Integrating Gesture and Snapping into a User Interface Toolkit

Tyson R. Henry Scott E. Hudson Gary L. Newell
Department of Computer Science
University of Arizona, Tucson AZ 85721 *

Abstract

This paper describes Artkit – the Arizona Retargetable Toolkit – an extensible object-oriented user interface toolkit. Artkit provides an extensible input model which is designed to support a wider range of interaction techniques than conventional user interface toolkits. In particular the system supports the implementation of interaction objects using dragging, snapping (or gravity fields), and gesture (or handwriting) inputs. Because these techniques are supported directly by the toolkit it is also possible to create interactions that mix these techniques within a single interface or even a single interactor object.

1 Introduction

The introduction of toolkits (see for example [1, 15, 14]) has made the task of creating window based user interfaces less daunting. Toolkits generally provide a set of predefined *interactor objects* (sometimes called *widgets*) and a means for composing these objects into larger groupings. These predefined objects and composition mechanisms provide tested and working interaction techniques which can often be used directly in an interface without modification. The use of predefined components avoids a great deal of duplication of effort and allows many of the complex details of screen update and response to input events to be hidden from the average programmer.

Unfortunately, the use of toolkits has also had a limiting effect on the class of interaction techniques used by some interfaces. Most toolkits support only a limited range of techniques such as menus, sliders, simulated buttons, etc., and tend to promote relatively static layouts (notable exceptions being [2] and [14]). Most toolkits do not, for example, provide good support for dragging interactions (a notable exception being [21]) much less more exotic techniques such as snapping or gesture.

This paper describes Artkit – the Arizona Retargetable Toolkit. An explicit goal of this toolkit is to support more sophisticated interaction techniques such as dragging, snapping, and gesture in the same framework as more conventional techniques such as simulated buttons, sliders, etc. Artkit is designed to be window system independent – it is currently supported under both SunView [20] and the X window system [19] and can easily be retargeted to other systems by replacing a simple set of interface routines. Artkit is also designed to be extensible. It is built using object-oriented techniques (in C++) and can be extended via inheritance at several levels ranging from global policies concerning how events are distributed to specific interactor object classes.

As an example of the kinds of interactions that can be supported by Artkit, Figures 1-4 show an interactive sequence from a sample application. This application is an entity relationship (ER) diagram editor [3, 4]. ER-diagrams contain a series of graphical symbols connected by lines. Symbols

*This work was supported in part by the National Science Foundation under grants IRI-8702784 and CDA-8822652

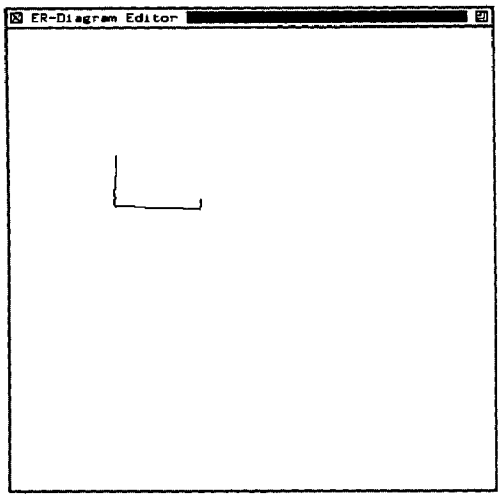

Figure 1: Drawing a Symbol

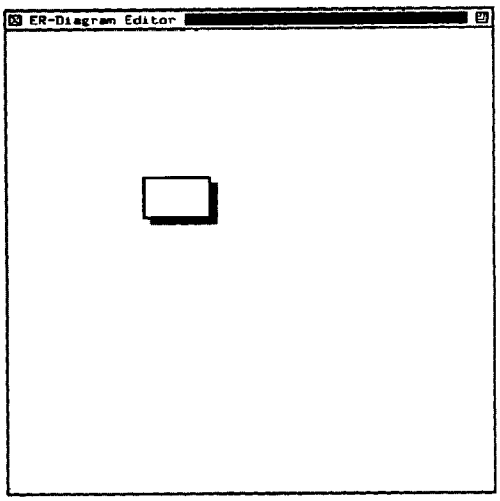

Figure 2: Positioning a Recognized Symbol

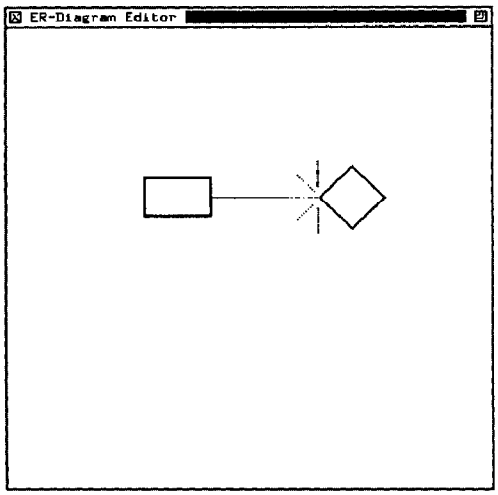

Figure 3: Connecting Two Symbols

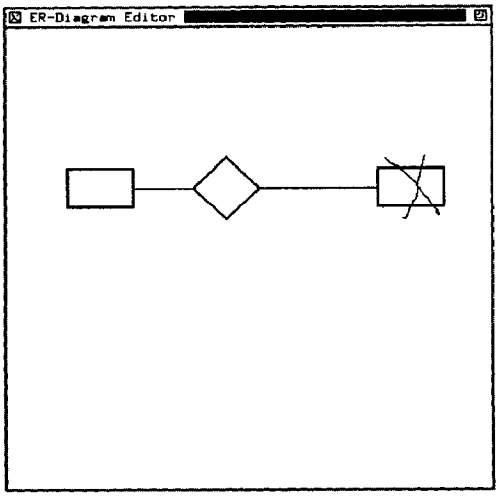

Figure 4: Deleting a Symbol

include, rectangles which represent *entities*, diamonds which represent *relationships* between entities, circles or ovals which represent *attributes* of entities, and triangles which represent special *is-a* relationships.

As illustrated in Figure 1, each of the basic components of an ER-diagram can be entered via a hand drawn gesture. As the symbol is being drawn, a gesture recognizer dynamically compares the symbol with a symbol alphabet. Once enough of the symbol has been drawn to unambiguously recognize it, a new interactor object corresponding to the symbol is created (this form of *eager* recognition is discussed, for example, in [18]). As shown in Figure 2, the newly created object is then placed directly under the cursor and the interaction continues smoothly by allowing the user to drag the object into its final position without releasing the mouse button.

Once several symbols have been created, they may be connected with lines. This interaction starts at particular *connection points* within each symbol and supports a rubber-banded line which snaps to legal positions on other symbols as illustrated in Figure 3.

Finally, commands can also be entered either as gestures, or via more conventional menus. For example, Figure 4 shows the X-shaped gesture used for deleting a symbol. Alternately, the same command could have been invoked by selecting an object and choosing delete from a pull-down menu. In general, the user interface designer is free to

mix and match input techniques to produce the best interface.

In the next section, the overall event handling architecture of the toolkit will be discussed. Section 3 will consider the specific structures used to handle event dispatch. Sections 4 and 5 will consider snapping and gesture support respectively, while Section 6 will consider the interactor object library provided by the system. Finally, Section 7 will describe the implementation of the system, consider its limitations, and discuss directions for future work.

2 Architecture

This section considers the overall architecture of the toolkit. It presents the basic foundations that the toolkit provides for the implementation of interactor object classes – in the terms used by the X window toolkit, the *intrinsics* layer. Because the toolkit is implemented directly in an object-oriented language, facilities for inheritance, specialization, and reuse are supported in a manner that is integrated with the language used for the application. This allows easy application specific specialization when the interface designer deems it appropriate.

Objects in the system (including windows as well as all drawing and interactor objects) are configured in a hierarchical manner – each object may contain zero or more child objects which contribute to its appearance and behavior. The toolkit manages the input and output of the (dynamic) hierarchy of objects making up the user interface. Output is handled by the redraw control portion of the system in a fairly conventional manner. Each object in the system responds to the `draw_self` message by producing its own output within a designated window object and by arranging for the proper drawing of its child objects (i.e., by sending additional `draw_self` messages).

The primary contribution of Artkit falls in the area of input. In Artkit, the management of input can be summerized by its first, accepting low level input events from the host window system, then transforming these low level events into higher level events, and finally delivering these higher level events (in the form of messages) to interactor objects. We call this overall process *event dispatch*. The event dispatch process attempts to deliver each low-event to one of several possible recipient objects. As described below, each eligible object is tried in turn until some object indicates that it has *consumed* the event. Events that are not eventually consumed are discarded.

Almost all events are dispatched using one of two general paradigms: *positional* dispatch or *focus* dispatch. Positional dispatch selects an object to receive an event based on the position of the locator at the time of the event (e.g., the object under the cursor receives the event). Focus based dispatch relies on the notion of a focus or current object (e.g., the current *text focus* object receives all events involving key presses).

While these two general paradigms for dispatching events cover most cases, there are occasions when different paradigms are needed. For example, the current library also supports a flavor of focus based dispatching which we call *monitor focus*. Under this paradigm all events are sent to a focus object but the events are never consumed (hence are always sent to other objects as well). This allows the construction of objects that monitor or record events without interfering with the rest of the interaction. This has been used for example, to implement a crosshair style cursor in a drawing application.

In general, we believe that there are a number of different general paradigms for dispatching events and that it should be possible to add new paradigms to the toolkit for special purposes. To accomplish this, the toolkit uses an explicit object to implement each dispatch paradigm. We call these objects *dispatch policy* objects. These objects are installed in a prioritized list. New dispatch policy objects can be created (for example by specializing existing objects) and added to this list at any time. When a new event arrives it is passed to each dispatch policy object in priority order until one of them indicates that it has consumed the event.

Each policy object is responsible for translating a low level event into zero or more higher level events and delivering those events to interactor objects in the interface. We chose to deliver high-level events instead of low-level events because in the process of building interactor objects in other systems we observed that certain patterns of low-

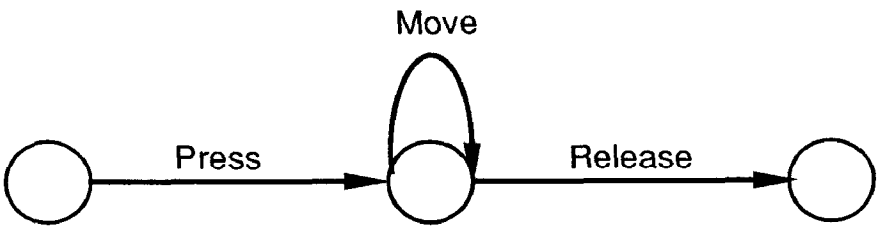

Figure 5: Part of a Typical Interactor State Machine

level interaction were repeated many times. In particular, if interactors are seen as controlled by finite state automata [26, 11, 12] certain stylized pieces of these machines occur over and over. For example, a wide range of objects respond to events in a fashion similar to the machine shown in Figure 5. This machine responds to the sequence: button-press, sequence of mouse moves, button-release. We have encapsulated a number of these prototypical partial machines into one or more higher level events[1]. These partial machines are each handled by objects which translate low-level events into high-level events and at the same time maintain their own state. We call these objects *dispatch agents.*

As shown in Figure 6, each dispatch policy object controls, and sends events to, a series of dispatch agent objects. For example, the positional dispatch policy object sends events to the click and double click agents, while the focus dispatch policy object sends events to the text entry agent. All current dispatch policy objects are implemented in an extensible fashion. They each maintain a simple prioritized list of agents. When a new event arrives, they pass it to each agent in turn until the event is consumed. New specialized dispatch agents can be included by simply adding another object to the list.

Creating new specialized dispatch agents is made easier by the use of inheritance. For example, the snap dispatch agent which handles dragging in the presence of gravity fields is a specialization of the drag dispatch agent. Implementing the snap agent was simplified by inheriting much of its implementation.

[1]Another response to this observation is to try to create one generalized state machine that covers all the common cases as was done in [21].

3 Dispatch Agents

The toolkit currently supports a standard library of dispatch agents. Under the positional dispatch policy, the standard library provides agents for:

- **press** and **release** mouse button down and mouse button up,
- **click** both press and release within a small area,
- **select** like click but maintains a set of currently selected objects,
- **double-click** two clicks within a small area,

The positional dispatch policy works by considering the set of objects "under" the cursor (as determined first by the bounding box of the object then, if necessary, by a more specialized test performed by the object itself). It attempts to dispatch an event to each of these objects by considering them in front to back order until some object indicates it has consumed the event. For each object considered, the dispatch policy attempts deliver the event via each of the positional dispatch agents (in priority order). Each interactor object can control the high-level events it receives in two ways. Each object registers with the system the set of high-level event messages it is currently interested in receiving. Objects also have the option of rejecting messages by not consuming them. In general, an object can base the decision to reject an event on any criteria including tests involving

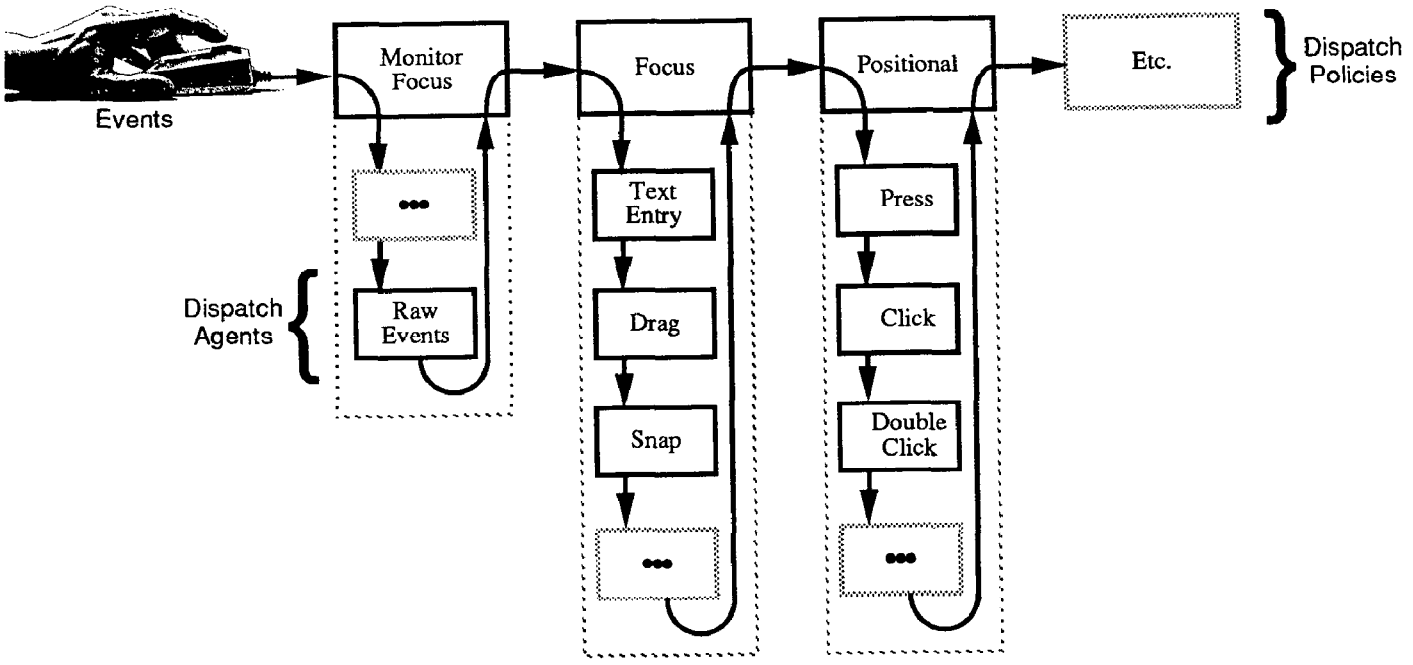

Figure 6: Organization of Dispatch Objects

the internal state of the object, and tests involving semantic information from the application.

As an example, assume that a low level mouse button down event, has been given to the positional dispatch policy object (after having been rejected by the monitor focus and focus policies). The positional dispatch policy object would first search for an object under the cursor. For the first such object, it would use the **press** dispatch agent to attempt to deliver the event (as a **press** message). If this failed it would try the **click** agent, and so forth. If the event was not consumed by the object (in any of the forms presented by these agents), the next object under the cursor would be considered, and so forth until the event was consumed. If no object consumed the event in any form, it would be passed to the next dispatch policy object (if any).

Each dispatch agent implements a message protocol which represents the high-level events it generates. For example the click agent sends a `click` message which contains the button number of the mouse button involved as a parameter. Similarly, the select agent sends two possible messages: `select` and `unselect`. The first message is used to inform an object of its entry into the current selection set, and the second is used to inform an object of its removal from the set. Note that the select agent does all the rest of the bookkeeping for properly maintaining the currently selected set – interaction objects normally just keep their graphical appearance up to date. For most messages, the low-level input event is also passed as a parameter. This allows the interactor object to access information such as the status of shift keys or the timestamp for the event. Rejection or consumption of the event corresponding to a message is indicated by a status code returned as the result of the message.

Focus based agents each maintain a current focus object (or object set) which receives all high-level event messages of a certain type. For example, text input always goes to the current text focus object. Each focus agent maintains a mes-

sage protocol for informing objects when focus is obtained and removed. The standard dispatch agents currently supported under the focus dispatch policy include:

- **text-entry** a key press directed at text entry focus object,
- **drag** a dragging interaction,
- **inking-drag** dragging which paints a trail along the drag path,
- **snap** a dragging interaction in the presence of gravity fields,
- **gesture-segmentation** inking-drag filtered into segements for gesture input.

As an example of how high-level event messages might be used by an interactor object, consider an object which makes use of simple dragging. This object would initially listen for a `press` message. Upon receiving a `press`, the object would make itself the drag focus. It would then receive a `start_drag` message, a series of `drag_feedback` messages (one for each mouse movement event), and a terminating `end_drag` message. Note that all of the state information needed to implement a drag is kept by the dispatch agent and no case analysis based on state is needed within the interactor object itself.

4 Snapping

Snapping and gesture are both implemented using focus based dispatch agents. Snapping is simply an extension of the normal drag dispatch agent. In addition to the `start_drag`, `drag_feedback`, and `end_drag` messages used in the dragging protocol, the snapping dispatch agent adds several new messages to indicate when snaps occur. The snap dispatcher implements additional behavior not found in the drag dispatch agent by noting when the cursor nears a *snap site* (a special interactor object acting as a *gravity field*). Artkit implements *semantic snapping* [9] – a snap is made when the cursor comes within "gravity distance" of the snap site **and** a semantic test function attached to the object owning the site indicates that the snap is semantically valid. When a semantically valid snap site is found, the `snap_feedback` message is used to inform the object. Conversely, when the cursor moves out of range and a snap is broken, the `unsnap_feedback` message is sent. In addition, if no semantically valid snap site can be found, but an invalid site occurs within snapping distance, special "anti-snap" feedback can be provided. This is done using the `anti_snap_feedback` and `anti_unsnap_feedback` messages. Feedback created based on anti-snapping serves the purpose of an error message as shown in Figure 7. Here an entity in our ER-diagram editor has been connected directly to another entity without the required intervening relationship.

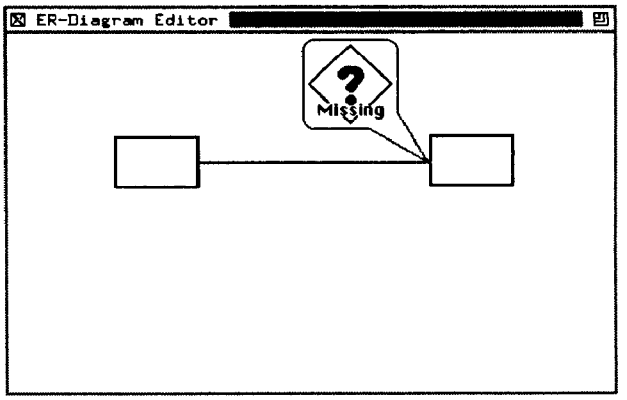

Figure 7: Anti-Snap Feedback

Note that because of the structure of the toolkit, it has been possible to encapsulate all the implementation of snapping within a single snap dispatch agent. This agent is responsible for registering and removing snap sites in search indices and for performing the actual search for snap sites during a drag. Since this agent is implemented only once, we can afford to implement sophisticated search techniques for efficient use of arbitrary semantic tests (for full details see [9]). On the other hand, objects using snaps need only register snap sites and listen for high level events indicating when feedback and actions are to occur – all other details are taken care of by the dispatch agent.

5 Gesture

Gesture or hand drawn input, although experimental in nature and often different in character than other forms of input [17, 25, 24], is also supported by the toolkit.

As is common in several systems [23], Artkit breaks the process of gesture recognition into two parts: filtering or *segmentation*, and recognition. The segmentation portion of the system acts in a lexical manner accepting raw input events from the device and grouping these into line (or short curve) segments. These segments are then used as input to a recognition algorithm which attempts to match the segments with a symbol from a given alphabet of symbols.

While segmentation can be done in an relatively alphabet independent manner, the best results for recognition seem to require a recognition algorithm tailored to the symbol set being recognized (for a survey of recognition algorithms see [23, 13, 16, 22, 18]). As a result, Artkit encapsulates each of these activities separately. A focus based dispatch agent (derived from the inking-drag agent) is used for segmentation. Recognition is performed by another *recognition* object which encapsulates both a particular symbol alphabet and a recognition algorithm. When an object is established as the current gesture focus, a recognition object is supplied as well. This recognition object serves as an intermediary between the segmentation dispatch agent and the focus object itself. As segments are recognized they are passed individually to the recognition object. When a complete symbol is recognized, it is passed in turn to the real focus object as a single high-level event message.

To show how this system is used in practice, we return to our ER-diagram editor example. In this application, two separate symbol alphabets are used: one for creating ER-diagram symbols (entities, relationships, and attributes) and one for entering commands (move and delete). Each alphabet is handled by one or more "sensitive area" objects – invisible rectangles that accept gesture input. As illustrated in Figure 8, one sensitive area covers the entire drawing window and is placed under all other objects. This object handles the alphabet used for symbol creation. When a symbol is recognized by this sensitive area object it creates a new interactor object corresponding to the symbol drawn by the user. A separate sensitive area also surrounds each existing symbol interactor object. These areas handle the alphabet used for the commands that can be applied to those objects. By separating sensitive areas that respond to different alphabets, we can apply different – highly tailored – recognition algorithms to each alphabet. The use of tailored algorithms offers several advantages over the use of one more general algorithm for recognizing all symbols. First, it makes the implementation of each recognition algorithm easier because there are fewer conflicts between similar symbols. Second, use of tailored algorithms can produce higher recognition rates since it is possible to use specialized techniques or feature metrics that work very well in limited cases but are not suitable for general use.

In addition, because gesture input is handled by a normal dispatch agent, it can be integrated with other forms of input. In this case, a different form of sensitive area surrounds each snap site of each symbol. If the mouse button is pressed in this area, a snapping interaction with rubberband line feedback is begun (i.e, the object is made the snap focus). Alternately, if the mouse button is pressed in some other area, it causes a gesture interaction to begin for a particular alphabet (i.e., the object is made the gesture-segment focus object with a particular recognizer object).

It is also possible to mix gesture with other input techniques in one interactive sequence. For example in the ER-diagram editor, an eager recognition algorithm [18] is used for the symbol creation alphabet. Once enough of a symbol has been entered to recognize it unambiguously, the system returns the symbol. A new interactor object is then created corresponding to the symbol and is immediately made the drag focus. From the user's point of view, the interaction starts by drawing the object and ends by positioning it – all in one smooth motion.

6 Interactor Object Library

The final component of the Artkit system is a library of actual interactor object classes. The library currently includes: simulated push buttons, radio buttons, editable lines and rectangles, dragable icons, horizontal and vertical sliders, pull down menus, and text edit fields. The toolkit also supports clipping and scrolling composition objects (in addition to simple composition which is supported by every object). Finally, a number of

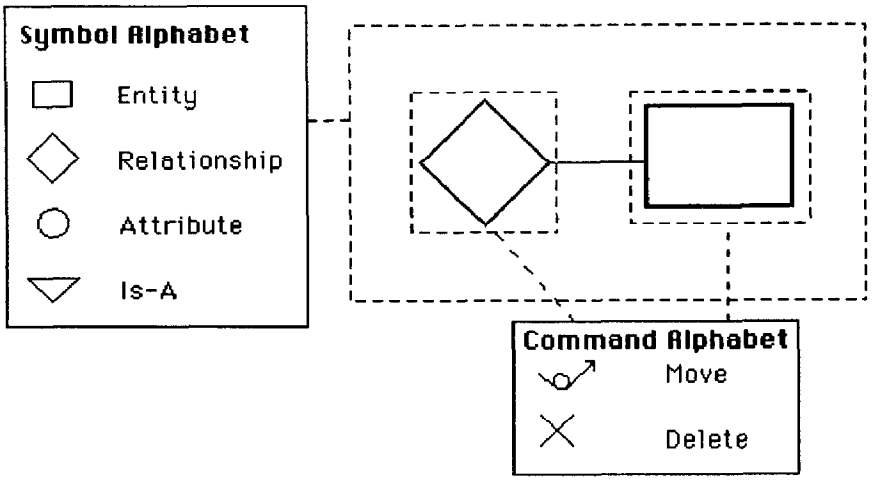

Figure 8: Use of Sensitive Areas with Different Alphabets

more complex interactors involving snapping and gesture are provided. These include a "connected icon" class for visual programming systems and "sensitive area" classes for snapping and gesture input.

7 Implementation, Limitations, and Future Work

Artkit is written in C++. The basic event dispatch and redraw facilities of the system are implemented in approximately 5000 lines of C++ code while another 5000 lines is used to implement the current set of base interactor classes which forms the default library.

Artkit has evolved through three major versions and has been used for several serious user interface projects. Artkit has formed the graphical underpinnings for the Penguims constraint based UIMS currently under development at the University of Arizona, as well as its predecessor the Apogee UIMS [5, 7]. The toolkit has also been used to implement the Opus user interface editor [8] (as shown in Figure 9) as well as a new system for the interactive display of graphs [6], and an interactive information system for use in Molecular Biology [10]. Finally, the toolkit has been used for a number of student projects in a user interface tools course.

The toolkit is still under active development and still has certain limitations. For example, since the emphasis in the development of this system has been on constructs for flexible input, Artkit currently supports only a minimal set of output primitives. These include support for lines, rectangles, text, *rasterop* operations on bitmaps, and simple fill operations. All of these operations can be clipped to rectangular regions and a simple trivial reject test based on bounding rectangles is also supported. However, the system currently only supports black and white output. Finally, the system only supports single-level, non-nested, windows (however, the functionality of nested windows can be achieved with the use of clipped composition objects with an opaque background).

Another limitation of the current system concerns gesture input. Currently gestures directed toward a particular object must start within that object. This does not allow gestures that specify their target object by means of a feature point in the middle of the gesture. While this type of gesture is not as common, it is sometimes useful. We are currently considering a new segmentation dispatch agent which would allow a series of segments to be rejected by one recognizer object and passed to another sensitive area found along the path of the gesture. This will require the implementation of a new hybrid focus/positional dispatch policy object. However, this should not re-

quire any changes to the basic input structure of the system.

References

[1] Apple Computer, *Inside Macintosh*, Addison-Wesley, Reading, Mass. (1985).

[2] Cardelli, L., Building User Interfaces by Direct Manipulation. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta (October 1988) pp. 152–166.

[3] Chen, P., The Entity Relationship Model—Towards a Unified View of Data, *ACM Transactions on Database Systems*, **1** No. 1 (March 1976).

[4] Czejdo, B., Elmasri, R., Rusinkiewicz, M, Embley, D., A Graphical Data Manipulation Language for an Extended Entity—Relationship Model, *IEEE Computer*, **23** No. 3 (March 1990) pp. 26–36.

[5] Henry, T. R., Hudson, S. E., Using Active Data in a UIMS, *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta (October 1988) pp. 167–178.

[6] Henry, T. R., Hudson, S. E., Viewing Large Graphs, *University of Arizona Technical Report* TR 90-13 (April 1990).

[7] Hudson, S. E., Graphical Specification of Flexible User Interface Displays, *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Williamsburg, Virginia (November 1989) pp. 105–114.

[8] Hudson, S. E., Mohamed, S. P., Interactive Specification of Flexible User Interface Displays *to appear in ACM Transactions on Information Systems* (1990).

[9] Hudson, S. E., Adaptive Semantic Snapping – A Technique for Semantic Feedback at the Lexical Level, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Seattle, Wash. (April 1990) pp. 65–70.

[10] Hudson, S. E., Peterson, L. L., Schatz, B. R., Systems Technology for Building a National Collaboratory *University of Arizona Technical Report* TR 90-24 (July 1990).

[11] Jacob, R. J. K., Executable Specifications for a Human-Computer Interface, *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference*, Boston (December 1983) pp. 28-34.

[12] Jacob, R. J. K., A Specification Language for Direct-Manipulation User Interfaces, *ACM Transactions on Graphics*, **5** No. 4 (October 1986) pp. 283-317

[13] Kim, J., Gesture Recognition by Feature Analysis, *IBM T. J. Watson Research Center Technical Report* RC 12472 (January 1987).

[14] Linton, M., A., Vlissides, J., M., Calder, P., R., Composing User Interfaces with InterViews, *IEEE Computer*, **22** No. 2 (February 1989) pp. 8–22.

[15] McCormack, J., Asente, P., An Overview of the X Toolkit, *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta (October 1988) pp. 46–55.

[16] Murase, H., Wakahara T., (1986). Online Hand-Sketched Figure Recognition, *Pattern Recognition*, **19** (1986) pp 147–159.

[17] Rhyne, J., Dialogue Management for Gestural Interfaces, *IBM T. J. Watson Research Center Technical Report* RC 12244 (October 1986).

[18] Rubine, D., H., The Automatic Recognition of Gestures, *Ph.D. Dissertation*, Carnegie-Mellon University (to appear 1990).

[19] Scheifler, R. W., Gettys J., The X Window System. *ACM Transactions on Graphics*, **5** No. 2, (April 1986), pp. 79–109.

[20] Sun Microsystems Inc., *SunView 1 Programmer's Guide*. Sun Microsystems Inc., Mountain View, CA, (1988).

[21] Szekely, P., A., Myers, B., A., A User–Interface Toolkit Based on Graphical Objects and Constraints, *Proceedings of OOPSLA '88*, San Diego, (September 1988) pp. 36–45.

[22] Tappert, C.C., Fox, A.S., Kim, J., Levy, S.E., Zimmerman, L.L., Handwriting Recognition on Transparent Tablet over Flat Display, *IBM T. J. Watson Research Center Technical Report* RC 11856 (March 1986).

[23] Tappert, C.C., On-Line Handwriting Recognition - A Survey, *IBM T. J. Watson Research Center Technical Report* RC 14045 (December 1987).

[24] Tappert, C.C. Applications of On-Line Handwriting Recognition, *IBM T. J. Watson Research Center Technical Report* RC 14071 (October 1988).

[25] Wolf, C.G., Can People Use Gesture Commands?, *IBM T. J. Watson Research Center Technical Report* RC 11867 (April 1986).

[26] Wasserman, A., I., Extending State Transition Diagrams for the Specification of Human-Computer Interactions, *IEEE Transactions on Software Engineering*, **SE-11** No. 8 (August 1985) pp. 699-713.

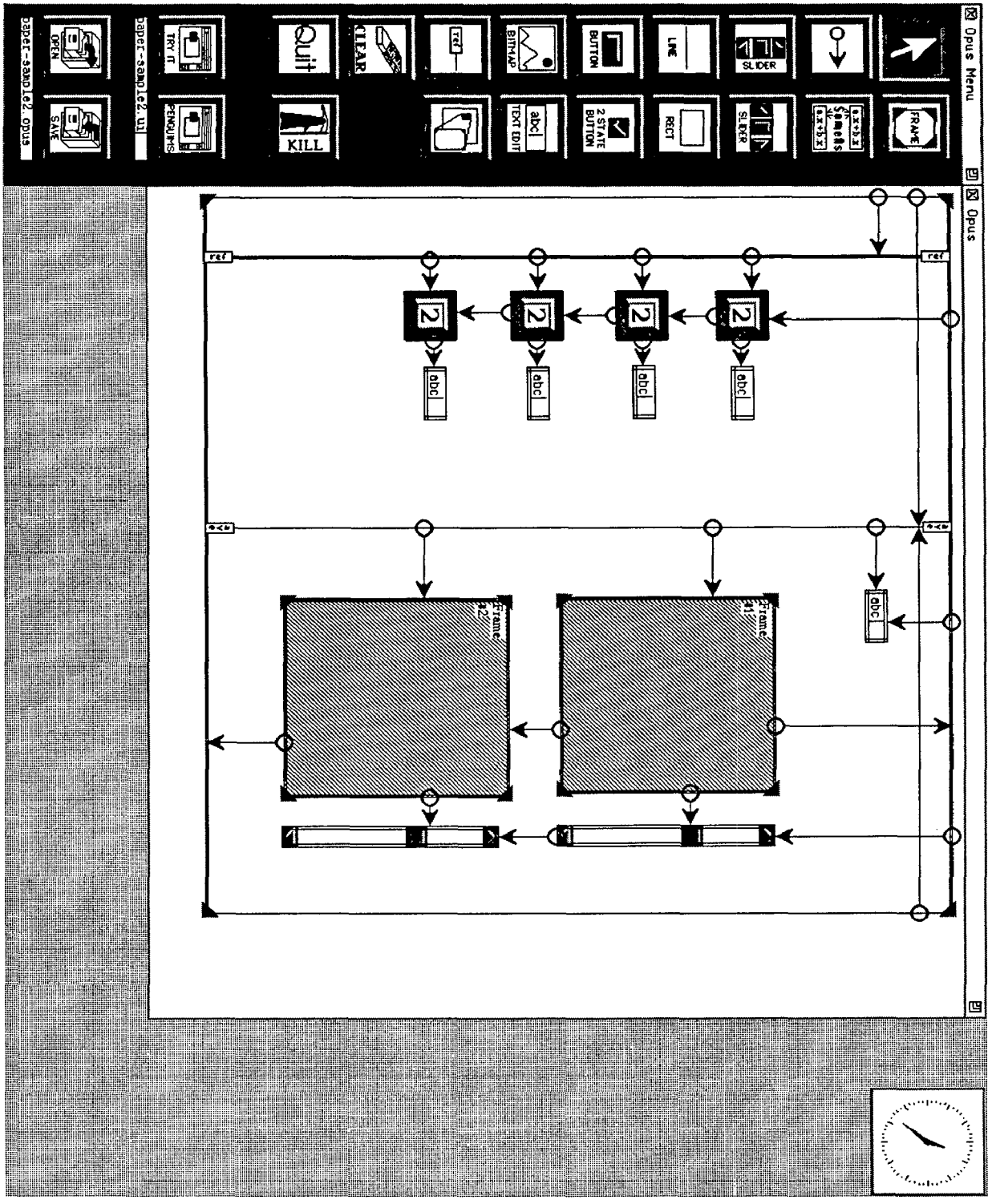

Figure 9: An Interactive Interface Builder Constructed With Artkit