EXHIBIT 4.21

```
    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 runtime 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.

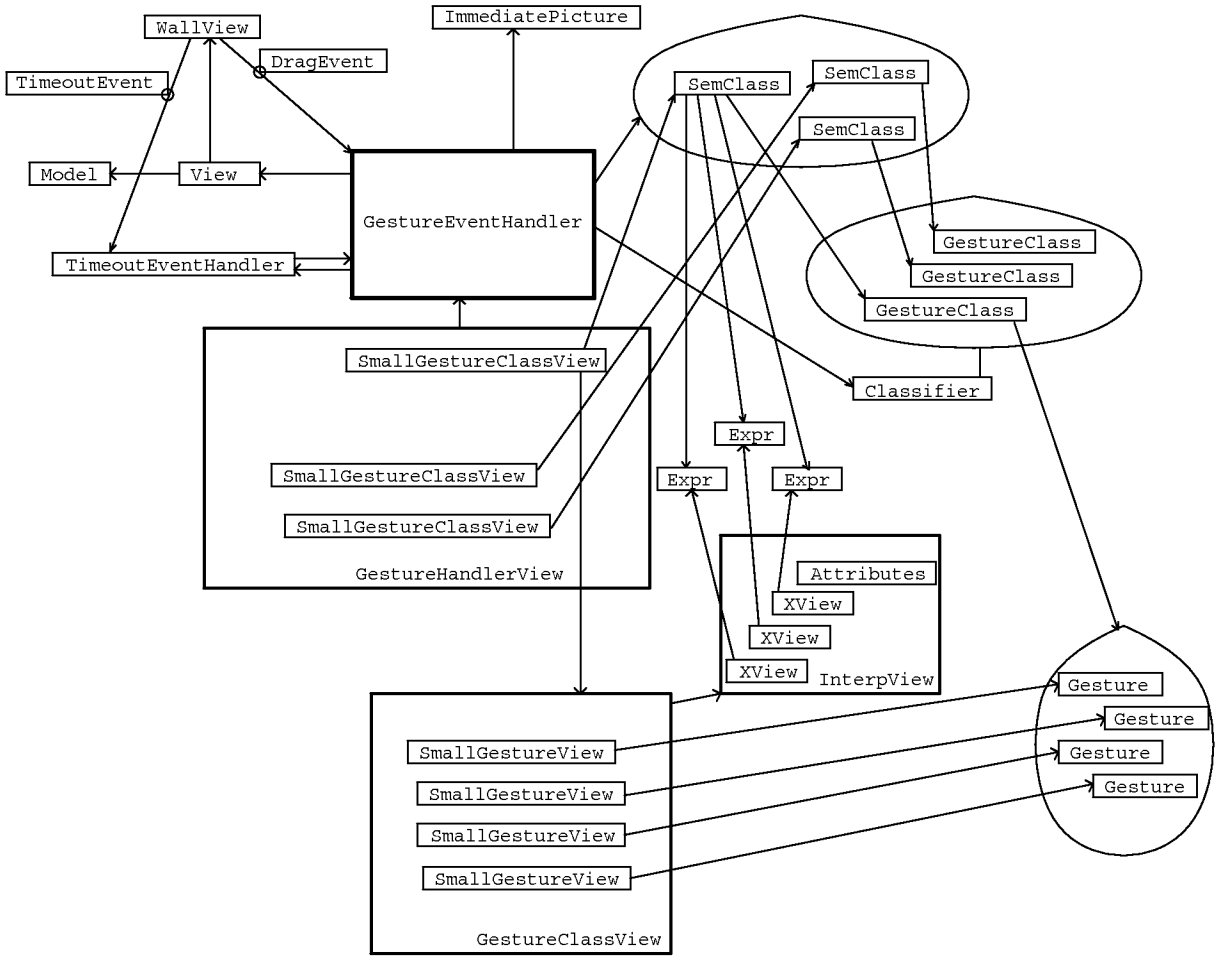

Figure 7.1: GRANDMA's gesture subsystem

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 ids 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 discrimination functions:
           w[i][j] is the weight on the jth feature of the ith class.
           cnst[i] is the constant term for 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

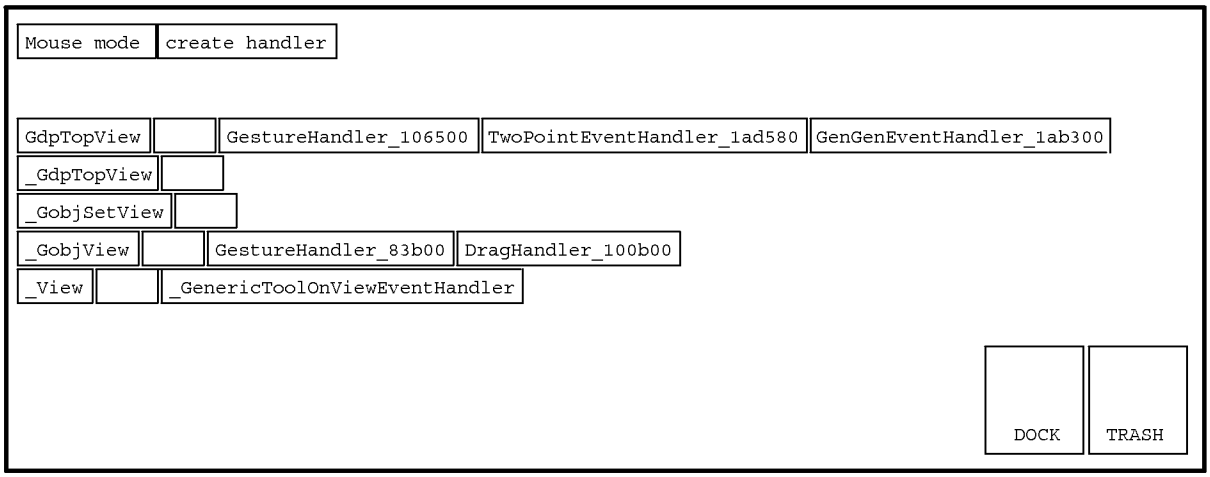

Figure 7.2: Passive Event Handler Lists

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

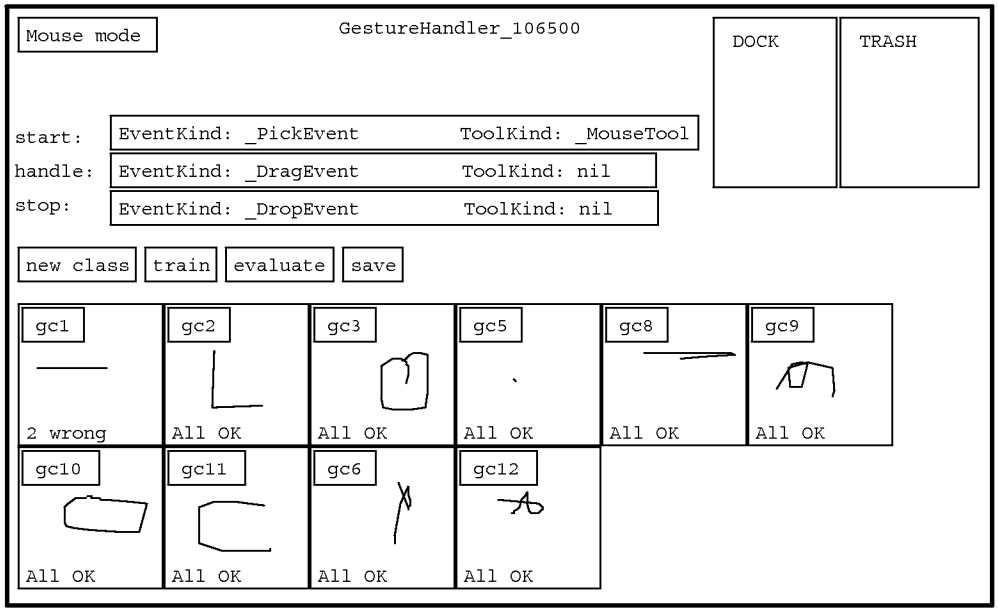

Figure 7.3: A Gesture Event Handler

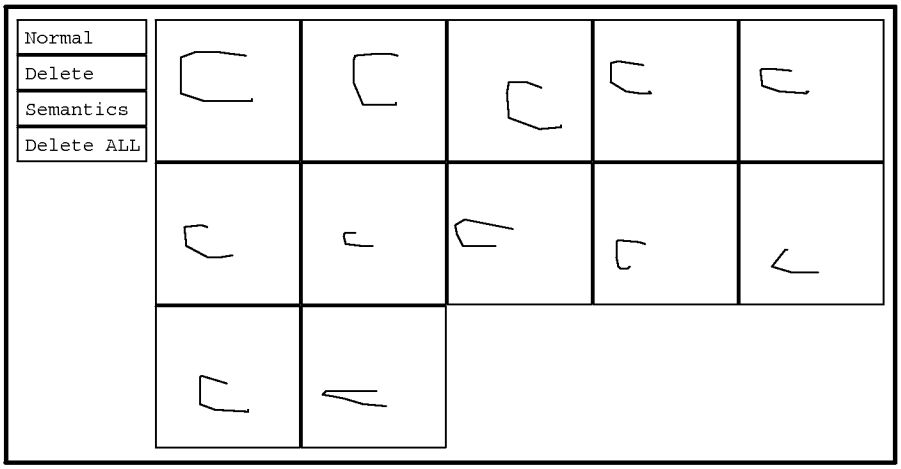

Figure 7.4: Window of examples of a gesture class

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 (| | = 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.

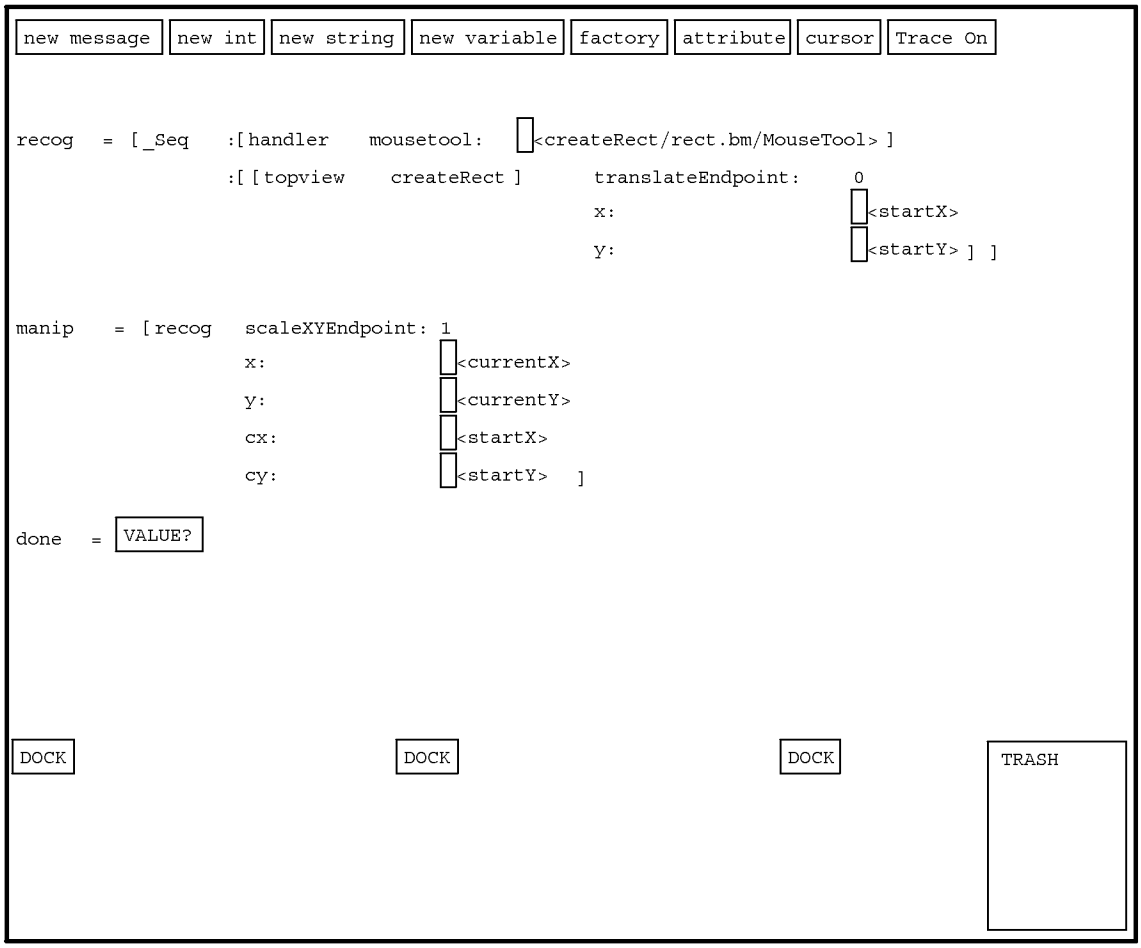

Figure 7.5: The interpreter window for editing gesture semantics

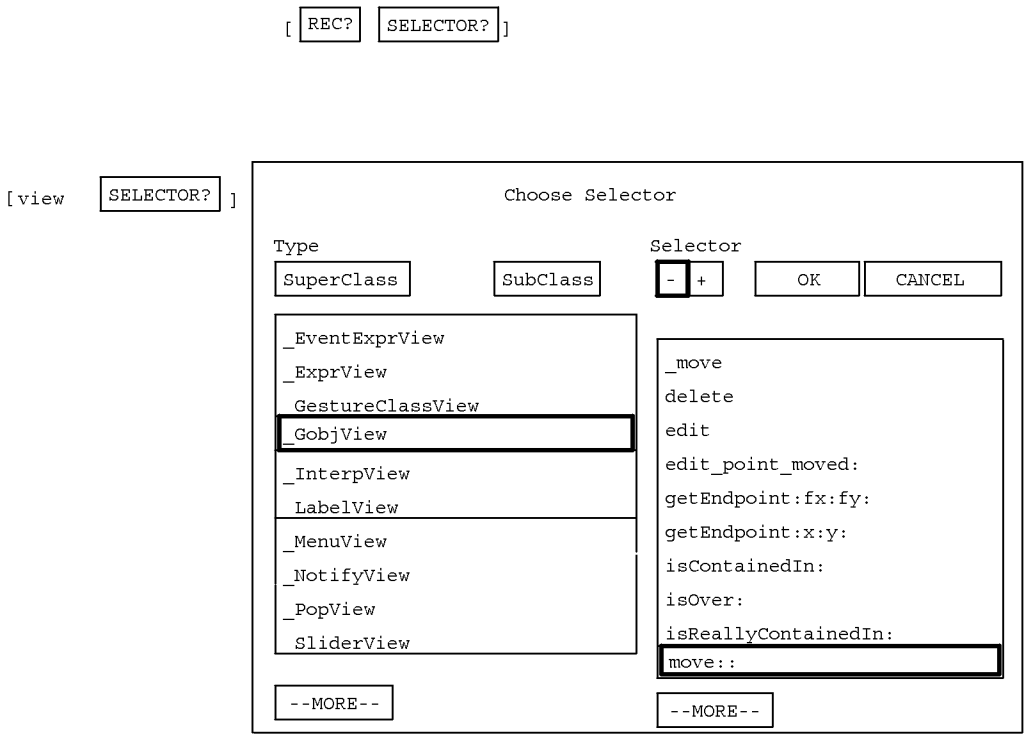

Figure 7.6: An empty message and a selector browser

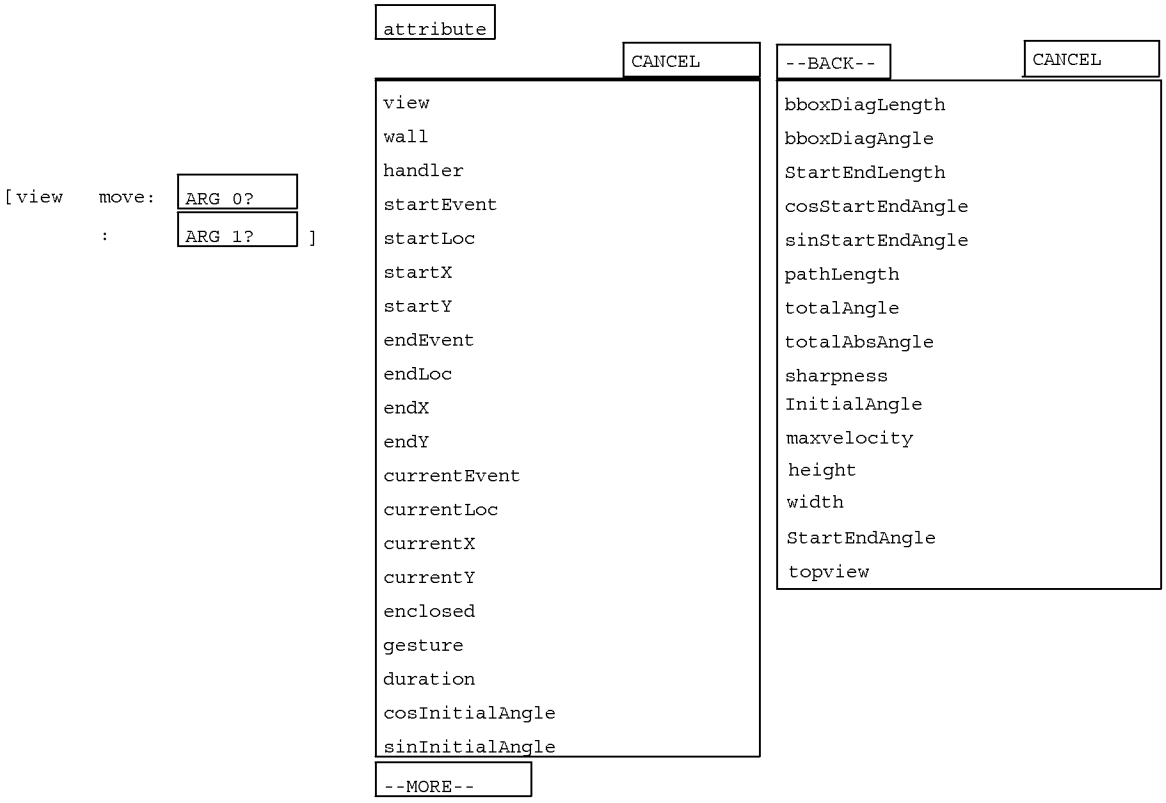

Figure 7.7: Attributes to use in gesture semantics

.

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.

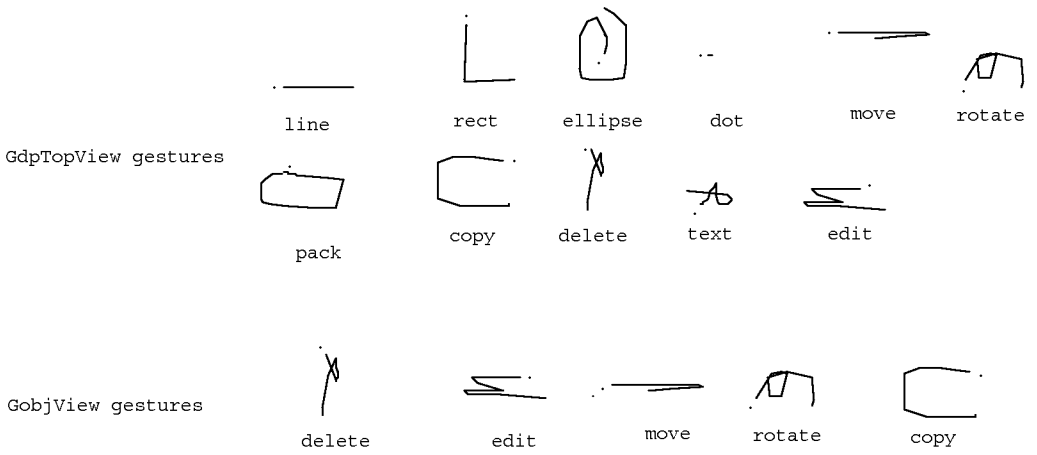

Figure 8.1: GDP gestures

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

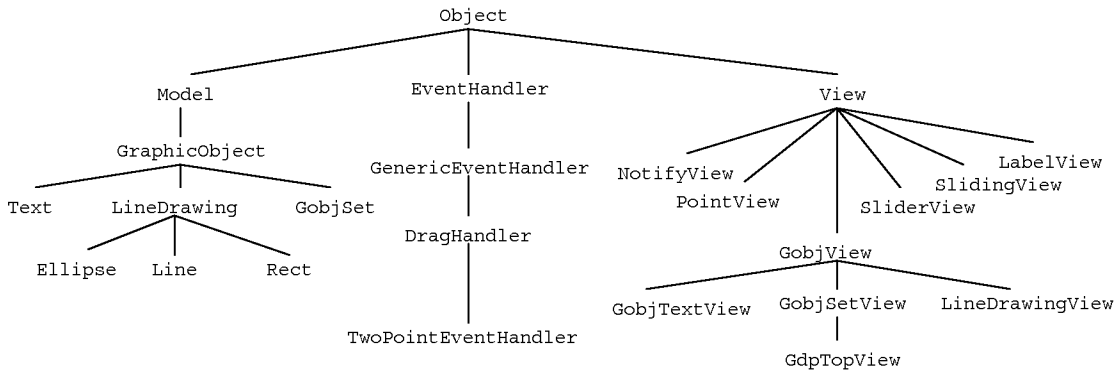

Figure 8.2: GDP's class hierarchy

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

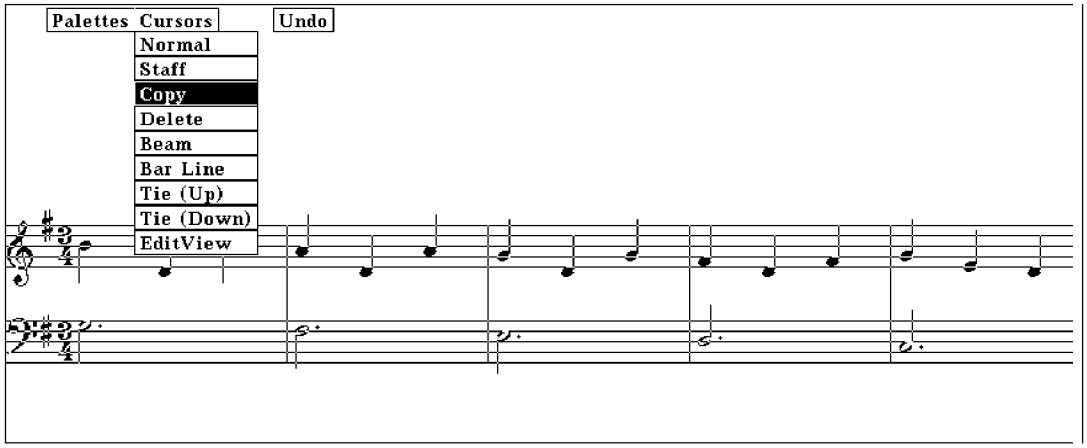

Figure 8.3: GSCORE's cursor menu

```
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

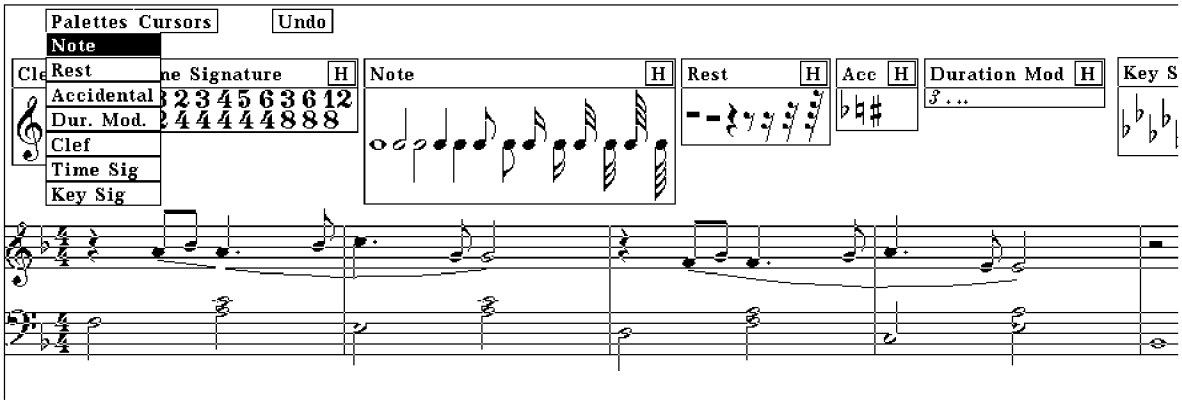

Figure 8.4: GSCORE's palette menu

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.

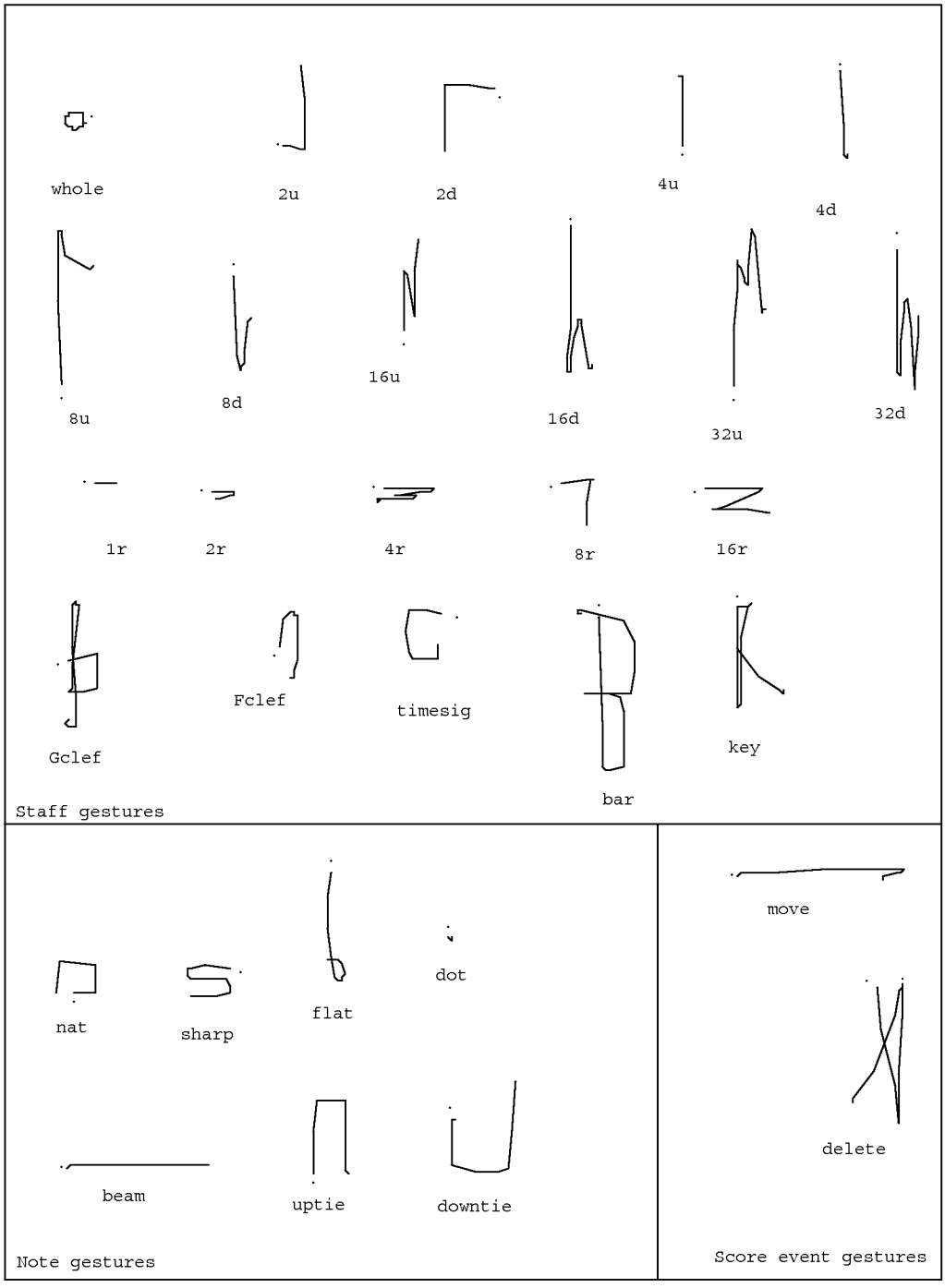

Figure 8.5: GSCORE gestures

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.

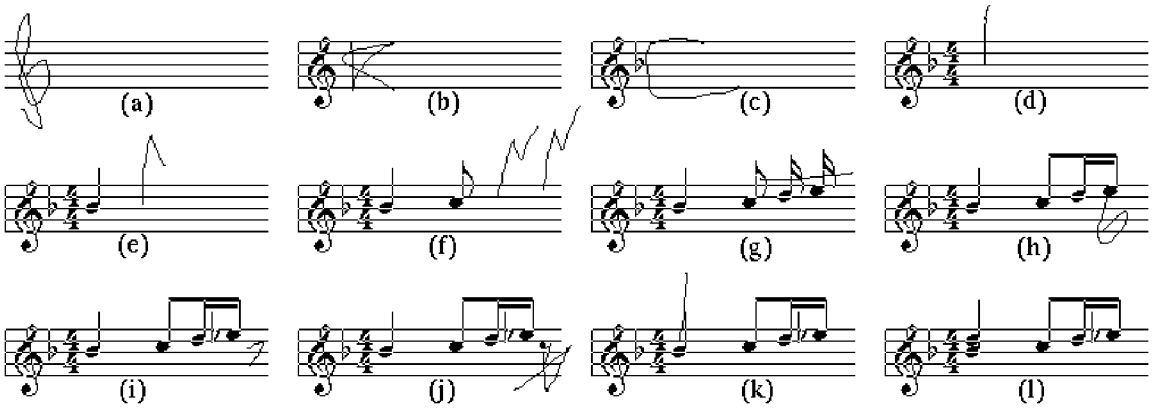

Figure 8.6: A GSCORE session

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).*

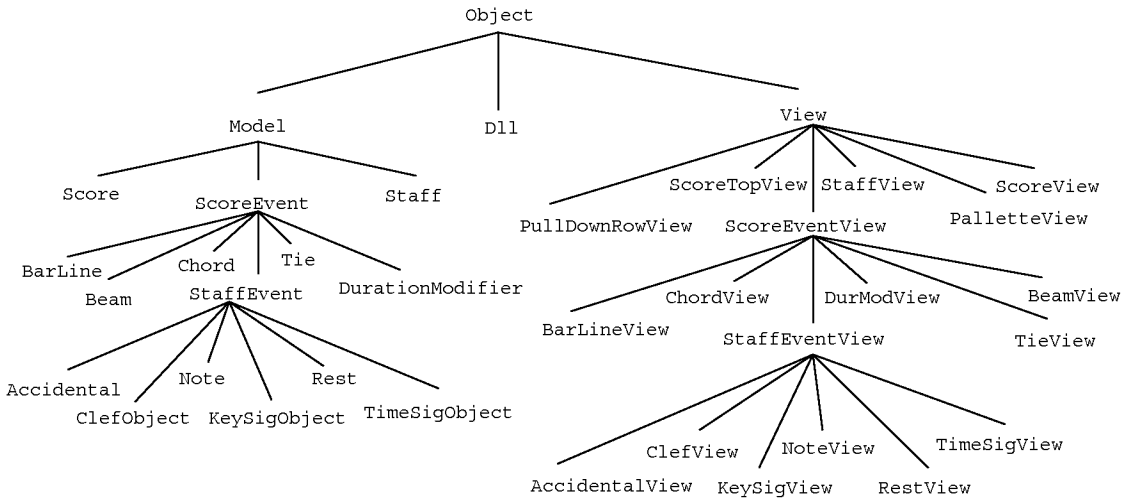

Figure 8.7: GSCORE's class hierarchy

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

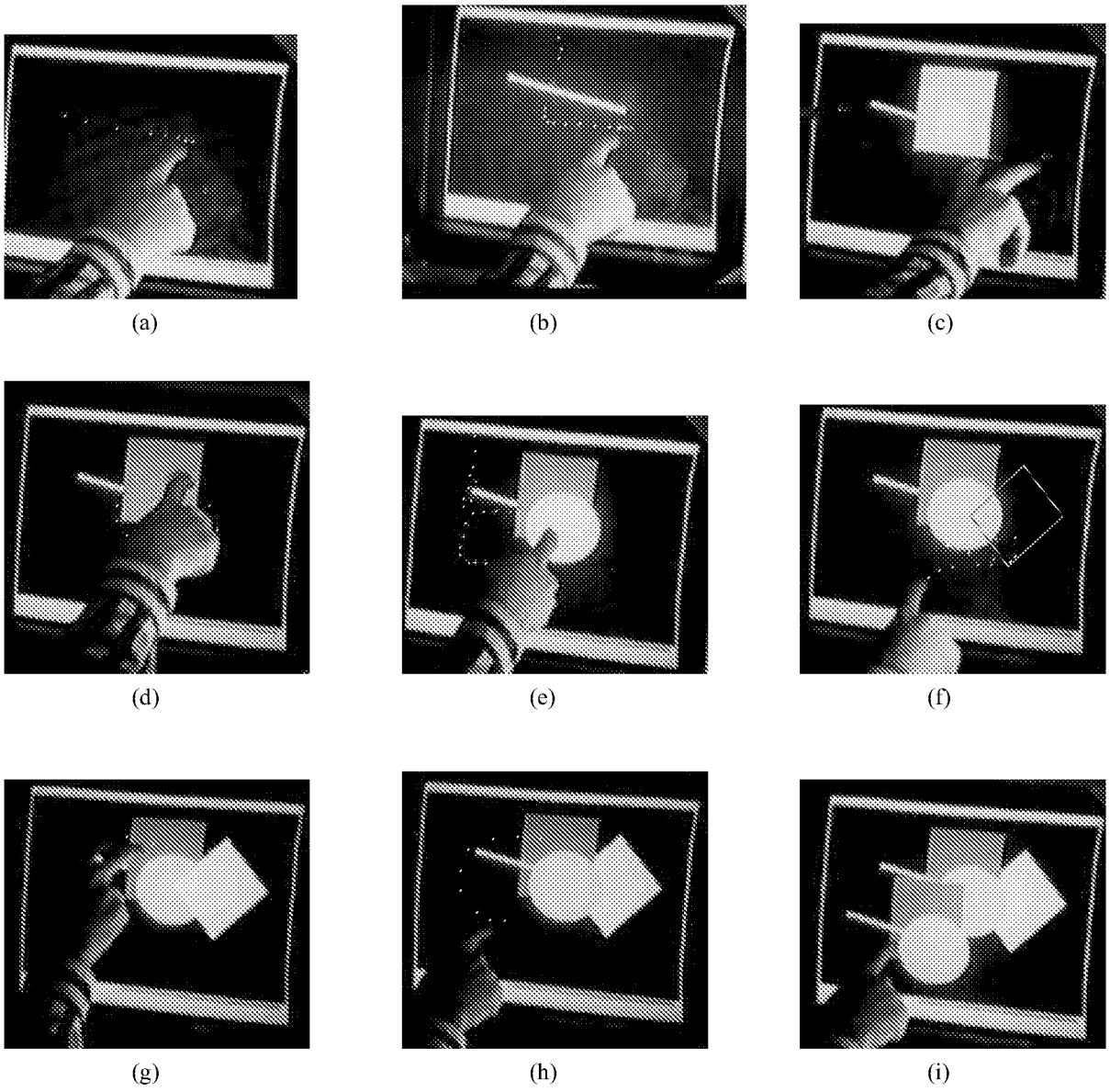

Figure 8.8: An example MDP session

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*

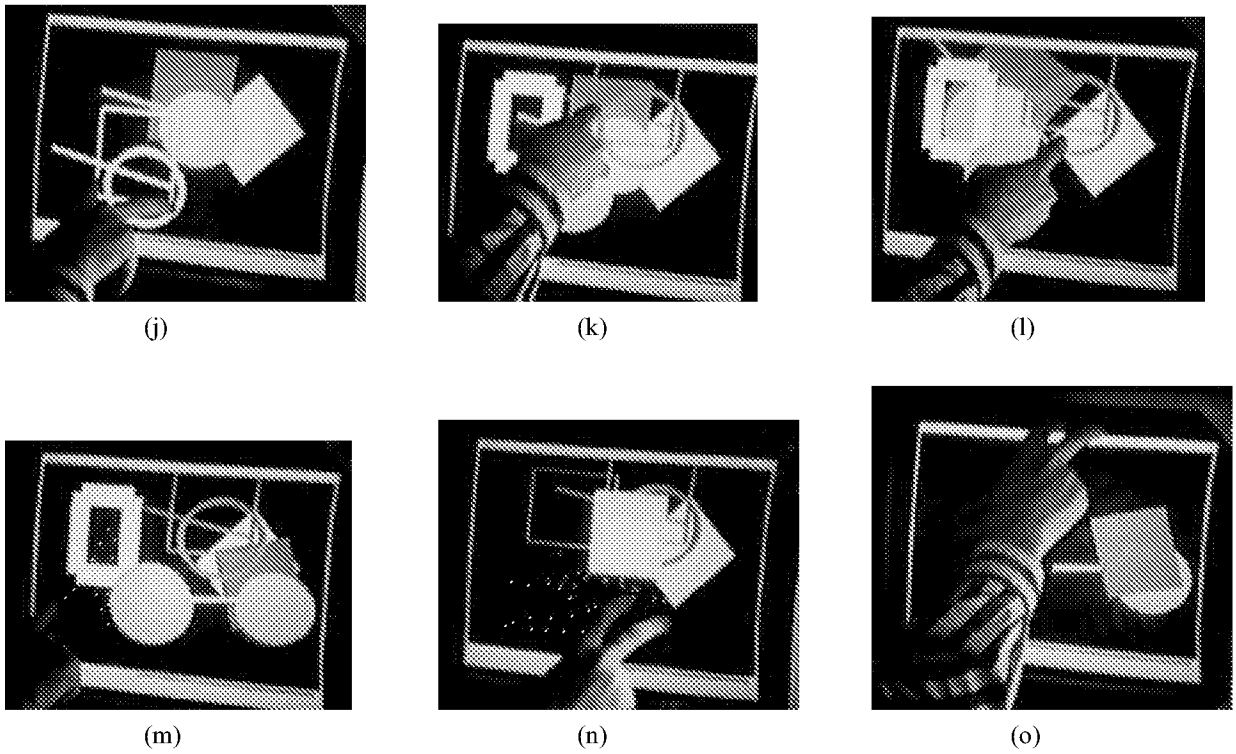

Fig 8.8 (continued)

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).*

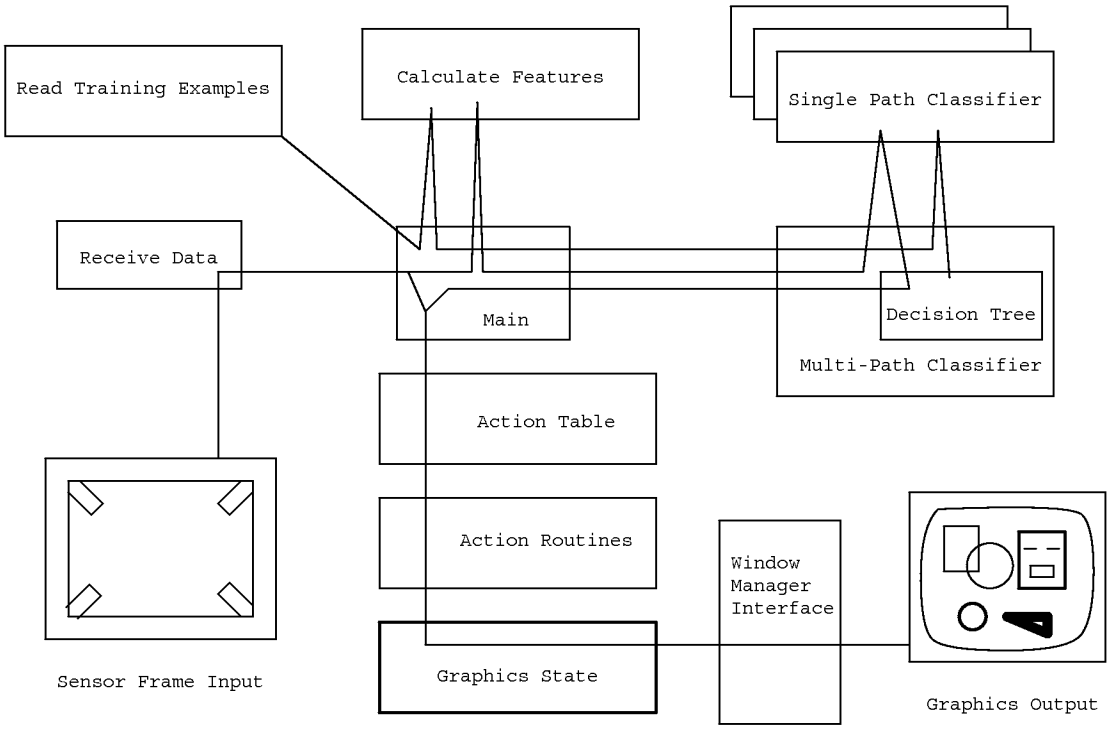

Figure 8.9: MDP internal structure

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

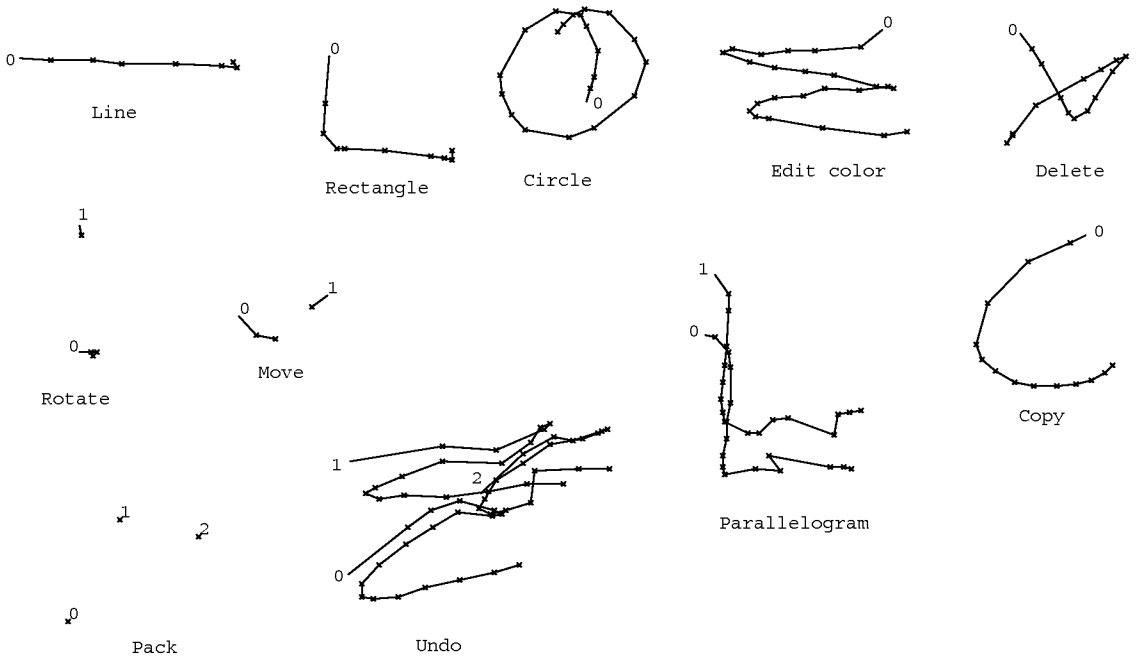

Figure 8.10: MDP gestures

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

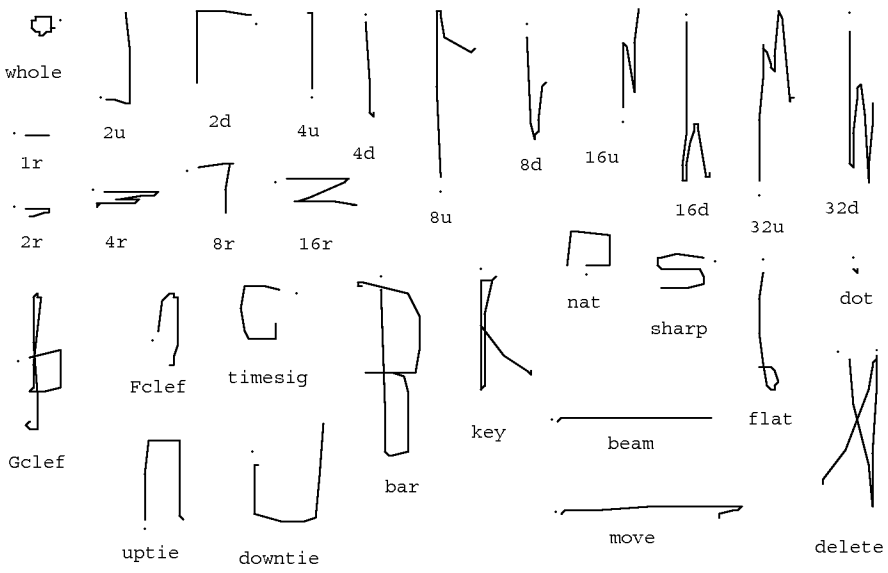

Figure 9.1: GSCORE gesture classes used for evaluation

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

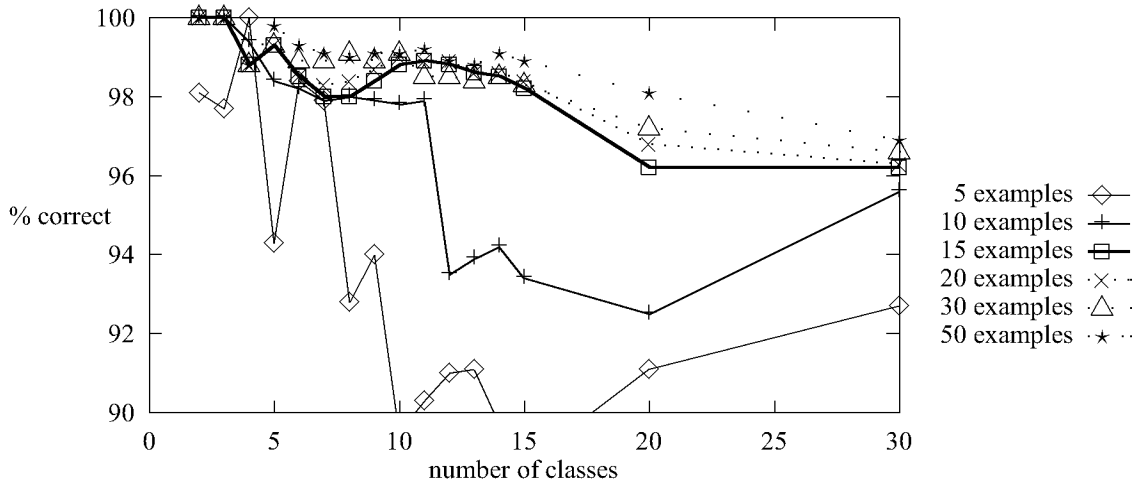

Figure 9.2: Recognition rate vs. number of classes

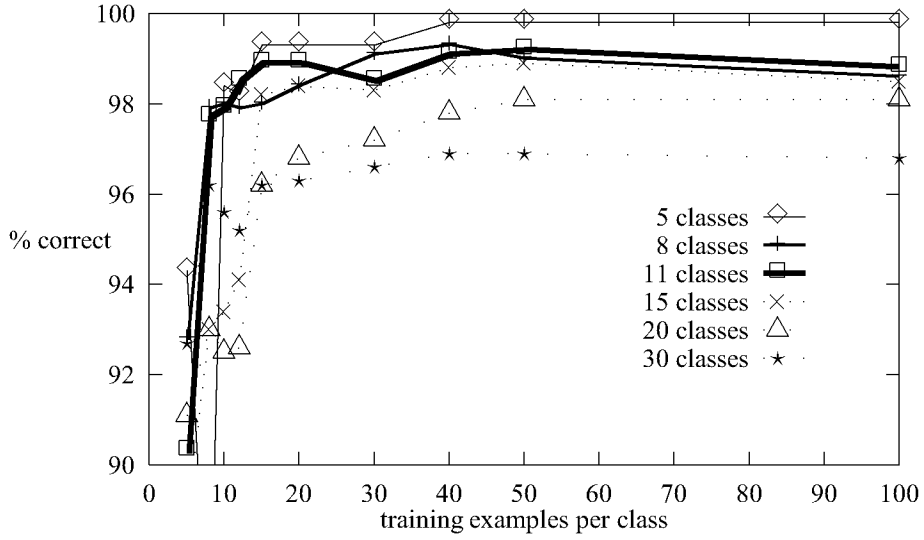

Figure 9.3: Recognition rate vs. training set size

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.