INTERFACE DESIGN FOR SIMULATIONS

Louis Weitzman
Mark Rosenstein
Jim Hollan

NPRDC-UCSD Intelligent Systems Group
Institute for Cognitive Science C-015
University of California, San Diego 92093

Abstract

New computer-based technologies coupled with recent advances from artificial intelligence and cognitive science enable the exploration of new forms of intelligent graphical interfaces. In order to facilitate the construction of these interfaces, powerful software tools (Sheil, 1983) are required. The ***Graphics Editor*** we discuss in this paper is a tool designed to assist in the construction of graphical interfaces to simulations and real-time systems. It was developed as part of a larger research effort concerned with the construction of intelligent computer-assisted instructional systems (Hollan, Hutchins, & Weitzman, 1984). The purpose of this paper is to provide a brief introduction to the editor, describe how it is useful for building interfaces to simulations and real-time systems, and discuss the advantages of an object-oriented approach to graphical interface design.

Introduction

In this paper we discuss an object-oriented Graphics Editor (Figure 1) designed to facilitate the implementation of interfaces to simulations and real-time systems. This editor is in use in a number of our research projects. It is also being used by instructors building and augmenting the interface to a propulsion simulation (Steamer). An overview of Steamer is presented in Hollan, Hutchins, and Weitzman (1984). During the last few years the Editor has been used to construct hundreds of graphical diagrams. A *diagram* provides an interactive graphical mechanism for monitoring and changing the state of an underlying simulation. In Steamer, diagrams allow students and instructors to view and manipulate a steam propulsion simulation at a variety of conceptual levels. The current Steamer system provides approximately one hundred color views specific to the propulsion domain, which range from high-level abstract representations of the plant like the *Basic Steam Cycle* (Figure 2), to gauge panels depicting sets of gauges quite like those found on a ship (Figure 3).

The opinions expressed in this paper are those of the authors, are not official, and do not necessarily reflect the views of the Navy Department.

The diagrammatic interfaces created with the Graphics Editor function in two ways. First, the states of variables in a simulation model or real-time interface are reflected graphically by objects in the diagram. Thus, the fluctuating operational status of particular components of a simulation can be depicted by changes in color or other graphical features of their iconic representations. In Steamer diagrams a pump's state is depicted as green if it is operating and red if it is off. The second function provided by the graphical interface is to permit control of the underlying simulation or real-time interface by allowing a user to point to components with a mouse and change their state by clicking on them. In Steamer one can change the level of a tank by simply pointing to a new position for the water level and clicking.

Steamer views also attempt to make the causal topology of the propulsion system more apparent by showing (via animation) the directions and rates of flow between components. In addition, diagrams are often constructed to reveal aspects of a system which assist in the development of understandings of its operation. For example, some diagrams focus directly on concepts such as feedback, while others provide simplified views of systems highlighting the functioning of important components and illustrating the principles underlying the system's operation.

The Graphics Editor

The Graphics Editor has been used to create the existing set of Steamer diagrams as well as diagrams in a number of other areas. It provides many functions commonly available in computer-aided design systems. One can save and restore diagrams from files, mark the elements of a diagram (individually, by type, within an area, etc.), and edit those marked elements (move, copy, delete, etc.). A grid facility is provided to assist in accurately positioning icons within a diagram. The multi-paned menu interface to the editor is shown in Figure 4.

The Graphics Editor has a number of unique features. It is these features and the object-oriented method of implementation that we focus on in this paper. A diagram is built up from a standard set of intelligent objects, or *icons*. The available icons consist of basic graphical primitives (rectangles, circles, lines, etc.), various indicators (dials, columns, graphs, etc.), and a large set of icons related to the propulsion domain (a variety of pumps, valves, pipes, and electrical components). Some of these icons are depicted in the Sampler Diagram in Figure 5. The facility also exists to create new graphical objects from this existing set.

In the process of creating a diagram, the major actions are *pointing* and *selecting*. For purposes of demonstration, consider creating a dial for the Steamer diagram, Boiler Console 1B (figure 3). The user would click on *dial* in a menu on the black and white screen. The cursor would then be taken to the color screen where one would position and size the dial.

Immediately, a dial with many characteristics defaulted (e.g. its color, its minimum and maximum values, the number of divisions on its scale, etc.) would be created (Figure 6a). Then through a process of incremental refinement one critiques that dial by changing various parameters (Figure 6b) until it has all the intended graphical properties (Figure 6c). The goal here is to allow a user of the editor to think about the objects of an interface in natural terms.

Object-Oriented Interfaces

The process of constructing diagrams with the Graphics Editor creates a program written in LISP. This code contains both the specification of the icons in the diagrams and how they interact with the simulation. We have designed and implemented the Graphics Editor in the object-oriented *Flavors* system of *Zetalisp* (Weinreb & Moon, 1981). The object-oriented style of programming facilitates design by providing abstraction mechanisms which allow the packaging of implementation details both hierarchically and through generic communication facilities. The resulting code is easier to read, understand, and maintain. In addition, this style of programming seems particularly suitable as a basis for building interactive graphical interfaces since one commonly wants a number of particular instantiations of the same general class of object.

Objects are entities comprised of other simpler entities. Each entity contributes the ability to handle various classes of messages. The concept of objects originated with the Simula language (Dahl & Nygaard, 1966) and was the basis of the Smalltalk language (Goldberg & Robson, 1983). In the Zetalisp programming environment, these abstract objects are known as *Flavors*. Each type of icon, such as a dial, is a flavor. Each particular icon is an *instance* of its respective icon flavor. A flavor provides a template for storage of *instance variables* and a set of contracts or *messages* each instance is capable of understanding. An instance allocates storage for its instance variables and is the object to which messages are sent.

One important abstraction technique of the flavors system can be described in terms of contracts between callers and objects. Each icon participates in a contract to display a value. This contract relieves the caller of any responsibility for the mechanism of how the value is shown. Thus, when a caller tells an icon to show the value 7, it must carry out the appropriate action. A tank must position its water level to show this value, a dial must adjust its needle to the appropriate point, and a pipe must show the appropriate flow rate. Thus the caller can be very simple. It just tells the icon to show 7, and the complexity of how the showing is actually implemented is hidden within the icon.

Requiring icons to fulfill these contracts not only allows the callers to become clearer, but also allows the generalization of contracts into classes of contracts. For instance, many of the icons, like the dial, can show any value between a minimum and maximum value. We have written a fairly general continuous contract that includes allocation for a current value and a minimum and maximum value that the icon can attain. It also provides ability to constrain a new value to lie between these limits. The individual icon's job is then to specify how to display legal values.

Generic facilities such as the continuous property mentioned above are implemented through *Mixins*. Mixins are just flavors that are mixed into icons instead of being instantiated themselves. The dial, for example, inherits many of its properties from a set of mixins. It is composed of a dial component which supplies dial specific information and eight other mixins which provide the additional templates and messages required (Figure 7).

One of a dial's more important mixins is the *Rectangular* mixin which provides the strategies for dealing with rectangular regions on the color screen. It gives the icon a bounding rectangle and contracts for dealing with it. Consider placing a new icon on the color screen. The editor creates a new instance of the icon type and sends that icon the message *Setup-Position*. Here, the contract is for the cursor to move to the color screen where the icon is to locate itself. For most icons, the user would click at the starting position and pull out a rectangle to graphically delimit the region for the icon. Each icon could do this individually, but clearly this is a generic function. The Rectangular mixin provides this functionality along with *Erase* and *Move* messages. The Erase message clears the icon's bounding rectangle, while the Move message changes the icon's screen position. An icon can, if necessary, override the default action that any mixin provides. These object oriented programming techniques make possible a very powerful generic interface which has proven to be exceedingly useful in building simulation interfaces.

Tapping Icons into a Simulation

The process of associating an icon with a variable in a simulation is known as *tapping*. To support the two way interface mechanism, icons must not only have the ability to reflect the state of variables but also must provide a means of changing the value of variables. The *Tap* mixin provides this functionality. Clicking on Tap in the Graphics Editor display pops up a menu for critiquing the tapping parameters of a selected icon. This menu allows the specification of the variable whose value will be monitored by the icon, the *probe variable,* and the variable that will be altered by interacting with the icon, the *set variable*. Figure 8a shows the pop-up menu for tapping a dial. Notice that a user has indicated this dial is to probe the variable DB2C and set the variable EB2C.

The tapping mechanism allows the designer to specify a mapping from the math model's representation to one that is closer to the way a person would talk about the component's state. For example, in Steamer a rotary pump can be in a secured, operating, or warmup state. In the mathematical model, these states are represented by the values 0, 1, or 2, respectively. The designer can specify an automatic mapping between the math model values and the more easily understood representation. Figure 8b shows the pop-up menu for tapping a rotary pump. Notice that a user has clicked on *secured-warmup-operating* in the *tap mapping* line of the menu. Facilities exist for adding additional mappings for icons.

In the simplest case of tapping, there is a variable in the math model which refects the characteristic we want an icon to depict. In this case the icon is tapped to that variable by entering the variable name in the tapping menu. Sometimes, however, there is no specific variable in a math model which reflects the characteristic we want the icon to depict. In this case, one might decide to add code directly to the math model to provide a representation of the desired characteristic. This is a solution, but not necessarily the optimal one. Since the expertise in building a math model is not necessarily related to the ability to design diagrams to represent the system, a diagram builder might corrupt the math model used by all diagrams.

The mechanism Steamer provides is called a *Model Augment.* It is a set of variables and a function. The augment is associated with a diagram and its function is run along with the math model when the diagram is being viewed. Often, a model augment is used to derive values for new variables based on existing ones in the math model. Just as with variables in the math model, icons can tap into these model augment variables. In Steamer, pipes are often tapped to variables in a model augment which calculate flow based on existing variables, such as valve and pump states. Through the use of model augments, a simulation can be enhanced where it is inadequate, complex tapping code can be more conveniently written, and stand-alone simulations can be implemented. The tapping options are summarized in Figure 9.

Summary

The Graphics Editor is a generic tool for constructing dynamic views and interfacing them to dynamic processes. These processes can either be mathematical simulations, as in Steamer, or real-time systems. Figure 10 shows a real-time interface to Unix running on a Vax 11/780. It graphically depicts system load, paging activity, and other dynamic aspects of the operating system. We have also found the editor to be valuable in assisting us in the development of simulation models. Numerous model augments for Steamer, small simulations in themselves, were developed, tested, and debugged with the aid of the Graphics Editor. Making the normally invisible simulation more visible allows for a better understanding of the underlying processes for students and interface designers alike.

The quick prototyping of interfaces and their subsequent modification with the Graphics Editor make interface design easier, more flexible, and doable by computer naive individuals. In fact, most of the diagrams in Steamer have been created or refined by propulsion system experts with no previous computer training. Through use of the Graphics Editor we have been able to provide non-programming domain experts with the capability to create interactive inspectable interfaces.

We are currently in the process of building other tools to assist in the construction of graphical interfaces. In one effort we are building an Icon Editor to facilitate construction of new icons. This editor is intended to allow a nonprogrammer to construct new graphical icons and to specify new behaviors for them. This should greatly expand the range of domains in which the Graphics Editor can be used. In another project we are building a system to assist in the process of graphical design. This system, Designer, will be able to critique diagrams in terms of graphic design principles, enforce consistent design specifications across diagrams, and help a user explore design alternatives.

References

Dahl, O-J., & Nygaard, K. (1966). Simula - An Algol-based simulation language. *Communications of the ACM, 9 (9),* 671-678.

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation,* Addison-Wesley, Reading, Mass.

Hollan, J. D. (1984). Intelligent Object-Based Graphical Interfaces, 293-297 in G. Salvendy (Ed.) *Human-Computer Interaction,* Elsevier, Amsterdam.

Hollan, J. D., Hutchins, E. L., & Weitzman, L. (1984). STEAMER: An Interactive Inspectable Simulation-Based Training System. *AI Magazine, 5 (2),* 15-27.

Hollan, J. D., Hutchins, E. L., Rosenstein, M., & Weitzman, L. (1984). Tools for Graphical Interface Design, *Combining Human and Artificial Intelligence: A New Frontier in Human Factors,* Proceedings from the Human Factors Society, New York, November 15, 1984.

Sheil, B. (1983). Power tools for programmers. *Datamation, 29,* 131-144.

Weinreb, D., & Moon, D. (1981). *Lisp Machine Manual,* Symbolics, Inc., Cambridge, Massachusetts.

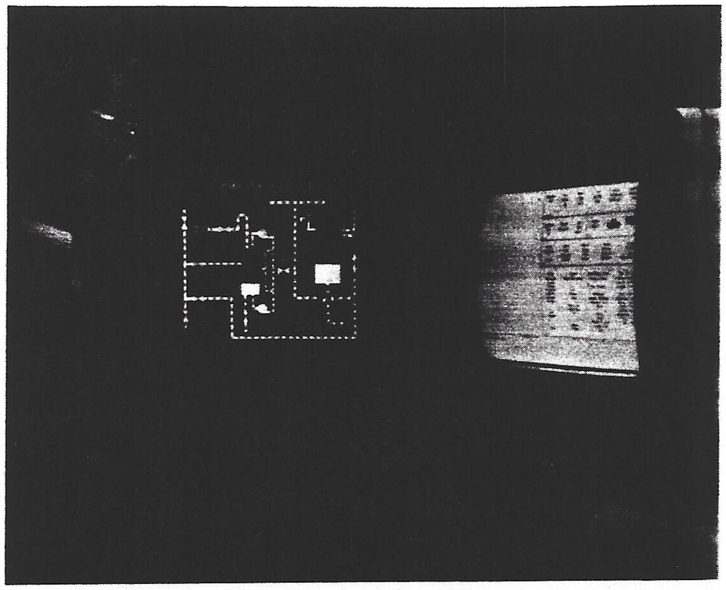


Figure 1. Typical Graphics Editor configuration.

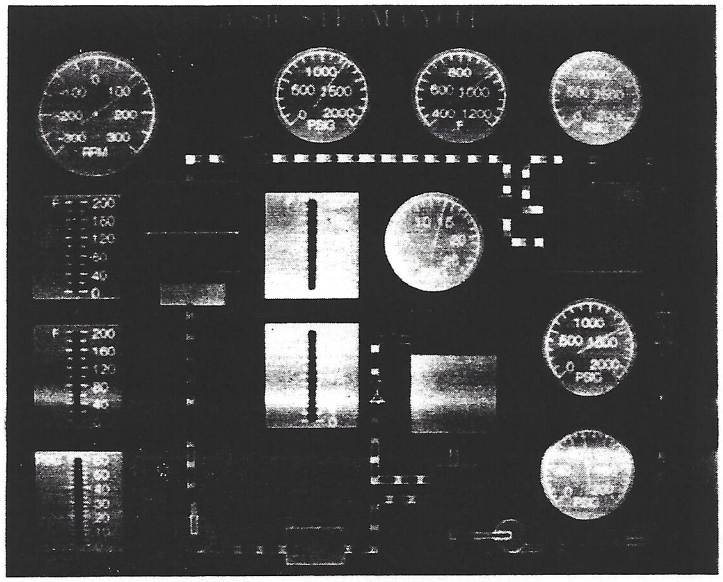


Figure 2. Abstract, high level depiction of the Basic Steam Cycle.

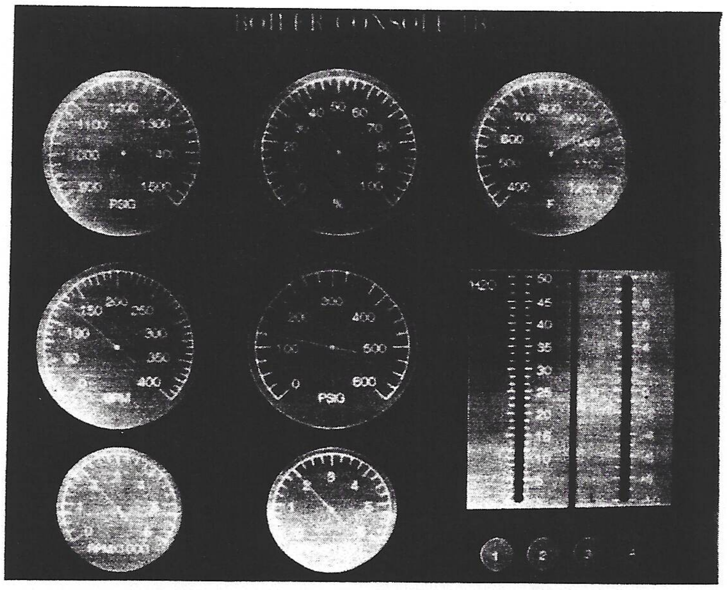


Figure 3. Boiler Console 1b.

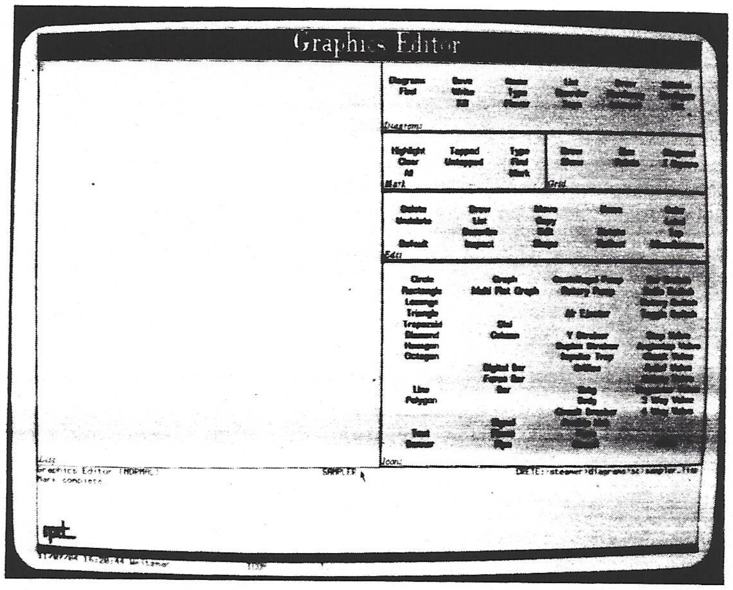


Figure 4. Graphics Editor command pane.

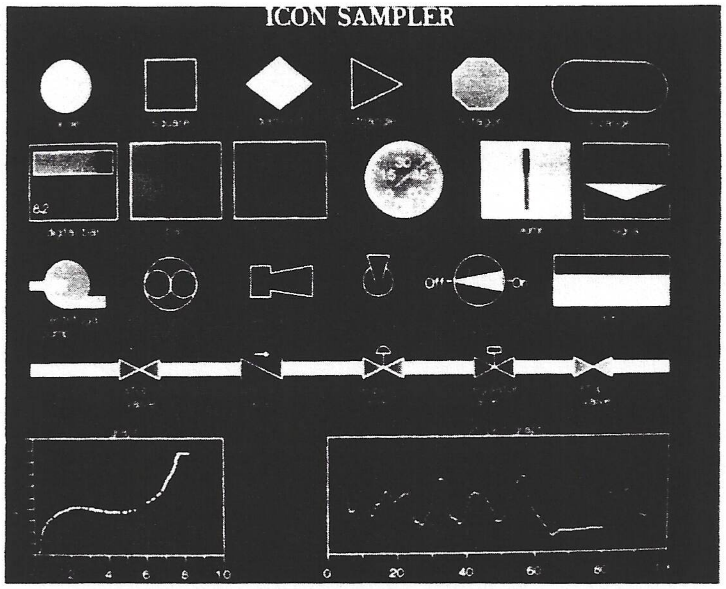


Figure 5. Sampler Diagram.

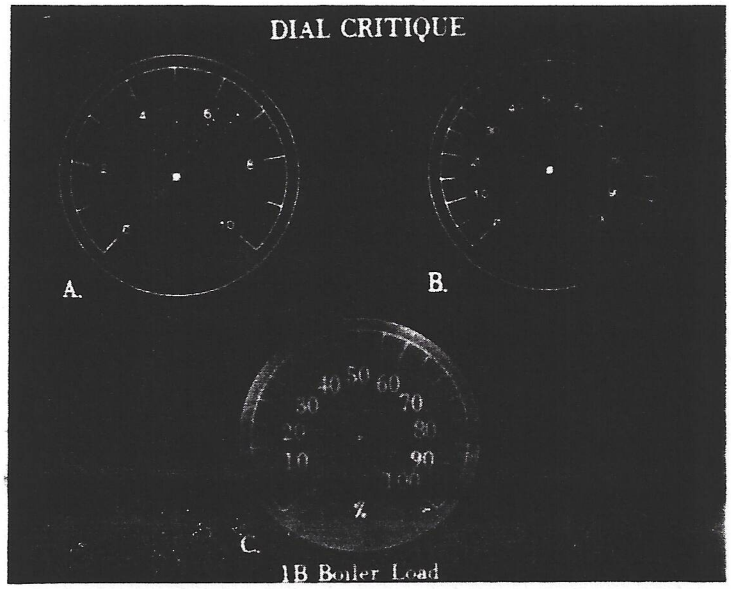


Figure 6. Critiquing a dial. **a).** The default dial. **b).** Dial after critiquing dial's range. **c).** Completed dial including label, units, and face color.

SQUARE-ASPECT-RATIO-MIXIN

INSTANCE VARIABLES:	METHODS:
	:aspect-ratio

BASIC-ICON

INSTANCE VARIABLES:	METHODS:
diagram	:set
	:inverse-probe
	:probe
	:show
	:to-show
	:tapped?
	:recompute-derived-parameters
	:modify-mix
	:change-flavor
	:remake-form
	:make-plist
	:get-component-flavors
	:get-all-instance-variables

GAGE-MIXIN

INSTANCE VARIABLES:	METHODS:
face-color	:setup-label
label-color	:before :recompute-derived-parameters
label-position	:things-to-save
tics	
tic-labels-color	
tic-labels-font	
units-color	
units-font	
units-string	

DIAL

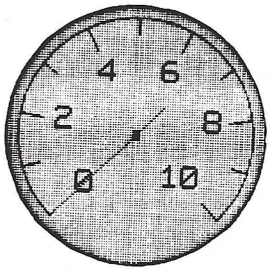
INSTANCE VARIABLES:	METHODS:
arc-start	:setup-miscellaneous
arc-end	:setup-color
needle-color	:to-show
radius	:draw
redline-value	:show
	:draw-needle
	:draw-units
	:draw-tic-labels
	:draw-tics
	:after :recompute-derived-parameters
	:things-to-save

NO-CENTER-LABEL-MIXIN

INSTANCE VARIABLES:	METHODS:
	:legal-label-positions

DISPLAY-MIXIN

INSTANCE VARIABLES:	METHODS:
locations	:wrapper :setup-miscellaneous
	:wrapper :setup-rotation
	:wrapper :setup-tap
	:wrapper :setup-label
	:wrapper :setup-color
	:before :setup-position
	:before :setup-move
	:wrapper :setup
	:wrapper :highlight
	:wrapper :draw
	:wrapper :show
	:wrapper :erase
	:animate
	:set-location
	:location
	:wrapper :zoom
	:zoom
	:highlight-momentarily



CONTINUOUS-MIXIN

INSTANCE VARIABLES:	METHODS:
value	:setup-tap
min-value	:inverse-show
max-value	:show-new-value?
fractional-change-to-show	:constrain-value
range	:before :recompute-derived-parameters
	:things-to-save

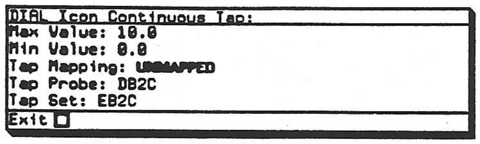
TAP-MIXIN

INSTANCE VARIABLES:	METHODS:
tap-set	:tapped?
tap-probe	:set?
tap-reading	:probe?
tap-mapping	:inverse-probe
tap-map	:set
	:probe
	:reading
	:compute-tap
	:compute-mapping
	:compute-set
	:compute-probe
	:after :init
	:things-to-save
	:after :set-tap-mapping
	:set-tap-mapping
	:after :set-tap-probe
	:set-tap-probe
	:after :set-tap-set
	:set-tap-set

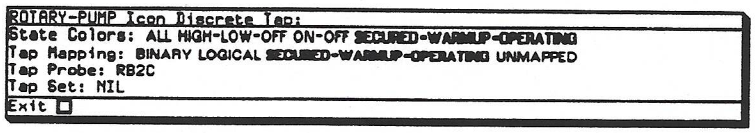
RECTANGULAR-MIXIN

INSTANCE VARIABLES:	METHODS:
outline-color	:setup-color
label-string	:setup-label
label-orientation	:after :reflect
label-position	:after :draw
label-font	:erase-label
label-color	:draw-label
matrix	:draw-label-aux
inverse-matrix	:label-position-aux
xl	:legal-label-positions
yb	:edit-position
xr	:move-relative
yt	:setup-move
xc	:setup-position
yc	:setup
dx	:latch
dy	:normalize
	:aspect-ratio
	:highlight-points
	:highlight
	:before :erase
	:erase
	:border
	:set-bounding-rectangle
	:extended-bounding-rectangle
	:bounding-rectangle
	:claim-rectangle
	:claim
	:before :recompute-derived-parameters
	:things-to-save

Figure 7. The instance variables and methods of mixins which combine to form a dial.



```
DIAL Icon Continuous Tap:
Max Value: 10.0
Min Value: 0.0
Tap Mapping: [✓ UNMAPPED]
Tap Probe: DB2C
Tap Set: EB2C
Exit ☐
```



```
ROTARY-PUMP Icon Discrete Tap:
State Colors: ALL HIGH-LOW-OFF ON-OFF [✓ SECURED-WARMUP-OPERATING]
Tap Mapping: BINARY LOGICAL [✓ SECURED-WARMUP-OPERATING] UNMAPPED
Tap Probe: RB2C
Tap Set: NIL
Exit ☐
```

Figure 8. Tapping menus. **a).** Tapping a dial to variables in the math model. **b).** Tapping a rotary pump using a variable transformation mapping.

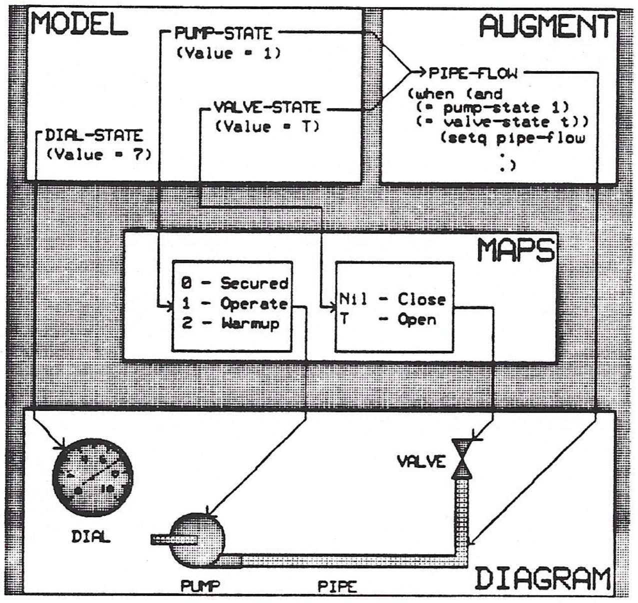


Figure 9. Summary of tapping mechanisms.

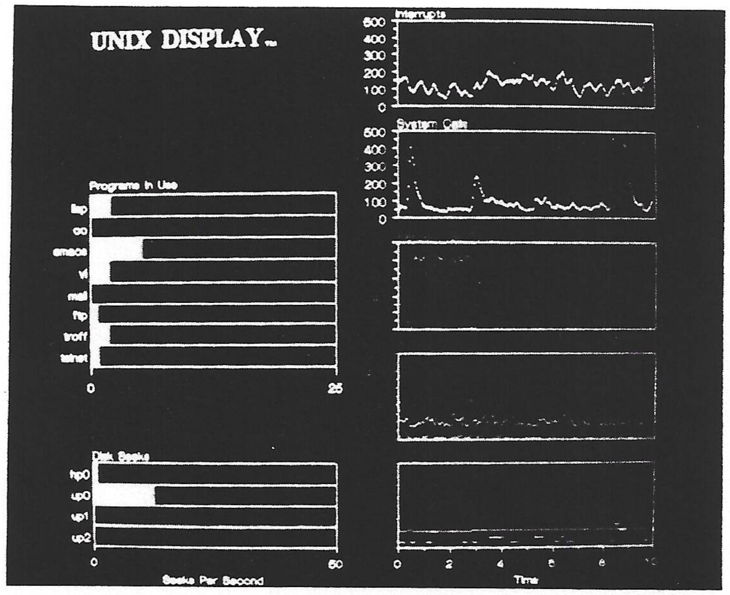


Figure 10. Real time interface to an operating system.