GRAPHICAL INTERFACES FOR SIMULATION

J. D. Hollan, E. L. Hutchins, T. P. McCandless, M. Rosenstein, and L. Weitzman

May 1986

ICS Report 8603
GRAPHICAL INTERFACES FOR SIMULATION

J. D. Hollan, E. L. Hutchins, T. P. McCandless, M. Rosenstein, and L. Weitzman

May 1986
ICS Report 8603

NPRDC-UCSD Intelligent Systems Group
Institute for Cognitive Science
University of California, San Diego
La Jolla, California 92037

To be published in W. B. Rouse (Ed.), Advances in man-machine systems (Vol. 3), Greenwich, CT: Jai Press.

This research was sponsored by the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval Research, under Contract No. N00014-85-C-0133, Contract Authority Identification Number, NR 667-541. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the sponsoring agencies. Approved for public release; distribution unlimited. Reproduction in whole or in part is permitted for any purpose of the United States Government. Requests for reprints should be sent to James D. Hollan, NPRDC-UCSD Intelligent Systems Group; Institute for Cognitive Science, C-015; University of California, San Diego; La Jolla, CA 92037.

Copyright © 1986 by James D. Hollan, Edwin L. Hutchins, Timothy P. McCandless, Mark Rosenstein, Louis Weitzman
**Title:** Graphical Interfaces for Simulation

**Personal Author(s):**

James D. Hollan, Edwin L. Hutchins, Timothy P. McCandless, Mark Rosenstein, Louis Weitzman

**Type of Report:** Technical

**Time Covered:** From Apr 86 TO Apr 86

**Date of Report:** May 1986

**Source of Funding Numbers:**

<table>
<thead>
<tr>
<th>PROGRAM ELEMENT NO.</th>
<th>PROJECT NO.</th>
<th>TASK NO.</th>
<th>WORK UNIT</th>
<th>ACCESSION NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>61153N</td>
<td>RR04206</td>
<td>RR04206-0A</td>
<td>N00014-85-C-0133</td>
<td></td>
</tr>
</tbody>
</table>

**Supplementary Notation:** To be published in W. B. Rouse (Ed.), Advances in man-machine systems (Vol. 3). Greenwich, CT: Jai Press.

**Abstract:**

The dynamic graphical display of the state of complex systems has immense potential value for mediating the development of richer understandings of process as well as for providing more effective mechanisms of interaction. For the past few years we have been involved in the development of a set of software tools to assist in the construction of interfaces to simulations and real-time systems. These tools have been used extensively in the development of an interactive inspectable simulation-based instructional system, Steamer. (Hollan, Hutchins, Weitzman, 1984). Underlying our efforts are three interrelated research activities: formulating a theory of interface design, understanding the effectiveness of interactive graphical representational systems, and implementing systems based on these developing theoretical notions. The dialectic between these activities has been very valuable for us as cognitive scientists and as system builders. In this chapter, we survey some of the presuppositions of our approach to interface design, describe the tools we have built to assist in the construction of graphical interfaces, and discuss the conclusions we have drawn from our experiences in graphical interface design.
# Contents

INTRODUCTION ................................................................................... 1  
UNDERLYING PERSUPTIONS .............................................................. 4  
TOOLS FOR CONSTRUCTING GRAPHICAL INTERFACES ......................... 7  
  Simulation Environment ....................................................................... 7  
  Model Control .................................................................................... 8  
  Graphics Editor .................................................................................. 9  
  Icon Editor .......................................................................................... 12  
  Knowledge-Base Editors ..................................................................... 15  
    Knowledge Editor and Grapher .......................................................... 15  
    Lesson Editor .................................................................................. 19  
    Behavior Editor .............................................................................. 21  
    Designer ........................................................................................... 22  
CONCLUSIONS .................................................................................... 25  
ACKNOWLEDGMENTS .......................................................................... 26  
REFERENCES ..................................................................................... 27
Graphical Interfaces for Simulation

JAMES D. HOLLAN, EDWIN L. HUTCHINS, TIMOTHY P. MCCANDLESS, MARK ROSENSTEIN, and LOUIS WEITZMAN

INTRODUCTION

In order to illustrate the type of graphical interface with which we are concerned, we begin by briefly describing Steamer, a system which employs an interactive graphical interface to a steam propulsion plant simulation. Steamer is a research project involved with evaluating the potential training applications of techniques from the new disciplines of artificial intelligence and cognitive science. While the project addresses a host of research issues ranging from how people understand complex dynamic systems to how AI software and hardware advances might be applied to training, it is focused around the construction of a computer-based system to assist in propulsion engineering instruction. The goal of the project is not only to build a training system with tutorial and explanation facilities but also to construct a set of software tools to assist in the implementation of simulation-based training systems and graphical interfaces.

Steamer currently consists of a graphical interface to a mathematical model of a steam propulsion plant. The interface allows a user to select from a library of propulsion plant views and interact with a selected view to change the state of the underlying simulation model. The evolution of plant states can be observed by graphical changes in the view on a color display. Views depict aspects of the propulsion system at various levels of detail. They vary from collections of gauges and indicators typically found in a real plant to schematic diagrams specifically designed to depict models similar to those experts seem to employ in reasoning about the operation of the propulsion plant. The potential for increased instructional effectiveness derives from representations with the ability to show global views of systems that are physically dispersed in the actual plant and thus difficult to see as a total system, to show simplified versions of systems designed to be easier to understand or to provide better models for reasoning about the plant, to look "inside" systems or components and see flows or other internal characteristics, and to make available indicators that depict aspects of the operation of a system not normally available but that are useful in developing an understanding of a system.

Figures 1 through 4 are black and white renditions of views a user of Steamer would see on a color graphics screen. State information is depicted by color, by animation, and by analog, digital, and textual changes. For example, operational status of a pump or valve is indicated by color (red for off; green for on); flow rates in pipes are dynamically shown by use of animation techniques; dials and graphs reflect plant parameters. The iconic representation serves both to provide state information and as a mechanism for changing state of an underlying simulation. By pointing to a component with a mouse-controlled cursor, a user can change its state by clicking on it. For example, clicking on a pump will toggle its state. Similarly one can vary the level of a tank, change the value of a dial, or position a throttle. Of course, the nature of the underlying simulation and the goals of the interface designer determine which variables and thus which components can be manipulated by a user. The important
point here is that the interface functions as a two-way communication device: depicting and allowing changes of state.

A high-level view of the Main Engine Lube Oil System is depicted in Figure 1. One can watch the state of the lube oil system and its responses to changes made to the propulsion system. The flows in the system are dynamically depicted by animation within pipes; the connectivity of the system is shown. The states of the lube oil service pumps (LOSPIA-C) are indicated by color changes. In Figure 1 the attached pump LOSPIC is operating, LOSPIB is off, and LOSPIA is operating at low speed (colors in the figure are represented by differing gray shades). A series of pressure sensors is shown at the lower right of the view, along with digital and analog representations of lube oil pressure. These sensors monitor pressure at the bearing most distant from the pumps and will automatically control the state of the service pumps if pressure drops below specified levels. The valve at the top left is the lube oil unloader valve and the column above it indicates how far it is open. As pressure in the system rises above a set threshold, the unloader valve opens and unloads lube oil back into the sump, preventing

![Main Engine Lube Oil System Diagram](image)

**FIGURE 1.** *Main Engine Lube Oil.* The lube oil system is distributed throughout the propulsion plant. This view provides a global view of its operation. Like most figures in this chapter, it is a black and white rendition of a view normally presented on a color screen. Dithering techniques have been used to map from colors to stipple patterns. The boxes shown within pipes are used on the color screen to provide animation of flows.
overpressurization. There are two controller boxes and a switch next to lube oil service pumps 1A and 1B. By touching the switch the system can be put in manual mode, and the controller boxes for the pumps can be touched and operated to change pump speed to high (H), low (L), or stop (S). As the controller is operated, the associated pump will change state and the ramifications of that change will be continuously reflected in other portions of the view.

Figure 2 shows a Throttle Board view that allows the user to control the Ahead and Astern throttle and monitor a number of important system parameters. Figures 3 and 4 depict portions of the Feed System. Figure 3 shows the states of the two boilers (1A and 1B), the level of the deaerating feed tank (DFT, a water storage tank), the states of the six pumps involved in the system, and the large number of valves used to control and direct the feed water to the boilers, as well as key system parameters.

The same type of graphical interface can be provided for a real-time simulation. For example, Figure 5 contains a view we designed to monitor the dynamic state of one of the computers on our local area network. It shows the number of users running various programs and continuously graphs interrupts, system calls, cpu idle time, characters in and out, users on the system, and processes waiting to be run.
These examples are intended to illustrate the type of interactive graphical interfaces with which we are concerned. The key notion is that the interface serves as a communication device to allow a user to see and manipulate the state of an underlying simulation or real-time system. In the Steamer application we have been concerned with using this form of interface in training. It should be clear that it is just as applicable for monitoring or controlling a system. Our primary concerns here are to discuss why we think this form of interactive graphical interface is powerful and describe a set of tools we have implemented to facilitate interface implementation.

UNDERLYING PRESUPPOSITIONS

A common way of describing the class of interfaces discussed above is as direct manipulation interfaces (Shneiderman, 1982). In our view, most of the treatments of direct manipulation interfaces focus at the wrong level of analysis. The naive notion seems to be that the key properties are bit-mapped displays and pointing devices. One of our presuppositions is that interfaces are representational systems designed for communication, and just as in the analysis of any other representational system, it is
FIGURE 4. Make-up and Excess Feed. This is another portion of the feed system designed to show control relationships between tank levels and states of associated valves and pumps.

essential to understand the cognitive task that the system is attempting to support. It is an egregious error to suppose that one can discuss interface design outside of the cognitive contexts of the task domain in which an interface is embedded. The directness associated with an interface comes from how directly the interface supports the user's cognitive task. Graphical displays and pointing devices are media of support but do not in themselves guarantee any directness. Directness results when the interface language closely matches the way in which a user thinks of a task. Directness is thus not a property of interfaces but involves the relationship between the task a user has in mind and the way in which the task can be accomplished via the interface.

An interface provides a language for the user to communicate with a system and for the system to communicate with a user. A key notion for us is the relationship between the meaning of an expression in the interface language and what the user wants to say. We have termed this relationship Semantic Distance (Hutchins, Hollan, & Norman, 1986). The extent that an interface language allows one to say what one wants to say without circumlocutions is the measure of its semantic directness. Another aspect of directness is the relationship between the meanings of expressions in the interface language and their physical forms. We call this Articulatory Distance (Hutchins, Hollan, & Norman, 1986). Nonarbitrary relationships between meaning and the way it is physically expressed provide this form of directness. A nonarbitrary graphical relationship between icons and what they depict is an example of articulatory directness. Similarly, on the input side of an interface language, interfaces that allow one to make statements about position by pointing provide examples of articulatory directness.
UNIX™ DISPLAY

Programs in Use

<table>
<thead>
<tr>
<th>Program</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>lisp</td>
<td>10</td>
</tr>
<tr>
<td>cc</td>
<td>2</td>
</tr>
<tr>
<td>emacs</td>
<td>5</td>
</tr>
<tr>
<td>vi</td>
<td>3</td>
</tr>
<tr>
<td>mail</td>
<td>2</td>
</tr>
<tr>
<td>troff</td>
<td>1</td>
</tr>
<tr>
<td>ftp</td>
<td>1</td>
</tr>
<tr>
<td>telnet</td>
<td>0</td>
</tr>
</tbody>
</table>

Disk Seeks

<table>
<thead>
<tr>
<th>Disk</th>
<th>Seeks</th>
</tr>
</thead>
<tbody>
<tr>
<td>hp0</td>
<td>10</td>
</tr>
<tr>
<td>up0</td>
<td>5</td>
</tr>
<tr>
<td>up1</td>
<td>2</td>
</tr>
<tr>
<td>up2</td>
<td>1</td>
</tr>
</tbody>
</table>

System Calls

<table>
<thead>
<tr>
<th>System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
</tr>
<tr>
<td>400</td>
</tr>
<tr>
<td>300</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Percent Idle Time

<table>
<thead>
<tr>
<th>Percent Idle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>70</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

TTY Characters Out

<table>
<thead>
<tr>
<th>TTY Characters Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
</tr>
<tr>
<td>400</td>
</tr>
<tr>
<td>300</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

TTY Characters In

<table>
<thead>
<tr>
<th>TTY Characters In</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
</tr>
<tr>
<td>400</td>
</tr>
<tr>
<td>300</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Users

<table>
<thead>
<tr>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Run Queue

<table>
<thead>
<tr>
<th>Run Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FIGURE 5. Unix Display. A view designed primarily to illustrate the connection of a graphical view to a real-time system. It provides a summary of the state of an operating system running on one of the computers on our local area network.

Thus, one of the important underlying presuppositions of our approach is to view an interface as a representational language and to place primary importance on the cognitive task that a user is attempting to accomplish. Another fundamental presupposition of our approach is the view that graphical forms of representation provide powerful ways of bringing abstract things into the realm of the perceptually knowable. We think there are important cognitive properties that make graphical representations effective. These properties derive from the types of processing activities people are especially good at: detecting patterns, constructing mental models or simulations of the world which support causal reasoning, and manipulating the world by actions on it or on representations of it.

Rumelhart, Smolensky, McClelland, and Hinton (1986) have argued that one of the effective problem-solving strategies people employ involves the creation of artifacts, physical representations that can be manipulated to get answers to questions. They suggest that the underlying strategy is to make "problems conform to problems we are good at solving" and argue that:

We are especially good at pattern matching. We seem to be able to quickly "settle" on an interpretation of an input pattern. This is an ability central to perceiving, remembering, and comprehending. Our ability to pattern match is probably not something which sets humans apart from other animals, but is probably the essential component to most cognitive behavior.

We are good at modeling our world. That is, we are good at anticipating the new state of affairs resulting from our actions or from an event that we might observe. This ability to
build up expectations by "internalizing" our experiences is probably crucial to the survival of all organisms in which learning plays a key role.

We are good at manipulating our environment. This is another version of man-the-tool-user, and we believe that this is perhaps the crucial skill which allows us to think logically, do mathematics and science, and in general to build culture. Especially important here is our ability to manipulate the environment so that it comes to represent something. This is what sets human intellectual accomplishments apart from other animals. (Rumelhart et al., 1986, pp. 44-45)

If these are indeed the types of activities that people are especially good at, there are clear implications for why graphical interfaces may have important cognitive properties. They provide a physical representational system that permits us to make abstractions perceptually available and thus allow the use of our powerful pattern-matching abilities. They make possible the depiction of models of the world that are similar to the mental models or simulations people seem to use to reason about the world. These models can depict physical state information, causal connections, and are runnable, permitting a user to see the effects that result from changes of state. Finally, graphical interfaces provide the potential of directly manipulable representations of systems. These factors comprise another set of presuppositions underlying our approach to graphical interface design and also provide motivation for our interest in simulation-based systems.

TOOLS FOR CONSTRUCTING GRAPHICAL INTERFACES

There is a need for software tools to assist in the creation of graphical interfaces. Here we describe a set of tools we have been evolving to facilitate interface design. First we describe a general simulation environment, which consists of a Model Controller and a Graphics Editor. This is the core of the system we are developing. It permits one to build interactive graphical interfaces to simulation models or real-time systems. The Graphics Editor makes available a set of icons, facilities for modifying characteristics of icons (e.g., size, shape, color, and placement), and the ability to associate icons with model variables so that the icons reflect the values of the variables and so that one can interact with the icons to change the values of their associated variables. The Model Controller allows one to run simulation models, observe the model's state via graphical views constructed with the Graphics Editor, and interact with the views to change the state of a simulation model.

We also will describe a number of related tools we are developing to support the construction of interfaces. Chief among them is an Icon Editor, which makes possible the construction of new icons without requiring a user to operate at the level of code writing. In addition we discuss a series of Knowledge-Base Editors for the specification of domain knowledge. We have implemented a Behavior Editor to explore the incorporation of simulation knowledge into icons and are in the process of developing a Lesson Editor to explore the incorporation of domain knowledge into graphical views so that they can explain themselves, pose problems to students, and monitor their answers. Designer, an interactive visual design consultant for users of the Graphics Editor, makes available graphical design knowledge during the process of constructing and critiquing graphical views. Underlying a number of these knowledge-base editors is a frame-based representational language. We will also briefly describe it and a graphical interface to it.

Simulation Environment

The Simulation Environment we have designed consists of a set of activities to allow users to build, observe, and manipulate views of a simulation model or real-time process. In our work we have used this facility to build interfaces to mathematical simulations such as the steam propulsion simulation...
used in Steamer, real-time systems, and Parallel Distributed Processing (PDP) models learning to recognize patterns in the operation of underlying systems. A system consists of a process and a set of user defined views connected to that process.

One interacts with a simulation system via its associated views. A view is a graphical collection of icons representing a portion of a simulation. We have designed a Model Controller to allow users to manipulate views, observe the effects of the manipulations, and control the underlying simulation model. Using this controller, a user selects two views. One is typically a view used to control global aspects of a system, while the other is used to manipulate and observe subsystems. In Steamer, for example, the global view contains the throttles for the ship and important status information and the other view can be alternated between any of the approximately one hundred views available.

Each simulation environment activity, such as editing or running a simulation model, is supported by a screen configuration which provides a set of functions and menus to help a user accomplish the tasks associated with the activity. Integrated into the current simulation environment are the Model Control and Graphics Editor activities as well as a related set of activities discussed below.

Model Control

The Model Control activity provides facilities for changing systems, models, and views. It also allows modification of the behavior of a model, such as the rate it runs in the case of a simulation, or the rate an interface is sampled in the case of a real-time system. Figure 6 shows a typical configuration for the Model Controller. The status line near the bottom of the screen maintains current state

![Figure 6. Model Controller. The screen configuration of the black and white display during operation of a simulation. It provides functions for controlling the running of a simulation, switching between graphical views, and selecting other activities. Typically used while viewing and interacting with a view on the color screen.](image)
information. In this case the Steamer system has been selected and its associated model is running with the Make Up and Excess Feed view displayed on the color screen. The right half of the middle section of the screen shows a Steamer control view. At the bottom of that view is the ahead throttle, and immediately above that are important data, such as the ship's speed, engine rpm, and the fact that the ship is currently operating with one boiler. Above this information are other indicators of the global state of the propulsion system. Across the top of the screen are menus of operations available for controlling the model and views. The right most menu choice allows changing activities.

The status line allows a standard interface to a set of general control functions. The way to view the status line is as a display of attribute value pairs that describe the system at a given time. The attributes explicitly displayed are the current system, model, subsystem, and view. An asterisk is used to indicate a value that has been modified. Whether a model is running or not is shown in square brackets.

**Graphics Editor**

The Graphics Editor originated from our work on Steamer (Hollan, Hutchins, & Weitzman, 1984) and the requirement to implement a large number of dynamic views of a propulsion system. We needed a tool which would allow instructors, who were knowledgeable about a domain but computer naive, to create graphical interfaces. The resulting Graphics Editor has been used to create more than a hundred views for Steamer and has been designed to allow its easy extension into other domains. The editor provides a user with a set of icons that can be arrayed on a graphics screen to create a view of an underlying simulation or real-time system. It provides functions commonly available in computer-aided design systems. One can save and restore views from files, mark the elements of a view (individually, by type, within an area, etc.), and edit those marked elements (moving, copying, deleting, changing color, size, etc.). A grid facility is provided to assist in accurately positioning icons within a view.

The multipaned menu interface to the editor activity is shown in Figure 7. The status line in the lower portion of the screen provides control facilities similar to those in the Model Control activity. In constructing a view, a user chooses icons from the menu of icons and positions them on the color screen. Figure 8 depicts a subset of the available icons. Typically a process of incremental refinement of the view takes place in which icons are moved around, reshaped, colored, and given labels and appropriate units.

An important characteristic of the Graphics Editor is the way in which it supports the association of icons to underlying variables in a mathematical model or real-time system. We call this process tapping. There are two types of taps. A probe tap associates a variable with an icon so that the icon reflects the current state of that variable. A set tap also associates a variable with an icon but in a way that allows one to change the value of the variable by interacting with the icon. Figure 9 illustrates the tapping mechanism and a simple math model. On the left are the variables in a simple math model and on the right are the icons to which they have been tapped. The toggle switch turns this model on and off. Each tick of the clock indicates that one unit of time has passed in the simulation. While the model is running, the valve sets its variable. This means that when one interacts with the column above the valve, it sets the value of its variable, %-VALVE-OPEN, to reflect the valve's degree of openness. The pipe, on the other hand, probes its value, PIPE-SPEED. This means that on each tick of the clock, the pipe reflects the value of PIPE-SPEED by changes in flow rate. When one interacts to change the state of the valve, the effect propagates via the math model and affects pipe flow. Also included in the simulation are the clock and toggle switch themselves. The clock probes its value, CLOCK-STATE, while the toggle switch sets its value, MODEL-RUNNING.

To set up the tappings of the pipe, a user of the graphics editor would mark the pipe icon, then click on Tap. This generates a pop-up menu (shown in Figure 10) for specifying the tapping parameters. Here the tap probe has been associated with the variable PIPE-SPEED. This will cause the pipe to probe that variable when run. Similarly, values of other tapping parameters are provided. The variable
-VALVE-OPEN would be entered as the tap set for the column associated with the valve so that it would be set in the model when a user interacted with it. Notice the tap mapping line for the toggle switch in Figure 11. The mapping mechanism allows the designer to map the icon's value from one scale to another. Selecting logical would map the "on" state to true and the "off" state to nil. Another choice is binary, which maps the "on" state to 1 and the "off" state to 0. All icons that depict a state or change a state of a math model variable are tapped in this manner.

An associated facility to aid the tapping process is the model augment. There are occasions when a math model insufficiently represents aspects of a simulation that a designer might wish to display. An augment allows one to enhance the simulation model where it is inadequate, to conveniently write more complex tapping code, and to provide stand-alone simulations for views. For example, the Steamer math model does not represent flow in pipes. Since we think that flow is important in understanding causality in a steam system, we often add a model augment to a view so the iconic representations of pipes depict flow rates. A model augment also allows the builder to derive values from existing variables. In the simple model augment shown in Figure 9 for the pipe and valve system, we wanted to relate pipe flow to the state of the valve: the more open the valve, the more flow. The model sets the pipe speed to be the openness of the valve divided by 100.

In designing and implementing the editor we have capitalized on the flexible object-oriented Flavors System of Zetalisp. What is created as a result of the view construction process is a Lisp program that contains a number of dynamic entities capable of responding to messages and of providing graphical support to an interactive instructional system. For example, consider a dial. Many of the properties of the dial actually come from simpler objects which can be used in a variety of icons. Figure 12 shows some of the component pieces of a dial: RECTANGULAR, CONTINUOUS, and TAP mixins. For any icon that displays continuous values, we mix in the object called CONTINUOUS. This object provides

---

**FIGURE 7. Graphics Editor.** Screen configuration of the black and white display for editing a graphical view.
ICON SAMPLER

circle square diamond triangle octagon lozenge

digital bar bar force bar dial column signal
centrifugal pump rotary pump air ejector toggle switch rotary switch tank

stop valve check valve regulator valve regulator valve stop valve

graph multi-plot graph

FIGURE 8. Icon Sampler. A subset of the graphical icons available for use with the graphics editor.

a place to hold the icon’s value and the minimum and maximum values the icon can display. The CONTINUOUS mixin also provides commands, or messages any instance of a dial icon can be sent. CONSTRAIN-VALUE is an example message. It provides a way of constraining a number to be between a minimum and maximum value. Figure 13 shows the complex structure of a dial, including its instance variables and the messages that it can handle. Of course, a user of the Graphics Editor does not need to think in terms of these implementation details but need only be concerned with critiquing visual characteristics of the dial and tapping it into the appropriate variable.

By having knowledge stored locally in icons, the Editor and Model Controller do not need to know about how the dial does its work. Since the editor does not need to know, neither does the view builder. Icons such as dials or pipes understand other basic messages like SHOW which, when received, cause the receiving icon to show the value provided in the message. If we send a dial the message to show the value 7, it reflects that value by positioning its needle. On the other hand, if we send a pipe the same message, it shows this value as flow. In neither case did we explicitly need to know how the icon worked, only that they understood the message sent. These messages make possible a very powerful generic interface ability which has been exceedingly useful in the development of the simulation environment.
Tapping Mechanism

MATH MODEL

Model-Running = ON
Clock-State = 1
Pipe-Speed = 0.72
Percent-Valve-Open = 72.0

GRAPHICAL ICONS

::Model Augment for TAPPING MECHANISM Diagram
(defun tapping-mechanism-model-augment ()
  (when model-running
    (if (< percent-valve-open 4)
        (setq percent-valve-open 0))
    (setq pipe-speed (// percent-valve-open 100.0))
    (if (= clock-state 0)
        (setq clock-state 1)
        (setq clock-state 0))
    (setq valve-state (if (> 0 pipe-speed) :off :on)))))

FIGURE 9. Tapping Mechanism. A depiction of the relationships between variables in a simple mathematical model and a set of icons. Set taps allow the value of a variable to be changed by interacting with an icon. Probe taps cause icons to reflect the value of associated variables. At the bottom of the figure is a simple model augment used with this view.

Icon Editor

While the Graphics Editor allows one to construct views from an existing set of icons, the Icon Editor allows users to construct new icons which can dynamically display and modify the state of a simulation. A user of the Icon Editor specifies both the icon's appearance and behavior. This specification determines the graphical states for the icon. For example, we have seen the appearance of a dial icon in a number of views. The behavior of the dial icon, like a real gauge, comes from the ability to position its needle. For the icons of the Graphics Editor these specifications were made by directly writing Lisp code. Experience with the Graphics Editor has demonstrated that incremental specification and refinement of the appearance of a view is most easily effected through the graphical manipulation of its components, the icons. The Icon Editor mimics this facility by allowing the specification of an icon's
appearance through graphical manipulation of the components of the icon and specification of its behavior through a critiquing process. This permits the development of a hierarchy of graphical primitives for use in the appearance of icons and a set of graphical behaviors for icons. Thus, the Icon Editor is a tool for the construction of icons without explicit programming.

A number of the Steamer icons can be thought of as combinations of existing icons. The digital bar icon in Figure 14 consists of a bar icon, a banner icon and a rectangle icon. To allow positioning these components, the same graphical techniques available in the Graphics Editor, such as marking and moving, are also available in the Icon Editor. The top of Figure 15 shows a flame icon. In a number of views, flickering of this icon is used to depict the flicker of an actual flame. To construct the icon with the Icon Editor, four lines were incrementally added. The Icon Editor thus makes possible the incremental design of new icons from basic components.

When a user builds a view in the Graphics Editor, the system is writing code. This level of activity is entirely invisible to the user. When constructing a view, the user has no feeling of coding, of describing the procedure the computer will follow to reconstruct and run the view. Similarly, with the Icon Editor we don’t want a person constructing an icon to feel that they are writing code. Thus, the Icon Editor must make available an appropriate toolkit of behaviors with semantically direct representation for each behavior.

It is unlikely that there is one general scheme to allow builders of icons access to primitive graphical behaviors. In the digital bar case, the component icons do all the work. The builder specifies the behavior of the digital bar by constraining its tap to be the taps for the bar and banner. In addition, the designer of the digital bar icon must decide which attributes to make available to the user of the icon. In the Icon Editor a pane of attributes for each component is displayed. The user can constrain two attributes to be identical, in a manner similar to the way taps were constrained, or place an attribute in

FIGURE 10. Tapping an Icon. Clicking on the tap menu item pops up a window in which probe and set variables can be specified as well as constraints and mappings of their values.
the color or miscellaneous menus for use in the Graphics Editor. For the digital bar, the bar's color and the banner's text color would be placed in the color menu. Since bar, rectangle, and banner have labels, the icon builder would not put any of the label attributes of the components in the menu. This prevents the user of the icon from accidentally placing labels in incorrect positions. The Icon Editor also provides color and miscellaneous buttons, to allow the builder to check the appearance of the menus as they are built.

The flame icon provides a more interesting case, one in which behavior is synthesized at the level of the new icon. The flickering action of flames is implemented using animation. Rotating the colors of the four lines which comprise a flame creates the flickering effect. A better effect is produced by maintaining a fixed color for one of the lines. At the bottom of Figure 15 are the three possible animation states. When rotation repeats in real time, the flame appears to be flickering. Underlying the animation is a complex negotiation between the icon and the graphics device, but the builder needs only to specify the three rotation colors.

Figure 16 shows an experimental configuration for the Icon Editor in which a flame is being designed. Near the bottom of the figure is the status line showing the current icon (flame). Across the middle of the screen is a pop-up menu an icon user would see in the Graphics Editor for specifying color. The topmost pane allows a icon builder to examine other Graphics Editor menus created from the construction of an icon.

We are in the process of implementing additional behaviors for the Icon Editor. These include allowing an icon to move along a trajectory and to vary the hue of its color to reflect the value of a tapped variable. The initial goal for the Icon Editor is to be able to reimplement the Steamer icons using the Icon Editor. We are using this reimplementations to assist us in identifying a set of primitive components sufficiently rich to permit specification of the diverse set of behaviors exemplified in the

---

**FIGURE 11. Mapping a Tap.** The tapping mechanism allows the mapping of state colors and variable types. Here a mapping has been made between a logical variable and the on-off state.
Internal Icon Structure - Dial

MIXINS

Dial
- Radius
- Arc-Start
- Arc-End
- Draw
- Show

Rectangular
- X1, Y1
- Claim
- Highlight
- Erase

Continuous
- Value
- Min-Value
- Max-Value
- Constrain-Value

Tap
- Set-Form
- Probe-Form
- Set
- Probe

Radius, Arc-Start, Arc-End, X1, Y1, X2, Y2, Value, Min-Value, Max-Value, Set-Form, Probe-Form

Draw, Show, Claim, Highlight, Erase, Constrain-Value, Set, Probe

FIGURE 12. Components of a Dial Icon. Some of the mixins which make up a dial icon. Each mixin contributes instance variables (top half of boxes) and messages (bottom half of boxes).

Steamer icons and as a method for exploring interface techniques for making the primitive behaviors available to a user.

Knowledge-Base Editors

We are in the process of designing and implementing a set of editors to assist an instructional designer or other simulation interface builder in specifying knowledge about a system. A data base of knowledge is required to integrate information about both the domain and the purposes of particular interfaces. This knowledge can be employed to allow simulation views to be able to describe themselves, run scenarios of simulation activities, pose questions to be answered by interacting with a view or collection of views, and to perform other forms of instructional activities. To facilitate the specification of the data base of knowledge we have designed four additional editors: a Knowledge Editor with a graphical interface, a Lesson Editor, a Behavior Editor, and a Graphical Design Editor.

Knowledge Editor and Grapher

We are using a frame-based knowledge representation facility originally designed by Bruce Roberts of BBN, Cambridge, Massachusetts. It essentially builds a class structure on top of Flavors to provide frame-based representational facilities. The underlying language is called MSG, for its flavor enhancing
DIAL

all instance variables (43)

ARC-END (DIAL)
ARC-START (DIAL)
DIAGRAM (BASIC-ICON)
DX (RECTANGULAR-MIXIN)
DY (RECTANGULAR-MIXIN)
FACE-COLOR (GAGE-MIXIN)
FRACTIONAL-CHANGE-TO-SHOW (CONTINUOUS-MIXIN)
FRAME (PICTURE-MIXIN)
INVERSE-MATRIX (RECTANGULAR-MIXIN)
LABEL-COLOR (GAGE-MIXIN)
LABEL-COLOR (RECTANGULAR-MIXIN)
LABEL-FONT (RECTANGULAR-MIXIN)
LABEL-ORIENTATION (RECTANGULAR-MIXIN)
LABEL-POSITION (GAGE-MIXIN)
LABEL-POSITION (RECTANGULAR-MIXIN)
LABEL-STRING (RECTANGULAR-MIXIN)
LOCATIONS (DISPLAY-MIXIN)
MATRIX (RECTANGULAR-MIXIN)
MAX-VALUE (CONTINUOUS-MIXIN)
MIN-VALUE (CONTINUOUS-MIXIN)
NEEDLE-COLOR (DIAL)
OUTLINE-COLOR (RECTANGULAR-MIXIN)
RADIUS (DIAL)

set

VALUES

X (RECTANGULAR-MIXIN)
VALUES

Y (RECTANGULAR-MIXIN)
VALUES

Z (GAGE-MIXIN)
VALUES

FIGURE 13. Dial Icon Details. A listing of the 43 instance variables and 70 messages actually contributed to a dial icon by the full set of its mixins.
capability. Here we present an overview of its representational capabilities and our implementation of a graphical interface to it. The MSG language provides the facility to define classes of objects. Each class defines local attributes that distinguish it from the other classes. It is the instances of these classes that we use to represent the objects in a world being modeled. In the hierarchy of the class structure, a class may have any number of abstractions or superior classes. A given class will inherit
all of the attributes of its abstractions. By defining additional attributes at the local level, a class can be made more specific. The inheritance mechanism allows the inheritance of roles and slots. A role is the semantic organization of a set of attributes, while a slot provides an actual placeholder for an attribute. In addition, the system provides a co-reference facility, the ability to reference the same attribute of a class using multiple descriptions.

When a new class is defined, an instance of a meta-class is created that will hold all pertinent information about the class. This includes how to create a new class instance, where to store these instances, and how to manipulate them. When a new class is defined, a new flavor is also defined. The name of this new flavor is the same as the name of the class being created. When instances of a class are created, an instance of the associated flavor is made. The instance variables of a class provide the typical role and slot descriptors of a frame-based representational language. A role consists of a role name and a list of slots which make up the role. A slot provides a name and the potential of specifying restrictions and default values. In addition, the language provides for subroles to further refine roles. Figure 17 shows an example of the MSG representation of a two port fluid device.

To access the value of a slot of an instance, a path to that slot's value is constructed. For example, the class of fuel-oil-service-pump has an instance called fosp-alpha and a slot called inlet-valve. To return the fosp-alpha's inlet-valve you would construct the path: (the fosp-alpha inlet-valve). This would return the object that is fosp-alpha's inlet-valve. If this value happens to be another object, one can access a slot of the new object by adding onto the path. If the inlet-valve has a slot called inlet you could expand the path to (the fosp-alpha inlet-valve inlet), which would return the inlet of the fosp-alpha's inlet-valve.

An important feature of the MSG language is co-reference, the ability to reference the same class element by different descriptors. Paths and synonyms play an important role in providing this facility. When a synonym is defined, a co-reference is built and used instead of the actual object. The
co-reference includes the actual object and the multiple paths to access the object. The data structure of a synonym is a list of synonym pairs. Each member of a pair defines a path to a slot which is synonymous with the slot found at the end of the path of the other member of the pair. For example, the synonym ((inlet-valve inlet) (suction)) states that the inlet of the inlet-valve is the same as the suction slot.

**KE-GRAPHER** is a tool to create, maintain, and inspect MSG objects using a graphic representation of the knowledge class hierarchy. It incorporates a graphing facility to display the class hierarchy with the nodes of the graph becoming mouse sensitive. Examples are shown in Figures 17 and 18. The window is divided into four parts: the title, the main graphing area which presents the class hierarchy, a margin choice area to select top level commands, and a keyboard input pane to change the objects to be graphed. The starting nodes of graphs are called roots. After a user types in an expression that will evaluate to the name of a class or a list of classes, the window will reset the roots and regraph the window. If the item evaluated is not the name of a class but the name of a flavor, then that flavor and those that depend on it will be graphed. One can pan around the graph, zoom its size larger or smaller by changing font sizes, hardcopy the graph, and save or load a class file from disk. Each node of the graph is mouse sensitive and via various button clicks one can describe, create, move, or edit a class. Similar facilities are provided for manipulating instances and flavors.

**Lesson Editor**

The **Lesson Editor** activity provides interface builders the ability to add instructional sequences to a simulation. In the current version of the Simulation Environment, the Lesson Editor Screen appears as in Figure 19. It maintains the full range of Model Control facilities while supplying additional
functions for creating and editing lesson sequences. These sequences are made up of sets of actions each of which are tied to some behavior within the running of a simulation. Each action can either display text or affect the state of the simulation in some way.

The Lesson Display Window shows a partial script of feedback lesson segments. This script could be used to show a feedback relationship in the Make Up and Excess Feed System. The first segment indicates an action to set the variable DFT (the level of a tank) to 825.0. The next segment is a condition which waits until the DFT is below 850. When that level is achieved, a set of icons is highlighted and text beginning with "When the DFT..." will be presented to the student. The highlighted icons consist of the components in a feedback loop which attempts to maintain the DFT between 895 and 1105. These script segments are added to the lesson by choosing a command in the Segments pane. The commands allow the user to present text to the student, highlight a set of icons of current importance, pause the presentation until a salient event occurs, set an icon to a value, and execute an arbitrary function. Much of this specification is done graphically. For example, setting the level of the DFT in this script was accomplished by pointing to the appropriate level in the iconic depiction (the DFT tank in Figure 4). Each segment can be edited, deleted, undeleted, or moved within a lesson. Lessons can be saved, read in from files, and performed to see the exact sequence that will be presented to the student.

We currently are in the process of expanding the Lesson Editor so that a user can employ knowledge which has been provided about simulation objects and views to construct instructional descriptions of components and behaviors within the simulation. We are also designing instructional systems which will gather information about a student's knowledge of a domain by monitoring interactions with the simulation. This system will pose questions to students, which are to be answered by interacting with graphical views.
Lesson Editor

<table>
<thead>
<tr>
<th>Root</th>
<th>Drop</th>
<th>Status</th>
<th>Save</th>
<th>Select</th>
<th>Misc</th>
<th>Select</th>
<th>Initiate</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mode</th>
<th>Config</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>Icon</td>
</tr>
<tr>
<td>Highlight</td>
<td>Function</td>
</tr>
<tr>
<td>Action</td>
<td>BFT</td>
</tr>
<tr>
<td>144</td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 19. Lesson Editor. Screen configuration for control of lesson editing activity. The result of interacting with a graphical view to specify a portion of a lesson sequence is also shown.

Behavior Editor

One obstacle to a domain expert's ability to use the Graphics Editor to construct an interface for a domain is the requirement of an existing mathematical model of the domain. Even if a model is available it is unlikely that a domain expert, without significant additional effort, will have sufficient understanding of the simulation model to be able to tap icons into it. The Behavior Editor is an initial exploration of a tool that would eliminate the need for an underlying simulation model. The icons in the Graphics Editor know how to "appear" in order to represent the status of variables to which they are tapped, but the behavior of the system is defined by a simulation model. The Behavior Editor is composed of icons that know aspects of both the behavior and the appearance of the objects they represent. The behavior of the system emerges from the interactions of the icons with each other. The ability to incorporate the behaviors required in a simulation is facilitated by the object-oriented implementation of icons.

To make intelligent icons capable of generating a simulation, it is necessary to add components with domain and simulation knowledge. We have built a number of icons that "know" the rudiments of fluid dynamics and understand about connections to other icons. These include, for example, tanks and pipes that know about pressure and flow, so they can be used in fluid systems. Figure 20 depicts a very simple system constructed in exactly the same manner as with the Graphics Editor, but involving behaviorally "smart" icons that can be connected together. These icons have mechanisms for recognizing when connections should be made and thus the topology of the view is automatically generated.

An excellent example of the flexibility engendered by icons with connectivity knowledge is the Super Sensor. This icon is a dial that asks icons it connects to what variables can be monitored and which variable to measure by default. In Figure 20 notice that a Super Sensor is connected to the left
Designing a tank to monitor its level. Sensors know how to monitor appropriate aspects of the objects to which they are connected. Users can manipulate these aspects by clicking on the *Miscellaneous* menu item on the Behavior Editor screen (Figure 21). The sensor has defaulted to measure *value*, which is highlighted. The sensor can be changed to measure fluid exchanged by clicking on that menu item.

When users interact with the tanks, setting their levels, we are actually setting the initial conditions of the simulation. On the Behavior Editor screen (Figure 21), they can then click on run, and the simulation will run until equilibrium. It is important to notice a fundamental difference between the Behavior Editor and the Graphics Editor. In the Graphics Editor, the designer would need a mathematical model of the tanks and pipes and then it would be necessary to find the appropriate variables to tap the icons into. Here, the icons themselves perform the required computations for the simulation.

**Designer**

Designer evolved out of our development experiences with Steamer. We found that instructors using the Graphics Editor sometimes created views that violated simple graphic design rules. They also had difficulty maintaining stylistic conventions across sets of views. Designer is a general tool for assisting with the design process. It functions as an interactive visual design consultant for users of the Graphics Editor and is intended to aid developers by providing graphic expertise during the construction and
FIGURE 21. Behavior Editor. Screen configuration of the Behavior Editor during the process of connecting a super sensor to a tank.

critiquing of graphical views. This expertise includes graphic design principles as well as standards of presentation. The underlying motivation is to improve the quality of the views by making them more consistent and visually more effective. In addition to merely describing design alternatives, the system allows exploration of the design space by explaining the advantages and disadvantages of alternative design solutions. Through interactive dialogue and constructive examples, the system tutors users of the Graphics Editor in principles of graphic design.

Designer consists of three interrelated processes, an Analyzer, a Critiquer, and a Synthesizer, coupled to a domain dependent knowledge base. This knowledge base consists of design elements and relationships, techniques for their identification, and sets of constraints used in distinguishing good design from bad. The Analyzer first parses the design based on the elements and relationships of the given domain. The Critiquer uses this information to indicate where the current design fails to conform with the principles of good design or predetermined guidelines. Finally, based on searches of the design space, the Synthesizer suggests alternative modifications to the current state of the design. The separation of these three processes from the knowledge base provides independence and modularity to the system.

Domain-based design constraints are the basis of the critiquing process. Constraints within Designer consist of both basic graphic design principles important in the construction of two-dimensional views and sets of view standards that are adopted for particular domains. The combination of principles and standards create a context or Style in which the design critique and subsequent modifications take place. By modifying the style within which a critique is made, one can ultimately affect the form of the final design. It thus becomes possible to request multiple critiques, each based on a different style. This is an especially powerful paradigm for designs that may need to be presented in different media, each with different constraints that need to be considered. For example, a style appropriate for a high-resolution color display may be inappropriate for a black and white hardcopy presentation.
An initial implementation of Designer has been completed. A functioning Analyzer and Critiquer used on existing Steamer views have provided useful feedback. It is very encouraging that even in views that we had thought were carefully crafted, the system has been able to note inconsistencies and suggest improvements. Progress has been made in identifying the basic elements, relationships, and principles of two-dimensional graphic design and incorporating them into Designer's processes. The Analyzer first evaluates design elements in terms of their size, shape, color, and location and then identifies relationships between them from information provided in the knowledge base. These relationships include similarity, proximity, grouping, and repetition. As new relationships are identified, they can easily augment the analysis process. Various techniques exist to interactively inspect the elements and relationships identified within the design.

The Critiquer locates examples and violations of the design constraints provided in the current style and creates a critique. These comments include descriptions and justifications based on the graphic constraints from which they were derived. Since the Critiquer works within the context of a current style, there are facilities to help define graphic constraints and maintain styles. A preliminary graphic constraint language allows the creation of new constraints, while a style editor has been developed to create, maintain, and switch between styles.

Figure 22 shows Designer's top-level user interface. The multipaned interface provides access to existing Graphics Editor functions and new Designer functions through scrolling command panes (upper right collection of panes). Access to the domain knowledge is provided in a mouse sensitive graphing pane (upper left pane). Comments of constraint violations are displayed in a scrolling pane (lower left pane), while descriptions of the violations can be displayed in the lisp interaction pane (lower left pane). In Designer, the status line, consistent throughout the simulation environment, includes the current style in which the analyses take place.

![Designer](image.png)

**FIGURE 22. Designer. Screen configuration of Designer.**
CONCLUSIONS

Interface design is currently very much more of an art than a science. There is a tremendous need for better theories of interface design and for more powerful tools to assist in their design and implementation. Currently there is virtually no theory of interface design. We do not even understand what contributes to the effectiveness of the most successful interfaces. We are in a state similar to when bridges were built by copying existing bridges without knowing in advance what would result from even the most minor variation. We need a more principled base for the design of interfaces, one that characterizes the dimensions of the space of interfaces. Such a theoretical characterization is the only way to be able to understand and intelligently make the myriad of tradeoff decisions inherent in interface design. Hopefully it is clear from this chapter that we think a theory needs to be erected from an understanding of interfaces as representational languages and based on an appreciation of the cognitive tasks that people are employing such representational systems to solve.

One of the factors that influences the development of a theory of interface design is that computer-based interfaces enable new forms of representational languages that are different in fundamental ways from more traditional representational systems. While most representational languages are static, these interfaces make possible dynamic languages. For example, when we use natural languages or mathematical notation to represent our knowledge about the world, the representational system itself is fundamentally static. The "action" comes from our interpretation of it. Compare this with a interactive graphical interface to a simulation. The representational system itself now can "behave," both in terms of reflecting state and in allowing us to directly manipulate it. We still are involved in a process of interpretation but it now concerns behaving entities. The interface becomes a kind of dynamic world in which we can think of and interact with objects as if they were the things themselves. Elsewhere we discuss this very different metaphor for interface design (Hutchins, Hollan, & Norman, 1986). The point here is that this is a novel form of representational system and one which we know very little about.

One of the appeals of these new forms of representational systems are the parallels that exist between them and the forms of representation we employ in perception. When one interacts with the world, the world changes and those differences are reflected in our perception. Interactive interfaces provide a similar form of behavior: we pick up, move, or otherwise modify some object, and the associated object in the simulation world changes. As we discussed earlier, this form of representation allows us to employ a number of very effective strategies and to do what we are particularly good at doing: detecting patterns, constructing mental models or simulations of the world that support causal reasoning, and manipulating the world by actions on it or on representations of it. The ability to bring an increasing portion of a dynamic world into the realm of the perceptually knowable is surely one of the major appeals of these new forms of interfaces. It is clear from our experiences developing Steamer that there is much power from interfaces that provide a form of conceptual fidelity. By this we mean interfaces that have characteristics similar to those normally attributed to people's mental models. These include the depiction of state, topology, hierarchical embedding, and the ability to run the models to make predictions about the consequences of changes.

Another conclusion we have reached about graphical interfaces derives from their ability to serve as filters of information. In most of the applications we have built there is an underlying simulation or real-time system. The collection of graphical views that comprise an interface to an underlying system can be conceived of as being a set of filters of information. The designer of a view filters information by selecting what and how to display it. Of particular interest to us has been the ways filtering can be employed to support the development of particular mental models. For example, a message-passing abstraction of a system can be given a graphical instantiation and thus serve to highlight characteristics not normally available and provide an effective way of thinking about the system. Often in Steamer we have found it advantageous to filter quantitative information and present it qualitatively. One of the very real problems in understanding dynamic systems like propulsion plants is their complexity. A major step in the understanding of process is the isolation of meaningful units to think about and attend to. Much of instruction and the development of expertise is dependent on isolating such units and
developing a language that permits talking and reasoning about them. If one looks at the language an
expert uses in explaining or predicting a system's behavior, one often sees a restatement of quantitative
events in qualitative terms. The filtering of quantitative information into qualitative terms may be an
extremely effective means of supporting these types of qualitative explanations and of providing information in forms that encourage development of the mental models needed in reasoning about physical
systems.

In order to build effective interfaces, better tools are required. A major portion of this chapter has
been devoted to descriptions of a set of tools we have built to aid in the implementation of interactive
graphical interfaces. The goals of our efforts have been to simplify the implementation of interfaces
and to make it possible for interface designers to operate at a higher level of abstraction than that normally provided. The object-based graphics editor allows a designer to operate at the level of graphical
objects which have been specifically designed for particular domains. It also provides the ability to
easily associate these iconic depictions with an underlying simulation or real-time system. The model
controller complements the graphics editor by providing an integrated set of facilities for controlling the
running of simulation models and interacting with views constructed with the editor. The icon editor
increases the generality of this simulation environment by allowing users to construct icons with
behaviors that are particularly tailored to the demands of new domains. It thus provides a mechanism
for extending the vocabulary of a graphical representational language. We have coupled these tools
with a series of knowledge-base editors to allow the incorporation of a wide variety of knowledge. The
behavior editor permits the specification of simulation knowledge within graphical icons themselves.
The lesson editor makes it possible to build instructional interactions and to make views capable of
explaining themselves and their constituents. Designer brings graphical design knowledge to users of
the graphics editor. To further augment and support the development of this growing set of tools, a
general frame-based representational language and graphical interface to it are also available to the
interface designer.

All of these software tools have been implemented using the object-based programming techniques of
Flavors. There are a number of reasons that an object-oriented paradigm is particularly advantageous
for supporting the development of graphical interfaces to simulations and real-time systems. One primary
reason is the natural mapping possible between the objects that a simulation models and their
graphical representation in an interface. Conceptually this makes possible a natural way of dividing up
the simulation world as seen via a graphical interface. Also of principle importance are the relations-
ships that can be made between object-based representations and mental models. The fact that objects
store state information, are made of simpler parts, and communicate with and share information with
other objects are obvious parallels between the two. These relationships facilitate building interfaces
that have some of the characteristics normally attributed to peoples' mental models. In addition, they
provide a number of programming features which have proven to be of considerable value. These
include nice modularity, the ability to inherit instance variables and messages and thus to easily share
common structure and functionality, and ready extensibility by the addition of new messages.

ACKNOWLEDGMENTS

This work is being conducted as part of the research activities of the NPRDC-UCSD Intelligent Sys-
tems Group in the Institute for Cognitive Science at the University of California, San Diego. It has
been supported by the Navy Personnel Research and Development Center as part of an Office of Naval
Technology's Program on Future Technologies for Training. Support has also been provided by con-
tract N00014-85-C-0133, NR 667-541 with the Personnel and Training Research Programs of the Office
of Naval Research.

We would like to express our appreciation to David Owen for commenting on a draft of this chapter
and to Kathy Farrelly and Barbara Morris for editorial assistance. The opinions expressed in this paper
are those of the authors and do not necessarily reflect the views of any government agency.
REFERENCES


ICS Technical Report List

The following is a list of publications by people in the Institute for Cognitive Science. For reprints, write or call:

Institute for Cognitive Science, C-015
University of California, San Diego
La Jolla, CA 92093
(619) 452-6771


Earlier Reports by People in the Cognitive Science Lab

The following is a list of publications by people in the Cognitive Science Lab and the Institute for Cognitive Science. For reprints, write or call:

Institute for Cognitive Science, C-015
University of California, San Diego
La Jolla, CA 92093
(619) 452-6771


Distribution List [UCSD/Worman, Hollow, Hutchins] NR 647-541

LCDR Cory deGroot Whitehead
Chief of Naval Operations
OP-112GI
Washington, DC 20370-2000

Dr. Christopher Wickens
Department of Psychology
University of Illinois
Champaign, IL 61820

Mr. Carl York
System Development Foundation
181 Lytton Avenue
Suite 210
Palo Alto, CA 94301

Dr. Michael Williams
IntelliCorp
1975 El Camino Real West
Mountain View, CA 94040-2216

Dr. Joseph L. Young
Memory & Cognitive Processes
National Science Foundation
Washington, DC 20550

A. E. Winterbauer
Research Associate
Electronics Division
Denver Research Institute
University Park
Denver, CO 80208-0454

Dr. Steven Sorman
Office of Naval Research
Code 1149
800 N. Quincy St.
Arlington, VA 22217-5000

Dr. Robert A. Wiser
U.S. Army Institute for the Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Martin F. Wiskoff
Navy Personnel R & D Center
San Diego, CA 92152-6800

Dr. Frank Withrow
U. S. Office of Education
400 Maryland Ave. SW
Washington, DC 20202

Mr. John H. Wolfe
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. Wallace Wolfeck, III
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. Joe Yasutake
AFRL/LIR
Lowry AFB, CO 80235
END
10-86
DTIC