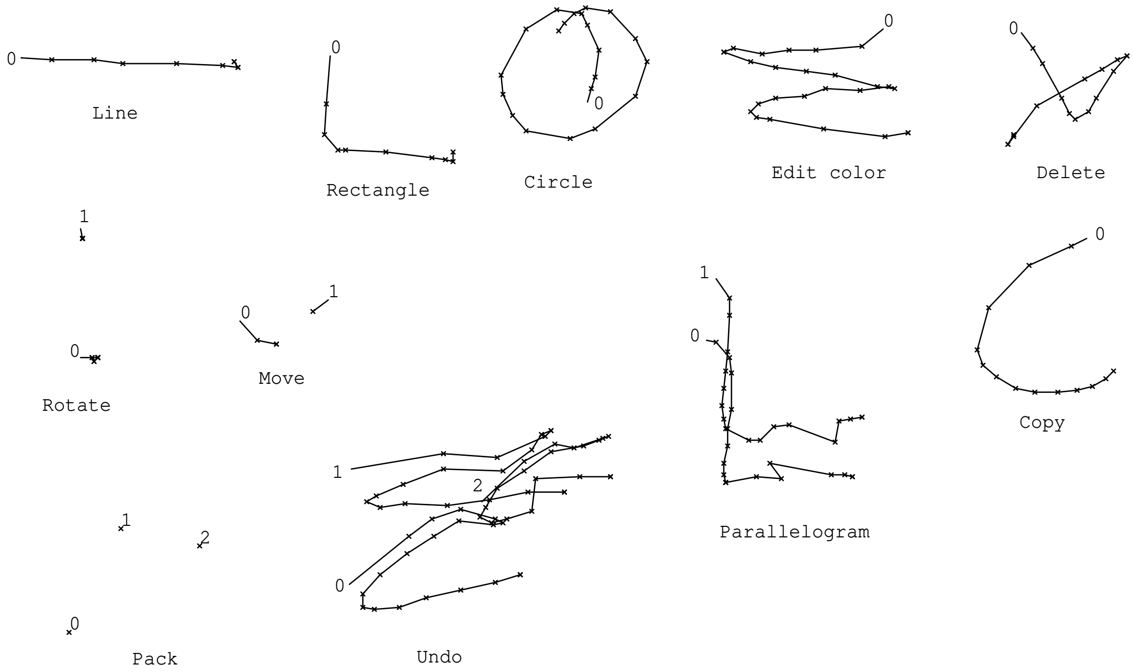


Figure 8.10: MDP gestures

where finger 1 first enters is displayed on the screen; the thickness of the line is proportional to the difference in *y* coordinate of finger 1's current point and initial point. Finger 2 controls the color of the line in a similar manner. (Here a color is represented simply by an index into a color map.)

The action table entry for line has already been listed in the previous section. The C routines called are listed here:

```
BltnCreate(arg) {
    E = StNewObj(arg);
    shouldCheckpoint = TRUE;
}
BltnSetPoint(arg, gx, gy) {
    if(E) StUpdatePoint(E, arg, gx, gy);
}
BltnThickness(arg, gx, gy) { int x, t;
    if(tx == -1) tx = gx, ty = gy;
    if(!E) return;
    x = arg==0 ? abs(tx-gx) : abs(ty-gy);
    t = Scale(x, 1, 2, 1, 100);
    StThickness(E, t);
    JotC(RED, tx, ty, arg==0 ? "TX%d" : "TY%d", t);
```

```
        JotC(RED, gx, gy+10, "t");
    }
    BltnColorFill(arg, gx, gy) { int color, fill;
        if(!E) return;
        if(cfx ==  −1) cfx = gx, cfy = gy;
        fill = Scale(cfx − gx, 1, 10, −1, 1);
        StFill(E, fill >= 0);
        color = Scale(cfy − gy, 1, 25, −15, 15);
        if(color < 0) color = −color;
        else if(color == 0) color = 1;
        StColor(E, color);
        JotC(GREEN, cfx, cfy, "CF%d/%d", color, fill);
        JotC(GREEN, gx, gy+10, "cf");
    }
    Scale(i, num, den, low, high) {
        int j = i ∗ num;
        int k = j >= 0 ?  j/den :  −((−j)/den);
        return k < low ?  low :  k > high ?  high :  k;
    }
```

The `BltnReset()` function sets `E` to `NULL`, and sets `tx`, `ty`, `cfx`, and `cfy` all to `-1`. `BltnReset()` calls `StCheckpoint()` if `shouldCheckpoint` is `TRUE` and then sets `shouldCheckpoint` to `FALSE`.

The functions `BltnThickness` and `BltnColorFill` provide feedback to the user by jotting some text (“TX” and “CF”, respectively) that indicates the location that the finger first entered the viewfield. Lower case text (“t” and “cf”) is drawn at the appropriate fingers, indicating to the user which finger is controlling which parameter.

Rectangle The rectangle gesture works similarly to the line gesture. After the gesture is recognized, a rectangle is created, one corner at the starting point of the gesture, the opposite corner tracking finger 0. Fingers 1 and 2 control the thickness and color as with the line gesture. Finger 2 also controls whether or not the rectangle is filled; if it is to the left of where it initially entered, the rectangle is filled, otherwise not.

```
    ACTION(_RECTrecog)
        { ALWAYS,        BltnCreate,    (int)Rect, },
        { START(0),      BltnSetPoint, 0, },
    END_ACTION

    ACTION(_RECTmanip)
        { CURRENT(0), BltnSetPoint,  2, },
        { CURRENT(1), BltnThickness, 0, },
        { CURRENT(2), BltnColorFill, 0, },
    END_ACTION
```

Circle The circle gesture causes a circle to be created, the starting point of the gesture being the center, and a point on the circle controlled by finger 0. Fingers 1 and 2 operate as they do in the rectangle gesture. Its semantics of the circle gesture are almost identical that of the line gesture, and are thus not shown here.

Edit color This gesture lets the user edit the color and "filled-ness" of an existing object. Beginning the gesture on an object edits that object. Otherwise, the user moves finger 0 until he touches an object to edit. Once selected, finger 0 determines the color and fill properties of the object as finger 2 did in the previous gestures.

```
ACTION(_COLORrecog)
    { START(0),    BltnPick,    0, },
END_ACTION

ACTION(_COLORmanip)
    { CURRENT(0),    BltnPickIfNull,    0, },
    { CURRENT(0),    BltnColorFill,     0, },
END_ACTION

BltnPick(arg, gx, gy) {
    E = StPick(gx, gy);
    if(E) px = gx, py = gy;
}
BltnPickIfNull(arg, gx, gy) {
    if(!E) BltnPick(arg, gx, gy);
}
```

Copy The copy gesture picks an element to be copied in the same manner as the edit-color gesture above. Once copied, finger 0 drags the new copy around, while finger 1 can be used to adjust the color and thickness of the copy.

```
ACTION(_COPYrecog)
    { START(0),        BltnPick,        0, },
END_ACTION

ACTION(_COPYmanip)
    { CURRENT(0),      BltnPickIfNull,    0, },
    { CURRENT(0),      BltnCopy,          0, },
    { CURRENT(0),      BltnMove,          0, },
    { CURRENT(1),      BltnColorFill,     0, },
END_ACTION
```

In the interest of brevity the C routines will no longer be listed, since they are very similar to those already seen.

Move Move is a two-finger "pinching" gesture. An object is picked as in the previous gestures, and then tracks finger 0.

```
ACTION(_MOVErecog)
    { START(0),        BltnPick,         0, },
END_ACTION

ACTION(_MOVEmanip)
    { CURRENT(0),      BltnPickIfNull,     0, },
    { CURRENT(0),      BltnMove,           0, },
END_ACTION
```

Delete The delete gesture picks an object just like the previous gestures, and then deletes it.

```
ACTION(_DELETErecog)
    { START(0),        BltnPick,         0, },
END_ACTION

ACTION(_DELETEmanip)
    { CURRENT(0),      BltnPickIfNull,   0, },
    { CURRENT(0),      BltnDelete,       0, },
END_ACTION
```

Parallelogram The parallelogram gesture is a two-finger gesture. One corner of the parallelogram is determined by the initial location of fingers 0; an adjacent corner tracks finger 0, and the opposite corner tracks finger 1. Adding a third finger (finger 2) moves the initial point of the parallelogram.

```
ACTION(_PARArecog)
    { ALWAYS,        BltnCreate,    (int)Rect, },
    { START(0),      BltnSetPoint,   0, },
END_ACTION

ACTION(_PARAmanip)
    { CURRENT(0),        BltnSetPoint,   1, },
    { CURRENT(1),        BltnSetPoint,   2, },
    { CURRENT(2),        BltnSetPoint,   0, },
END_ACTION
```

Rotate Rotate is a two-finger gesture. An object is picked with either finger. At the time of the pick, each finger becomes attached to a point on the picked object. Each finger then drags its respective point; the object can thus be rotated by rotating the fingers, scaled by moving the fingers apart or together, or translated by moving the fingers in parallel.

```
ACTION(_ROTATErecog)
    { START(0),    BltnPick,        0, },
    { START(1),    BltnPickIfNull,  0, },
END_ACTION
```

```
ACTION(_ROTATEmanip)
    { CURRENT(0),  BltnPickIfNull,  0, },
    { CURRENT(1),  BltnPickIfNull,  0, },
    { CURRENT(0),  BltnRotate,      0, },
    { CURRENT(1),  BltnRotate,      1, },
END_ACTION
```

Pack The pack gesture is a three-finger gesture. Any objects touched by the any of the fingers are added to a newly created `SetOfObjects`.

```
ACTION(_PACKrecog)
END_ACTION

ACTION(_PACKmanip)
    { CURRENT(0),        BltnPick,        0, },
    { ALWAYS,        BltnAddToSet,        0, },
    { CURRENT(1),        BltnPick,        0, },
    { ALWAYS,        BltnAddToSet,        0, },
    { CURRENT(2),        BltnPick,        0, },
    { ALWAYS,        BltnAddToSet,        0, },
END_ACTION
```

Undo The undo gesture is also a three-finger gesture, basically a “Z” made with three fingers moving in parallel. After it is recognized, moving finger 0 up causes more and more of the edits to be undone, and moving finger 0 down causes those edits to be redone.

```
ACTION(_UNDOrecog)
    { CURRENT(0),        BltnUndo,        0, },
END_ACTION

ACTION(_UNDOmanip)
    { CURRENT(0),        BltnUndo,        0, },
END_ACTION
```

8.3.3 Discussion

MDP is the only system known to the author which uses non-DataGlove multiple finger gestures. Thus, a brief discussion of the gestures themselves is warranted.

MDP’s single finger gestures are taken directly from GDP. After recognition, additional fingers may be brought into the sensing plane to control additional parameters. Wherever an additional finger is first brought into the sensing plane becomes the position that gives the current value of the parameter which that finger controls; the position of the finger relative to this initial position determines the new value of the parameter. This relative control was felt by the author to be less awkward than other possible schemes, though this of course needs to be studied more thoroughly.

The multiple finger gestures are designed to be intuitive. The parallelogram gesture is, for example, two fingers making the rectangle gesture in parallel. The move gesture is meant to be a pinch, whereby the object touched is grabbed and then dragged around. The two finger rotate gesture allows two distinct points on an object to be selected carefully. During the manipulation phase, each of these points tracks a finger, allowing for very intuitive translation, rotation, and scaling of the object. The three finger undo gesture is intended to simulate the use of an eraser on a blackboard.

The Sensor Frame is not a perfect device for gestural input. One problem with the Sensor Frame is that the sensing plane is slightly above the surface of the screen. It is difficult to precisely pull a finger out without changing its position. This often results in parameters that were carefully adjusted during the manipulation phase of the interaction being changed accidentally as the interaction ends. This problem happens more often in multiple finger gestures, where, due to problems with the Sensor Frame, removing one finger may change the reported position of other fingers even though those fingers have not moved. Also, it is more difficult to pull out one finger carefully when other fingers must be kept still in the sensing plane. Finally, it does not take very long for a gesturer's arm to get tired when using a Sensor Frame attached to a vertically mounted display.

In MDP, the two-phase interaction technique is applied in the context of multiple fingers. As each finger's position represents two degrees of freedom, multi-path interactions allow many more parameters to be manipulated than do single-path interactions. Also, since people are used to gesturing with more than one finger, multiple fingers potentially allows for more natural gestures. Even though sometimes only one or two fingers are used to enter the recognized part of the gesture, additional fingers can then be utilized in the manipulation phase. The result is a new interaction technique that needs to be studied further.

8.4 Conclusion

This chapter described the major applications which were built to demonstrate the ideas of this thesis. Two, GDP and GSCORE, were built on top of GRANDMA, and show how single-path gestures may be integrated into MVC-based applications. The third, MDP, demonstrates the use of multi-path gestures, and shows how gestures may be integrated in a quick and dirty fashion in a non-objected-oriented context.

Chapter 9

Evaluation

The previous chapters report on some algorithms and systems used in the construction of gesture-based applications. This chapter attempts to evaluate how well those algorithms and systems work. When possible, quantitative evaluations are made. When not, subjective or anecdotal evidence is presented.

9.1 Basic single-path recognition

Chapter 3 presents an algorithm for classifying single-path gestures. In this section the performance of the algorithm is measured in a variety of ways. First, the recognition rate of the classifier is measured, as a function of the number of classes and the number of training examples. By examining the gestures that were misclassified, various sources of errors are uncovered. Next, the effect of the rejection parameters on classifier performance is studied. Then, the classifier is tested on a number of different gesture sets. Finally, tests are made to determine how well a classifier trained by one person recognizes the gestures of another.

9.1.1 Recognition Rate

The *recognition rate* of a classifier is the fraction of example inputs that it correctly classifies. In this section, the recognition rates of a number of classifiers trained using the algorithm of Chapter 3 are measured. The gesture classes used are drawn from those used in GSCORE (Section 8.2). There are two reasons for testing on this set of gestures rather than others discussed in this dissertation. First, it consists of a fairly large set of gestures (30) used in a real application. Second, the GSCORE set was not used in the development or the debugging of the classification software, and so is unbiased in this respect.

GRANDMA provides a facility through which the examples used to train a classifier are classified by the classifier. While running the training examples through the classifier is useful for discovering ambiguous gestures and determining approximately how well the classifier can be expected to perform, it is not a good way to measure recognition rates. Any trainable classifier will be biased toward recognizing its training examples correctly. Thus in all the tests described below, one set of

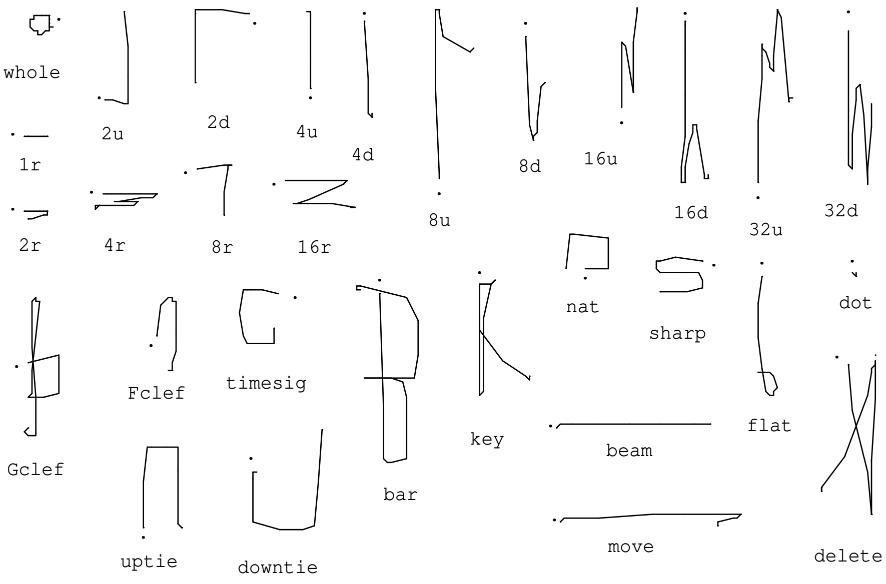


Figure 9.1: GSCORE gesture classes used for evaluation

example gestures is used to train the classifier, while another, entirely distinct, set of examples is used to evaluate its performance.

Figure 9.1 shows examples of the gesture classes used in the first test. All were entered by the author, using the mouse and computer system described in Chapter 3. First, 100 examples of each class were entered; these formed the *training set*. Then, the author entered 100 more examples of each class; these formed the *testing set*. For both sets, no special attempt to was made to gesture carefully, and obviously poor examples were not eliminated.

There was no classification of the test examples as they were entered; in other words, no feedback was provided as to the correctness of each example immediately after it was entered. Given such feedback, a user would tend to adapt to the system and improve the recognition of future input. The test was designed to eliminate the effect of this adaptation on the recognition rate.

The performance of the statistical gesture recognizer depends on a number of factors. Chief among these are the number of classes to be discriminated between, and the number of training examples per class. The effect of the number of classes is studied by building recognizers that use only a subset of classes. In the experiment, a class size of C refers to a classifier that attempts to discriminate between the first C classes in figure 9.1. Similarly, the effect of the training set size is studied by varying E, the number of examples per class. A given value of E means the classifier was trained on examples 1 through E of the training data for each of C classes.

Figure 9.2 plots the recognition rate against the number of classes C for various training set sizes E. Each point is the result of classifying 100 examples of each of the first C classes in the testing set. The number of correct classifications is divided by the total number of classifications attempted ($100C$) to give the recognition rate. (Rejection has been turned off for this experiment.) Figure 9.3 shows the results of the same experiment plotted as recognition rate versus E for various values of

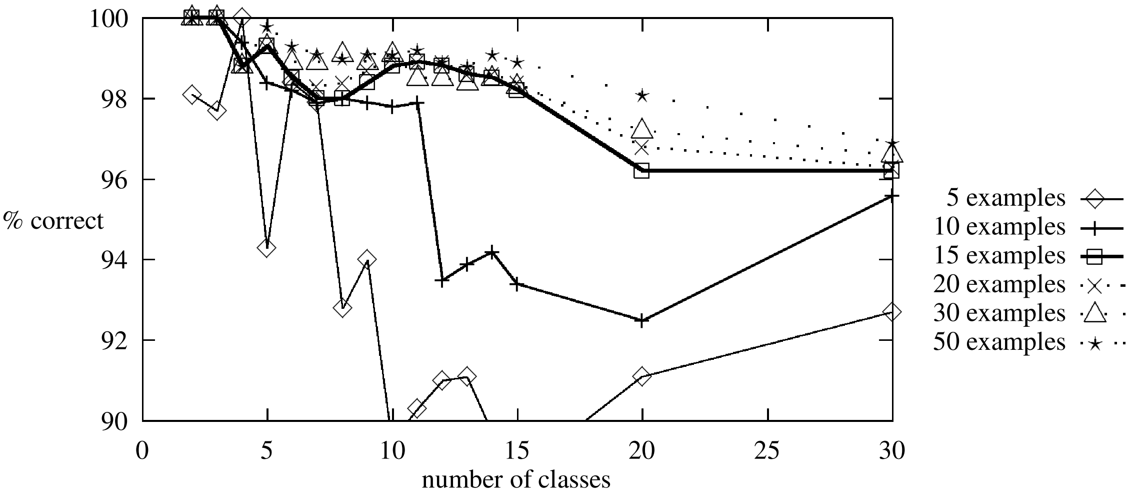


Figure 9.2: Recognition rate vs. number of classes

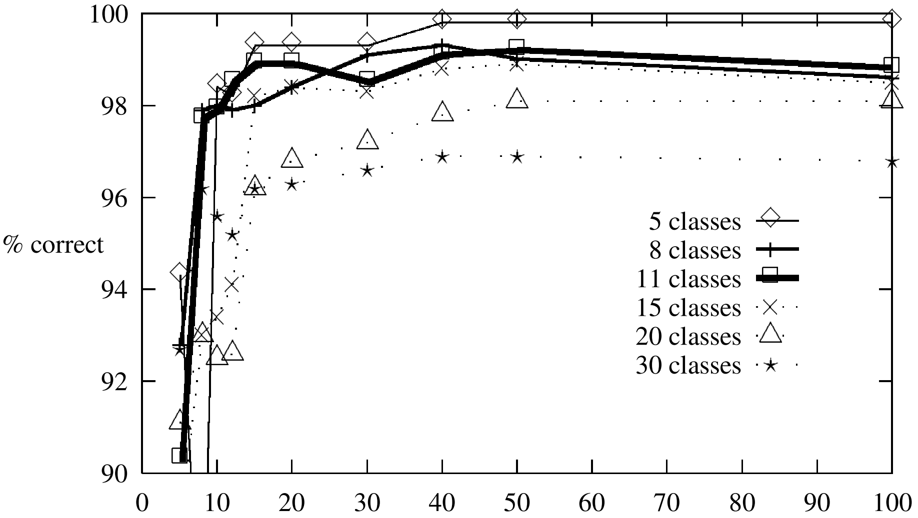


Figure 9.3: Recognition rate vs. training set size

C.

In general, the data are not too surprising. As expected, recognition rate increases as the training set size increases, and decreases as the number of classes increases. For $C = 30$ classes, and $E = 40$ examples per class, the recognition rate is 96.9%. For $C = 30$ and $E = 10$ the rate is 95.6%. $C = 10$ and $E = 40$ gives a rate of 99.3%, while for $C = 10$ and $E = 10$ the rate is 97.8%.

Of practical significance for GRANDMA users is the number of training examples needed to give good results. Using $E = 15$ examples per class gives good results, even for a large number of classes. Recognition rate can be marginally improved by using $E = 40$ examples per class, above which no significant improvement occurs. $E = 10$ results in poor performance for more than $C = 10$ classes. It is comforting to know that GRANDMA, a system designed to allow experimentation with gesture-based interfaces, performs well given only 15 examples per class. This is in marked contrast to many trainable classifiers, which often require hundreds or thousands of examples per class, precluding their use for casual experimentation [125, 47].

Analysis of errors

It is enlightening to examine the test examples that were misclassified in the above experiments. Figure 9.4 shows examples of all the kinds of misclassifications by the $C = 30$, $E = 40$ classifier. Not every misclassification is shown in the figure, but there is a representative of every A classified as B, for all $A \neq B$. The label "A as B (x n)" indicates that the example was labeled as class A in the test set, but classified as B by the classifier. The n indicates the number of times an A was classified as a B, when it is more than once.

The following types of errors can be observed in the figure. Many of the misclassifications are the result of a combination of two of the types.

Poorly drawn gestures. Some of the mistakes are simply the result of bad drawing on the part of the user. This may be due to carelessness, or to the awkwardness of using a mouse to draw. Examples include "8u as uptie," "2r as sharp," "8r as 2r," and "delete as 16d." "Fclef as dot" was due to an accidental mouse click, and in "delete as 8d" the mouse button was released prematurely. The example "key as delete" was likely an error caused by the mouse ball not rolling properly on the table. "4u as 8u" and "16d as delete" each have extraneous points at the end of the gesture that are outside the range normally eliminated by the preprocessing. "4r as 16r" is drawn so that the first corner in the stroke is looped (figure 9.5); this causes the accumulated-angle features f_9, f_{10}, and f_{11} to be far from their expected value (see Section 3.3).

Poor mouse tracking. Many of the errors are due to poor tracking of the mouse. Typically, the problem is a long time between the first mouse point of a gesture and the second. This occurs when the first mouse point causes the system to page in the process collecting the gesture; this may take a substantial amount of time. The underlying window manager interface queues up every mouse event involving the press or release of a button, but does not queue successive mouse-movement events, choosing instead to keep only the most recent. Because of this, mouse movements are missed while the process is paged in.

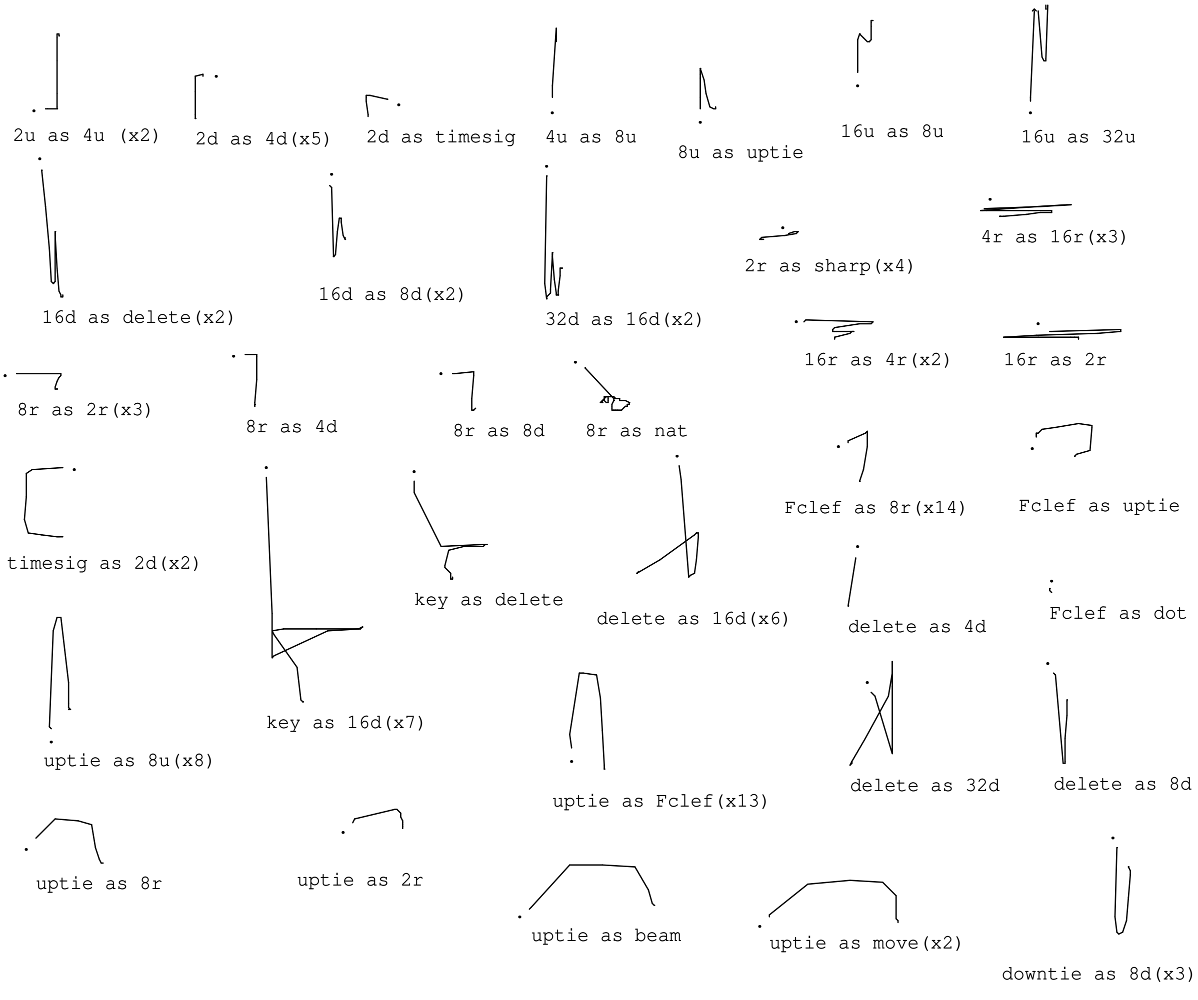


Figure 9.4: Misclassified GSCORE gestures

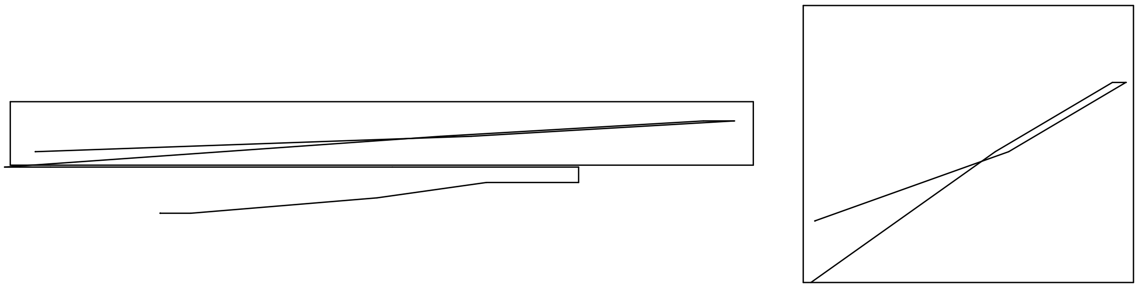


Figure 9.5: A looped corner

The left figure is a magnification of a misclassified “4r as 16r” shown in the previous figure. The portion of the gesture enclosed in the rectangle has been copied and its aspect ratio changed, resulting in the figure on the right. As can been seen, the corner, which should be a simple angle, is looped. This resulted in the angle-based features having values significantly different from the average 4r *gesture, thus the misclassification.*

In “2u as 4u,” “2d as 4d,” “8r as 4d,” “8r as 8d,” and “timesig as 2d” there is no point between the initial point and the first corner, probably due to the paging. This interacts badly with the features f_1 and f_2, the cosine and sine of initial angle. Features f_1 and f_2 are computed from the first and *third* point; this usually results in a better measurement than using the first and second point. In these cases, however, this results in a poor measurement, since the third point is after the corner.

“8r as nat” was the result of a very long page in, during which the author got impatient and jiggled the mouse.

Ambiguous classes. Some classes are very similar to each other, and are thus likely to be mistaken for each other. The 14 misclassifications of “Fclef as 8r” are an example. Actually, these may also be considered examples of poor mouse tracking, since points lost from the normally rounded top of the Fclef gesture caused the confusion. The mistakes “uptie as 8u,” and “uptie as Fclef” are also examples of ambiguity.

Ideally, the gesture classes of an application should be designed so as to be as unambiguous as possible. Given nearly ambiguous classes, it is essential that the input device be as reliable and as ergonomically sound as possible, that the features be able to express the differences, and that the decider be able to discriminate between them. Without all of these properties, it is inevitable that there will be substantial confusion between the classes.

Inadequacy of the feature set. The examples where the second mouse point is the first corner show one way in which the features inadequately represent the classes. For example, the “2r as sharp” examples appear to the system as simple left strokes. Sometimes, a small error in the drawing results in a large error in a feature. This occurs most often when a stroke doubles back on itself; a small change results in a large difference in the angle features f_9, f_{10}, and f_{11} (see figure 9.5). The mistakes “4r as 16r” and “16d as delete” are in this category. “16u as 8u” and “16u as 32u” point to other places where the features may be improved.

The mapping from gestures to features is certainly not invertible; many different gestures might have the same feature vector in the current scheme. This results in ambiguities not due entirely to similarities between classes, but due to a feature set unable to represent the difference. Example "key as 16d" is an illustration of this, albeit not a great one.

Inadequacy of linear, statistical classification. Given that the differences between classes can be expressed in the feature vector, it still may be possible that the classes cannot be separated well by linear discrimination functions. This typically comes about when a class has a feature vector with a severely non-multivariate-normal distribution. In the current feature set, this most often happens in a class where the gesture folds back on itself (as discussed earlier), causing f_9, and thus the entire feature vector, to have a bimodal distribution.

The averaging of the covariance matrix in essence implies that a given feature is equally important in all classes. In the above class, the initial angle features are deemed important by the classifier. When compounded with errors in the tracking, this leads to bad performance on examples such as "uptie as beam" and "uptie as move." It is possible for a linear classifier to express the per-class importance of features in a linear classifier; in essence this is what is done by the neural-network-like training procedures (*a.k.a* back propagation, stochastic gradient, proportional increment, or perceptron training).

Inadequate training data. Drawing and tracking errors occur in the training set as well as the testing set. Given enough good examples, the effect of bad examples on the estimates of the average covariance matrix and the mean feature vectors is negligible. This is not the case when the number of examples per class is very small. Bad or insufficient training data causes bad estimates for the classifier parameters, which in turn causes classification errors. The gestures classified correctly by the $C = 30$, $E = 40$ classifier, but incorrectly by the $C = 30$, $E = 10$ classifier are examples of this.

Analyzing errors in this fashion leads to a number of suggestions for easy improvements to the classifier. Timing or distance information can be used to decide whether to compute f_1 and f_2 using the first two points or the first and third points of the gesture. Mouse events could be queued up to improve performance in the presence of paging. Some new features can be added to improve recognition even in the face of other errors; in particular, the cosine and sine of the final angle of the gesture stroke would help avoid a number of errors. These modification are left for future work, as the author, at the present time, has no desire to redo the above evaluation using 6000 examples from a different gesture set.

One error not revealed in these tests, but seen in practice, is misclassification due to a premature timeout in the two-phase interaction. This results in a gesture being classified before it is completely entered.

9.1.2 Rejection parameters

Section 3.6 considered the possibility of rejecting a gesture, *i.e.* choosing not to classify it. Two parameters potentially useful for rejection were developed. An estimate of the probability that a gesture is classified unambiguously, $\tilde{P}$, is derived from the values of the per-class evaluation

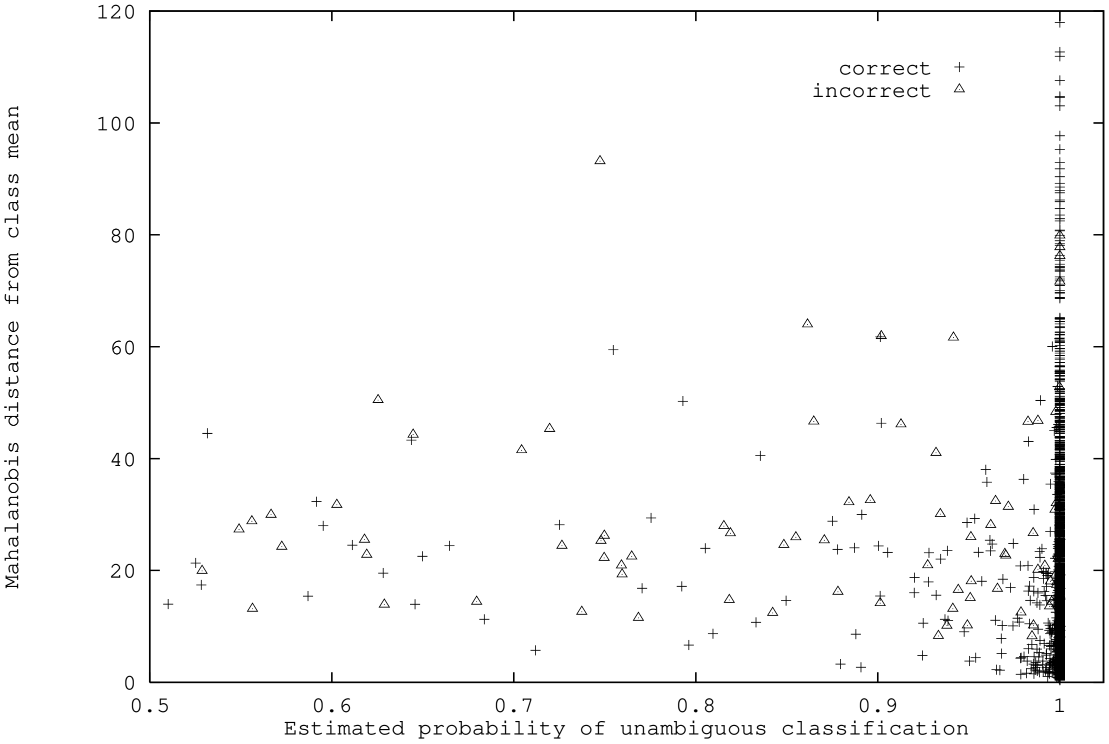


Figure 9.6: Rejection parameters

functions. An estimate of the Mahalanobis distance, d^2, is used to determine how close a gesture is to the norm of its chosen class.

It would be nice if thresholds on the rejection parameters could be used to neatly separate correctly classified example from incorrectly classified examples. It is clear that it would be impossible to do a perfect job; as "delete as 8d" illustrates, the system would need to read the user's mind. The hope is that most of the incorrectly classified gestures can be rejected, without rejecting too many correctly classified gestures.

A little thought shows that any rejection rule based solely on the ambiguity metric $\tilde{P}$ will on the average reject at least as many correctly classified gestures as incorrectly classified gestures. This follows from the reasonable conjecture that the average ambiguous gesture is at least as likely to be classified correctly as not. (This assumes that the gesture is not equally close to three or more classes. In practice, this assumption is almost always true.)

Figure 9.6 is a scatter plot that shows the value for both rejection parameters for all the gestures in the GSCORE test set. A plus sign indicates a gesture classified correctly; a triangle indicates each gesture classified incorrectly, *i.e.* those represented in figure 9.4. Most of the correctly classified examples have an estimated unambiguity probability of very close to one, thus accounting for the dark mass of points at the right of the graph. 96.3% of the correctly classified examples had

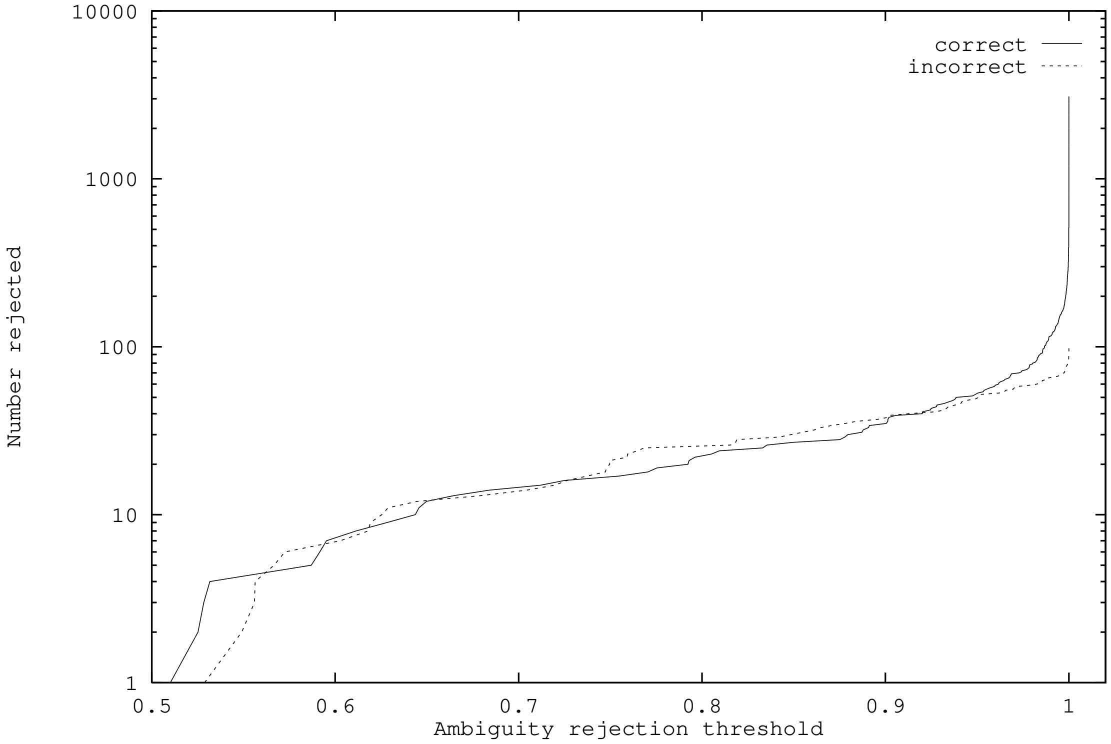


Figure 9.7: Counting correct and incorrect rejections

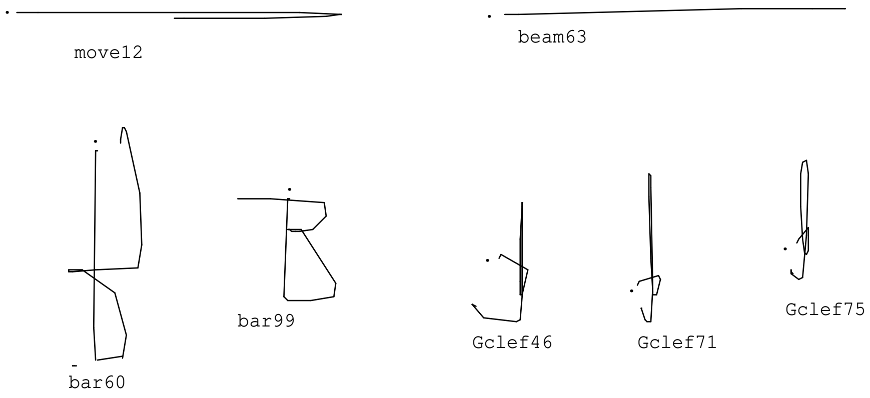


Figure 9.8: Correctly classified gestures with $d^2 \geq 90$

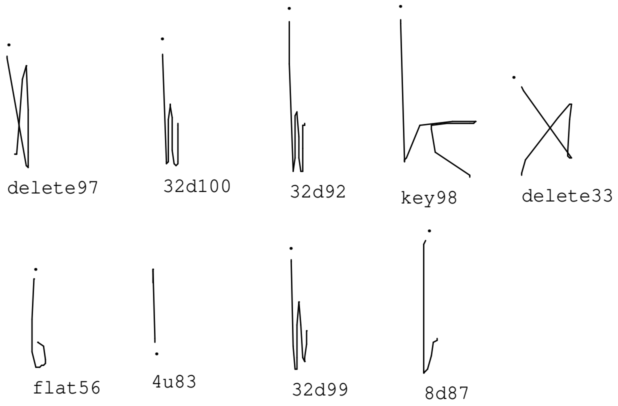


Figure 9.9: Correctly classified gestures with $\tilde{P} \leq .95$

$\tilde{P} \geq 0.99$. However, the same interval contained 33.7% of the incorrectly classified examples. Figure 9.7 shows how many correctly classified and how many incorrectly classified gestures would be rejected as a function of the threshold on $\tilde{P}$.

Examining exactly which of the incorrect examples have $\tilde{P} \geq 0.99$ is interesting. The garbled “8r as nat” and the left stroke “2r as sharp” have $\tilde{P} = 1.0$ within six decimal places. In retrospect this is not surprising; those gestures are far from *every* class, but happen to be unambiguously closest to a single class. This is borne out in the d^2 for those gestures, which is 380 (off the graph) for “8r as nat” and at least 70 for each “2r as sharp” gesture. Other mistakes have $\tilde{P} > .999$ but $d^2 < 20$. In this category are “Fclef as 8r,” “uptie as Fclef,” “delete as 8d,” and “4u as 8u”; these gestures go beyond ambiguity to look like their chosen classes so could not be expected to be rejected.

Also interesting are those correctly classified test examples that are candidates for rejection based on their $\tilde{P}$ and d^2 values. Figure 9.8 shows some GSCORE gestures whose $\tilde{P} = 1$ and $d^2 \geq 90$. Examples “move12” and “beam63” are abnormal only by virtue of the fact that they are larger than normal. The two bar examples have their endpoints in funny places, among other things, while the three Gclef examples are fairly unrecognizable. The algorithm does however classify all of these correctly; and it would be too bad to reject them. Figure 9.9 shows gestures whose ambiguity probability is less that .95. In many of the examples this is caused by at least one corner being made by two mouse points rather than one. In "delete33" one corner is looped. These gestures look so much like their prototypes it would be too bad to reject them.

The Mahalanobis estimate is mainly useful for rejecting gestures that were deliberately entered poorly. This is not as silly as it sounds; a user may decide during the course of a gesture not to go through with the operation, and at that time extend the gesture into gibberish so that it will be rejected.

One possible improvement would be to use the per-class covariance matrix of the chosen class in the Mahalanobis distance calculation. Compared to using the average covariance matrix, this would presumably result in a more accurate measurement of how much the input gesture differs for the norm of its chosen class.

9.1.3 Coverage

Figure 9.10 shows the performance of the single-path gesture recognition algorithm on five different gesture sets. The classifier for each set was trained on fifteen examples per class and tested on an additional fifteen examples per class. The first set, based on Coleman's editor [25], had a substantial amount of variation within each class, both in the training and the testing examples. The remaining sets had much less variation with each class. As the rates demonstrate, the single-path gesture recognition algorithm performs quite satisfactorily on a number of different gesture sets.

9.1.4 Varying orientation and size

One feature that distinguishes gesture from handwriting is that the orientation or size of a gesture in a given class may be used as an application parameter. For this to work, gestures of such classes must be recognized as such independent of their orientation or size. However, the recognition algorithm should not be made completely orientation and size independent, as some other classes may depend on orientation and size to distinguish themselves.

It is straightforward to indicate those classes whose gestures will vary in size or orientation: simply vary the size or orientation of the training examples. The goal of the gesture recognizer is to make irrelevant those features in classes for which they do not matter, while using those feature in classes for which they do.

Theoretically, having some classes that vary in size and orientation, while other that depend on size or orientation for correct classification should be a problem for any statistical classifier based on the assumptions of a multivariate normal distribution of features per class, with the classes having a common covariance matrix. A class whose size is variable is sure to have a different covariance matrix than one whose size remains relatively constant; the same may be said of orientation. Thus, we would suspect the classifier of Chapter 3 to perform poorly in this situation. Surprisingly, this does not seem to be the case.

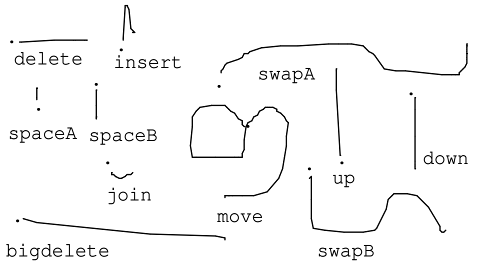
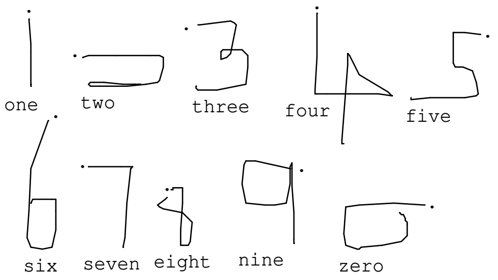
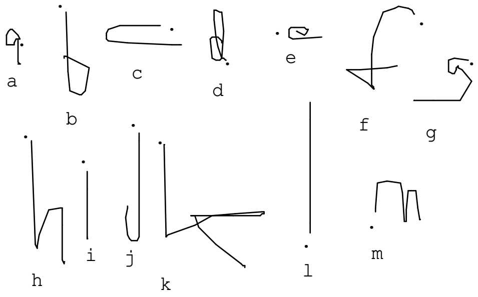
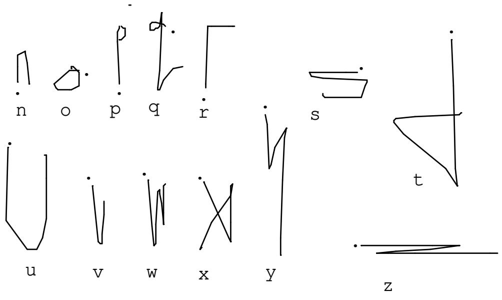
Set Name	Gesture Classes	Number of Classes	Recognition Rate
Coleman		11	100.0%
Digits		10	98.5%
Let:a-m		13	99.2%
Let:n-z		13	98.4%
Letters	Union of Let:a-m and Let:n-z	26	97.1%

Figure 9.10: Recognition rates for various gesture sets

Each set was trained with 15 examples per class and tested on an additional 15 examples per class.

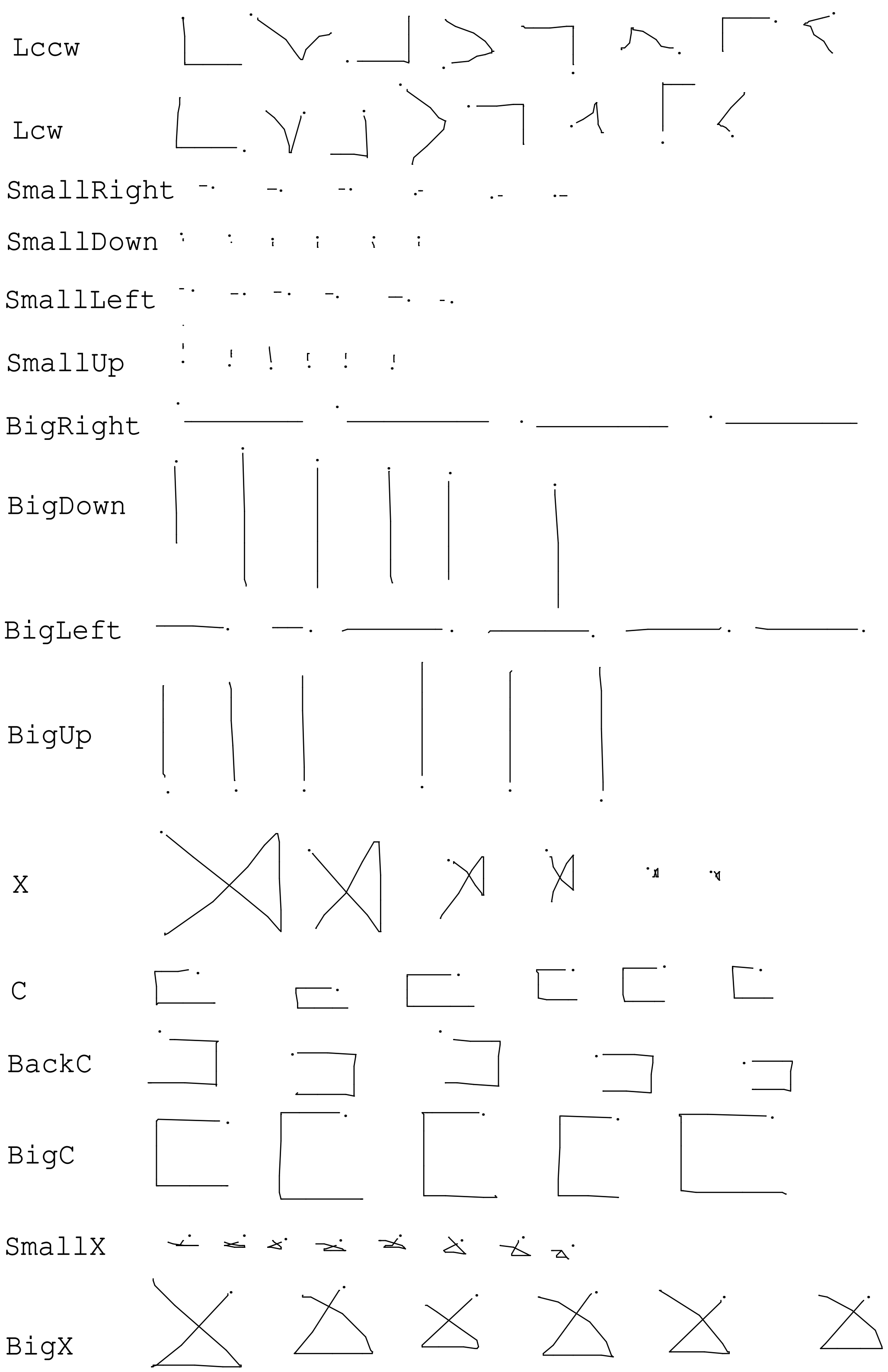


Figure 9.11: Classes used to study variable size and orientation

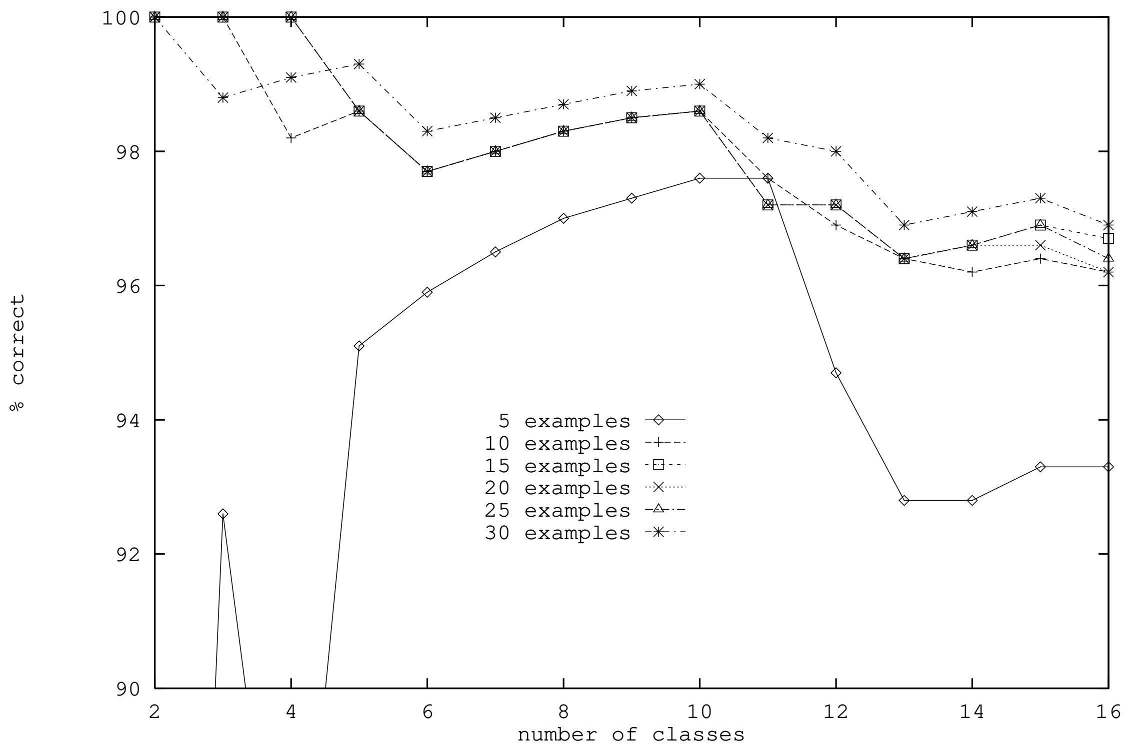


Figure 9.12: Recognition rate for set containing classes that vary

Figure 9.11 shows 16 classes, some of which vary in size, some of which vary in orientation, others of which depend on size or orientation to be distinguishable. The training set consists of thirty examples of each class; variations in size or orientation were reflected in the training examples, as shown in the figure. A testing set with thirty or so examples per class was similarly prepared.

Figure 9.12 shows the recognition rate plotted against the number of classes for various numbers of examples per class in the training set. As can be seen, the performance is good; 96.9% correct on 16 classes trained with 30 examples per class. Using only 15 examples per class results in a recognition rate of 96.7%.

Figure 9.13 shows all the mistakes made by the classifier. None of the mistakes appear to be a result of the size or orientation of a gesture being confused. Rather, the mistakes are quite similar to those seen previously. The conclusion in that the gesture classifier does surprisingly well on gesture sets in which some classes have variable size or orientation, while others are discriminated on the basis of their size or orientation.

9.1.5 Interuser variability

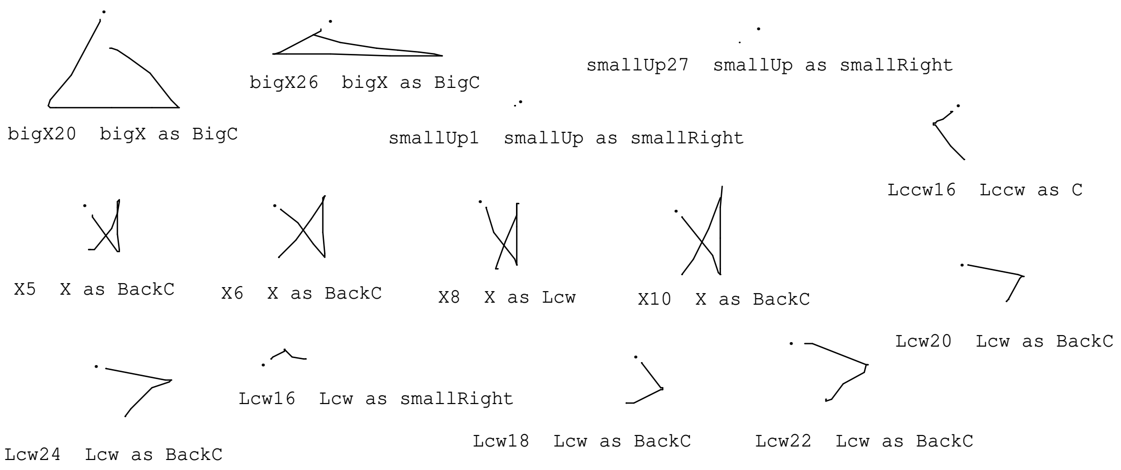
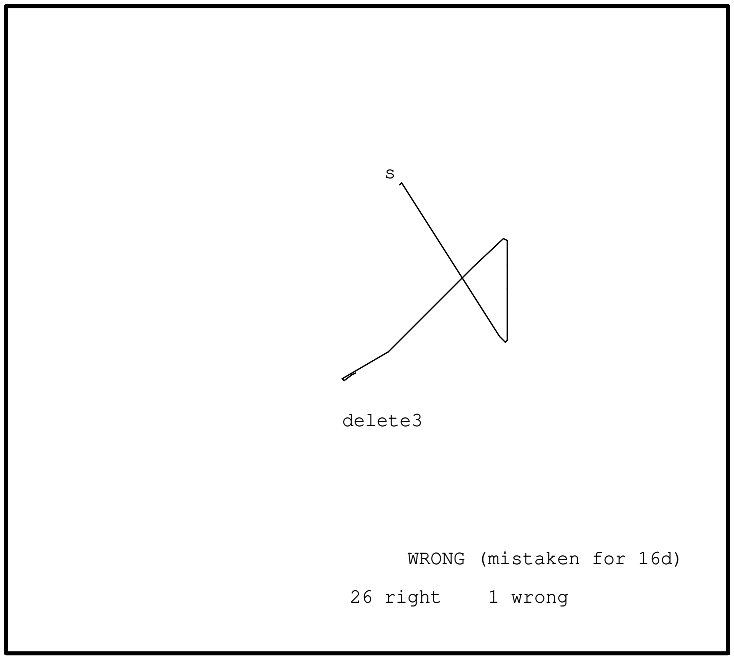


Figure 9.13: Mistakes in the variable class test



```
WRONG (mistaken for 16d)
26 right    1 wrong
```

Figure 9.14: Testing program (user's gesture not shown)

All the gestures shown thus far have been those of the author. It was deemed necessary to show validity of the current work by demonstrating that the gestures of at least one other person could be recognized. Two questions come to mind: what recognition rate can be achieved when a person other than the author gestures at a classifier trained with the author's gestures, and can this rate be improved by allowing the person to train the classifier using his or her own gestures?

Setup

As preparation for someone besides the author actually using the GSCORE application (see Section 9.4.2 below), the GSCORE gesture set (figure 9.1) was used in the evaluation. The hardware used was the same hardware used in the majority of this work, a DEC MicroVAX II running UNIX and X10.

A simple testing program was prepared for training and evaluation (Figure 9.14). In a trial, a prototype gesture of a given class is randomly chosen and displayed on the screen, with the start point indicated. The user attempts to enter a gesture of the same class. That gesture is then classified, and the results fed back to the user. In training mode, if the system makes an error, the trial is repeated. In evaluation mode, each trial is independent.

Subject PV is a music professor, a professional musician, and an experienced music copyist. He is also an experienced computer user, familiar with Macintosh and NeXT computers, among others.

Procedure

The subject was given one half hour of practice with the testing program in training mode. He was also given a copy of figure 9.1 and instructed to take notes at his own discretion. After the half hour, the tester was put in evaluation mode, and two hundred trials run. The test was repeated one week later, without any warmup. The subject was then instructed to create his own gesture set, borrowing from the set he knew as much as he liked. Thirty examples of each gesture class were recorded, and two hundred evaluation trials run on the new set.

Results

During the initial training there was some confusion on the subject's part regarding which hand to use. The subject normally uses his right hand for mousing, but, being left handed, always writes music with his left. After about ten minutes, the subject opted to use his left hand for gesturing.

In the initial evaluation trial the system classified correctly 188 out of 200 gestures. The subject felt he could do better and was allowed a second run, during which 179 out of 200 gestures were correctly classified. By his own admission, he was more "cocky" during the second run, generally making the gestures faster than during the first. The average recognition rate is 91.8%.

After the test, the subject commented that he felt much of his difficulty was due to the fact that he was not used to using the mouse with his left hand, and that the particular mouse felt very different than the one he was used to (NeXT's). He felt his performance would further improve with additional practice.

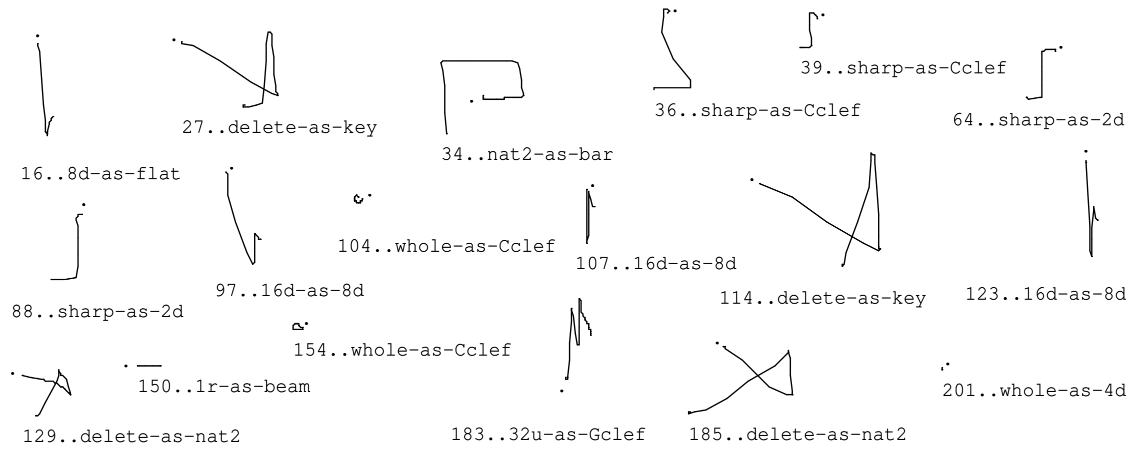


Figure 9.15: PV's misclassified gestures (author's set)

His notes are interesting. Although the subject had no particular knowledge of the recognition algorithm being used, in many cases his notes refer to the particular features used in the algorithm. For gestures whole and sharp he wrote "start up" and "don't begin too vertically" respectively, noting the importance of the correct initial angle. For 1r he wrote "make short," for bar he wrote "make large," and for delete he wrote "make quickly." For 2u and 2d he wrote "sharp angle."

The subject commented on places where the gesture classes used did not conform to standard copyist strokes. For example, he stated the loop in flat goes the wrong way. He explained that many music symbols are written with two strokes, and said that he might prefer a system that could recognize multiple-stroke symbols.

When the test was repeated a week later, the subject, without any warmup, achieved a score of 183 out of 200, 91.5%. Figure 9.15 shows the misclassified gestures. The subject was again unsure of which hand to use, but used his left hand at the urging of the author.

The subject then created his own gesture set, examples of which are shown in figure 9.16. A training set consisting of 30 examples of each class was entered. Running the training set through the resulting classifier resulted in the rather low recognition rate of 94.7% (by comparison, running the author's training set through the classifier it was used to train yielded 97.7%.) The low rate was due to the some ambiguity in the classes (*e.g.* "flat" and "16d" were frequently confused) as well as many classes where the corners were looped (as seen before in section 9.1.1), causing a bimodal distributions for f_9, f_{10}, and f_{11}

The problems in the new gesture set nonwithstanding, PV ran two hundred trials of the tester on the new set. He was able to get a score of 186 out of 200, 93%.

At the time of this writing, PV has not yet made the attempt to remove the ambiguities from the new gesture set and to be more careful on the sharp corners.

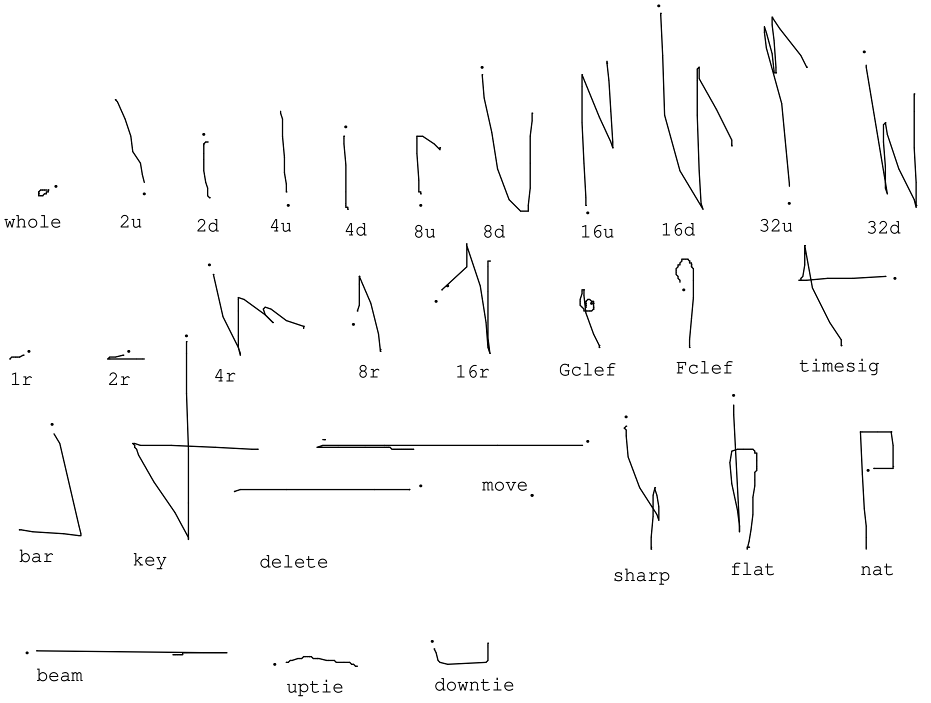


Figure 9.16: PV's gesture set

Conclusion

It is difficult to draw a conclusion given data from only one subject. The author expected the recognition rate to be higher when PV trained the system on his own gestures than when he used the author's set. The actual rate *was* slightly higher, but not enough to make a convincing argument that people do better on their own gestures (some slightly more convincing evidence is presented in section 9.4.2 below). In retrospect, PV should have created a training set that copied the author's gestures before attempting to significantly modify that gesture set. The author's gesture set turned out to be better designed than PV's, in the sense of having less inherent ambiguities; this tended to compensate for any advantage PV gained from using his own gestures.

However, PV's new gesture set is not without merit; on the contrary, it has a number of interesting gestures. The new delete gesture, a quick, long, leftward stroke, gives the user the impression of throwing objects off the side of the screen. The new move gesture is like a delete followed by a last minute change of mind. The flat gesture is much closer to the way PV writes the symbol, as are the leftward whole and half rests gestures 1r and 2r. The stylized "4" for timesig is clever, as is the way it relates to key. PV's bar gesture is much more economical that the author's.

The experiments indicate that a person can use a classifier trained on another person's gesture with moderately good results. Also indicated is that people can create interesting gesture sets on their own. Some modification to the feature set also seems desirable, mainly to make the features less sensitive to “looped” corners. It would be useful to give more feedback to the gesture design as to which classes are confusable. This should be simple to do simply by examining the Mahalanobis distance between every pair of classes.

9.1.6 Recognition Speed

It is well known that a user interface must respond quickly in order to satisfy users; thus for gesture-based systems the speed of recognition is an important factor in the usability of the system. This section reports on measurements of the speed of the components of the recognition process.

The statistical gesture recognizer described in Chapter 3 was designed with speed in mind. Each feature is incrementally calculated in constant time; thus $O(F)$ work must be done per mouse point, where F is the number of features. Given a gesture of P mouse points, it thus takes $O(PF)$ time to compute its feature vector. The classification computes a linear evaluation function over the features for each of C classes; thus classification take $O(CF)$ time.

Feature calculation

The abstract datatype `FV` is used to encapsulate the feature calculation as follows:

`FV FvAlloc()` allocates an object of type `FV`. A classifier will generally call `FvAlloc()` only once, during program initialization.

`FvInit(fv)` initializes `fv`, an object of type `FV`. `FvInit(fv)` is called once per gesture, before any points are added.

`FvAddPoint(fv, x, y, t)` adds the point (x, y) which occurs at time t to the gesture. `FvAddPoint` performs the incremental feature calculation. It is called for every mouse point the program receives. There are thirteen features calculated ($F = 13$).

`Vector FvCalc(fv)` returns the feature vector as an array of double precision floating point numbers. It performs any necessary calculations needed to transform the incrementally calculated auxiliary features into the feature set used for classification. It is called once per gesture.

The function `CalcFeatures(g)` represents the entire work of computing the feature vector for a gesture that is in memory:

```
Fv fv;          /* allocated via FvAlloc() elsewhere */
Vector
CalcFeatures(g)
register Gesture g;
{
    register Point p;
    FvInit(fv);
    for(p = g->point; p < &g->point[g->npoints]; p++)
        FvAddPoint(fv, p->x, p->y, p->t);
```

Processor	Time(sec)	Relative Speed
MicroVAX II	227.95	0.76
VAX 11/780	172.20	1.0
MicroVAX III	60.97	2.8
PMAX-3100	11.30	15

Table 9.1: Speed of various computers used for testing

Processor	milliseconds per call		
	`FvAddPoint`	`FvCalc`	`CalcFeatures`
MicroVAX II	0.22	0.34	3.9
MicroVAX III	0.074	0.13	1.3
PMAX-3100	0.029	0.040	0.44

Table 9.2: Speed of feature calculation

```
        return FvCalc(fv);
    }
```

To obtain the timings, the testing set of Section 9.1.1 was read into memory, and then each gesture was passed to `CalcFeatures`. Three processors were used: the DEC MicroVAX II that was used for the majority of the work reported in this dissertation, a DEC MicroVAX III, and a DEC PMAX-3100 (to get an idea of the performance on a more modern system). The UNIX profiling tool was used to obtain the times. In all cases, the times are virtual times, *i.e.* the time spent executing the program by the processor. All tests were run on unloaded systems, and the real times were never more than 10% more than the virtual times.

Before timing any code related to gesture recognition, the following code fragment (compiled with “cc -O”) was timed on a number of processors, MicroVAX II, VAX 11/780, MicroVAX III, and PMAX-3100, in order to compare the speed of the processors used in the following tests to that of a VAX 11/780:

```
register int  i, n = 1000000;
double s, a[15], b[15];
for(i = 0; i < 15; i++) a[i] = i, b[i] = i*i;
do {
    s = 0.0;
    for(i = 0; i < 15; i++) s += a[i] * b[i];
} while(--n);
```

The times for the above fragment shown in table 9.1.

Note that on this code fragment the PMAX-3100 runs about 20 times faster than the MicroVAX II. On more typical code, it usually runs only 10-15 times faster.

The testing set averaged 13.4 points per gesture. The timings for the routines that calculate features are shown in table 9.2.

The cost per mouse point to incrementally process a mouse point is a small fraction of a millisecond, even on the slowest processor. Since mouse points typically come no faster than 40

Processor	Computation time (milliseconds)				
	$v^{\hat{c}}$ (one class)	max $v^{\hat{c}}$ (30 classes)	$\tilde{P}$	d^2	total
MicroVAX II	0.27	8.0	0.8	3.7	12.6
MicroVAX III	0.074	2.2	0.3	1.1	3.6
PMAX-3100	0.022	0.66	0.01	0.26	0.99

Table 9.3: Speed of Classification

per second, only a small fraction of the processor is consumed incrementally calculating the feature vector. Indeed, substantially more of the processor is consumed communicating with the window manager to receive the mouse point and perform the inking.

Classification

Once the feature vector is calculated it must be classified. This involves computing a linear evaluation function $v^{\hat{c}}$ on F features ($F = 13$) for each of C classes. If the rejection parameters are desired, it takes an additional $O(C)$ work to estimate the ambiguity $\tilde{P}$ and $O(F^2)$ work to estimate the Mahalanobis distance d^2. The computation times for each of these is shown in table 9.3.

To get these times, four runs were made. Every gesture in the testing set was classified in every run. The first run did not calculate either rejection parameter. The average time to classify a gesture as one of thirty classes is reported the $maxv^{\hat{c}}$ column; the $v^{\hat{c}}$ column is computed as $\frac{1}{30}$ of that time. (The $v^{\hat{c}}$ column thus gives the time to compute the evaluation function for a single class; multiply this by the number of classes to estimate the classification time of a particular classifier.) The second run computed $\tilde{P}$ after each classification; the difference between that time and the max $v^{\hat{c}}$ time is reported in the $\tilde{P}$ column. The third run computed d^2 and is reported similarly. The fourth run computed both $\tilde{P}$ and d^2; the average time per gesture is reported in the "total" column.

For a 30-class discrimination with both rejection parameters being used, after the last mouse point of a gesture is entered it takes a MicroVAX II 13 milliseconds to finish calculating the feature vector (`FvCalc`) and then classify it. This is acceptable, albeit not fantastic, performance. If the end of the gesture is indicated by no mouse motion for a timeout interval, the classification can begin before the timeout interval expires, and the result be ignored if the user moves the mouse before the interval is up.

Currently, all arithmetic is done using double precision floating point numbers. There is no conceptual reason that the evaluation functions could not be computed using integer arithmetic, after suitably rescaling the features so as not too lose much precision. The resulting classifier would then run much faster (on most processors). This has not been tried in the present work.

If eager recognition is running, classification must occur at every mouse point, and the number of classes is $2C$. This puts a ceiling on the number of the classes that the eager recognizer can discriminate between in real-time. On a MicroVAX II, the cost per mouse point includes `FvAddPoint` (0.22 msec) plus `FvCalc` (0.34 msec) plus the per class evaluation of $2C$ classes, $0.54C$. If mouse points come at a maximum rate of one every 25 milliseconds, $C = 45$ classes would consume the entire processor. Practically, since there is other work to do (*e.g.* inking), $C = 20$ is probably the maximum that can be reasonably expected from an eager recognizer on a MicroVAX II. On today's processors, instead of computation time, the limiting factor will be the lower recognition rate when given a large number of classes.

One approach tried to increase the number of classes in eager recognizers was to use only a subset of the features. While this improved the response time of the system, the performance degraded significantly, so the idea was abandoned. There is no point getting the wrong answer quickly.

9.1.7 Training Time

The stated goal of the thesis work is to provide tools that allow user interface designers to experiment with gesture-based systems. One factor impacting on the usability of such tools is the amount of time it takes for gesture recognizers to retrain themselves after changes have been made to the training examples. In almost all trainable character recognizers, deleting even a single training example requires that the training be redone from scratch. For some technologies, notably neural networks, this retraining may take minutes or even hours. Such a system would not be conducive to experimenting with different gesture sets.

By contrast, statistical classifiers of the kind described in Chapter 3 can be trained very rapidly. Training the classifier from scratch requires $O(EF)$ to compute the mean feature vectors, $O(EF^2)$ time to calculate the per-class covariance matrices, $O(CF^2)$ to average them, $O(F^3)$ to invert the average, and $O(CF^2)$ to compute the weights used in the evaluation functions. If the average covariance matrix is singular, an $O(F^4)$ algorithm is run to deal with the problem.

Often, a fair amount of work can be reused in retraining after a change to some training examples. Adding or deleting an example of a class requires $O(F)$ work to incrementally update its per-class class mean vector, and $O(F^2)$ work to incrementally update its per-class covariance matrix [137]. Retraining then involves repeating the steps starting from computing the average covariance matrix. Thus, for retraining, the dependency on E, the total number of examples, is eliminated. The retraining time is instead a function of the number of examples added or deleted.

The Objective C implementation does not attempt to incrementally update the per-class covariance matrix when an example is added. Instead, only the averages are kept incrementally, and the per-class covariance matrix is recomputed from scratch. This involves $O(E^{\hat{c}}F^2)$ work for each class c changed, where $E^{\hat{c}}$ is the number of training examples for class c. This results in worse performance when a small number of examples are changed, but better performance when all the examples of a class are deleted and a new set entered. The latter operation is common when experimenting with gesture-based systems.

The author has implemented both C and Objective C versions of the single path classifier. Besides maintaining the per-class covariance matrices incrementally, the C version differs in that it does not store the list of examples that have been used to train it. (It is not necessary to store the list to add and remove examples, since the mean vector and covariance matrix are updated incrementally.) It is thus more efficient since it does not need to maintain the lists of examples. (Objective C's `Set` class, implemented via hashing, is used to maintain the lists in that version.) It also does not have the overhead of separate objects for examples, classes and classifiers that the Objective C version has (see Section 7.5).

Processor	Time (milliseconds per call)			
	`sAddExample`	`sRemoveExample`	`sDoneAdding` (10 classes)	`sDoneAdding` (30 classes)
MicroVAX II	3.7	3.8	130	234
MicroVAX III	0.90	0.90	43	78
PMAX-3100	0.024	0.026	14	22

Table 9.4: Speed of classifier training

Since only the C version could be ported to the PMAX-3100, it was used for the timings. (C versions of the feature computation and gesture recognition were used for the timings above; however in these cases the Objective C methods are straightforward translations of their corresponding C functions. In some cases, the methods merely call the corresponding C function.) The following C functions encapsulate the process of training a classifier:

`sClassifier sNewClassifier()` allocates and returns a handle to a new classifier. Initially it has no classes and no examples. The "s" at the beginning of the type and function names refers to "single-path"; there are corresponding types and functions for the multi-path classifiers.

`sAddExample(sClassifier sc, char *classname, Vector e)` adds the training example (feature vector `e`) to the named class `classname` in the passed classifier. The class is created if it has not been seen before. Linear search is used to find the class name; however, it is optimized for successive calls with the same name. The `sAddExample` function incrementally maintains the per-class mean vectors and covariance matrices.

`sRemoveExample(sClassifier sc, char *classname, Vector e)` removes example `e`, assumed to have been added earlier, from the named class. The per-class mean vector and covariance matrix are incrementally updated.

`sDoneAdding(sClassifier sc)` trains the classifier on its current set of examples. It computes the average covariance matrix, inverts it (fixing it if singular), and computes the weights.

`sClass sClassify(sClassifier sc, Vector e, double *p, *d2)` actually performs the classification of `e`. If `p` is non-`NULL` the probability of ambiguity is estimated; if `d2` is non-`NULL` the estimated Mahalanobis distance of `e` to its computed class is returned. This is the function timed in the previous section.

The functions were exercised first by adding every example in the training set, training the classifier, and then looping, removing and then re-adding 10 consecutive examples before retraining. No singular covariance matrix was encounted, due to the large number of examples. Table 9.4 shows the performance of the various routines.

Even on a MicroVAX II, training a 30 class classifier once all the examples have been entered takes less than a quarter second. Thus GRANDMA is able to produce a classifier immediately the first time a gesture is made over a set of views whose combined gesture set has not been encountered before (see Sections 7.2.2 and 7.4). The user has to wait, but does not have to wait long.

9.2 Eager recognition

This section evaluates the effectiveness of the eager recognition algorithm on several single-stroke gesture sets. Recall that eager recognition is the recognition of a gesture while it is being made, without any explicit indication of the end of the gesture. Ideally, the eager recognizer classifies a gesture as soon as enough of it has been seen to do so unambiguously (see Chapter 4).

In order to determine how well the eager recognition algorithm works, an eager recognizer was created to classify the eight gestures classes shown in 9.17. Each class named for the direction of its two segments, e.g. ur means "up, right." Each of these gestures is ambiguous along its initial segment, and becomes unambiguous once the corner is turned and the second segment begun.

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

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

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

Figure 9.18 shows the performance of the eager recognizer on GDP gestures. The eager recognizer was trained with 10 examples of each of 11 gesture classes, and tested on 30 examples of each class, five of which are shown in the figure. The GDP gesture set was slightly altered to increase eagerness: the group gesture was trained clockwise because when it was counterclockwise it prevented the copy gesture from ever being eagerly recognized. For the GDP gestures, the full classifier had a 99.7% correct recognition rate as compared with 93.5% for the eager recognizer. On the average 60.5% of each gesture was examined by the eager recognizer before classification occurred. For this set no attempt was made to determine the minimum average gesture percentage that needed to be seen for unambiguous classification.

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

Computationally, eager recognition is quite tractable on modest hardware. A fixed amount of

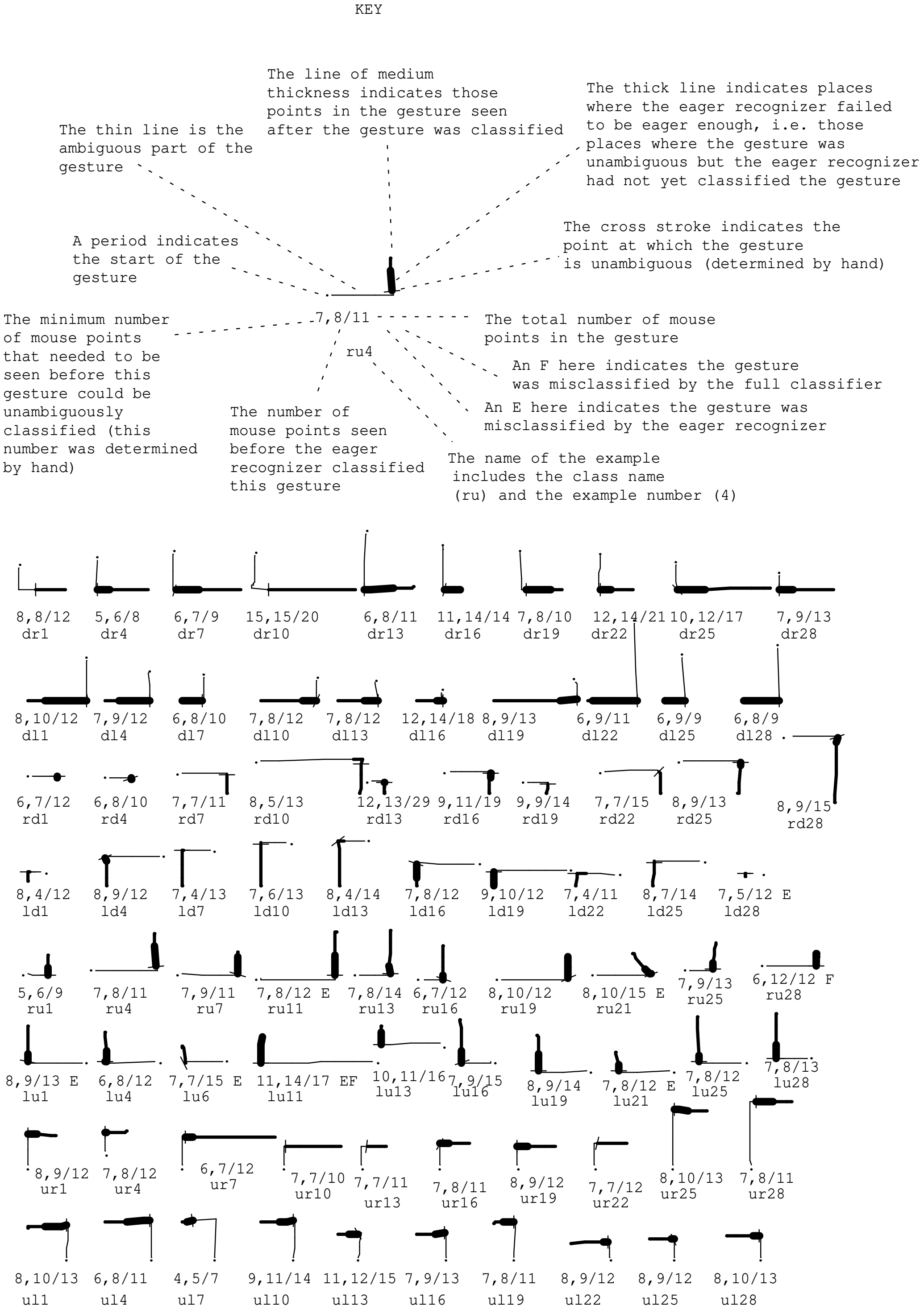


Figure 9.17: The performance of the eager recognizer on easily understood data

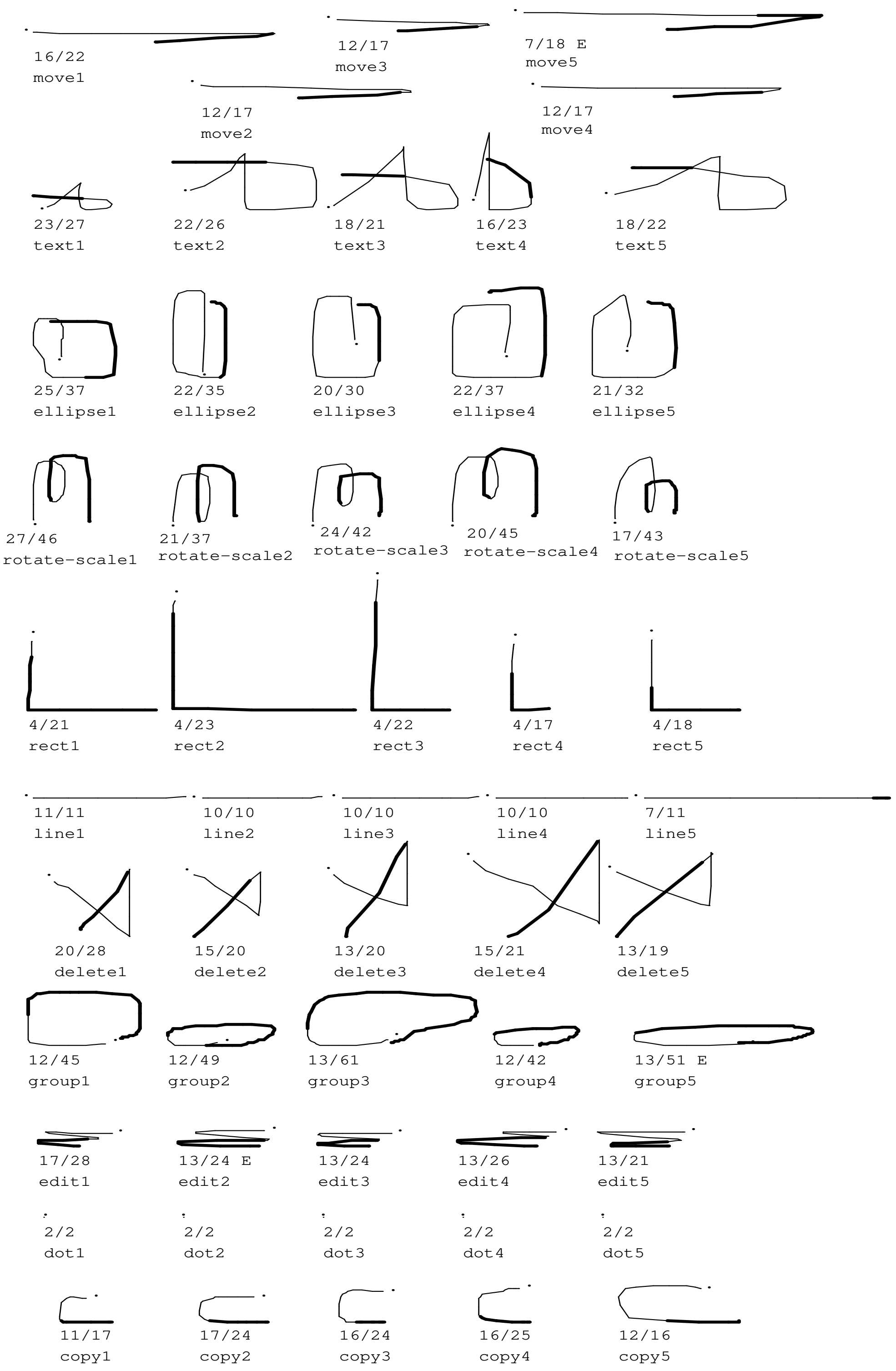


Figure 9.18: The performance of the eager recognizer on GDP gestures

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

computation needs to occur on each mouse point: first the feature vector must be updated (taking 0.5 msec on a DEC MicroVAX II), and then the vector must be classified by the AUC (taking 0.27 msec per class, or 6 msec in the case of GDP).

9.3 Multi-finger recognition

Multi-finger gestural input is a significant innovation of this work. Unfortunately, circumstances have conspired to make the evaluation of multi-finger recognition both impossible and irrelevant. The Sensor Frame is the only input device upon which the multi-finger recognition algorithm was tested. Unfortunately, there is only one functioning Sensor Frame in existence, and that was damaged sometime after the multi-finger recognition was running, but before formal testing could begin. (Fortunately, a videotape of MDP in action was made while the Sensor Frame was working.) No progress was made in repairing the Sensor Frame for over a year; testing was thus impossible. Eventually the Sensor Frame was repaired, but Sensor Frame, Inc. went out of business shortly afterward, making any detailed evaluation irrelevant. The owner of the Sensor Frame has left the country, taking the device with him.

An informal estimate of the multi-finger recognition accuracy may be estimated from ten minutes of videotape of the author using MDP. This version of MDP uses the path sorting multi-finger recognition algorithm (Section 5.2). As shown in figure 8.10, MDP recognizes 11 gestures (6 one finger gestures, 3 two finger gestures, and 2 three finger gestures). In the videotape, the author made 30 gestures, 2 of which appear to have been misclassified, and one of which was rejected, resulting in a correct recognition rate of 90%. The processing time appears to be negligible.

All three misclassifications are the result of the Sensor Frame seeing more fingers in the gesture than were intended. This was due to knuckles of fingers curled up (so as not to be used in the gesture) accidentally penetrating the sensing plane and being counted as additional fingers. As there are distinct classifiers for single finger, two finger, and three finger gestures, an incorrect number of fingers inevitably leads to a misclassification. While it is possible to imagine methods for dealing with such errors during recognition, the main cause of this problem is the ergonomics of the Sensor Frame.

For the small gesture set examined, the recognition rate is 100% once the errors due to "extra fingers" are eliminated. This is to be expected, given the small number of gestures for each number of fingers. It is expected that the multi-path classifier operating on one finger gestures would perform about as well as the single-path classifier, as the algorithms are essentially identical. The single-path classifier, when given only six classes to discriminate among, has been shown (on mouse data) to perform at 100% in almost all cases. When operating on two finger gestures, it is expected that the performance of the recognition algorithm would be similar to that of the single-path classifier on *twice* the number of classes. Actually, it is possible that some of the paths in the two-finger gestures will be similar to other paths in the set, and be merged into a single class by the training algorithm (Section 5.4). Thus, when the number of unique paths will be less than twice the number of two-finger gesture classes, performance may be expected to improve accordingly. Similarly, the three finger gesture classifier may be expected to perform as well as a single-path classifier the recognizes between one and three times the number of three finger gesture classes, depending on the number of unique paths in the class set.

One more factor to consider is that mouse data tends to be much less noisy than Sensor Frame data. The triangulation by the Sensor Frame is erratic, especially when multiple fingers are being tracked. For example, both horizontal segments of the Parallelogram gesture of figure 8.10 should be straight lines. Until this problem can be solved, it is expected that recognition rates for Sensor Frame gesture sets will suffer.

9.4 GRANDMA

Evaluating GRANDMA is much more subjective than evaluating the low-level recognition rates. GRANDMA may be evaluated on several levels: the effort required to build new interaction techniques, to build new applications, to add gestures to an application, to change an application's gestures, or to use an application to perform a task.

No attempt was made to formally evaluate any of these. In order to get statistically valid results, it would have been necessary to run carefully designed experiments on a number of users, something the author had neither the time, space, inclination, or qualifications to do. Furthermore, the author does not wish to claim that GRANDMA is superior to existing object-oriented toolkits for any particular task. GRANDMA is simply the platform through which some ideas for input processing in object-oriented toolkits were explored. GRANDMA's significance, if any, will be its influence on future toolkits, rather than any more direct results.

Nonetheless, this section informally reports on the author's experience building gesture-based systems with GRANDMA. (No one besides the author tried to program with GRANMDA.) Also, in order to confirm that GRANDMA can be used by someone other than the author, this section also reports on observations of a subject trying to use GSCORE and GRANDMA to due some tasks.

9.4.1 The author's experience with GRANDMA

GRANDMA took approximately seven months to design and develop. It consists of approximately 12000 lines of Objective C code. There are an additional 5000 lines of C code which implement a graphics layer as well as the feature vector calculation. GDP took an additional 2000 lines of Objective C code to implement. GDP was developed at the same time as GRANDMA, as it was the primary application used to test GRANDMA.

Initially, only two GDP gestures were used to test GRANDMA's gesture handler and associated utilities. Once these were working well, it took four days to add the remaining gestures to GDP. Most of this time was spent writing Objective C methods to use in semantic expressions. These were methods that were not needed for the existing direct manipulation interface.

GSCORE consists of 6000 lines of Objective C code. It took six weeks to design and implement GSCORE, including its palette-based interface. Much of this time was spent on the details of representing common music notation, mechanisms for displaying music notation, and producing usable music fonts. The palette, an interaction technique that did not as yet exist in GRANDMA, took about eight hours to implement. It took two weeks to add the gestural interface to GSCORE,

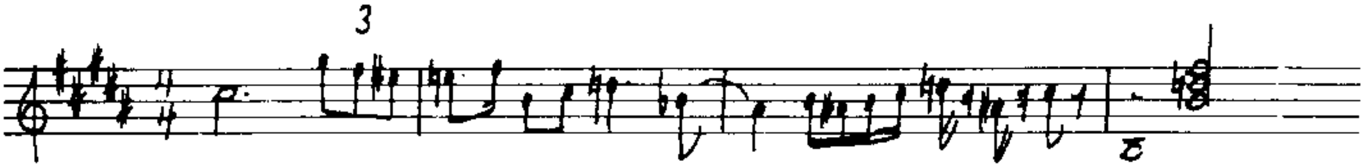



Figure 9.19: PV's task

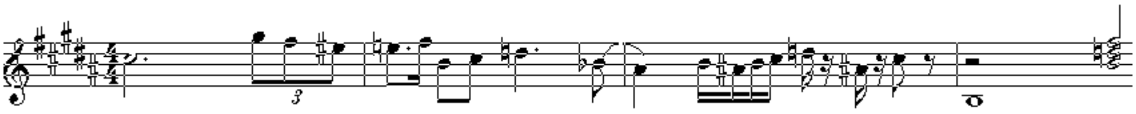



Figure 9.20: PV's result

including writing some additional methods. Much of this time was spent experimenting with different semantics for the gestures.

Section 10.1.3 lists features of GRANDMA that will be important to incorporate into future toolkits that support gestures.

9.4.2 A user uses GSCORE and GRANDMA

This sections informally reports on subject PV's (see Section 9.1.5) attempts to use the GSCORE program.

The task was to enter the music shown in figure 9.19. The music was chosen to exercise many of the GSCORE gestures. PV is an experienced music copyist, and it took him 100 seconds to write out the music as shown, copying it from an earlier attempt.

Using gestures, the author was able to enter the above score in 280 seconds (almost five minutes). He made a total of 53 gestures, four of which did not give the desired results and were immediately undone. Only two of those were misclassifications; the other two were notes gestures where the note was created having the wrong pitch, due to misplacement of the cursor at the start of the gesture. Turning off gestures and using only the palette interface, it took the author 670 seconds (eleven minutes). No mistakes needed to be undone in the latter trial.

PV's first attempt was at using the GSCORE program trained with the author's gestures. PV had already gained experienced with this set of gestures during the study of interuser variation. PV practiced for one half hour with the GSCORE program before attempting the task. The author coached PV during this time, as no other documentation or help was available.

PV took 600 seconds (10 minutes) to complete the task. He made a total of 73 gestures, 16 of which were immediately undone. It appeared to the author, who was silently observing, that each undo was used to recover from a misclassification. Figure 9.20 shows the product of his labor. PV then turned off gestures, and used the palette interface to enter the example. He completed the task in 680 seconds (11 minutes).

PV then entered his own gestures in place of some of the author's. In particular, he substituted his own gestures for the nine classes: delete, move, beam, 1r, 2r, 8r, 16r, keysig, and bar, entering 15 or more examples of each. The total time to do this, including incremental testing of the new gestures and periodic saves to disk, was 25 minutes. PV did not attempt to emulate the author's gestures; instead, he used the forms he had created earlier (see Section 9.1.5).

Once done, PV repeated the experiment. It took him 310 seconds (5 minutes) to enter the music. He made 58 gestures, 4 of which were undone.

PV was interviewed after the tests, and made the following comments: The biggest problem, he stated, was that mouse tracking in the GSCORE program was much more sluggish than in the recorder and tester. (This is accurate, as the time required for GSCORE events to be created and consumed adds significant overhead to the mouse tracking. Much of this is overhead due to GRANDMA.) PV characterized the system as "sluggish." Bad tracking, especially at the start of the gesture, contributed significantly to the number of misclassifications.

PV stated that he thought the system "intuitive" for entering notes. He described the gesture-based interface as "excellent" compared to the palette-based system, but when asked how the gesture-based interface compared to writing on paper, he replied "it sucks." He did not like using the mouse for gesturing, and believed that a stylus and tablet would be much better.

It is again difficult to draw conclusions from an informal study of one user. Did PV's performance improve because he tailored the gestures to his liking, or because he had been practicing? This is unknown. Some things are clear: GRANDMA makes it easy to experiment with new gesture sets, and, in GSCORE, with moderate practice the gesture-based interface improved task performance by a factor of two over the palette-based interface. Whether gesture-based interfaces generally improve task performance over non-gesture-based interfaces is a question that requires *much* further study.

Chapter 10

Conclusion and Future Directions

This chapter summarizes the contributions of this thesis and indicates some directions for future work.

10.1 Contributions

This thesis makes contributions in four areas:

- New interaction techniques
- New recognition-related algorithms
- Integrating gestures into interfaces
- Input in object-oriented toolkits

Each of these will be discussed in turn.

10.1.1 New interactions techniques

A major contributions of this work has been the invention and exploration of three new interaction techniques.

The two-phase single-stroke interaction The two-phase interaction enables gesture and direct manipulation to be integrated in a single interaction that combines the power of each. The first phase is collection, during which the points that make up the gesture are collected. In the simplest case, the end of the collection phase is indicated by a motion timeout, classification occurs, and the second phase, manipulation, is entered. In the manipulation phase, the user moves the mouse to manipulate some parameters in the application. The particular parameters manipulated depend on the classification of the collected gesture. The collection phase is like character entry in handwriting interfaces; the manipulation phase is like a drag interaction in direct-manipulation interfaces. Generally, the operation, operands, and some parameters are determined at the phase transition (when the gesture is recognized), and then the manipulation phase allows additional parameters to be set in the presence of application feedback.

Eager recognition Eager recognition is a modification of the two-phase single-stroke interaction in which the phase transition from gesturing to manipulation occurs as soon as enough of the gesture has been seen so that it may be classified unambiguously. The result is an interaction that combines gesturing and direct manipulation in a single, smooth interaction.

The two-phase multiple-finger interaction Gesture and direct manipulation may be combined for multiple path inputs in a way similar to the two-phase single-stroke interaction. With multiple finger input, opportunities exist for expanding the power of each phase of the interaction. By allowing multiple fingers in the collection phase, the repertoire of possible gestures is greatly increased, and a multiple finger gesture allows many parameters to be specified simultaneously when the gesture is recognized. Similarly, even when only one finger is used for the gesture, additional fingers may be brought in during the manipulation phase. Thus, the two-phase multiple-finger interaction allows a large number of parameters to be specified and interactively manipulated.

10.1.2 Recognition Technology

This thesis discloses five new algorithms of general utility in the construction and use of gesture recognizers.

Automatic generation of single-stroke gesture recognizers from training examples A practical and efficient algorithm for generating gesture recognizers has been developed and tested. In it, a gesture is represented as a vector of real-valued features, and a standard pattern recognition technique is used to generate a linear classifier that discriminates between the vectors of different gesture classes. The training algorithm depends on aggregate statistics of each gesture class, and empirically it has been shown that usually only fifteen examples of each class are needed to produce accurate recognizers. It is simple to incorporate dynamic attributes, such as the average speed of the gesture, into the feature set. The algorithm has been shown to work even when some classes vary in size and orientation while others depend on size or orientation to be recognized. The recognizer size is independent of the number of training examples, and both the recognition and training times have been shown to be small. A features set that is both meaningful and extensible potentially allows the algorithm to be adapted to future input devices and requirements.

Incremental feature calculation The calculation used to generate features from the input points of a gesture is incremental, meaning that it takes constant time to update the features given a new input point. This allows arbitrarily large gestures to be processed with no delay in processing.

Rejection algorithms Two algorithms for rejecting ill-formed gestures have been developed and tested. One estimates the probability of correct classification, enabling input gestures that are ambiguous with respect to a set of gesture classes to be rejected. The other uses a normalized distance metric to determine how close an input gesture is to the typical gesture of the its computed class, allowing outliers to be rejected.

Automatic generation of eager recognizers from training examples An eager recognizer classifies a gesture as soon as it is unambiguous, alleviating the need for the end of the gesture to be explicitly indicated. An algorithm for generating eager recognizers from training examples has been developed and tested. The algorithm produces a two-class classifier which is run on every input point and used to determine if the gesture being entered is unambiguous.

Automatic generation of multi-path gesture recognizers The single-stroke recognition work has been extended so that a number of single-stroke recognizers may be combined into a multi-finger gesture recognizer. The described algorithm produces a multi-path recognizer given training examples. Relative path timing information is considered during the recognition, and global classification is attempted when the individual path classifications do not uniquely determine the class of the multi-path gesture. For dealing with the problems that arise from multi-path input devices that do not *a priori* determine "which path is which," two approaches, path sorting and path clustering, have been explored. The resulting algorithm has been demonstrated using the Sensor Frame as a multi-finger input device.

10.1.3 Integrating gestures into interfaces

A paradigm for integrating gestures into object-oriented interfaces has been developed and demonstrated. The key points are:

A gesture set is associated with a view or view class. Each class of object in the user interface potentially responds to a different set of gestures. Thus, for example, notes respond to a different set of gestures than staves in the GSCORE music editor.

The gesture set is dynamically determined. From the first point of a gesture, the system dynamically determines the set of gestures possible. The first point determines the possible views at which the gesture is directed. For each of these views, inheritance up the class hierarchy determines the set of gestures it handles. These sets are combined, and if need be, a classifier for the resulting union is dynamically created.

The gesture class and attributes map to an application operation, operands, and parameters. Gestures are powerful because they contain additional information beyond the class of the gesture. The attributes of a gesture, such as size, orientation, length, speed, first point, and enclosed area, can all be mapped to parameters (including operands) of application routines. In the two-phase interaction, after the gesture is recognized there is an opportunity to map subsequent input to application parameters in the presence of application feedback.

Gesture handlers may be manipulated at runtime. In order to encourage exploration of gesture-based systems, all aspects of the gestural interface can be specified while the application is running. A new gesture handler may be created at runtime and associated with one or more views or view classes. Gesture classes may be added, deleted, or copied from other handlers.

Examples of each gesture class can be entered and modified at runtime. Finally, the semantics of the gesture class can be entered and modified at runtime. Three semantic expressions are specifiable: one evaluated when the gesture is first recognized, one evaluated on each subsequent mouse point, and one evaluated when the interaction completes.

10.1.4 Input in Object-Oriented User Interface Toolkits

A number of new ideas in the area of input in object-oriented user interface toolkits arose in the course of this work.

Passive and active event handlers A single passive event handler may be associated with multiple views. When input occurs on one such view, the handler usually activates a copy of itself. Thus, the active/passive dichotomy eliminates the need to have a controller object instantiated for each view that expects input, a major expense in many MVC systems.

Event handlers may be associated with view classes Instead of having to associate a handler with every instance of a view, the handler may be associated with one or more view classes. A view may have multiple handlers associated with it, and handlers are queried in a specific order to determine which handler will handle particular input.

Unified mouse input and virtual tools All input devices are tools, but when desired a single input device may at times be different tools, one way to implement modes in the interface. Tools may also be software objects, and some views are indeed such virtual tools. Tools often have an action, which allows them to operate on any views that respond to that action. The test of whether a given view responds to a given tool is made by an event handler associated with every view; this allows semantic feedback to occur automatically without any explicit action on the part of the view or the tool.

Automatic semantic feedback As just mentioned, the feedback as to whether a given tool operates upon a view over which it is has been dragged happens automatically. For example, objects that respond to the delete message will automatically highlight when a delete tool is dragged over them. If desired, an object can do more elaborate processing to determine if it truly responds to a given tool, e.g. an object may check that the user has permission to delete it before indicating it responds to the delete tool.

Runtime creation and manipulation of event handlers Event handlers may be created and associated with views or view classes at runtime. For example, a drag handler may be associated with an object, allowing that object to be dragged (i.e. have its position changed). In addition, such handlers may be modified at runtime, for example, to change the predicate that activates the handler.

10.2 Future Directions

In this section, directions for future work are discussed. These directions include remedies for deficiencies of the current work as well as extensions.

The single-stroke training and recognition algorithm is the most robust and well-tested part of the current work, and even in its current form it is probably suitable for commercial applications. However, a number of simple modifications should improve performance. Sections 9.1.1 and 9.1.5 contain suggestions for additional features as well as modifications to existing features; these should be implemented. Tracking the mouse in the presence of paging has proved to be a problem, and a significant improvement in recognition rate would be achieved if real-time response to mouse events could be guaranteed.

It should be simple to extend the algorithm to three dimensional gestures. All that would be required would be to add several more features to capture motion in the extra dimension. The training algorithm and linear classifier would be untouched by this extension.

Alternatives for rejection should be explored further. The estimated probability of ambiguity is useful, though using it will always result in rejections of about as many gestures that would have been correctly classified as not. The estimated Mahalanobis distance based on the common covariance matrix is really only useful for rejecting deliberately garbled gestures. The Mahalanobis distance based on the per-class covariance matrix fares somewhat better, but requires significantly more training examples to work well.

Given the obvious false assumption of equal per-class covariance matrices, it seems that the statistical classifier should not perform well on gesture sets, some classes of which vary in size and orientation, others of which do not. In practice, when the gesture classes are unambiguous, the classifiers have tended to perform admirably. Presumably this would not be the case for all such gesture sets. One area for exploration is a method for calculating the common covariance matrix differently, in particular, by not weighing the per class contributions by the number of examples of that class.

Another challenge would be to handle such gesture sets without giving up linear classification with a closed form training formula. There seems to be only one candidate, which relies on the multiclass minimum squared error and the pseudoinverse of the matrix of examples [30]. It should be explored as a potential alternative to classifiers that rely on estimates of a common covariance matrix.

It would be interesting to explore the possibility of allowing the user to indicate declaratively that a given gesture classes will vary in size and/or orientation. This might be handled simply by generating additional training examples by varying the user-supplied examples accordingly. Alternatively, it may be possible to augment the training algorithm so that the evaluation functions for certain classes are constrained to ignore certain features.

Relaxing the requirement that a closed form exist for the per-class feature weights allows iterative training methods to be considered. They have been ignored in this dissertation since they are expensive in training time and tend to require many training examples. However, as processor speed increases, iterative methods become more practical for use in a tool for experimenting with gesture-based interfaces.

Similarly, relaxing the requirement that the classifier be a linear discriminator opens the door for many other possibilities. Quadratic discrimination, and various non-parametric discrimination algorithms are but a few. These too are expensive and require many training examples.

Perhaps recognition technologies that require expensive training may be used in a production system while the cheaper technology developed here used for prototyping. This is analogous to using a fast compiler for development and an optimizing compiler for production. At the time of this writing it seems likely that neural networks will soon be in common use, and gesture recognition is but one application.

Additional attention should be given to the problem of detecting ambiguous sets of gesture classes and useless features. The triangular matrix of Mahalanobis distance between each pair of gesture classes is a useful starting point for determining similar gesture classes. Multivariate analysis of variance techniques [74] can determine which features contribute to the classification and which features are irrelevant. These techniques can be used to support the design of new features.

Eager recognition needs to be explored further. The classifiers generated by the algorithm of Chapter 4 are less eager than they could possibly be, due to the conservative choices being made. Hand labeling of ambiguous and unambiguous subgestures should be explored more fully; it is not difficult to imagine an interface that makes such labeling relatively painless, and it is likely to give better results than the current automatic labeling. Another possible improvement comes from the observation that, during eager recognition, the full classifier is being used to classify subgestures, upon which it was not trained. It might be worth trying to retrain the full classifier on the complete subgestures. Even better, perhaps a new classifier, trained on the *newly* complete subgestures (*i.e.* those made complete by their last point), should be substituted for the full classifier. Also, eager recognition needs to be extended to multi-path gestures.

Algorithms for automatically determining the start of a gesture would also be useful, especially for devices without any discrete signaling capability (most notably the DataGlove). In the current work, gestures are considered atomic, essentially having no discernible structure. It is easy to imagine separate gestures such as select, copy, move, and delete that are concatenated to make single interactions: select and move, select and delete. This raises the segmentation question: when does one gesture end and the next begin? Specifying allowable combinations of gestures opens up the possibility of gesture grammars, an interesting area for future study.

This dissertation has concentrated on single-path gestures that are restricted to be single strokes, for reasons explained previously. The utility of multiple-stroke gestures needs to be examined more thoroughly. In a multiple-stroke gesture, does the relaxation between strokes ruin the correspondence between mental and physical tension that makes for good interaction? Does the need for segmentation make the system less responsive than it otherwise might be? Can a manipulation phase and eager recognition be incorporated into a system based on multiple-stroke gestures? These questions require further research.

Due to the interest in multiple stroke recognition, the question arises as to whether the single-stroke algorithm can be extended to handle multiple stroke gestures. First, the segmentation problem (grouping strokes into gestures) needs to be addressed. One way this might be done is to add a large timeout to determine the end of a gesture. The distance of a stroke from the previous stroke might also be used. A sequence of strokes determined to be a single gesture might then be treated as a single stroke, with the exception of an additional feature which records the number of strokes in the gesture. The single-stroke recognition algorithm may then be applied.

Multi-path recognition is really still in its infancy. While the recognition algorithms of Chapter 5 seem to work well, there is not much to compare them against. Many others methods for multi-path recognition need to be explored. That said, the author is somewhat wary that multiple finger input devices are so seductive that gesture research will concentrate on such devices to the exclusion of single-path devices. This would be unfortunate, as it seems likely that single-path devices will be much more prevalent for the foreseeable future, and thus more users will potentially benefit from the availability of single-path gesturing. Also, a thorough understanding of the issues involved in single-path gesturing will likely be of use in solving the more difficult problems encountered in the multi-path case.

The advent of pen-based computers leads to the question of how the single-stroke recognition described here may be combined with handwriting recognition. One approach is to pass input to the gesture recognizer after it has been rejected by the handwriting recognizer. The context in which the stroke has been made (e.g. drawing window or text window) can also be used to determine whether to invoke handwriting recognition or stroke recognition first.

The start of a single-stroke gesture is used to determine the set of possible gestures by looking at possible objects at which the gesture is directed. It may be desirable to explore the possibility that the gesture is directed at an object other than one indicated by the first point, e.g. an object may be indicated by a hot point of the gesture (e.g. the intersection point of the delete gesture). A similar ambiguity occurs when the input is a multiple-finger gesture; which of the fingers should be used to determine the object(s) at which the gesture is directed? In this case, a union of the gestures recognized by objects indicated by each finger could be used, but the possibility of conflict remains.

One problem with gesture-based systems is that there is usually no indication of the possible gestures accepted by the system.[1] This is a difficulty that will potentially prevent novices from using the system. One approach would be to use animation[6] to indicate the possible gestures and their effects, although how the user asks to see the animation remains an open question.

Also daunting to beginners is the timeout interval, where “stillness” is used to indicate that collection is over and manipulation is to begin. Typically, a beginner presses a mouse button and then thinks about what to do next; by that time the system has already classified the gesture as a dot. The timeout cannot be totally disabled, since it is the only way to enter the manipulation phase for some gestures. Perhaps some scheme where the timeouts are long (0.75 seconds) for novices and decrease with use is desirable. Another possibility is eliminating the timeout totally at the beginning of the gestures, thus disallowing dot gestures.

The current work suffers from a lack of formal user evaluation. Additional studies are needed to determine classifier performance as a function of training examples, and whether one user can use a classifier trained by another. In general, the costs and the benefits of fixed verses trainable recognition strategies need to be studied. The usability of eager recognizers is also of interest.

Recognizers that gradually adapt to users need to be studied as well. Such a recognizer requires the user to somehow indicate when a gesture is misclassified by the system. Lerner [78] demonstrated a potentially applicable scheme in which the system monitored subsequent actions to see if the user was satisfied with the result of an applied heuristic. There are dangers inherent in doubly-adaptive systems–if the system adapts to the user and the user to the system, both are aiming at moving targets, and thrashing is possible. The current approach requires the user explicitly to replace the

---

[1] Kurtenbach et. al. [75] say that gesture-based interfaces are “non-revealing,” and present an interesting solution that unifies gesturing and pie-menu selection.

existing training examples with his own–a workable, if not glamorous, solution.

The low-level recognition work in this thesis is quite usable in its current state, and may be directly incorporated into systems as warranted. GRANDMA, however, is not useful as a base for future development. It is purely a research system, built as a platform for experimenting with input in user interface toolkits. Its output facilities are totally inadequate for real applications. GRANDMA was built solely by and for the author, who has no plans to maintain it. Nonetheless, GRANDMA embodies some important concepts of how gestures are to be integrated into object-oriented user interface tools.

The obvious next step is to integrate gestures into some existing user interface construction tools. Issues of technical suitability are important, but not paramount, in deciding which system to work on. Any chosen system must be well supported and maintained, so that there is a reasonable assurance that the system will survive. Furthermore, any chosen system must be widely distributed, in order to make the technology of gesture recognition available to as many experimenters as possible.

A number of existing systems are candidates for the incorporation of gestures. The NeXT Application Kit is technically the ideal platform–it is even programmed in Objective C. The appropriate hooks seem to be there to capture input at the right level in order to associate gestures with view classes. It is probably not worth the effort to implement an entire interpreter for entering gesture semantics at runtime, as this is not something a user will typically manipulate. A graphical interface to control semantics, based on constraints, would be an interesting addition. In general, a simpler way for mapping gestural attributes to application parameters needs to be determined.

The Andrew Toolkit (ATK) is another system into which gestures may be incorporated. ATK uses its own object-oriented programming language on top of C, so runtime representation of the class hierarchy, if not already present, should be straightforward to add. ATK has implemented dynamic loading of objects into running programs–this should make it possible to compile gesture semantics and load them into a running program without restarting the program. Unfortunately, due to their overhead, views tend to be large objects in ATK (e.g. individual notes in a score editor would not be separate views in ATK) making it difficult to associate different gestures with the smaller objects of interest in the interface. Scott Hassan, in a different approach, has added the author's gesture recognizer to the ATK text object, creating an interface that allows text editing via proofreader's marks.

Integrating gestures into Garnet is another possibility. What would be required is a gesture interactor, analogous to the gesture event handler in GRANDMA. Garnet interactors routinely specify their semantics via constraints, with an escape into Lisp available for unusual cases. Specifying gesture semantics should therefore be no problem in Garnet. James Landay has begun work integrating the author's recognizer into Garnet.

Gestures could also be added to MacApp. Besides being widely used, MacApp has the advantage that it runs on a Macintosh, which historically has run only one process at a time and has no virtual memory (this has changed with a recent system software release). While these points sound like disadvantages, the real-time operation needed to track the mouse reliably should be easy to achieve because of them. Because MacApp is implemented in Object Pascal, minimal meta-information about objects is available at runtime. In particular, message selectors are not first class objects in Object Pascal, it is not possible to ask if a given object responds to a message at runtime, and there is no runtime representation of the class hierarchy. Many things that happen automatically because GRANDMA is written in Objective C will need to be explicitly coded in MacApp.

It would be desirable to have additional attributes of the gesture available for use in gesture semantics. Notably missing from the current set are locations where the path intersects itself and locations of sharp corners of the stroke. Both kinds of attributes can be used for pointing with a gesture, and allow for multiple points to be indicated with one single-path gesture. Also, having the numerical attributes also available in a scaled form (*e.g.* between zero and one) would simplify their use as parameters to application functions.

10.3 Final Remarks

The utility of gesture-based interfaces derives from the ability to communicate an entire primitive application transaction with a single gesture. For this to be possible, the gesture needs to be classified to determine the operation to be performed, and attributes of the gesture must be mapped to the parameters of the operation. Some parameters may be culled at the time the gesture is recognized, while others are best manipulated in the presence of feedback from the application. This is the justification for the two-phase approach, where gesture recognition is followed by a manipulation phase, which allows for the continuous adjustment of parameters in the presence of application feedback.

From the outset, the goal of this work was to provide tools to allow the easy creation of gesture-based applications. This research has led to prototypes of such tools, and has thus laid much of the groundwork for building such tools in the future. However, the goal will not have been achieved until gestures are integrated into existing user interface construction tools that are both well maintained and highly available. This involves more development and marketing than it does research, but it is vitally important to the future of gesture-based systems.

Appendix A

Code for Single-Stroke Gesture Recognition and Training

This appendix contains the actual C code used to recognize single-stroke gestures. The feature vector calculation, classifier training algorithm, and the linear classifier are all presented. The code may be obtained free of charge via anonymous ftp to emsworth.andrew.cmu.edu (subdirectory gestures) and is also available as part of the Andrew contribution to the X11R5 distribution.

A.1 Feature Calculation

The lowest level of the code deals with computing a feature vector from a sequence of mouse points that make up a gesture. Type `FV` is a pointer to a structure that holds a feature vector as well as intermediate results used in the calculation of the features. The function `FvAlloc` allocates an `FV`, which is initialized before processing the points of a gesture via `FvInit.` `FvAddPoint` is called for each input point of the gesture, and `FvCalc` returns the feature vector for the gesture once all the points have been entered.

The following is a sample code fragment demonstrating the use of these functions:

```
#include "matrix.h"
#include "fv.h"

Vector
InputAGesture()
{
     static FV fv;
     int x, y; long t; Vector v;

     /* FvAlloc() is typically called only once per program invocation. */
     if(fv == NULL) fv = FvAlloc();
```

```
    /* A prototypical loop to compute a feature vector from a gesture
       being read from a window manager: */
    FvInit(fv);
    while(GetNextPoint(&x, &y, &t) != END_OF_GESTURE)
        FvAddPoint(fv, x, y, t);
    v = FvCalc(fv);
    return v;

}
```

The returned vector `v` might now be passed to `sClassify` to classify the gesture.

The remainder of this section shows the header file, fv.h, which defines the `FV` type and the feature vector interface. This interface is implemented in fv.c, shown next.

```
/**********************************************************************
fv.h - Create a feature vector, useful for gesture classification,
   from a sequence of points (e.g. mouse points).
**********************************************************************/

/* ----------- compile time settable parameters ----------- */
/* some of these can also be set at runtime, see fv.c */

#undef  USE_TIME
                /* Define USE_TIME to enable the duration and maximum */
                /* velocity features.  When not defined, 0 may be passed */
                /* as the time to FvAddPoint. */

#define DIST_SQ_THRESHOLD (3*3)
                /* points within sqrt(DIST_SQ_THRESHOLD) */
                /* will be ignored to eliminate mouse jitter */

#define SE_TH_ROLLOFF   (4*4)
                /* The SE_THETA features (cos and sin of */
                /* angle between first and last point) will */
                /* be scaled down if the distance between the */
                /* points is less than sqrt(SE_TH_ROLLOFF) */

/* ----------- Interface ----------- */

typedef struct fv *FV;
                /* During gesture collection, an FV holds */
                /* all intermediate results used in the */
                /* calculation of a single feature vector */
```

```
FV      FvAlloc();      /* */
void    FvFree();       /* Fv fv */
void    FvInit();       /* FV fv */
void    FvAddPoint();   /* FV fv; int x, y; long time; */
Vector  FvCalc();       /* FV fv;  */

/*------ internal data structure ------*/
#define MAXFEATURES 32
        /* maximum number of features, occasionally useful as an array bound */

/* indices into the feature Vector returned by FvCalc */

#define PF_INIT_COS    0 /* initial angle (cos) */
#define PF_INIT_SIN    1 /* initial angle (sin) */
#define PF_BB_LEN      2 /* length of bounding box diagonal */
#define PF_BB_TH       3 /* angle of bounding box diagonal */
#define PF_SE_LEN      4 /* length between start and end points */
#define PF_SE_COS      5 /* cos of angle between start and end points */
#define PF_SE_SIN      6 /* sin of angle between start and end points */
#define PF_LEN         7 /* arc length of path */
#define PF_TH          8 /* total angle traversed */
#define PF_ATH         9 /* sum of abs vals of angles traversed */
#define PF_SQTH       10 /* sum of squares of angles traversed */

#ifndef USE_TIME
#   define NFEATURES       11
#else
#   define PF_DUR          11      /* duration of path */
#   define PF_MAXV         12      /* maximum speed */
#   define NFEATURES       13
#endif

/* structure which holds intermediate results during feature vector calculation */

struct fv {

    /* the following are used in calculating the features */
    double      startx, starty; /* starting point */
    long        starttime;      /* starting time */

    /* these are set after a few points and then left alone */
```

```
    double      initial_sin, initial_cos;  /* initial angle to x axis */

    /* these are updated incrementally upon every point */
    int     npoints;            /* number of points in path */

    double      dx2, dy2;       /* differences: endx-prevx, endy-prevy */
    double      magsq2;         /* dx2*dx2 + dy2*dy2 */

    double      endx, endy;     /* last point added */
    long        endtime;

    double      minx, maxx, miny, maxy;   /* bounding box */

    double      path_r, path_th;  /* total length and rotation (in radians) */
    double      abs_th;           /* sum of absolute values of path angles */
    double      sharpness;        /* sum of squares of path angles */
    double      maxv;             /* maximum velocity */


    Vector      y;                /* Actual feature vector */
};


/********************************************************************
fv.c - Creates a feature vector, useful for gesture classification,
  from a sequence of points (e.g. mouse points).
********************************************************************/

#include <stdio.h>
#include <math.h>
#include "matrix.h"  /* contains Vector and associated functions */
#include "fv.h"

/* runtime settable parameters */
double dist_sq_threshold = DIST_SQ_THRESHOLD;
double se_th_rolloff = SE_TH_ROLLOFF;

#define EPS (1.0e-4)

/* allocate an FV struct including feature vector */

FV
```

```
FvAlloc()
{
    register FV fv = (FV) mallocOrDie(sizeof(struct fv));
    fv->y = NewVector(NFEATURES);
    FvInit(fv);
    return fv;
}

/* free memory associated with an FV struct */
void
FvFree(fv)
FV fv;
{
    FreeVector(fv->y);
    free((char *) fv);
}

/* initialize an FV struct to prepare for incoming gesture points */
void
FvInit(fv)
register FV fv;
{
    register int i;

    fv->npoints = 0;
    fv->initial_sin = fv->initial_cos = 0.0;
    fv->maxv = 0;
    fv->path_r = 0;
    fv->path_th = 0;
    fv->abs_th = 0;
    fv->sharpness = 0;
    fv->maxv = 0;
    for(i = 0; i < NFEATURES; i++)
        fv->y[i] = 0.0;
}

/* update an FV struct to reflect a new input point */
void
FvAddPoint(fv, x, y, t)
register FV fv; int x, y; long t;
{
    double dx1, dy1, magsq1;
```

```
    double th, absth, d;
#ifdef PF_MAXV
    long lasttime;
#endif

    ++fv−>npoints;
    if(fv−>npoints == 1) {        /∗ first point, initialize some vars ∗/
        fv−>starttime = fv−>endtime = t;
        fv−>startx = fv−>endx = fv−>minx = fv−>maxx = x;
        fv−>starty = fv−>endy = fv−>miny = fv−>maxy = y;
        fv−>endx = x; fv−>endy = y;
        return;
    }

    dx1 = x − fv−>endx; dy1 = y − fv−>endy;
    magsq1 = dx1 ∗ dx1 + dy1 ∗ dy1;

    if(magsq1 <= dist_sq_threshold) {
        fv−>npoints−−;
        return;       /∗ ignore a point close to the last point ∗/
    }

    if(x < fv−>minx) fv−>minx = x;
    if(x > fv−>maxx) fv−>maxx = x;
    if(y < fv−>miny) fv−>miny = y;
    if(y > fv−>maxy) fv−>maxy = y;

#ifdef PF_MAXV
    lasttime = fv−>endtime;
#endif
    fv−>endtime = t;

    d = sqrt(magsq1);
    fv−>path_r += d;        /∗ update path length feature ∗/

    /∗ calculate initial theta when the third point is seen ∗/
    if(fv−>npoints == 3) {
        double magsq, dx, dy, recip;
        dx = x − fv−>startx; dy = y − fv−>starty;
        magsq = dx ∗ dx + dy ∗ dy;
        if(magsq > dist_sq_threshold)  {
            /∗ find angle w.r.t. positive x axis e.g. (1,0) ∗/
```

```
            recip = 1 / sqrt(magsq);
            fv->initial_cos = dx * recip;
            fv->initial_sin = dy * recip;
        }
    }

    if(fv->npoints >= 3) {   /* update angle-based features */
        th = absth = atan2(dx1 * fv->dy2 - fv->dx2 * dy1,
                     dx1 * fv->dx2 + dy1 * fv->dy2);
        if(absth < 0) absth = -absth;
        fv->path_th += th;
        fv->abs_th += absth;
        fv->sharpness += th*th;

#ifdef PF_MAXV   /* compute max velocity */
        if(fv->endtime > lasttime &&
           (v = d / (fv->endtime - lasttime)) > fv->maxv)
            fv->maxv = v;
#endif
    }

    /* prepare for next iteration */
    fv->endx = x; fv->endy = y;
    fv->dx2 = dx1; fv->dy2 = dy1;
    fv->magsq2 = magsq1;

    return;
}

/* calculate and return a feature vector */
Vector
FvCalc(fv)
register FV fv;
{
    double bblen, selen, factor;

    if(fv->npoints <= 1)
        return fv->y;       /* a feature vector of all zeros */

    fv->y[PF_INIT_COS] = fv->initial_cos;
    fv->y[PF_INIT_SIN] = fv->initial_sin;
```

```
	/* compute the length of the bounding box diagonal */
	bblen = hypot(fv->maxx - fv->minx, fv->maxy - fv->miny);

	fv->y[PF_BB_LEN] = bblen;

	/* the bounding box angle defaults to 0 for small gestures */
	if(bblen * bblen > dist_sq_threshold)
	    fv->y[PF_BB_TH] = atan2(fv->maxy - fv->miny,
	                            fv->maxx - fv->minx);

	/* compute the length and angle between the first and last points */
	selen = hypot(fv->endx - fv->startx,
	              fv->endy - fv->starty);
	fv->y[PF_SE_LEN] = selen;

	/* when the first and last points are very close, the angle features
	   are muted so that they satisfy the stability criterion */
	factor = selen * selen / se_th_rolloff;
	if(factor > 1.0) factor = 1.0;
	factor = selen > EPS ? factor/selen :  0.0;
	fv->y[PF_SE_COS] = (fv->endx - fv->startx) * factor;
	fv->y[PF_SE_SIN] = (fv->endy - fv->starty) * factor;

	/* the remaining features have already been computed */
	fv->y[PF_LEN] =  fv->path_r;
	fv->y[PF_TH] = fv->path_th;
	fv->y[PF_ATH] = fv->abs_th;
	fv->y[PF_SQTH] = fv->sharpness;

#ifdef PF_DUR
	fv->y[PF_DUR] = (fv->endtime - fv->starttime)*.01;
#endif

#ifdef PF_MAXV
	fv->y[PF_MAXV] = fv->maxv * 10000;
#endif

	return fv->y;
}
```

A.2 Deriving and Using the Linear Classifier

Type `sClassifier` points at an object that represents a classifier able to discriminate between a set of gesture classes. Each gesture class is represented by an `sClassDope` type. The functions `sRead` and `sWrite` read and write a classifier to a file. The function `sNewClassifer` creates a new (empty) classifier. A training example is added using `sAddExample`. There is no function to explicitly add a new class to a classifier. When an example of a new class is added, the new class is created automatically. To train the classifier based on the added examples, call `sDoneAdding`. Once trained, `sClassify` and `sClassifyAD` are used to classify a feature vector as one of the classes; `sClassifyAD` optionally computes the rejection information.

Here is an example fragment for creating a new classifier, entering new training examples, and writing the resulting classifier out to a file. Some of these functions are timed (and further described) in section 9.1.7.

```
#include <stdio.h>
#include <math.h>
#include "bitvector.h"
#include "matrix.h"
#include "sc.h"

#define NEXAMPLES   15

sClassifier
MakeAClassifier()
{
    sClassifier sc = sNewClassifier();
    Vector InputAGesture();
    char name[100];
    int i;

    for(;;) {
        printf("Enter class name, newline to exit: ");
        if(gets(name) == NULL || name[0] == '\0')
            break;
        for(i = 1; i <= NEXAMPLES; i++) {
            printf("Enter %s example %d\n", name, i);
            sAddExample(sc, name, InputAGesture());
        }
    }
    sDoneAdding(sc);
    sWrite(fopen("classifier.out", "w"), sc);
    return sc;
```

```
}
```

Once a classifier has been created it can be used to classifier gestures as follows:

```
TestAClassifier(sc)
sClassifier sc;
{
    Vector v;
    sClassDope scd;
    double punambig, distance;

    for(;;) {
        printf("Enter a gesture\n");
        v = InputAGesture();
        scd = sClassifyAD(sc, v, &punambig, &distance);
        printf("Gesture classified as %s ", scd−>name);
        printf("Probability of unambiguous classification: %g\n",
                punambig);
        printf("Distance from class mean: %g\n", distance);
    }
}
```

What follows is the header file and code to implement the statistical classifier.

```
/∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗
sc.h − create single path classifiers from feature vectors of examples,
  as well as classifying example feature vectors.
∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗/

#define MAXSCLASSES 100 /∗ maximum number of classes ∗/

typedef struct sclassifier ∗sClassifier;   /∗ classifier ∗/
typedef int sClassIndex;                   /∗ per−class index ∗/
typedef struct sclassdope  ∗sClassDope;    /∗ per−class information ∗/

struct sclassdope { /∗ per gesture class information within a classifier ∗/
    char        ∗name;        /∗ name of a class ∗/
    sClassIndex number;       /∗ unique index (small integer) of a class ∗/
    int         nexamples;    /∗ number of training examples ∗/
    Vector      average;      /∗ average of training examples ∗/
    Matrix      sumcov;       /∗ covariance matrix of examples ∗/
};
```

```
struct sclassifier {  /* a classifier */
    int       nfeatures;       /* number of features in feature vector */
    int       nclasses;        /* number of classes known by this classifier */
    sClassDope  *classdope; /* array of pointers to per class data */

    Vector      cnst;          /* constant term of discrimination function */
    Vector      *w;            /* array of coefficient weights */
    Matrix      invavgcov;     /* inverse covariance matrix */
};

sClassifier sNewClassifier();       /* */
sClassifier sRead();                /* FILE *f */
void        sWrite();               /* FILE *f; sClassifier sc; */
void        sFreeClassifier();      /* sc */
void        sAddExample();          /* sc, char *classname; Vector y */
void        sDoneAdding();          /* sc */
sClassDope  sClassify();            /* sc, y */
sClassDope  sClassifyAD();          /* sc, y, double *ap; double *dp */
sClassDope  sClassNameLookup();     /* sc, classname */
double      MahalanobisDistance();  /* Vector v, u; Matrix sigma */

/*********************************************************************
sc.c - creates classifiers from feature vectors of examples, as well as
  classifying example feature vectors.
**********************************************************************/


#include <stdio.h>
#include <math.h>
#include "bitvector.h"
#include "matrix.h"
#include "sc.h"

#define EPS (1.0e-6)      /* for singular matrix check */

/* allocate memory associated with a new classifier */

sClassifier
sNewClassifier()
{
    register sClassifier sc =
```

```
            (sClassifier) mallocOrDie(sizeof(struct sclassifier));
        sc−>nfeatures = −1;
        sc−>nclasses = 0;
        sc−>classdope = (sClassDope ∗)
            mallocOrDie(MAXSCLASSES ∗ sizeof(sClassDope));
        sc−>w = NULL;
        return sc;
}

/∗ free memory associated with a new classifier ∗/
void
sFreeClassifier(sc)
register sClassifier sc;
{
        register int i;
        register sClassDope scd;

        for(i = 0; i < sc−>nclasses; i++) {
            scd = sc−>classdope[i];
            if(scd−>name) free(scd−>name);
            free(scd);
            if(sc−>w && sc−>w[i]) FreeVector(sc−>w[i]);
            if(scd−>sumcov) FreeMatrix(scd−>sumcov);
            if(scd−>average) FreeVector(scd−>average);
        }
        free(sc−>classdope);
        if(sc−>w) free(sc−>w);
        if(sc−>cnst) FreeVector(sc−>cnst);
        if(sc−>invavgcov) FreeMatrix(sc−>invavgcov);
        free(sc);
}

/∗ given a string name of a class, return its per−class information ∗/
sClassDope
sClassNameLookup(sc, classname)
register sClassifier sc;
register char ∗classname;
{
        register int i;
        register sClassDope scd;
        static sClassifier lastsc;
        static sClassDope lastscd;
```

```
    /* quick check for last class name */
    if(lastsc == sc && STREQ(lastscd->name, classname))
        return lastscd;

    /* linear search through all classes for name */
    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        if(STREQ(scd->name, classname))
            return lastsc = sc, lastscd = scd;
    }
    return NULL;
}

/* add a new gesture class to a classifier */
static sClassDope
sAddClass(sc, classname)
register sClassifier sc;
char *classname;
{
    register sClassDope scd;

    sc->classdope[sc->nclasses] = scd = (sClassDope)
               mallocOrDie(sizeof(struct sclassdope));
    scd->name = scopy(classname);
    scd->number = sc->nclasses;
    scd->nexamples = 0;
    scd->sumcov = NULL;
    ++sc->nclasses;
    return scd;
}

/* add a new training example to a classifier */
void
sAddExample(sc, classname, y)
register sClassifier sc;
char *classname;
Vector y;
{
    register sClassDope scd;
    register int i, j;
    double nfv[50];
```

```
    double nm1on, recipn;

    scd = sClassNameLookup(sc, classname);
    if(scd == NULL)
        scd = sAddClass(sc, classname);

    if(sc->nfeatures == -1)
        sc->nfeatures = NROWS(y);

    if(scd->nexamples == 0) {
        scd->average = NewVector(sc->nfeatures);
        ZeroVector(scd->average);
        scd->sumcov = NewMatrix(sc->nfeatures,sc->nfeatures);
        ZeroMatrix(scd->sumcov);

    }

    if(sc->nfeatures != NROWS(y)) {
        PrintVector(y, "sAddExample: funny vector nrows!=%d",
            sc->nfeatures);
        return;
    }

    scd->nexamples++;
    nm1on = ((double) scd->nexamples-1)/scd->nexamples;
    recipn = 1.0/scd->nexamples;

    /* incrementally update covariance matrix */
        for(i = 0; i < sc->nfeatures; i++)
        nfv[i] = y[i] - scd->average[i];

    /* only upper triangular part computed */
        for(i = 0; i < sc->nfeatures; i++)
       for(j = i; j < sc->nfeatures; j++)
        scd->sumcov[i][j] += nm1on * nfv[i] * nfv[j];

    /* incrementally update mean vector */
    for(i = 0; i < sc->nfeatures; i++)
        scd->average[i] =
             nm1on * scd->average[i] + recipn * y[i];

}
```

```
/* run the training algorithm on the classifier */
void
sDoneAdding(sc)
register sClassifier sc;
{
    register int i, j;
    int c;
    int ne, denom;
    double oneoverdenom;
    register Matrix s;
    register Matrix avgcov;
    double det;
    register sClassDope scd;

    if(sc->nclasses == 0)
        error("sDoneAdding: No classes\n");

    /* Given covariance matrices for each class (* number of examples - 1)
       compute the average (common) covariance matrix */

    avgcov = NewMatrix(sc->nfeatures, sc->nfeatures);
    ZeroMatrix(avgcov);
    ne = 0;
    for(c = 0; c < sc->nclasses; c++) {
        scd = sc->classdope[c];
        ne += scd->nexamples;
        s = scd->sumcov;
        for(i = 0; i < sc->nfeatures; i++)
            for(j = i; j < sc->nfeatures; j++)
                avgcov[i][j] += s[i][j];
    }

    denom = ne - sc->nclasses;
    if(denom <= 0) {
        printf("no examples, denom=%d\n", denom);
        return;
    }

    oneoverdenom = 1.0 / denom;
    for(i = 0; i < sc->nfeatures; i++)
        for(j = i; j < sc->nfeatures; j++)
```

```
            avgcov[j][i] = avgcov[i][j] *= oneoverdenom;

    /* invert the avg covariance matrix */

    sc->invavgcov = NewMatrix(sc->nfeatures, sc->nfeatures);
    det = InvertMatrix(avgcov, sc->invavgcov);
    if(fabs(det) <= EPS)
        FixClassifier(sc, avgcov);

    /* now compute discrimination functions */
    sc->w = (Vector *)
              mallocOrDie(sc->nclasses * sizeof(Vector));
    sc->cnst = NewVector(sc->nclasses);
    for(c = 0; c < sc->nclasses; c++) {
        scd = sc->classdope[c];
        sc->w[c] = NewVector(sc->nfeatures);
        VectorTimesMatrix(scd->average, sc->invavgcov,
                        /* product = */ sc->w[c]);
        sc->cnst[c] = -0.5 *
                 InnerProduct(sc->w[c], scd->average);
        /* could add log(priorprob class c) to cnst[c] */
    }

    FreeMatrix(avgcov);
    return;
}

/* classify a feature vector */
sClassDope
sClassify(sc, fv) {
    return sClassifyAD(sc, fv, NULL, NULL);
}

/* classify a feature vector, possibly computing rejection metrics */
sClassDope
sClassifyAD(sc, fv, ap, dp)
sClassifier sc;
Vector fv;
double *ap;
double *dp;
{
    double disc[MAXSCLASSES];
```

```
    register int i, maxclass;
    double denom, exp();
    register sClassDope scd;
    double d;

    if(sc->w == NULL)
        error("sClassifyAD: %x no trained classifier", sc);

    for(i = 0; i < sc->nclasses; i++)
        disc[i] = InnerProduct(sc->w[i], fv) + sc->cnst[i];

    maxclass = 0;
    for(i = 1; i < sc->nclasses; i++)
        if(disc[i] > disc[maxclass])
            maxclass = i;

    scd = sc->classdope[maxclass];

    if(ap) {    /* calculate probability of non-ambiguity */
        for(denom = 0, i = 0; i < sc->nclasses; i++)
            /* quick check to avoid computing negligible term */
            if((d = disc[i] - disc[maxclass]) > -7.0)
                denom += exp(d);
        *ap = 1.0 / denom;
    }

    if(dp)  /* calculate distance to mean of chosen class */
        *dp = MahalanobisDistance(fv, scd->average,
                                  sc->invavgcov);

    return scd;
}

/* Compute the Mahalanobis distance between two vectors v and u */
double
MahalanobisDistance(v, u, sigma)
register Vector v, u;
register Matrix sigma;
{
    register i;
    static Vector space;
    double result;
```

```
    if(space == NULL || NROWS(space) != NROWS(v)) {
        if(space) FreeVector(space);
        space = NewVector(NROWS(v));
    }
    for(i = 0; i < NROWS(v); i++)
        space[i] = v[i] - u[i];
    result =  QuadraticForm(space, sigma);
    return result;
}

/* handle the case of a singular average covariance matrix by removing features */
FixClassifier(sc, avgcov)
register sClassifier sc;
Matrix avgcov;
{
    int i;
    double det;
    BitVector bv;
    Matrix m, r;

    /* just add the features one by one, discarding any that cause
       the matrix to be non-invertible */

    CLEAR_BIT_VECTOR(bv);
    for(i = 0; i < sc->nfeatures; i++) {
        BIT_SET(i, bv);
        m = SliceMatrix(avgcov, bv, bv);
        r = NewMatrix(NROWS(m), NCOLS(m));
        det = InvertMatrix(m, r);
        if(fabs(det) <= EPS)
            BIT_CLEAR(i, bv);
        FreeMatrix(m);
        FreeMatrix(r);
    }

    m = SliceMatrix(avgcov, bv, bv);
    r = NewMatrix(NROWS(m), NCOLS(m));
    det = InvertMatrix(m, r);
    if(fabs(det) <= EPS)
        error("Can't fix classifier!");
    DeSliceMatrix(r, 0.0, bv, bv, sc->invavgcov);
```

```
    FreeMatrix(m);
    FreeMatrix(r);

}

/* write a classifier to a file */
void
sWrite(outfile, sc)
FILE *outfile;
sClassifier sc;
{
    int i;
    register sClassDope scd;

    fprintf(outfile, "%d classes\n", sc->nclasses);
    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        fprintf(outfile, "%s\n", scd->name);
    }
    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        OutputVector(outfile, scd->average);
        OutputVector(outfile, sc->w[i]);

    }
    OutputVector(outfile, sc->cnst);
    OutputMatrix(outfile, sc->invavgcov);
}

/* read a classifier from a file */
sClassifier
sRead(infile)
FILE *infile;
{
    int i, n;
    register sClassifier sc;
    register sClassDope scd;
    char buf[100];


    printf("Reading classifier "), fflush(stdout);
```

```
    sc = sNewClassifier();
    fgets(buf, 100, infile);
    if(sscanf(buf, "%d", &n) != 1) error("sRead 1");
    printf("%d classes ", n), fflush(stdout);
    for(i = 0; i < n; i++) {
        fscanf(infile, "%s", buf);
        scd = sAddClass(sc, buf);
        scd->name = scopy(buf);
        printf("%s ", scd->name), fflush(stdout);
    }
    sc->w = allocate(sc->nclasses, Vector);
    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        scd->average = InputVector(infile);
        sc->w[i] = InputVector(infile);
    }
    sc->cnst = InputVector(infile);
    sc->invavgcov = InputMatrix(infile);
    printf("\n");
    return sc;
}

/* compute pairwise distances between classes, and print the closest ones,
   as a clue as to which gesture classes are confusable */

sDistances(sc, nclosest)
register sClassifier sc;
{
    register Matrix d = NewMatrix(sc->nclasses, sc->nclasses);
    register int i, j;
    double min, max = 0;
    int n, mi, mj;

    printf("-----------\n");
    printf("%d closest pairs of classes\n", nclosest);
    for(i = 0; i < NROWS(d); i++) {
        for(j = i+1; j < NCOLS(d); j++) {
            d[i][j] = MahalanobisDistance(
                          sc->classdope[i]->average,
                          sc->classdope[j]->average,
                          sc->invavgcov);
            if(d[i][j] > max) max = d[i][j];
```

```
            }
        }

        for(n = 1; n <= nclosest; n++) {
            min = max;
            mi = mj = -1;
            for(i = 0; i < NROWS(d); i++) {
                for(j = i+1; j < NCOLS(d); j++) {
                    if(d[i][j] < min)
                        min = d[mi=i][mj=j];
                }
            }
            if(mi == -1)
                break;

            printf("%2d) %10.10s to %10.10s d=%g nstd=%g\n",
                n,
                sc->classdope[mi]->name,
                sc->classdope[mj]->name,
                d[mi][mj],
                sqrt(d[mi][mj]));

            d[mi][mj] = max+1;
        }
        printf("-----------\n");
        FreeMatrix(d);
    }
```

A.3 Undefined functions

The above code uses some functions whose definitions are not included in this appendix. These fall into four classes: standard library functions (including the math library), utility functions, bitvector functions, and vector/matrix functions, The standard library calls will not be discussed.

The utility functions used are

`STREQ(s1, s2)` returns FALSE iff strings `s1` and `s2` are equal.

`scopy(s)` returns a copy of the string `s`.

`error(format, arg1...)` prints a message and causes the program to exit.

`mallocOrDie(nbytes)` calls malloc, dying with an error message if the memory cannot be obtained.

The bit vector operations are an efficient set of functions for accessing an array of bits.

`CLEAR_BIT_VECTOR(bv)` resets an entire bit vector `bv` to all zeros,

`BIT_SET(i, bv)` sets the `i`th bit of `bv` to one, and

`BIT_CLEAR(i, bv)` sets the `i`th bit of `bv` to zero.

The vector/matrix functions are declared in matrix.h. Objects of type `Vector` and `Matrix` may be accessed like one and two dimensional arrays, respectively, but also contain additional information as to the size and dimensionality of the object (accessible via macros `NROWS`, `NCOLS`, and `NDIM`. It should be obvious from the names and the use of most of the functions (`NewVector`, `NewMatrix`, `FreeVector`, `FreeMatrix`, `ZeroVector`, `ZeroMatrix`, `PrintVector`, `PrintMatrix`, `InvertMatrix`, `InputVector`, `InputMatrix`, `OutputVector`, `OutputMatrix`, `VectorTimesMatrix`, and `InnerProduct`) what they do. As for the remaining functions,

`double QuadraticForm(Vector V, Matrix M)` computes the quantity $V'MV$, where the prime denotes the transpose operation.

`Matrix SliceMatrix(Matrix m, BitVector rowmask, BitVector colmask)` creates a new matrix, consisting only of those rows and columns in `m` whose corresponding bits are set it `rowmask` and `colmask`, respectively.

`Matrix DeSliceMatrix(Matrix m, double fill, BitVector rowmask; BitVector colmask; Matrix result)` first sets every element in `result` to `fill`, and then, every element in `result` whose row number is on in `rowmask` and whose column number is on in `colmask`, is set from the corresponding element in the input matrix `m`, which is smaller than `r`. The result of `SliceMatrix(DeSliceMatrix(m, fill, rowmask, colmask, result), rowmask, colmask)` is a copy of `m`, given legal values for all parameters.

These auxiliary functions, as well as a C-based X11R5 version of GDP, are all available as part of the ftp distribution mentioned above.

Bibliography

[1] Apple. *Inside Macintosh*. Addison-Wesley, 1985.

[2] Apple. *Macintosh System Software User's Guide, Version 6.0*. Apple Computer, 1988.

[3] H. Arakawa, K. Okada, and J. Masuda. On-line recognition of handwritten characters: Alphanumerics, Hiragana, Katakana, Kanji. In *Proceedings of the 4th International Joint Conference on Pattern Recognition*, pages 810–812, 1978.

[4] R. Baecker. Towards a characterization of graphical interaction. In Guedj, R. A., et. al., editor, *Methodology of Interaction*, pages 127–147. North Holland, 1980.

[5] R. Baecker and W. A. S. Buxton. *Readings in Human-Computer Interaction - A Multidisciplinary Approach*. Morgan Kaugmann Readings Series. Morgan Kaufmann, Los Altos, California, 1987.

[6] Ronald Baecker, Ian Small, and Richard Mander. Bringing icons to life. In *CHI'91 Conference Proceedings*, pages 1–6. ACM, April 1991.

[7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), 1975.

[8] M. Berthod and J. P. Maroy. Learning in syntactic recognition of symbols drawn on a graphic tablet. *Computer Graphics Image Processing*, 9:166–182, 1979.

[9] J. Block and R. Dannenberg. Polyphonic accompaniment in real time. In *International Computer Music Conference*, Cambridge, Mass., 1985. Computer Music Association.

[10] R. Boie. Personnel communication. 1987.

[11] R. Boie, M. Mathews, and A. Schloss. The Radio Drum as a synthesizer controller. In *1989 International Computer Music Proceedings*, pages 42–45. Computer Music Association, November 1989.

[12] R. A. Bolt. *The Human Interface: where people and computers meet*. Lifetime Learning Publications, 1984.

[13] Radmilo M. Bozinovic. *Recognition of Off-line Cursive Handwriting: A Case of Multi-level Machine Perception*. PhD thesis, State University of New York at Buffalo, March 1985.

[14] W. A. S. Buxton. Chunking and phrasing and the design of human-computer dialogues. In *Information Processing 86*, North Holland, 1986. Elsevier Science Publishers B.V.

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

[16] W. A. S. Buxton. Smoke and mirrors. *Byte*, 15(7):205–210, July 1990.

[17] W. A. S. Buxton. A three-state model of graphical input. *Proceedings of Interact 90*, August 1990.

[18] W. A. S. Buxton, R. Hill, and P. Rowley. Issues and techniques in touch-sensitive tablet input. *Computer Graphics*, 19(3):215–224, 1985.

[19] W. A. S. Buxton and B. Myers. A study in two-handed input. In *Proceedings of CHI '86*, pages 321–326. ACM, 1986.

[20] W. A. S. Buxton, W. Reeves, R. Baecker, and L. Mezei. The user of hierarchy and instance in a data structure for computer music. In Curtis Roads and John Strawn, editors, *Foundations of Computer Music*, chapter 24, pages 443–466. MIT Press, Cambridge, Massachusetts, 1985.

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

[22] S. K. Card, Moran, T. P., and A. Newell. The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23(7):601–613, 1980.

[23] L. Cardelli and R. Pike. Squeak: A language for communicating with mice. *SIGGRAPH '85 Proceedings*, 19(3), April 1985.

[24] R. M. Carr. The point of the pen. *BYTE*, 16(2):211–221, February 1991.

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

[26] P. W. Cooper. Hyperplanes, hyperspheres, and hyperquadrics as decision boundaries. In J. T. Tou and R. H. Wilcox, editors, *Computer and Information Sciences*, pages 111–138. Spartan, Washington, D.C., 1964.

[27] Brad J. Cox. Message/object programming: An evolutionary change in programming technology. *IEEE Software*, 1(1):50–61, January 1984.

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

[29] R. B. Dannenberg. A structure for representing, displaying, and editing music. In *Proceedings of the 1986 International Computer Music Conference*, pages 153–160, San Francisco, 1986. Computer Music Association.

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

[31] The Economist. Digital quill. *The Economist*, 316(7672):88, September 15 1990.

[32] H. Eglowstein. Reach out and touch your data. *Byte*, 15(7):283–290, July 1990.

[33] W. English, D. Engelbart, and M. L. Berman. Display-selection techniques for text manipulation. *IEEE Transactions on Human Factors in Electronics*, HFE-8(1):21–31, 1967.

[34] S. S. Fels and Geoffrey E. Hinton. Building adaptive interfaces with neural networks: The glove-talk pilot study. Technical Report CRG-TR-90-1, University of Toronto, Toronto, Canada, 1990.

[35] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936.

[36] Flecchia and Nergeron. Specifying complex dialogs in ALGAE. *SIGCHI+GI 87 Proceedings*, April 1987.

[37] K. S. Fu. *Syntactic Recognition in Character Recognition*, volume 112 of *Mathematics in Science and Engineering*. Academic Press, 1974.

[38] K. S. Fu. Hybrid aproaches to pattern recognition. In K. S. Fu J. Kittler and L. F. Pau, editors, *Pattern Recognition Theory and Applications*, NATO Advanced Study Institute, pages 139–155. D. Reidel, 1981.

[39] K. S. Fu. *Syntactic Pattern Recognition and Applications*. Prentice Hall, 1981.

[40] K. S. Fu and T. S. Yu. *Statistical Pattern Classification using Contextual Information*. Pattern Recognition and Image Processing Series. Research Studies Press, New York, 1980.

[41] J. Gettys, R. Newman, and R. W. Schiefler. *Xlib - C Language Interface X11R2*. Massachusetts Institute of Technology, 1988.

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

[43] R. Glitman. Startup readies 4-pound stylus pc. *PC Week*, 7(34):17–18, August 27 1990.

[44] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley series in Computer Science. Addison-Wesley, 1983.

[45] D. Goodman. *The complete HyperCard handbook*. Bantam Books, 1988.

[46] G. H. Granlund. Fourier preprocessing for hand print character recognition. *IEEE Transactions on Computers*, 21:195–201, February 1972.

[47] I. Guyon, P. Albrecht, Y. Le Cun, J. Denker, and W. Hubbard. Design of a neural network character recognizer for a touch terminal. *Pattern Recognition*, 24(2):105–119, 1991.

[48] D. J. Hand. *Kernel Discriminant Analysis*. Pattern Recognition and Image Processing Research Studies Series. Research Studies Press (A Division of John Wiley and Sons, Ltd.), New York, 1982.

[49] A. G. Hauptmann. Speech and gestures for graphic image manipulation. In *CHI '89 Proceedings*, pages 241–245. ACM, May 1989.

[50] Frank Hayes. True notebook computing arrives. *Byte*, pages 94–95, December 1989.

[51] P. J. Hayes, P. A. Szekely, and R. A. Lerner. Design alternatives for user interface management systems based on experience with COUSIN. In *CHI '85 Proceedings*, pages 169–175. ACM, April 1985.

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

[53] C. A. Higgins and R. Whitrow. On-line cursive script recognition. In B. Shackel, editor, *Human-Computer Interaction - Interact '84, IFIP*, pages 139–143, North-Holland, 1985. Elsevier Science Publishers B.V.

[54] R. Hill. Supporting concurrency, communication, and synchronization in human-computer interaction. *ACM Transactions on Graphics*, 5(3):179–210, July 1986.

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

[56] Bruce L. Horn. An introduction to object oriented programming, inheritance and method combination. Technical Report CMU-CS-87-127, Carnegie Mellon University Computer Science Department, 1988.

[57] A. B. S. Hussain, G. T. Toussaint, and R. W. Donaldson. Results obtained using a simple character recognition procedure on Munson's handprinted data. *IEEE Transactions on Computers*, 21:201–205, February 1972.

[58] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. In D. A. Norman and S. W. Draper, editors, *User Centered System Design*, pages 118–123. Lawrence Erlbaum Associates, Hillsdale, N.J., 1986.

[59] Pencept Inc. Software control at the stroke of a pen. In *SIGGRAPH Video Review*, volume Issue 18: Edited Compilations from CHI '85. ACM, 1985.

[60] F. Itakura. Minimum prediction residual principle applied to speech recognition. *IEEE Trans. Acoustics, Speech, Signal Processing*, ASSP-23(67), 1975.

[61] J. C. Jackson and Renate J. Roske-Hofstrand. Circling: A method of mouse-based selection without button presses. In *CHI '89 Proceedings*, pages 161–166. ACM, May 1989.

[62] Mike James. *Classification Algorithms*. Wiley-Interscience. John Wiley and Sons, Inc., New York, 1985.

[63] R. E. Johnson. Model/View/Controller. November 1987 (unpublished manuscript).

[64] Dan R. Olsen Jr. Syngraph: A graphical user interface generator. *Computer Graphics*, 17(3):43–50, July 1983.

[65] K. G. Morse Jr. In an upscale world. *Byte*, 14(8), August 1989.

[66] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, New Jersey, 1978.

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

[68] Nancy T. Knolle. Variations of model-view-controller. *Journal of Object-Oriented Programming*, 2:42–46, September/October 1989.

[69] D. Kolzay. Feature extraction in an optical character recognition machine. *IEEE Transactions on Computers*, 20:1063–1067, 1971.

[70] Glenn E. Krasner and Stephen T. Pope. A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, August 1988.

[71] M. W. Kreuger. *Artificial Reality*. Addison-Wesley, Reading, MA., 1983.

[72] M. W. Kreuger, T. Gionfriddo, and K. Hinrichsen. Videoplace: An artificial reality. In *Proceedings of CHI'85*, pages 35–40. ACM, 1985.

[73] E. Kreyszig. *Advanced Engineering Mathematics*. Wiley, 1979. Fourth Edition.

[74] W. J. Krzanowski. *Principles of Multivariate Analysis: A User's Perspective*. Oxford Statistical Science Series. Clarendon Press, Oxford, 1988.

[75] G. Kurtenbach and W. A. S. Buxton. GEdit: A test bed for editing by contiguous gestures. *SIGCHI Bulletin*, pages 22–26, 1991.

[76] Martin Lamb and Veronica Buckley. New techniques for gesture-based dialog. In B. Shackel, editor, *Human-Computer Interaction - Interact '84, IFIP*, pages 135–138, North-Holland, 1985. Elsevier Science Publishers B.V.

[77] C. G. Leedham, A. C. Downton, C. P. Brooks, and A. F. Newell. On-line acquisition of pitman's handwritten shorthand as a means of rapid data entry. In B. Shackel, editor, *Human-Computer Interaction - Interact '84, IFIP*, pages 145–150, North-Holland, 1985. Elsevier Science Publishers B.V.

[78] Barbara Staudt Lerner. *Automated Customization of User Interfaces*. PhD thesis, Carnegie Mellon University, 1989.

[79] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

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

[81] D. J. Lyons. Go Corp. gains ground in pen-software race. *PC Week*, 7(29):135, July 23 1990.

[82] Gale Martin, James Pittman, Kent Wittenburg, Richard Cohen, and Tome Parish. Sign here, please. *Byte*, 15(7):243–251, July 1990.

[83] J. T. Maxwell. Mockingbird: An interactive composer's aid. Master's thesis, MIT, 1981.

[84] P. McAvinney. The Sensor Frame–a gesture-based device for the manipulation of graphic objects. Available from Sensor Frame, Inc., Pittsburgh, Pa., December 1986.

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

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

[87] P. Morrel-Samuels. Clarifying the distinction between lexical and gestural commands. *International Journal of Man-Machine Studies*, 32:581–590, 1990.

[88] G. Muller and R. Giulietti. High quadlity music notation: Interactive editing and input by piano keyboard. In *Proceedings of the 1987 International Computer Music Conference*, pages 333–340, San Francisco, 1987. Computer Music Association.

[89] Hiroshi Murase and Toru Wakahara. Online hand-sketched figure recognition. *Pattern Recognition*, 19(2):147–160, 1988.

[90] B. Myers and W. A. S. Buxton. Creating highly-interactive and graphical user interfaces by demonstration. *Computer Graphics*, 20(3):249–258, 1986.

[91] B. A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 1990.

[92] B. A. Myers, D. Giuse, R. B. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, Andrew Mickish, and Phillippe Marchal. Garnet: comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, Nov 1990.

[93] B. A. Myers, B. Vander Zanden, and R. B. Dannenberg. Creating graphical interactive application objects by demonstration. In *UIST '89: Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 95–104. ACM, November 1989.

[94] Brad A. Myers. A taxonomy of user interfaces for window managers. *IEEE Computer Graphics and Applications*, 8(5):65–84, 1988.

[95] Brad A. Myers. Encapsulating interactive behaviors. In *Human Factors in Computing Systems*, pages 319–324, Austin, TX, April 1989. Proceedings SIGCHI'89.

[96] Brad A. Myers. User interface tools: Introduction and survey. *IEEE Software*, 6(1):15–23, January 1989.

[97] Brad A. Myers. Demonstration interfaces: A step beyond direct manipulation. Technical Report CMU-CS-90-162, Carnegie Mellon School of Computer Science, Pittsburgh, PA, August 1990.

[98] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, March 1990.

[99] L. Nakatani. Personal communication, Bell Laboratories, Murray Hill, N.J. January 1987.

[100] T. Neuendorffer. *Adew Reference Manual*. Information Technology Center, Pittsburgh, PA, 1989.

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

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

[103] L. Norton-Wayne. A coding approach to pattern recognition. In J. Kittler, K. S. Fu, and L. F. Pau, editors, *Pattern Recognition Theory and Applications*, NATO Advanced Study, pages 93–102. D. Reidel, 1981.

[104] K. K. Obermeier and J. J. Barron. Time to get fired up. *Byte*, 14(8), August 1989.

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

[106] PC Computing. Ten other contendors in the featherweight division. *PC Computing*, 2(12):89–90, December 1989.

[107] J. A. Pickering. Touch-sensitive screens: the tecnologies and their application. *International Journal of Man-Machine Studies*, 25:249–269, 1986.

[108] R. Probst. Blueprints for building user interfaces: Open Look toolkits. Technical report, Sun Technology, August 1988.

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

[110] J. Rosenberg, R. Hill, J. Miller, A. Schulert, and D. Shewmake. UIMSs: Threat or menace? In *CHI '88*, pages 197–212. ACM, 1988.

[111] D. Rubine and P. McAvinney. The Videoharp. In *1988 International Computer Music Proceedings*. Computer Music Association, September 1988.

[112] D. Rubine and P. McAvinney. Programmable finger-tracking instrument controllers. *Computer Music Journal*, 14(1):26–41, 1990.

[113] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[114] K. J. Schmucker. MacApp: An application framework. *Byte*, 11(8):189–193, August 1986.

[115] Kurt J. Schmucker. *Object-Oriented Programming for the Macintosh*. Hayden Book Company, 1986.

[116] A. C. Shaw. Parsing of graph-representable pictures. *JACM*, 17(3):453, 1970.

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

[118] John L. Sibert, William D. Hurley, and Teresa W. Bleser. An object-oriented user interface management system. In *SIGGRAPH '86*, pages 259–268. ACM, August 1986.

[119] Jack Sklanksy and Gustav N. Wassel. *Pattern Classifiers and Trainable Machines*. Springer-Verlag, New York, 1981.

[120] W. W. Stallings. Recognition of printed chinese characters by automatic pattern analysis. *Computer Graphics and Image Processing*, 1:47–65, 1972.

[121] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*, 6(4):40–62, Winter 1986.

[122] Jess Stein, editor. *The Random House Dictionary of the English Language*. Random House, Cambridge, Mass., 1969.

[123] Martin L. A. Sternberg. *American Sign Language: A Comprehensive Dictionary*. Harper and Row, New York, 1981.

[124] M. D. Stone. Touch-screens for intuitive input. *PC Magazine*, pages 183–192, August 1987.

[125] C. Y. Suen, M. Berthod, and S. Mori. Automatic recognition of handprinted characters: The state of the art. *Proceedings of the IEEE*, 68(4):469–487, April 1980.

[126] Sun. *SunWindows Programmers' Guide*. Sun Miscrosystems, Inc., Mountain View, Ca., 1984.

[127] Sun. *NeWS Preliminary Technical Overview*. Sun Miscrosystems, Inc., Mountain View, Ca., 1986.

[128] Shinichi Tamura and Shingo Kawasaki. Recognition of sign language motion images. *Pattern Recognition*, 21(4):343–353, 1988.

[129] C. C. Tappert, C. Y. Suen, and Toru Wakaha. The state of the art in on-line handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(8):787–808, August 1990.

[130] E. R. Tello. Between man and machine. *Byte*, 13(9):288–293, September 1988.

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

[132] D. S. Touretzky and D. A. Pomerleau. What's hidden in the hidden layers? *Byte*, 14(8), August 1989.

[133] V. M. Velichko and N. G. Zagoruyko. Automatic recognition of 200 words. *Int. J. Man-Machine Studies*, 2(2):223, 1970.

[134] A. Waibel and J. Hampshire. Building blocks for speech. *Byte*, 14(8):235–242, August 1989.

[135] A. I. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering*, SE-11(8):699–713, August 1985.

[136] D. Weimer and S. K. Ganapathy. A synthetic visual environment with hand gesturing and voice input. In *CHI '89 Proceedings*, pages 235–240. ACM, 1989.

[137] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, August 1962.

[138] A. P. Witkin. Scale-space filtering. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1019–1022, 1983.

[139] C. G. Wolf. A comparative study of gestural and keyboard interfaces. *Proceedings of the Humans Factors Society*, 32nd Annual Meeting:273–277, 1988.

[140] C. G. Wolf and J. R. Rhyne. A taxonomic approach to understanding direct manipulation. *Proceedings of the Human Factors Society*, 31st Annual Meeting:576–580, 1987.

[141] Catherine G. Wolf. Can people use gesture commands? Technical Report RC11867, IBM Research, April 1986.

[142] Xerox Corporation. JUNO. In *SIGGRAPH Video Review Issue 19: CHI '85 Compilation*. ACM, 1985.