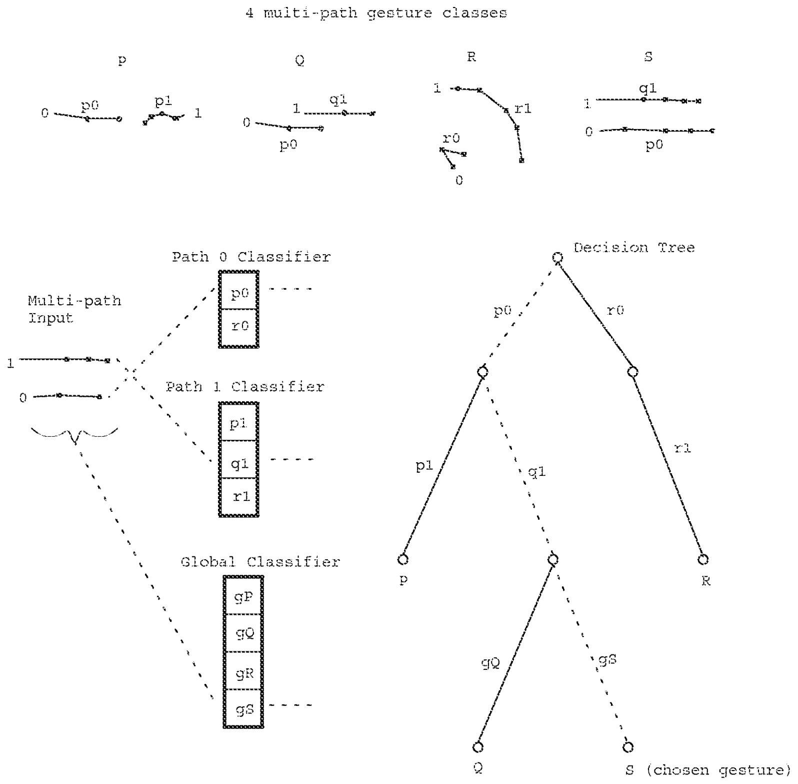


Figure 5.3: Classifying multi-path gestures

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

5.4 Training a Multi-path Classifier

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

5.4.1 Creating the statistical classifiers

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

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

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

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

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

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

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

5.4.2 Creating the decision tree

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

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

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

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

5.5 Path Features and Global Features

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

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

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

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

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

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

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

5.6 A Further Improvement

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

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

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

5.7 An Alternate Approach: Path Clustering

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

5.7.1 Global features without path sorting

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

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

5.7.2 Multi-path recognition using one single-path classifier

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

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

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

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

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

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

5.7.3 Clustering

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

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

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

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

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

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

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

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

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

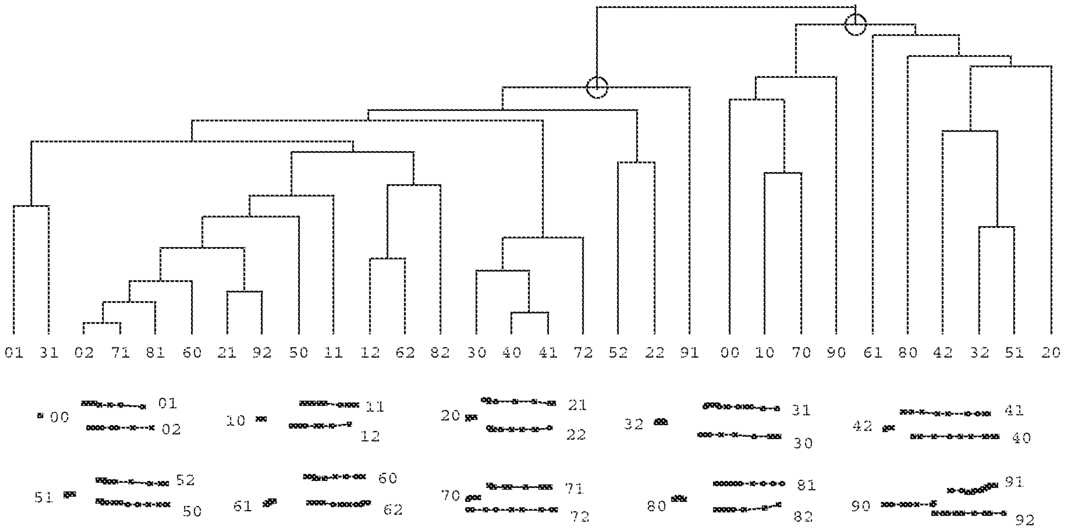


Figure 5.4: Path Clusters

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

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

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

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

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

5.7.4 Creating the decision tree

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

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

5.8 Discussion

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

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

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

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

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

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

5.9 Conclusion

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

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

Chapter 6

An Architecture for Direct Manipulation

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

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

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

6.1 Motivation

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

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

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

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

6.2 Architectural Overview

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

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

6.2.1 An example: pressing a switch

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

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

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

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

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

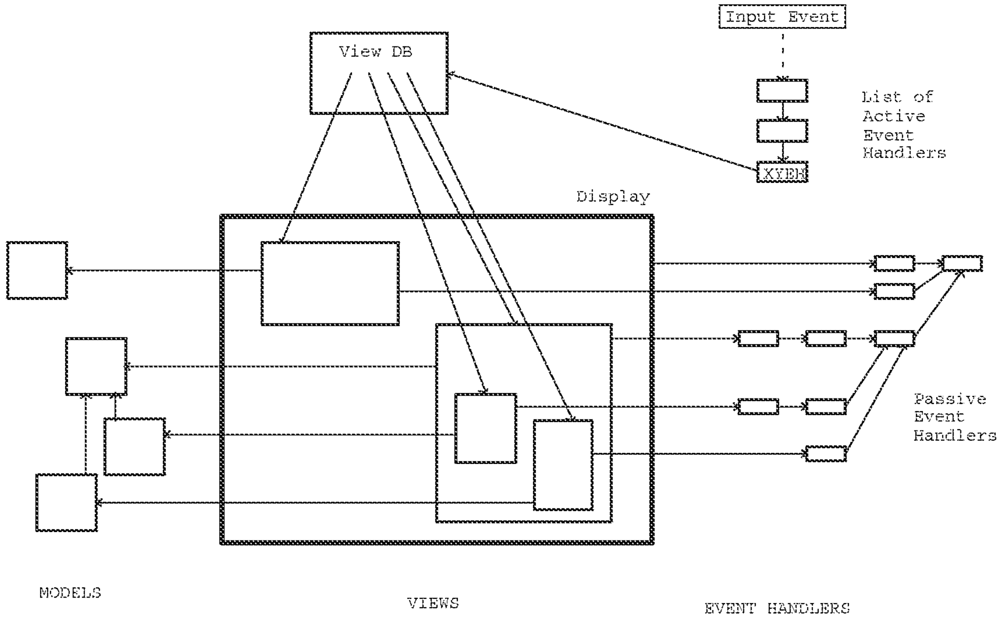


Figure 6.1: GRANDMA's Architecture

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

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

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

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

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

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

6.2.2 Tools

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

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

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

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

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

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

6.3 Objective-C Notation

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

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

```
        id aSet;
```

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

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

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

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

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

newly allocated instance of itself.

Messages may also have parameters, as in

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

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

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

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

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

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

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

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

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

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

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

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

6.4 The Two Hierarchies

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

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

6.5 Models

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

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

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

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

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

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

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

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

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

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

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

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

6.6 Views

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

Views have a number of instance variables (fields):

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

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

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

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

To create a toggle switch view:

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

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

The graphics for the switch are implemented as:

```
 = SwitchView ...
 - updatePicture {
     id p = [self VbeginPicture];
```

```
        [p rectangle 0:0 :10:10];
        if([model getState])
            [p rectangle 2:2 :8:8];
        [self VendPicture];
        return self;
    }
```

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

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

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

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

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

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

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

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

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

6.7 Event Handlers

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

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

6.7.1 Events

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

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

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

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

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

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

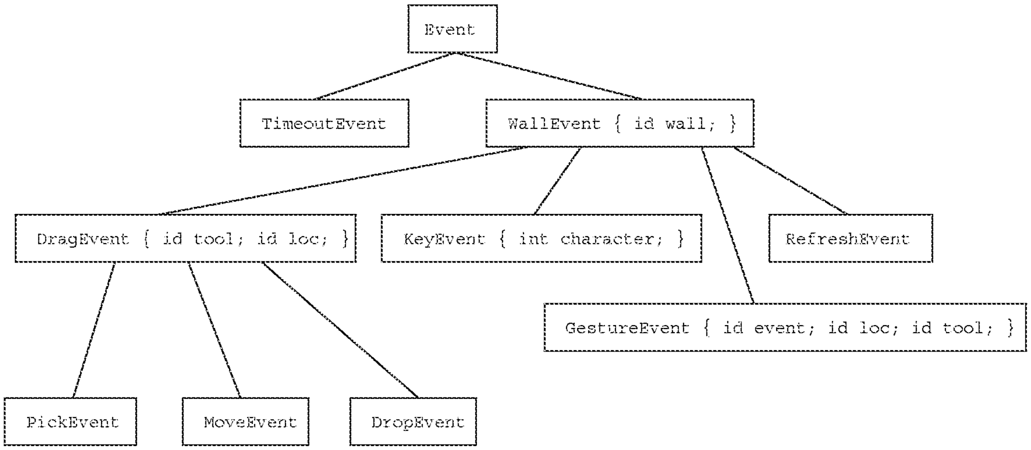

Event
TimeoutEvent
WallEvent { id wall; }
DragEvent { id tool; id loc; }
KeyEvent { int character; }
RefreshEvent
GestureEvent { id event; id loc; id tool; }
PickEvent
MoveEvent
DropEvent

Figure 6.2: The `Event` Hierarchy

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

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

6.7.2 Raising an Event

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

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

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

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

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

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

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

6.7.3 Active Event Handlers

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

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

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

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

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

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

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

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

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

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

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

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

6.7.4 The View Database

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

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

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

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

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

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

6.7.5 The Passive Event Handler Search Continues

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

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

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

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

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

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

6.7.6 Passive Event Handlers

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

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

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

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

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

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

```
        return YES;
    }
```

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

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

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

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

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

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

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

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

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

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

6.7.7 Semantic Feedback

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

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

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

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

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

```
        return self;
    }
```

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

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

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

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

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

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

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

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

6.7.8 Generic Event Handlers

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

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

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

- shouldActivate { shouldActivate = YES; return self; }

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

- stopp:_stopp { stopp = _stopp; return self; }
- stopp { return stopp; }
- (BOOL)evalstop:env { return [[stopp eval:env] asBOOL]; }
- handlep:_handlep { handlep = _handlep; return self; }
- handlep { return handlep; }
- (BOOL)evalhandle:env
    { return [[handlep eval:env] asBOOL]; }

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

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

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

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

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

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

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

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

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

6.7.9 The Drag Handler

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

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

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

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

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

/* changing default parameters: */

/* copyviewON causes the view to be copied and then the copy to be dragged */
- copyviewON { copyview = YES; return self; }

/* genEventsOFF makes the handler not raise any events */
- genEventsOFF { genevents = NO; return self; }

/* relativeON makes the handler send the move:: message, passing
  relative coordinates (deltas from the current position) */
- relativeON { relative = YES;
             whenmoved = @selector(move::); }

/* whendone: sets the message sent on the event that terminates the drag */
- whendone:(SEL)sel { whendone = sel; return self; }

/* whenmoved: sets the message sent for every point in the drag */
- whenmoved:(SEL)sel { whenmoved = sel; return self; }

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

```
        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.

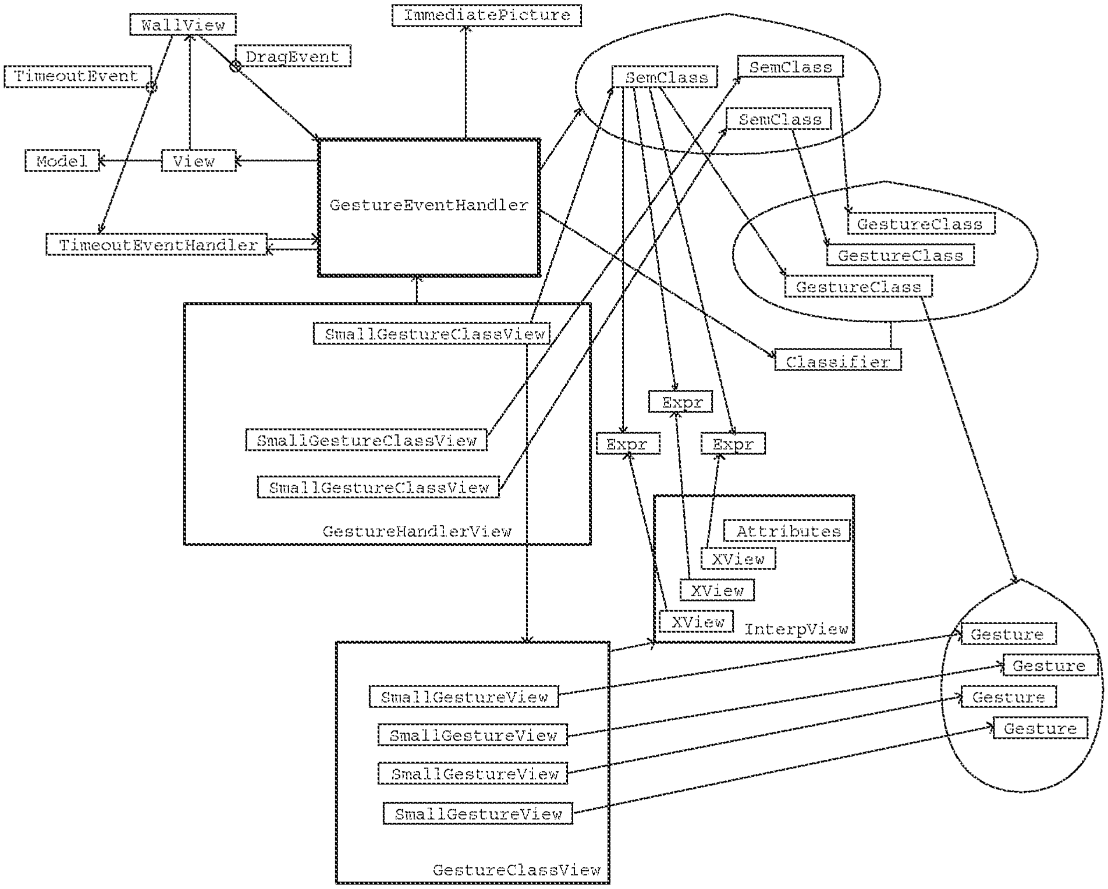


Figure 7.1: GRANDMA's gesture subsystem

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 `id`s as instance variables), made it trivial to save interpreter expressions to disk.

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

7.5.4 Class `Classifier`

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

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

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

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

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

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

    /* calculate the average covariance matrix from the (unnormalized)
       covariance matrices of the gesture classes. */
    avgcov = NewMatrix(nfeatures, nfeatures);
    ZeroMatrix(avgcov);
    for(seq = [gestureclasses eachElement];
        c = [[seq next] gclass]; ) {
            denom += [c nexamples] - 1;
            s = [c sumcov];
            for(i = 0; i < nfeatures; i++)
                for(j = i; j < nfeatures; j++)
                    avgcov[i][j] += s[i][j];
    }

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

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

    /* calculate the 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

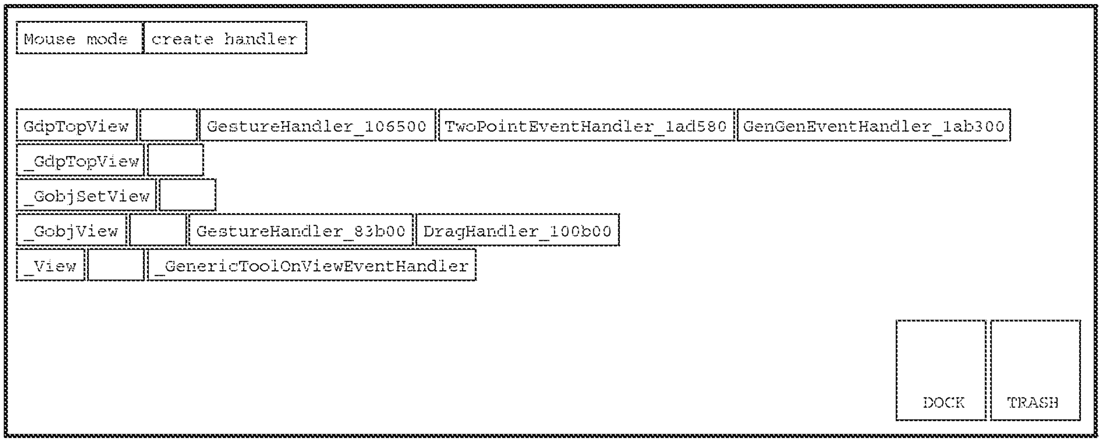


Figure 7.2: Passive Event Handler Lists

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

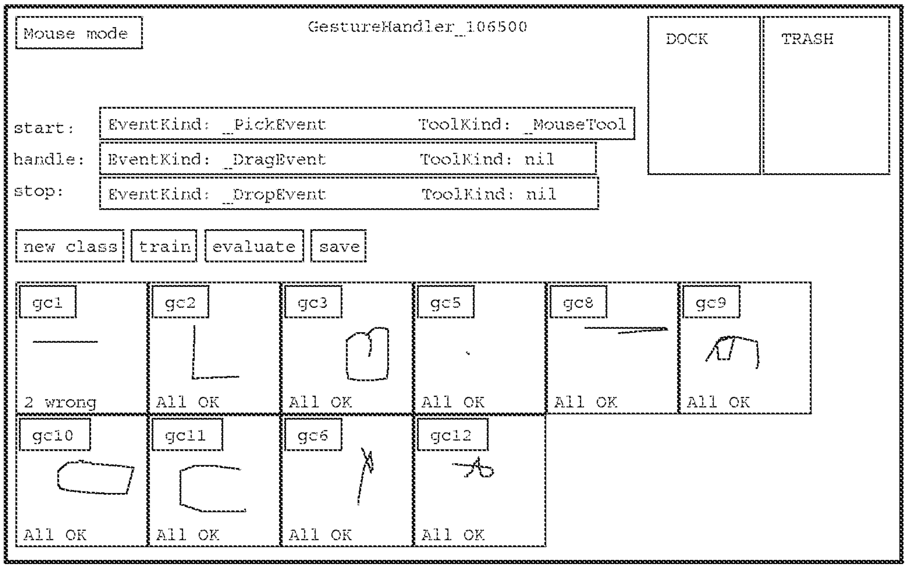
The other mouse cursor, "edit handler", may be clicked upon any passive event handler. It results in a new window being created which shows the details of a particular edit handler. Figure 7.3 shows the window for a typical gesture handler.

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



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

Figure 7.3: A Gesture Event Handler

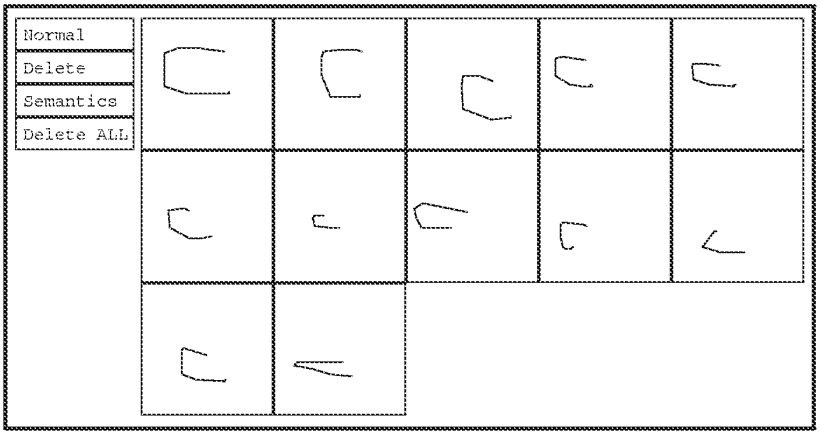


Figure 7.4: Window of examples of a gesture class

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

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

7.7 Gesture Semantics

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

7.7.1 Gesture Semantics Code

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.7.2 The User Interface

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

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

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

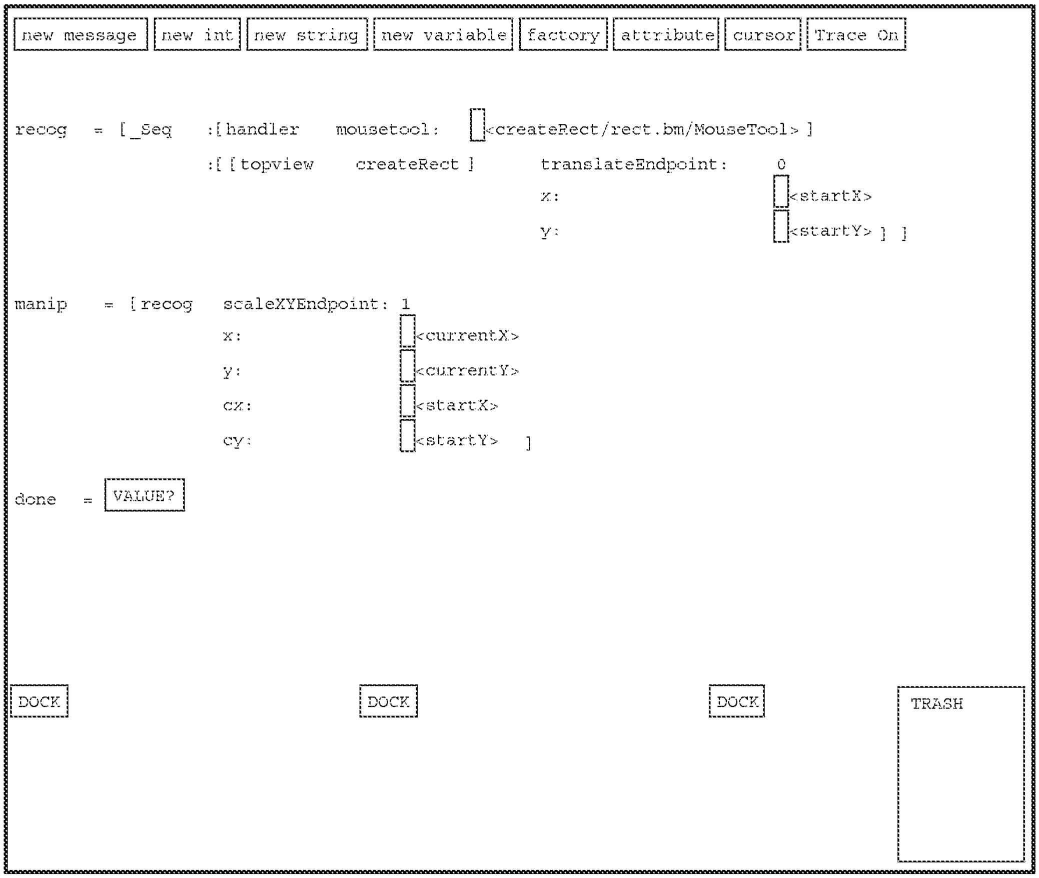
new message The new message button creates a template of a message send, with a slot for the receiver and the message selector. Any expression may then be dragged into the receiver ("REC?") slot. Clicking on the "SELECTOR?" box causes a dialogue box to be displayed (figure 7.6). Users can then browse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

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

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

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

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



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

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

done   =  VALUE?
```

Figure 7.5: The interpreter window for editing gesture semantics

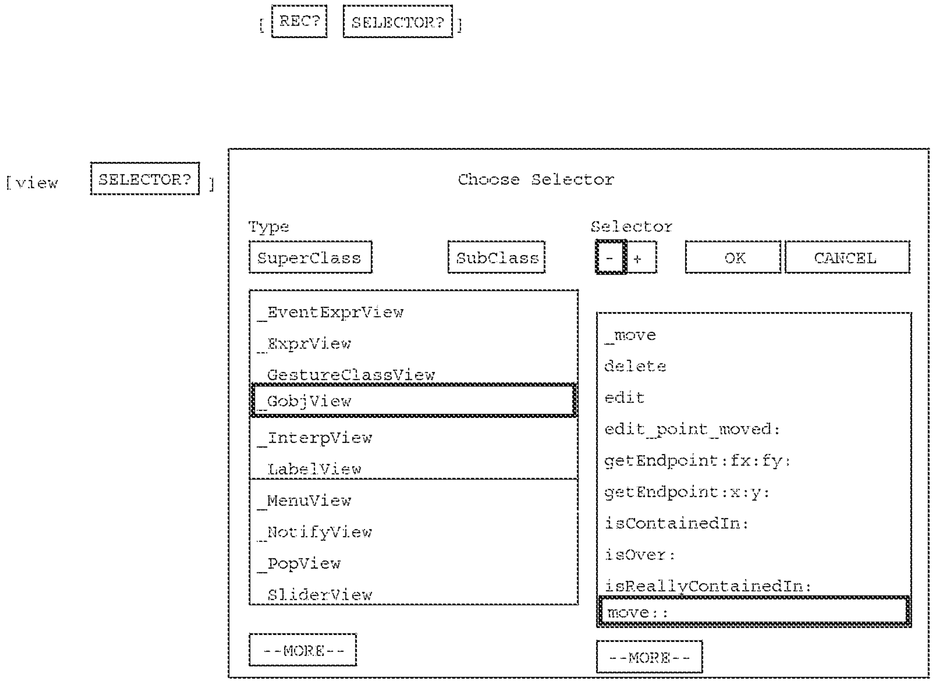


Figure 7.6: An empty message and a selector browser

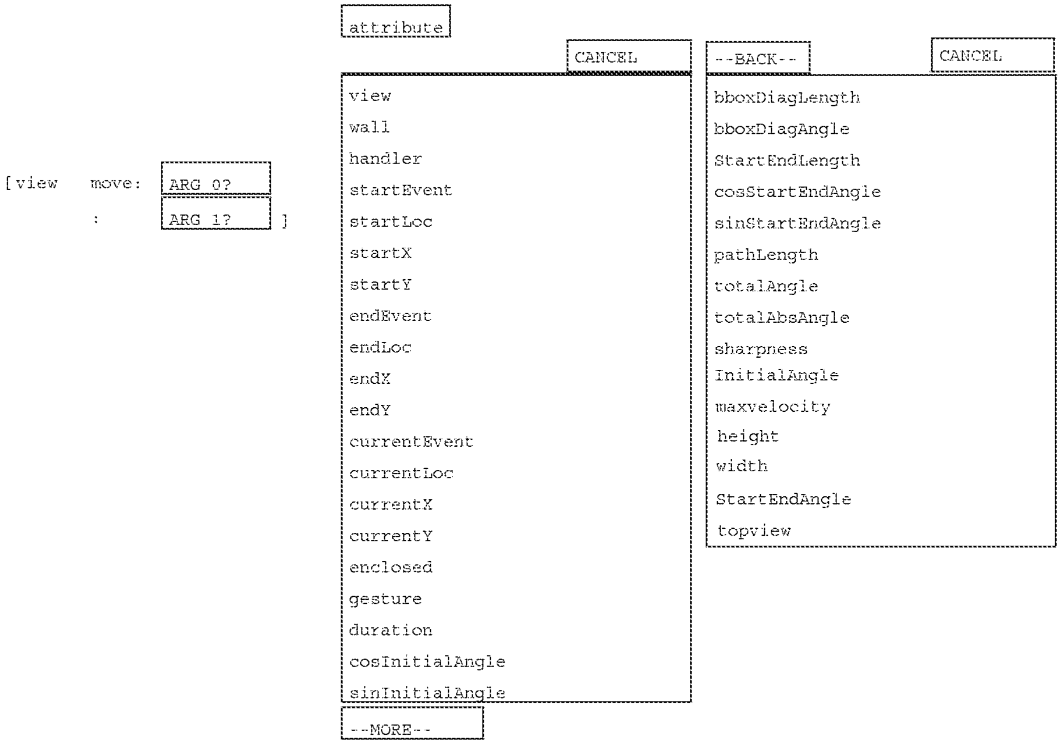


Figure 7.7: Attributes to use in gesture semantics

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

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

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

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

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

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

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

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

7.7.3 Interpreter Implementation

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

Expression Evaluation

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

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

Consider the implementation of `TypeInt`:

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

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

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

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

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

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

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

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

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

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

The most interesting case is the evaluation of a MessageExpr:

```
= MessageExpr:  Expression {
        id     sel;              /* Selector object */
        id     rec;              /* (unevaluated) receiver object */
        id      arg[MAXARGS];       /* unevaluated arguments */
}

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

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

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

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

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

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

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

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

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

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

Interface Implementation

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

Control Constructs

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

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

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

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

Attributes and Cursors

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

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

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

Internally,

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

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

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

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

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

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

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

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

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

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

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

7.8 Conclusion

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

Chapter 8

Applications

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

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

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

8.1 GDP

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

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

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

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

8.1.1 GDP's gestural interface

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

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

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

8.1.2 GDP Implementation

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

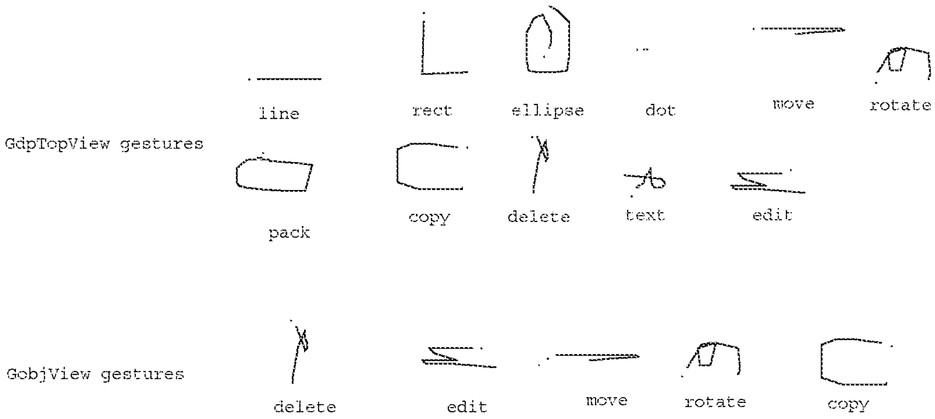


Figure 8.1: GDP gestures

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

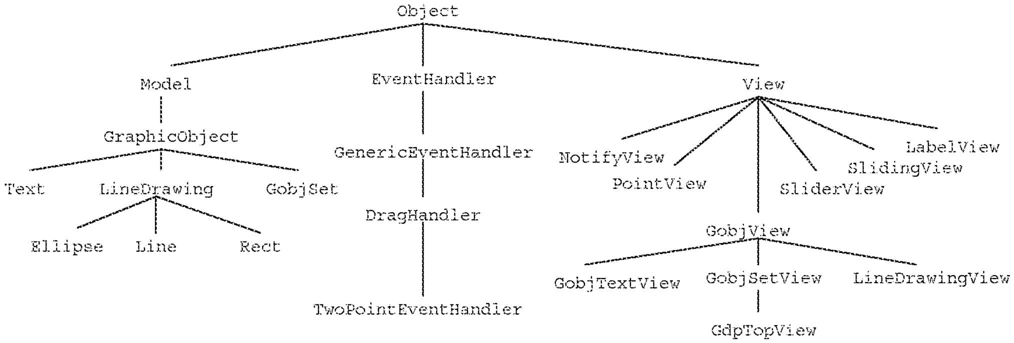


Figure 8.2: GDP's class hierarchy

8.1.3 Models

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

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

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

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

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

8.1.4 Views

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

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

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

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

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

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

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

8.1.5 Event Handlers

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

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

8.1.6 Gestures in GDP

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

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

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

The semantics for the line gesture are:

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

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

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

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

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

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

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

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

The pack gesture has semantics:

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

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

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

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

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

The delete gesture has semantics

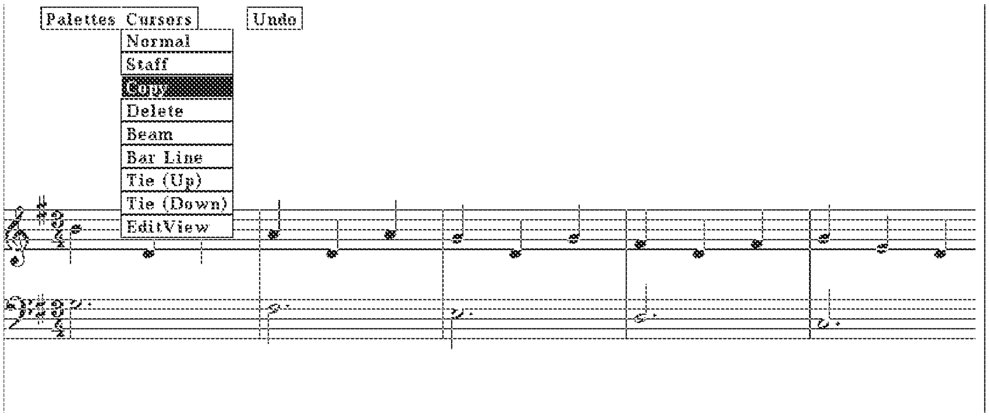


Figure 8.3: GSCORE's cursor menu

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

The edit gesture semantics are similar.

The rotate gesture has semantics:

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

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

8.2 GSCORE

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

8.2.1 A brief description of the interface

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

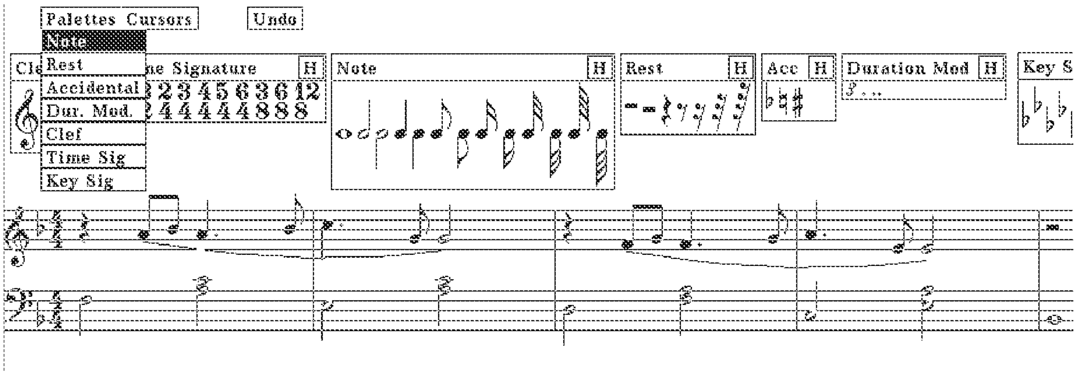


Figure 8.4: GSCORE's palette menu

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

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

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

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

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

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

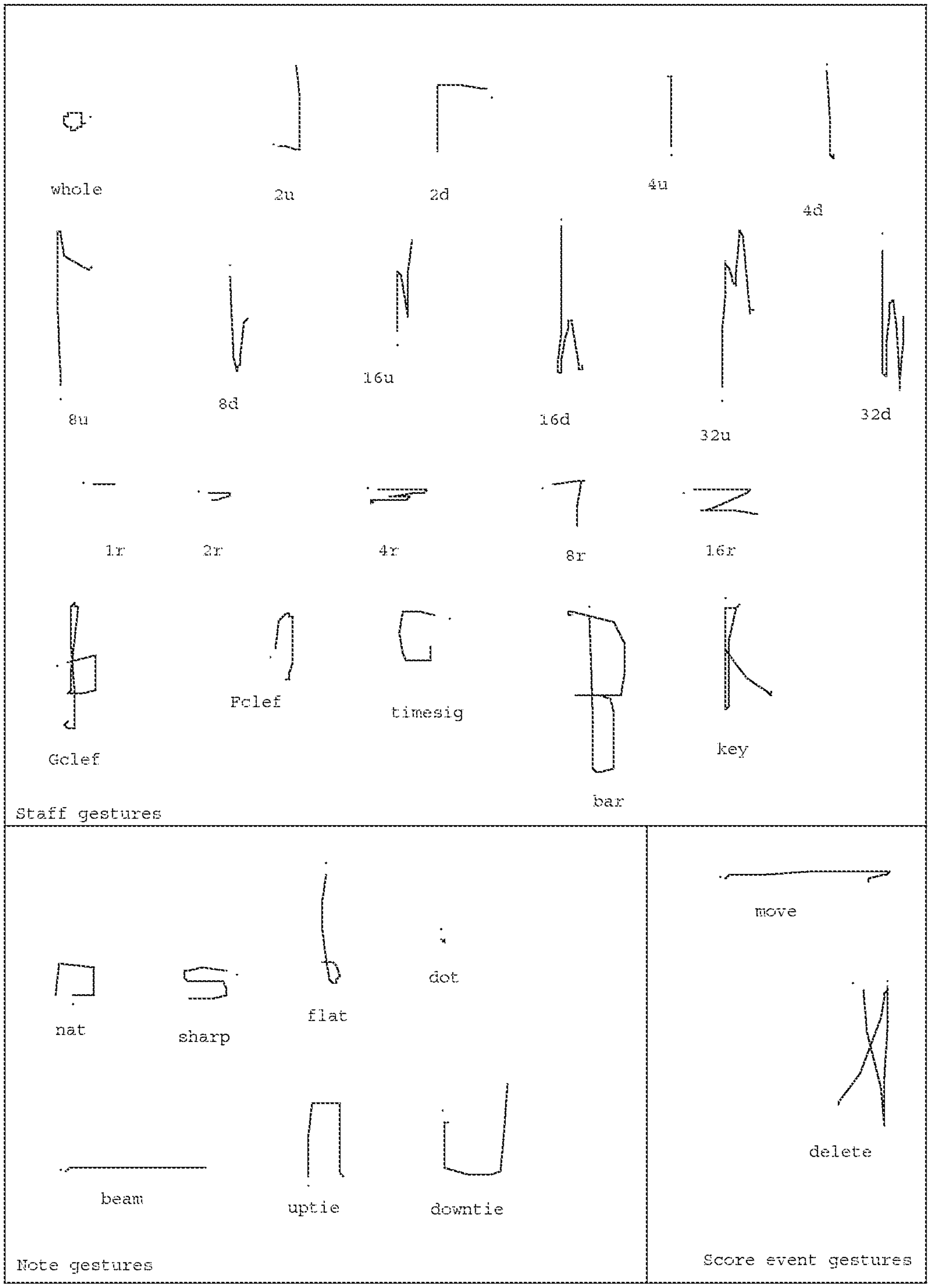


Figure 8.5: GSCORE gestures

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

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

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

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

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

Figure 8.6 shows an example session with GSCORE.

8.2.2 Design and implementation

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

Generally useful views

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

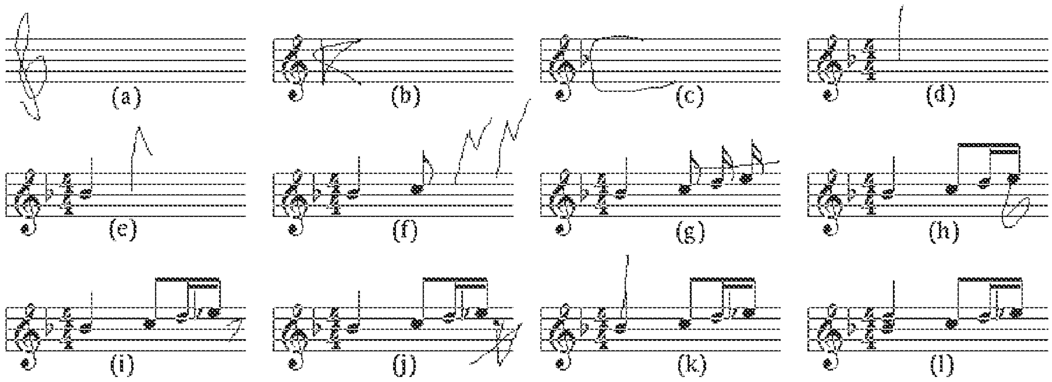


Figure 8.6: A GSCORE session

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

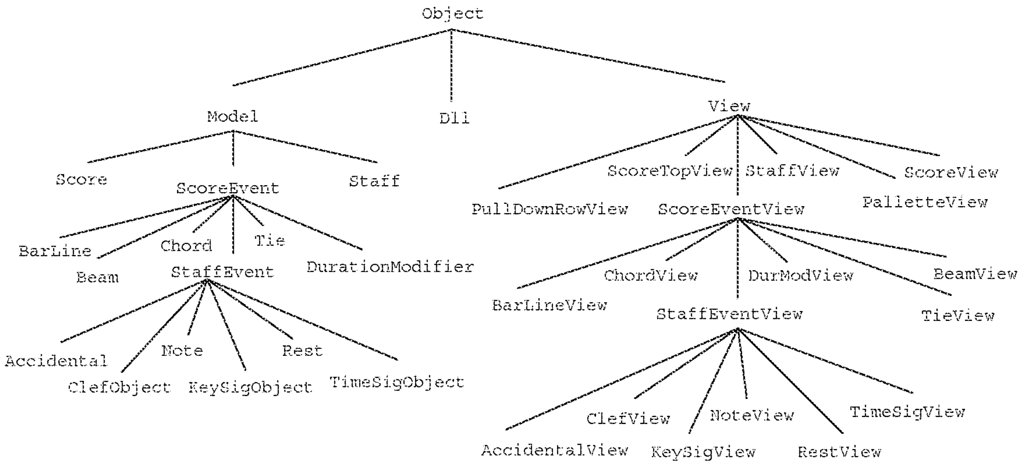


Figure 8.7: GSCORE's class hierarchy

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

GSCORE Models

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

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

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

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

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

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

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

GSCORE Views

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

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

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

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

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

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

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

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

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

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

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

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

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

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

GSCORE's gesture semantics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.3 MDP

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

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

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

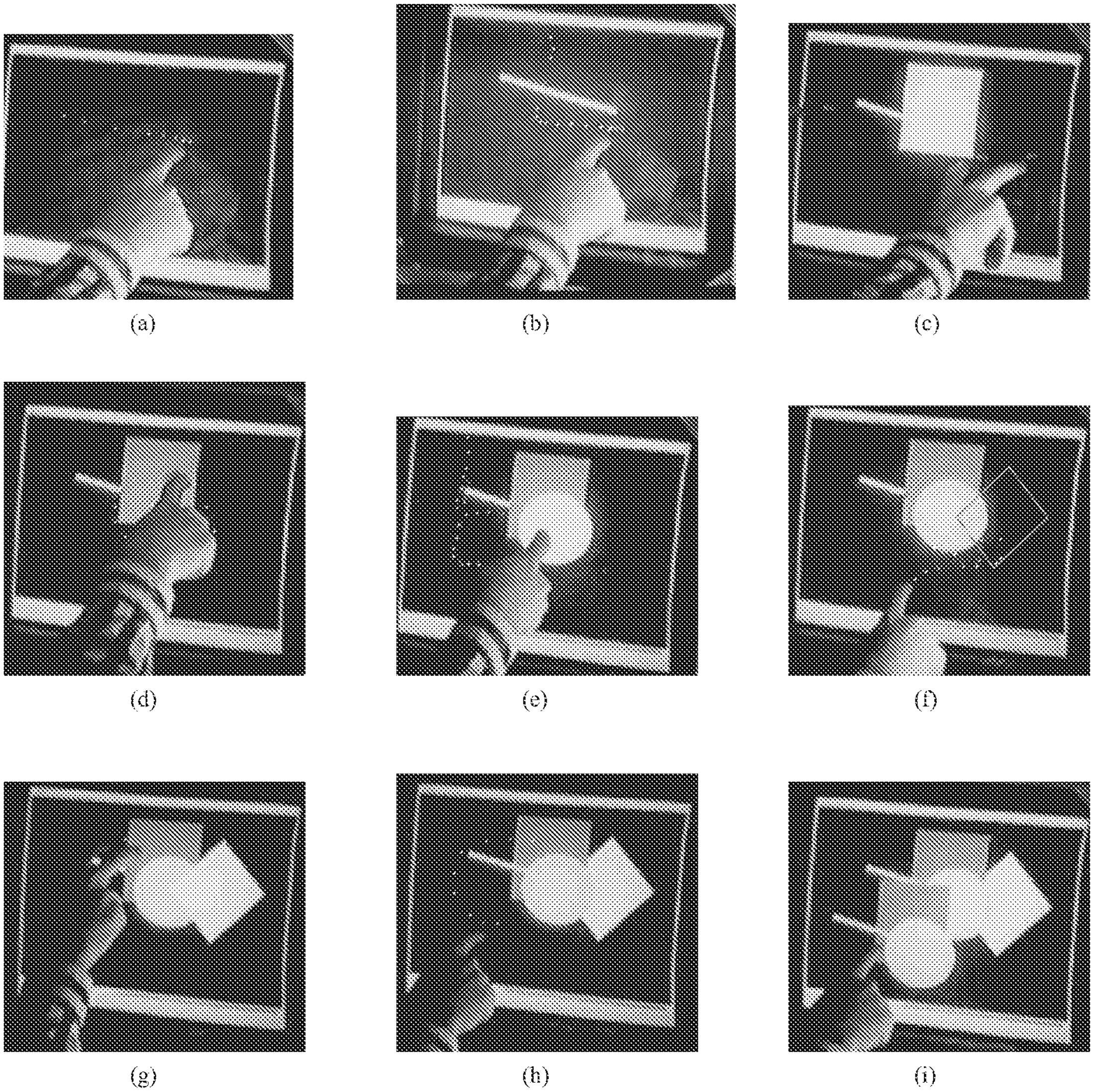
8.3.1 Internals

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

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

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

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



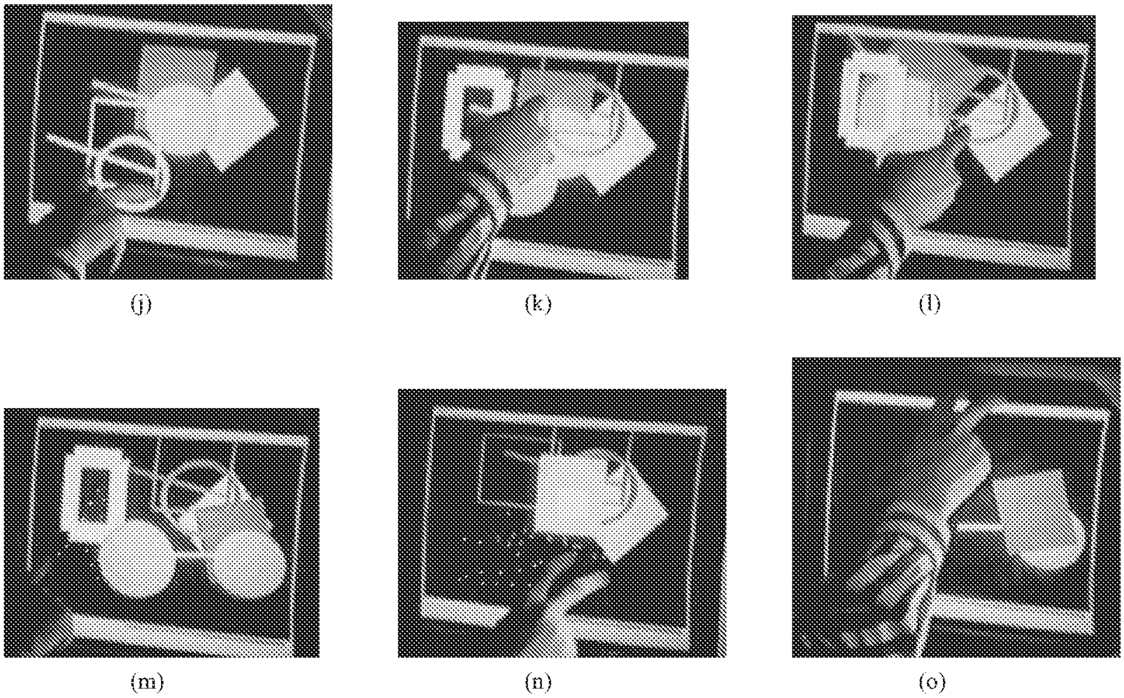
(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figure 8.8: An example MDP session

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



(j) (k) (l)

(m) (n) (o)

Fig 8.8 (continued)

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

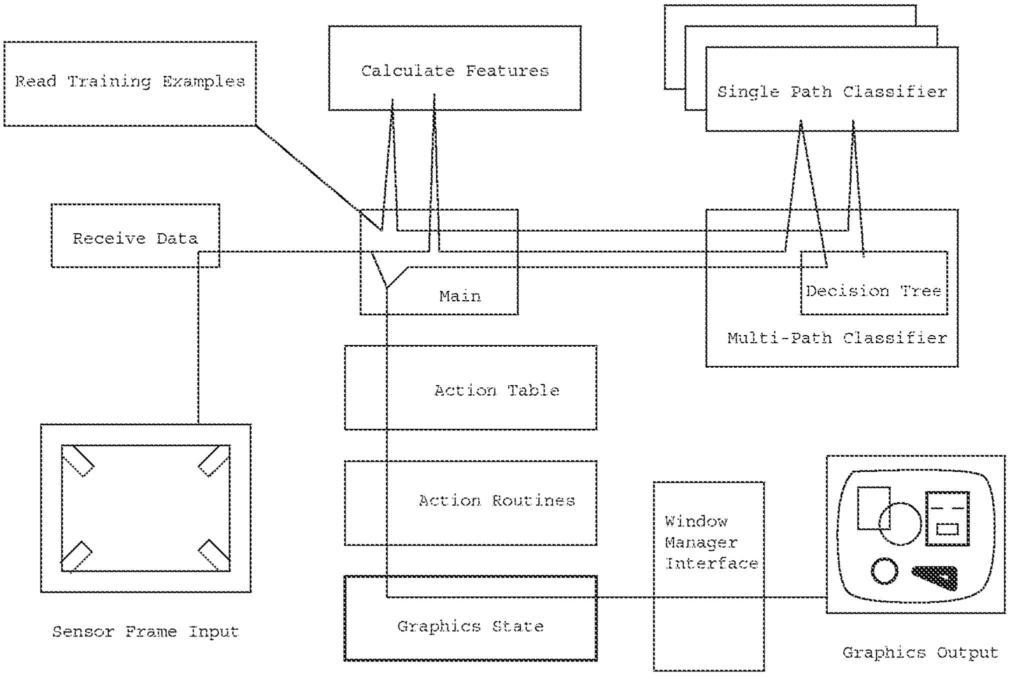


Figure 8.9: MDP internal structure

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    typedef enum { Nothing, Line, Rect,
                   Circle, SetOfObjects } Type;
    typedef struct { /* ... */ } *Element;
    typedef struct { /* ... */ } *Trans;
```

Assume the following declarations for expositional purposes:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.3.2 MDP gestures and their semantics

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

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

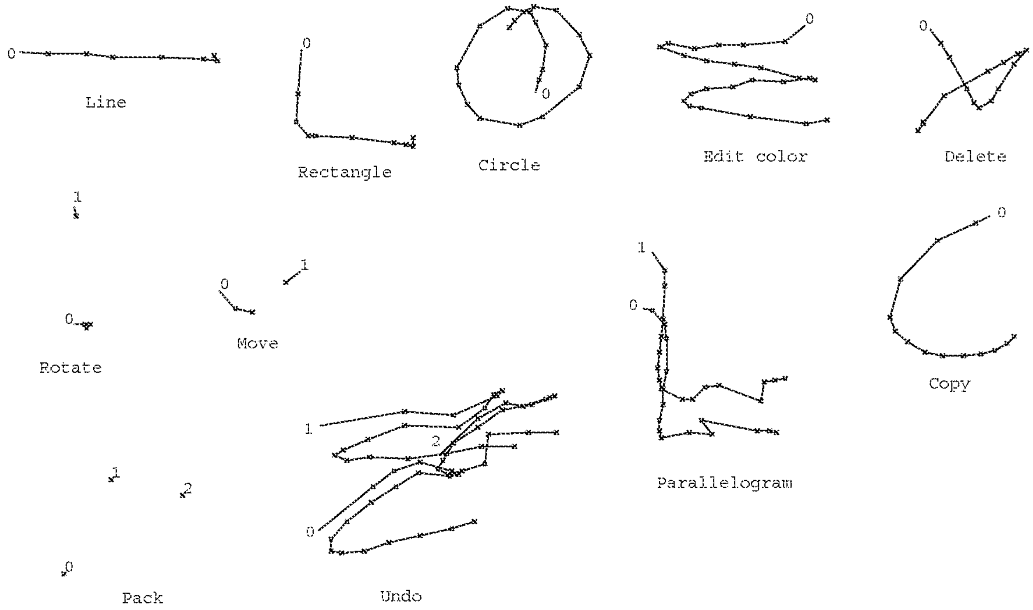


Figure 8.10: MDP gestures

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
ACTION(_PACKrecog)
END_ACTION

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

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

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

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

8.3.3 Discussion

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

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

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

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

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

8.4 Conclusion

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

Chapter 9

Evaluation

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

9.1 Basic single-path recognition

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

9.1.1 Recognition Rate

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

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

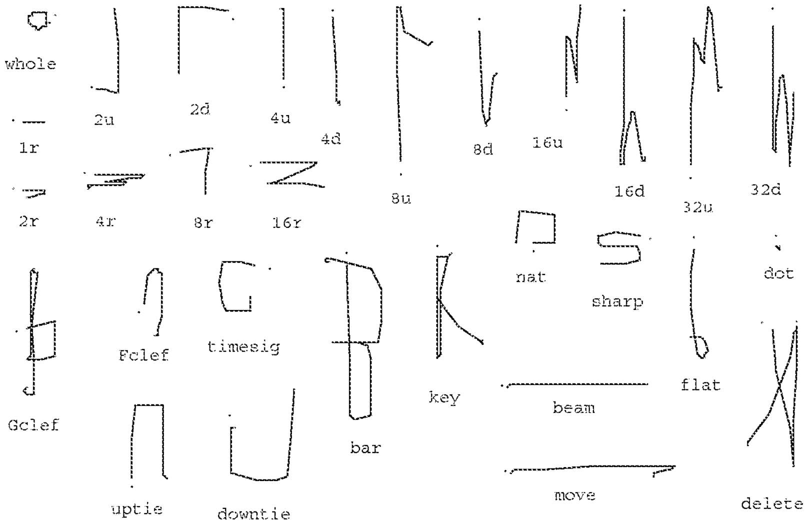


Figure 9.1: GSCORE gesture classes used for evaluation

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

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

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

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

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

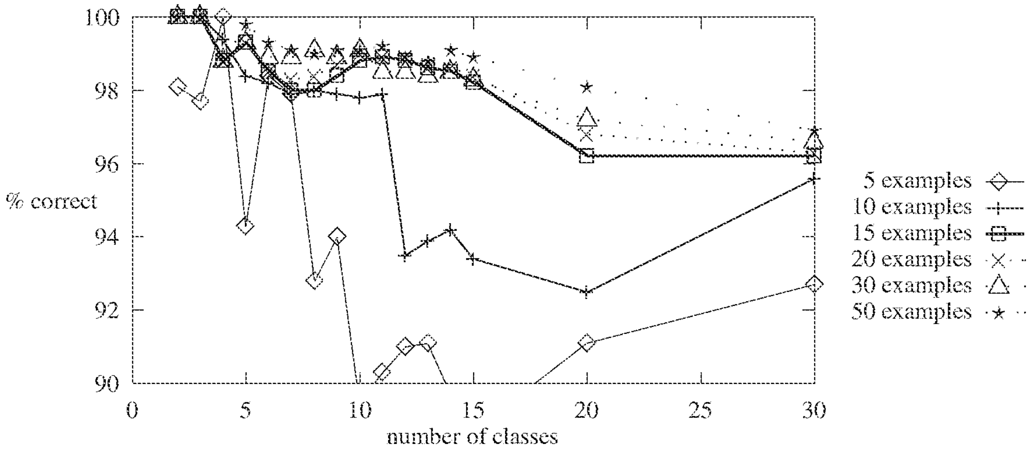


Figure 9.2: Recognition rate vs. number of classes

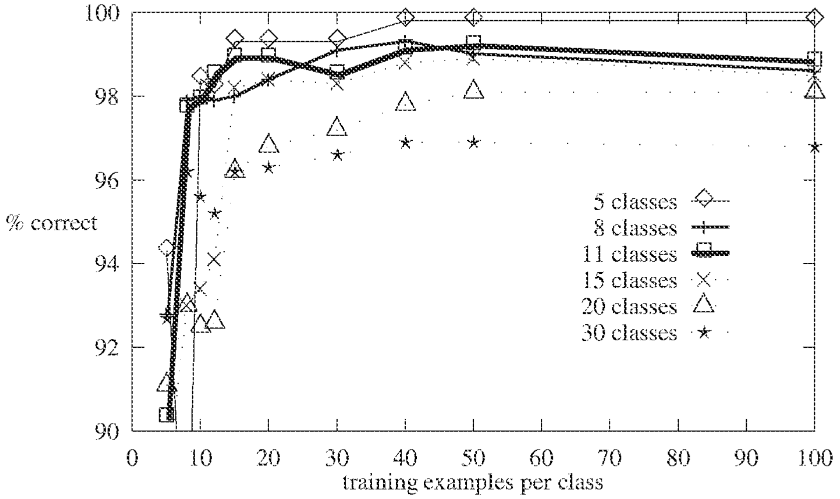


Figure 9.3: Recognition rate vs. training set size

C.

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

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

Analysis of errors

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

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

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

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

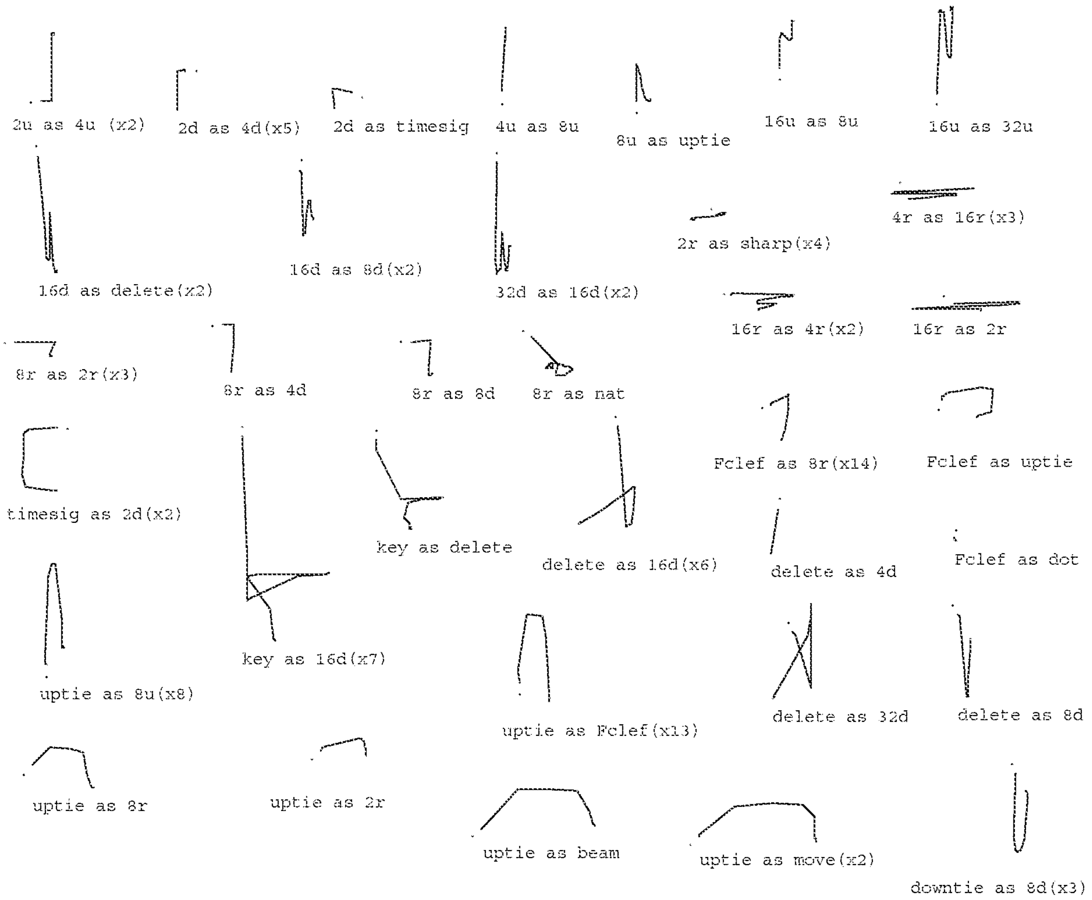


Figure 9.4: Misclassified GSCORE gestures

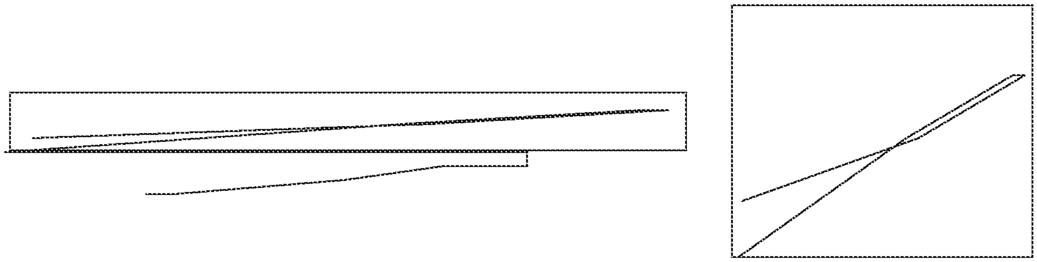


Figure 9.5: A looped corner

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

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

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

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

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

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

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

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

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

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

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

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

9.1.2 Rejection parameters

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

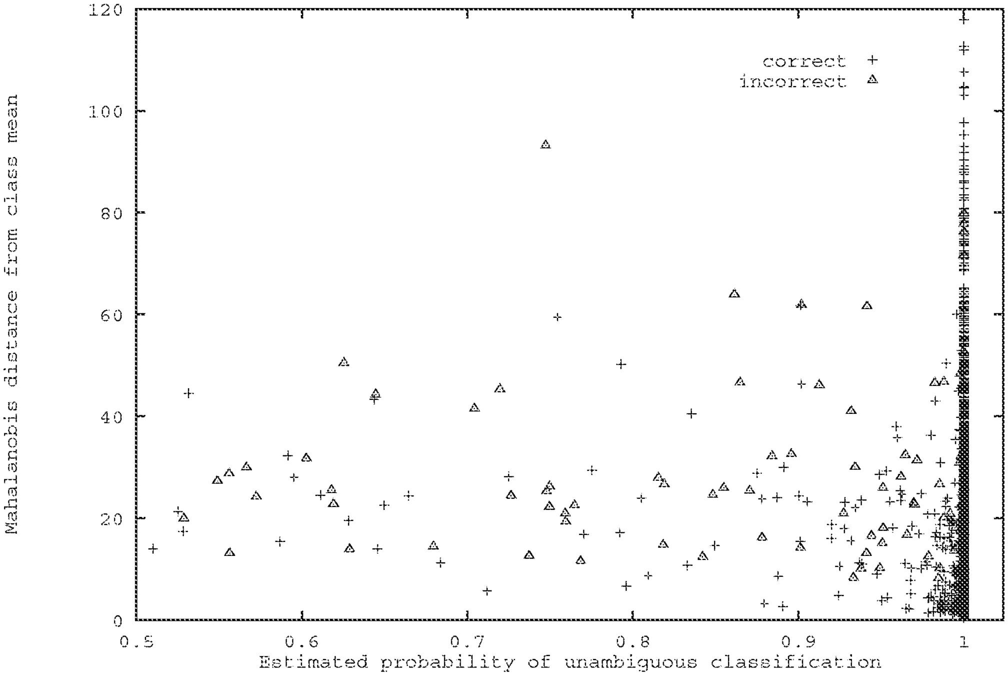


Figure 9.6: Rejection parameters

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

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

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

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

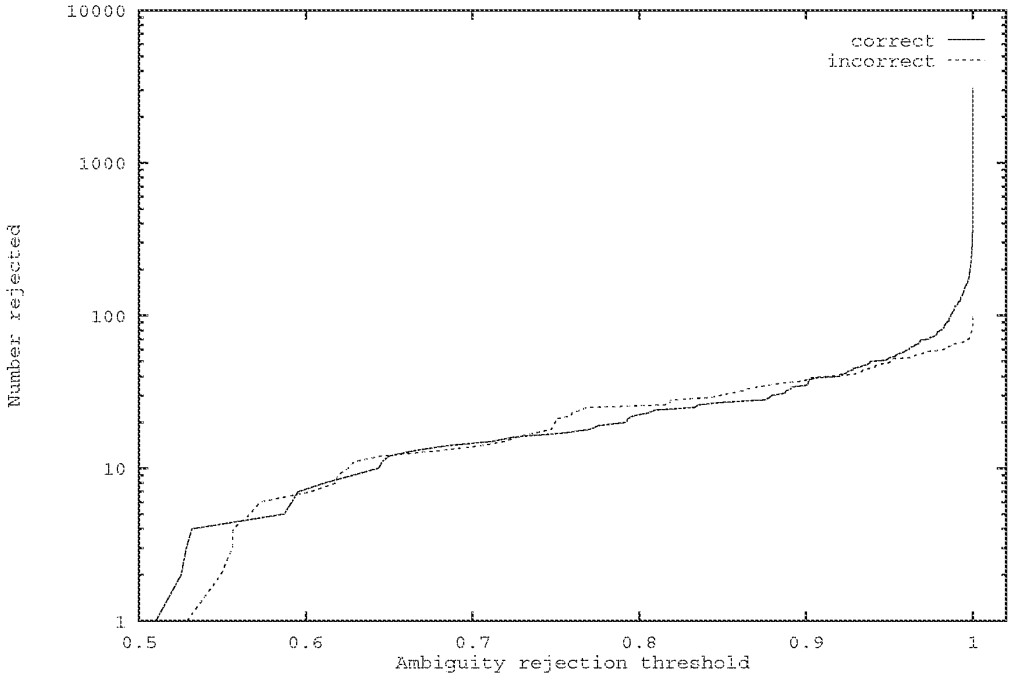


Figure 9.7: Counting correct and incorrect rejections

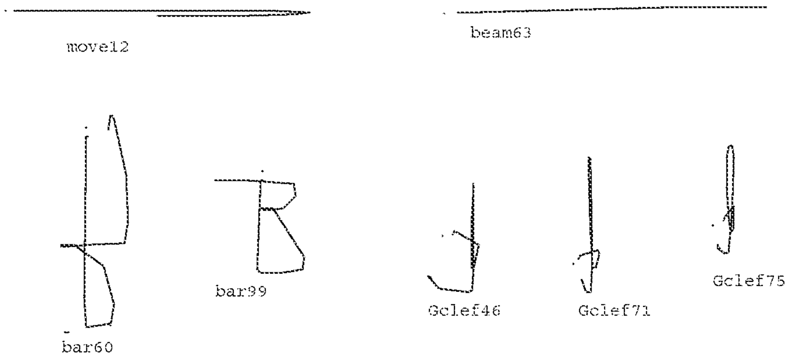


Figure 9.8: Correctly classified gestures with $d^2 \geq 90$

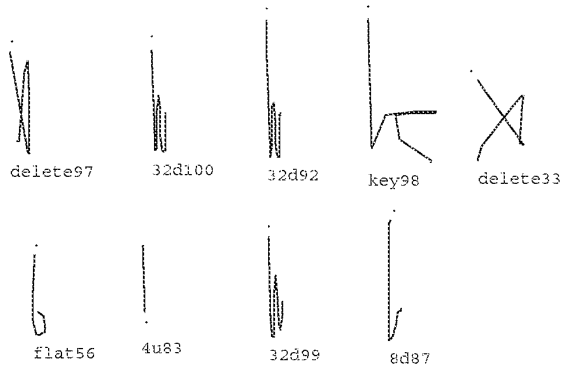


Figure 9.9: Correctly classified gestures with $\tilde{P} \leq .95$

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

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

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

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

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

9.1.3 Coverage

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

9.1.4 Varying orientation and size

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

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

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

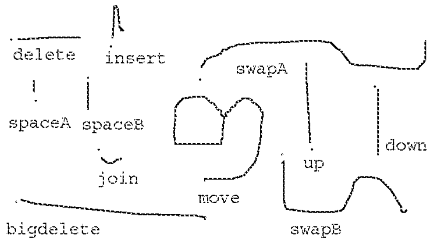
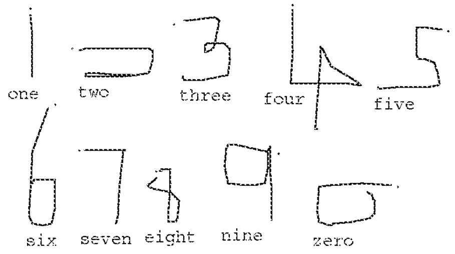
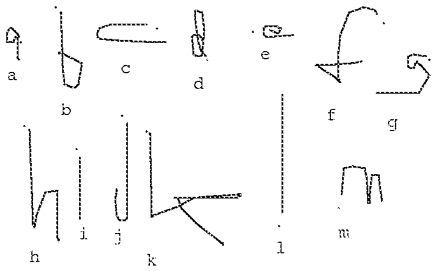
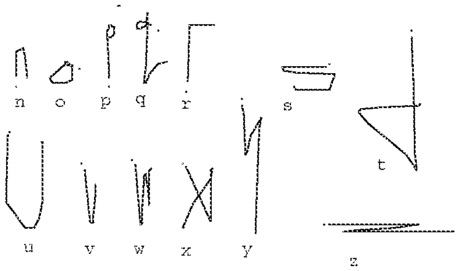
Set Name	Gesture Classes	Number of Classes	Recognition Rate
Coleman	 delete insert swapA spaceA spaceB down up join move bigdelete swapB	11	100.0%
Digits	 one two three four five six seven eight nine zero	10	98.5%
Let:a-m	 a c d e b f g h i j k l m	13	99.2%
Let:n-z	 n o p q r s t u v w x y z	13	98.4%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

Figure 9.10: Recognition rates for various gesture sets

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

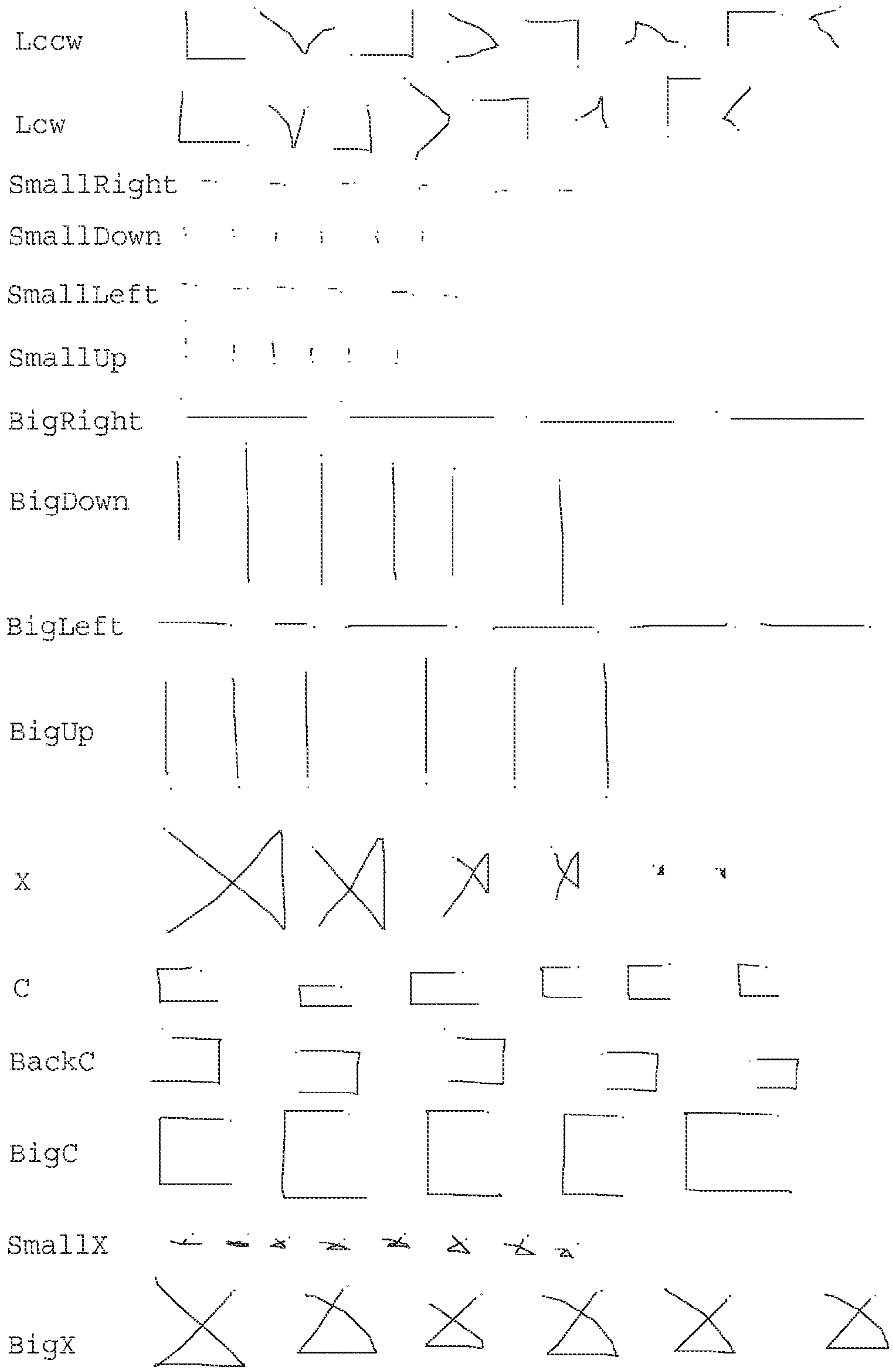


Figure 9.11: Classes used to study variable size and orientation

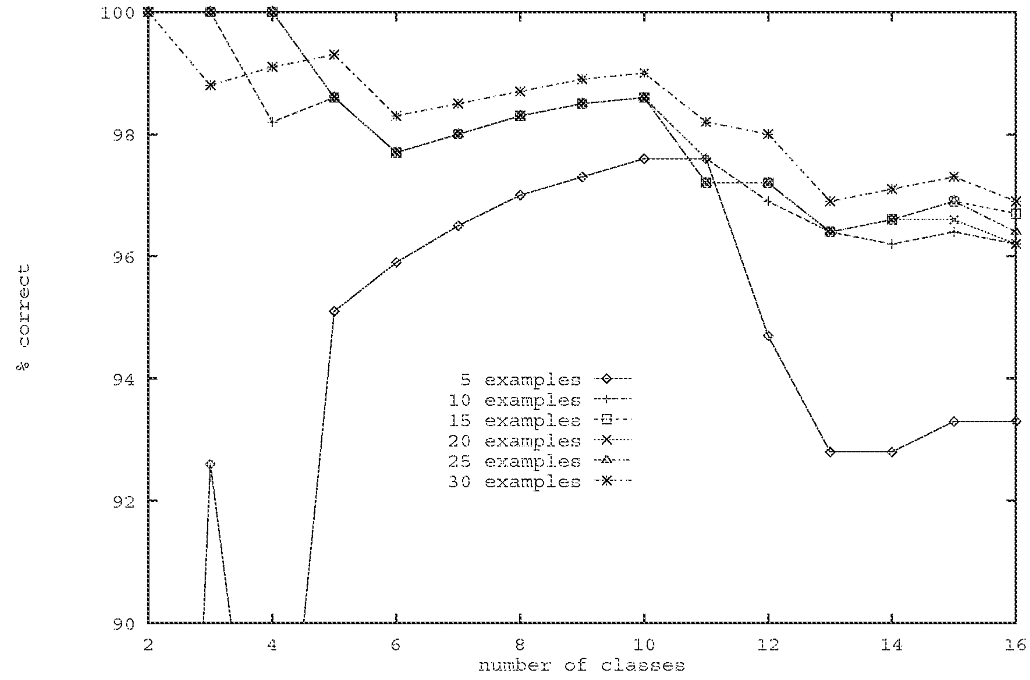


Figure 9.12: Recognition rate for set containing classes that vary

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

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

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

9.1.5 Interuser variability

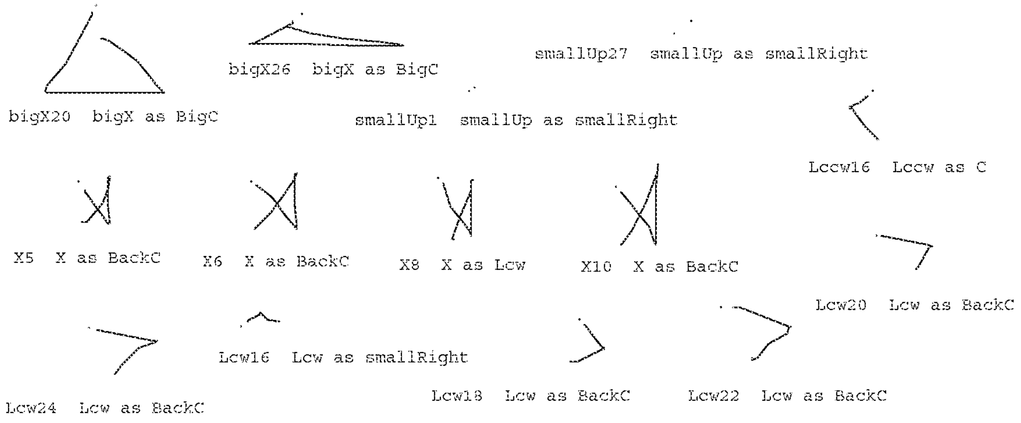
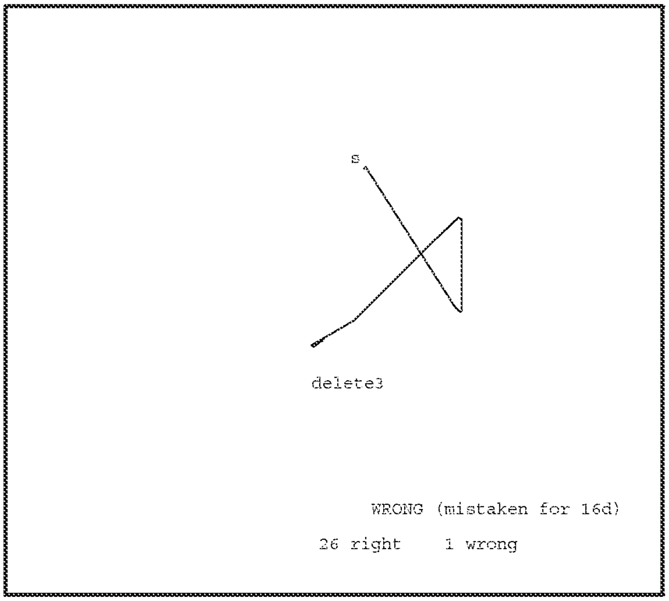


Figure 9.13: Mistakes in the variable class test



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

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

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

Setup

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

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

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

Procedure

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

Results

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

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

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

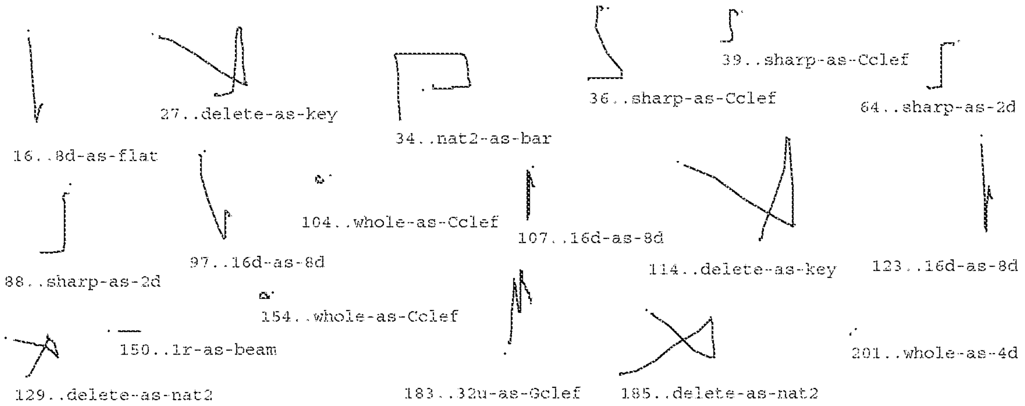


Figure 9.15: PV's misclassified gestures (author's set)

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

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

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

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

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

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

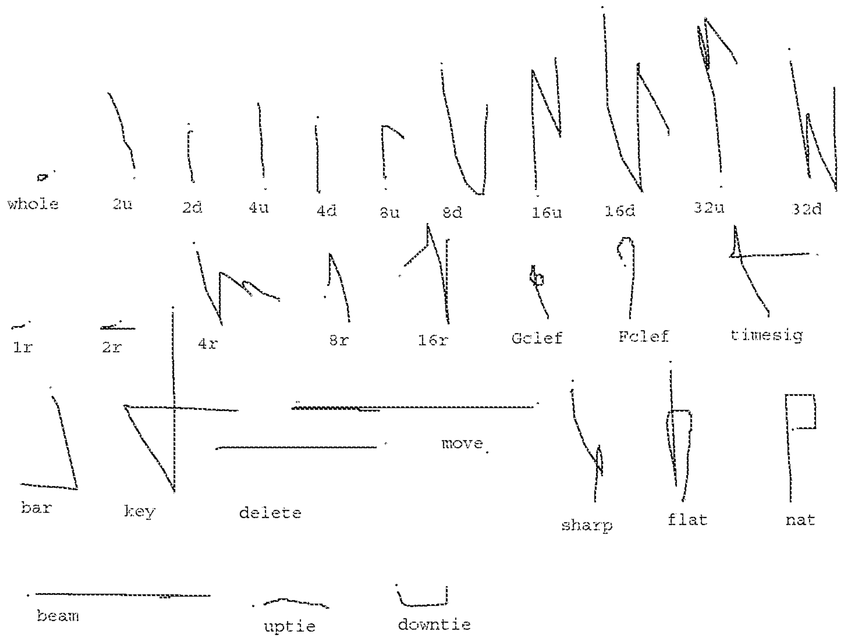


Figure 9.16: PV's gesture set

Conclusion

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

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

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

9.1.6 Recognition Speed

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

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

Feature calculation

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

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

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

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

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

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

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

Processor	Time(sec)	Relative Speed
MicroVAX II	227.95	0.76
VAX 11/780	172.20	1.0
MicroVAX III	60.97	2.8
PMAX-3100	11.30	15

Table 9.1: Speed of various computers used for testing

Processor	milliseconds per call		
	FvAddPoint	FvCalc	CalcFeatures
MicroVAX II	0.22	0.34	3.9
MicroVAX III	0.074	0.13	1.3
PMAX-3100	0.029	0.040	0.44

Table 9.2: Speed of feature calculation

```
        return FvCalc(fv);
    }
```

To obtain the timings, the testing set of Section 9.1.1 was read into memory, and then each gesture was passed to `CalcFeatures`. Three processors were used: the DEC MicroVAX II that was used for the majority of the work reported in this dissertation, a DEC MicroVAX III, and a DEC PMAX-3100 (to get an idea of the performance on a more modern system). The UNIX profiling tool was used to obtain the times. In all cases, the times are virtual times, *i.e.* the time spent executing the program by the processor. All tests were run on unloaded systems, and the real times were never more than 10% more than the virtual times.

Before timing any code related to gesture recognition, the following code fragment (compiled with "cc -O") was timed on a number of processors, MicroVAX II, VAX 11/780, MicroVAX III, and PMAX-3100, in order to compare the speed of the processors used in the following tests to that of a VAX 11/780:

```
register int  i, n = 1000000;
double s, a[15], b[15];
for(i = 0; i < 15; i++) a[i] = i, b[i] = i*i;
do {
    s = 0.0;
    for(i = 0; i < 15; i++) s += a[i] * b[i];
} while(--n);
```

The times for the above fragment shown in table 9.1.

Note that on this code fragment the PMAX-3100 runs about 20 times faster than the MicroVAX II. On more typical code, it usually runs only 10-15 times faster.

The testing set averaged 13.4 points per gesture. The timings for the routines that calculate features are shown in table 9.2.

The cost per mouse point to incrementally process a mouse point is a small fraction of a millisecond, even on the slowest processor. Since mouse points typically come no faster than 40

Processor	Computation time (milliseconds)				
	$v^{\hat{c}}$	max $v^{\hat{c}}$	$\tilde{P}$	d^2	total
	(one class)	(30 classes)			
MicroVAX II	0.27	8.0	0.8	3.7	12.6
MicroVAX III	0.074	2.2	0.3	1.1	3.6
PMAX-3100	0.022	0.66	0.01	0.26	0.99

Table 9.3: Speed of Classification

per second, only a small fraction of the processor is consumed incrementally calculating the feature vector. Indeed, substantially more of the processor is consumed communicating with the window manager to receive the mouse point and perform the inking.

Classification

Once the feature vector is calculated it must be classified. This involves computing a linear evaluation function $v^{\hat{c}}$ on F features ($F = 13$) for each of C classes. If the rejection parameters are desired, it takes an additional $O(C)$ work to estimate the ambiguity $\tilde{P}$ and $O(F^2)$ work to estimate the Mahalanobis distance d^2. The computation times for each of these is shown in table 9.3.

To get these times, four runs were made. Every gesture in the testing set was classified in every run. The first run did not calculate either rejection parameter. The average time to classify a gesture as one of thirty classes is reported the $\mathit{maxv}^{\hat{c}}$ column; the $v^{\hat{c}}$ column is computed as $\frac{1}{30}$ of that time. (The $v^{\hat{c}}$ column thus gives the time to compute the evaluation function for a single class; multiply this by the number of classes to estimate the classification time of a particular classifier.) The second run computed $\tilde{P}$ after each classification; the difference between that time and the max $v^{\hat{c}}$ time is reported in the $\tilde{P}$ column. The third run computed d^2 and is reported similarly. The fourth run computed both $\tilde{P}$ and d^2; the average time per gesture is reported in the "total" column.

For a 30-class discrimination with both rejection parameters being used, after the last mouse point of a gesture is entered it takes a MicroVAX II 13 milliseconds to finish calculating the feature vector (`FvCalc`) and then classify it. This is acceptable, albeit not fantastic, performance. If the end of the gesture is indicated by no mouse motion for a timeout interval, the classification can begin before the timeout interval expires, and the result be ignored if the user moves the mouse before the interval is up.

Currently, all arithmetic is done using double precision floating point numbers. There is no conceptual reason that the evaluation functions could not be computed using integer arithmetic, after suitably rescaling the features so as not too lose much precision. The resulting classifier would then run much faster (on most processors). This has not been tried in the present work.

If eager recognition is running, classification must occur at every mouse point, and the number of classes is $2C$. This puts a ceiling on the number of the classes that the eager recognizer can discriminate between in real-time. On a MicroVAX II, the cost per mouse point includes `FvAddPoint` (0.22 msec) plus `FvCalc` (0.34 msec) plus the per class evaluation of $2C$ classes, $0.54C$. If mouse points come at a maximum rate of one every 25 milliseconds, $C = 45$ classes would consume the entire processor. Practically, since there is other work to do (*e.g.* inking), $C = 20$ is probably the maximum that can be reasonably expected from an eager recognizer on a MicroVAX II. On today's processors, instead of computation time, the limiting factor will be the lower recognition rate when given a large number of classes.

One approach tried to increase the number of classes in eager recognizers was to use only a subset of the features. While this improved the response time of the system, the performance degraded significantly, so the idea was abandoned. There is no point getting the wrong answer quickly.

9.1.7 Training Time

The stated goal of the thesis work is to provide tools that allow user interface designers to experiment with gesture-based systems. One factor impacting on the usability of such tools is the amount of time it takes for gesture recognizers to retrain themselves after changes have been made to the training examples. In almost all trainable character recognizers, deleting even a single training example requires that the training be redone from scratch. For some technologies, notably neural networks, this retraining may take minutes or even hours. Such a system would not be conducive to experimenting with different gesture sets.

By contrast, statistical classifiers of the kind described in Chapter 3 can be trained very rapidly. Training the classifier from scratch requires $O(EF)$ to compute the mean feature vectors, $O(EF^2)$ time to calculate the per-class covariance matrices, $O(CF^2)$ to average them, $O(F^3)$ to invert the average, and $O(CF^2)$ to compute the weights used in the evaluation functions. If the average covariance matrix is singular, an $O(F^4)$ algorithm is run to deal with the problem.

Often, a fair amount of work can be reused in retraining after a change to some training examples. Adding or deleting an example of a class requires $O(F)$ work to incrementally update its per-class class mean vector, and $O(F^2)$ work to incrementally update its per-class covariance matrix [137]. Retraining then involves repeating the steps starting from computing the average covariance matrix. Thus, for retraining, the dependency on E, the total number of examples, is eliminated. The retraining time is instead a function of the number of examples added or deleted.

The Objective C implementation does not attempt to incrementally update the per-class covariance matrix when an example is added. Instead, only the averages are kept incrementally, and the per-class covariance matrix is recomputed from scratch. This involves $O(E^{\hat{c}}F^2)$ work for each class c changed, where $E^{\hat{c}}$ is the number of training examples for class c. This results in worse performance when a small number of examples are changed, but better performance when all the examples of a class are deleted and a new set entered. The latter operation is common when experimenting with gesture-based systems.

The author has implemented both C and Objective C versions of the single path classifier. Besides maintaining the per-class covariance matrices incrementally, the C version differs in that it does not store the list of examples that have been used to train it. (It is not necessary to store the list to add and remove examples, since the mean vector and covariance matrix are updated incrementally.) It is thus more efficient since it does not need to maintain the lists of examples. (Objective C's `Set` class, implemented via hashing, is used to maintain the lists in that version.) It also does not have the overhead of separate objects for examples, classes and classifiers that the Objective C version has (see Section 7.5).

Processor	Time (milliseconds per call)			
	`sAddExample`	`sRemoveExample`	`sDoneAdding` (10 classes)	`sDoneAdding` (30 classes)
MicroVAX II	3.7	3.8	130	234
MicroVAX III	0.90	0.90	43	78
PMAX-3100	0.024	0.026	14	22

Table 9.4: Speed of classifier training

Since only the C version could be ported to the PMAX-3100, it was used for the timings. (C versions of the feature computation and gesture recognition were used for the timings above; however in these cases the Objective C methods are straightforward translations of their corresponding C functions. In some cases, the methods merely call the corresponding C function.) The following C functions encapsulate the process of training a classifier:

`sClassifier sNewClassifier()` allocates and returns a handle to a new classifier. Initially it has no classes and no examples. The "s" at the beginning of the type and function names refers to "single-path"; there are corresponding types and functions for the multi-path classifiers.

`sAddExample(sClassifier sc, char *classname, Vector e)` adds the training example (feature vector `e`) to the named class `classname` in the passed classifier. The class is created if it has not been seen before. Linear search is used to find the class name; however, it is optimized for successive calls with the same name. The `sAddExample` function incrementally maintains the per-class mean vectors and covariance matrices.

`sRemoveExample(sClassifier sc, char *classname, Vector e)` removes example `e`, assumed to have been added earlier, from the named class. The per-class mean vector and covariance matrix are incrementally updated.

`sDoneAdding(sClassifier sc)` trains the classifier on its current set of examples. It computes the average covariance matrix, inverts it (fixing it if singular), and computes the weights.

`sClass sClassify(sClassifier sc, Vector e, double *p, *d2)` actually performs the classification of `e`. If `p` is non-NULL the probability of ambiguity is estimated; if `d2` is non-NULL the estimated Mahalanobis distance of `e` to its computed class is returned. This is the function timed in the previous section.

The functions were exercised first by adding every example in the training set, training the classifier, and then looping, removing and then re-adding 10 consecutive examples before retraining. No singular covariance matrix was encounted, due to the large number of examples. Table 9.4 shows the performance of the various routines.

Even on a MicroVAX II, training a 30 class classifier once all the examples have been entered takes less than a quarter second. Thus GRANDMA is able to produce a classifier immediately the first time a gesture is made over a set of views whose combined gesture set has not been encountered before (see Sections 7.2.2 and 7.4). The user has to wait, but does not have to wait long.

9.2 Eager recognition

This section evaluates the effectiveness of the eager recognition algorithm on several single-stroke gesture sets. Recall that eager recognition is the recognition of a gesture while it is being made, without any explicit indication of the end of the gesture. Ideally, the eager recognizer classifies a gesture as soon as enough of it has been seen to do so unambiguously (see Chapter 4).

In order to determine how well the eager recognition algorithm works, an eager recognizer was created to classify the eight gestures classes shown in 9.17. Each class named for the direction of its two segments, e.g. ur means "up, right." Each of these gestures is ambiguous along its initial segment, and becomes unambiguous once the corner is turned and the second segment begun.

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

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

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

Figure 9.18 shows the performance of the eager recognizer on GDP gestures. The eager recognizer was trained with 10 examples of each of 11 gesture classes, and tested on 30 examples of each class, five of which are shown in the figure. The GDP gesture set was slightly altered to increase eagerness: the group gesture was trained clockwise because when it was counterclockwise it prevented the copy gesture from ever being eagerly recognized. For the GDP gestures, the full classifier had a 99.7% correct recognition rate as compared with 93.5% for the eager recognizer. On the average 60.5% of each gesture was examined by the eager recognizer before classification occurred. For this set no attempt was made to determine the minimum average gesture percentage that needed to be seen for unambiguous classification.

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

Computationally, eager recognition is quite tractable on modest hardware. A fixed amount of

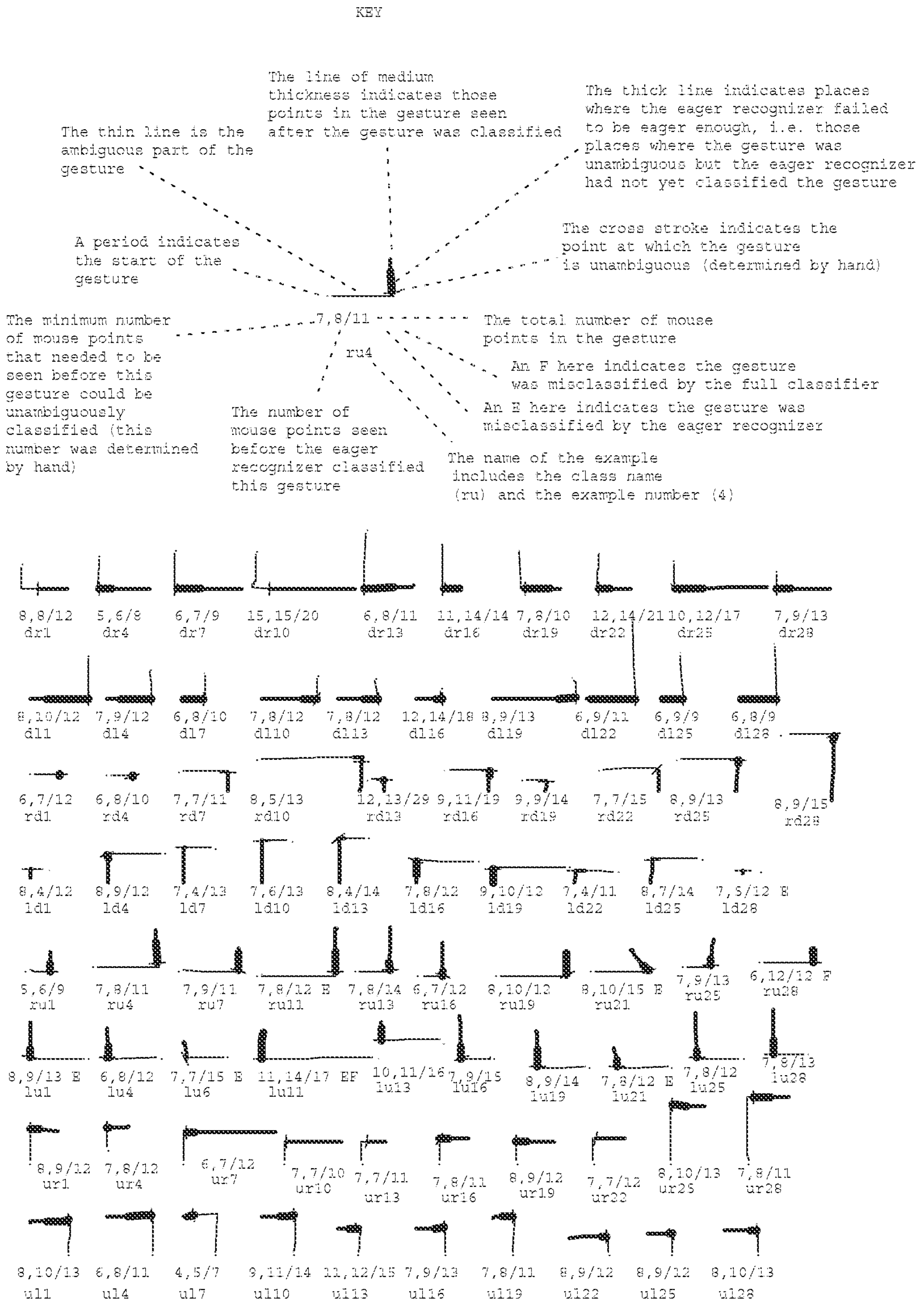

KEY
The line of medium thickness indicates those points in the gesture seen after the gesture was classified
The thick line indicates places where the eager recognizer failed to be eager enough, i.e. those places where the gesture was unambiguous but the eager recognizer had not yet classified the gesture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

Figure 9.17: The performance of the eager recognizer on easily understood data

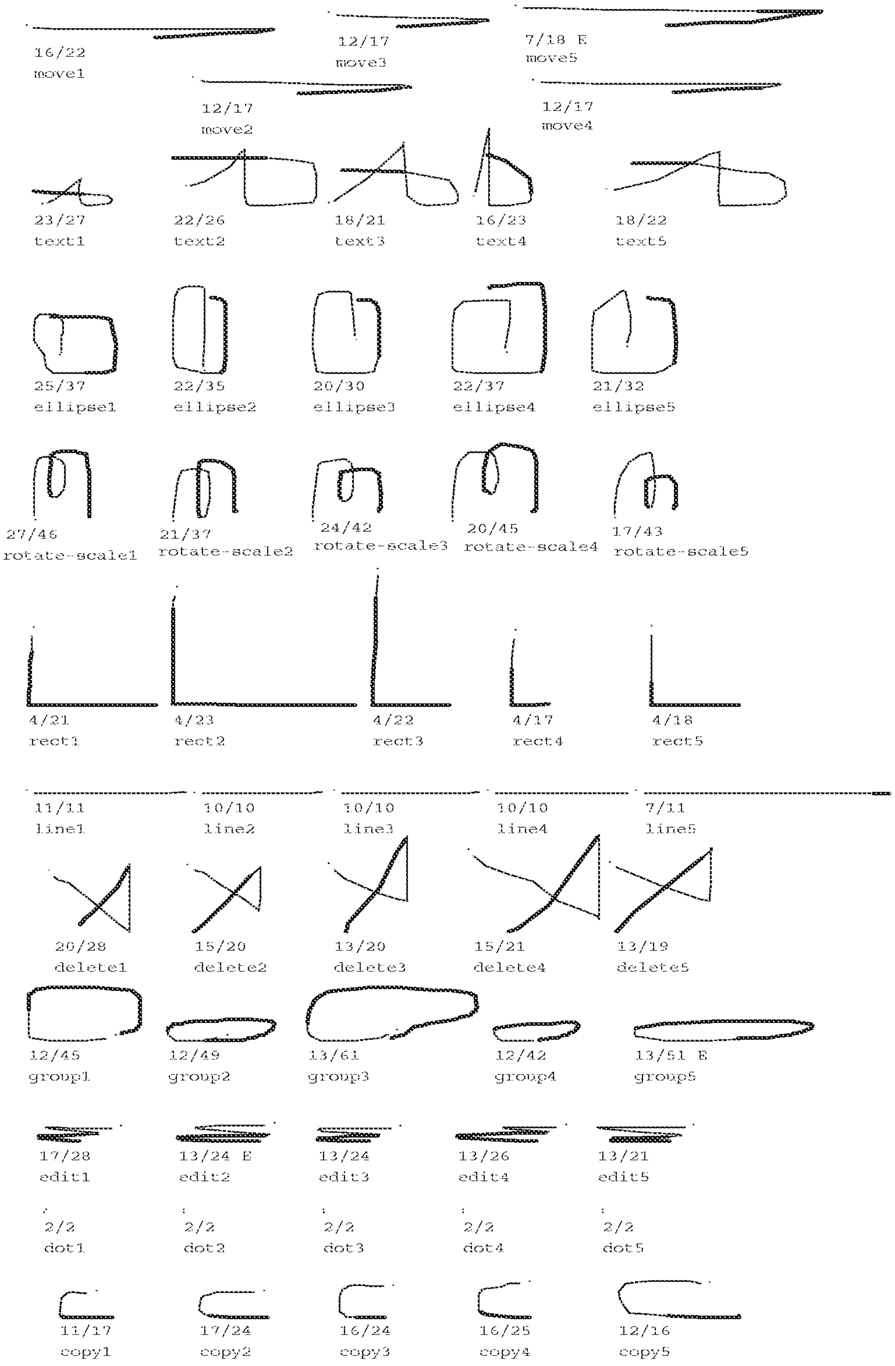


Figure 9.18: The performance of the eager recognizer on GDP gestures

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

computation needs to occur on each mouse point: first the feature vector must be updated (taking 0.5 msec on a DEC MicroVAX II), and then the vector must be classified by the AUC (taking 0.27 msec per class, or 6 msec in the case of GDP).

9.3 Multi-finger recognition

Multi-finger gestural input is a significant innovation of this work. Unfortunately, circumstances have conspired to make the evaluation of multi-finger recognition both impossible and irrelevant. The Sensor Frame is the only input device upon which the multi-finger recognition algorithm was tested. Unfortunately, there is only one functioning Sensor Frame in existence, and that was damaged sometime after the multi-finger recognition was running, but before formal testing could begin. (Fortunately, a videotape of MDP in action was made while the Sensor Frame was working.) No progress was made in repairing the Sensor Frame for over a year; testing was thus impossible. Eventually the Sensor Frame was repaired, but Sensor Frame, Inc. went out of business shortly afterward, making any detailed evaluation irrelevant. The owner of the Sensor Frame has left the country, taking the device with him.

An informal estimate of the multi-finger recognition accuracy may be estimated from ten minutes of videotape of the author using MDP. This version of MDP uses the path sorting multi-finger recognition algorithm (Section 5.2). As shown in figure 8.10, MDP recognizes 11 gestures (6 one finger gestures, 3 two finger gestures, and 2 three finger gestures). In the videotape, the author made 30 gestures, 2 of which appear to have been misclassified, and one of which was rejected, resulting in a correct recognition rate of 90%. The processing time appears to be negligible.

All three misclassifications are the result of the Sensor Frame seeing more fingers in the gesture than were intended. This was due to knuckles of fingers curled up (so as not to be used in the gesture) accidentally penetrating the sensing plane and being counted as additional fingers. As there are distinct classifiers for single finger, two finger, and three finger gestures, an incorrect number of fingers inevitably leads to a misclassification. While it is possible to imagine methods for dealing with such errors during recognition, the main cause of this problem is the ergonomics of the Sensor Frame.

For the small gesture set examined, the recognition rate is 100% once the errors due to "extra fingers" are eliminated. This is to be expected, given the small number of gestures for each number of fingers. It is expected that the multi-path classifier operating on one finger gestures would perform about as well as the single-path classifier, as the algorithms are essentially identical. The single-path classifier, when given only six classes to discriminate among, has been shown (on mouse data) to perform at 100% in almost all cases. When operating on two finger gestures, it is expected that the performance of the recognition algorithm would be similar to that of the single-path classifier on *twice* the number of classes. Actually, it is possible that some of the paths in the two-finger gestures will be similar to other paths in the set, and be merged into a single class by the training algorithm (Section 5.4). Thus, when the number of unique paths will be less than twice the number of two-finger gesture classes, performance may be expected to improve accordingly. Similarly, the three finger gesture classifier may be expected to perform as well as a single-path classifier the recognizes between one and three times the number of three finger gesture classes, depending on the number of unique paths in the class set.

One more factor to consider is that mouse data tends to be much less noisy than Sensor Frame data. The triangulation by the Sensor Frame is erratic, especially when multiple fingers are being tracked. For example, both horizontal segments of the Parallelogram gesture of figure 8.10 should be straight lines. Until this problem can be solved, it is expected that recognition rates for Sensor Frame gesture sets will suffer.

9.4 GRANDMA

Evaluating GRANDMA is much more subjective than evaluating the low-level recognition rates. GRANDMA may be evaluated on several levels: the effort required to build new interaction techniques, to build new applications, to add gestures to an application, to change an application's gestures, or to use an application to perform a task.

No attempt was made to formally evaluate any of these. In order to get statistically valid results, it would have been necessary to run carefully designed experiments on a number of users, something the author had neither the time, space, inclination, or qualifications to do. Furthermore, the author does not wish to claim that GRANDMA is superior to existing object-oriented toolkits for any particular task. GRANDMA is simply the platform through which some ideas for input processing in object-oriented toolkits were explored. GRANDMA's significance, if any, will be its influence on future toolkits, rather than any more direct results.

Nonetheless, this section informally reports on the author's experience building gesture-based systems with GRANDMA. (No one besides the author tried to program with GRANMDA.) Also, in order to confirm that GRANDMA can be used by someone other than the author, this section also reports on observations of a subject trying to use GSCORE and GRANDMA to due some tasks.

9.4.1 The author's experience with GRANDMA

GRANDMA took approximately seven months to design and develop. It consists of approximately 12000 lines of Objective C code. There are an additional 5000 lines of C code which implement a graphics layer as well as the feature vector calculation. GDP took an additional 2000 lines of Objective C code to implement. GDP was developed at the same time as GRANDMA, as it was the primary application used to test GRANDMA.

Initially, only two GDP gestures were used to test GRANDMA's gesture handler and associated utilities. Once these were working well, it took four days to add the remaining gestures to GDP. Most of this time was spent writing Objective C methods to use in semantic expressions. These were methods that were not needed for the existing direct manipulation interface.

GSCORE consists of 6000 lines of Objective C code. It took six weeks to design and implement GSCORE, including its palette-based interface. Much of this time was spent on the details of representing common music notation, mechanisms for displaying music notation, and producing usable music fonts. The palette, an interaction technique that did not as yet exist in GRANDMA, took about eight hours to implement. It took two weeks to add the gestural interface to GSCORE,

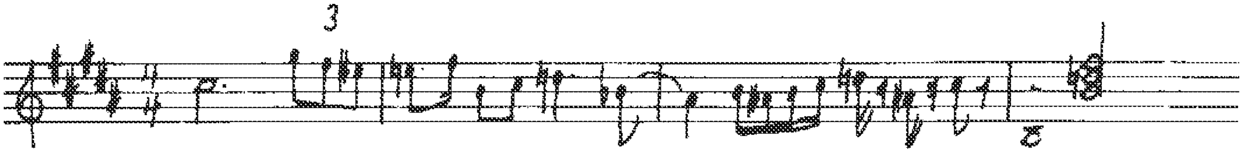


Figure 9.19: PV's task

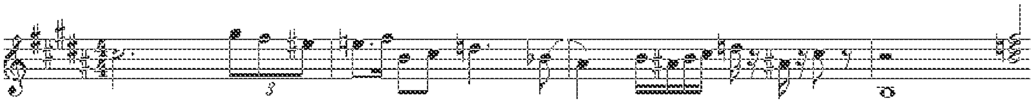


Figure 9.20: PV's result

including writing some additional methods. Much of this time was spent experimenting with different semantics for the gestures.

Section 10.1.3 lists features of GRANDMA that will be important to incorporate into future toolkits that support gestures.

9.4.2 A user uses GSCORE and GRANDMA

This sections informally reports on subject PV's (see Section 9.1.5) attempts to use the GSCORE program.

The task was to enter the music shown in figure 9.19. The music was chosen to exercise many of the GSCORE gestures. PV is an experienced music copyist, and it took him 100 seconds to write out the music as shown, copying it from an earlier attempt.

Using gestures, the author was able to enter the above score in 280 seconds (almost five minutes). He made a total of 53 gestures, four of which did not give the desired results and were immediately undone. Only two of those were misclassifications; the other two were notes gestures where the note was created having the wrong pitch, due to misplacement of the cursor at the start of the gesture. Turning off gestures and using only the palette interface, it took the author 670 seconds (eleven minutes). No mistakes needed to be undone in the latter trial.

PV's first attempt was at using the GSCORE program trained with the author's gestures. PV had already gained experienced with this set of gestures during the study of interuser variation. PV practiced for one half hour with the GSCORE program before attempting the task. The author coached PV during this time, as no other documentation or help was available.

PV took 600 seconds (10 minutes) to complete the task. He made a total of 73 gestures, 16 of which were immediately undone. It appeared to the author, who was silently observing, that each undo was used to recover from a misclassification. Figure 9.20 shows the product of his labor. PV then turned off gestures, and used the palette interface to enter the example. He completed the task in 680 seconds (11 minutes).

PV then entered his own gestures in place of some of the author's. In particular, he substituted his own gestures for the nine classes: delete, move, beam, 1r, 2r, 8r, 16r, keysig, and bar, entering 15 or more examples of each. The total time to do this, including incremental testing of the new gestures and periodic saves to disk, was 25 minutes. PV did not attempt to emulate the author's gestures; instead, he used the forms he had created earlier (see Section 9.1.5).

Once done, PV repeated the experiment. It took him 310 seconds (5 minutes) to enter the music. He made 58 gestures, 4 of which were undone.

PV was interviewed after the tests, and made the following comments: The biggest problem, he stated, was that mouse tracking in the GSCORE program was much more sluggish than in the recorder and tester. (This is accurate, as the time required for GSCORE events to be created and consumed adds significant overhead to the mouse tracking. Much of this is overhead due to GRANDMA.) PV characterized the system as "sluggish." Bad tracking, especially at the start of the gesture, contributed significantly to the number of misclassifications.

PV stated that he thought the system "intuitive" for entering notes. He described the gesture-based interface as "excellent" compared to the palette-based system, but when asked how the gesture-based interface compared to writing on paper, he replied "it sucks." He did not like using the mouse for gesturing, and believed that a stylus and tablet would be much better.

It is again difficult to draw conclusions from an informal study of one user. Did PV's performance improve because he tailored the gestures to his liking, or because he had been practicing? This is unknown. Some things are clear: GRANDMA makes it easy to experiment with new gesture sets, and, in GSCORE, with moderate practice the gesture-based interface improved task performance by a factor of two over the palette-based interface. Whether gesture-based interfaces generally improve task performance over non-gesture-based interfaces is a question that requires *much* further study.

Chapter 10

Conclusion and Future Directions

This chapter summarizes the contributions of this thesis and indicates some directions for future work.

10.1 Contributions

This thesis makes contributions in four areas:

- New interaction techniques
- New recognition-related algorithms
- Integrating gestures into interfaces
- Input in object-oriented toolkits

Each of these will be discussed in turn.

10.1.1 New interactions techniques

A major contributions of this work has been the invention and exploration of three new interaction techniques.

The two-phase single-stroke interaction The two-phase interaction enables gesture and direct manipulation to be integrated in a single interaction that combines the power of each. The first phase is collection, during which the points that make up the gesture are collected. In the simplest case, the end of the collection phase is indicated by a motion timeout, classification occurs, and the second phase, manipulation, is entered. In the manipulation phase, the user moves the mouse to manipulate some parameters in the application. The particular parameters manipulated depend on the classification of the collected gesture. The collection phase is like character entry in handwriting interfaces; the manipulation phase is like a drag interaction in direct-manipulation interfaces. Generally, the operation, operands, and some parameters are determined at the phase transition (when the gesture is recognized), and then the manipulation phase allows additional parameters to be set in the presence of application feedback.

Eager recognition Eager recognition is a modification of the two-phase single-stroke interaction in which the phase transition from gesturing to manipulation occurs as soon as enough of the gesture has been seen so that it may be classified unambiguously. The result is an interaction that combines gesturing and direct manipulation in a single, smooth interaction.

The two-phase multiple-finger interaction Gesture and direct manipulation may be combined for multiple path inputs in a way similar to the two-phase single-stroke interaction. With multiple finger input, opportunities exist for expanding the power of each phase of the interaction. By allowing multiple fingers in the collection phase, the repertoire of possible gestures is greatly increased, and a multiple finger gesture allows many parameters to be specified simultaneously when the gesture is recognized. Similarly, even when only one finger is used for the gesture, additional fingers may be brought in during the manipulation phase. Thus, the two-phase multiple-finger interaction allows a large number of parameters to be specified and interactively manipulated.

10.1.2 Recognition Technology

This thesis discloses five new algorithms of general utility in the construction and use of gesture recognizers.

Automatic generation of single-stroke gesture recognizers from training examples A practical and efficient algorithm for generating gesture recognizers has been developed and tested. In it, a gesture is represented as a vector of real-valued features, and a standard pattern recognition technique is used to generate a linear classifier that discriminates between the vectors of different gesture classes. The training algorithm depends on aggregate statistics of each gesture class, and empirically it has been shown that usually only fifteen examples of each class are needed to produce accurate recognizers. It is simple to incorporate dynamic attributes, such as the average speed of the gesture, into the feature set. The algorithm has been shown to work even when some classes vary in size and orientation while others depend on size or orientation to be recognized. The recognizer size is independent of the number of training examples, and both the recognition and training times have been shown to be small. A features set that is both meaningful and extensible potentially allows the algorithm to be adapted to future input devices and requirements.

Incremental feature calculation The calculation used to generate features from the input points of a gesture is incremental, meaning that it takes constant time to update the features given a new input point. This allows arbitrarily large gestures to be processed with no delay in processing.

Rejection algorithms Two algorithms for rejecting ill-formed gestures have been developed and tested. One estimates the probability of correct classification, enabling input gestures that are ambiguous with respect to a set of gesture classes to be rejected. The other uses a normalized distance metric to determine how close an input gesture is to the typical gesture of the its computed class, allowing outliers to be rejected.

Automatic generation of eager recognizers from training examples An eager recognizer classifies a gesture as soon as it is unambiguous, alleviating the need for the end of the gesture to be explicitly indicated. An algorithm for generating eager recognizers from training examples has been developed and tested. The algorithm produces a two-class classifier which is run on every input point and used to determine if the gesture being entered is unambiguous.

Automatic generation of multi-path gesture recognizers The single-stroke recognition work has been extended so that a number of single-stroke recognizers may be combined into a multi-finger gesture recognizer. The described algorithm produces a multi-path recognizer given training examples. Relative path timing information is considered during the recognition, and global classification is attempted when the individual path classifications do not uniquely determine the class of the multi-path gesture. For dealing with the problems that arise from multi-path input devices that do not *a priori* determine "which path is which," two approaches, path sorting and path clustering, have been explored. The resulting algorithm has been demonstrated using the Sensor Frame as a multi-finger input device.

10.1.3 Integrating gestures into interfaces

A paradigm for integrating gestures into object-oriented interfaces has been developed and demonstrated. The key points are:

A gesture set is associated with a view or view class. Each class of object in the user interface potentially responds to a different set of gestures. Thus, for example, notes respond to a different set of gestures than staves in the GSCORE music editor.

The gesture set is dynamically determined. From the first point of a gesture, the system dynamically determines the set of gestures possible. The first point determines the possible views at which the gesture is directed. For each of these views, inheritance up the class hierarchy determines the set of gestures it handles. These sets are combined, and if need be, a classifier for the resulting union is dynamically created.

The gesture class and attributes map to an application operation, operands, and parameters. Gestures are powerful because they contain additional information beyond the class of the gesture. The attributes of a gesture, such as size, orientation, length, speed, first point, and enclosed area, can all be mapped to parameters (including operands) of application routines. In the two-phase interaction, after the gesture is recognized there is an opportunity to map subsequent input to application parameters in the presence of application feedback.

Gesture handlers may be manipulated at runtime. In order to encourage exploration of gesture-based systems, all aspects of the gestural interface can be specified while the application is running. A new gesture handler may be created at runtime and associated with one or more views or view classes. Gesture classes may be added, deleted, or copied from other handlers.

Examples of each gesture class can be entered and modified at runtime. Finally, the semantics of the gesture class can be entered and modified at runtime. Three semantic expressions are specifiable: one evaluated when the gesture is first recognized, one evaluated on each subsequent mouse point, and one evaluated when the interaction completes.

10.1.4 Input in Object-Oriented User Interface Toolkits

A number of new ideas in the area of input in object-oriented user interface toolkits arose in the course of this work.

Passive and active event handlers A single passive event handler may be associated with multiple views. When input occurs on one such view, the handler usually activates a copy of itself. Thus, the active/passive dichotomy eliminates the need to have a controller object instantiated for each view that expects input, a major expense in many MVC systems.

Event handlers may be associated with view classes Instead of having to associate a handler with every instance of a view, the handler may be associated with one or more view classes. A view may have multiple handlers associated with it, and handlers are queried in a specific order to determine which handler will handle particular input.

Unified mouse input and virtual tools All input devices are tools, but when desired a single input device may at times be different tools, one way to implement modes in the interface. Tools may also be software objects, and some views are indeed such virtual tools. Tools often have an action, which allows them to operate on any views that respond to that action. The test of whether a given view responds to a given tool is made by an event handler associated with every view; this allows semantic feedback to occur automatically without any explicit action on the part of the view or the tool.

Automatic semantic feedback As just mentioned, the feedback as to whether a given tool operates upon a view over which it is has been dragged happens automatically. For example, objects that respond to the delete message will automatically highlight when a delete tool is dragged over them. If desired, an object can do more elaborate processing to determine if it truly responds to a given tool, e.g. an object may check that the user has permission to delete it before indicating it responds to the delete tool.

Runtime creation and manipulation of event handlers Event handlers may be created and associated with views or view classes at runtime. For example, a drag handler may be associated with an object, allowing that object to be dragged (i.e. have its position changed). In addition, such handlers may be modified at runtime, for example, to change the predicate that activates the handler.

10.2 Future Directions

In this section, directions for future work are discussed. These directions include remedies for deficiencies of the current work as well as extensions.

The single-stroke training and recognition algorithm is the most robust and well-tested part of the current work, and even in its current form it is probably suitable for commercial applications. However, a number of simple modifications should improve performance. Sections 9.1.1 and 9.1.5 contain suggestions for additional features as well as modifications to existing features; these should be implemented. Tracking the mouse in the presence of paging has proved to be a problem, and a significant improvement in recognition rate would be achieved if real-time response to mouse events could be guaranteed.

It should be simple to extend the algorithm to three dimensional gestures. All that would be required would be to add several more features to capture motion in the extra dimension. The training algorithm and linear classifier would be untouched by this extension.

Alternatives for rejection should be explored further. The estimated probability of ambiguity is useful, though using it will always result in rejections of about as many gestures that would have been correctly classified as not. The estimated Mahalanobis distance based on the common covariance matrix is really only useful for rejecting deliberately garbled gestures. The Mahalanobis distance based on the per-class covariance matrix fares somewhat better, but requires significantly more training examples to work well.

Given the obvious false assumption of equal per-class covariance matrices, it seems that the statistical classifier should not perform well on gesture sets, some classes of which vary in size and orientation, others of which do not. In practice, when the gesture classes are unambiguous, the classifiers have tended to perform admirably. Presumably this would not be the case for all such gesture sets. One area for exploration is a method for calculating the common covariance matrix differently, in particular, by not weighing the per class contributions by the number of examples of that class.

Another challenge would be to handle such gesture sets without giving up linear classification with a closed form training formula. There seems to be only one candidate, which relies on the multiclass minimum squared error and the pseudoinverse of the matrix of examples [30]. It should be explored as a potential alternative to classifiers that rely on estimates of a common covariance matrix.

It would be interesting to explore the possibility of allowing the user to indicate declaratively that a given gesture classes will vary in size and/or orientation. This might be handled simply by generating additional training examples by varying the user-supplied examples accordingly. Alternatively, it may be possible to augment the training algorithm so that the evaluation functions for certain classes are constrained to ignore certain features.

Relaxing the requirement that a closed form exist for the per-class feature weights allows iterative training methods to be considered. They have been ignored in this dissertation since they are expensive in training time and tend to require many training examples. However, as processor speed increases, iterative methods become more practical for use in a tool for experimenting with gesture-based interfaces.

Similarly, relaxing the requirement that the classifier be a linear discriminator opens the door for many other possibilities. Quadratic discrimination, and various non-parametric discrimination algorithms are but a few. These too are expensive and require many training examples.

Perhaps recognition technologies that require expensive training may be used in a production system while the cheaper technology developed here used for prototyping. This is analogous to using a fast compiler for development and an optimizing compiler for production. At the time of this writing it seems likely that neural networks will soon be in common use, and gesture recognition is but one application.

Additional attention should be given to the problem of detecting ambiguous sets of gesture classes and useless features. The triangular matrix of Mahalanobis distance between each pair of gesture classes is a useful starting point for determining similar gesture classes. Multivariate analysis of variance techniques [74] can determine which features contribute to the classification and which features are irrelevant. These techniques can be used to support the design of new features.

Eager recognition needs to be explored further. The classifiers generated by the algorithm of Chapter 4 are less eager than they could possibly be, due to the conservative choices being made. Hand labeling of ambiguous and unambiguous subgestures should be explored more fully; it is not difficult to imagine an interface that makes such labeling relatively painless, and it is likely to give better results than the current automatic labeling. Another possible improvement comes from the observation that, during eager recognition, the full classifier is being used to classify subgestures, upon which it was not trained. It might be worth trying to retrain the full classifier on the complete subgestures. Even better, perhaps a new classifier, trained on the *newly* complete subgestures (*i.e.* those made complete by their last point), should be substituted for the full classifier. Also, eager recognition needs to be extended to multi-path gestures.

Algorithms for automatically determining the start of a gesture would also be useful, especially for devices without any discrete signaling capability (most notably the DataGlove). In the current work, gestures are considered atomic, essentially having no discernible structure. It is easy to imagine separate gestures such as select, copy, move, and delete that are concatenated to make single interactions: select and move, select and delete. This raises the segmentation question: when does one gesture end and the next begin? Specifying allowable combinations of gestures opens up the possibility of gesture grammars, an interesting area for future study.

This dissertation has concentrated on single-path gestures that are restricted to be single strokes, for reasons explained previously. The utility of multiple-stroke gestures needs to be examined more thoroughly. In a multiple-stroke gesture, does the relaxation between strokes ruin the correspondence between mental and physical tension that makes for good interaction? Does the need for segmentation make the system less responsive than it otherwise might be? Can a manipulation phase and eager recognition be incorporated into a system based on multiple-stroke gestures? These questions require further research.

Due to the interest in multiple stroke recognition, the question arises as to whether the single-stroke algorithm can be extended to handle multiple stroke gestures. First, the segmentation problem (grouping strokes into gestures) needs to be addressed. One way this might be done is to add a large timeout to determine the end of a gesture. The distance of a stroke from the previous stroke might also be used. A sequence of strokes determined to be a single gesture might then be treated as a single stroke, with the exception of an additional feature which records the number of strokes in the gesture. The single-stroke recognition algorithm may then be applied.

Multi-path recognition is really still in its infancy. While the recognition algorithms of Chapter 5 seem to work well, there is not much to compare them against. Many others methods for multi-path recognition need to be explored. That said, the author is somewhat wary that multiple finger input devices are so seductive that gesture research will concentrate on such devices to the exclusion of single-path devices. This would be unfortunate, as it seems likely that single-path devices will be much more prevalent for the foreseeable future, and thus more users will potentially benefit from the availability of single-path gesturing. Also, a thorough understanding of the issues involved in single-path gesturing will likely be of use in solving the more difficult problems encountered in the multi-path case.

The advent of pen-based computers leads to the question of how the single-stroke recognition described here may be combined with handwriting recognition. One approach is to pass input to the gesture recognizer after it has been rejected by the handwriting recognizer. The context in which the stroke has been made (e.g. drawing window or text window) can also be used to determine whether to invoke handwriting recognition or stroke recognition first.

The start of a single-stroke gesture is used to determine the set of possible gestures by looking at possible objects at which the gesture is directed. It may be desirable to explore the possibility that the gesture is directed at an object other than one indicated by the first point, e.g. an object may be indicated by a hot point of the gesture (e.g. the intersection point of the delete gesture). A similar ambiguity occurs when the input is a multiple-finger gesture; which of the fingers should be used to determine the object(s) at which the gesture is directed? In this case, a union of the gestures recognized by objects indicated by each finger could be used, but the possibility of conflict remains.

One problem with gesture-based systems is that there is usually no indication of the possible gestures accepted by the system.[1] This is a difficulty that will potentially prevent novices from using the system. One approach would be to use animation[6] to indicate the possible gestures and their effects, although how the user asks to see the animation remains an open question.

Also daunting to beginners is the timeout interval, where "stillness" is used to indicate that collection is over and manipulation is to begin. Typically, a beginner presses a mouse button and then thinks about what to do next; by that time the system has already classified the gesture as a dot. The timeout cannot be totally disabled, since it is the only way to enter the manipulation phase for some gestures. Perhaps some scheme where the timeouts are long (0.75 seconds) for novices and decrease with use is desirable. Another possibility is eliminating the timeout totally at the beginning of the gestures, thus disallowing dot gestures.

The current work suffers from a lack of formal user evaluation. Additional studies are needed to determine classifier performance as a function of training examples, and whether one user can use a classifier trained by another. In general, the costs and the benefits of fixed verses trainable recognition strategies need to be studied. The usability of eager recognizers is also of interest.

Recognizers that gradually adapt to users need to be studied as well. Such a recognizer requires the user to somehow indicate when a gesture is misclassified by the system. Lerner [78] demonstrated a potentially applicable scheme in which the system monitored subsequent actions to see if the user was satisfied with the result of an applied heuristic. There are dangers inherent in doubly-adaptive systems–if the system adapts to the user and the user to the system, both are aiming at moving targets, and thrashing is possible. The current approach requires the user explicitly to replace the

[1] Kurtenbach et. al. [75] say that gesture-based interfaces are "non-revealing," and present an interesting solution that unifies gesturing and pie-menu selection.

existing training examples with his own–a workable, if not glamorous, solution.

The low-level recognition work in this thesis is quite usable in its current state, and may be directly incorporated into systems as warranted. GRANDMA, however, is not useful as a base for future development. It is purely a research system, built as a platform for experimenting with input in user interface toolkits. Its output facilities are totally inadequate for real applications. GRANDMA was built solely by and for the author, who has no plans to maintain it. Nonetheless, GRANDMA embodies some important concepts of how gestures are to be integrated into object-oriented user interface tools.

The obvious next step is to integrate gestures into some existing user interface construction tools. Issues of technical suitability are important, but not paramount, in deciding which system to work on. Any chosen system must be well supported and maintained, so that there is a reasonable assurance that the system will survive. Furthermore, any chosen system must be widely distributed, in order to make the technology of gesture recognition available to as many experimenters as possible.

A number of existing systems are candidates for the incorporation of gestures. The NeXT Application Kit is technically the ideal platform–it is even programmed in Objective C. The appropriate hooks seem to be there to capture input at the right level in order to associate gestures with view classes. It is probably not worth the effort to implement an entire interpreter for entering gesture semantics at runtime, as this is not something a user will typically manipulate. A graphical interface to control semantics, based on constraints, would be an interesting addition. In general, a simpler way for mapping gestural attributes to application parameters needs to be determined.

The Andrew Toolkit (ATK) is another system into which gestures may be incorporated. ATK uses its own object-oriented programming language on top of C, so runtime representation of the class hierarchy, if not already present, should be straightforward to add. ATK has implemented dynamic loading of objects into running programs–this should make it possible to compile gesture semantics and load them into a running program without restarting the program. Unfortunately, due to their overhead, views tend to be large objects in ATK (e.g. individual notes in a score editor would not be separate views in ATK) making it difficult to associate different gestures with the smaller objects of interest in the interface. Scott Hassan, in a different approach, has added the author's gesture recognizer to the ATK text object, creating an interface that allows text editing via proofreader's marks.

Integrating gestures into Garnet is another possibility. What would be required is a gesture interactor, analogous to the gesture event handler in GRANDMA. Garnet interactors routinely specify their semantics via constraints, with an escape into Lisp available for unusual cases. Specifying gesture semantics should therefore be no problem in Garnet. James Landay has begun work integrating the author's recognizer into Garnet.

Gestures could also be added to MacApp. Besides being widely used, MacApp has the advantage that it runs on a Macintosh, which historically has run only one process at a time and has no virtual memory (this has changed with a recent system software release). While these points sound like disadvantages, the real-time operation needed to track the mouse reliably should be easy to achieve because of them. Because MacApp is implemented in Object Pascal, minimal meta-information about objects is available at runtime. In particular, message selectors are not first class objects in Object Pascal, it is not possible to ask if a given object responds to a message at runtime, and there is no runtime representation of the class hierarchy. Many things that happen automatically because GRANDMA is written in Objective C will need to be explicitly coded in MacApp.

It would be desirable to have additional attributes of the gesture available for use in gesture semantics. Notably missing from the current set are locations where the path intersects itself and locations of sharp corners of the stroke. Both kinds of attributes can be used for pointing with a gesture, and allow for multiple points to be indicated with one single-path gesture. Also, having the numerical attributes also available in a scaled form (*e.g.* between zero and one) would simplify their use as parameters to application functions.

10.3 Final Remarks

The utility of gesture-based interfaces derives from the ability to communicate an entire primitive application transaction with a single gesture. For this to be possible, the gesture needs to be classified to determine the operation to be performed, and attributes of the gesture must be mapped to the parameters of the operation. Some parameters may be culled at the time the gesture is recognized, while others are best manipulated in the presence of feedback from the application. This is the justification for the two-phase approach, where gesture recognition is followed by a manipulation phase, which allows for the continuous adjustment of parameters in the presence of application feedback.

From the outset, the goal of this work was to provide tools to allow the easy creation of gesture-based applications. This research has led to prototypes of such tools, and has thus laid much of the groundwork for building such tools in the future. However, the goal will not have been achieved until gestures are integrated into existing user interface construction tools that are both well maintained and highly available. This involves more development and marketing than it does research, but it is vitally important to the future of gesture-based systems.

Appendix A

Code for Single-Stroke Gesture Recognition and Training

This appendix contains the actual C code used to recognize single-stroke gestures. The feature vector calculation, classifier training algorithm, and the linear classifier are all presented. The code may be obtained free of charge via anonymous ftp to emsworth.andrew.cmu.edu (subdirectory gestures) and is also available as part of the Andrew contribution to the X11R5 distribution.

A.1 Feature Calculation

The lowest level of the code deals with computing a feature vector from a sequence of mouse points that make up a gesture. Type `FV` is a pointer to a structure that holds a feature vector as well as intermediate results used in the calculation of the features. The function `FvAlloc` allocates an `FV`, which is initialized before processing the points of a gesture via `FvInit`. `FvAddPoint` is called for each input point of the gesture, and `FvCalc` returns the feature vector for the gesture once all the points have been entered.

The following is a sample code fragment demonstrating the use of these functions:

```
#include "matrix.h"
#include "fv.h"

Vector
InputAGesture()
{
    static FV fv;
    int x, y; long t; Vector v;

    /* FvAlloc() is typically called only once per program invocation. */
    if(fv == NULL) fv = FvAlloc();
```

```
        /* A prototypical loop to compute a feature vector from a gesture
           being read from a window manager: */
        FvInit(fv);
        while(GetNextPoint(&x, &y, &t) != END_OF_GESTURE)
            FvAddPoint(fv, x, y, t);
        v = FvCalc(fv);
        return v;

    }
```

The returned vector v might now be passed to sClassify to classify the gesture.

The remainder of this section shows the header file, fv.h, which defines the FV type and the feature vector interface. This interface is implemented in fv.c, shown next.

```
/**********************************************************************
fv.h -  Create a feature vector, useful for gesture classification,
   from a sequence of points (e.g. mouse points).
**********************************************************************/

/* -------------- compile time settable parameters ---------------- */
/* some of these can also be set at runtime, see fv.c */

#undef  USE_TIME
                    /* Define USE_TIME to enable the duration and maximum */
                    /* velocity features.   When not defined, 0 may be passed */
                    /* as the time to FvAddPoint. */

#define DIST_SQ_THRESHOLD (3*3)
                    /* points within sqrt(DIST_SQ_THRESHOLD) */
                    /* will be ignored to eliminate mouse jitter */

#define SE_TH_ROLLOFF    (4*4)
                    /* The SE_THETA features (cos and sin of */
                    /* angle between first and last point) will */
                    /* be scaled down if the distance between the */
                    /* points is less than sqrt(SE_TH_ROLLOFF) */

/* ------------------ Interface ---------------- */

typedef struct fv *FV;
                    /* During gesture collection, an FV holds */
                    /* all intermediate results used in the */
                    /* calculation of a single feature vector */
```

```
FV      FvAlloc();      /* */
void    FvFree();       /* Fv fv */
void    FvInit();       /* FV fv */
void    FvAddPoint();   /* FV fv; int x, y; long time; */
Vector  FvCalc();       /* FV fv; */

/*---------- internal data structure ------------*/
#define MAXFEATURES 32
        /* maximum number of features, occasionally useful as an array bound */

/* indices into the feature Vector returned by FvCalc */

#define PF_INIT_COS    0 /* initial angle (cos) */
#define PF_INIT_SIN    1 /* initial angle (sin) */
#define PF_BB_LEN      2 /* length of bounding box diagonal */
#define PF_BB_TH       3 /* angle of bounding box diagonal */
#define PF_SE_LEN      4 /* length between start and end points */
#define PF_SE_COS      5 /* cos of angle between start and end points */
#define PF_SE_SIN      6 /* sin of angle between start and end points */
#define PF_LEN         7 /* arc length of path */
#define PF_TH          8 /* total angle traversed */
#define PF_ATH         9 /* sum of abs vals of angles traversed */
#define PF_SQTH       10 /* sum of squares of angles traversed */

#ifndef USE_TIME
#   define NFEATURES      11
#else
#   define PF_DUR         11      /* duration of path */
#   define PF_MAXV        12      /* maximum speed */
#   define NFEATURES      13
#endif

/* structure which holds intermediate results during feature vector calculation */

struct fv {

    /* the following are used in calculating the features */
    double      startx, starty;  /* starting point */
    long        starttime;       /* starting time */

    /* these are set after a few points and then left alone */
    double      initial_sin, initial_cos; /* initial angle to x axis */

    /* these are updated incrementally upon every point */
    int     npoints;            /* number of points in path */

    double      dx2, dy2;       /* differences: endx-prevx, endy-prevy */
    double      magsq2;         /* dx2*dx2 + dy2*dy2 */

    double      endx, endy;     /* last point added */
    long        endtime;

    double      minx, maxx, miny, maxy;    /* bounding box */

    double      path_r, path_th; /* total length and rotation (in radians) */
    double      abs_th;          /* sum of absolute values of path angles */
    double      sharpness;       /* sum of squares of path angles */
    double      maxv;            /* maximum velocity */


    Vector      y;               /* Actual feature vector */
};


/*********************************************************************
fv.c - Creates a feature vector, useful for gesture classification,
  from a sequence of points (e.g. mouse points).
*********************************************************************/

#include <stdio.h>
#include <math.h>
#include "matrix.h"  /* contains Vector and associated functions */
#include "fv.h"

/* runtime settable parameters */
double dist_sq_threshold = DIST_SQ_THRESHOLD;
double se_th_rolloff = SE_TH_ROLLOFF;

#define EPS (1.0e-4)

/* allocate an FV struct including feature vector */

FV
```

```
FvAlloc()
{
    register FV fv = (FV) mallocOrDie(sizeof(struct fv));
    fv->y = NewVector(NFEATURES);
    FvInit(fv);
    return fv;
}

/* free memory associated with an FVstruct */
void
FvFree(fv)
FV fv;
{
    FreeVector(fv->y);
    free((char *) fv);
}

/* initialize an FVstruct to prepare for incoming gesture points */
void
FvInit(fv)
register FV fv;
{
    register int i;

    fv->npoints = 0;
    fv->initial_sin = fv->initial_cos = 0.0;
    fv->maxv = 0;
    fv->path_r = 0;
    fv->path_th = 0;
    fv->abs_th = 0;
    fv->sharpness = 0;
    fv->maxv = 0;
    for(i = 0; i < NFEATURES; i++)
        fv->y[i] = 0.0;
}

/* update an FVstruct to reflect a new input point */
void
FvAddPoint(fv, x, y, t)
register FV fv; int x, y; long t;
{
    double dx1, dy1, magsq1;
    double th, absth, d;
#ifdef PF_MAXV
    long lasttime;
#endif

    ++fv->npoints;
    if(fv->npoints == 1) {      /* first point, initialize some vars */
        fv->starttime = fv->endtime = t;
        fv->startx = fv->endx = fv->minx = fv->maxx = x;
        fv->starty = fv->endy = fv->miny = fv->maxy = y;
        fv->endx = x; fv->endy = y;
        return;
    }

    dx1 = x - fv->endx; dy1 = y - fv->endy;
    magsq1 = dx1 * dx1 + dy1 * dy1;

    if(magsq1 <= dist_sq_threshold) {
        fv->npoints--;
        return;         /* ignore a point close to the last point */
    }

    if(x < fv->minx) fv->minx = x;
    if(x > fv->maxx) fv->maxx = x;
    if(y < fv->miny) fv->miny = y;
    if(y > fv->maxy) fv->maxy = y;

#ifdef PF_MAXV
    lasttime = fv->endtime;
#endif
    fv->endtime = t;

    d = sqrt(magsq1);
    fv->path_r += d;        /* update path length feature */

    /* calculate initial theta when the third point is seen */
    if(fv->npoints == 3) {
        double magsq, dx, dy, recip;
        dx = x - fv->startx; dy = y - fv->starty;
        magsq = dx * dx + dy * dy;
        if(magsq > dist_sq_threshold)  {
            /* find angle w.r.t. positive x axis e.g. (1,0) */
            recip = 1 / sqrt(magsq);
            fv->initial_cos = dx * recip;
            fv->initial_sin = dy * recip;
        }
    }

    if(fv->npoints >= 3) {    /* update angle-based features */
        th = absth = atan2(dx1 * fv->dy2 - fv->dx2 * dy1,
                     dx1 * fv->dx2 + dy1 * fv->dy2);
        if(absth < 0) absth = -absth;
        fv->path_th += th;
        fv->abs_th += absth;
        fv->sharpness += th*th;

#ifdef PF_MAXV    /* compute max velocity */
        if(fv->endtime > lasttime &&
           (v = d / (fv->endtime - lasttime)) > fv->maxv)
            fv->maxv = v;
#endif
    }

    /* prepare for next iteration */
    fv->endx = x; fv->endy = y;
    fv->dx2 = dx1; fv->dy2 = dy1;
    fv->magsq2 = magsq1;

    return;
}

/* calculate and return a feature vector */
Vector
FvCalc(fv)
register FV fv;
{
    double bblen, selen, factor;

    if(fv->npoints <= 1)
        return fv->y;           /* a feature vector of all zeros */

    fv->y[PF_INIT_COS] = fv->initial_cos;
    fv->y[PF_INIT_SIN] = fv->initial_sin;
    /* compute the length of the bounding box diagonal */
    bblen = hypot(fv->maxx - fv->minx, fv->maxy - fv->miny);

    fv->y[PF_BB_LEN] = bblen;

    /* the bounding box angle defaults to 0for small gestures */
    if(bblen * bblen > dist_sq_threshold)
        fv->y[PF_BB_TH] = atan2(fv->maxy - fv->miny,
                                fv->maxx - fv->minx);

    /* compute the length and angle between the first and last points */
    selen = hypot(fv->endx - fv->startx,
                  fv->endy - fv->starty);
    fv->y[PF_SE_LEN] = selen;

    /* when the first and last points are very close, the angle features
       are muted so that they satisfy the stability criterion */
    factor = selen * selen / se_th_rolloff;
    if(factor > 1.0) factor = 1.0;
    factor = selen > EPS ? factor/selen :  0.0;
    fv->y[PF_SE_COS] = (fv->endx - fv->startx) * factor;
    fv->y[PF_SE_SIN] = (fv->endy - fv->starty) * factor;

    /* the remaining features have already been computed */
    fv->y[PF_LEN] =  fv->path_r;
    fv->y[PF_TH] = fv->path_th;
    fv->y[PF_ATH] = fv->abs_th;
    fv->y[PF_SQTH] = fv->sharpness;

#ifdef PF_DUR
    fv->y[PF_DUR] = (fv->endtime - fv->starttime)*.01;
#endif

#ifdef PF_MAXV
    fv->y[PF_MAXV] = fv->maxv * 10000;
#endif

    return fv->y;
}
```

A.2 Deriving and Using the Linear Classifier

Type sClassifier points at an object that represents a classifier able to discriminate between a set of gesture classes. Each gesture class is represented by an sClassDope type. The functions sRead and sWrite read and write a classifier to a file. The function sNewClassifer creates a new (empty) classifier. A training example is added using sAddExample. There is no function to explicitly add a new class to a classifier. When an example of a new class is added, the new class is created automatically. To train the classifier based on the added examples, call sDoneAdding. Once trained, sClassify and sClassifyAD are used to classify a feature vector as one of the classes; sClassifyAD optionally computes the rejection information.

Here is an example fragment for creating a new classifier, entering new training examples, and writing the resulting classifier out to a file. Some of these functions are timed (and further described) in section 9.1.7.

```
#include <stdio.h>
#include <math.h>
#include "bitvector.h"
#include "matrix.h"
#include "sc.h"

#define NEXAMPLES   15

sClassifier
MakeAClassifier()
{
    sClassifier sc = sNewClassifier();
    Vector InputAGesture();
    char name[100];
    int i;

    for(;;) {
        printf("Enter class name, newline to exit: ");
        if(gets(name) == NULL || name[0] == '\0')
            break;
        for(i = 1; i <= NEXAMPLES; i++) {
            printf("Enter %s example %d\n", name, i);
            sAddExample(sc, name, InputAGesture());
        }
    }
    sDoneAdding(sc);
    sWrite(fopen("classifier.out", "w"), sc);
    return sc;
}
```

Once a classifier has been created it can be used to classifier gestures as follows:

```
TestAClassifier(sc)
sClassifier sc;
{
    Vector v;
    sClassDope scd;
    double punambig, distance;

    for(;;) {
        printf("Enter a gesture\n");
        v = InputAGesture();
        scd = sClassifyAD(sc, v, &punambig, &distance);
        printf("Gesture classified as %s ", scd->name);
        printf("Probability of unambiguous classification: %g\n",
                 punambig);
        printf("Distance from class mean: %g\n", distance);
    }
}
```

What follows is the header file and code to implement the statistical classifier.

```
/*********************************************************************
sc.h - create single path classifiers from feature vectors of examples,
  as well as classifying example feature vectors.
*********************************************************************/

#define MAXSCLASSES 100 /* maximum number of classes */

typedef struct sclassifier *sClassifier;  /* classifier */
typedef int sClassIndex;                  /* per-class index */
typedef struct sclassdope  *sClassDope;   /* per-class information */

struct sclassdope { /* per gesture class information within a classifier */
    char        *name;        /* name of a class */
    sClassIndex number;       /* unique index (small integer) of a class */
    int         nexamples;    /* number of training examples */
    Vector      average;      /* average of training examples */
    Matrix      sumcov;       /* covariance matrix of examples */
};
```

```
struct sclassifier {  /* a classifier */
     int       nfeatures;       /* number of features in feature vector */
     int       nclasses;        /* number of classes known by this classifier */
     sClassDope   *classdope; /* array of pointers to per class data */

     Vector       cnst;         /* constant term of discrimination function */
     Vector       *w;           /* array of coefficient weights */
     Matrix       invavgcov;    /* inverse covariance matrix */
};

sClassifier sNewClassifier();        /* */
sClassifier sRead();                 /* FILE *f */
void        sWrite();                /* FILE *f; sClassifier sc; */
void        sFreeClassifier();       /* sc */
void        sAddExample();           /* sc, char *classname; Vector y */
void        sDoneAdding();           /* sc */
sClassDope  sClassify();             /* sc, y */
sClassDope  sClassifyAD();           /* sc, y, double *ap; double *dp */
sClassDope  sClassNameLookup();      /* sc, classname */
double      MahalanobisDistance(); /* Vector v, u; Matrix sigma */

/*****************************************************************
sc.c – creates classifiers from feature vectors of examples, as well as
  classifying example feature vectors.
*****************************************************************/


#include <stdio.h>
#include <math.h>
#include "bitvector.h"
#include "matrix.h"
#include "sc.h"

#define EPS (1.0e-6)      /* for singular matrix check */

/* allocate memory associated with a new classifier */

sClassifier
sNewClassifier()
{
     register sClassifier sc =
            (sClassifier) mallocOrDie(sizeof(struct sclassifier));
        sc->nfeatures = -1;
        sc->nclasses = 0;
        sc->classdope = (sClassDope *)
            mallocOrDie(MAXSCLASSES * sizeof(sClassDope));
        sc->w = NULL;
        return sc;
    }

    /* free memory associated with a new classifier */
    void
    sFreeClassifier(sc)
    register sClassifier sc;
    {
        register int i;
        register sClassDope scd;

        for(i = 0; i < sc->nclasses; i++) {
            scd = sc->classdope[i];
            if(scd->name) free(scd->name);
            free(scd);
            if(sc->w && sc->w[i]) FreeVector(sc->w[i]);
            if(scd->sumcov) FreeMatrix(scd->sumcov);
            if(scd->average) FreeVector(scd->average);
        }
        free(sc->classdope);
        if(sc->w) free(sc->w);
        if(sc->cnst) FreeVector(sc->cnst);
        if(sc->invavgcov) FreeMatrix(sc->invavgcov);
        free(sc);
    }

    /* given a string name of a class, return its per-class information */
    sClassDope
    sClassNameLookup(sc, classname)
    register sClassifier sc;
    register char *classname;
    {
        register int i;
        register sClassDope scd;
        static sClassifier lastsc;
        static sClassDope lastscd;
    /* quick check for last class name */
    if(lastsc == sc && STREQ(lastscd->name, classname))
        return lastscd;

    /* linear search through all classes for name */
    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        if(STREQ(scd->name, classname))
            return lastsc = sc, lastscd = scd;
    }
    return NULL;
}

/* add a new gesture class to a classifier */
static sClassDope
sAddClass(sc, classname)
register sClassifier sc;
char *classname;
{
    register sClassDope scd;

    sc->classdope[sc->nclasses] = scd = (sClassDope)
               mallocOrDie(sizeof(struct sclassdope));
    scd->name = scopy(classname);
    scd->number = sc->nclasses;
    scd->nexamples = 0;
    scd->sumcov = NULL;
    ++sc->nclasses;
    return scd;
}

/* add a new training example to a classifier */
void
sAddExample(sc, classname, y)
register sClassifier sc;
char *classname;
Vector y;
{
    register sClassDope scd;
    register int i, j;
    double nfv[50];
    double nm1on, recipn;

    scd = sClassNameLookup(sc, classname);
    if(scd == NULL)
        scd = sAddClass(sc, classname);

    if(sc->nfeatures == -1)
        sc->nfeatures = NROWS(y);

    if(scd->nexamples == 0) {
        scd->average = NewVector(sc->nfeatures);
        ZeroVector(scd->average);
        scd->sumcov = NewMatrix(sc->nfeatures,sc->nfeatures);
        ZeroMatrix(scd->sumcov);

    }

    if(sc->nfeatures != NROWS(y)) {
        PrintVector(y, "sAddExample: funny vector nrows!=%d",
            sc->nfeatures);
        return;
    }

    scd->nexamples++;
    nm1on = ((double) scd->nexamples-1)/scd->nexamples;
    recipn = 1.0/scd->nexamples;

    /* incrementally update covariance matrix */
        for(i = 0; i < sc->nfeatures; i++)
        nfv[i] = y[i] - scd->average[i];

    /* only upper triangular part computed */
        for(i = 0; i < sc->nfeatures; i++)
       for(j = i; j < sc->nfeatures; j++)
        scd->sumcov[i][j] += nm1on * nfv[i] * nfv[j];

    /* incrementally update mean vector */
    for(i = 0; i < sc->nfeatures; i++)
        scd->average[i] =
             nm1on * scd->average[i] + recipn * y[i];

}
```

```
/* run the training algorithm on the classifier */
void
sDoneAdding(sc)
register sClassifier sc;
{
    register int i, j;
    int c;
    int ne, denom;
    double oneoverdenom;
    register Matrix s;
    register Matrix avgcov;
    double det;
    register sClassDope scd;

    if(sc->nclasses == 0)
        error("sDoneAdding: No classes\n");

    /* Given covariance matrices for each class (* number of examples - 1)
       compute the average (common) covariance matrix */

    avgcov = NewMatrix(sc->nfeatures, sc->nfeatures);
    ZeroMatrix(avgcov);
    ne = 0;
    for(c = 0; c < sc->nclasses; c++) {
        scd = sc->classdope[c];
        ne += scd->nexamples;
        s = scd->sumcov;
        for(i = 0; i < sc->nfeatures; i++)
            for(j = i; j < sc->nfeatures; j++)
                avgcov[i][j] += s[i][j];
    }

    denom = ne - sc->nclasses;
    if(denom <= 0) {
        printf("no examples, denom=%d\n", denom);
        return;
    }

    oneoverdenom = 1.0 / denom;
    for(i = 0; i < sc->nfeatures; i++)
        for(j = i; j < sc->nfeatures; j++)
```

```
            avgcov[j][i] = avgcov[i][j] *= oneoverdenom;

    /* invert the avg covariance matrix */

    sc->invavgcov = NewMatrix(sc->nfeatures, sc->nfeatures);
    det = InvertMatrix(avgcov, sc->invavgcov);
    if(fabs(det) <= EPS)
        FixClassifier(sc, avgcov);

    /* now compute discrimination functions */
    sc->w = (Vector *)
              mallocOrDie(sc->nclasses * sizeof(Vector));
    sc->cnst = NewVector(sc->nclasses);
    for(c = 0; c < sc->nclasses; c++) {
        scd = sc->classdope[c];
        sc->w[c] = NewVector(sc->nfeatures);
        VectorTimesMatrix(scd->average, sc->invavgcov,
                        /* product = */ sc->w[c]);
        sc->cnst[c] = -0.5 *
                 InnerProduct(sc->w[c], scd->average);
        /* could add log(priorprob class c) to cnst[c] */
    }

    FreeMatrix(avgcov);
    return;
}

/* classify a feature vector */
sClassDope
sClassify(sc, fv) {
    return sClassifyAD(sc, fv, NULL, NULL);
}

/* classify a feature vector, possibly computing rejection metrics */
sClassDope
sClassifyAD(sc, fv, ap, dp)
sClassifier sc;
Vector fv;
double *ap;
double *dp;
{
    double disc[MAXSCLASSES];
    register int i, maxclass;
    double denom, exp();
    register sClassDope scd;
    double d;

    if(sc->w == NULL)
        error("sClassifyAD: %x no trained classifier", sc);

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

    maxclass = 0;
    for(i = 1; i < sc->nclasses; i++)
        if(disc[i] > disc[maxclass])
            maxclass = i;

    scd = sc->classdope[maxclass];

    if(ap) {        /* calculate probability of non-ambiguity */
        for(denom = 0, i = 0; i < sc->nclasses; i++)
            /* quick check to avoid computing negligible term */
            if((d = disc[i] - disc[maxclass]) > -7.0)
                denom += exp(d);
        *ap = 1.0 / denom;
    }

    if(dp)    /* calculate distance to mean of chosen class */
        *dp = MahalanobisDistance(fv, scd->average,
                                  sc->invavgcov);

    return scd;
}

/* Compute the Mahalanobis distance between two vectors v and u */
double
MahalanobisDistance(v, u, sigma)
register Vector v, u;
register Matrix sigma;
{
    register i;
    static Vector space;
    double result;
    if(space == NULL || NROWS(space) != NROWS(v)) {
        if(space) FreeVector(space);
        space = NewVector(NROWS(v));
    }
    for(i = 0; i < NROWS(v); i++)
        space[i] = v[i] - u[i];
    result =  QuadraticForm(space, sigma);
    return result;
}

/* handle the case of a singular average covariance matrix by removing features */
FixClassifier(sc, avgcov)
register sClassifier sc;
Matrix avgcov;
{
    int i;
    double det;
    BitVector bv;
    Matrix m, r;

    /* just add the features one by one, discarding any that cause
       the matrix to be non-invertible */

    CLEAR_BIT_VECTOR(bv);
    for(i = 0; i < sc->nfeatures; i++) {
        BIT_SET(i, bv);
        m = SliceMatrix(avgcov, bv, bv);
        r = NewMatrix(NROWS(m), NCOLS(m));
        det = InvertMatrix(m, r);
        if(fabs(det) <= EPS)
            BIT_CLEAR(i, bv);
        FreeMatrix(m);
        FreeMatrix(r);
    }

    m = SliceMatrix(avgcov, bv, bv);
    r = NewMatrix(NROWS(m), NCOLS(m));
    det = InvertMatrix(m, r);
    if(fabs(det) <= EPS)
        error("Can't fix classifier!");
    DeSliceMatrix(r, 0.0, bv, bv, sc->invavgcov);
    FreeMatrix(m);
    FreeMatrix(r);

}

/* write a classifier to a file */
void
sWrite(outfile, sc)
FILE *outfile;
sClassifier sc;
{
    int i;
    register sClassDope scd;

    fprintf(outfile, "%d classes\n", sc->nclasses);
    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        fprintf(outfile, "%s\n", scd->name);
    }
    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        OutputVector(outfile, scd->average);
        OutputVector(outfile, sc->w[i]);

    }
    OutputVector(outfile, sc->cnst);
    OutputMatrix(outfile, sc->invavgcov);
}

/* read a classifier from a file */
sClassifier
sRead(infile)
FILE *infile;
{
    int i, n;
    register sClassifier sc;
    register sClassDope scd;
    char buf[100];


    printf("Reading classifier "), fflush(stdout);
    sc = sNewClassifier();
    fgets(buf, 100, infile);
    if(sscanf(buf, "%d", &n) != 1) error("sRead 1");
    printf("%d classes ", n), fflush(stdout);
    for(i = 0; i < n; i++) {
        fscanf(infile, "%s", buf);
        scd = sAddClass(sc, buf);
        scd->name = scopy(buf);
        printf("%s ", scd->name), fflush(stdout);
    }
    sc->w = allocate(sc->nclasses, Vector);
    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        scd->average = InputVector(infile);
        sc->w[i] = InputVector(infile);
    }
    sc->cnst = InputVector(infile);
    sc->invavgcov = InputMatrix(infile);
    printf("\n");
    return sc;
}

/* compute pairwise distances between classes, and print the closest ones,
   as a clue as to which gesture classes are confusable */

sDistances(sc, nclosest)
register sClassifier sc;
{
    register Matrix d = NewMatrix(sc->nclasses, sc->nclasses);
    register int i, j;
    double min, max = 0;
    int n, mi, mj;

    printf("-------------\n");
    printf("%d closest pairs of classes\n", nclosest);
    for(i = 0; i < NROWS(d); i++) {
        for(j = i+1; j < NCOLS(d); j++) {
            d[i][j] = MahalanobisDistance(
                            sc->classdope[i]->average,
                            sc->classdope[j]->average,
                            sc->invavgcov);
            if(d[i][j] > max) max = d[i][j];
            }
        }

        for(n = 1; n <= nclosest; n++) {
            min = max;
            mi = mj = -1;
            for(i = 0; i < NROWS(d); i++) {
                for(j = i+1; j < NCOLS(d); j++) {
                    if(d[i][j] < min)
                        min = d[mi=i][mj=j];
                }
            }
            if(mi == -1)
                break;

            printf("%2d) %10.10s to %10.10s d=%g nstd=%g\n",
                n,
                sc->classdope[mi]->name,
                sc->classdope[mj]->name,
                d[mi][mj],
                sqrt(d[mi][mj]));

            d[mi][mj] = max+1;
        }
        printf("----------------\n");
        FreeMatrix(d);
    }
```

A.3 Undefined functions

The above code uses some functions whose definitions are not included in this appendix. These fall into four classes: standard library functions (including the math library), utility functions, bitvector functions, and vector/matrix functions, The standard library calls will not be discussed.

The utility functions used are

`STREQ(s1, s2)` returns FALSE iff strings `s1` and `s2` are equal.

`scopy(s)` returns a copy of the string `s`.

`error(format, arg1...)` prints a message and causes the program to exit.

`mallocOrDie(nbytes)` calls malloc, dying with an error message if the memory cannot be obtained.

The bit vector operations are an efficient set of functions for accessing an array of bits.

`CLEAR_BIT_VECTOR(bv)` resets an entire bit vector `bv` to all zeros,

`BIT_SET(i, bv)` sets the `i`th bit of `bv` to one, and

`BIT_CLEAR(i, bv)` sets the `i`th bit of `bv` to zero.

The vector/matrix functions are declared in matrix.h. Objects of type `Vector` and `Matrix` may be accessed like one and two dimensional arrays, respectively, but also contain additional information as to the size and dimensionality of the object (accessible via macros `NROWS`, `NCOLS`, and `NDIM`. It should be obvious from the names and the use of most of the functions (`NewVector`, `NewMatrix`, `FreeVector`, `FreeMatrix`, `ZeroVector`, `ZeroMatrix`, `PrintVector`, `PrintMatrix`, `InvertMatrix`, `InputVector`, `InputMatrix`, `OutputVector`, `OutputMatrix`, `VectorTimesMatrix`, and `InnerProduct`) what they do. As for the remaining functions,

`double QuadraticForm(Vector V, Matrix M)` computes the quantity $V'MV$, where the prime denotes the transpose operation.

`Matrix SliceMatrix(Matrix m, BitVector rowmask, BitVector colmask)` creates a new matrix, consisting only of those rows and columns in `m` whose corresponding bits are set it `rowmask` and `colmask`, respectively.

`Matrix DeSliceMatrix(Matrix m, double fill, BitVector rowmask; BitVector colmask; Matrix result)` first sets every element in `result` to `fill`, and then, every element in `result` whose row number is on in `rowmask` and whose column number is on in `colmask`, is set from the corresponding element in the input matrix `m`, which is smaller than `r`. The result of `SliceMatrix(DeSliceMatrix(m, fill, rowmask, colmask, result), rowmask, colmask)` is a copy of `m`, given legal values for all parameters.

These auxiliary functions, as well as a C-based X11R5 version of GDP, are all available as part of the ftp distribution mentioned above.

Bibliography

[1] Apple. *Inside Macintosh.* Addison-Wesley, 1985.

[2] Apple. *Macintosh System Software User's Guide, Version 6.0* Apple Computer, 1988.

[3] H. Arakawa, K. Okada, and J. Masuda. On-line recognition of handwritten characters: Alphanumerics, Hiragana, Katakana, Kanji. In *Proceedings of the 4th International Joint Conference on Pattern Recognition*, pages 810–812, 1978.

[4] R. Baecker. Towards a characterization of graphical interaction. In Guedj, R. A., et. al., editor, *Methodology of Interaction*, pages 127–147. North Holland, 1980.

[5] R. Baecker and W. A. S. Buxton. *Readings in Human-Computer Interaction - A Multidisciplinary Approach.* Morgan Kaugmann Readings Series. Morgan Kaufmann, Los Altos, California, 1987.

[6] Ronald Baecker, Ian Small, and Richard Mander. Bringing icons to life. In *CHI' 91 Conference Proceedings*, pages 1–6. ACM, April 1991.

[7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM* 18(9), 1975.

[8] M. Berthod and J. P. Maroy. Learning in syntactic recognition of symbols drawn on a graphic tablet. *Computer Graphics Image Processing*, 9:166–182, 1979.

[9] J. Block and R. Dannenberg. Polyphonic accompaniment in real time. In *International Computer Music Conference*, Cambridge, Mass., 1985. Computer Music Association.

[10] R. Boie. Personnel communication. 1987.

[11] R. Boie, M. Mathews, and A. Schloss. The Radio Drum as a synthesizer controller. In *1989 International Computer Music Proceedings*, pages 42–45. Computer Music Association, November 1989.

[12] R. A. Bolt. *The Human Interface: where people and computers meet.* Lifetime Learning Publications, 1984.

[13] Radmilo M. Bozinovic. *Recognition of Off-line Cursive Handwriting: A Case of Multi-level Machine Perception.* PhD thesis, State University of New York at Buffalo, March 1985.

[14] W. A. S. Buxton. Chunking and phrasing and the design of human-computer dialogues. In *Information Processing 86*, North Holland, 1986. Elsevier Science Publishers B.V.

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

[16] W. A. S. Buxton. Smoke and mirrors. *Byte*, 15(7):205–210, July 1990.

[17] W. A. S. Buxton. A three-state model of graphical input. *Proceedings of Interact 90*, August 1990.

[18] W. A. S. Buxton, R. Hill, and P. Rowley. Issues and techniques in touch-sensitive tablet input. *Computer Graphics*, 19(3):215–224, 1985.

[19] W. A. S. Buxton and B. Myers. A study in two-handed input. In *Proceedings of CHI '86*, pages 321–326. ACM, 1986.

[20] W. A. S. Buxton, W. Reeves, R. Baecker, and L. Mezei. The user of hierarchy and instance in a data structure for computer music. In Curtis Roads and John Strawn, editors, *Foundations of Computer Music*, chapter 24, pages 443–466. MIT Press, Cambridge, Massachusetts, 1985.

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

[22] S. K. Card, Moran, T. P., and A. Newell. The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23(7):601–613, 1980.

[23] L. Cardelli and R. Pike. Squeak: A language for communicating with mice. *SIGGRAPH '85 Proceedings*, 19(3), April 1985.

[24] R. M. Carr. The point of the pen. *BYTE*, 16(2):211–221, February 1991.

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

[26] P. W. Cooper. Hyperplanes, hyperspheres, and hyperquadrics as decision boundaries. In J. T. Tou and R. H. Wilcox, editors, *Computer and Information Sciences*, pages 111–138. Spartan, Washington, D.C., 1964.

[27] Brad J. Cox. Message/object programming: An evolutionary change in programming technology. *IEEE Software*, 1(1):50–61, January 1984.

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

[29] R. B. Dannenberg. A structure for representing, displaying, and editing music. In *Proceedings of the 1986International Computer Music Conference*, pages 153–160, San Francisco, 1986. Computer Music Association.

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

[31] The Economist. Digital quill. *The Economist*, 316(7672):88, September 15 1990.

[32] H. Eglowstein. Reach out and touch your data. *Byte*, 15(7):283–290, July 1990.

[33] W. English, D. Engelbart, and M. L. Berman. Display-selection techniques for text manipulation. *IEEE Transactions on Human Factors in Electronics*, HFE-8(1):21–31, 1967.

[34] S. S. Fels and Geoffrey E. Hinton. Building adaptive interfaces with neural networks: The glove-talk pilot study. Technical Report CRG-TR-90-1, University of Toronto, Toronto, Canada, 1990.

[35] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936.

[36] Flecchia and Nergeron. Specifying complex dialogs in ALGAE. *SIGCHI+ GI 87Proceedings*, April 1987.

[37] K. S. Fu. *Syntactic Recognition in Character Recognition*, volume 112 of *Mathematics in Science and Engineering*. Academic Press, 1974.

[38] K. S. Fu. Hybrid aproaches to pattern recognition. In K. S. Fu J. Kittler and L. F. Pau, editors, *Pattern Recognition Theory and Applications*, NATO Advanced Study Institute, pages 139–155. D. Reidel, 1981.

[39] K. S. Fu. *Syntactic Pattern Recognition and Applications*. Prentice Hall, 1981.

[40] K. S. Fu and T. S. Yu. *Statistical Pattern Classification using Contextual Information*. Pattern Recognition and Image Processing Series. Research Studies Press, New York, 1980.

[41] J. Gettys, R. Newman, and R. W. Schiefler. *Xlib - C Language Interface X11R2* Massachusetts Institute of Technology, 1988.

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

[43] R. Glitman. Startup readies 4-pound stylus pc. *PC Week*, 7(34):17–18, August 27 1990.

[44] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley series in Computer Science. Addison-Wesley, 1983.

[45] D. Goodman. *The complete HyperCard handbook*. Bantam Books, 1988.

[46] G. H. Granlund. Fourier preprocessing for hand print character recognition. *IEEE Transactions on Computers*, 21:195–201, February 1972.

[47] I. Guyon, P. Albrecht, Y. Le Cun, J. Denker, and W. Hubbard. Design of a neural network character recognizer for a touch terminal. *Pattern Recognition*, 24(2):105–119, 1991.

[48] D. J. Hand. *Kernel Discriminant Analysis*. Pattern Recognition and Image Processing Research Studies Series. Research Studies Press (A Division of John Wiley and Sons, Ltd.), New York, 1982.

[49] A. G. Hauptmann. Speech and gestures for graphic image manipulation. In *CHI '89 Proceedings*, pages 241–245. ACM, May 1989.

[50] Frank Hayes. True notebook computing arrives. *Byte*, pages 94–95, December 1989.

[51] P. J. Hayes, P. A. Szekely, and R. A. Lerner. Design alternatives for user interface management systems based on experience with COUSIN. In *CHI '85 Proceedings*, pages 169–175. ACM, April 1985.

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

[53] C. A. Higgins and R. Whitrow. On-line cursive script recognition. In B. Shackel, editor, *Human-Computer Interaction - Interact '84, IFIP*, pages 139–143, North-Holland, 1985. Elsevier Science Publishers B.V.

[54] R. Hill. Supporting concurrency, communication, and synchronization in human-computer interaction. *ACM Transactions on Graphics*, 5(3):179–210, July 1986.

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

[56] Bruce L. Horn. An introduction to object oriented programming, inheritance and method combination. Technical Report CMU-CS-87-127, Carnegie Mellon University Computer Science Department, 1988.

[57] A. B. S. Hussain, G. T. Toussaint, and R. W. Donaldson. Results obtained using a simple character recognition procedure on Munson's handprinted data. *IEEE Transactions on Computers*, 21:201–205, February 1972.

[58] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. In D. A. Norman and S. W. Draper, editors, *User Centered System Design*, pages 118–123. Lawrence Erlbaum Associates, Hillsdale, N.J., 1986.

[59] Pencept Inc. Software control at the stroke of a pen. In *SIGGRAPH Video Review*, volume Issue 18: Edited Compilations from CHI '85. ACM, 1985.

[60] F. Itakura. Minimum prediction residual principle applied to speech recognition. *IEEE Trans. Acoustics, Speech, Signal Processing*, ASSP-23(67), 1975.

[61] J. C. Jackson and Renate J. Roske-Hofstrand. Circling: A method of mouse-based selection without button presses. In *CHI '89 Proceedings*, pages 161–166. ACM, May 1989.

[62] Mike James. *Classification Algorithms*. Wiley-Interscience. John Wiley and Sons, Inc., New York, 1985.

[63] R. E. Johnson. Model/View/Controller. November 1987 (unpublished manuscript).

[64] Dan R. Olsen Jr. Syngraph: A graphical user interface generator. *Computer Graphics*, 17(3):43–50, July 1983.

[65] K. G. Morse Jr. In an upscale world. *Byte*, 14(8), August 1989.

[66] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, New Jersey, 1978.

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

[68] Nancy T. Knolle. Variations of model-view-controller. *Journal of Object-Oriented Programming*, 2:42–46, September/October 1989.

[69] D. Kolzay. Feature extraction in an optical character recognition machine. *IEEE Transactions on Computers*, 20:1063–1067, 1971.

[70] Glenn E. Krasner and Stephen T. Pope. A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, August 1988.

[71] M. W. Kreuger. *Artificial Reality*. Addison-Wesley, Reading, MA., 1983.

[72] M. W. Kreuger, T. Gionfriddo, and K. Hinrichsen. Videoplace: An artificial reality. In *Proceedings of CHI'85*, pages 35–40. ACM, 1985.

[73] E. Kreyszig. *Advanced Engineering Mathematics*. Wiley, 1979. Fourth Edition.

[74] W. J. Krzanowski. *Principles of Multivariate Analysis: A User's Perspective*. Oxford Statistical Science Series. Clarendon Press, Oxford, 1988.

[75] G. Kurtenbach and W. A. S. Buxton. GEdit: A test bed for editing by contiguous gestures. *SIGCHI Bulletin*, pages 22–26, 1991.

[76] Martin Lamb and Veronica Buckley. New techniques for gesture-based dialog. In B. Shackel, editor, *Human-Computer Interaction - Interact '84, IFIP*, pages 135–138, North-Holland, 1985. Elsevier Science Publishers B.V.

[77] C. G. Leedham, A. C. Downton, C. P. Brooks, and A. F. Newell. On-line acquisition of pitman's handwritten shorthand as a means of rapid data entry. In B. Shackel, editor, *Human-Computer Interaction - Interact '84, IFIP*, pages 145–150, North-Holland, 1985. Elsevier Science Publishers B.V.

[78] Barbara Staudt Lerner. *Automated Customization of User Interfaces*. PhD thesis, Carnegie Mellon University, 1989.

[79] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

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

[81] D. J. Lyons. Go Corp. gains ground in pen-software race. *PC Week*, 7(29):135, July 23 1990.

[82] Gale Martin, James Pittman, Kent Wittenburg, Richard Cohen, and Tome Parish. Sign here, please. *Byte*, 15(7):243–251, July 1990.

[83] J. T. Maxwell. Mockingbird: An interactive composer's aid. Master's thesis, MIT, 1981.

[84] P. McAvinney. The Sensor Frame–a gesture-based device for the manipulation of graphic objects. Available from Sensor Frame, Inc., Pittsburgh, Pa., December 1986.

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

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

[87] P. Morrel-Samuels. Clarifying the distinction between lexical and gestural commands. *International Journal of Man-Machine Studies*, 32:581–590, 1990.

[88] G. Muller and R. Giulietti. High quadlity music notation: Interactive editing and input by piano keyboard. In *Proceedings of the 1987 International Computer Music Conference*, pages 333–340, San Francisco, 1987. Computer Music Association.

[89] Hiroshi Murase and Toru Wakahara. Online hand-sketched figure recognition. *Pattern Recognition*, 19(2):147–160, 1988.

[90] B. Myers and W. A. S. Buxton. Creating highly-interactive and graphical user interfaces by demonstration. *Computer Graphics*, 20(3):249–258, 1986.

[91] B. A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 1990.

[92] B. A. Myers, D. Giuse, R. B. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, Andrew Mickish, and Phillippe Marchal. Garnet: comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, Nov 1990.

[93] B. A. Myers, B. Vander Zanden, and R. B. Dannenberg. Creating graphical interactive application objects by demonstration. In *UIST '89 Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 95–104. ACM, November 1989.

[94] Brad A. Myers. A taxonomy of user interfaces for window managers. *IEEE Computer Graphics and Applications*, 8(5):65–84, 1988.

[95] Brad A. Myers. Encapsulating interactive behaviors. In *Human Factors in Computing Systems*, pages 319–324, Austin, TX, April 1989. Proceedings SIGCHI'89.

[96] Brad A. Myers. User interface tools: Introduction and survey. *IEEE Software*, 6(1):15–23, January 1989.

[97] Brad A. Myers. Demonstration interfaces: A step beyond direct manipulation. Technical Report CMU-CS-90-162, Carnegie Mellon School of Computer Science, Pittsburgh, PA, August 1990.

[98] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, March 1990.

[99] L. Nakatani. Personal communication, Bell Laboratories, Murray Hill, N.J. January 1987.

[100] T. Neuendorffer. *Adew Reference Manual*. Information Technology Center, Pittsburgh, PA, 1989.

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

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

[103] L. Norton-Wayne. A coding approach to pattern recognition. In J. Kittler, K. S. Fu, and L. F. Pau, editors, *Pattern Recognition Theory and Applications*, NATO Advanced Study, pages 93–102. D. Reidel, 1981.

[104] K. K. Obermeier and J. J. Barron. Time to get fired up. *Byte*, 14(8), August 1989.

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

[106] PC Computing. Ten other contendors in the featherweight division. *PC Computing*, 2(12):89–90, December 1989.

[107] J. A. Pickering. Touch-sensitive screens: the tecnologies and their application. *International Journal of Man-Machine Studies*, 25:249–269, 1986.

[108] R. Probst. Blueprints for building user interfaces: Open Look toolkits. Technical report, Sun Technology, August 1988.

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

[110] J. Rosenberg, R. Hill, J. Miller, A. Schulert, and D. Shewmake. UIMSs: Threat or menace? In *CHI '88*, pages 197–212. ACM, 1988.

[111] D. Rubine and P. McAvinney. The Videoharp. In *1988 International Computer Music Proceedings*. Computer Music Association, September 1988.

[112] D. Rubine and P. McAvinney. Programmable finger-tracking instrument controllers. *Computer Music Journal*, 14(1):26–41, 1990.

[113] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[114] K. J. Schmucker. MacApp: An application framework. *Byte*, 11(8):189–193, August 1986.

[115] Kurt J. Schmucker. *Object-Oriented Programming for the Macintosh*. Hayden Book Company, 1986.

[116] A. C. Shaw. Parsing of graph-representable pictures. *JACM*, 17(3):453, 1970.

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

[118] John L. Sibert, William D. Hurley, and Teresa W. Bleser. An object-oriented user interface management system. In *SIGGRAPH '86*, pages 259–268. ACM, August 1986.

[119] Jack Sklanksy and Gustav N. Wassel. *Pattern Classifiers and Trainable Machines*. Springer-Verlag, New York, 1981.

[120] W. W. Stallings. Recognition of printed chinese characters by automatic pattern analysis. *Computer Graphics and Image Processing*, 1:47–65, 1972.

[121] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*, 6(4):40–62, Winter 1986.

[122] Jess Stein, editor. *The Random House Dictionary of the English Language*. Random House, Cambridge, Mass., 1969.

[123] Martin L. A. Sternberg. *American Sign Language: A Comprehensive Dictionary.* Harper and Row, New York, 1981.

[124] M. D. Stone. Touch-screens for intuitive input. *PC Magazine*, pages 183–192, August 1987.

[125] C. Y. Suen, M. Berthod, and S. Mori. Automatic recognition of handprinted characters: The state of the art. *Proceedings of the IEEE*, 68(4):469–487, April 1980.

[126] Sun. *SunWindows Programmers' Guide.* Sun Miscrosystems, Inc., Mountain View, Ca., 1984.

[127] Sun. *NeWS Preliminary Technical Overview.* Sun Miscrosystems, Inc., Mountain View, Ca., 1986.

[128] Shinichi Tamura and Shingo Kawasaki. Recognition of sign language motion images. *Pattern Recognition*, 21(4):343–353, 1988.

[129] C. C. Tappert, C. Y. Suen, and Toru Wakaha. The state of the art in on-line handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(8):787–808, August 1990.

[130] E. R. Tello. Between man and machine. *Byte*, 13(9):288–293, September 1988.

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

[132] D. S. Touretzky and D. A. Pomerleau. What's hidden in the hidden layers? *Byte*, 14(8), August 1989.

[133] V. M. Velichko and N. G. Zagoruyko. Automatic recognition of 200 words. *Int. J. Man-Machine Studies*, 2(2):223, 1970.

[134] A. Waibel and J. Hampshire. Building blocks for speech. *Byte*, 14(8):235–242, August 1989.

[135] A. I. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering*, SE-11(8):699–713, August 1985.

[136] D. Weimer and S. K. Ganapathy. A synthetic visual environment with hand gesturing and voice input. In *CHI '89 Proceedings*, pages 235–240. ACM, 1989.

[137] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, August 1962.

[138] A. P. Witkin. Scale-space filtering. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1019–1022, 1983.

[139] C. G. Wolf. A comparative study of gestural and keyboard interfaces. *Proceedings of the Humans Factors Society*, 32nd Annual Meeting:273–277, 1988.

[140] C. G. Wolf and J. R. Rhyne. A taxonomic approach to understanding direct manipulation. *Proceedings of the Human Factors Society*, 31st Annual Meeting:576–580, 1987.

[141] Catherine G. Wolf. Can people use gesture commands? Technical Report RC11867, IBM Research, April 1986.

[142] Xerox Corporation. JUNO. In *SIGGRAPH Video Review Issue 19: CHI '85 Compilation*. ACM, 1985.

THE DESIGN AND EVALUATION OF MARKING MENUS

by

Gordon Paul Kurtenbach

A thesis submitted in conformity with the requirements
of the Degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

This research focuses on the use of hand drawn marks as a human-computer input technique. Drawing a mark is an efficient command input technique in many situations. However, marks are not intrinsically self-explanatory as are other interactive techniques such as buttons and menus. This research develops and evaluates an interaction technique called marking menus which integrates menus and marks such that both self-explanation and efficient interaction can be provided.

A marking menu allows a user to perform a menu selection by either popping up a radial menu and then selecting an item, or by drawing a straight mark in the direction of the desired menu item. Drawing a mark avoids popping up the menu. Marking menus can also be hierarchic. In this case, hierarchic radial menus and "zig-zag" marks are used. Marking menus are based on three design principles: self-revelation, guidance and rehearsal. Self-revelation means a marking menu reveals to a user what functions or items are available. Guidance means a marking menu guides a user in selecting an item. Rehearsal means that the guidance provided by the marking menu is a rehearsal of making the mark needed to select an item. Self-revelation helps a novice determine what functions are available, while guidance and rehearsal train a novice to use the marks like an expert. The intention is to allow a user to make a smooth and efficient transition from novice to expert behavior.

This research evaluates marking menus through empirical experiments, a case study, and a design study. Results shows that (1) 4, 8 and 12 item menus are advantageous when selecting using marks, (2) marks can be used to reliably select from four-item menus that are up to four levels deep or from eight-item menus that are up to two levels deep, (3) marks can be performed more accurately with a pen than a mouse, but the difference is not large, (4) in a practical application, users tended towards using the marks 100% of the time, (5) using a mark, in this application, was 3.5 times faster than selection using the menu, (6) the design principles of marking menus can be generalized to other types of marks.

THE DESIGN AND EVALUATION OF MARKING MENUS

Gordon Paul Kurtenbach
Degree of Doctor of Philosophy
Graduate Department of Computer Science, University of Toronto, 1993

abstract

This research focuses on the use of hand drawn marks as a human-computer input technique. Drawing a mark is an efficient command input technique in many situations. However, marks are not intrinsically self-explanatory as are other interactive techniques such as buttons and menus. This research develops and evaluates an interaction technique called marking menus which integrates menus and marks such that both self-explanation and efficient interaction can be provided.

A marking menu allows a user to perform a menu selection by either popping up a radial menu and then selecting an item, or by drawing a straight mark in the direction of the desired menu item. Drawing a mark avoids popping up the menu. Marking menus can also be hierarchic. In this case, hierarchic radial menus and "zig-zag" marks are used. Marking menus are based on three design principles: self-revelation, guidance and rehearsal. Self-revelation means a marking menu reveals to a user what functions or items are available. Guidance means a marking menu guides a user in selecting an item. Rehearsal means that the guidance provided by the marking menu is a rehearsal of making the mark needed to select an item. Self-revelation helps a novice determine what functions are available, while guidance and rehearsal train a novice to use the marks like an expert. The intention is to allow a user to make a smooth and efficient transition from novice to expert behavior.

This research evaluates marking menus through empirical experiments, a case study, and a design study. Results shows that (1) 4, 8 and 12 item menus are advantageous when selecting using marks, (2) marks can be used to reliably select from four-item menus that are up to four levels deep or from eight-item menus that are up to two levels deep, (3) marks can be performed more accurately with a pen than a mouse, but the difference is not large, (4) in a practical application, users tended towards using the marks 100% of the time, (5) using a mark, in this application, was 3.5 times faster than selection using the menu, (6) the design principles of marking menus can be generalized to other types of marks.

Acknowledgments

Many years ago when I was in high school my classmates and I spent three days writing occupation aptitude tests. Months later the computer graded tests were returned to us. I remember my friends' and my own excitement as we ripped open the envelopes to see what the computer had recommended. My friends cheered as they read out their long list of possibilities: doctor! lawyer! pilot! writer! scientist! With great anticipation I opened my computer recommendation. There, before my eyes, was one lonely recommendation: *pre-cast concrete worker.*

Although I have failed to fulfill my destiny as pre-cast concrete worker, I have created this thesis with the support of many people. In particular, I would like to thank:

- My supervisor and friend, Bill Buxton. Bill's creativity, intellect, and humor inspired me to pursue research and make bad jokes.

- The members of my committee: Ron Baecker, Mark Chignell, Marilyn Mantei, Ken Sevcik, and Cathy Wolf. Each contributed in helping me polish my research into a doctoral thesis.

- Great researchers and friends. Abigail Sellen greatly helped by designing experiments, writing, and putting on excellent parties; Tom Moran, Stuart Card, and Ken Pier provided creative insights and guidance; George Fitzmaurice and Beverly Harrison waded through treacherous drafts of my thesis, helped me make it a better document, and listened to my concerns over many a cappuccino; Gary Hardock utilized my work in his research and put up with my kidding; George Drettakis and Dimitri Nastos kept the lab systems running, humored me, and organized the most delicious Greek barbecues; Tim Brecht advised me, made me laugh way too loud and long, yet still managed to keep me sane.

I don’t think I’ll thank the computer that graded the aptitude tests...

To my parents, Helen and Leo,

and my brother and sisters,

Robert, Beverly, Donna, Carole, and Tammy:

"My thesis is done, you can probably reach me at home now"

Table of Contents

Chapter 1: Introduction

Research in the last forty years has brought great improvements in the quality of human-computer interactions. In the past, human-computer dialogs were optimized for the computer; humans communicated with computers using protocols that were easy for the computer to understand but were hard for a human to understand and use, for example, machine languages. Advances in human-computer interaction have changed this situation. Controlling a computer no longer requires memorizing obtuse, cryptic codes or an intimate understanding of the internal workings of the computer. In well-designed systems, human-computer interactions are optimized for the human. Interfaces now make use of sophisticated graphics, sound, and pointing devices to make the human's job easier.

The major advances in human-computer interaction have been in making computers easier to use. Specifically, research on methods to reduce the amount of training a person needs before being able to operate a computer has come a long way. For example, the *Apple Macintosh* has set standards for the minimal amount of instruction that a person needs before operating a computer. Because of these advances, the world of computers opened up for people who otherwise would not have invested the time in training to operate a computer system.

Given these advances in human-computer interaction, we can think of the interface as currently being optimized for the human, specifically, the novice computer user. Clearly, this is of great value, but we can consider another important class of user—the expert. Human capacity for the development of skills is great. Virtuoso pianists are proof of this. Virtuosos invest a great deal of time in practicing their skills—eight hours of practice a day is not uncommon. Now consider expert computer users. It is not uncommon for an expert computer user to spend eight hours a day working on the computer. Therefore, there is untapped potential for human skill development in human-computer interactions. A good interface should take advantage of this potential and not limit the efficiency of a skilled user.

In order for this skill potential to be tapped, an interface must have certain properties. First, the interface must provide interaction methods that are suitable for an expert. Experts require efficient interactions. As a result, interactions may be terse and unprompted. Second, and most critically, the interface must also provide support for a novice to become expert. We look at the interface design not so much as making the interface easier to use but rather as *accelerating the rate at which novices begin to perform like experts*. This goal demands three components: support for the novice, support for the expert, and an efficient mechanism to support the transition from novice to expert (see Figure 1.1).

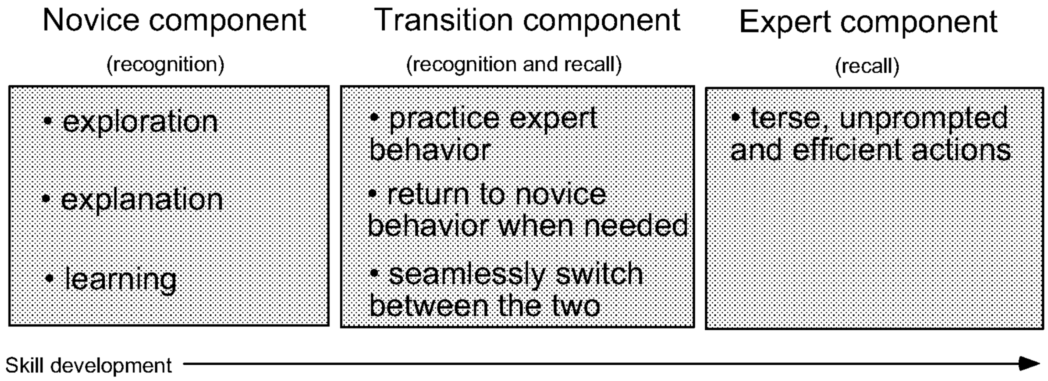


Figure 1.1: The components required to accelerate the rate at which users begin to perform like experts. The novice component allows a user to issue commands by searching for them and recognizing them. The expert component allows a user to efficiently issue commands by recalling the action associated with the command. The transition component allows a user to efficiently switch between these two methods to learn and practice command action associations.

In this dissertation, we focus on an interaction technique that is intended to take advantage of this skill potential and support the development of skill. We propose an interaction technique which has a two modes. In the first mode, the style of interaction is intended to facilitate novice use. In the second mode, the style of interaction is intended for skilled expert behavior. The first mode is also designed to allow a novice to practice the skills required in the second mode. A user can switch to the second mode by operating the technique quickly. One can think of this in metaphorical terms. When you are learning to drive a car, its suitable to have a car that is designed for a student driver. However, as your driving skills improve, the car incrementally transforms into a Ferrari.

1.1. GENERAL AREA AND DEFINITIONS

To support the expert component described in the previous section, we focus on a style of human computer interaction in which a user "writes" on the display surface. This style of interaction is similar to writing or drawing with a pen on ordinary paper. Writing on a display, however, is accomplished with a special pen and the computer simulates the appearance of ink.[1]

We define a *mark* as the series of pixels that are changed to a special "ink" color when the pen is pressed and then moved across the display. The pixels that are changed to an ink color are those which lay directly under the tip of the pen as it is moved across the display. Free hand drawings, ranging from meaningless scribbles to meaningful line drawings and symbols, including handwriting, are examples of marks. The act of drawing a mark is referred to as *marking*.

Marks can be created not only with a pen but also with other types of input devices. For example, a mouse can leave a trail of ink (commonly referred to as an *ink-trail*) behind the tracking symbol when the mouse button is pressed and the mouse is dragged. Some systems use a pen and tablet. In this case, marks are made on the display by writing on the tablet instead of the display.

From a user's point of view, these interfaces allow one to make marks and then have the system interpret those marks. There are, however, systems in which marks can be made but not recognized by the system. They are interpreted strictly as annotations, for example, *Freestyle* (Perkins, Blatt, Workman, & Ehrlich, 1989). The focus of this dissertation, however, is on systems in which marks are interpreted as commands and parameters.

Much of the literature refers to marks as *gestures*. However, the term gesture is inappropriate in this context. Indeed creating a mark does involve a physical

[1] The pen, in these types of systems, is sometimes referred to as a stylus.

gesture but the real object of interpretation is the mark itself.[2] For example, the "X" mark requires a completely different physical gesture if performed with a pen instead of a mouse. Gesture is an important aspect of mark because some marks may require awkward physical gestures with the input device. However, the two terms should be distinguished. The term gesture is more appropriate for systems in which the gestures leave no marks, for example, *VideoPlace* (Krueger, Giofriddo & Hinrichsen, 1985). The term mark is more appropriate for pen-based computer systems or applications that emulate paper and pen.

1.2. WHY USE MARKS?

Current human-computer interfaces are asymmetric in terms of input and output capabilities. There a number of computer output modes: visual, audio and tactile. Most computers extensively utilize the visual mode; high resolution images which use thousands of colors of can be displayed quickly and in meaningful ways to a user. In contrast, a computer's ability to sense user input is limited. Humans have a wide range of communication skills such as speech and touch, but most computers sense only a small subset of these. For example, keyboards only sense finger presses (but not pressure) and mice only sense very simple arm or wrist movements. Therefore, we believe the advent of the pen as a computer input device provides the opportunity to increase input bandwidth through the use of marks.[3]

There are two major motivations for using marks. The first addresses the problem of efficiently accessing the increasing number of functions in applications. The second motivation is that there are some intrinsic qualities that marks have which can provide a more "natural" way to articulate otherwise difficult or awkward concepts (such as spatial or temporal information). Both of these motivations will now be examined in more detail.

[2] There are systems where interpretation depends not only on what is drawn but also how it is drawn. For example, an "X" drawn quickly may have a different interpretation from a "X" that is drawn slowly. By this dissertation's terminology, these systems would contain a combination of marking and gesture recognition.

[3] It is ironic that one of the first input devices for graphics was a light pen which wrote directly on the display surface (Sutherland, 1963).

1.2.1. Symbolic nature

The inadequacy of mouse and keyboard interfaces is exemplified by applications that are controlled through button presses and position information.[4] Buttons must be accessible and thus require physical space. Problems occur when an application has more functions than can be mapped to buttons or reasonably managed on the display. Other problems also exist: arbitrary mappings between functions and buttons can be confusing, and user management of the display and removal of graphical buttons can be tedious.

Expert users of these types of systems find the interface inadequate because button interfaces are inefficient. The existence of interaction techniques that override buttons for the sake of efficiency is evidence of this. Experts, having great familiarity with the interface, are aware of the set of available commands. Menus are no longer needed to remind them of available commands and invoking commands through menu display becomes very tedious.

Designers have addressed this problem in several ways. One solution is *accelerators keys* which allow experts direct access to commands. An accelerator key is a key on the keyboard which, when pressed, immediately executes a function associated with a menu item or button. The intention is that using an accelerator key saves the user the time required to display and select a menu item or button. Many systems display the names of accelerator keys next to menu items or buttons to help users learn and recall the associations between accelerator keys and functions.

Another way of supporting an expert is by supplying a command line interface in addition to a direct manipulation interface. Commodore's command line interface, *CLI*, and graphical user interface, *Intuition*, are an example of this approach.

Both these approaches have their problems. In the case of accelerator keys, arbitrary mappings between functions and keys can be hard to learn and remember. Sometimes mnemonics can be established between accelerator key and function (e.g., control-o for "open"), but mnemonics quickly run out as the number of accelerator keys increases. Further confusion can be caused by different applications

[4] The term buttons is used as a generic way of describing menus items, dialog box items, icons, keys on a keyboard, etc., which are typical of direct manipulation interfaces.

using a common key for different functions or by different applications using different keys for a common function. Experts must then remember arbitrary or complex mappings between keys and functions depending on application. Command line interfaces are problematic because they are radically different from direct manipulation interfaces. To become an expert, a novice must learn another entirely different interface.

Marks, because of their symbolic nature, can make functions more immediately accessible. Rather than triggering a function by a button press, a mark can signal a command. For example, a symbolic mark can be associated with a function and a user can invoke the function by drawing the symbol. In theory, because marks can be used to draw any symbol or series of symbols, marks can provide a quicker method of choosing a command than searching for a physical or graphical button and pressing it. In practice, the number of marks is limited by the system's ability to recognize symbols and a human's ability to remember the set of symbols. Nevertheless, even if only a small set of marks are used, a user can invoke the associated functions immediately.

Marks can also be used to hide functions because they are user generated symbols. For example, researchers at Xerox PARC made use of this property when faced with a dilemma during the design of a pen-based application. This application runs on a wall sized display where a user can write on the display using an electric pen (Elrod et. al., 1992). There were two major design requirements. First, the designers wanted the application to look and operate like a whiteboard and maximize the size of the area where drawing could take place. Second, they wanted to provide additional functions commonly found in computer drawing programs. This second requirement meant that many graphical buttons would need to appear on the screen. This, however, violated the first design requirement because the numerous graphical buttons would consume too much of the drawing area and make the interface look complicated.

The design solution was to assign many of the drawing functions to marks. Marks provided a way to hide additional functionality from novices while expert users could use the marks to access additional functions. This design also avoided using buttons for these functions and, in many cases, marks were a much more effective way of articulating a function.

1.2.2. Intrinsic advantages

The advantages of pen input and marks have been expressed in the literature (Bush, 1945; Licklider 1960; Ellis & Sibley, 1967; Hornbuckle, 1967; Coleman, 1969; Ward & Blesser, 1985; Rhyne & Wolf, 1986; Wolf, 1986; Buxton, 1986; Welbourn & Whitrow, 1988; Wolf, Rhyne, & Ellozy, 1989; Morrel-Samuels, 1990; Kurtenbach & Hulteen, 1990). Specifically, marks provide the ability to:

- embed many command attributes into a single mark;

- reduce learning time due to the mnemonic nature of marks and users' existing knowledge of pen and paper marks;

- capture and recognize handwriting and drawing;

- enter different types of data without switching input device. For example, text, menu selections, button presses, and screen locations can be entered without changing input device;[5]

- replace the computer keyboard, thus making computers smaller and more portable;

- maintain a visible audit trail of operations;

- maintain a clear figure/ground relationship (Hardock, 1991). For example, marks written over formatted text can be distinguished from the text.

1.3. SELF-REVELATION, GUIDANCE AND REHEARSAL

Despite all of these advantages, pen input and marks have not been widely used. Pen-based interfaces have many difficult technological requirements. Historically, hardware for pen-based systems was too expensive and recognition was not reliable (Sibert, Buffa, Crane, Doster, Rhyne, & Ward, 1987). Given these limitations pen-based applications presented no advantage (in reality, more of a disadvantage) over a mouse-based version of the application.

[5] This eliminates homing time between physical input devices but it does not eliminate homing time between graphical devices such as graphical buttons, sliders, etc.

This situation is changing and this change is clearly evident in the marketplace (Normile & Johnson, 1990; Rebello, 1990). Several companies such as Go, Grid, IBM, Apple, Microsoft, and NCR are introducing pen-based systems. Hardware and recognition has improved to the point where pen-based systems are technically possible. Applications such as portable notebook computers and large whiteboard size computer screens make the pen an attractive input device (Goldberg & Goodisman, 1991; Weiser, 1991).

On the surface, it appears that once the recognition and hardware problems are solved, pen-based systems will be successful. However, there is still a serious interface problem when using marks.

1.3.1. The problem: learning and using marks

An intrinsic problem with marks is that they are not *self-revealing.* In contrast, menus and buttons are self-revealing; the set of available commands and how to invoke a command is readily visible as a byproduct of the way commands are invoked. An interface which uses only marks as a means of command entry cannot support *walk-up-and-use* situations. A first time user has no way of finding out interactively from the system what marks/commands are available. This situation is reminiscent of command line interfaces such as the *UNIX* shell or *MS-DOS* where the only information presented by the system is a command line prompt. Some source of information distinct from the process of making a mark must be consulted before commands can be generated.

The problem is even more acute. Not only do users need to know what marks can be made but also when or where these marks can be made. In menu and button interfaces, one can find out when and where a command can be invoked by which buttons or menu items appear active when an interface object is selected. Marks do not have this property.

Is there a problem? Aren't the existing pen-driven systems easy to use and self-revealing? Hybrid interfaces which use both direct manipulation and marks (e.g., the *PenPoint* or *Momenta* interfaces (Go, 1991; Momenta, 1991)) may be somewhat capable of walk-up-and-use. However, only the direct manipulation components of the interface can be used without external instruction.[6] Manuals must still be used to find out about marks. Hence these system do not solve the self-revealing/marks problem.

The motivation for creating walk-up-and-use interfaces is strong. Successful computer interfaces such as the Macintosh are based on the notion that "nobody reads manuals". These types of interfaces are designed to help a user learn and remember how to operate the interface without explicit external help such as on-line help or manuals (Sellen, & Nicol, 1990). This situation can be viewed practically: a user wants to get a certain task done; this task can be accomplished using a computer tool; the shortest path between the user and task completion is using the tool; a manual will be consulted only if the tool cannot be used directly.

If we expect a worker in the information age to utilize many different applications, a huge amount of training for each application is an unrealistic demand. Users expect interfaces that are consistent and permit transfer of skills from other applications. They also expect interfaces to be self-explanatory and to guide a user in the operation of the application. Thus, the motivation for walk-up-and-use self-revealing interfaces is paramount.

An argument can be made that walk-up-and-use interfaces are not efficient, but this argument misses the point. The reason to make marks self-revealing is so a user can graduate from using the walk-up-and-use techniques to the more efficient marks. Once this graduation has taken place, the user can benefit from the advantages of marks such as efficient articulation and conservation of screen space. The key to the success of this scheme is in how easily a novice can acquire expert skills.

It can be argued that if marks are mnemonic, then no self-revealing mechanism is needed. However, this argument is analogous to using mnemonic names for commands in command line interfaces. This technique relies on the user "being a good guesser" and it has been shown that they are not; command naming behavior of individuals is extremely variable (Furnas, et al., 1982; Carroll, 1985; Jorgensen et al. 1983; Wixon et al., 1983). The more fail-safe approach is to provide an explicit mechanism which explains the command set (Barnard & Grudin, 1988). On the

[6] Of course, even some of the direct manipulation components may require instruction.

other hand, other researchers have shown or argued that users commonly agree on certain marks for certain operations (Wolf, 1986; Wolf & Morrel-Samuels; Gould, & Salaun, 1987; Buxton, 1990). Nevertheless, if we wish to use marks for operations which do not have commonly agreed upon marks, a mechanism must be provided for learning about these marks.

We define three design principles to support learning and using marks. We do not claim that these principles are unique. Other researchers have described similar general principles, and many systems have interactions which obey these general principles. However, we define specific design principles for two reasons. First, our application of the general design principles to marks is novel, and second, our own specific definitions help us to explain and discuss the details of the application.

The three design principles to support learning and using marks are self-revelation, guidance, and rehearsal.

Self-revelation

> *The system should interactively provide information about what commands are available and how to invoke those commands.*

When an interface provides information to a user about what commands are available and how to invoke those commands, we refer to this as self-revelation or the system being self-revealing. Menus and buttons, for example, are self-revealing. The available commands and how to invoke those commands can be inferred from the display of menus or buttons. Marks, on the other hand, are not self-revealing because they must be generated by the user.

To ensure that every aspect of a system is self-revealing is a difficult task. For example, displaying menu items may help a user understand what functions are available but does not guarantee that the user will understand, from the display, the mechanics of selecting a menu item.

A common approach to interface design, and the approach that we adopt in this dissertation, is to rely on a user receiving a small amount instruction before starting to use the system. These instructions explain the basic mechanics and semantics of operating the interface. For example, pointing, dragging, double clicking, and the meaning of these actions may be explained. The Macintosh computer uses this technique. The intention is that with this small set of skills a user can start interactively exploring and learning about the remainder of the system.

The interaction technique developed in this dissertation uses this type of design. A user must be informed, *a priori*, that in order to display a menu the pen must be pressed against the display and held still for a fraction of second. We call this "press and wait for more information". Once users have this bit of information, however, they receive further instructions interactively from the system. In our model of the interface, users can interactively learn about what functions can be applied to various displayed objects by "pressing and waiting" on the objects for menus.

The principle of self-revelation is based on interface design principles and psychological mechanisms proposed by others. Norman and Draper (1986) propose a design principle to "bridge the gulfs of execution and evaluation". Specifically, a designer should make interface objects visible so users can see what actions are possible, how actions can be done, and the effects of their actions. Shneiderman (1987) proposes a similar principle: "offer informative feedback". The principle states that objects and actions of interest should be made visible to the user. Shneiderman claims that this design principle is the basis of direct manipulation interfaces.

The principle of self-revelation is distinct from affordance theory (Gibson, 1979; Gibson 1982). Self-revelation is concerned with absence/presence of information about what functions are available and how to invoke those functions. Affordance theory, in human computer interaction, is concerned with an interface object's appearance suggesting its function (Gaver, 1991). These two notions, however, are related. For example, consider the display of a pop-up menu. The principle of self-revelation dictates, first, that function names or icons must be displayed, and, second, that they are displayed in a menu so that a user knows by convention how to invoke them. Affordance theory, on the other hand, dictates that the name or icon for an item accurately suggests its function, and that the appearance of the menu suggest items are selectable. Correct use of affordances may help reduce the amount of *a priori* instruction a user requires. For example, items in a menu may "look" selectable (they "afford" selection) and therefore the user does not have to be explicitly taught these mechanics.

Guidance

> *The way in which self-revelation occurs should guide a user through invoking a command.*

If an interface actually assists a user in the articulation of commands we refer to this as guidance. For example, in the editor *emacs*, by hitting a "command completion key" while typing a command, emacs will display all the command names that match the partially completed command. In effect, emacs "guides" a user in completion of the command, as opposed to waiting for the command to be completely typed before examining its validity. Another example is selection from a hierarchic menu. In this case, selection of an item guides a user to the next menu.

Guidance does not necessarily have to be triggered by the user. Some on-line help systems prompt the user with information to guide them through a command. The critical point is that in these systems getting or receiving helpful information on how to invoke a command (guidance) does not interrupt the articulation of a command. On the other hand, a system like the on-line manual pages in UNIX violates the principle of guidance. In this case, in order to receive information about what commands are available and how to invoke those commands, a user must terminate or at least suspend the act of invoking a command.

Rehearsal

> *Guidance should be a physical rehearsal of the way an expert would issue the command.*

Rehearsal is the notion of designing interactions such that the physical actions made by a novice in articulating a command are a rehearsal of the actions an expert would make invoking the same command. The goal of rehearsal is to develop skills in a novice that transfer to expert behavior. It is hoped that this leads to an efficient transition from novice to expert.

Many interaction techniques support rehearsal. When the basic action of the novice and the expert are the same for a particular function we can say that rehearsal takes place. For example, novices may draw lines, move icons, or select from menus using the same actions as an expert when there is one and only one way of issuing the command. In many cases, the single way of issuing the command may be suitable for both the novice and expert.

There are also many situations, however, where a single method for invoking a command is not sufficient. The popularity of accelerator techniques is proof of this. Typically, good interfaces provide two modes of operation. The first mode, designed for novices, is self-revealing. Conventional menu-driven interactions are an example of this. The self-revealing component of this mode is emphasized over efficiency of interaction because novices are more concerned with how to do things rather than how quickly things can be done. The second mode, designed for experts, typically allows terse, non-prompted interactions. Command line interfaces and accelerator keys are examples of this mode. However, usually there is a dramatic difference between novice and expert behavior at the level of physical action. For example, a novice uses the mouse to select from a menu whereas an expert presses an accelerator key.

The intention of the three design principles is to reduce this discrepancy in action without reducing the efficiency of the expert and ease of learning for the novice. The basic actions of the novice and expert should be the same. It is hoped that as novice performance develops the skills that lead to expert performance will develop in a smooth and direct manner.

1.3.2. Unfolding interfaces

The principles of self-revelation, guidance and rehearsal support the notion of an *unfolding interface*. An unfolding interface works as follows. Initially, a novice is provided with a small amount of information about how to get information on parts of the interface. For example, double clicking on an object may open it up or "unfold" it to reveal additional functions. Thus, given this key to unfolding objects, a user can explore the interface, learning and using new functions. The intention is that, with experience, exploration and use leads to expert knowledge of the system.

There are other schemes which control the number and types of functions available to a user, for example, *Training Wheels* (Carroll & Carrithers, 1984). These types of systems provide explicit novice/expert modes in which the novice mode has fewer functions than the expert mode. The intention is to avoid confusing a novice with a large set of complex functions. Once the reduced set of functions is mastered, the novice can switch to the larger "expert" set of functions. The major difference between this approach and the notion of an unfolding interface is that an unfolding interface has no explicit novice and expert modes. An unfolding interface allows users to incrementally add functions to their repertoire.

Marks, self-revelation, guidance and rehearsal can play important roles in an unfolding interface. Unfolding is essentially an inefficient operation. As suggested earlier, by associating marks with "hidden" functions, unfolding can be avoided. For example, rather than double clicking on an object to unfold it and then clicking on a function button, a mark can be made on the object to invoke the function. To help users learn the marks associated with functions, it would be beneficial if unfolding a function also revealed its mark. This is an application of the principle of self-revelation. Ideally, we want the principles of guidance and rehearsal to hold as well; we want to design an interface such that exploration is equivalent to invoking commands, and exploration allows a novice to practice skills that lead to expert behavior.

1.3.3. Solution: ways of learning and using marks

The concerns of this research are interfaces that use marks but are also self-revealing. Therefore, solutions for making marks self-revealing can be classified by how tightly coupled the act of marking is with the act of getting information about command/mark associations.

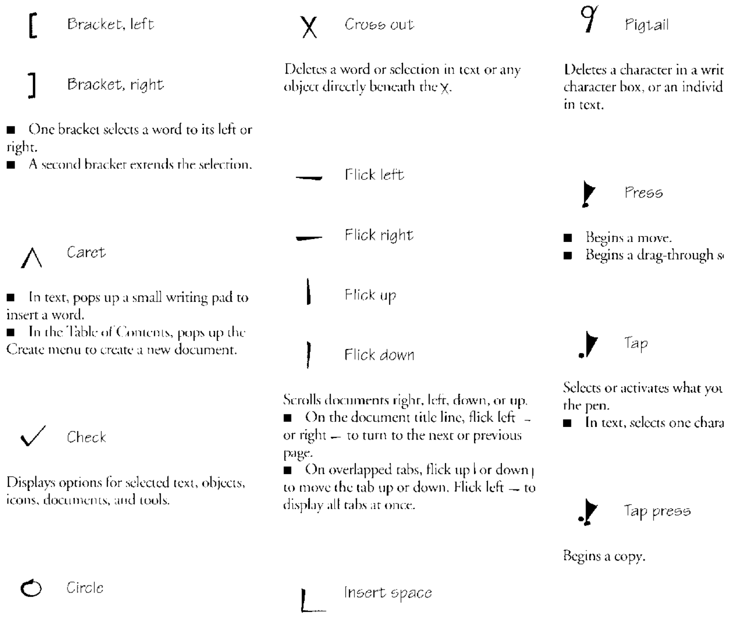
Interfaces that use marks and only supply information about those marks through off-line manuals are considered to be at one end of a self-revelation continuum. These interfaces are not interactively self-revealing. Interfaces which supply information about marks as a command is actually being articulated can be considered the other end of the self-revelation continuum. These would be considered interactively self-revealing interfaces.

In the following sections we classify solutions based on this criterion. Since interfaces that use marks are still in their infancy there are few pre-existing examples.

Off-line documentation

Off-line documentation consists of manuals which provide information about how marks are used in an interface. Examples of the marks are displayed and text or graphics provides information on their usage. Although this type of scheme is not self-revealing it is of interest because, first, it is the status quo for pen-based products and, second, it demonstrates the type of information needed for a user to understand marks.

Figure 1.2 shows a section from a pen-based system's manual. Clearly this type of scheme is not interactively self-revealing. However, if the mark set is small, the documentation could be placed directly on the computer in the form of a "cheat sheet". This scheme would be partially self-revealing.



[Bracket, left

] Bracket, right

- One bracket selects a word to its left or right.
- A second bracket extends the selection.

∧ Caret

- In text, pops up a small writing pad to insert a word.
- In the Table of Contents, pops up the Create menu to create a new document.

✓ Check

Displays options for selected text, objects, icons, documents, and tools.

Circle

X Cross out

Deletes a word or selection in text or any object directly beneath the X.

Flick left

Flick right

Flick up

Flick down

Scrolls documents right, left, down, or up.
- On the document title line, flick left – or right – to turn to the next or previous page.
- On overlapped tabs, flick up | or down | to move the tab up or down. Flick left – to display all tabs at once.

Insert space

Pigtail

Deletes a character in a writ character box, or an individ in text.

Press

- Begins a move.
- Begins a drag-through s

Tap

Selects or activates what you the pen.
- In text, selects one chara

Tap press

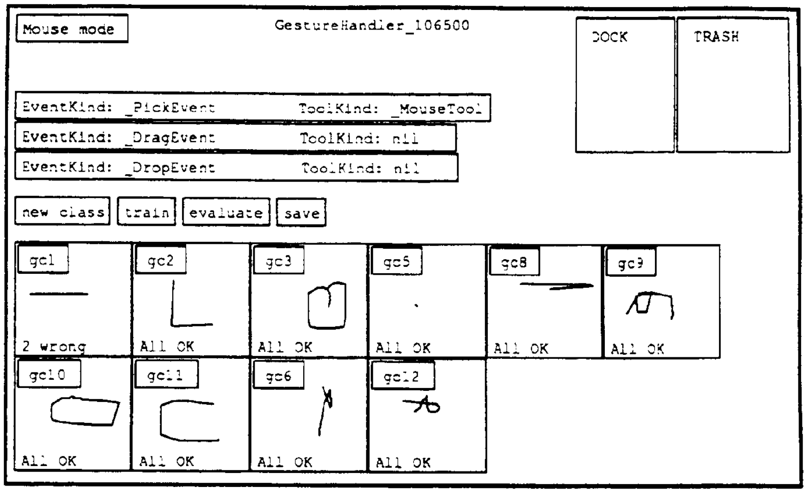
Begins a copy.

Figure 1.2: Typical off-line documentation for mark commands (PenPoint system, Go, 1991)

On-line documentation

This class is essentially the "on-screen" version of off-line documentation. A user can display manual pages on-screen while the application in question is running. Note that this does constrain the user into suspending the real task of issuing a command while obtaining command information.

Sometimes command information can be found in the application used to train the software module that recognizes marks. Figure 1.3 shows one such example.



EventKind: _PickEvent ToolKind: _MouseTool
EventKind: _DragEvent ToolKind: nil
EventKind: _DropEvent ToolKind: nil

Figure 1.3: Gesture handler window allows inspection of marks associated with a view in Rubine's system. This window is, however, intended for the system programmer. The window shows ten classes of marks but does not shows the semantics associated with each mark. (from Rubine, 1990).

Unfortunately, training interfaces are not designed specifically to deliver this type of information, and the information can be very minimal and confusing to the user.

Microsoft's *Windows for Pen Computing* uses on-line documentation. A special application provides a tutorial which features animations demonstrating marks and editing operations. A user can also practice using the marks on sample text. While the tutorial is effective, a user still has to change context (i.e., switch from the working application to the tutorial application) in order to get information on marks.

On-line interactive methods

On-line interactive methods supply information about marks as one issues a command. Figure 1.4 shows an example where sample marks are displayed beside menu items. Windows for Pen Computing using this technique to a limited degree. This technique relies initially on another interaction method such as menus or buttons to invoke commands. In Figure 1.4, the interaction technique initially relied on is a menu. As the menu is used, it reveals the marks that can be used. Once a user remembers the mark associated with a command, the revealing technique (the menu) can be bypassed and a more efficient mark can be used. Figure 1.5 shows a system called *XButtons* which also uses this method. In contrast to on-line documentation, an on-line interactive method does not constrain the user into suspending the real task of issuing a command, while obtaining command information.

This method is similar to accelerator keys. Every time a user uses a menu item or button, the mark is seen. Like accelerator keys, the mark can be memorized and used as a shortcut in calling the command. Note that "accelerator marks" are more powerful that accelerator keys because they are not limited to characters on the keyboard, they indicate the object of the requested action by the location of the mark, and they can contain command attributes, such as destinations or modifiers.

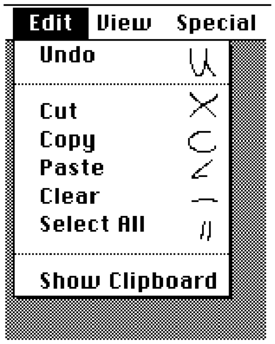


Figure 1.4: An example of "accelerator marks" which allow quick access to menu items similar to accelerator keys.

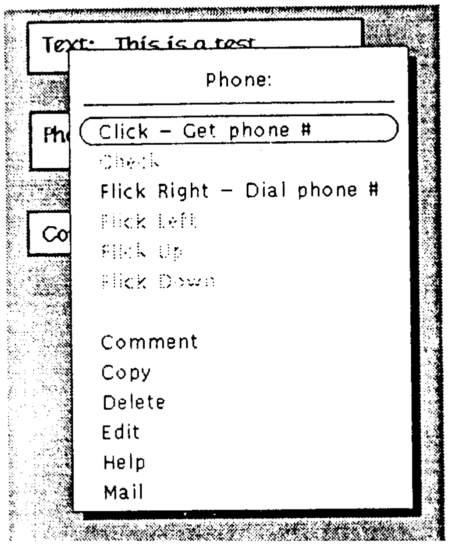


Figure 1.5: XButtons provides a menu which shows what commands are available from a button and the associated marks. A command can be invoked by either a menu selection or by making the mark on the button (Robertson, et al, 1991).

On-line interactive rehearsal methods

This category is similar to on-line interactive methods except invoking a command using the self-revealing technique (i.e., a menu) makes the user physically rehearse making the corresponding mark. In contrast, when using on-line interactive methods, the user does not physically rehearse making the mark (e.g., selecting "copy" from the menu in Figure 1.4 requires a vertical movement, not a hand drawn "C" movement).

Marking menus, the technique focused on in this dissertation, is an example of this class (Kurtenbach & Buxton, 1991). The complete definition of this technique is given in Chapter 2. Figure 1.6 illustrates this technique in the context of creating three simple objects. An expert uses simple shorthand marks to create and place circles, square, or triangles.

If a user is unsure of what marks can be made, the user presses the pen against the display and waits for approximately 1/3 of a second. This signals to the system that no mark is being made and it then prompts the user with a radial menu of the available commands, which appears directly under the cursor. The user may then select a command from the radial menu by keeping the pen tip pressed and making a stroke towards the desired menu item. This results in the item being highlighted (see Figure 1.7). The selection is confirmed when the pen is lifted from the display.

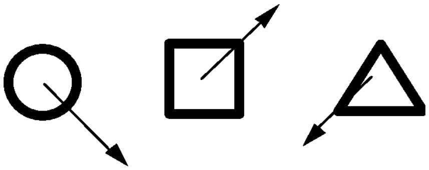


Figure 1.6: An example of the technique using three simple shorthand marks. Three objects can be defined: a circle, square and triangle. A mark which is a simple straight line (shown here with an arrowhead to indicate drawing direction) defines the type of object created, and its placement.

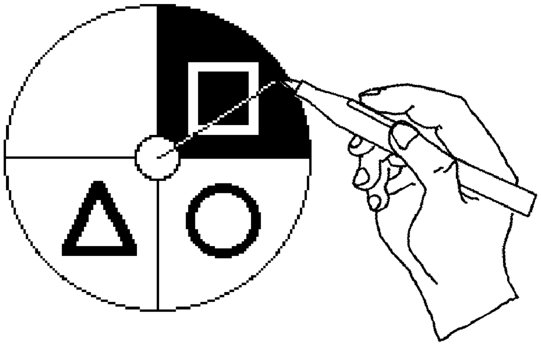


Figure 1.7: A radial (or "pie") menu can also be popped up if the user does not know what commands or marks are available. Rather than drawing a mark as in Figure 1.6 a novice keeps the pen pressed and a menu appears. An object can then be selected from the menu.

The important point is that the physical movement involved in selecting a command is identical to the physical movement required to make the mark corresponding to that command. For example, a command that requires an up-and-to-the-right movement for selection from the pie menu, requires an up-and-to-the-right mark in order to invoke that command. The intention is that selection from the menu is a rehearsal of making a mark.

Other menu layouts can be used for interactive rehearsal methods besides radial menus. Another possibility is a "bull's eye menu" which is a menu that is divided into concentric circles rather than sectors, where each concentric circle corresponds to a different command (Figure 1.8).[7] The corresponding marks are therefore discriminated by length rather than angle. Many more exotic schemes have been proposed and are as of yet unexplored.[8] Chapter 2 presents the motivation for choosing radial menus, and describes in detail the design of marking menus.

[7] We thank Professor John W. Senders for this suggestion originally called "donut menus". Professor William Buxton later took great exception to the use of the word "donut" and suggested the more dramatic name of "bull's eye menu".

[8] Dr. Tom Moran has proposed a combination of donut and pie menus. Dr. Stuart Card has proposed a continuous version of hierarchical marking menus.

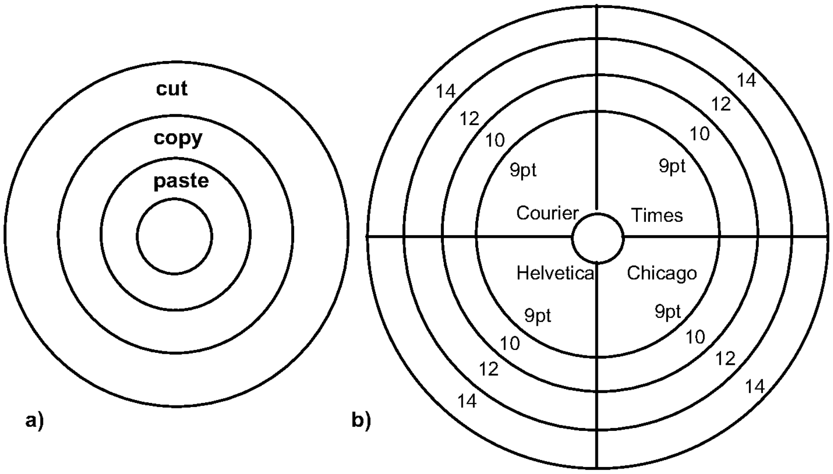


Figure 1.8 Examples of alternate menu styles in which selection will result in a unique marks. a) is a "bull's eye" menu which discriminates by mark length rather than angle. b) is a "dart board" menu which discriminates by length and angle.

1.4. THESIS STATEMENT

This dissertation is an in-depth investigation of marking menus. We present the thesis that marking menus are a valuable interaction technique. When used in the proper situation, marking menus are easy and efficient to use, can be used with different input devices, and integrate well with existing interface techniques. Furthermore, marking menus allow a user to take advantage of writing skills with a pen and attain levels of performance not possible with other interaction techniques. To support this thesis, we present a design for marking menus, evaluate marking menus by means of user behavior experiments, and provide a case study of marking menus in practice. We conclude our investigation by showing how the design concepts of marking menus, self-revelation, guidance, and rehearsal, can be generalized to other situations.

The intention of this investigation is to provide practical guidelines for interface designers interested in using marking menus. With this in mind, we describe when and where marking menus would be an effective technique, and the limitations and properties that must be observed and maintained for marking menus to work well in an interface. We also describe the design principles behind marking menus and give examples of how these principles can be applied to other contexts.

1.5 SUMMARY

This chapter has provided motivation for marks as an interaction technique, described a basic interface problem with marks, set out design principles to solve this problem and introduced an approach, marking menus, which observes these design principles. In Chapter 2 we expand on our motivation for using marking menus and explain in detail the design and design rationale behind marking menus. Chapter 3 reports on an empirical study of the non-hierarchic marking menus. Chapter 3 is a condensed version of a paper that appears in *Human Computer Interaction* (Kurtenbach, Sellen, & Buxton, 1993). Chapter 4 is a case study which reports on how marking menus can be designed into an application and investigates user behavior with marking menus in an "everyday work" situation. Chapter 5 presents an empirical study on the limits of user performance with hierarchic marking menus. Chapter 5 is an expanded version of a paper published in *The Proceedings of InterCHI '93* (Kurtenbach & Buxton, 1993). Chapter 6 describes how we integrated marking menus into a pen-based application and applied the notions of self-revelation, guidance and rehearsal to this application. Chapter 7 summarizes this dissertation and its contributions, and proposes future research.

Chapter 2: Marking menus

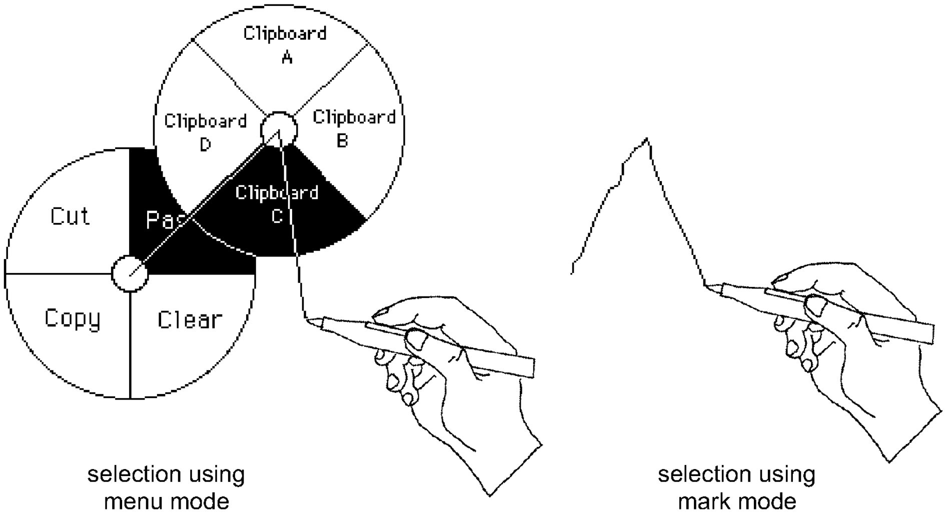
In this chapter we expand on our description of marking menus. First, we present a definition of marking menus and the motives for investigation. Next, we describe previous research that is related to marking menus and we identify open research questions and the issues pursued in this dissertation. Finally, we complete our description of marking menus by providing the complete rationale behind our design.

2.1. DEFINITION

A *marking menu* is an interaction technique that allows a user to select from a menu of items. There are two basic ways (or modes) in which a selection can be performed:

menu mode In this mode a user makes a selection by displaying a menu. A user enters this mode by pressing the pen against the display and waiting for approximately 1/3 of a second. We refer to this action as *press-and-wait*. A *radial menu* of items is then displayed centered around the pen tip. A radial menu is a menu where the menu items are positioned in a circle surrounding the cursor and each item is associated with a certain sector of the circle. A user can select a menu item by moving the pen tip into the sector of the desired item. The selected item is highlighted and the selection is confirmed when the pen is lifted from the display. (See Figure 2.1)

mark mode In this mode, a user makes a selection by drawing a mark. A user enters this mode by pressing the pen against the display and immediately moving in the direction of the desired menu item. Rather than displaying a menu, the system draws an ink-trail following the pen tip. When the pen is lifted, the item that corresponds to the direction of movement is selected. (See Figure 2.1)



selection using menu mode

selection using mark mode

Figure 2.1: The two basic ways of selecting from a marking menu.

The key concept of marking menus is that the physical movement involved in selecting an item in menu mode mimics the physical movement required to select an item using a mark.

Marking menus may also be hierarchic. In menu mode, if a menu item has a subitem associated with it, rather than lifting the pen to select the item, the user waits with the pen pressed to trigger the display of the submenu. The submenu is also a radial menu. The user can then select an item from the submenu in the manner previously described. In mark mode, a user makes a selection by drawing a mark where changes in direction correspond to selections from submenus. Figure 2.1 show an example of selecting from hierarchic menus using menu mode and mark mode.

Using radial menus in this way produces a set of mark which consist of a series of line segments at various angles ("zig-zag" marks). Marking menus which have no hierarchic items produce strictly straight line segments. Figure 2.2 shows an example of a menu hierarchy and the associated marks.

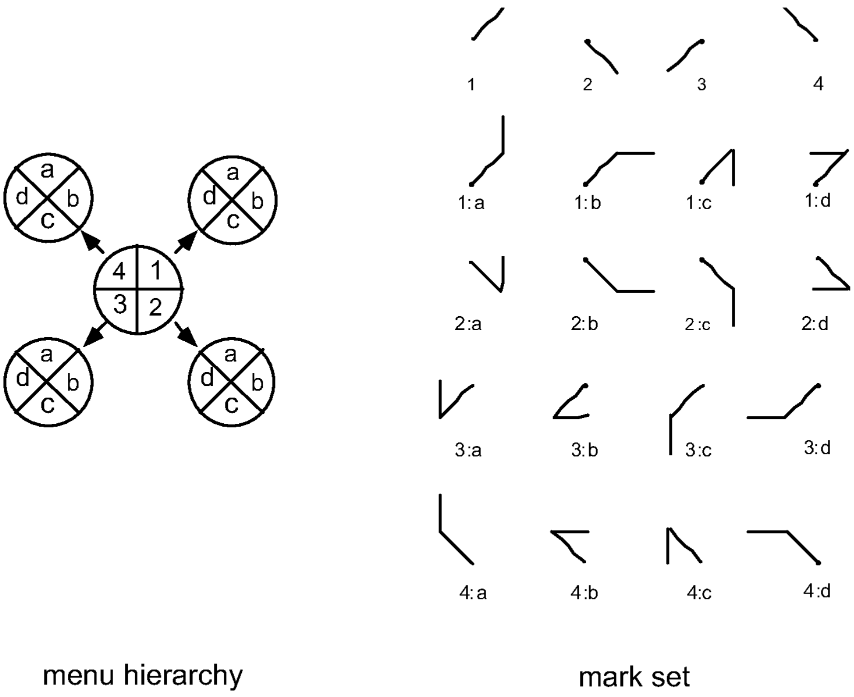


Figure 2.2: An example of a radial menu hierarchy and the marks that select from it. Each item in the numeric menu has a submenu consisting of the items a, b, c and d. A mark's label indicates the menu items it selects. A dot indicates the starting point of a mark.

It is also possible to verify the items associated with a mark or a portion of a mark. We refer to this as *mark confirmation.* In this case a user draws a mark but presses-and-waits at the end of drawing the mark. The system then displays radial menus along the mark "as if" the selection were being performed in menu mode. Figure 2.3 shows an example of this.

Other types of behavior can occur when selecting from a marking menu such as backing-up in a menu hierarchy or reselecting an item in menu mode. Details of the behavior are discussed in Section 2.5.

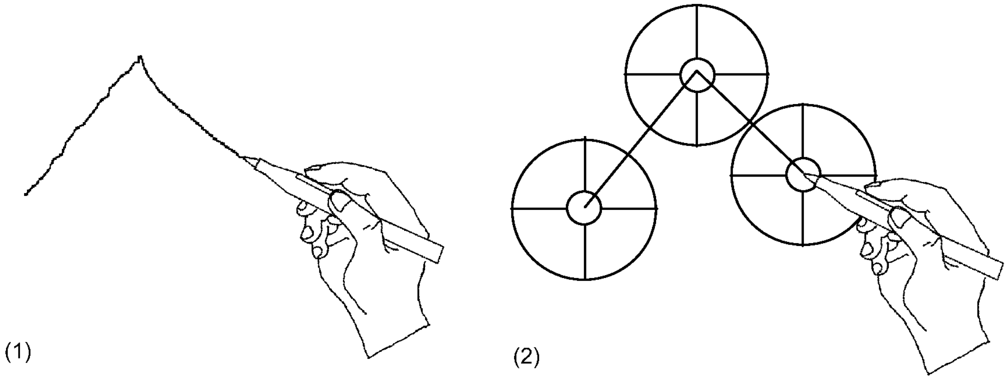


Figure 2.3: An example of mark-confirmation in a menu with three levels of hierarchy. In (1), the user draws the first part of the mark then waits with the pen pressed for the system to recognize the selection so far. In (2), the system then displays its interpretation of the mark and goes into menu mode for completion of the selection.

2.2. MOTIVATION FOR STUDY

We have many motives for studying marking menus; they have advantages over traditional menus; they use marks that are easy to draw and that are easy for computer to recognize; they can be used for functions that have no intuitive mark; they are compatible with different interface styles; and they exploit human motor skills. In this section, we expand on these motivations.

2.2.1. Advantages over traditional menus

One motivation for studying marking menus is that they have many differences and potential advantages over the traditional menus used in current practice. Examples of the current practice in menu design are the pop-up menus or pull-down menus on the Macintosh. With these types of menus, selection is performed by popping up the menu and selecting items by pointing with the mouse. Menu items can also be selected by pressing an accelerator key associated with a menu item. There are several specific advantages marking menus have over these traditional menus:

Keyboardless acceleration

Marking menus allow menu selection acceleration without a keyboard. With traditional linear menus, keypresses must be used to accelerate selection. Marking menus provide a method of accelerating menu selections when no keyboard is available. This is extremely important for portable, keyboardless, pen-based computers.

Acceleration on all items

Marking menus, if configured accordingly, can permit acceleration on all menu items. With traditional menus, it is common for the application developer to assign accelerator keys to the most frequently used menu items. This assumes that the application designer is able to predict the most frequently used menu items. In many cases, however, it is not possible to accurately predict which menu items will be frequently used, if there is a large variance in the way an application may be used. In contrast, with marking menus, the selection of all items can be accelerated by the user making a mark. The designer does not have to predict, *a priori*, which items will be the most frequently used.

Menu selection mimics acceleration

Marking menus minimize the difference between the menu selection and accelerated selection. Selecting a menu item from a marking menu physically mimics the act of making the accelerating mark. The design intention is to help users become skilled at the movements required for accelerated menu selection. This is dramatically different from traditional menus and accelerator keys where menu selection is performed with the mouse and accelerated selection is performed with the keyboard. In this case selection from the menu in no way physically mimics selection using an accelerator key.

Combining pointing and selecting

Marking menus permit pointing and menu selection acceleration with the same input device. This is an intrinsic property of marks and has been utilized by other researchers (e.g., Coleman, 1969; Rhyne 1987; Wolf & Morrel-Samuels, 1987). In mouse-based direct manipulation interfaces it is very common to point to an object and then select a menu item. If accelerator keys are used, this operation requires coordinating pointing with the mouse and pressing on the keyboard. With a marking menu, not requiring a hand to be on the keyboard frees the hand to control other input devices or perform auxiliary tasks such as controlling a VCR transport or turning the pages of a book.

Spatial mnemonics

Marking menus use a spatial method for learning and remembering the association between menu items and marks. In contrast, traditional menus and accelerator keys, rely on symbolic mnemonics to help users remember the associations between menu items and keys. Due the limited number of symbols on a keyboard, mnemonics often cannot be established between all menu items and their accelerators keys. This results in menu item/key associations that may be arbitrary or inconsistent. Marking menus avoid this problem by relying on a consistent method to establish mnemonics: the shape of a mark corresponds to the spatial layout of a menu item in the menu hierarchy.

2.2.2. Ease of drawing and recognition

Marking menus use a very simple set of marks consisting of straight and zig-zag marks. This simple set of marks has three advantages. First, these types of marks are easy and fast to draw and are therefore suitable for accelerated performance. Ease of drawing is especially important when drawing precision is hampered by imperfect pen/display technology. Second, computer recognition of these types of marks can be reliable, fast and user independent. The recognizer requires little processing power and no training. Third, any interface designer, by using marking menus, can make use of some of the advantages of marks without having to design their own mark symbols. Of course, it is still necessary to design the layout of the menus.

The single contiguous marks in marking menus have several advantages. Other types of marks which require multiple non-contiguous pen strokes create many problems. Recognizer design is more complicated when groups of strokes must be recognized. This is referred to as the *segmentation problem*. Sometimes groups of strokes are distinguished by constraining the user to put all the strokes associated with a mark in a certain region. Alternatively, strokes may be grouped by time. This constrains the user to momentarily pause between making different marks. With a marking menu mark, a user is not constrained by timing, size of mark, or location. Recognition takes places the moment the pen is lifted.

The marking menu mark set does have disadvantages. First, a designer has no choice in the shape of the marks (besides what can be controlled through the layout of the menus). Fortunately, marking menus do not prohibit the use of other mark sets and mark recognition techniques (see Chapter 6 for a detailed discussion of this issue). Second, the size of the mark set is limited by a user's accuracy at drawing lines at various angles. Third, the mark set is not particularly expressive. The angle at which the stroke is drawn is used to define the type of mark. The line must also be somewhat straight. This leaves starting point, ending point and temporal information about how the line was drawn to be used as additional information encoding parameters. In contrast, other mark vocabularies permit many more parameters to be controlled by the shape of the mark (Makuni, 1986). Nevertheless, we have discovered that the limited set of parameters of a marking menu mark can be quite useful (see Chapter 4).

2.2.3. Marks when no obvious marks exists

Researchers have shown or argued that users commonly agree on certain marks for certain functions (Wolf, 1986; Gould, & Salaun, 1987; Morrel-Samuels, 1990; Buxton, 1990). However, we believe that there are many situations where invoking a function with a mark could be beneficial but no commonly agreed upon mark exists for the function. This is similar to icon design where some functions have no intuitive icon. For example, there is no "natural mark" for "change pen width to thin". Marking menus might work well in these types of situations because the menu can provide textual or pictorial explanations of functions while the mark for the menu item provides a quick way to invoke the function.

2.2.4. Compatibility with unfolding interfaces

Marking menus are compatible with unfolding interfaces (described in Section 1.3.2). The intention is that menus pop up to self-reveal or unfold functions and the marks provide way to efficiently invoke the functionality. Guidance and rehearsal are intended to help a novice learn the efficient way of invoking a function.

2.2.5. Compatibility with existing interfaces

Marking menus are compatible with popular input devices and interface paradigms. First, the type of marks used can be reasonably drawn with a mouse (Chapters 3 and 5 explore this issue in detail). Second, since traditional menus are created by the application calling library routines, by replacing the library routines, marking menus could be used in place of pop-up menus without changing a single line of application code or changing application functionality. Finally, marking menus can extend existing dialogue styles without major changes to an interface paradigm. An example of this is *HyperMarks*, developed by the author (Kurtenbach & Baudel, 1992), which is a *Hypercard xcommand* that supports marking menus in Hypercard (Apple Computer, 1992). When a marking menu is used from a Hypercard button, the Hypercard button still retains its single function when pressed. However, if the button is kept pressed, a marking menu pops up with more commands. A user can select from the marking menu using menu mode or marks. In this way, the function of a button can be extended.

Marking menus can be effective because they are a pop-up interaction technique. When displays become small or very large, marking menus can be effective. On large displays, a mark or a menu selection can be made at a user's current location without a long trip to a menu bar or tool pallet. On small screens, since both the menu and mark "go-away" once performed, no valuable screen space is consumed.

2.2.6. Novices, experts, and rehearsal

Marking menus are intended to support both the novice and expert user. The intention is that a novice uses menu mode and an expert uses the marks. Menu mode can provide the self-revelation and guidance needed for a novice to invoke a command. The marks can provide efficient interactions for experts.

Marking menus are also intended to support the transition between novice and expert. Selection in menu mode provides the user with rehearsal for making a mark. In essence, using the menu trains a novice to use marks. We believe that rehearsal helps in learning the association between mark and command.

There are other menuing schemes which support the novice and expert and the transition between the two. For example, the Macintosh supports novices by providing menus and supports experts by providing menu accelerator keys. The transition between novice and user is supported by the user being reminded of the keystrokes associated with particular menu items every time a menu is displayed. This is done by having the names of the accelerator keys appear next to menu items in the menu. However, actually using an accelerator key is avoidable. The user can always just select from the menu. Furthermore, this is easiest because the user is already displaying the menu. The end result is that accelerator keys are sometimes not used even after extensive exposure to the menu. With marking menus the user is not only reminded, but rehearses the physical movement involved in making the mark every time a selection from the menu is made. What makes marking menus unique from the accelerator key scheme is that rehearsal is unavoidable. We believe this helps in learning the association between mark and command.

2.2.7. Utilizing motor skills

The idea of using physical rehearsal to train novices to become experts is a unique concept and is worth investigating for pedagogical reasons. Marking menus purport to reduce the cognitive load of memorizing mark/command association by relying on muscle memory (since each mark/command is a distinct physical movement). This technique is similar to the approach used in the Information Visualizer Project (Card et al, 1991). The Information Visualizer relies on low level sensory input processing such as depth or motion perception to reduce the burden on higher cognitive processes in visualizing information. Marking menus can be thought of in a similar manner. It is believed that low level sensory output processes (muscle memory) are used to reduce the load on higher level cognitive processes. We explore this issue in this dissertation.

2.2.8. "Eyes-free" selection

Selection by a distinct physical movement with a marking menu lends itself to "eyes-free" selection. For example, most of us can draw the eight directions of a compass without looking. Eyes-free selection is useful in situations where a user's visual attention must be on something other than the selection process, for example, selecting commands while watching a video tape. An eyes-free selection technique is also extremely valuable to the visually impaired.

2.3. RELATED WORK AND OPEN PROBLEMS

This dissertation develops and explores the use of marking menus. There is no previous research on this technique, *per se*, however, marking menus are based on radial menus (see Section 2.1 for the definition of radial menus). Therefore, research on radial menus is relevant. The most widely used instance of a radial menu is the pie menu (Hopkins, 1991). A pie menu is a radial menu where the visual representation of the menu resembles a sliced pie. Other types of visual representations are possible, for example, we have developed an alternative representation for a radial menu which does not look like a pie (see Figure 2.12).

Two instances of radial menus are pie menus and command compasses. We now describe these two techniques, contrast them with marking menus, and report on the current state of research on their design and usage.

2.3.1. Pie menus

To date, there is little research on pie menus. The origin of pie menus can be traced back to radial menus proposed by Wiseman, Lemke, & Hiles (1969). Since then, research on pie menus has mainly been concerned with menu layout and suitable applications (Hopkins, 1991; Hopkins, 1987). The only empirical study of pie menus investigated menu item selection time and error rates for 8-item menus but concentrated on comparing them to linear menus (Callahan, Hopkins, Weiser, & Shneiderman, 1988). It was found that selection from pie menus was significantly faster (15%) and produced marginally significant fewer errors (42%) than linear menus. The experiment also investigated the effect of using menu items with a natural linear ordering (i.e., "First", "Second", "Third", etc.), with a natural radial ordering (i.e., "North", "North-east", "East", etc.), and with an unclassifiable ordering (i.e., "Center", "Bold", "Italic", etc.). Callahan et al. hypothesized that certain types of menus (pie or linear) would perform better with items that have a certain type of natural ordering (radial, linear, or unclassified). A marginally significant correlation was found between menu types and types of orderings. The weak correlation occurred because selection time means for the pie menus were lower even on items with natural linear orderings. Results also showed that unclassified menu items produced significantly slower selections than ordered menu items regardless of menu type.

What has not been extensively studied is the claim that muscle memory for different gestures plays a helpful role in menu selection. Anecdotal evidence from designers of pie menu systems suggest that item selection from a menu hierarchy is possible without displaying the menus after practice (Hopkins, 1987). Not only was unprompted selection possible but it was also desirable for efficiency reasons.

Unprompted selection is supported in pie menus by a technique called *mousing-ahead*. Mousing-ahead means the user does not have to wait for the system to display the menu before moving the cursor to make a selection. As the user moves the cursor, the input system buffers cursor location data. When the menu is finally displayed, the system reads the buffered data and analyzes it as if it were generated with the menu displayed. The system then immediately selects a menu item and removes the menu. In this way a user can make a selection without waiting for the menu to display (in effect, the mouse is being operated "ahead" of the display, hence the term mousing-ahead). Hopkins' implementation is slightly more sophisticated than just described. Menu display is suppressed until the user stops moving the cursor.

On the surface, it appears as if a marking menu is a pie menu with an ink-trail added to cursor. However, there is a major difference in the way the two techniques behave. Marking menus, depending on the context, may use sophisticated recognition. Marking menus analyze the path of a cursor as a mark, looking for certain features. If the interface recognizes other types of marks, a mark has to "look like" a marking menu mark before it can select from the menu. For example, suppose an interface recognizes a "C" mark (e.g., "C" triggers the copy command) and also marking menu marks (i.e., zig-zag marks). If mousing-ahead was used, the "C" would select the bottom item of a menu (assuming the user started drawing from the top of the "C"). With marking menus, the recognizer identifies the mark as a "C" and not as a zig-zag mark. Chapter 6 discusses in more detail, issues of integrating marking menu marks with other types of marks.

As a consequence of mark recognition, marking menu marks can be performed more casually than mousing-ahead movements with pie menus, especially with hierarchic menus. Mousing-ahead on pie menus must be an exact imitation of cursor movement used when selecting with the menu displayed. Marking menus, on the other hand, recognize the shape of the mark, independent of size and therefore the user can be more casual when drawing marks as opposed to mousing-ahead. There are designs where mousing-ahead can be made independent of movement size but, in general, this is not possible. See Section 2.5.6 for a detailed discussion of these issues.

The visual difference between marking and mousing-ahead is that marking leaves an ink-trail after the cursor, whereas mousing-ahead does not. We believe that, without an ink-trail during selection, a user must visualize selection from the menu. With an ink-trail, the user does not have to visualize selection, but rather remember the mark associated with a menu item and then correctly draw the mark. We believe the ink trial provides feedback which helps the user to correctly draw the mark.

2.3.2. Command compass

An interface mechanism very similar to a marking menu is the command compass used in the Momenta pen-based computer. Figure 2.3 shows how the command compass is used to move text.

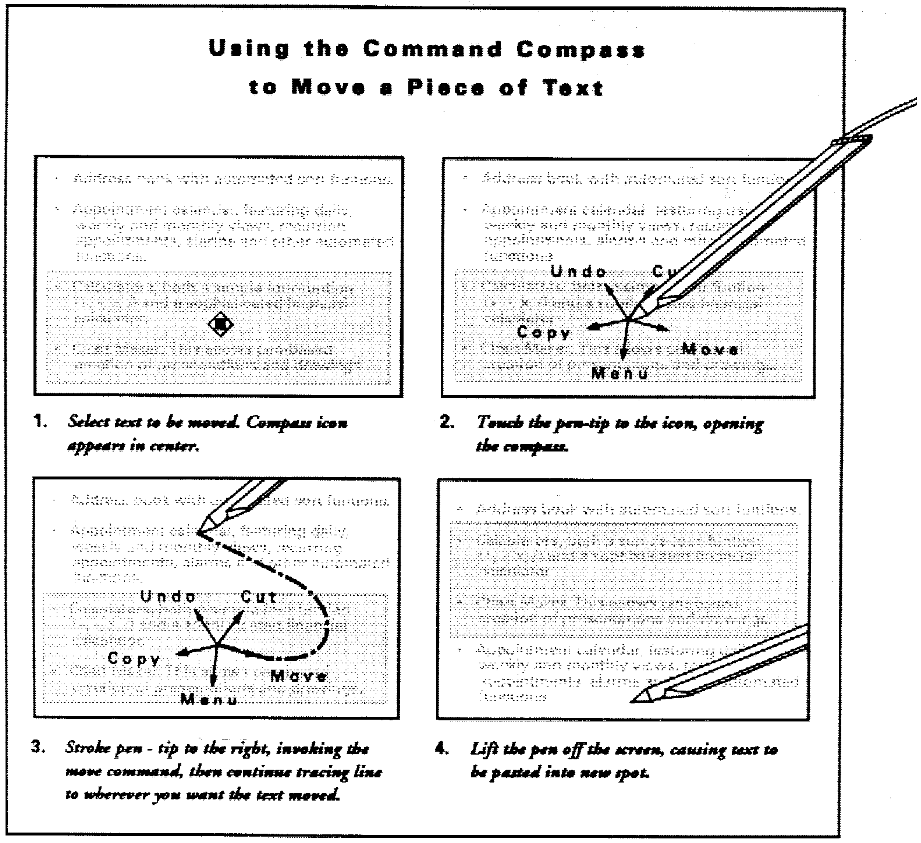

Using the Command Compass
to Move a Piece of Text
Undo
Cut
Copy
Move
Menu
1. Select text to be moved. Compass icon appears in center.
2. Touch the pen-tip to the icon, opening the compass.
3. Stroke pen - tip to the right, invoking the move command, then continue tracing line to wherever you want the text moved.
4. Lift the pen off the screen, causing text to be pasted into new spot.

Figure 2.3: The Momenta Command Compass (Momenta, 1991).

There are several differences between the command compass and marking menus. First, the command compass does not permit reselection. Once the pen is moved in the direction of a command, that command is immediately selected. Second, an explicit unprompted selection mode is not provided. No ink-trail is provided and unprompted selection relies on mousing-ahead (or "penning-ahead", since Momenta is a pen-based computer). While the Momenta interface uses marks, the command compass does not utilize marks. Finally, only one type and size of command compass is used. No hierarchic command compasses are supported.

The subtle difference in the way selection is done with a command compass versus selection with a marking menu affects the type of interactions each technique can support. With marking menus command selection occurs after a sector has been moved into and the pen lifted. With the command compass, command selection is done the moment a sector is moved into. Thus when selection occurs, the user is still in a physical mode (keeping the pen pressed). This physical mode can be used to express more parameters for the command, hence, physically pairing a command verb and its parameters. This is, of course, at the expense of not permitting reselection.

2.4. RESEARCH ISSUES

The ultimate goal of this research is to create a useful interaction technique. To attain this goal, several things must be accomplished. First, we must create a design for marking menus. Next, this design must be evaluated to determine its limitations and possible applications. From these evaluations, we can refine our design and develop recommendations for interface designers about when, where, and how marking menus can be beneficial. Given these goals, research issues surround the following question: what characteristics of marking menus do we need to understand to effectively incorporate this mechanism into the interface?

The most immediate question about marking menus is: how many items can be placed in the menus before it becomes too difficult to make selections using marks? Common sense tell us that parameters governing this aspect are articulation accuracy (i.e., how precisely can a human draw directional strokes), and human memory limitations (i.e., how quickly can a human learn and remember associations between menu items and marks). Other issues concern how hierarchic structure affects selection performance, how command parameters can be attached to marks, and how the design can be varied to accommodate the constraints of an interface. The following sections expand on these issues.

2.4.1. Articulation

Accuracy in selecting menu items and in marking is limited by the human motor system and the input device being used. This constrains the number of items that can be placed in a marking menu. *Articulation* refers to the motor system activities associated with selecting from a menu or making a mark, not memory activities like recalling the mark associated with a menu item. For example, suppose a user remembers the mark for a desired menu item. Can the user draw the mark accurately enough to select the menu item? In other words, can the user successfully articulate the mark once it is remembered?

Many factors may affect the success of articulation:

The type and characteristics of the input device. While the pen appears to be a natural input device for marks, operating marking menus with other types of input devices is also desirable. Thus, it is of interest to study users' performance not only with a pen but also with other popular types of input devices.

The number of items in a menu. As the number of items in a menu increases, the size of the menu items decreases and therefore pointing to them will become more error-prone and slower. Using a mark for selection should behave in a similar fashion. Precision of marking must increase as the number of items increases.

The type of articulation feedback provided. Feedback helps a user verify that a selection is being successfully articulated. For example, highlighting a menu item provides feedback. Supplying an ink-trail is another form of feedback, but is perhaps less salient. Finally no ink-trail (i.e., just the pen's or cursor's movement) provides even less feedback.

Chapters 3 and 5 investigate the effect of these factors through empirical experiments which measure speed and accuracy of selection when using marking menus. The results from these experiments are then interpreted to produce design guidelines.

2.4.2. Memory

Another aspect of marking menus concerns human memory. Using a mark to select from a marking menu involves, first, learning the association between menu item and mark, and then, recalling the association from memory before articulating the mark. There are several ways in which learning and recall can occur. For example, a user can memorize the association by rote memory ("this mark invokes this command"), or a user can reconstruct a mental image of the spatial layout of the menu or process of selection.

There are other factors affecting learning and recall. Differences in the angles between items must be memorable enough so the angle can be reproduced in drawing the mark. For example, a user may remember an item was the third from the top in a very densely packed menu, but the angular difference between items may be so small that it cannot be remembered precisely enough.

Whatever technique is used to remember the mark/item association, the exact limitations of marking menus relative to the limitations of human memory is a very complex question. Human memory in some situations can be considered almost infinite. For example, humans are capable of memorizing many complex symbol systems such as languages. With enough practice, the paths through extremely complex hierarchies of menus could be memorized and recalled. The question of how quickly one "learns the marks" depends on many variables: frequency of use, presence or absence of mnemonics or metaphors, menu layout, intelligence, motivation, application, etc.

Determining hard figures for "learning time" or "maximum number of items" relative to human memory is not possible. These measures depend largely on the user and the application. The intent of this research is to come up with guidelines that help designers exploit aspects such as frequency of use, metaphors, and menu layout to help make marking menus easier to learn.

In the case of marking menus note that training time is not as critical as with other interface techniques because a user "trains on the job". A user of marking menus does not have to spend time training before the selections can be performed. A novice can use the menus while a forgetful expert may occasionally have to use the menu. In either case, the user will still be performing "training on the job".

Do users learn and use marking menus the way the design suggests? The three modes of interacting with a marking menu (menu, mark-confirmation and mark modes) are intended to support the transition from problem solving to skilled behavior in a user. Card, Moran and Newell (1983) suggest that novices exhibit problem solving behavior ("how do I do this?") and experts exhibit skilled behavior (an expert knows how to solve the problem and does it efficiently). Rasmussen (1984) further refines this notion to include a middle step called rule-based behavior. Informally, rule-based behavior can be thought of as the user explicitly thinking "in order to do this I must do this". As Figure 2.3 shows, these stages of behavior can be mapped to the three modes of marking menus. The intention is that these modes are designed such that use of one mode builds the skills for the next mode and this assists in making the transition between modes. Do users actually behave this way with marking menus? If not, what sort of behavior is occurring and why?

We examine these issues of learning and remembering through empirical experiments (Chapter 3 and 5), and user behavior case studies (Chapter 4). The empirical experiments reveal learning curves and insights into the sort of menu structures that assist in learning and remembering menu layout and marks. A case study of user behavior using marking menus in a real application investigates learning and behavior patterns when marking menus are used in "everyday work" situations.

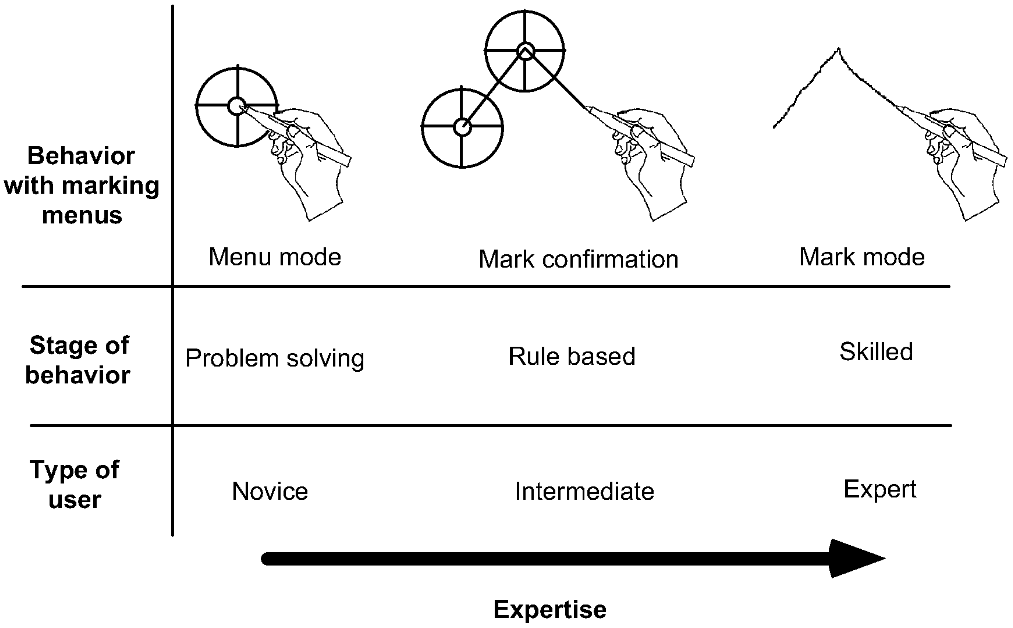


Figure 2.3: The relationship between stages of behavior, type of user, a user's behavior with a marking menu and expertise.

2.4.3. Hierarchic structuring

Another question concerns the effect that the structure of the menu hierarchy has on user performance. Specifically, how is user performance affected when breadth or depth is increased? (Depth is the number of levels in a hierarchy of menus; breadth is the number of items in a menu.)

Most of the research on hierarchic structuring of traditional menu systems focuses on depth versus breadth. This research can be divided into two types of studies: (1) theoretical models describing menu structure and user performance, and (2) empirical studies of menu usage. The theoretical studies concern models that describe menu search performance based on structure. From these models, structures that optimize search-time can be produced. The empirical studies attempt to verify the theoretical models, and estimate search time and error rates. These research efforts have addressed some basic issues concerning depth versus breadth.

The *navigation problem* (getting lost or using an inefficient path to find a menu item) becomes more likely as depth increases. Snowberry, Parkinson and Sission (1983) showed that error rates increased from 4% to 34% as menu depth increases from one to six levels.

Despite the problem of errors, there are several reasons to increase menu depth: *crowding, insulation* and *funneling.* Crowding refers to the problem of not having enough space on the screen to simultaneously display all the menu items. Insulation refers to the hiding information in deeper menus to protect a user from information overload. Funneling refers to the structuring of menus such that the hierarchy helps a user "narrow down" the choice and access items more quickly than using a flat menu structure.

Lee and MacGregor (1985) examine the tradeoff between funneling and response-execution time. Assuming all items were viewed before a selection is made, they found that optimal breadth was between 3 to 8 items per menu level depending on user response time and computer processing response time. Depth was effective when user response times were fast and computer processing time per option was slow. If it is assumed that the search terminates on average halfway through the items, then the optimal breadth is between 3 to 13 items at each level. These results should be tempered by the fact that they are based on a theoretical model and not on empirical user tests.

If meaningful groupings of items are used, Paap and Roske-Hofstrand (1988) show that optimal breadth at any level tends to be in the range of 16 to 36 and sometimes as high as 78 for traditional menu systems depending on human and computer response time. In terms of marking menus, these ranges are well outside the maximum number of items that can be selected with a mark. This raises the issue that reduction of breadth in a marking menu may increase the performance of marking but degrade the efficiency of menu selection in the menu mode.

Menu search time increases monotonically with depth (Landauer & Nachbar, 1985). This produces a log-linear relationship between search time and number of menu items. Kiger (1984) also found that performance (time and accuracy) decreased as depth increased further confirming that depth presents navigation problems to users.

Kiger also included error recovery in his analysis. This increased the variance in search time from 6 seconds to 20 seconds. Since error recovery occurs in the real world, this study more realistically characterizes the costs associated with hierarchical structuring. Kiger tested five types of hierarchical structures varying the depth from two to six levels and the breadth from two to eight items.

Performance can vary at different levels of the hierarchy. Snowberry, Parkinson and Sission (1983) report on error rate versus hierarchy level in a six level hierarchy. A higher proportion of errors occurred at the top two levels of the hierarchy than at the bottom two despite the fact that every level was a binary choice. The explanation for this is that higher level items are more abstract and therefore more subject to misinterpretation. Kiger also found that search times gradually become faster as a user came closer to the goal item. Other studies have revealed opposite results—better performance occurred at top levels (Allen, 1983). The explanation offered for the differences is that users were much more familiar with the top level items than the lower level items. This lends support to the notion that performance, structure, and item semantics in menus are intimately related.

Paap and Roske-Hofstrand (1986) point out that users restrict navigation because the menu structure has semantics or because they have experience with the menu. Both Card (1982), and McDonald, Stone, & Liebelt (1983) report that effects of organization disappear with practice. In other words, with practice, users navigate directly to the desired menu item. With experience, users move from a state of great uncertainty to one of total certainty. This lends support to the hypothesis that marking menu users will use marks with practice.

The previous research on depth versus breadth in menus indicates two important points relative to marking menus. First, users need to explore to make selections from menus with which they are not familiar, and the semantics associated with the structure has an effect on human performance. Marking menus behave somewhat like traditional menu systems when used in the menu mode (i.e., users can see item names and navigate through the hierarchy). Therefore, we can assume that the research findings mentioned above are applicable in menu mode. Second, once familiar with the menu structure, users of traditional menu systems want to directly select an item. In other words, users no longer require a menu. This behavior bodes well with using a mark to select from a marking menu.

Since the previous research in this area is somewhat applicable to the menuing mode of marking menus, the open research issues concern using mark mode to access hierarchic marking menus. The main issue is the effect of breadth and depth on user performance when using marks. Specifically, how deep and how wide can menus be made before marking becomes too slow or error prone? What sort of structuring makes mark articulation easier? For example, selection using marks from a menu with 16 items seems difficult. Selection from a menu with two levels of four item menus (16 items in total) seems more reasonable. In Chapter 5, we examine the effect of breadth and depth on marking by means of an empirical experiment on human performance using marks to select items from hierarchic marking menus.

2.4.4. Command parameters and design rationale

Besides the angle of a mark specifying the command verb, other aspects of a mark can express command parameters. For example, a mark's starting point, ending point and size can all contribute to command semantics. The question is how can these aspects of a mark be exploited in an interface? Issues of this type are examined in a case study which involved implementing marking menus in a real application (Chapter 4).

Subtle differences in design may have a profound effect on the way in which marking menus can be used. For example, a design that uses selection upon sector entry (e.g., the Momenta command compass) must be used differently than a design that uses selection on pen release (e.g., marking menus). These small design details can have a large impact on a design's ability to support hierarchic menus, command/parameter pairing, and reselection. In section 2.5, we describe this design space and present a design rationale for marking menus.

2.4.5. Generalizing self-revelation, guidance and rehearsal

Marking menus provide self-revelation, guidance, and rehearsal for the particular class of mark. Specifically, this is the type of mark that is created as a byproduct in selecting from directional menus. We referred to this class of marks as "zig-zag" marks. A pen-based application may also use other types of marks (e.g., editing symbols). There are two issues concerning the relationship of marking menus and other types of marks. First, can marking menu marks be integrated with other types of marks? Second, can a mechanism be developed to provide self-revelation, guidance and rehearsal for other types of marks?

A major advantage of marks is the ability to use features of a mark as additional command parameters. For example, a copy mark not only specifies that a copy command should be executed but also specifies what should be copied and to where it should be copied. How self-revelation, guidance and rehearsal can be provided for this type of information is an open question. Chapter 6 addresses this question.

2.5. DESIGN RATIONALE

This section presents the design rationale behind marking menus. First, the fundamental goals and the space of the design are defined. Next, an explanation and taxonomy of design options is presented. Finally, the rationale for choosing a particular set of options for the design of marking menus is given.

2.5.1. Fundamental design goals

The fundamental design goals of marking menus are:

- in the mark mode, speed of selection is emphasized over the self-revealing features.

- in the menu mode, self-revelation and guidance are emphasized over speed of selection

- in menu mode movement must be as close as possible to a rehearsal of marking. Ultimately, using the menu must facilitate learning the marks.

The last goal dictates that marking must mimic selecting in menu mode. Furthermore, marks must be distinguishable from one another. This provides a further goal for the design:

- selection in menu mode must create a unique path which can be reliably recognized by a computer.

We next examine the types of designs that address these goals.

2.5.2. The design space

In the most general sense, the design space can be described as: "discriminating selections from menus by cursor movements". Linear menus, array menus, and radial menus all fall into this design space. Linear menus are menus where the items are laid out in sequential linear fashion (top to bottom, or left to right). Array menus are menus where the items are laid out in both a top to bottom and left to right fashion. Radial menus are menus where the items are laid out in a circle. In these types of menus, the position of the cursor ultimately determines the item selected. A design that does not fit in this class would be menu selection based on time. In this case, the computer cyclically displays each menu item and the user presses a button when the desired item appears. This type of menu selection is often used in interfaces for handicapped users.

In this space, selection is performed relative to a starting point and the amount and direction of movement determines the selection being made. For example, in a linear menu, when the cursor is initially placed on the first item in the list, selection is determined by how far the cursor is moved down the menu.

Within this design space we are only considering designs in which menu selection is a physical rehearsal of marking. We want each movement path traced by a menu selection to be unique relative to the other movement paths involved in selecting from the menu. This will result in an unambiguous language of movements (or marks when the cursor leaves an ink-trail).

Within this design space we can identify several important design issues. These issues are discrimination, control, selection, display, backing up, and aborting.

2.5.3. Discrimination method

Discrimination method is defined as the type of movement used to discriminate selections. This can be either angle, length, or a combination of the two. Figure 1.8 shows a menu that uses length, and another menu that uses the combination of length and angle. Whether humans are better at discrimination by length or by angle is an open question.[9] In our context, discrimination by angle is preferable to discrimination by length for two reasons: efficiency, and scaling and rotation issues.

Under certain conditions, discrimination by angle (radial menus and angular marks)

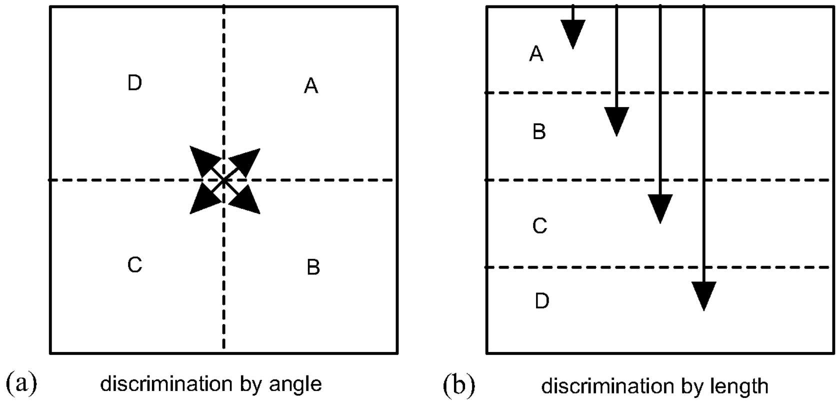


Figure 2.4: An example where discrimination by angle makes selection faster than discrimination by length. The lines with arrow heads show the movement needed to select an item. In the discrimination by angle case, selection of any item requires a movement of distance d. In the discrimination by length case, assuming all items are accessed with the same frequency and distance is equivalent to movement time, the average selection time will be 2L, where L is the height of a menu item. Assuming d is 0.5L, selection is four times faster with discrimination by angle.

allow faster selection than discrimination by length (linear menus and linear marks). First, because all the menu items are equidistant from the center of the menu in a radial menu, selection time is approximately the same for any item in the menu. In contrast, with linear menus, the first item can be selected more quickly than the last item in the menu. Figure 2.4 shows an example which compares a four-item radial menu and a four item linear menu. As described in Section 2.3.1, Callahan, Hopkins, Wieser, & Shneiderman (1988) have empirical evidence that eight-item radial menus are 15% faster and produce 42% fewer errors than eight-item linear menus. Treating selection from a radial menus as a one dimensional pointing task, and assuming that the amount of area used by a radial menu and a linear menu are the same, it can be shown that target size in a radial menu will always be larger than target size in a linear menu. For example, in Figure 2.4, the target size in the radial menu is the diagonal of an item. In contrast, target size in the linear menu is the height of an item. However, as the number of items increase in a radial menu, pointing to the narrow slices will become more difficult. To compensate for this, users will have to move farther away from the center, thus slowing their selection time. Determining the point where performance with a radial menu will degrade to the performance level of a linear menu is an open problem. Current research on two dimensional pointing (Mackenzie & Buxton, 1992) only deals with rectangular targets and therefore cannot be directly applied to radial menu slices.

There are also issues related to mark-based interfaces that make discrimination by angle preferable. Angular marks are preferred over linear marks because an angular mark can be scaled without changing its meaning (or, rather, changing the item the mark selects). In terms of a mark-based interface this means that a user is not restricted to draw the marks at a prescribed size. For example, a small "L" shaped mark would have the same meaning as a large "L" shaped mark. This is not the case with marks that are discriminated by length.

However, the meaning of angular marks changes if the mark is rotated. Rotating a horizontal to the right mark 45 degrees will cause it to be interpreted as a down to-

[9] It should be noted that discrimination can be performed at the reading or at the writing level (i.e., perception versus production of marks). These are significantly different problems. This dissertation examines production of angular marks. See Westheimer & McKee (1977) for a discussion of the perception of angle and length.

the-right mark by the system. In contrast, linear marks are not affected by rotation (i.e., a bull's eye menu. See Figure 1.8).

Discrimination by angle better reflects the way marks are interpreted in everyday life. Marks are generally insensitive to scaling but sensitive to rotation. For example, a small "l" has the same meaning as a large "l" but if it is rotated 90 degrees it perhaps takes on the meaning of "dash".

There is also the issue of *C:D ratio*. C:D ratio is defined as the ratio between the amount of movement of the input device (Control) and the amount of movement this imparts to the cursor (Display). On a pen-based system, the C:D ratio is constant and one to one because the cursor follows directly under the pen tip. For example, a one inch movement of the pen corresponds to a 1 inch mark. Therefore, with pen-based systems, C:D ratio is not an issue. However, with input devices that do not write directly on the display, (i.e., the mouse), C:D ratio is an issue. A one inch movement of the mouse may result in different lengths of marks on different computers if they have different C:D ratios. C:D ratios that vary depending on the speed of the movement (referred to as *cursor acceleration*) complicate this situation even further. A one inch movement made quickly can generate a much longer mark than the same movement made slowly, for example. Therefore, under these conditions, discrimination by length may be unreliable. However, discrimination by angle is not affected by varying C:D ratios. For example, a 45 degree mark is a 45 degree mark whether or not it is one or two inches long. Since it is desirable that our technique be usable with other input devices besides the pen, discrimination by angle is a better choice.

2.5.4. Control methods

Selection from a menu with a pointing device is generally accomplished by *dragging*, by *tapping*, or a combination of the two. We refer to these as the control methods. When dragging is the control method, pressing the pen down on the screen ("pen-down") displays the menu; moving the pen while it pressed against the screen ("dragging") selects different items; lifting the pen from the screen ("pen-up") confirms the selection. When menus are hierarchic, dragging into certain areas may cause submenus to be displayed for selection. When tapping is the control method, a pen-down followed quickly by a pen-up (a "tap") causes the menu to be displayed; a "tap" over an item confirms its selection. If the menu is hierarchic, the selection will result in another menu being displayed.

Dragging is preferred because selection in menu mode must be a rehearsal of the movement needed to make the mark. Marks are created by dragging the pen across the display surface and therefore dragging is a more accurate rehearsal of marking than tapping.

Marking menus use an action called press-and-wait to allow the user to switch into menu mode. We elected to use this action for several reasons. First, it deviates very slightly from the act of marking (the wait is only 1/3 of second). Thus the principle of rehearsal is not dramatically violated. For example, an action such as holding down a special key or making a special movement to invoke the menu would be a much more dramatic violation of rehearsal. Second, when a user wants to avoid menu mode, it usually means one wants to articulate the command quickly. Press-and-wait is easily avoided by quick articulation and avoiding it also makes selection faster. Third, according to our design goals, we assume that novices are not concerned with fast selection and therefore a slight delay in selection is a minor inconvenience. However, as users become more experienced with the menus and desires faster selection, the delay may also provide incentive to use marks.

There are other reasons why delaying the pop-up of the menu is valuable: it can be distracting; it can obliterate part of the screen; and it takes time. For a novice user these may not be problem since displaying the menu is desirable. For expert users, however, a delayed menu pop-up allows the creation of marks and avoids the negative side effects of the menu's display.

2.5.5. Selection events: preview, confirm and terminate

There are several events that occur when making a selection. Selection from a menu generally involves some sort of feedback indicating which item is about to be selected, for example, an item highlights. We refer to this capability as *selection preview*. Selection also involves an action which indicates to the system that it should actually carry out the selection. We refer to this as *selection confirmation*.

In the non-hierarchic case, selection confirmation results in the termination of the entire selection process. In the hierarchic case, selection confirmation will not necessarily terminate the selection process if the item selected has a sub-menu. We