

50008  
P-235

# **Final Technical Report**

## **Instructional Authoring by Direct Manipulation of Simulations: Exploratory Applications of RAPIDS**

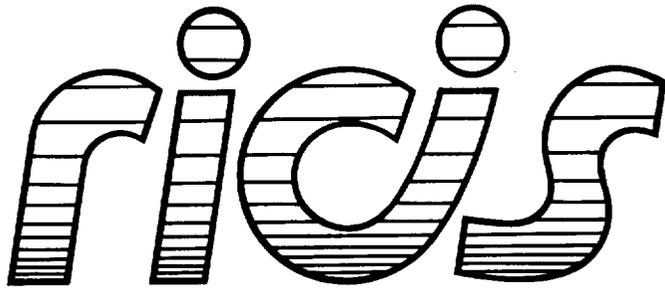
### **RAPIDS II Authoring Manual**

**Behavioral Technology Laboratories**

**August, 1990**

**Cooperative Agreement NCC 9-16  
Research Activity No. ET.13**

**NASA Johnson Space Center  
Mission Operations Directorate  
Space Station Training Office**



*Research Institute for Computing and Information Systems  
University of Houston - Clear Lake*

(NASA-CR-180001) INSTRUCTIONAL AUTHORING BY  
DIRECT MANIPULATION OF SIMULATIONS:  
EXPLORATORY APPLICATIONS OF RAPIDS. RAPIDS 2  
AUTHORING MANUAL Final Technical Report  
(University of Southern California) 235 p  
G3/01 0020008  
Unclas

# *The RICIS Concept*

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

***Final Technical Report***

***Instructional Authoring by Direct  
Manipulation of Simulations:  
Exploratory Applications of RAPIDS***

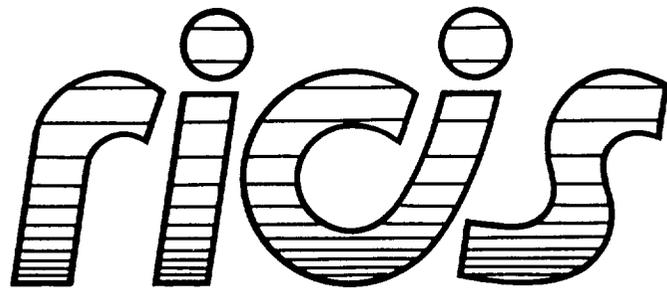
***RAPIDS II Authoring Manual***

**Behavioral Technology Laboratories**

**August, 1990**

**Cooperative Agreement NCC 9-16  
Research Activity No. ET.13**

**NASA Johnson Space Center  
Mission Operations Directorate  
Space Station Training Office**



*Research Institute for Computing and Information Systems  
University of Houston - Clear Lake*

---

**T · E · C · H · N · I · C · A · L      R · E · P · O · R · T**



## Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by the Behavioral Technology Laboratories, University of Southern California. Dr. Glenn B. Freedman served as RICIS research coordinator.

Funding has been provided by the Mission Operations Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Barbara N. Pearson, of the Systems/Elements Office, Space Station Training Office, Mission Operations Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representative of the official policies, either express or implied, of NASA or the United States Government.



*Final Technical Report*

*Instructional Authoring by Direct  
Manipulation of Simulations:  
Exploratory Applications of RAPIDS*

*RAPIDS II  
Authoring Manual*

*August 1990*

Documentation:  
Allen Munro

Design of *RAPIDS II*:  
Lee D. Coller, Allen Munro, Quentin A. Pizzini, David S. Surmon,  
Douglas M. Towne, and James L. Wogulis

Implementation of *RAPIDS II*:  
Lee D. Coller, Quentin A. Pizzini, David S. Surmon, and James L. Wogulis

Behavioral Technology Laboratories  
University of Southern California  
250 North Harbor Drive, Suite 309  
Redondo Beach, CA 90277

(213) 379-0844

Information in this document is subject to change without notice and does not represent a commitment on the part of the University of Southern California.

© Behavioral Technology Laboratories, USC, 1988, 1989, 1990

## ACKNOWLEDGEMENTS

The development of *RAPIDS II* was supported by the Air Force Human Resources Laboratory under RICIS Research Activity No. ET.13 (NASA Cooperative Agreement NCC9-16). J. Wesley Regian served as AFHRL scientific officer for this project.

*RAPIDS II* is based in part on the Intelligent Maintenance Training System (IMTS) and on *RAPIDS*, which were developed at Behavioral Technology Labs, USC, under the sponsorship of the Office of Naval Research, the Navy Personnel Research and Development Center, and the Air Force Human Resources Laboratories, under ONR contracts N00014-85-C-0040, N00014-86-K-0793, and N00014-87-C00489.

# TABLE OF CONTENTS

- Preface: *RAPIDS II* and Original RAPIDS *vii*
- 1 Rapid Development of Simulation-Based Instruction 1
  - Why *RAPIDS II* 2
  - Overview of Course Authoring 3
  - Simulation Composition 4
  - Authoring Instructional Content 8
  - Developing an Instructional Organization 11
  - Installing *RAPIDS II* 14
  - Using this Manual 16
- 2 The *RAPIDS II* Student Interface 17
  - Examples of Content Presentation 19
  - The Options Menu 27
  - Modes of Instruction 28
- 3 Building Generic Objects 29
  - The Role of Generic Objects 29
  - Using the Generic Editor 31
  - Object Operations 36
  - State Operations 48
  - Drawing Operations 57
  - Window Operations 61
- 4 Rule Authoring 64
  - Rules in Rapids II 64
  - Internal Rules 66
  - External Rules 74
  - Rule Editor Features 78
  - Rule Syntax 80
- 5 Developing Simulations 83
  - The Role of the Simulation Scene 83
  - Building a Simple Simulation 88
  - How Simulation Works 96
  - Viewing Simulation Data 99
  - Editor Operations 107
  - Object Operations 108
  - Simulation Operations 111
  - Run-Time Corrections 117
  - Simulation Debugging 124
  - Multi-Scene Simulations 134
  - Display-Window Operations 137

<b>6</b>	<b>Using Attribute handles</b>	<b>140</b>
	<b>An Example Simulation: A Simple Electrical Relay</b>	<b>141</b>
	<b>Connecting with the Mouse</b>	<b>143</b>
	<b>Using Make Connection</b>	<b>154</b>
	<b>Creating Test Equipment</b>	<b>159</b>
<b>7</b>	<b>Authoring Instructional Content</b>	<b>163</b>
	<b>Content Units</b>	<b>165</b>
	<b>Editing Content Unit Data</b>	<b>168</b>
	<b>The Content Items Menu</b>	<b>175</b>
	<b>Content Item Data</b>	<b>177</b>
	<b>Student Actions</b>	<b>178</b>
	<b>Expositions</b>	<b>187</b>
	<b>The Global Editor Commands</b>	<b>191</b>
<b>8</b>	<b>Instructional Organization</b>	<b>192</b>
	<b>A Sample Instructional Organization</b>	<b>193</b>
	<b>Creating a New Instructional Organization</b>	<b>194</b>
	<b>Student Evaluation in RAPIDS II</b>	<b>202</b>
	<b>Authoring Conditional Course Sequences</b>	<b>205</b>
	<b>Local Editing in Large Trees</b>	<b>209</b>
<b>9</b>	<b>Instructor Utilities</b>	<b>211</b>
	<b>Testing a Course</b>	<b>211</b>
	<b>Building a Turnkey Training Environment</b>	<b>213</b>
	<b>Examining Student Data</b>	<b>214</b>
	<b>References</b>	<b>217</b>
	<b>Index</b>	<b>219</b>

## Preface

### ***RAPIDS II* and Original *RAPIDS***

*RAPIDS II* is based, in part, on *RAPIDS*, a simulation-based intelligent-CBI authoring system. In original *RAPIDS*, simulations were created using IMTS, a simulation-composition and -delivery system. IMTS showed the productivity of a direct-manipulation approach to creating interactive graphical simulations. Experience with IMTS suggested ways that simulation editing could be made more powerful and yet be easier to use. *RAPIDS II*, the successor to both *RAPIDS* and IMTS, provides a fundamentally improved approach to simulation modeling. It also permits the development of simulations with ongoing processes, animation, and scheduled events.

**Locality of Effect** In IMTS, two different approaches to modeling devices were provided. In one approach, called *deep simulation*, the behavior of the simulation depended on the defined behaviors of generic objects. These object behaviors were defined in terms of the values of immediate neighbor objects. In the second simulation approach, called *surface simulation*, behaviors were defined in terms of values at other objects, which could be arbitrarily distant.

The deep simulation approach resulted in simulations that could more easily be modified, and required less painstaking clerical work. The surface simulation approach was more appropriate when the author did not understand the behavior of a device in terms of its components, or when a device was so complex that it was impractical to define its behavior in terms of the behavior of its components.

The problem with the deep/surface distinction in IMTS was that it was absolute. An author could not create a simulation in which some portions were based on strictly local passing of values, while other parts of the simulation made use of more remote ('surface') references to determine behavior.

There are a number of situations in which the strict deep/surface distinction was inappropriate. For many training simulations, it would have been convenient to mix the two approaches. It also would be very useful for authors to develop simulations incrementally, using 'surface' methods (non-local references and behavior rules) to prototype their device simulations quickly. Parts of the simulation could then be made 'deeper' (by using more generic rules and only local references) in a step-by-step way. This approach would let authors get simple versions of the whole simulation working quickly, so that feedback could be elicited from instructional developers, instructors, and perhaps students at an early stage of simulation development.

In addition to the impossibility of combining deep and surface approaches in a single simulation, the two styles of authoring simulation behavior were so different that few authors learned to employ both approaches. This meant that the simulation approach chosen for a particular IMTS simulation would depend more on the previous experience of the author than on the requirements of the training domain.

In *RAPIDS II*, a new unified approach to simulation replaces both the deep and surface simulation methods of IMTS.

**RAPIDS II Rules**     *Rules* describe and control the behavior of objects in *RAPIDS II*. Rule syntax and semantics is described in detail in chapter 4. At this point, it is enough to know that many features have been added to make rule authoring easier and less error-prone. In addition, rules are more widely available to simulation authors than was the case in IMTS. Rules can be either generic (universal for objects of a given type) or specific to a particular simulation. Authors can create and edit rules at the scene level as well as at the generic object level. The propagation of effects in a simulation is determined, in part, by rules created in the scene editor.

Rule editing in *RAPIDS II* is facilitated by a more powerful menu-driven editing system than was available in IMTS. In addition, the Envos Interlisp structure editor has been made available for rule editing, for the use of authors who are comfortable with that approach to editing structures.

**Attributes**     *Attributes* are data structures associated with objects. Rules can refer to attribute values. In IMTS, objects could have only one sort of attribute, called *ports*. Ports permitted values to be automatically passed from one object to another. Behavior rules in generic objects manipulated port values. In *RAPIDS II*, a more flexible approach is taken. Authors can use attributes to manage the same kinds of values that were carried by ports in IMTS. In addition, however, authors can make other uses of the *attribute* mechanism.

Certain attributes are created automatically in *RAPIDS II*. These include an object's location and its current state. Rules can refer to these standard system attributes, as well as to attributes created by authors.

**Continuous Appearances**     A number of different types of actions can be carried out by rules. A very common action is to assign a value to an attribute. In addition, however, rules can change the location or rotation of objects or of object states. This makes it possible to write rules that change the appearance of an object to reflect some computed value.

In IMTS, objects with moveable parts had to be represented by some fixed number of images of the states of that object. In *RAPIDS II*, the moveable part can simply be moved or rotated by a rule when the values that control the object change. This permits much more realistic simulations of continuous graphic changes, while requiring less graphical authoring.

**Processes**

In IMTS, a user event (such as throwing a simulated switch) would lead to a number of simulation events that would propagate through the simulation until no more rules needed to be activated. After the simulation had settled, the user would be able to carry out another action. In *RAPIDS II*, student users can manipulate the simulation while it is active.

This feature of *RAPIDS II* makes it possible to write rules that set up ongoing processes, such as incrementing or decrementing attribute values. The appearances of objects can be made to reflect these changing values, so that a simulation appears to be continuously animated. This means that a simulation user can carry out a series of interactions without waiting for all the effects of one action to settle before carrying out the next one. (The smoothness of the animation effects is dependent on the power of the delivery platform employed. On Xerox 1108 and 1186 computers, the animations will sometimes not appear very smooth.)

The combination of the new *process* and *continuous appearance* features make it possible to create real-time task simulations in *RAPIDS II*, greatly extending the training domains that can appropriately be attacked with this tool.

**Instructional Control**

The lesson editing features of RAPIDS have been extended in *RAPIDS II* to exploit the new real-time features of its simulation composition system. Lesson authors can require that certain actions be carried out before a given attribute attains a particular value, for example.

**Summary**

In summary, *RAPIDS II* offers a number of advantages over the original IMTS-RAPIDS combination. It permits the development of simulation training for real-time tasks. It provides techniques for including animations. Its authoring system permits a more flexible approach to choosing the locality of effect of rules in a simulation, while employing a more integrated approach to simulation authoring than did IMTS. The top-level menu of authoring options is much smaller and simpler in *RAPIDS II* than in IMTS/RAPIDS, even though *RAPIDS II* offers greater power and flexibility to the author.

<b>RAPIDS II Tools</b>	
<b>Simulation</b>	<b>Instruction</b>
<b>Generic Editor</b>	<b>Content Editor</b>
<b>Scene Editor</b>	<b>Plan Editor</b>
<b>Build Simulation</b>	<b>Run Instruction</b>
<b>Run Simulation</b>	

*The Top-Level Menu of Authoring Options*

*RAPIDS II* should prove to be an important milestone on the road to a fully integrated system for authoring and delivering intelligent simulation-based instruction.



# Rapid Development of Simulation-Based Instruction

---

RAPIDS II is a simulation-based ITS (intelligent tutoring system) environment. Many other computer aided instruction systems provide tools or programming language features that support the development of brief simulation segments intermixed with other presentations. RAPIDS II is a system for producing computer-based training courses that are built on the foundation of graphical simulations. RAPIDS II simulations can be animated. They can have continuously updating elements. They can be small and simple, or so large and complex that many screenfuls of graphics are employed in a single simulation.

The simulation-based characteristic of RAPIDS II makes it very appropriate for certain training tasks, but less appropriate for others. It is particularly well-suited for teaching people about the design, structure, maintenance, and operation of complex devices. It is an appropriate medium for operator training and for maintenance training. On the other hand, it is not designed for the presentation of inherently discursive materials. It would therefore be less appropriate for developing courses about art or language, for example.

Because the instructional authoring system relies on the presence of a computer-based simulation, its tools can exploit this simulation to permit very quick authoring of computer-based instruction. This manual will teach you how to use RAPIDS II to create simulations and to author training courses based on those simulations.

RAPIDS II is the successor to RAPIDS and IMTS. If you are already familiar with these tools, read the preface *RAPIDS II and Original RAPIDS*, above.

---

---

---

## Why RAPIDS II?

*RAPIDS* stands for *Rapid Prototyping ITS* (Intelligent Tutoring System) Development System. Because of its simulation-based style of authoring, it encourages the rapid development of interactive instructional courses that take advantage of computer graphic simulations.

RAPIDS II is the successor to the original RAPIDS authoring system, and it provides the same ease of authoring of instruction based on simulation. It also incorporates an improved approach to authoring simulations, and supports advanced simulation features, including ongoing processes. These features make it possible to develop simulation-based computer graphics training courses for real-time tasks.

### Advantages for the Student

The most significant advantage of RAPIDS II for the student is that high-quality computer-based courses can be developed that would have been too expensive to create using conventional methods. This means that students can benefit from the advantages of self-paced adaptive interactive instruction in a much wider range of courses than would otherwise be possible.

A second advantage for students is that RAPIDS II-built courses have a number of features that improve the chances that the presented material is correct. The simulations that underlie instruction must be self-consistent in order to function; therefore, an instructional sequence built on these simulated effects is guaranteed to present effects that actually work in the simulated world. This approach can be contrasted with other instructional systems in which it is possible to 'simulate' simply by displaying a canned sequence of pictures.

A third advantage for students is that the RAPIDS II delivery system has been developed in such a way that students cannot easily be stranded by authors. If a student cannot come up with the answer, the run-time software will get the student through that content item and on to the next one.

### Advantages for the Author

The advantages of authoring with RAPIDS II include

- Quick Acquisition of the Authoring Tools
- Speed of Development
- Quality of the Resulting Course

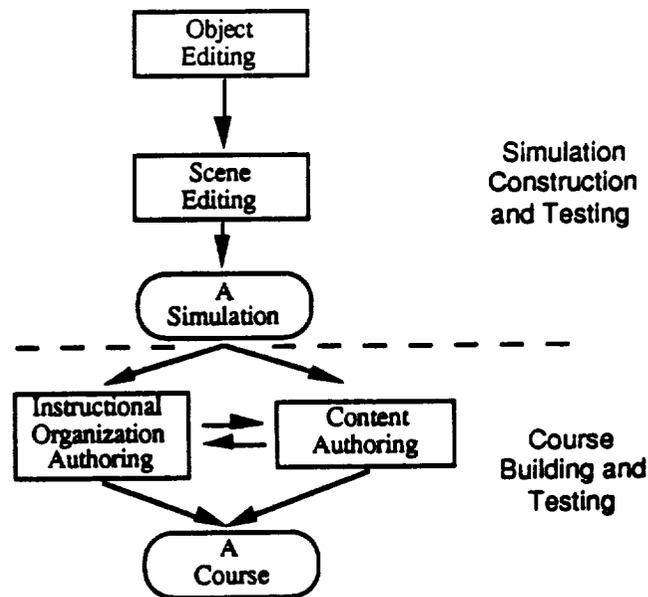
**Quick Acquisition of the Authoring Tools.** RAPIDS II authoring can be learned quite quickly because it does not require programming language skills. Development of course materials is carried out through the use of menus and buttons in the editor windows, and by directly manipulating graphic simulations in essentially the same ways that students do. This straightforward authoring interface makes it possible to begin building useable courses the first day that you are introduced to RAPIDS II (providing that the simulation on which your course is to be based is already available to you).

**Speed of Development.** Courses can be developed very quickly (once simulations have been built) because authoring is done largely by modeling the desired student behavior. Authors create a content item by performing the next action that is to be required of the student (and optionally by typing in prompts and explanations). This direct manipulation approach to authoring courses offers the potential for very high rates of productivity for course developers.

**Quality of the Resulting Course.** Basing your course on an existing simulation gives you opportunities for exploiting the work put into the simulation in many ways. You can easily add instructional units that would be too laborious to author by conventional means. In addition, many authoring errors that can be made with other computer-based instruction systems are likely to be avoided

## Overview of Course Authoring

There are two stages to RAPIDS II course authoring. They are simulation construction and course building. Simulation authoring has two major components, object editing and scene editing. Simulation authoring must be complete before course building is undertaken. Course building also has two components, content authoring and instructional organization authoring. The instructional plan and content may be authored in any order. It is common for these two authoring phases to proceed in tandem.



A good part of the technical literature on course development presupposes that all instructional planning precedes all course development. While RAPIDS II makes it possible to develop courses in this way, it does not require that the author do so. We believe that it is usually a good idea to begin courseware development with a good instructional plan, but we have found that the process of developing content often informs the planning process. An iterative approach may therefore prove to be the most effective one for many course development projects.

Simulation construction, which is described in detail in Chapters 3 - 5, is carried out using direct manipulation. A simulation author builds a simulation largely by drawing it. Content authoring is performed in a similarly direct fashion. To build a content item, you type in explanatory texts and student prompts, and you then carry out the action that you want students to perform, using the previously authored simulation. Instructional organization authoring means the construction of a hierarchical plan. The plan is shown as a tree structure, where the terminal nodes are content units. Once again, you will use direct manipulation techniques to build this part of your course, the instructional organization.

The next three sections of this chapter present brief overviews of these three aspects of RAPIDS II course authoring: simulation composition, content authoring, and instructional plan building.

## Simulation Composition

Simulation composition is presented in some detail in Chapters 3 - 5. In this section we present only a conceptual overview of the RAPIDS II simulation composition system.

### Modelling at the Element Level

The elements, or objects, used in RAPIDS II simulations can be produced by non-programmers, and they can be saved and used in any number of specific applications. This contrasts with some other approaches to simulation composition, such as that employed in STEAMER (Hollan, 1983; Hollan & Hutchins, 1984), in which the simulated device is modelled with a specially written computer program. (STEAMER's graphical indicators — such as gauges and indicator lights — are generic elements that can be used at different points in a simulation, or in different simulations.) The RAPIDS II approach has the advantage of permitting faster and easier simulation development, for the class of systems that can be simulated in this manner.

### Propagation of Effect

Some objects in RAPIDS II simulations are directly manipulable. Students click the mouse on points called *handles* to change the states of those objects. (State changes in an object are usually accompanied by changes in appearance.) Changing an object's state will typically activate one or more rules associated with the object. Activation of these rules will cause the values of certain attributes to be changed.

When a student changes the state of a simulated control object, the object's rules determine new values for some or all of its attributes. These values are referred to by the rules of neighboring objects, some of which may change state as a result of the activation of their rules. These neighboring objects may also have attributes that change as a result of the activation of their rules. These changes will, in turn, result in the activation of rules associated with other objects. In a complex simulation, hundreds of objects may be affected by a single manipulation, and thousands of attribute values may be recomputed.

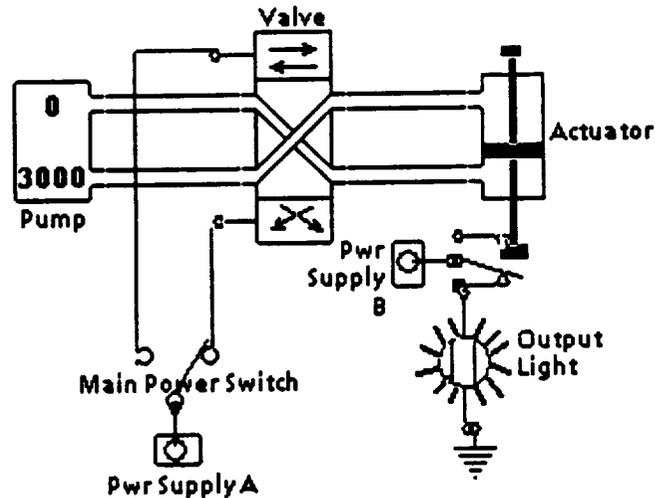
Complex system-level behaviors are derived from simpler component-level behaviors. This permits accurate free-play simulations without requiring authoring an immense number of combinatorial effects (as did an earlier simulation training system developed by this research group, described in Towne, 1986; Towne & Munro, 1981; and Towne, Munro, Johnson & Lahey, 1983).

To minimize simulation development time and effort, authors should be able to build simulations largely by drawing them. To the extent possible, authoring should be direct and concrete, rather than indirect and abstract (Norman & Draper, 1986).

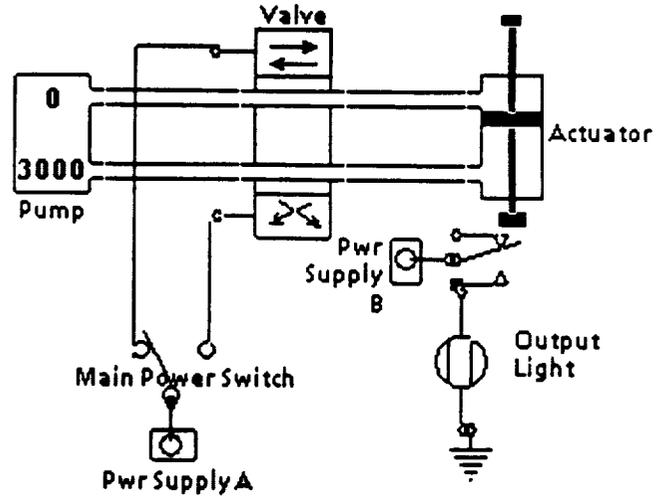
**A Simple Simulation**

A simulation is composed of instances of generic objects. Below is a simple simulation of a Rube Goldberg machine that uses electrical, hydraulic, and mechanical components to turn a light on and off. Power Supply A provides power to an electrically operated control valve, while Power Supply B provides power to the Output Light.

When the user moves the Main Power Switch to the right, the valve is put in its crossed position. This directs hydraulic pressure to the mechanical Actuator (at the right in the diagram), causing it to extend. The actuator pushes a contact closed, and electrical power turns on the Output Light.



If a user moves the switch to the left, the valve goes into its straight state and the actuator is retracted. The contact below opens, and the Output light goes out. All these responses are produced in accord with the behavior rules stored with each generic object.



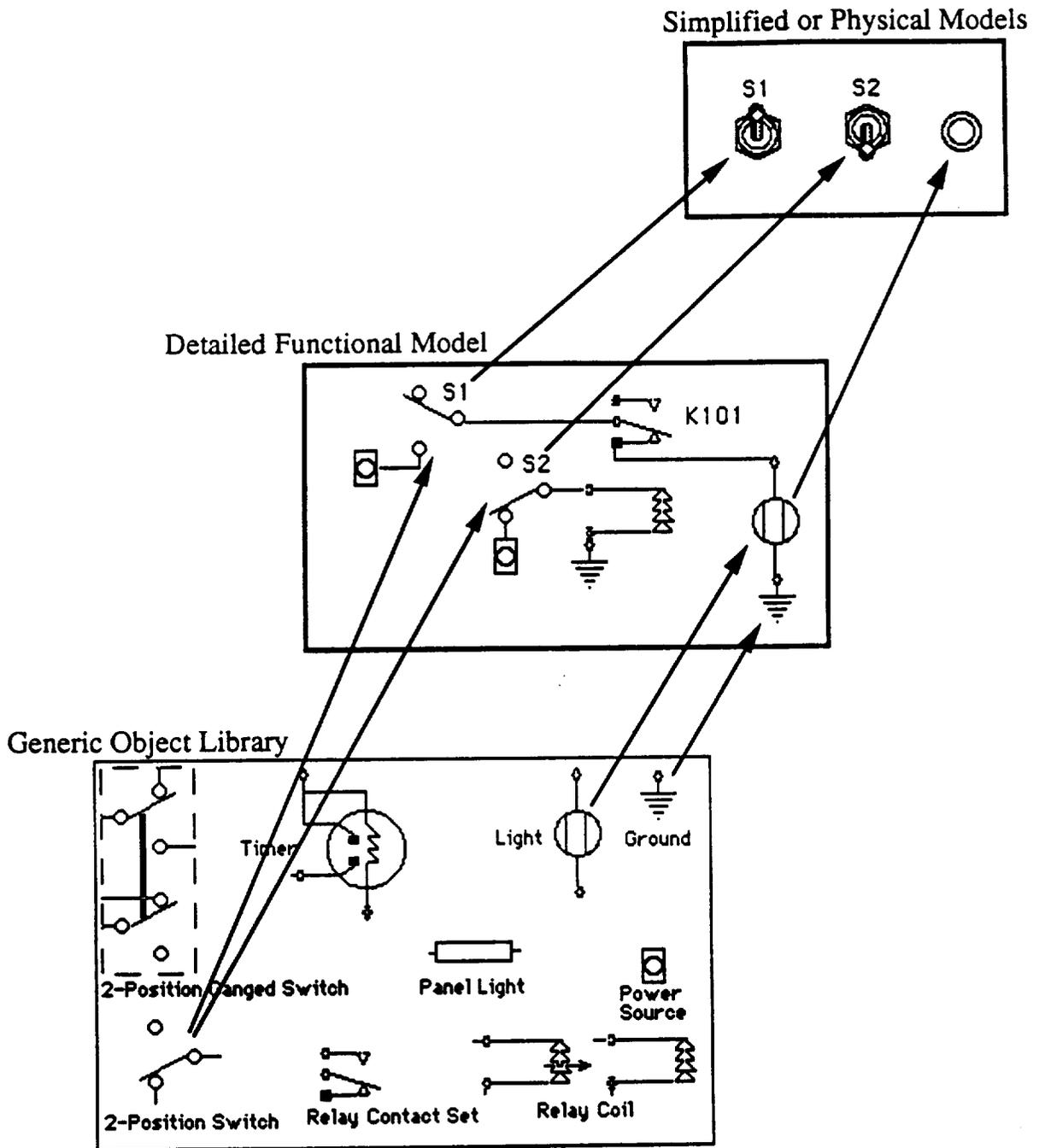
In addition to manipulating controls and observing simulated front panel indicators and internal actions of objects, students can examine values at object ports using simulated test equipment. When they work with large simulations, students sometimes discover things about the behavior of simulated worlds that even the authors were not aware of.

**Simulation  
Authoring**

Building a RAPIDS II simulation consists of composing diagrams from a library of generic objects. Authors can build very large simulations by dividing the target system into a number of subsections, called *scenes*, that can be displayed in their entirety in the main simulation window. Simulation effects propagate from one scene to the next through connections identified by the author.

When the scenes of the functional model are completed the author may interact with the simulated system to check out its system behaviors. These interactions may include setting switches, inserting one or more failures, observing indicator readings, and using test equipment at test points.

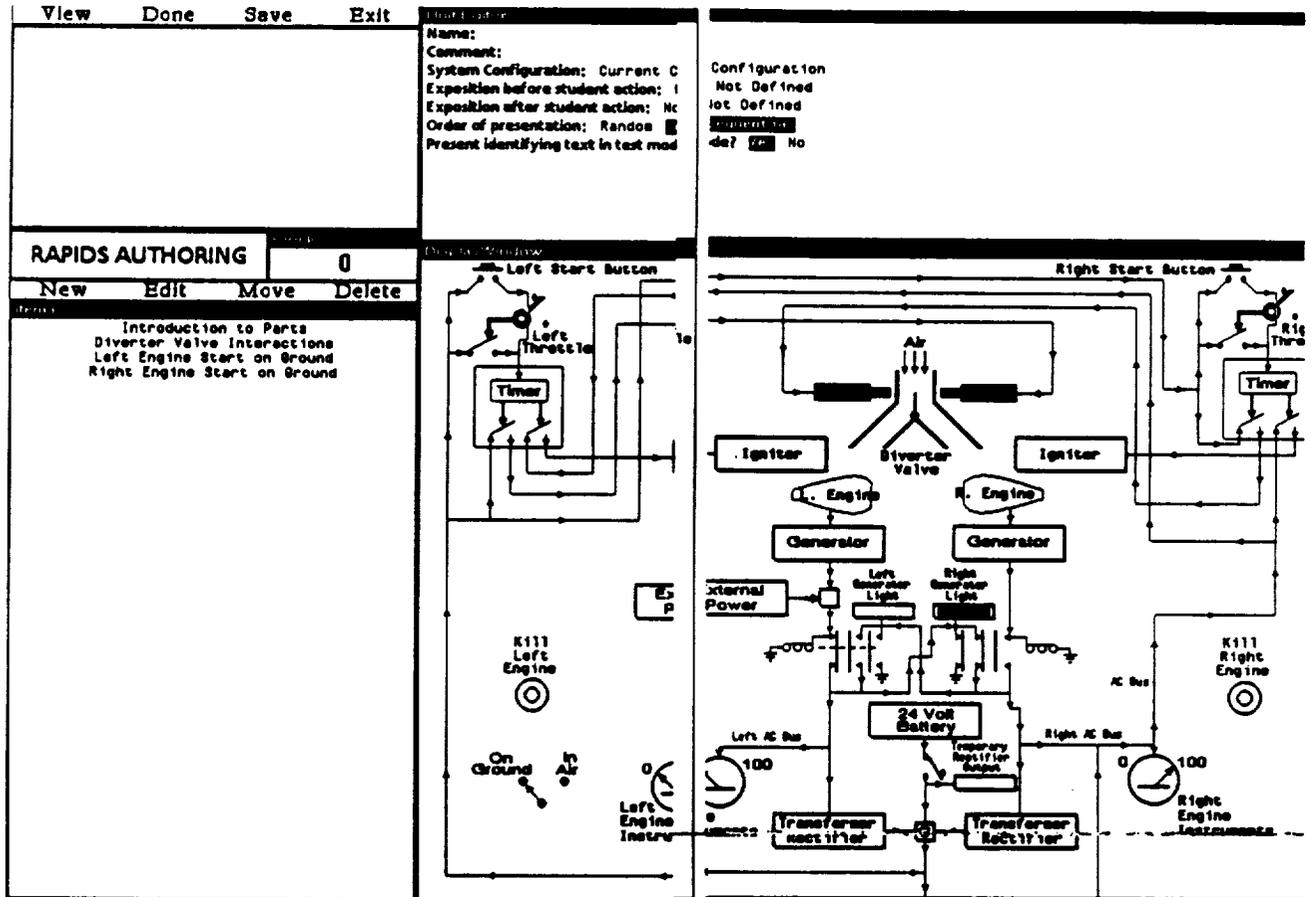
As sketched in the diagram below, the author may go on to produce additional scenes that represent the target system in simplified ways, or in more physical, less schematic forms.



Three Levels of Representation in Composed Simulations

## Authoring Instructional Content

The RAPIDS II instructional content editor gives you an environment that includes a live simulation. The figure below shows a simulation of an engine starting system. Instructional content based on this simulation can be developed using RAPIDS II, in part by manipulating the simulation in the same way that a student would.



The above figure shows the RAPIDS II *content unit editor* in use, being used to build a set of content units based on a simulation of an aircraft engine starting system.

### Content Units

A *content unit* consists of a group of related content items, and may optionally include preceding and/or following expository material. In the figure above, each of the lines in the window titled 'Content Units' names a content unit that has been defined. 'Introduction to Parts' is one such content unit, 'Diverter Valve Interactions' is another, and so on.

In addition to having one or more content items and pre- and post-expositions (explanatory presentations), a content unit has an associated *system configuration*. This is a stored state of the simulation. During training, the RAPIDS II run-time software restores the simulation to this state before the first content item of the unit is presented. This ensures that students will always perform the actions of a content unit in the same simulated environment in which they were authored.

### Content Items

A *content item* is a general-purpose element that handles the presentation of one small chunk of information to the learner, through a combination of text and RAPIDS II graphics, and that requires a response from the student. According to the way an item is authored it can serve to present technical theory, to acquaint a learner with the topology of a front panel, or to instruct in the performance of some action.

A content item, like the larger content unit to which it belongs, may have a preceding or trailing exposition. It always has a prompt, which serves to label the associated student action. Finally, it includes a required student action.

Content items present the bulk of instructional content. An item consists of:

- exposition (text, video, and/or graphics) to be presented *before* the action:
- exposition to be presented *after* the action:
- *identifying text* that identifies the subject of the action
- a specification of the correct *student action*

The *before-the-action exposition* is a combination of text and graphics that explains, describes, or illustrates a single item to be learned. The text is authored by the subject matter expert as s/he manipulates the simulation graphics. The student sees the identical text and graphics during the learning presentation. The exposition may highlight associated areas on the graphic simulation, and it may display video disc images.

The *after-action exposition* has identical capabilities as the before-action exposition. Typically, it might point out and explain important effects of the action (either by the student or the expert) and it might summarize what important points should have been learned by doing the item.

The text in expositions may be presented in a standard text window at the side of the simulation graphics or it may be positioned on the graphic simulation to relate closely to particular parts of the device representation.

The *identifying text* of an item describes the expected student response. In certain modes, it is used to prompt the student to respond. In a front-panel drill, one item might be to locate the Standby switch. The identifying text would be

Standby switch

The training system would use this text to compose the prompt  
locate the Standby switch.

In this example, the correct response would be to click in the region of the switch on the graphical scene.

**Student Actions** A *student action* is a specification of what the student is expected to do in order to successfully complete the content item. (So that, for example, the next content item can be delivered.) The specification of a student action may consist of

- clicking on one or more objects on a scene or scenes of the simulation
- manipulating one or more switches into specified states
- clicking in one or more regions on a scene or scenes
- make one or more selections from a menu of text items

The last of these options provides a mechanism for specifying multiple choice questions and answers. The simple click-on-the-menu-item user interface provides a straightforward implementation that does not require any special authoring.

Student actions are assumed to be the fundamental units of RAPIDS II authoring. They are the most important components of content items. RAPIDS II provides very direct methods for authoring the required actions of a simulation-based course.

**Expositions** The RAPIDS II content unit editor makes it possible to build *expositions*, which are used to produce explanations, admonitions, and other presentations for students. An exposition consists of a sequence of exposition elements. There are a number of distinct types of exposition elements that are supported in the content unit editor. These include

- presenting text in the message window
- clearing the message window
- playing a videodisc segment
- highlighting an object in the simulation window
- highlighting a region in the simulation window
- changing the scene displayed
- waiting for a student click
- waiting a specified amount of time
- presenting text in a *floating window*

*Floating windows* are a special exposition feature that makes it possible for authors to open, shape, and position windows that overly the simulation window. Authors can specify what text should appear in these windows, and can clear and close them as well.

The exposition editor generates a *script* of exposition events, which appears in the window to the left of the simulation window. In the figure shown at the right, an exposition script with six events is displayed. The first of these is a text event — it will have the effect of presenting the authored text in the message window. The second and third events shape and open a floating window.

Any exposition event in a script can be selected by the author, and the selected event is highlighted in the script. Selected events can be deleted, edited, or moved to a new position in the script. The next chapter presents examples of expositions as they appear to the students.

View			Done			Save			Exit		
RAPIDS AUTHORIZING										Clock	
										0	
Add				Done				Move			
Edit				Delete				Run			
Text: Every source of electrical power must be routed through the cross-start relay --->											
Exposition											
Text:											
There are four sources of electrical power. You will now be asked to point to each of them in turn.											
Floating-Window: reshape: (88 175 175 182)											
Floating-Window: open window											
Floating-Window: show text:											
Every source of electrical power must be routed through the cross-start relay --->											
Wait-for-student:											
Floating-Window: close window											

As you will see when you develop courses, the set of authoring choices in RAPIDS II is quite constrained. This constrained instructional syntax makes it possible for RAPIDS II to automatically generate a good deal of rich semantics for the simulation-based instruction. This approach has three advantages:

- 1 Authoring is largely direct and not symbolic
- 2 There can be no 'programming bugs' in authored instruction
- 3 Presentation quality is typically very high

## Developing an Instructional Organization

A course's *instructional organization* or *instructional plan* is used to determine the order that content units will be presented to students, and whether a particular student will be presented with certain content units at all. The instructional plan of a course is its highest level component. The lowest level elements in such a plan are the content units described above.

RAPIDS II provides a special tool for creating and editing instructional plans for courses, called the instructional plan editor. The window below shows a plan for a simple course about an engine starter system. Plans are organized

as tree structures, with the root of the tree at the left. The editor is used to add, delete, move, and edit the nodes that make up the tree.

Exit

Save

AddUnit

DeleteUnit

MoveToParent

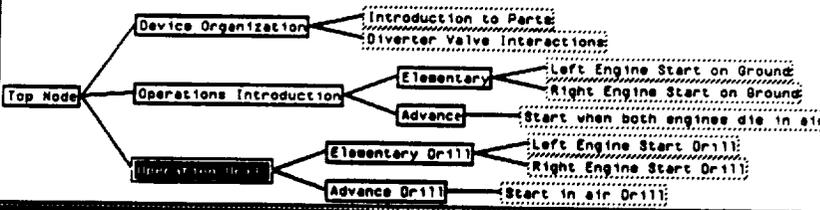
MoveToTop

MoveToNode

SetDepth

RepositionUnit

Save to file: ENGINESTARTER  
Saving...done.



---

Name: Operation Dr111  
 Comment:  
 Order of presentation: Random    
 Content:

Unit	Weight	Mode	Condition	Maximum	Minimum	Limit	Accuracy	Speed
Elementary Dr1	3	Dr111	Defined	2	1	0	N/A	N/A
Advance Dr111	7	Dr111	Not defined	4	1	10	N/A	N/A

In the tree shown above, the instructional plan calls for three major components. Students will first learn about the device organization, then be introduced to operations, and finally be drilled on operations. The nodes that represent these sections of the course are called *organizational units*, in contrast to the *content units* described in the previous section. Content units include the subject matter to be delivered. Organizational units serve to group related content units or other organizational units. In the tree displayed by the instructional plan editor, organizational unit nodes have solid borders and content unit nodes have dashed borders.

The window shown below the tree window displays data about the currently selected node. The data can be edited in this window. In this example, the organizational unit called 'Operation Drill' has been selected, and the data shown in the lower window pertain to it.

**Structure of Instructional Plan Units**

An *organizational unit* lists other units to be presented. The member units may be content units or other organizational units. Associated with each unit called in a plan are these data fields:

- weight: the importance of the called unit (relative to the others in the list)
- mode: whether to execute a called content unit in Instruct, Drill, or Test mode
- condition: an optional expression that controls whether to present the unit

- maximum: the maximum number of times to present the unit
- minimum: the minimum number of times to present the unit
- limit: the time limit for the unit, in minutes
- accuracy: the accuracy score (%) required to complete the content unit successfully
- speed: the speed score required to complete the content unit successfully, in minutes

Certain of these fields apply only to content units. The mode, accuracy, and speed fields have undefined values for organizational units. The data fields used to control the presentation of organizational units are condition, maximum, minimum, and limit. Accuracy scores for organizational units are computed and returned, however.

Here is a fuller description of the uses of instructional unit fields:

**Weight.** The least important unit in a parent unit should be assigned a weight of 1. The others should be assigned integer values of 1 or more to reflect their relative importance. The composite score of student proficiency on the parent unit is the weighted average of the proficiency scores (%'s) of the member units.

**Mode.** This is only meaningful when a unit is calling a content unit. In this case the mode determines whether the items in the content unit are presented in Instruct, Drill, or Test mode.

**Condition.** This is an optional Boolean expression that is used to determine whether it would be appropriate to present this portion of the RAPIDS II course. A condition is evaluated prior to each presentation of its associated unit. If the condition evaluates to true, then the unit is performed.

A simple example is

Accuracy of Drill3 < 65

which evaluates to true if the accuracy score on unit Drill3 was less than 65%.

A more complex example is

((Accuracy of Drill3 < 65) and (Speed of Drill3 > 5)) or (Performances of Drill3 < 3)

This condition specifies that a unit will be performed if the student's accuracy and speed were poor on unit Drill3 or if it was presented less than 3 times.

A condition can refer to the following measures for the current unit or for any other unit in the course:

- the number of presentations of the unit
- whether the unit was successfully completed
- the total time spent by the student in the unit, on all repetitions
- the latest speed score
- the latest accuracy score, if any

If the unit has not been presented, then speed score is infinite and accuracy score is 0.

Authors don't need to learn the names that are used to refer to these unit data (accuracy, speed, number of presentations, and so on), because conditions

are composed by making menu selections. The process is described in Chapter 4.

**Maximum.** This specifies the most times a unit will be repeated in a row, in an attempt to achieve the proficiency criteria. This would be set to a very high number (or left unspecified) if the planner wishes to repeat until time runs out or until the student meets the performance criterion.

**Minimum.** This field specifies the fewest times a unit will be repeated in a row. This is usually set to 1, however some planners might wish to repeat a unit some number of times, regardless of the student's performance.

**Limit.** This is the most time (expressed in minutes) that will be allocated to the called unit.

**Accuracy.** The accuracy score required to complete a content unit, expressed as a percentage.

**Speed.** The speed score required to complete a content unit, expressed in minutes.

## Installing *RAPIDS II*

This chapter has briefly exposed you to the major concepts that underlie *RAPIDS II*. In Chapter 2, you will learn what a *RAPIDS II* course looks like to a student as it is presented. Chapter 3 will show you how to create and edit generic objects using the generic editor. Chapter 4 describes rule editing. Chapter 5 covers scene authoring and simulation-building. Chapter 6 deals with authoring instructional content, while Chapter 7 treats course organization. Chapter 8 briefly presents the instructor utilities. In order to carry out the examples presented in the manual (and in order to develop your own courses), you must install the *RAPIDS II* system on your computer.

- Installation Steps**
- 1 Create a clean partition.
  - 2 Using the Filebrowser, copy all of the files on the release floppies onto a new hard disk subdirectory called *RAPIDSII*; i.e.,  
 {DSK}<LISPPFILES>RAPIDSII>
  - 3 (DV DIRECTORIES)  
 Edit your Directories variable so that it includes  
 {DSK}<LISPPFILES>RAPIDSII>  
 {DSK}<LISPPFILES>LIBRARY>  
 {DSK}<LISPPFILES>LISPPUSERS>  
 {FLOPPY}  
 {DSK}

4 **Note:** The RAPIDS II release includes a new set of simulation tools. RAPIDS II will not work with original IMTS.

5 (CNDIR '{DSK}<LISPPFILES>RAPIDSII>)

6 (FILESLOAD RAPIDS-IIMENU GEREAL SEREAL INST INST-CUE  
INST-PLAN SIM-STUDENT)  
(When building a student environment in which no course editing will take place, you can simply call (FILESLOAD INST SIM-STUDENT).

The above FILESLOAD command will take quite some time to be completed, because it loads all the functions that are needed by all of the simulation and instruction editors. At the end of the load, the RAPIDS II Tools Menu will appear on your screen. (See the figure on the next page.)

**Build a Simulation**

In order to begin working on a course, you will also have to build the simulation on which the course depends. Chapters 3 - 5 describe how to build a simulation in your environment.

To begin with, you might like to build the EngineStarter simulation that is used in the examples in this manual. You can build this simulation by typing this command in an exec window:

(BuildRapidsSimulation 'NEWSTARTER 'ENGINESTARTER)  
or by using the *Build Simulation* button on the RAPIDS II main menu.

RAPIDS II Tools	
Simulation	Instruction
Generic Editor	Content Editor
Scene Editor	Plan Editor
Build Simulation	Run Instruction
Run Simulation	

After a short delay, the simulation will be built and you will be able to carry out most of this manual's examples in your own environment.

---

---

## Using this Manual

We recommend that you at least skim this entire manual before attempting to build your first RAPIDS II course. Implement the examples on your own machine as you read, so that you will become familiar with RAPIDS II authoring features in a simple training environment. You should not start building your own course until you are comfortable with these examples.

Make certain that your simulation is working correctly before you attempt to build a RAPIDS II course using it. Your course may not work correctly if you make changes to your simulation after you have authored the course.

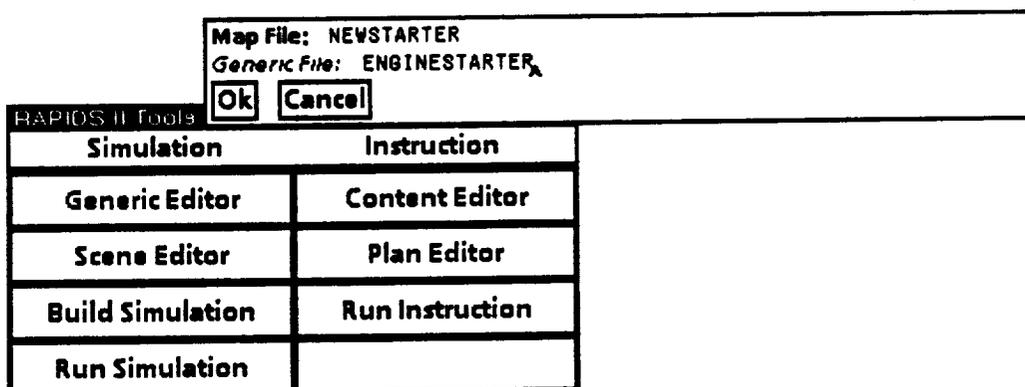
## The *RAPIDS II* Student Interface

To create a RAPIDS II course, you must first load the RAPIDS II authoring and instructional environment, as described in Chapter 1. Three steps are then required to build a course:

- Create a device simulation
- Build content units — instructional materials based on the simulation
- Make an instructional plan that will control presentation of the content

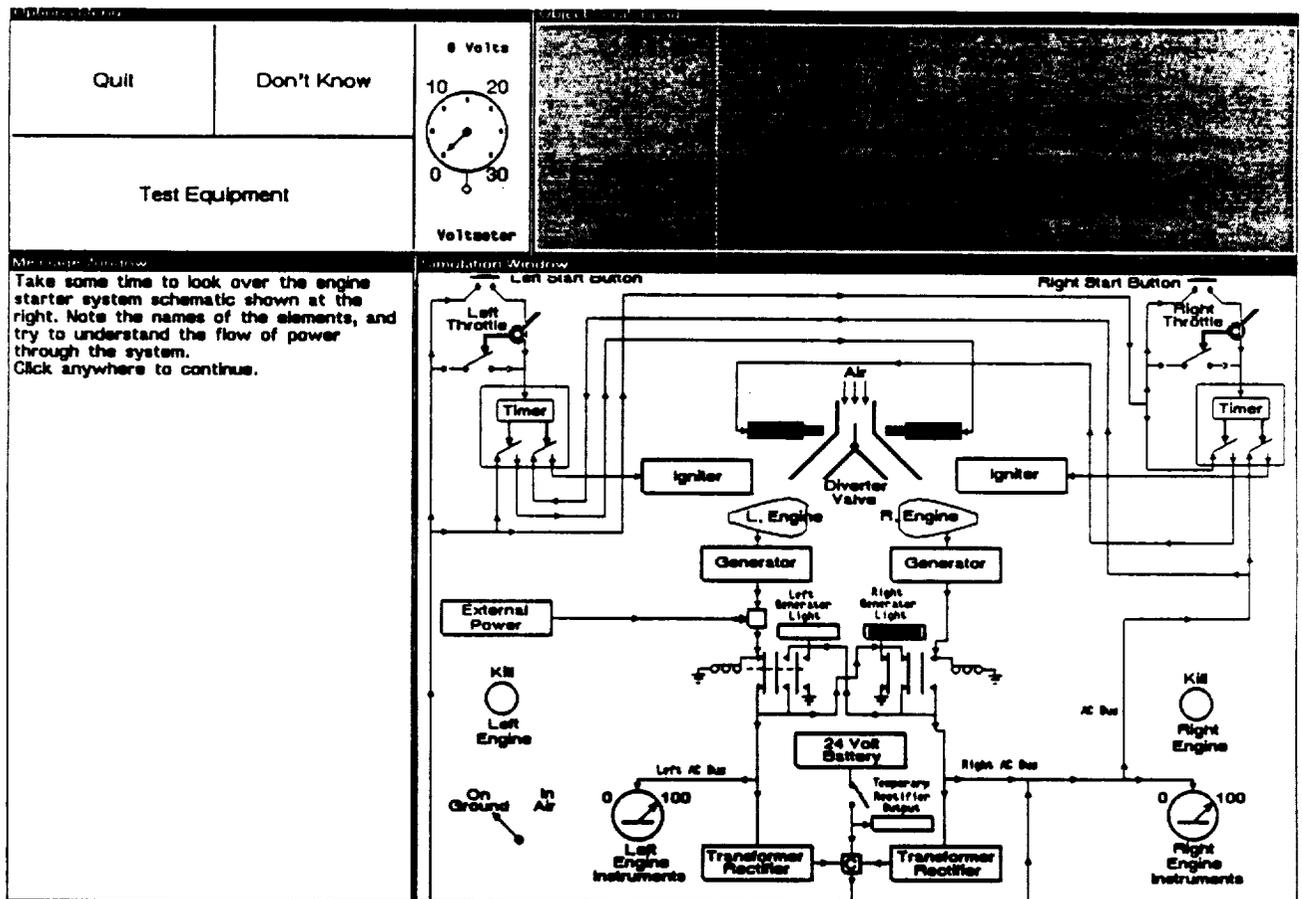
When all three steps have been completed, a *course* is available for students. Every course must have a simulation, instructional content, and an instructional organization or plan. The simulation must be created before the instructional content and the plan. Content and plans, however, may be developed in any order. Authors often alternate between plan and content authoring.

Your release of RAPIDS II includes a small course on a jet engine starting system. You can run this course, but first its simulation must be built in your environment. To build the simulation, you must first instal RAPIDS II as described in Chapter 1, and then use the *Build Simulation* command:



Some time will be required to build the simulation. At the end, a message will appear saying that the simulation has been built.

The figure below shows the RAPIDS II student interface. You can bring up this course display by running the simple course on your computer. Click on the *Run Instruction* option in the *RAPIDS II Tools* menu (see above). A numeric keypad will appear on your screen. Click on the "0" key and then on "ok" on the keypad. You will be asked what course you want to run. Type *ENGINESTARTER*. A display similar to the one shown below will appear. A menu to the right of the Voltmeter asks whether you want to "Start next unit." Click on that command, and you will see the display shown below.



As is appropriate in a simulation-based training system, the largest window in the student environment presents a graphical view of a simulation. The window to its immediate left is a message window, in which text and instruction created by the author and, in some cases, by the RAPIDS II instructional environment itself, are presented. Above the simulation window are a small window for simulated test equipment and a larger window that serves as an object scratchpad. The object scratchpad is an area into which

authors may place small windows that view parts of other scenes in the simulation.

The buttons in the top left corner show student controls that are usually available in the RAPIDS II student environment. These button controls are used to carry out meta-simulation activities. Students can also directly manipulate controls in the simulation, if the course they are using permits this. They can also point to objects and regions of interest on the screen, and they can make multiple-choice text responses using menus. None of the student actions requires the use of a keyboard.

In the next section you will see examples of a number of different styles of student-course interaction. Note that in all of them, student interaction always involves some type of pointing response, rather than typing.

In the last chapter, you learned in the abstract about the structure of courses, content units, content items, student actions, and expositions. That is, you learned something about what kinds of data are associated with each of these constructs in RAPIDS II. In this chapter, you will see how these constructs appear to student users. In the next two chapters, you will learn how to use the RAPIDS II editors to create and edit your own courses, content units, content items, student actions, and expositions.

---

---

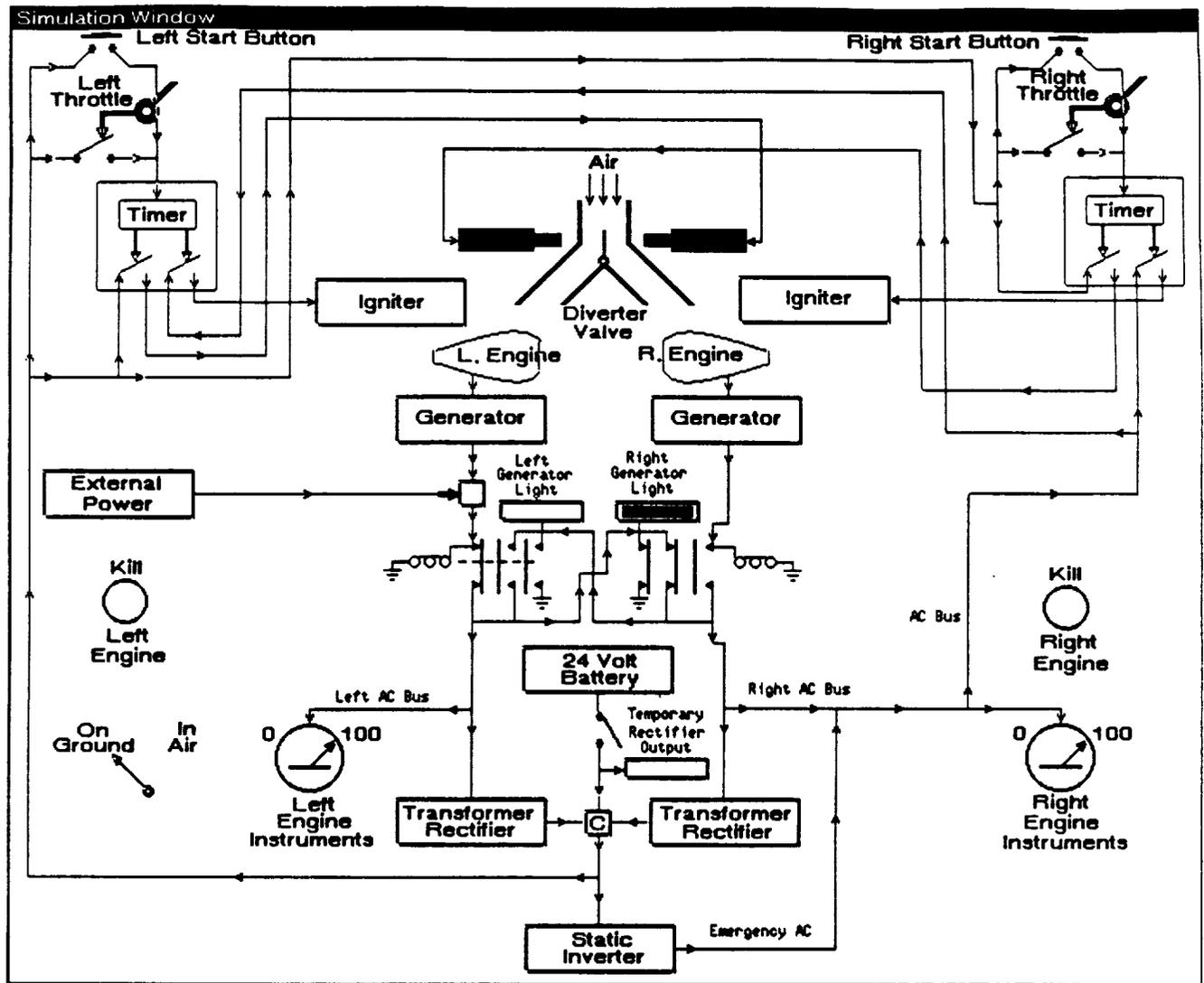
## Examples of Content Presentation

An easy way to learn about the student interface is to work through the simple course that is distributed with RAPIDS II. As you work through the sample EngineStarter course, you will see the actual appearances of a number of different types of content unit items. In this section of this chapter, a few of these presentation types are displayed and discussed.

### About the EngineStarter Course

The EngineStarter course is presented here only as a simple example of RAPIDS II course development, not as an exemplar of a complete course meant for actual use. The simulation is adapted from a training system developed by Kieras (1988). The details of the EngineStarter simulation are not important for learning about RAPIDS II, but a simple explanation of the functions of the system may help you to follow the examples. See the picture on the next page.

EngineStarter is a simulation of an aircraft engine starting system. When such a system is on the ground, it is hooked up to an external source of electrical and hydraulic power. Once engines have been started, this power is disconnected. In a typical on-the-ground startup sequence, one engine is started (using an engine start button) using the external power. The live engine drives a generator, which provides power for starting the second engine.

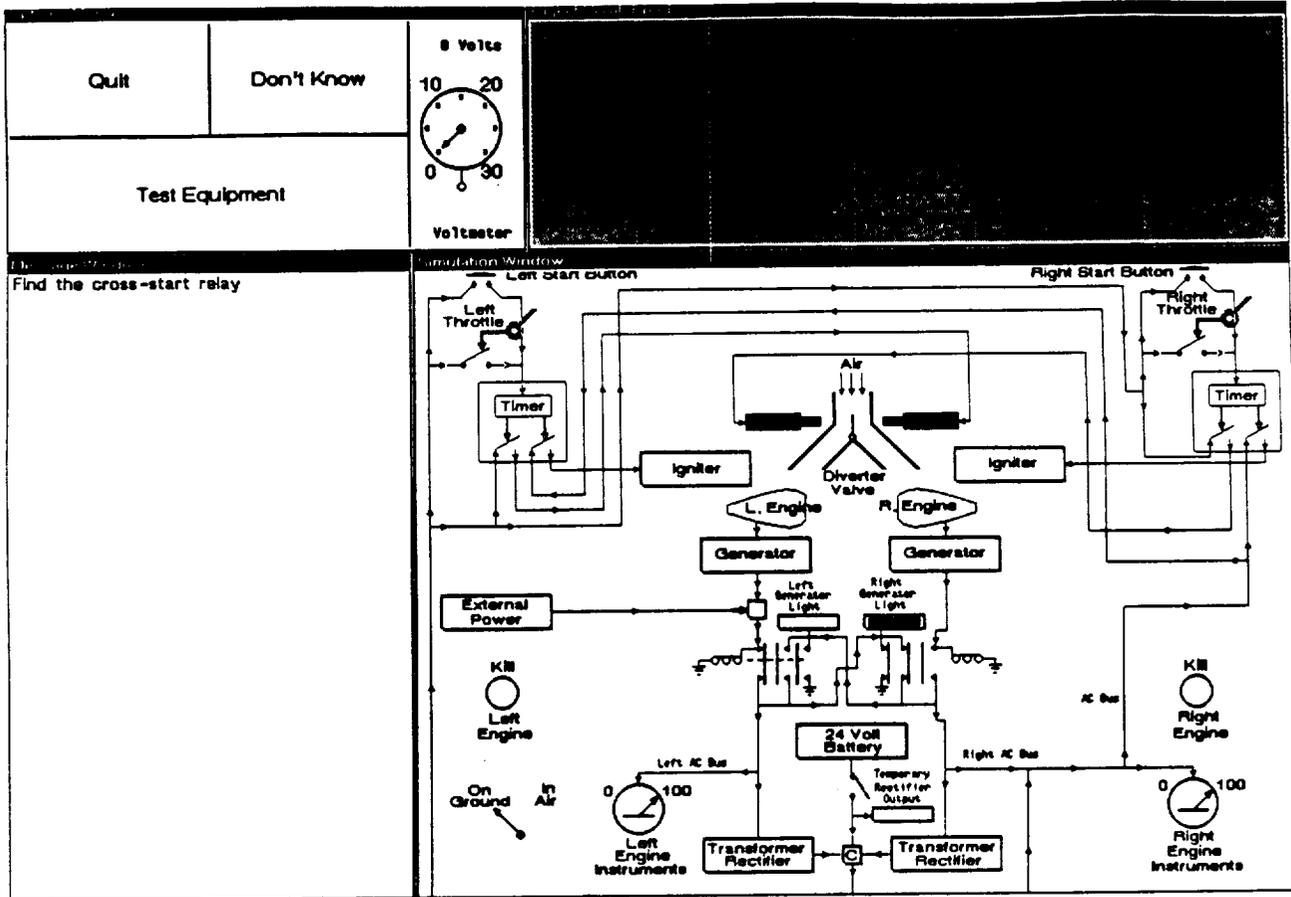


If one engine goes out in the air, it is possible to restart it using the power provided by the other engine's generator. If both engines die in the air, an emergency power switch is closed to route power from a 24-volt on-board battery. In this case, the left throttle is used, rather than the left start button, to perform the actual engine start.

In the actual aircraft, a timer opens a circuit after an engine igniter has fired long enough to start an engine. This effect is accurately simulated in RAPIDS II.

**Object Designation**

The first content unit begins with a text exposition about the schematic of the EngineStarter system and then asks the student to identify four objects on the scene. As in the figure below, the message window directs the student to click on a named object.



Try answering one or more of these questions incorrectly. Students are automatically remediated when they make an incorrect object selection. When a content unit is being played in *Drill* mode, RAPIDS II tells the student the name of the object that was selected and asks that he try again. After the second error, RAPIDS II again describes the error and then highlights the object that should have been selected, as in the figure below.

Quit	Don't Know	0 Volts 10 20 0 50 Voltmeter	
Test Equipment			
<p>Take some time to look over the engine starter system schematic shown at the right. Note the names of the elements, and try to understand the flow of power through the system. Find the cross-start relay.</p> <p>Try again. You selected the Air Diverter Valve, which was incorrect.</p> <p>You selected the Right Actuator Motor, which was incorrect.</p> <p>The correct object will now be highlighted.</p> <p>Click anywhere to continue.</p>			<p>simulation window</p> <p>Left start button      Right Start Button</p> <p>Left Throttle      Right Throttle</p> <p>Timer      Igniter      Igniter      Timer</p> <p>Air      Diverter Valve</p> <p>L. Engine      R. Engine</p> <p>Generator      Generator</p> <p>External Power      Left Generator Light      Right Generator Light</p> <p>24 Volt Battery      Temporary Rectifier Output</p> <p>Left AC Bus      Right AC Bus</p> <p>Left Engine Instruments      Right Engine Instruments</p> <p>On Ground      In Air</p>

Here the student first clicked on the air diverter valve, rather than on the cross start relay. RAPIDS II described the error and asked the student to try again. Then he or she chose the right actuator motor. At this point, the relay was highlighted to show the student what should have been selected.

As soon as the student follows the directive to "Click anywhere to continue," the highlighting will be removed and the lesson will continue with the next content item.

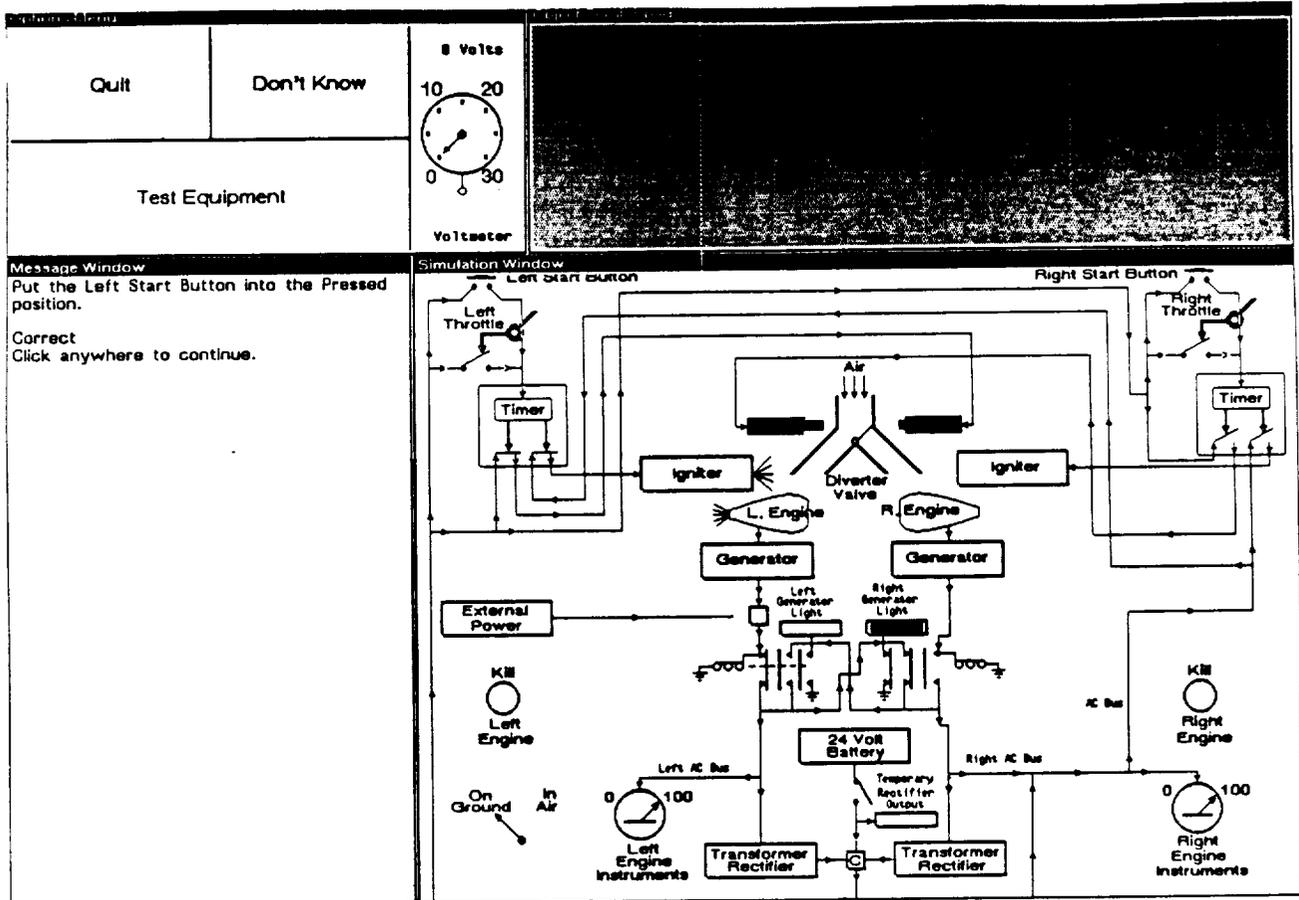
The presentation order of the four items of this small content unit has been specified as random, so you can expect to see the items in different orders if you repeat the unit.

**Required Switch Settings**

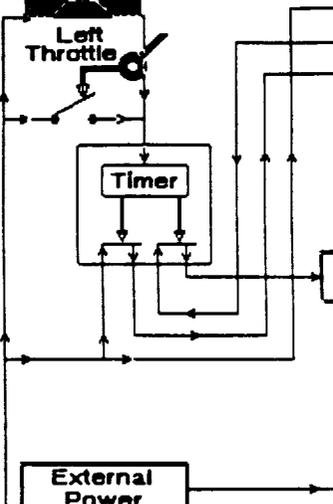
Content items can require that students change the position of a switch or other control. When the student performs the required action, the simulation is activated and all the normal effects of the action are propagated.

In the figure below, a student was asked to put the left start button into the pressed position. A number of simulation effects were propagated, including a change in the position of the diverter valve, the firing of the left igniter, and

the starting of the left engine. The external power line to the aircraft was also automatically disconnected, and the right generator warning light came on.



If a student attempts to manipulate the wrong control, the manipulation does not result in propagated simulation effects, and the control is reset to its former position. RAPIDS II automatically presents error feedback to the student in the message window. If the student makes another error, RAPIDS II gives textual feedback again, and then performs the switch setting itself, graphically highlighting the switch. These responses to incorrect student actions do not have to be authored. The RAPIDS II run time environment handles most aspects of evaluating and responding to student actions without the need for explicit authoring of those responses by the author.

Message Window	Simulation Window
<p>The simplest engine starting operation is starting the left engine when the aircraft is on the ground. The normal way to start this engine is to press the left engine start button. Put the Left Start Button into the Pressed position.</p> <p>Try again. You set the Left Throttle, which was the wrong switch.</p> <p>You set the Left Timer, which was the wrong switch.</p> <p>The Left Start Button will now be set to Pressed.</p> <p>Click anywhere to continue.</p>	

**Emphasis In Authored Presentations**

Authors have a number of tools available for emphasizing elements in the simulation during instruction. These include

- Highlighting one or more objects
- Highlighting an entire rectangular region
- Opening scratchpad windows onto other scenes
- Changing scenes
- Creating floating text windows

In the figure below, a floating text window has been opened near the cross start relay to make a point about that element of the system.

Quit	Don't Know	0 Volts 10 20  0 30 Voltmeter	
Test Equipment			
Message Window There are four sources of electrical power. You will now be asked to point to each of them in turn.	Simulation Window 		

**Multiple-Choice Text Responses**

Although RAPIDS II is particularly well-suited for authoring student interactions based on graphical direct manipulation responses (such as object selection and switch manipulations), it can also be used to prepare and present text choices to students.

These choices are presented in menu form. Authors can specify which answers are correct. Students choose START OVER if they accidentally make a choice they don't want. Once their menu selections are complete, they click on NO MORE ANSWERS.

Quit	Don't Know	0 Volts 10 20  0 30 Voltmeter	Left Start Button Left Throttle Left Timer Switch Right Start Button Right Throttle Right Timer Switch START OVER NO MORE ANSWERS
Test Equipment			
Message Window What are the two controls used to perform a normal start of the left engine?	Simulation Window 		

Multiple-choice questions can be created that have only one correct answer, or that have a number of correct responses. As the student makes choices, the text of the menu items selected appears in the message window, to the left of the simulation window. If a student makes an error, RAPIDS II automatically provides information about the error and the choice that should have been made, as is shown below.

Here the student chooses 'Left Start Button' and 'Left Throttle' from the menu, and those choices are shown in the message window.

After 'NO MORE ANSWERS' has been selected, the message window evaluates all the responses that were made. The student can try again, and RAPIDS II will evaluate the second attempt. If the student fails again, the set of correct answers is presented.

Message Window What are the two controls used to perform a normal start of the left engine?
Left Start Button Left Throttle
The following selection that you made was correct: Left Start Button
The following selection that you made was incorrect: Left Throttle
. .
The correct answers are: Left Start Button Left Timer Switch
Click anywhere to continue.

**Other Student Actions**

Authors can also create content units that call for other types of student actions, such as making indicator observations, taking test equipment readings, and replacing objects. These types of student actions are frequently used in constructing troubleshooting courses. As with the types of required student responses described above, corrective feedback is generated automatically when the student makes an error.

---

---

## The Options Menu

The Options Menu has two or three items: **Quit**, **Don't Know**, and **View** (which appears if there is more than one scene in the simulation). The **Quit** command lets a student stop a training session. If a student chooses **Don't Know**, the message window will display the correct answer and will ask the student to click the mouse to continue the training session. If appropriate, objects or regions in the simulation window will be highlighted to clarify the supplied answer. Items that the student responds to with **Don't Know** are scored as errors by the RAPIDS II scoring mechanism.

**View**

If the simulation used in a RAPIDS II course has more than one scene, then the Options Menu will include a third item, **View**. This command brings up a tree of scenes in the simulation. Clicking on one of these names brings that scene into the simulation window.

---

## Modes of Instruction

Content units may be delivered in any of three modes: instruct mode, drill mode, and test mode. The examples of content item interactions presented above are all based on drill mode.

### Instruct Mode

Instruct mode is not very demanding of the student. In this mode, a content unit's pre-exposition is presented. Then, for each item, RAPIDS II presents the item's pre-exposition (if one exists) and its identifying text. The student must then click the mouse, but need not perform the identified student action. RAPIDS II carries out the student action. Then post-exposition is presented. See the example at the right.

The student's task in instruct mode is simply to pace the presentation of text and graphics by clicking the mouse.

Do you want to start the next topic?

The simplest engine starting operation is starting the left engine when the aircraft is on the ground. The normal way to start this engine is to press the left engine start button.

We will put the Left Start Button into the Pressed position.

Click anywhere to continue.

Note that the diverter valve is positioned to channel air to the left engine and the igniter is powered.

When an engine is started in the actual aircraft, the timer opens the circuit that powers the igniter after enough time has passed to start the engine. In order to make the sequence of events clear, this timer effect is not performed automatically in this simulation. You must change the ganged switch in the timer by clicking on it. We will put the Left Timer into the Open position.

Click anywhere to continue.

### Drill Mode

In drill mode, expositions and identifying texts are presented, as in instruct mode. In this mode, however, the student must actually perform the action described by the identifying text. Performing any other action results in the presentation of generated corrective feedback, as shown in the examples of the previous section.

For many training tasks for which RAPIDS II is an appropriate development and delivery system, drill mode is likely to prove the most useful mode of instruction.

### Test Mode

In test mode, pre- and post-expositions are not presented. Identifying text may or may not be presented, at the discretion of the content unit author. In response to each identifying text, the student must perform the identified task. As in drill mode, corrective feedback is generated in response to student mistakes. In test mode, however, the corrective feedback does not discuss what action the student actually took. Instead, it simply indicates the correct action.

## Building Generic Objects

### The Role of Generic Objects

Generic objects are the prototypes for specific objects that appear in RAPIDS-II simulation scenes. Generic objects store object appearances and may contain much of the specification of object behavior as well. (In unusual circumstances, generic objects may contain no graphics or no behavior. These special cases are discussed later in this chapter.)

Generic objects are created and edited using the Generic Editor. It can be used to draw the appearances of objects and to open the RAPIDS-II rule editor for describing object behavior.

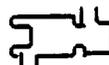
#### Appearances

The appearance of an object is determined by its *object graphics* and its *state graphics*. The *object graphics* part is always the same, no matter what the object's state. The *state graphics* change depending on the state of the object.

Consider the sequence valve shown below. When pressure at the top port on the valve is greater than pressure at the bottom port, then the plunger in the valve is pushed to the left and pressure can pass through the valve from top to bottom ports. This state of the valve is depicted on the left. When pressure is greater at the bottom port of the valve, then the plunger is extended and pressure cannot be passed from the lower port to the upper one.



No matter what state the sequence valve is in, a portion of its appearance is unchanged. That portion is the *object graphics* of the valve, shown below.

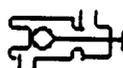


You will create the *object graphics* of objects using the generic editor's **Object Graphics** command and a drawing palette, which is described below in this chapter.

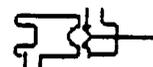
### State Graphics

State graphics can change in either of two ways. A state part can vary continuously, by being moved or rotated with respect to the static part. Alternatively, a state can be replaced in its entirety by a different state for that object. Whether you want a particular object's appearances to be handled by a continuous state or by alternative state graphics is your decision. For many objects, either approach will work fairly well.

If you choose the alternative state graphics approach, you will draw the object graphics and each state graphic alternative. You will give each state a name that corresponds to its appearance. (In the case of the sequence valve, you will create the second state by copying the appearance of the first state and dragging the state graphics to a new position.)

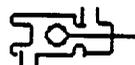


Open



Closed

If you choose the *continuous graphics* approach, you will draw the object graphics and one state. You will then write a behavior rule that describes how the state graphics should be moved or rotated to reflect attribute values. Creating and editing rules is described later in this chapter. When you choose the *continuous graphics* approach, the position (or rotation) of the state part is computed during the simulation process.



Depending on computed values, a state may be shown at any intermediate position, as in in the figure above.

### Alternative States or Continuous?

Continuous state graphics make sense if a simulation is computing attribute values that can be used to determine the position or rotation of the changeable part of an object. In some simulations, you may be able to achieve quite smooth animation effects without having to specify all the intermediate positions (or rotations) of an object's states.

If an object changes its shape (by deforming or taking on a very different appearance), rather than merely repositioning or rotating some part, then you must use the approach of separately drawing and naming the different state appearances.

### Generic Behavior Rules

The rules of a generic object can refer only to attributes of that object, not to the attributes of any other objects. These are the prototypes for a specific object's *internal rules*. In many simulations, most of the 'behavior' derives from such generic object rules. In these simulations, the *external rules* of a simulation scene are used primarily to link specific objects together.

## Using the Generic Editor

### Starting the Generic Editor

There are 2 different ways to start the generic editor. The normal way is to start the editor using the RAPIDS-II menu. The top menu item in the *Simulation* menu is *Generic Editor*.

RAPIDS II Tools	
Simulation	Instruction
Generic Editor	Content Editor
Scene Editor	Plan Editor
Build Simulation	Run Instruction
Run Simulation	

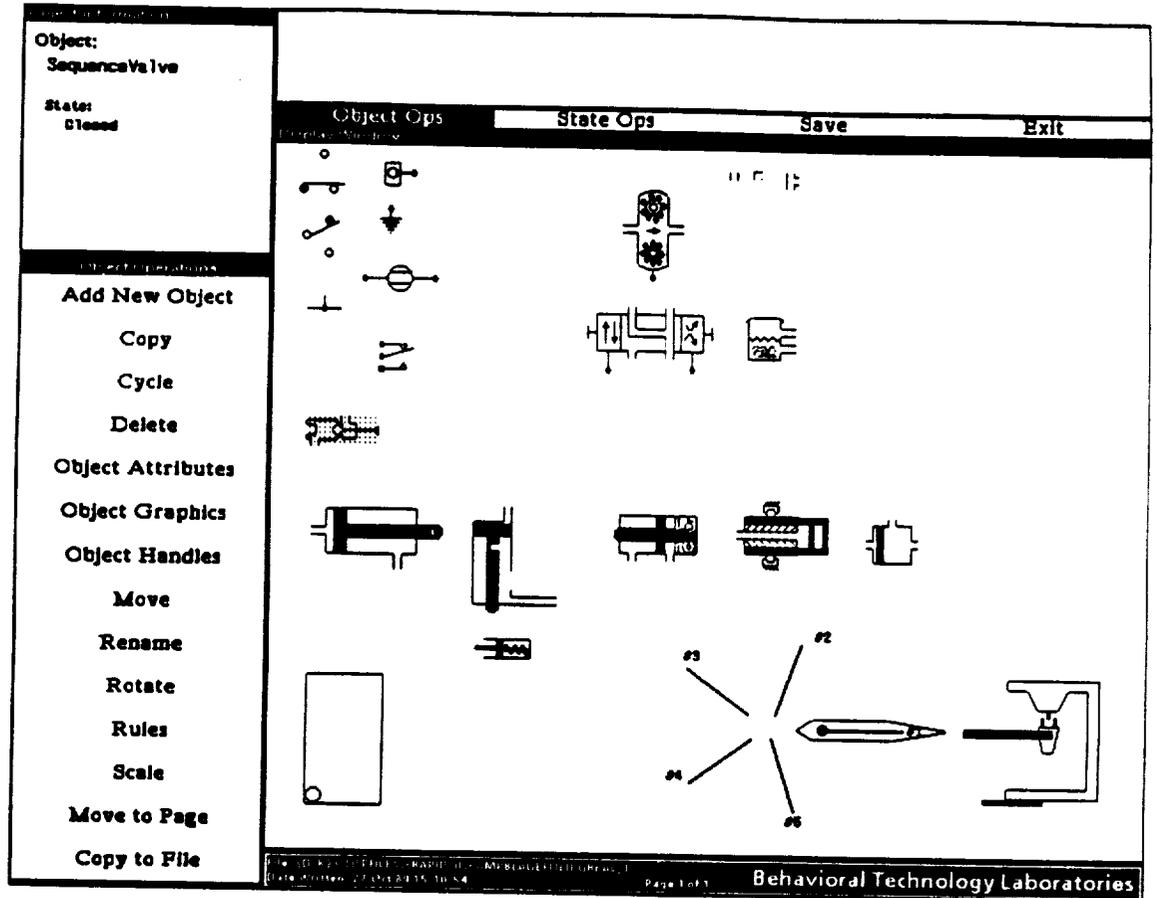
When you click on this button, a dialog box will appear that will ask you for the name of the library of generic objects (the *Generic File*) that should be edited. Click on *Generic File*, then type in the file name, then click on the *OK* button. (If you want to convert an old IMTS library to RAPIDS II data, then change the *IMTS File?* field in the dialog box to *T* before clicking on *OK*.)

Generic File: ENGINESTARTER<sub>A</sub>

IMTS File?: NIL

(The second way to invoke the generic editor is to call the *GenericEdReal* function by typing  
*(GenericEdReal 'LibraryName)*  
in an Exec window.)

When the generic editor has been successfully started, you will see a set of windows that look like those shown below.



The long shallow window at the top of this editor is the *message window*. In addition to displaying messages relevant to the editor tools you choose, this window displays prompts for data, such as object names, that must be typed. Your typed responses appear in this window during these interactions with the editor.

The menu below the message window is used to change between the object operations mode and the state operations mode of the editor. It is also used to save changes and to exit the editor.

The large window is the *display window*. It displays the objects that are in the library you are editing.

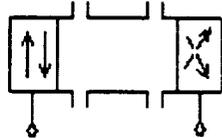
To the left of the display window is the *options menu*. Creating RAPIDS-II generic objects requires the use of two major modes, one for object-level operations and one for object-state operations. Each of these modes has its own associated menu of available options, which is always displayed to the immediate left of the display window. In the figure above, the menu is that associated with the object operations mode.

At the bottom of the set of windows is a long thin window that provides *file information* about the current library. It shows which file you are editing. If changes have been made but not saved, this window will appear in inverse

video — it will display white characters on a black background, rather than the normal black on white.

**The Graphic Parts of a Multi-State Object**

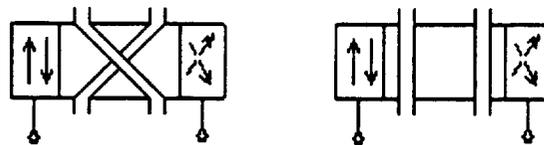
RAPIDS-II generic objects have two kinds of appearance elements. One is a static portion. This is the part of an object's appearance that doesn't change when the object changes states. The other kind of appearance element is the state appearance. This is the part of the object that is visually different in different object states.



*The Static Part of an Object's Appearance*

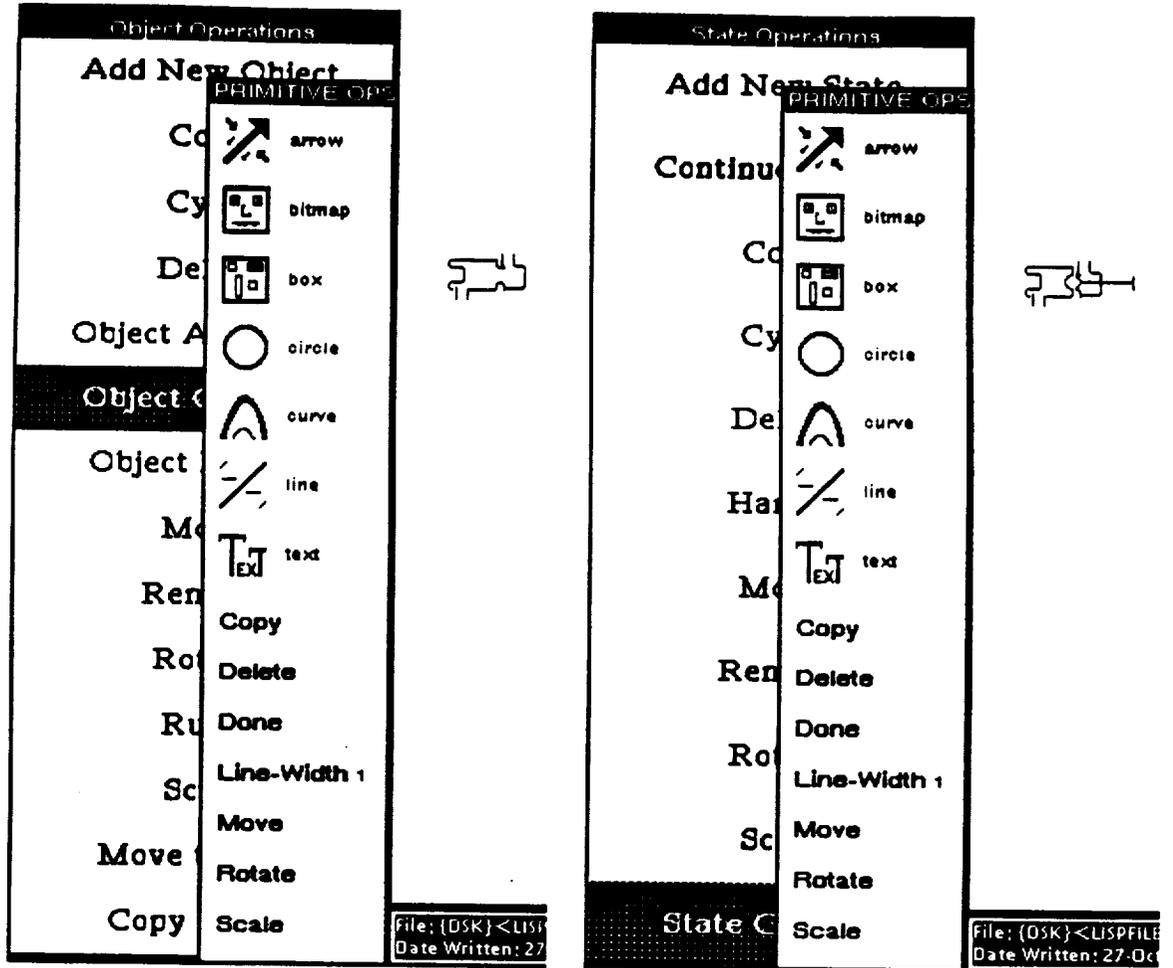


*Two State Parts*



*The Object Shown in the Two States*

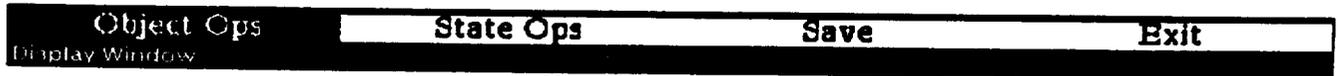
To draw an object you will have to use three menus. The *Object Operations* menu is used to issue commands relevant to entire objects. When you actually draw the static part of the object, a *Primitive Ops* menu, which has the drawing tools, will appear. The *State Operations* menu lets you perform special operations on object states in addition to providing access to the *Primitive-Ops* drawing menu. In the two figures below, the *Primitive-Ops* menu is shown overlaid on the two top-level menus (*Object Operations* and *State Operations*).



Drawing the Static Part of the Sequence Valve

Drawing the State Part of the Sequence Valve

As the above figures indicate, the drawing menu (which is labeled PRIMITIVE-OPS) can be overlaid on either the *object operations* menu or the *state operations* menu. These two modes are the basic modes of the generic editor. You change between these modes by clicking on the appropriate option in the menu bar over the Display Window.



The menu bar is used to choose between the Object Operations mode and the State Operations modes of the generic editor. It also has the *Save* and *Exit* commands. *Save* simply saves the current version of the library you are working on, using the original name you specified when you opened the generic editor. As with other Interlisp-D applications, a version number extension will be appended to the file. By renaming an earlier copy of a library file (or by explicitly specifying the extension), you can edit an earlier version of the library, if necessary.

**Exit**

Use this option to end the current session of the generic editor and close its windows. If your changes have already been saved, you will be presented with two choices

*Exit*  
*CancelExit*

If your latest changes have not been saved, you will be presented with a menu that gives you three choices

*SaveFileBeforeExiting*  
*ExitWithoutSaving*  
*CancelExit*

If you click on the first option, the results of your work in the session will be preserved. The second option lets you stop working with the generic editor but throws away your changes to the object library. You'll be required to confirm this command before it will be carried out. The last option cancels the *Exit* choice and lets you continue the editing session.

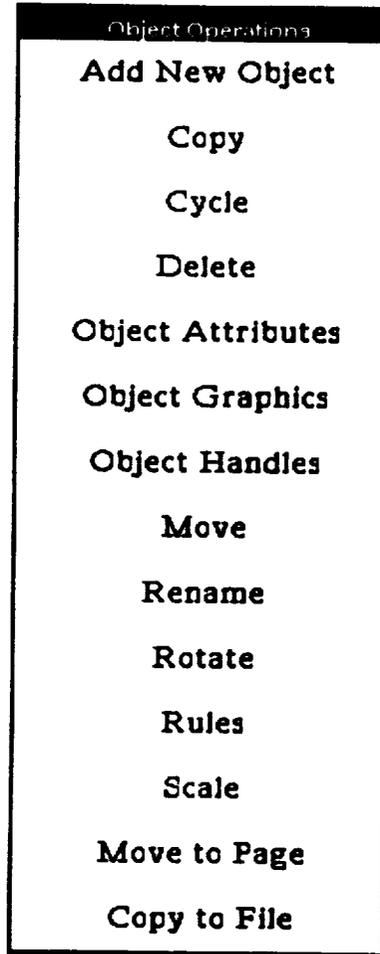
The *Exit* option may also lead to further prompting, if you used the command that copies an object from the current library to another library. You will be asked whether you want to complete that transfer by saving the destination file with the new object.

**The Two Major Modes of the Generic Editor**

The object operations mode is used to perform actions on an object as a whole. The state operations mode is used to carry out actions on a state part of an object. The next two sections of this chapter explore these modes of the generic editor.

## Object Operations

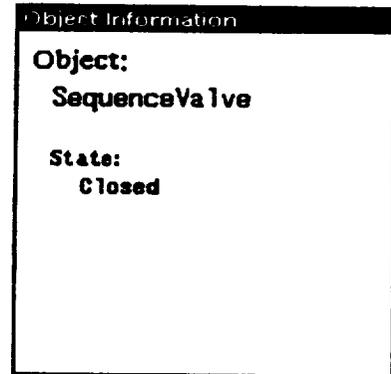
### The Object Operations Menu



The *Object-Ops* menu, as with most other RAPIDS-II editor menus, appears at the left of the main editor window. Its primary uses are

- to examine already-defined objects
- to edit such objects
- to create new objects.

The window at the top of this menu to displays the object name and state name of the currently selected generic object. This window is the *Object Information Window*. As is discussed below, other information is displayed in the object information window when certain operations are being performed.



Most object operations are performed on a *selected* object. To *select* an object, point to the object and click the left button.

Some objects may have no appearances, and therefore cannot be selected by clicking on them. To select such objects, hold down the middle button anywhere in the display window. A menu will appear, asking you to choose which of the listed invisible objects you wish to select. If there are no invisible objects, then the message window will tell you that there are no invisible objects to select from.



**A Note on Hiding Objects**

Depending on the library of generic objects you are working on, you may find that the display window is quite cluttered. Once you have selected an object to edit, the rest of the displayed library can be thought of as background. Sometimes you want these background elements to be displayed, because you will want to draw a new object in proportion to the other object types that it will appear with. At other times, however, you will want to make this background of objects invisible, so that you can concentrate on the selected object. To hide the background after you have selected an object

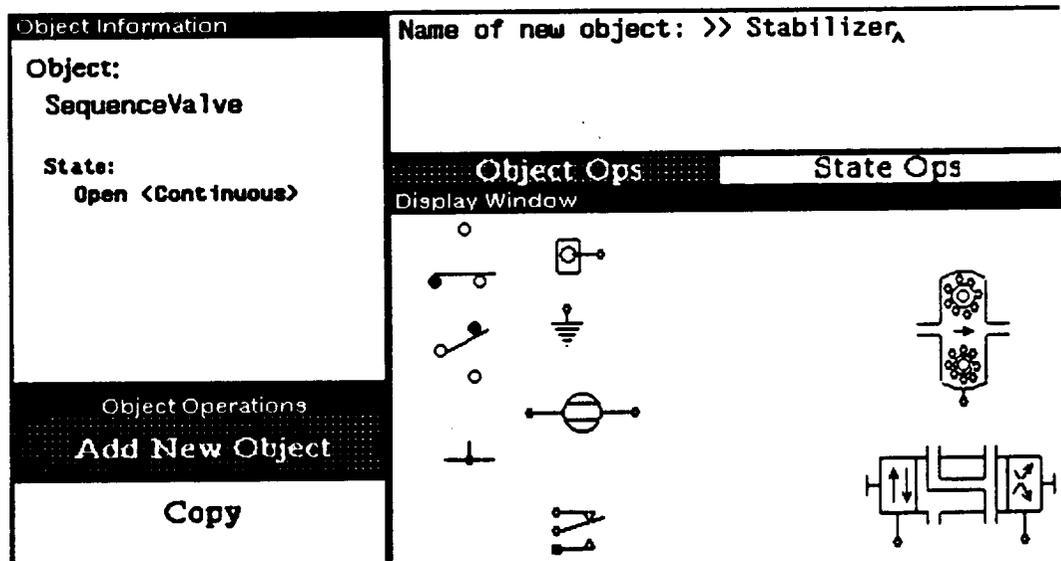
- Move the mouse pointer into the display window.
- Press and hold down the right mouse button. A menu will pop up.
- Point to the *Background* menu item and release the mouse button.

All the objects but the one selected will disappear. To bring them back, use the right mouse button to bring up the display window menu and choose *Background* again. The Background command is used to toggle the visibility of all the objects that are not currently selected.

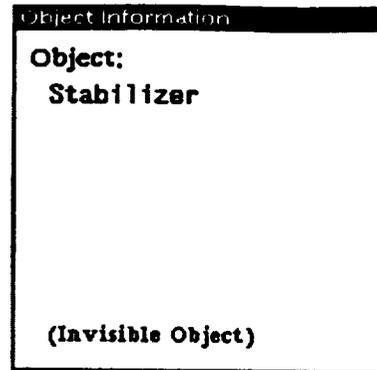
The remainder of this section describes the *Object Operations* commands.

**Add New Object**

The *Add New Object* command is used to create new generic objects. When you select this menu option, you will be asked to name the new generic object. If the name you choose is already in use, then you will be prompted to choose again.

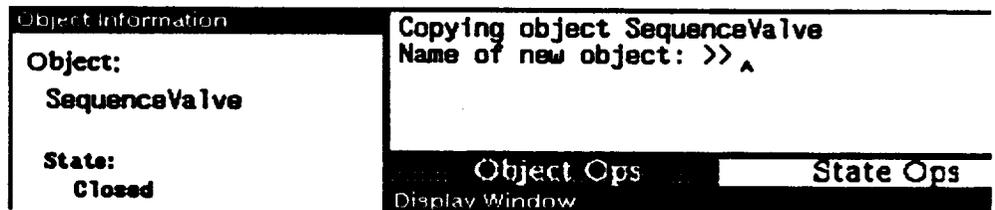


When you type the Return key, a new object with that name is created. The new object is the selected object. This object is invisible, since no graphics have yet been created for it. The object information window reflects this state of affairs:



**Copy**

*Copy* is used to create a new object that is identical to the currently selected object. First select the object you want to copy, then click on *Copy*. You will be prompted to type the name of the new object. After you type the new name and press Return, the new copy will appear on the screen. It will be selected, and it will be ready to move (without first clicking on the *Move* menu item). Position it where you want it by moving the mouse in the library window and click the left button to position it. The copy remains selected, so subsequent operations will apply to it until you select a different object.



**Cycle**

Clicking on *Cycle* makes the selected object appear in its next state. You can repeatedly click *Cycle* to see all of the appearances of a generic object. Notice that the state name is displayed in the object information window at the top of the menu.

**Delete**

The selected object is deleted. Naturally, the behaviors as well as all the appearances of the object are deleted. After you click on *Delete*, the message window asks whether you want to delete the object and the mouse pointer turns into a little picture of a mouse. Click the left button to confirm the deletion; click the right button to abort the deletion of the object.

**Object Attributes**

An object's attributes are the named variables that hold values associated with the object. When you click on *Object Attributes*, an attribute editing window appears to the left of the display window. In the figure below, the attributes for the Sequence Valve generic object are displayed.

Object Attributes		
Object Graphics		
Attribute Operations		
Add	Delete	Done
Object Attributes		
Attribute Name	Type	Handle Region
<i>StateLocX</i>	<i>Real</i>	
<i>CurrentState</i>	<i>Atom</i>	
<i>ObjectLocX</i>	<i>Integer</i>	
<i>ObjectLocY</i>	<i>Integer</i>	
<b>extend-pressure</b>	<b>Integer</b>	<b>(47 130 10 10)</b>
<b>retract-pressure</b>	<b>Integer</b>	<b>undefined</b>
<b>Force</b>	<b>Boolean</b>	<b>undefined</b>



Some of the attributes appear in italic type, while others are in boldface. The ones in italics are attributes that are created and maintained automatically for the object. Any object with an appearance will have the attributes *ObjectLocX* and *ObjectLocY*. Any object with state parts will have the attribute *CurrentState*. Any object with a movable state will have the attribute *StateLocX* or *StateLocY* or both. Any object with a rotatable state will have *StateRotation* and *StateRotationCenter*. These attributes cannot be changed or deleted using the *Attribute Operations* tool.

The attributes shown in bold type style are ones that obtain their values from the attribute value *assignments* performed by your rules. Those in italics receive their values as a result of certain operations such as moving an object or rotating a state. Your rules control these values when they include built-in functions such as *MoveLocX* and *Rotate*.

You can change the name of an attribute in the object attribute editor by clicking in its name with the left button and then backspacing and typing. If you click the name with the right button, the name will be deleted and you can type a new one. Within an object, attribute names must be unique. If you try to give an attribute a name that is already in use, such as *StateLocX*, then the editor will append a number to the new name, as in *StateLocX1*. You cannot use the name of predefined attributes even if the attribute is not currently in use.

You can also change the type of an attribute. Click on the type name, and a menu of type options will pop up. Choose the appropriate type for the attribute.

If an attribute has a handle, then the region of that handle will appear in the column labeled *Handle Region* of the *Attribute Operations* window. (A region is displayed as a set of four numbers, representing the values of the left, top, bottom, and right of the region rectangle.) If an attribute does not have a

handle, then the handle region will appear as *undefined* in the *Handle Region* column. You can change the handle region of an attribute by clicking on its value with the left button. You will be presented with options for changing, adding, or deleting the handle, similar to the corresponding actions for Object Handles. If you click with the middle button on the handle region, then that attribute's region will be highlighted in the display window.

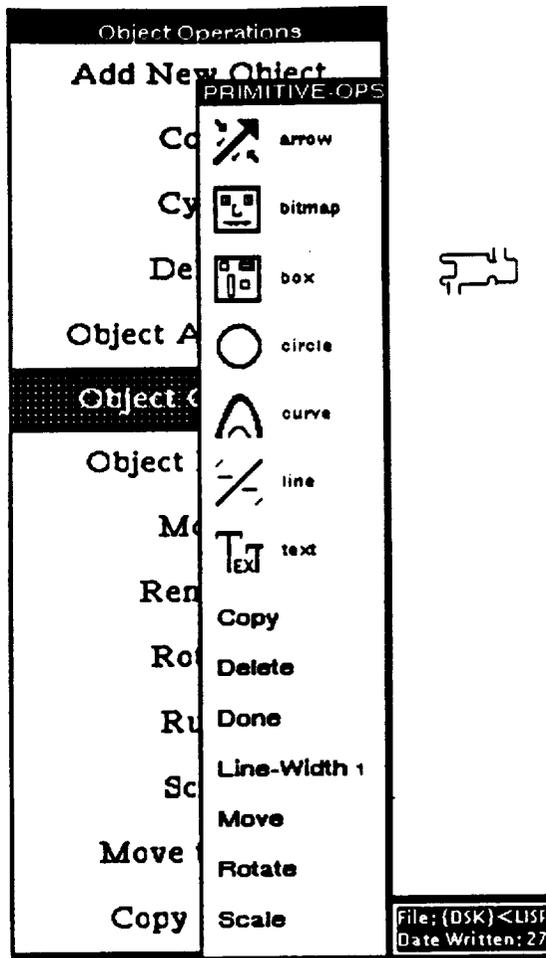
The object attribute editor will also let you add new attributes and delete old ones. To add a new attribute, simply click *Add* in the editor's three-item menu bar. A new, generated attribute name, such as 'Attribute102' will appear, and its type will be 'Atom.' You can change the name and type of this new attribute in the same ways that you would edit an old attribute's name or type.

Attribute Operations		
Add	Delete	Done
Object Attributes		
Attribute Name	Type	Handle Region
<i>CurrentState</i>	<i>Atom</i>	
<i>ObjectLox</i>	<i>Integer</i>	
<i>ObjectLocY</i>	<i>Integer</i>	
<i>P-rear</i>	<i>Integer</i>	<i>undefined</i>
<i>P-front</i>	<i>Integer</i>	<i>undefined</i>
<i>Force</i>	<i>Boolean</i>	<i>undefined</i>
<i>Attribute102</i>	<i>Atom</i>	<i>undefined</i>

To delete an attribute, first click on the *Delete* option in the menu bar. You will be instructed to select the attribute to be deleted. Clicking anywhere in the line that represents the attribute will delete it. You will not be asked to confirm the deletion; it will just happen. Always click carefully after selecting *Delete*.

### Object Graphics

The *Object Graphics* command is used to create and edit the static part of an object's appearance. Objects need not have any static part, so it is possible to create a functional generic object without using this command.



When you choose the *Object Graphics* command, all the objects except the currently selected one disappear from the display window. (You can make these objects reappear using the *Background* command in the *Window Operations* menu. You may want to have other objects visible so that you can draw the current object so that it will mesh appropriately with the other objects that will appear with it.) Any displayed state appearance graphics will also disappear.

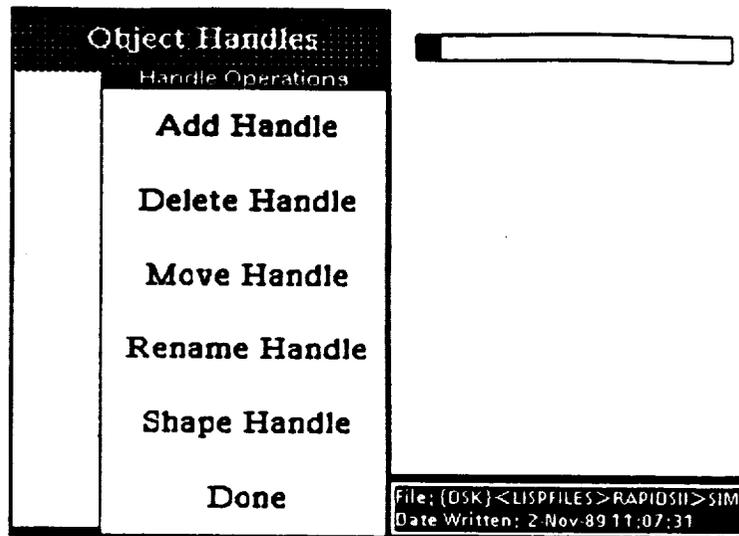
The *Primitive-Ops* menu will appear, partially overlying the *Object Operations* menu. Using the graphics tools and the commands of the *Primitive-Ops* menu, you can draw or graphically edit the unchanging part of the selected object's appearance. Graphic editing is described below in the section called 'Drawing Operations.'

To get out of the graphic editing mode, click *Done* on

the *Primitive-Ops* menu. The menu will disappear, and you will again be able to issue other object operation commands.

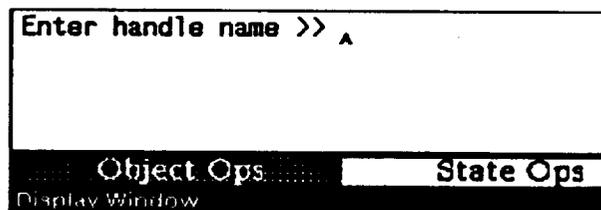
**Object Handles**

In RAPIDS-II, the term *handle* refers to a designated graphical area (associated with an object) that is sensitive to mouse clicks. Any object that you would like students to be able to manipulate directly with the mouse must be given one or more handles. Most handles are state handles. They are not created using the *Object Handles* command, but rather the *Handle* command in the *State Operations* menu, which is described below in the section on 'State Operations'. Object handles are used when you want the handles to relate to some attributes of the object rather than directly to its states.



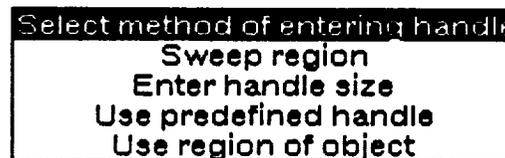
When you select *Object Handles*, a new menu of *Handle Operations* appears, overlying the lower portion of the *Object Operations* menu. So long as this menu is visible, the generic editor is in the *handle operations* mode. This mode is used to add, delete, move, rename, and shape the handles of an object. (Remember that most objects do not have handles. Only those that can be *directly* manipulated with the mouse should have handles.)

If you click on *Add Handle*, the message window will prompt you for the name of the handle that you want to add. Type in an appropriate name for the handle and type the Return key.



When the selected switch or other control is made up of a small number of alternative states, you will ordinarily create a handle that corresponds with each state. It may therefore be appropriate to give each handle a name that corresponds to the state into which it will put the object.

After you have entered the name of the object, a menu will appear that asks what kind of handle shape you want to use:

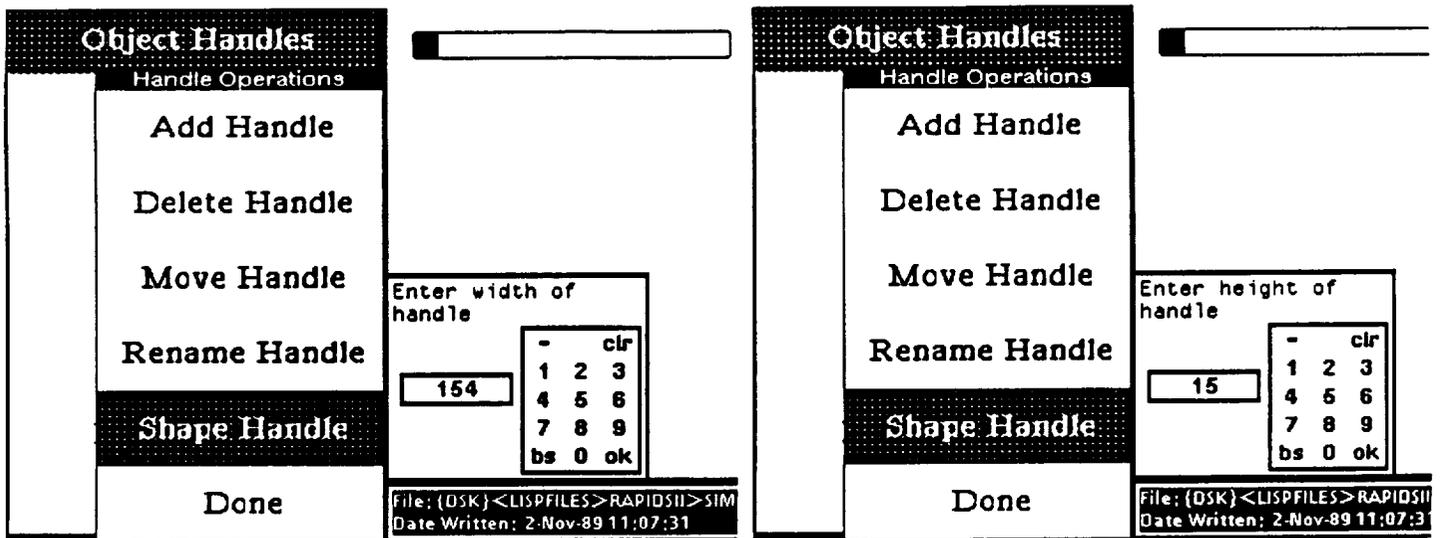


There are four different handle shape options. Whichever method you use to create a handle, make sure that the handle is within the graphics of the object.

RAPIDS-II is not able to detect handle manipulations outside of the rectangular bounding box that encloses all the graphics of the object. In any case, it is always a good idea to graphically indicate the 'mouse-active' areas of an object, so that it will be clear to students how they can manipulate them.

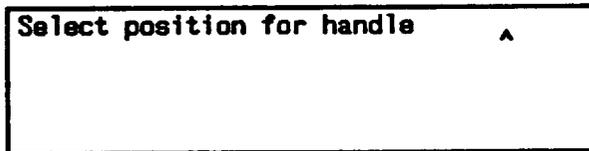
If you select the first handle shape option, *Sweep region*, the mouse shape will change to the standard Expanding Box cursor. Drag out a rectangular region that will serve as the named handle. Any student click within this area will be considered a handle manipulation.

The second way of designating a handle is to *Enter handle size*. If you choose this method, you will be prompted to enter the width and height of the handle region.



For objects with discrete states, it is often appropriate to use the *predefined handle* type when creating *state* handles. It is also possible, however, to make one of these small square regions an object handle.

When you are creating a predefined handle, the cursor will change into the shape of one of these handles, a small black box, and the message window will prompt you to place the handle at the appropriate place on the object.

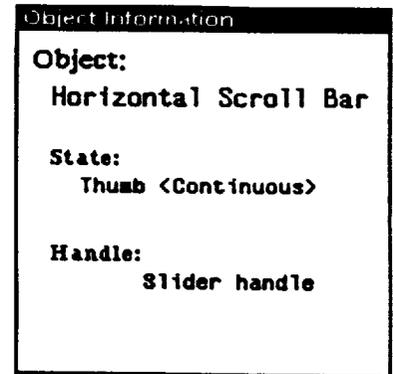


Just click the left button of the mouse where you want the handle to be placed. Don't be concerned about the appearance of a black square at that point. This visual representation of the handle appears only when you are in the generic editor's handle-editing mode.

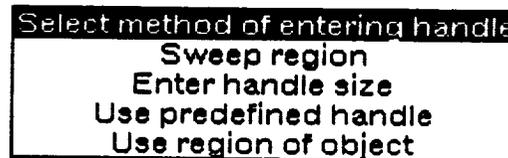
The fourth shape options for handles is the easiest one to use. The choice *Use region of object* will simply treat the entire bounding region of the object as a

handle region. That region will be highlighted, just as the handle regions you can create using any of the other methods are highlighted during the Handle Operations mode.

Handles can be selected by clicking on them when you are in the Handle Operation mode. A selected handle will be highlighted by a rectangular border. When a handle has been selected, its name appears in the *Object Information* window, below the state name.



Most of the *Handle Operations* menu commands apply to the selected handle. *Delete Handle* will ask that you confirm the deletion of the selected handle by clicking the left mouse button. If you don't want to carry out the deletion, click the right button. *Move Handle* will let you move the selected handle using the mouse. Click the left button when it is positioned where you want it. *Rename Handle* will bring up a prompt for a new name in the message window. The *Shape Handle* command will pop up the menu of handle shape options discussed above, so you can use any of the standard handle authoring methods to edit the selected handle's shape.



Finally, clicking on the *Done* command in the *Handle Operations* menu will take you out of the Handle Operation mode, back to the Object Operations mode.

**Move**

To move the currently selected object, click on the *Move* command on the *Object Operations* menu. The message window will display a message showing that the selected object is 'hooked to the mouse' and will move with the mouse. Click the left button to position the object and end the *Move* operation.



**Rename**

Choose the *Rename* command on the *Object Operations* menu to change the name of an object. You will be prompted to enter a new name for the selected object type. Type the name and press the Return key. The name appears in the *Object Information* window. (See below.)

Object Information	Rename Slider to: >> Horizontal Scroll Bar <sub>A</sub>
Object: Slider	Object Ops
	State Ops

Remember that you are naming a **generic** object, not a specific instance. Don't call a two-position generic switch a "power switch" just because you plan to use an instance as a power switch in a specific simulation. Give it a more generic name, such as "Two position switch."

**Rotate**

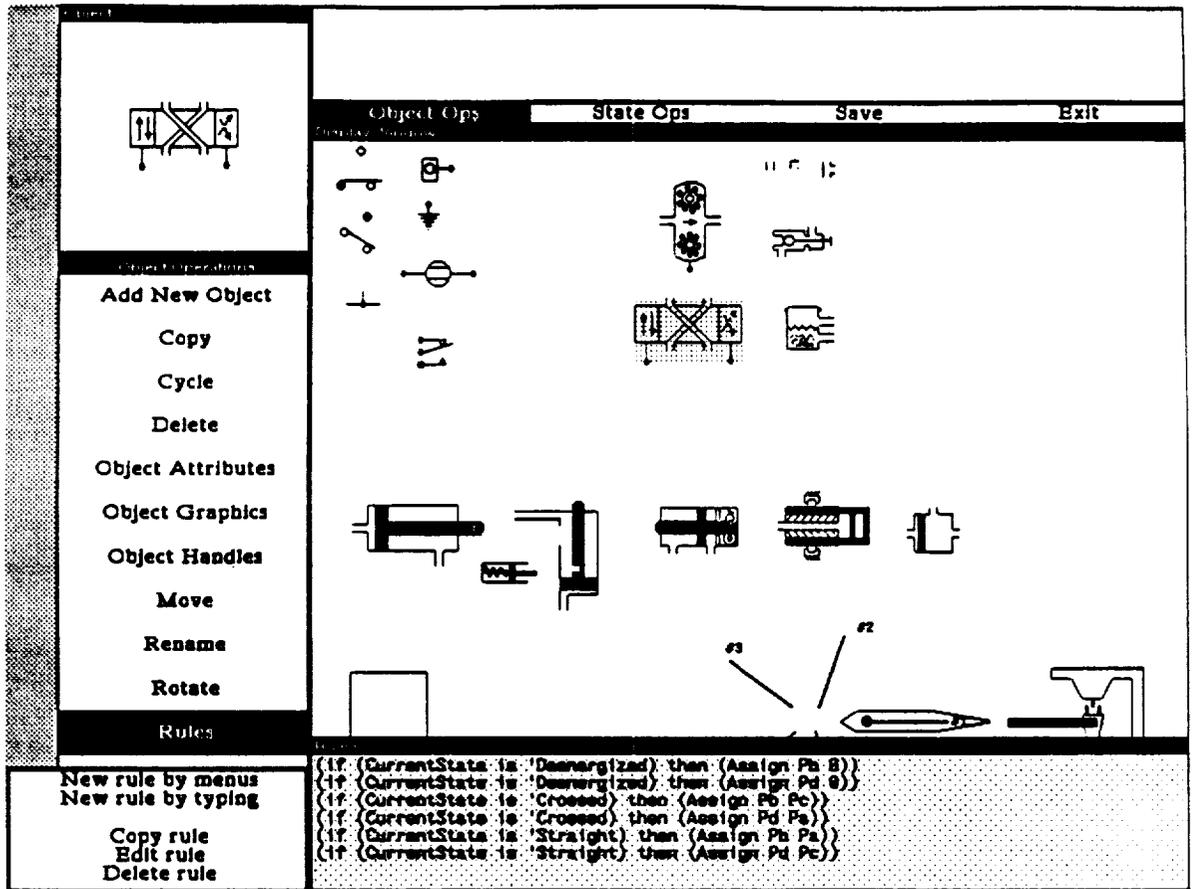
This command is used to rotate a selected object. You will be prompted to enter the number of degrees of rotation. Type in a number (counter clockwise rotation is positive) and press the Return key. Rotation will be about the center of the bounding region of the object.

How many degrees do you wish to rotate object Vertical Scroll Bar? >>
Object Ops
State Ops
Save

**Rules**

The *Rules* command closes the generic editor windows (temporarily, just to reduce screen clutter). It opens a new set of windows, the RAPIDS-II rule editor. This rule editor is available in both the generic editor and in the scene editor, although there are minor variations in the availability of features in the two environments. The figure below presents the initial appearance of the rule editor.

The top window shows a picture of the selected object (here, the sequence valve). The rules displayed in the window immediately below are the rules that are associated with that generic object.



When an existing rule is being edited, other windows open as well. The section on 'Internal Rules' later in this chapter presents the use of the rule editor in some detail.

After you finish working on the rules of the selected object, you must click *Done* in the leftmost menu. The rule editor windows will close and the windows of the generic editor will reopen, just as they were when you clicked on *Rules* in the *Object Operations* menu.

**Scale**

Generic objects can also be scaled. When you choose this menu command, you will be asked for the amount of scaling. To make an object one-and-a-half times as big, enter 1.5. To make it half its former size, enter .5 and type the Return key. The object will be redrawn in the main window in the new size you have specified.

By what factor do you wish to scale object Horizontal Scroll Bar? >>

In some cases you will find it convenient to have copies of objects with different scaling and rotation. First use the *Copy* command, then the *Scale* and *Rotate* commands to build such copies.

#### Move to Page

Libraries can have multiple *pages*. A library page is a view the size of the generic editor's display window. You can change pages using the display window's right button menu. The *Move to Page* option is used to move the selected object from one page to another. You will be prompted for the number of the page to which you want to move the object.

```
There are 2 pages.
Which page should object Horizontal Scroll Bar be moved to (enter 0 for a new page)?
>> A
```

You can add a new page to the library by responding *0* to the request for the destination page number. The actual page number of the new page will be one greater than the number of the former last page.

#### Copy to File

Sometimes you may find that a given generic object should be included in a different library. You can move a copy of the selected object to a library other than the one that is currently open. Enter the name of the file you want to put the object into when the prompt below appears in the message window. Don't precede the file name with a quote mark.

```
Copy object Horizontal Scroll Bar to library? >> A
```

If, when you are prompted for the name of the destination library, you type the name of a file that doesn't exist, then the generic editor will create a new library of that name and will put a copy of the selected object into it.

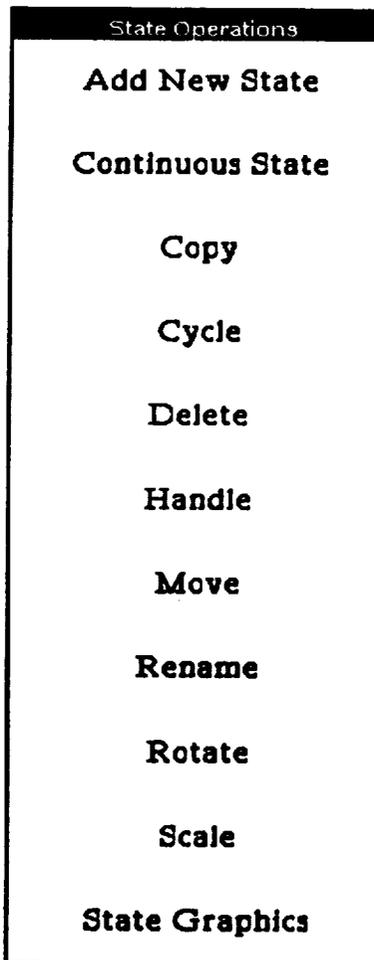
The copy will not actually be completed until you issue a *Save* command or exit the generic editor. At that time the file that includes the other library will be rewritten with the new (copied) object. If you choose to terminate your editing session without saving, you will be asked whether you want the object copy to another file to actually take place.

Be aware that it is possible to copy the same object two or more times to another library. If the object has been copied to the library from the same source library, then all the copies will be in exactly the same place on the screen in the destination library. If it was copied an even number of times (e.g., twice), then it will appear not to be present at all — the copies will erase each other. If this happens to you, then when you later edit that library, you should delete the superfluous objects. It is also possible to copy into a library an object with the same name as a different object that is already there. In this case you should edit the library and change the name of one of the objects.

**A Hint: Create an Empty, Invisible Object**

Authors often find it useful to have an empty generic object, one with no appearances or behavior. Later, when building a simulation in the specific editor, you can create instances of this 'empty' object, assign appropriate graphics, attributes, and behavior rules to each instance, and thereby customize your simulation in ways you had not planned on when you created the library. These invisible objects can be very useful, but they cannot serve as the basis for complete graphical simulations, because they cannot have appearances that change under the control of rules.

## State Operations



Once you have created the unchanging static part of an appearance, you will ordinarily build the state part or parts. The *State Operations* menu is used to name, draw, and position the variable parts of an object's appearance, the state-dependent graphics. To get into the State Operations mode, choose *State Operations* from the menu bar at the top of the display window. As in the Object Operations mode, the Object Information window appears at the top of the menu to display the name and state of the selected object.

While you are in the State Operations mode, you can change the selected object in just the same way that you do in the Object Operations mode. Simply click on the next object that you want to edit the states of.

The state operations commands operate on the selected state, just as object operations (in the *Object Operations* menu) operate on the whole object. If you want to create a new state to work on, you can use the *Add New State* command. See its description below.

**Add New State**

You can add a new state to a selected object using the *Add New State* command. The message window will prompt you for the name of the new state, as shown in the figure below. After entering the name (by typing it and then pressing the Return key), you will usually draw the state, using the *State Graphics* command.

Object Information	Please enter a name for the state >> Maximum P
Object: HydraulicPump	
State:	
	Object Ops      State Ops

**Continuous State**

The *Continuous State* construct provides one of the most powerful features in RAPIDS-II. It is used to create objects that vary continuously in appearance, typically based on the value of some attribute.

When you choose the *Continuous State* command from the *State Operations* menu, a new set of windows, labeled 'Continuous State Adjustments' appears at the bottom of the *State Operations* menu. These windows are shown at the right. The upper window provides a control interface for experimenting with the appearance of the continuous object.

The lower window is used to set the movement and rotation limits of the continuous state. For example, if a state part is to move horizontally, you would enter values for *MinX* and for *MaxX*. A value of 0 refers to the location of the state as it was created using the other state operation tools, such as the drawing tools and *Move*. Negative values are to the left of the state's original location, positive values are to the right.

State Operations	
Add New State	
Continuous State	
Copy	
Continuous State Adjustments	
X Translation: <input type="text"/> %	
Y Translation: <input type="text"/> %	
Rotation: <input type="text"/> %	
<b>Update</b>	
Min X <input type="text"/>	Max X <input type="text"/>
Min Y <input type="text"/>	Max Y <input type="text"/>
Min Rot. <input type="text"/>	Max Rot. <input type="text"/>
Rotation Center: X <input type="text"/>	Y <input type="text"/>
<b>Update</b>	<b>Done</b>

For *MinY* and *MaxY*, negative values are below the original location of the state, and positive values are higher on the screen. For both X and Y values, the numbers refer to screen pixels.

The *MinRotation* and *MaxRotation* fields of the continuous state object refer to the number of degrees that the object can be rotated from its originally authored orientation. Positive values are clockwise; negative values are counter-clockwise.

If the continuous object's state is to be rotated, you must also set the center of rotation for that state. These values are to be expressed in window coordinates (where 0,0 is the top left corner of the window).

There are two ways to set range boundaries such as MinX and MaxX. You can click in the box to the right of a range boundary label (such as 'MaxX') and enter a number or edit one that is already displayed there. Alternatively, you can click on the label itself (e.g., on 'MaxX') and then position the continuous state at the point that represents the labeled range bound. When you click the left mouse button (signifying acceptance of the displayed state as the named boundary), the numerical value will automatically be filled into the box by the label.

When you want to see the visual effects of the continuous state boundary changes you make, click on the button labeled *Update* in the lower window. The state appearance of the selected object will be updated. When you have finished making all the changes you want in a continuous state, clicking on the *Done* button will exit the Continuous State Adjustment mode.

Rules that control continuous state objects refer to their translations as a *percentage of the range of translation*. Rotation numbers should be interpreted as *percentages of the range of rotation*.

You can easily experiment with the appearances of a continuous state object by using the upper window. The horizontal movements of the piston in the actuating cylinder assembly (shown below) can be explored in several ways in the Continuous State Adjustment mode. You can click in the *X Translation Ruler* to reposition the position marker (the small black triangle in the ruler). The marker will move to the location of the mouse in the ruler, and the continuous state graphic will be updated to reflect the new position.

Continuous State Adjustments

X Translation: 63.0 %

Y Translation: %

Rotation: %

Update

Min X: -3      Max X: 29

Min Y:      Max Y:     

Min Rot.:      Max Rot.:     

Rotation Center: X      Y     

Update      Done

File: (DSK) <LISPPFILES>RAPIDSII  
Date Written: 26 Mar-90 16:23:

In the figure above, a click at the 63% point on the X Translation ruler has had the effect of moving the state graphic 63 percent of the way from -3 to 29 on the X axis.

You can also experiment with continuous state appearances by dragging the position marker in the ruler. Hold down the left button while pointing at the position marker and drag it along the ruler. The continuous state graphic will update simultaneously.

A third way to explore a continuous state appearance is to enter a number in the box above the ruler. In this case, you must click on the *Update* button at the bottom of the upper of the two Continuous State Adjustment windows to see the change reflected in the appearance of the object.

You can explore rotating continuous states by using similar techniques. In the figure below, a rotatable hinge mechanism is displayed in a state of rotation that is 63% of the range (from 0 to -20). The signs of the Min and Max Rotation values are important. the maximum rotation of -20 specifies the same rotation position as 340 would. However, 0 to 340 would imply a clockwise rotation, while 0 to -20 prescribes a counter-clockwise rotation.

A continuous state rotation may be more than 360 degrees. For example, if Min Rotation was set a 0 and Max Rotation at 720, then moving the rotation slider bar from 0.0 to 1.0 would cause the state to rotate two full revolutions. Setting Min and Max Rotation by clicking on the label (*Min Rot.* or *Max Rot.*), and then rotating the object to the desired position using the mouse. The object will rotate, tracking the mouse until the left button is clicked. The values for Min and Max Rotation will be continuously updated in the box to the right of the label.

**Continuous State Adjustments**

X Translation:  %

Y Translation:  %

Rotation:  %

**Update**

Min X  Max X

Min Y  Max Y

Min Rot.  Max Rot.

Rotation Center: X  Y

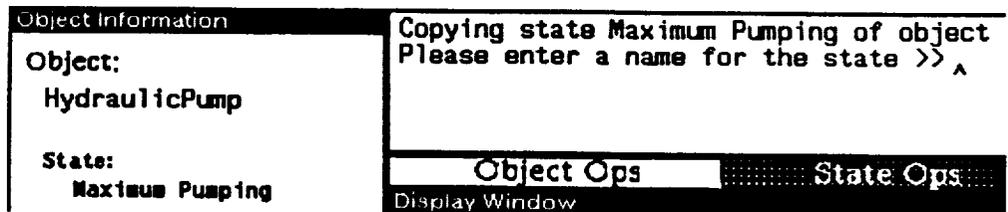
**Update**

File: (DSK) <LISPFILS> RAPIDSII > SIMP  
Date Written: 26-Mar-90 16:23:39

Once you have finished exploring a continuous state, you must click on the *Done* button in the lower window to leave the Continuous State Adjustment mode.

**Copy**

Often the difference in appearance between two states is quite simple. Sometimes, for example, one part of an object shifts position in different states. This variable part can be drawn in one position for the first state. To make the next state, use *Copy* to duplicate the appearance of the first state's graphics, and then use *Move* or *Rotate* to put them in a new position.



After you click on *Copy*, you will be prompted to type the new state name. The new state is the selected state, so you can easily modify the copied state.

**Cycle**

The *Cycle* option is used to step through the states of an object. After the last state for an object, *Cycle* will bring up the first state. (You can create a state with no graphics by not using the *State Graphics* command while you are working on a state, or by deleting all the graphical primitives in a state.)

**Delete**

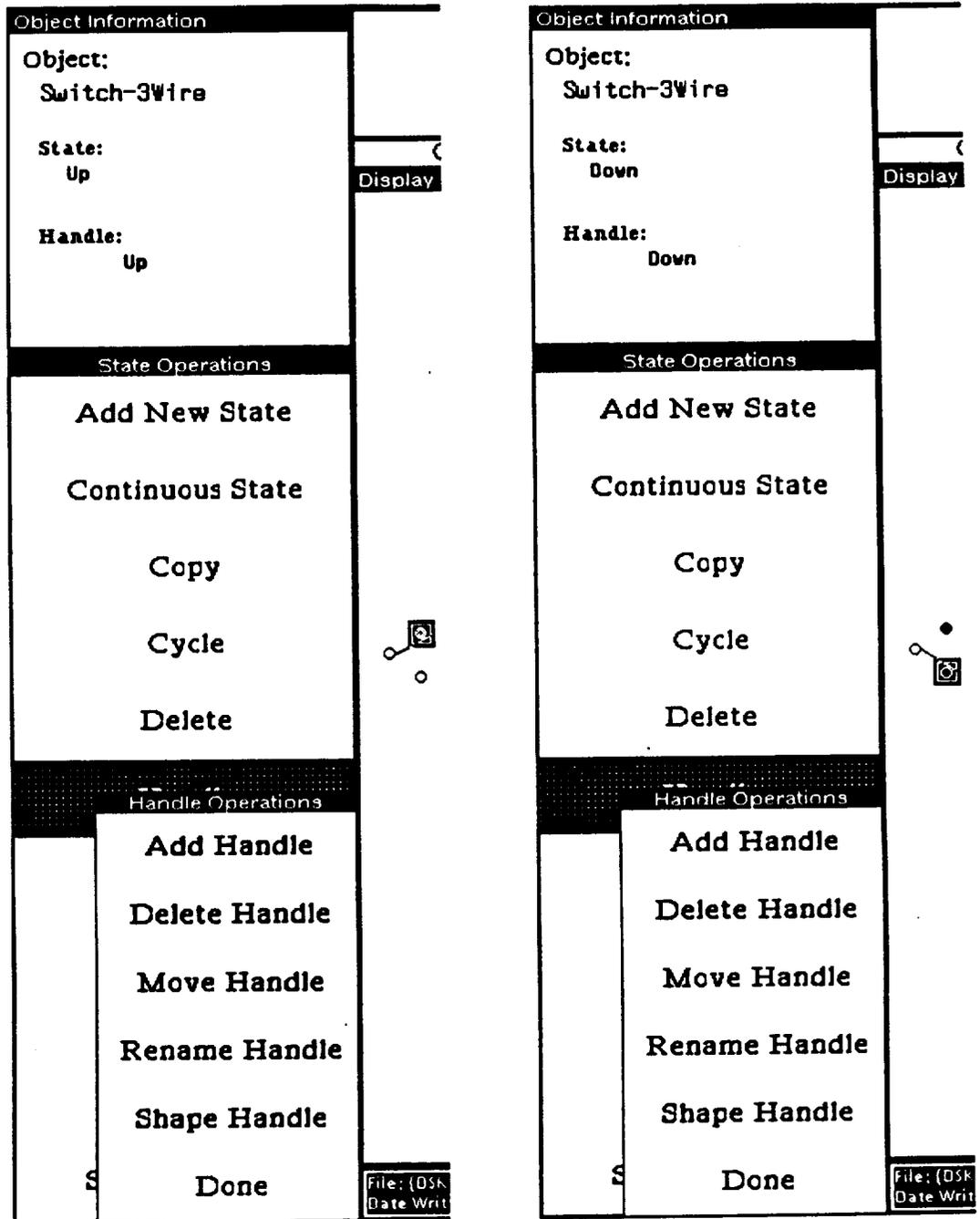
This menu command deletes the selected state. It requires that you confirm (by clicking the left mouse button) that you want to delete the state. The next state becomes the selected (and depicted) state

**Handle**

The term *handle* refers to a designated graphical area (associated with an object or an object state) that is sensitive to mouse clicks. Any object that you would like students to be able to manipulate directly with the mouse must be given one or more handles. Objects with a fixed number of static alternative states can have a single handle associated with each of those states. Such objects are treated specially by the RAPIDS-II simulation driver. Using the *state handle* feature, you can easily build working switches and other controls without writing rules to handle the mouse actions that manipulate the state of the object. In effect, these rules are hard-wired into your simulation. (They cannot be edited using the RAPIDS-II rule editor.) If you want to write a rule that refers to the action of changing a control, the rule should refer to the object's state, rather than to a *Mouse Down in Handle* action.

State handles will put objects directly into the states associated with the handle when the simulation is running and a student clicks on the handle. Authoring such state handles is closely linked with the states. In the two figures below, we see *State Handle Operations* being applied to two states of a switch. The predefined handle shape is used to establish an *Up* handle that corresponds to the *Up* state of the switch in the figure at the left. A predefined handle shape is also used to identify the *Down* handle that is associated with the *Down* state of the switch. It is a good idea to give handles the same names as their corresponding states for alternative state objects.

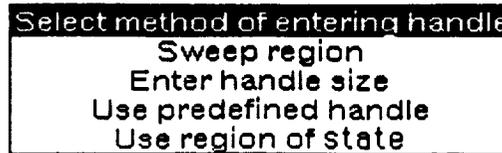
These handles were created by the following sequence of steps. First, the switch object was cycled (using *Cycle*) to its *Up* state. Then, the *Handle* command was invoked. *Add Handle* was used to create a new handle and to give it the name *Up*. The predefined shape was chosen and was positioned on the upper contact of the switch. Then *Done* was used to exit from the Handle Operations mode. The switch was cycled to the next state, *Down*, and the process was repeated to add the *Down* state handle.



**Using Handle Operations**

If you click on *Add Handle*, the message window will prompt you for the name of the handle that you want to add. Type in an appropriate name for the handle and type the Return key.

After you have entered the name of the object, a menu will appear that asks what kind of handle shape you want to use:

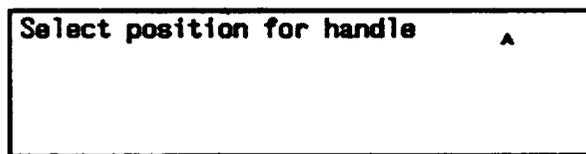


There are four different handle shape options. Whichever method you use to create a handle, make sure that the handle is within the graphics of the object. RAPIDS-II is not able to detect handle manipulations outside of the rectangular bounding box that encloses all the graphics of the object. In any case, it is always a good idea to graphically indicate the 'mouse-active' areas of an object, so that it will be clear to students how they can manipulate them.

If you select the first handle shape option, *Sweep region*, the mouse shape will change to the standard Expanding Box cursor. Drag out a rectangular region that will serve as the named handle. Any student click within this area will be considered a handle manipulation.

The second way of designating a handle is to *Enter handle size*. If you choose this method, you will be prompted to enter the width and height of the handle region.

For objects with discrete states, it is often appropriate to use the *predefined handle* type when creating *state* handles. When you are creating a predefined handle, the cursor will change into the shape of one of these handles, a small black box, and the message window will prompt you to place the handle at the appropriate place on the object.



Just click the left button of the mouse where you want the handle to be placed. Don't be concerned about the appearance of a black square at that point. This visual representation of the handle appears only when you are in the generic editor's handle-editing mode.

The fourth shape options for handles is the easiest one to use. The choice *Use region of state* will simply treat the entire bounding region of the state as a handle region. That region will be highlighted, just as the handle regions you can create using any of the other methods are highlighted during the Handle Operations mode.

*Delete Handle* will ask that you confirm the deletion of the handle by clicking the left mouse button. If you don't want to carry out the deletion, click the right button. *Move Handle* will let you move the handle using the mouse. Click the left button when it is positioned where you want it. *Rename Handle* will bring up a prompt for a new name in the message window. The *Shape Handle* command will pop up the menu of handle shape options discussed above, so you can use any of the standard handle authoring methods to edit the handle's shape. Finally, clicking on the *Done* command in the *Handle Operations* menu will take you out of the Handle Operation mode, back to the State Operations mode.

**Move**

This command attaches the mouse pointer to the state graphics. When you move the mouse, the state appearance moves. To set the new position, click the left button.

<b>Object Information</b>	Move object state to desired location.
<b>Object:</b> Switch-3Wire	
<b>State:</b> Down	
	<input type="button" value="Object Ops"/> <input type="button" value="State Ops"/>
	Display Window

**Rename**

After choosing the *Rename* menu item, you will be prompted to type the new state name. Press the Return key at the end of the name.

<b>Object Information</b>	Please enter a name for the state >> Down
<b>Object:</b> Switch-3Wire	
<b>State:</b> Down	
	<input type="button" value="Object Ops"/> <input type="button" value="State Ops"/>
	Display Window

**Rotate**

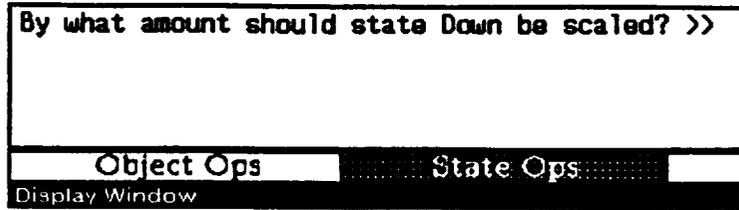
The *Rotate* option behaves just as it does in the object operations mode. You will be asked how much the state should be rotated.

<b>Object Information</b>	By what amount should state Down be rotated? >
<b>Object:</b> Switch-3Wire	
<b>State:</b> Down	
	<input type="button" value="Object Ops"/> <input type="button" value="State Ops"/>
	Display Window

After you enter the number of degrees of rotation you want, you will be prompted to pick the center of rotation. Clicking the right button makes the state rotate about its own center. If you click the left button, the mouse pointer turns into a crosshair in a circle. Put the center of this pointer shape over the desired center of rotation and click the left button. The state will then be shown in its new orientation.

**Scale**

Like *Rotate*, the *Scale* option in the state operations mode behaves just as it does in the object operations mode. Naturally, it affects only the selected state, rather than the entire object.

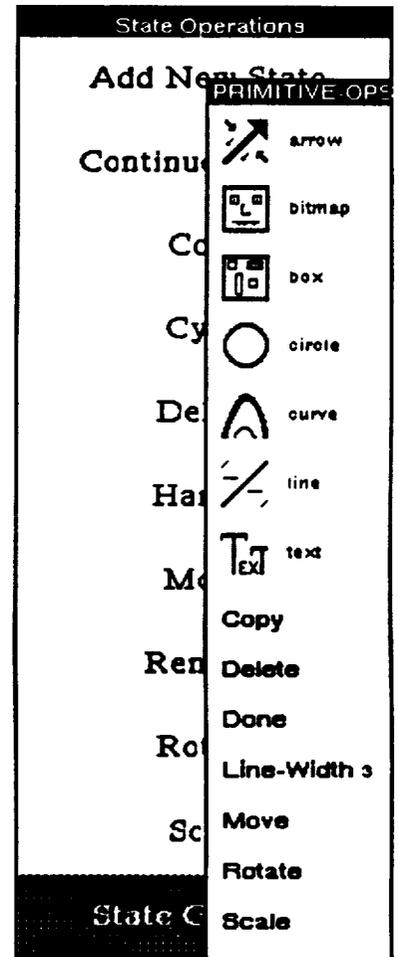


Because of rounding anomalies, it is almost always preferable to draw objects and their states in the size that will be required in the simulations that they will be used in, rather than to scale them.

**State Graphics**

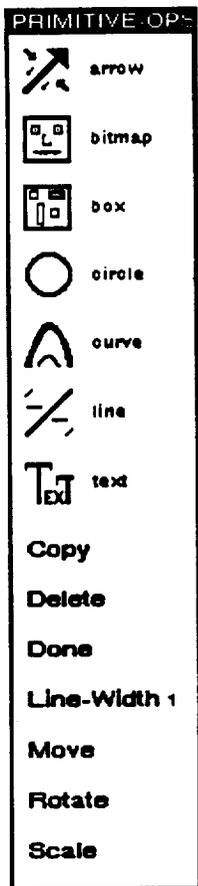
To build or modify the appearance of a state, you usually have to draw some part of it. The *State Graphics* command takes you to the Primitive Operations mode. The *Primitive-Ops* menu is described below in the section labeled *Drawing Operations*. In this mode, you can add, delete, copy, move, rotate, and scale the primitive graphic elements that comprise a state appearance.

This is the same graphic primitives menu that is brought up when you choose the *Object Graphics* command in Object Operations mode. The next section of this chapter deals with these drawing tools.



## Drawing Operations

### The Primitive-Ops Menu



The *Primitive-Ops* menu provides the primitive operations for controlling the appearance of graphical objects in RAPIDS-II. There are two ways to get to this menu. To draw or modify the static part of an object, you select *Object Graphics* from the *Object Operations* menu. To do the same kinds of things to state parts, choose *State Graphics* from the *State Operations* menu.

The appearances of RAPIDS-II consist of a number of simple or 'primitive' graphical elements, such as lines, circles, boxes, curves, and arrows. Unlike some 'paint' programs, RAPIDS-II remembers which of these primitive elements comprise each appearance. It is therefore possible to edit individual primitives in existing graphics. This is what makes it possible to delete, scale, rotate, move, or copy a individual graphic element (such as a line or a circle).

The first seven menu items are primitive drawing tools. Most of them require that you click the left mouse button twice in the main window drawing area: once to start the graphic element, and once to end it. When you are done drawing the element, click on another choice in the *Primitive-Ops* menu. Naturally, you can click on the same tool if you want to draw another primitive of the same type. (The *line* tool works slightly differently from the others, in that it permits the drawing of multiple connected line segments, as is explained below.)

#### Arrow

When you click on this option, it is grayed out on the menu to show that it is the tool currently in use. You then create the arrow's *anchor* by clicking at the point where you want the tail of the arrow. As you move the mouse pointer to the location where you want the head of the arrow, a line is 'rubberbanded' between the anchor point and the mouse pointer. When this rubberbanded line is lined up just as you want the arrow to be, click the mouse button again. A completed arrow then appears on the screen.

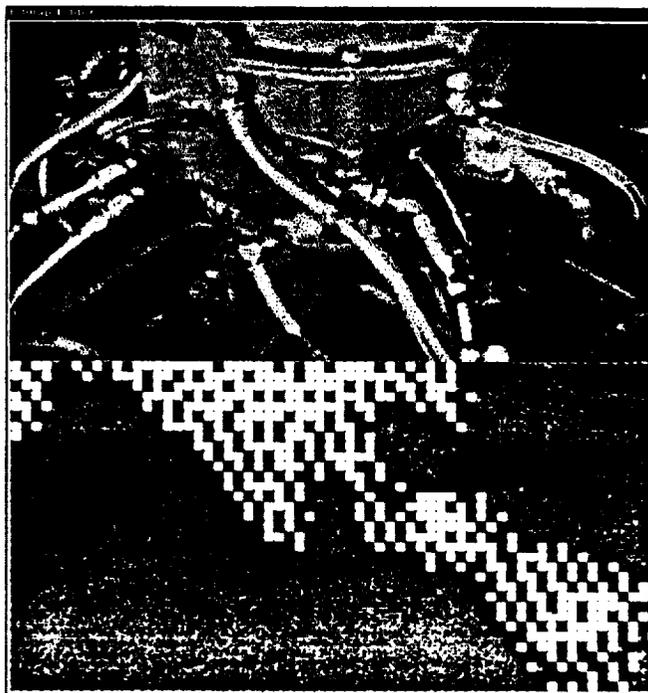
#### Bitmap

This menu item pops up a set of these menu choices:  
*Edit Bitmap Primitive*  
*Get Bitmap From File*  
*Create New Bitmap*

The second option, *Get Bitmap From File*, lets you type in the name of a file that contains a bitmap that you would like to use as a graphic primitive in your simulation. After you give the file name, there will be a short delay and then the bitmap will appear in the display window.

The first and third options — *Edit Bitmap Primitive* and *Create New Bitmap* — invoke the Interlisp-D bitmap editing function `EDIT.BITMAP` to let you create or edit bitmap graphical elements. `EDIT.BITMAP` gives you a menu of bit map manipulation commands. These include commands for shifting, rotating, inverting, and hand editing bit maps. The hand editor gives you an expanded view of the bitmap, making it easy to click bits on (black) with the left mouse button and off (white) with the middle button. To learn more about the features of the standard `EDIT.BITMAP` tool, read its description in the *Envos Lisp Library Packages Manual*.

ORIGINAL PAGE IS  
OF POOR QUALITY



*Hand Editing a Bitmap*

The appearance of a generic object can consist simply of one or more bitmaps, or such an object can contain bitmap elements in addition to other graphical elements.

You can copy parts of the screen to bitmaps, making it possible to rough out a graphic using the object tools, and then do the detailed work using the bitmap editor. Be warned, however, that extensive use of bitmaps may slow down your simulations significantly. Bitmaps usually require a great deal more memory and storage space than do roughly equivalent object-oriented drawings.

#### **Box**

The box tool is used to draw rectangles. After clicking on the *Box* choice, put the mouse pointer where you want one corner of the box. Click the left button. Then put the pointer where you want the opposite corner. As you move the mouse to this point, you will see a 'rubberbanded' rectangle drawn on the screen. Click the left mouse button when you have dragged out a rectangle of the desired size and shape.

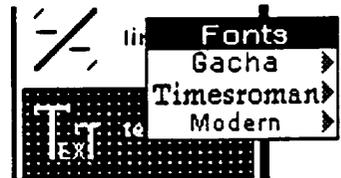
**Circle** To draw a circle after clicking on this menu choice, just click the mouse at the point that is to be the center of the circle. Move the mouse pointer to any point on the circle's circumference. (Again, a circle will be 'rubberbanded' as you move the mouse.) Click when the circle is the size you want.

**Curve** After clicking on the curve option, put the mouse pointer where you want the curve to begin. Then click once where you want the middle of the curve and once at the end of the curve. Three points are required to specify the curve. Finally, click the left mouse button to confirm that this is the curve you want. It doesn't matter where the mouse is pointing for this confirming click.

If you don't like the curve, instead of clicking the left button to confirm, click the middle button. This will remove the last point of the curve. You can then try again, using the left button to set the last point, as before. If you're not happy with the positions of the first two points either, you can click the middle button twice at confirmation time to remove all three points. Your next left button click will set the first point of the curve again.

**Line** This tool lets you draw a series of connected line segments. Each successive segment begins where the previous segment ended. Click *Line*, then click where you want one end of the line, drag a rubberbanded line out by moving the mouse, and click where you want the line segment to end. When you are finished making connected line segments, drag the mouse outside the display window and click. The last 'rubberbanded' line segment (which extends to the edge of the display window) will simply disappear.

**Text** When you choose the text option, you will be given a choice of fonts in a popup menu. Select the font you want by clicking the font name. To choose font size, hold down the mouse button on the font name you want and drag the pointer off to the right. A secondary menu of the sizes available for that font will pop up. Drag into the size you want and release the mouse button. (Or, if you want to choose a bold style, drag to the right of the size option to bring up a menu of style options. Drag the pointer into the one you want and release the mouse button.) You will then be prompted to type in the text you want displayed. End by typing the Return key. Your text will then appear on the screen. Put it into position using the mouse. Click the left button to drop the string where you want it.



If a generic object is to be rotated or scaled, the text should be added after these operations. Text can only be displayed horizontally and vertically (up-reading or downreading).

The first seven options of the primitive operations menu let you choose among the seven primitive graphic elements of RAPIDS-II generic objects: arrows, bitmaps, boxes, circles, curves, lines, and text strings. The appearance of every graphical object consists of one or more of these graphic primitives. Most of the other menu options let you manipulate these primitives in various ways.

- Copy** You can create a copy of any primitive graphic element. After clicking on *Copy*, just click on the primitive element you want to copy. The copy will appear, slightly offset. Drag it to the position you want it to have and click the left button again to position it.
- Delete** Any graphic primitive can be removed from an appearance. After choosing the *Delete* option, click on the element (line, circle, text, etc.) that you want to remove.
- Done** This option ends the primitive operations mode and returns to the menu that called it. If you were working on a static appearance, you will be returned to the *Object Operations* menu. If you were working on a state appearance, you will go to the *State Operations* menu.
- Line-Width** You can change the width of future lines with this option. The current default line width is displayed beside the *Line-Width* menu item. The graphic primitives (except for Bitmap and Text) use the current line width setting to determine how thick their lines will be.
- Move** To move a primitive element, click on this menu option, then select the graphic primitive to be moved by clicking on it. Use the mouse to move the primitive to the location you want, and then click the left button again to drop it there.
- Rotate** The Rotate feature works a little differently for different kinds of primitives. Arrows, boxes, curves, and lines can all be rotated an arbitrary number of degrees. (But notice that rotating a circle about its center is meaningless.) If, after choosing the *Rotate* option, you click on one of these elements, then you will be prompted to type the number of degrees to rotate. Positive values represent counterclockwise rotation. When you rotate one of these primitives, you will be asked whether you want to specify the center of rotation. If you click the right mouse button, the center of the selected primitive will be used as the center of rotation.
- For text and bitmap primitives, you won't be asked to type in the amount to rotate. Instead, you will be given a menu of the allowable rotation values, which are in 90-degree increments.
- Scale** The interface for scaling is similar to that for rotating. You will be prompted to enter a number that will serve as the scaling factor.

---



---

## Window Operations

<b>Window Ops</b>
Background
Bury
Grid
Grid-On/Off
Hardcopy
Move
PreviousPage
NextPage
CreatePage
Redisplay
Shrink

Window operations are used to control global features of the display window used in the generic editor. To bring up the menu of window operations, move the mouse so that the cursor is in the display window, and press and hold down the right mouse button. A *window operations menu* (titled *Window Ops*) will appear. Select a menu item by moving the cursor to the desired item and releasing the mouse button.

**Background**

The background of an object is all the other objects in the library that is currently being edited with the generic object editor. You may want to turn on the background to make certain that the size of an object is proportional to the other objects that it will appear with. At other times, you'll find that it is less distracting to turn off the background so that you can concentrate on the selected object.

**Bury**

This command puts the generic object editor windows **behind** any other windows on the screen. The other windows will then overlay the generic editor windows. To bring the generic editor back into the foreground, simply click the mouse once in one of its windows.

**Grid**

You can change the display window's grid size with *Grid*. The grid is a coordinate system that overlays the display window. A grid is specified in terms of the number of pixels between grid points. All graphic elements can be automatically aligned to the nearest grid point when they are drawn. It is often useful to draw objects that will appear together with the same specified grid, so that their ports will match up correctly when they are positioned next to each other in a scene.

When you choose the *Grid* menu option, the message window will ask you to enter the new grid size. Type a number and the Enter key. A grid finer than 3 cannot be displayed on the screen.

**Grid On/Off**

The grid you specify is always in effect. (If you want to be able to draw to any pixel in the window, you must specify a grid size of 1.) You can choose whether or not the grid points are visible or not using the *Grid On/Off* menu command. Clicking on this option toggles the visibility of the grid.

The first time that the generic editor is opened in your environment the grid size will be 1. The grid size you select will be preserved when you change libraries and across editing sessions. Grid size is tied to the generic editor itself rather than to object libraries.

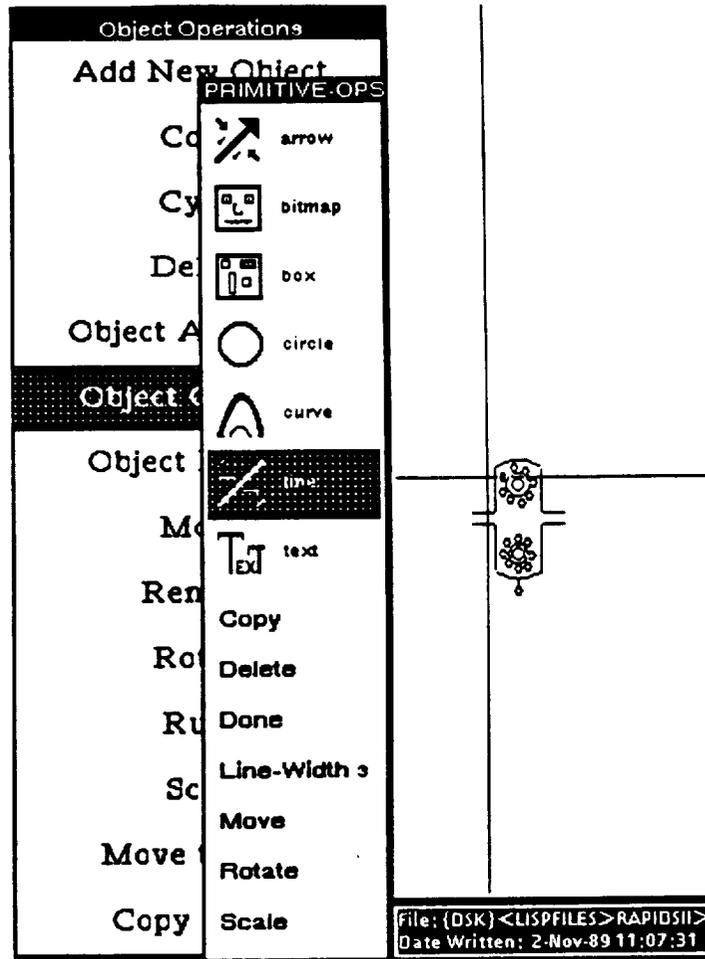
- Hardcopy** This command prints the display window to an attached printer. Naturally, this command works only if someone has already installed the printer drivers appropriate for your printer in your Lisp environment.
- Move** You can use this command to move all the windows of the generic editor, as an attached group, to a new position on the screen. This option is rarely used.
- Previous Page** Libraries can have several pages of objects, where each page is a group of objects to be shown in the display window. The *Previous Page* command brings up the page before the currently displayed page of objects. If the currently displayed page is the first one, *Previous Page* will print a message to that effect.
- Next Page** This command brings up the next page in the library. If the current page is the last page, then the a message will tell you so.
- Create Page** Use *Create Page* to make a new page in the library, which will be placed after the last page. The new page, which will be blank, of course, will then be shown in the display window.
- Redisplay** On rare occasions, you may find that spurious graphic elements are displayed that don't seem to really be there. (For example, they can't be selected or deleted.) The *Redisplay* option repaints the display window, eliminating any such graphical anomalies. You may never have to use this command.
- Shrink** This command suspends the generic editor and shrinks its windows to an icon that represents the generic editor. The name of the library being edited appears in small letters near the top of the icon.  You can later resume the same editing session by opening up the icon. The icon can be opened either by choosing the *Expand* option from the right button menu in the icon, or by clicking the middle button in the icon.
- Graphic Utilities** The Graphic Utilities consist of three functions (*Crosshairs*, *ChangeGridsize*, *DisplayGrid*) that can be executed while you are performing a graphic operation. For example, if you are moving an object or primitive and decide that the grid should be changed, select *ChangeGridsize* by hitting the "G" key on the keyboard.

Here are descriptions of the Graphic Utilities functions:

*Crosshairs* (*C* or *c* on the keyboard) — Toggles the display of large crosshairs. This option is very useful for lining up elements that are not very close to each other. See the screen snapshot with crosshairs below.

*ChangeGridSize* (*G* or *g* on the keyboard) — Changes the grid size. This command is the same as the *Grid* command in the window operations menu.

*DisplayGrid* (*D* or *d* on the keyboard) — Toggles the display of the grid.



The Generic Editor with Crosshairs Turned On

**Leaving the Generic Editor**

There are two ways to leave the generic editor. You can suspend an editing session by choosing the *Shrink* item on the display window's right button menu (the *Window Ops* menu). If you suspend an editing session, the additions and changes that you have made will not be saved to the disk file that stores the library of objects. If something were to happen to corrupt your Interlisp-D environment before you resumed the session and saved, those additions and changes would be lost.

The other way to leave the generic editor is to use the *Done* command on the *Object-Ops* menu. This command will actual end the session. You'll be prompted to decide whether you want the additions and changes made in the session to be saved or not.

# Rule Authoring

---

## Rules in Rapids II

*Rules* describe and control the behavior of objects in *RAPIDS II*. Rules can be either generic (universal for objects of a given type) or specific to a particular simulation. Rules for generic objects are created and edited in the generic editor. Authors can create and edit rules at the scene level as well as at the generic object level. These rules are edited in the scene editor. The propagation of effects in a simulation is determined, in part, by rules created in the scene editor.

Rule editing in *RAPIDS II* is facilitated by a powerful menu-driven editing system. In addition, the Envos Interlisp structure editor has been made available for rule editing, for the use of authors who are comfortable with that approach to editing structures.

### Attributes

*Attributes* are data structures associated with objects. Rules can refer to attribute values. Attributes can include values such as voltages, fluid pressures, and mechanical forces. Behavior rules in generic objects typically manipulate such values. In many cases, a generic object rule transforms the value of some input attribute to compute the value of an output attribute. In addition, however, authors can make other uses of the *attribute* mechanism.

Certain attributes are created automatically in *RAPIDS II*. These include an object's location and its current state. Rules can refer to these standard system attributes, as well as to attributes created by authors.

**Processes**

In *RAPIDS II*, student users can manipulate the simulation while it is active. This feature of *RAPIDS II* makes it possible to write rules that set up ongoing processes, such as incrementing or decrementing attribute values. The appearances of objects can be made to reflect these changing values, so that a simulation appears to be continuously animated. A simulation user can carry out a series of interactions without waiting for all the effects of one action to settle before carrying out the next one.

The combination of the new *process* and *continuous appearance* features make it possible to create real-time task simulations in *RAPIDS II*, greatly extending the training domains that can appropriately be approached with this tool.

**Internal and  
External Rules**

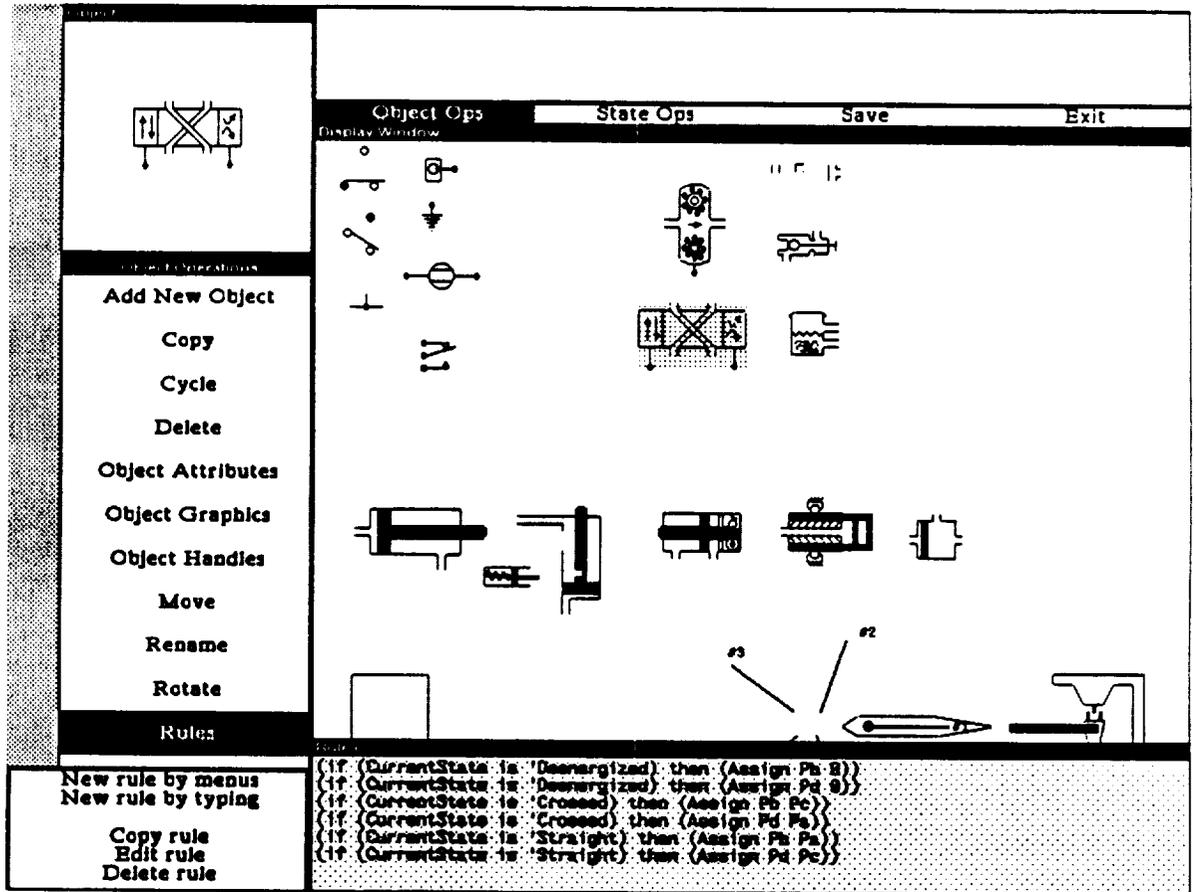
Essentially the same rule editor is used to build all *RAPIDS II* rules. Rules that are associated with *generic* objects are created when the rule editor is invoked from within the generic editor. Rules that are associated with *specific* objects (and, therefore, with particular simulations) are created when the rule editor is invoked from the scene editor. The former type of rules (those associated with generic objects) are called *internal rules*. Internal rules cannot contain references to specific objects. They can only refer to attributes of the generic object itself. Internal rules are described in the next section.

Rules associated with specific objects are called *external rules*. They refer to attributes of one or more specific objects. External rules often perform the job of passing values from one object to another. External rules are created and edited when the rule editor is called from the scene editor. This process is described later in this chapter.

The rule editor behaves in largely the same way in the two environments. When it is invoked from the generic editor, the generic editor disappears from the screen while the rule editing is taking place. When the rule editor is invoked from the scene editor, the scene editor windows do not disappear. They remain present because the author may have to point to an object on a scene while composing a rule.

## Internal Rules

When you click on *Rules* in the *Object Operations* menu, the generic editor windows close and a new set of windows opens for editing the rules of the currently selected object. These windows are shown below.



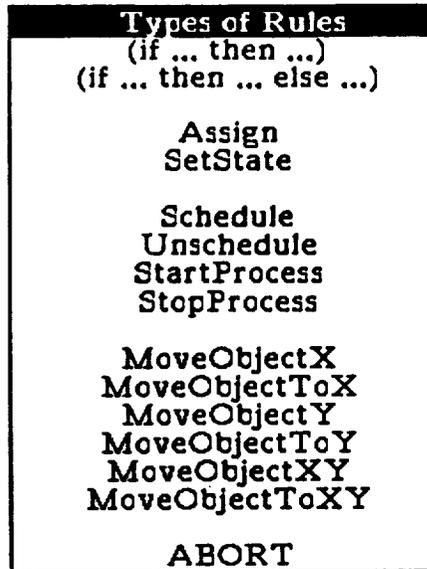
The window at the top left displays the appearance of the object. (In this figure, the generic object is a hydraulic valve.) The bottom window lists the rules that have already been defined for the object. The menu that at first appears at the lower left provides global functions, such as adding a new rule; copying, editing, or deleting an existing rule; and leaving the RAPIDS-II rule editor, returning to the standard generic editor interface.

### Rule Authoring by Menu

The rule editor makes it very easy to create syntactically correct rules by using a set of menus to compose the rules. The sequence of menus permits only legal rules to be composed in this way. The first example demonstrates the authoring of a rule that determines the visual appearance of an object by setting the object to one of its pre-defined visual states.

As soon as you choose *New rule by menus* from the top level of the rule editor (shown above), then a menu for the types of RAPIDS-II rules appears. The choice that you make here determines which menu will be presented next.

The last choice on the menu, *ABORT*, lets you change your mind, and abandon the process of creating a new rule. The first two choices are used to create complex rules that are *if-constructs* at the top level. The other choices are used to create rules that perform a straightforward action, such as assign some attribute a value, set an object to a certain state, and so on.



Let's build a rule of the *if...then...* type. In English, this rule is to say:

*If V-left is greater than zero, then set the state of the valve to Straight, else set the state to Crossed,*

where *V-left* is an arbitrarily chosen name for an attribute of the valve, the voltage it receives at its left electrical connection.

To begin with, click on the second choice in the menu (*if...then...else...*). A new window appears, as shown in the figure below. It shows the text of the new rule, as it has been authored to this point. The text portions between angled brackets (< >) are those parts of the rule that remain to be specified. This window is actually a sophisticated text editor, called *SEdit*, that is part of the Interlisp-D system. You can build rules without using *SEdit*, but it is often useful to be able to edit rules using this editor. To learn about its operation, read the *Envos* manual that describes *SEdit*.

The Generic Editor message window displays authoring instructions during the menu-based rule-authoring process.

	Select an expression to replace <Condition>.
Object Ops State Ops Save Exit	
Add New Object Copy Cycle Delete Object Attributes Types of Conditions (<condition> AND <condition>) (<condition> OR <condition>) (NOT <condition>) is = > >= < <= <> MouseDown MouseDownInHandle MouseDownInObject <attribute-of-this-object> ABORT	(if <Condition> then <Effect1> else <Effect2>) (if {CurrentState is 'Deenergized'} then {Assign Pd B}) {if {CurrentState is 'Deenergized'} then {Assign Pd B}} {if {CurrentState is 'Crossed'} then {Assign Pd Bc}} {if {CurrentState is 'Crossed'} then {Assign Pd Pa}} {if {CurrentState is 'Straight'} then {Assign Pd Pa}} {if {CurrentState is 'Straight'} then {Assign Pd Pc}}

The *Types of Rules* menu that was used to select *(if...then...else...)* disappears as soon as the selection was made. In its place is the *Types of Conditions* menu, which asks you to select a condition for the menu. Boolean combinations can be selected (in which case the condition menu appears again).

In this example, we want to specify a *greater-than* condition, so > should be selected from the *Types of Conditions* menu. (As menu selections are made, the rule text displayed in the SEdit window will be updated to reflect the more fully fleshed-out form of the rule.)

Numeric components
SIN
COS
TAN
LOG
ANTILOG
SQRT
ABS
+
-
x
/
POWER
MODULO
RandomNumber
MAX
MIN
XPositionToPercent
YPositionToPercent
PercentToXPosition
PercentToYPosition
MouseX
MouseY
Clock
<number>
<attribute-of-this-object>
ABORT

Since > requires numeric arguments, the *Numeric components* menu appears. The condition being written is that the value of an attribute of the valve is greater than zero, so <attribute-of-this-object> is selected.

If the valve contained any user-specified numeric attributes, a menu of those attributes would appear. Since there are none yet in this example, you are prompted in the message window to type a name for a new attribute.

Messages
Select an expression to replace <Condition>.
Select an expression to replace <numeric1>.
What is the name of this new attribute? >> V-left

You are also asked to specify whether this numeric attribute is of type *Integer* or *Real*.

Attribute Type
Integer
Real
ABORT

The *Numeric components* menu appears again at this point. To complete the condition, select *<number>*, then in response to the prompt in the message window, type the number *0.0*.

```

Messages
What type of attribute is V-left?
Select an expression to replace <numeric2>.
Type a number.
>>
    
```

Note that as you build a rule by making selections from menus, the textual body of the rule appears and is modified in the SEdit window to the right. At this point, the rule body is

```
(if (V-left > 0.0) then <Effect1> else <Effect2>)
```

Now that the condition (*V-left > 0.0*) has been completely specified, the message will prompt you to specify *<Effect1>*, the part of the rule that will apply if the condition is satisfied. For this, the *Type of Rules* menu appears, indicating the types of rules that can be specified for this object. The possibilities can vary from one object to another, depending on how the object has been defined in the generic editor. For example, this valve has not been defined as being rotatable, so the rule types *Rotate* and *RotateTo* do not appear on this menu.

```

Types of Rules
(if ... then ...)
(if ... then ... else ...)

Assign
SetState

Schedule
Unschedule
StartProcess
StopProcess

MoveObjectX
MoveObjectToX
MoveObjectY
MoveObjectToY
MoveObjectXY
MoveObjectToXY

ABORT
    
```

As the menu suggests, *if...then...* clauses can include other embedded *if...then...* clauses. In this example, however, all that is to be done in the *then*-clause is to set the state of the valve. Choosing the *SetState* option will bring up yet another menu, which asks you for either a state name or an attribute whose value will provide the state name. Choose the state name *'Straight*.

```

'Straight
'Crossed

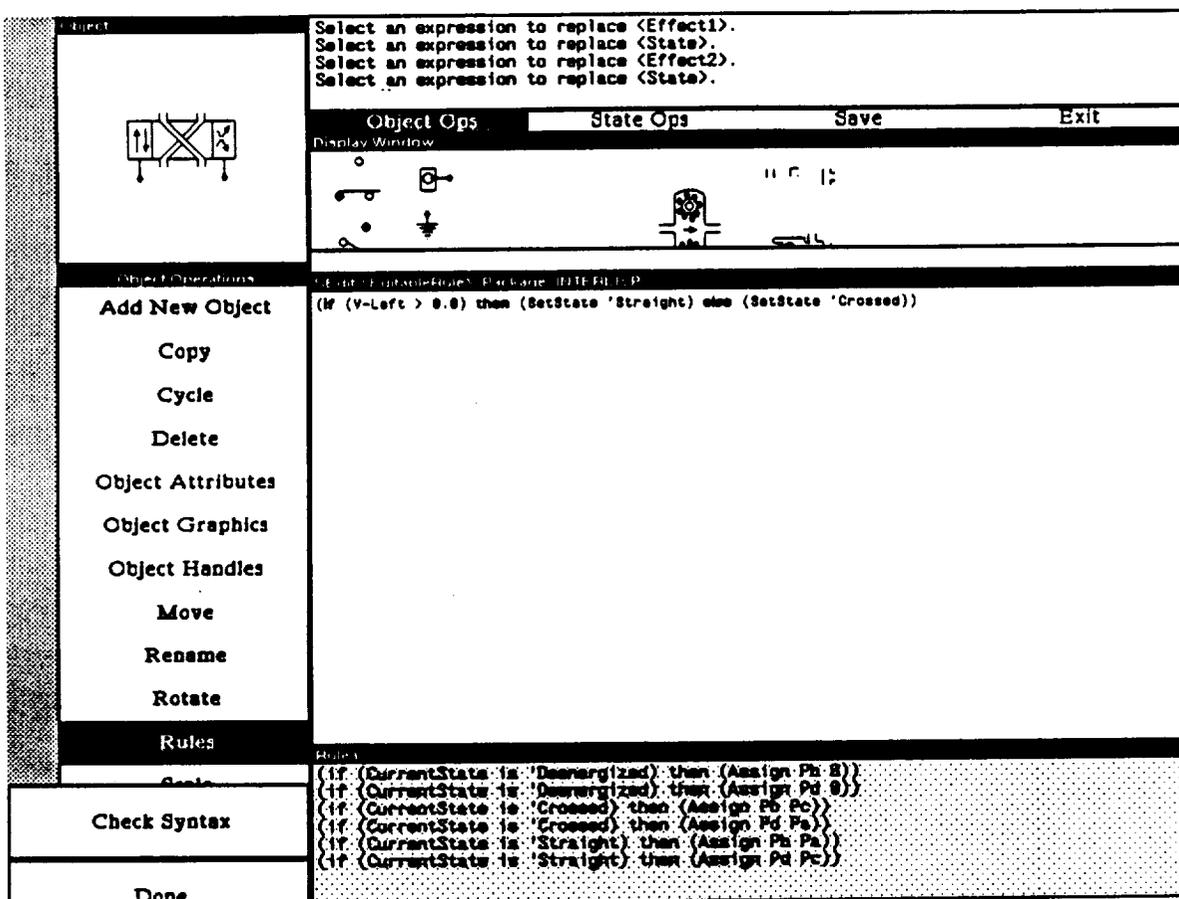
MouseState

<attribute-of-this-object>

ABORT
    
```

The final steps involve specifying *<Effect2>*, the *else* part of the rule. For this a reduced menu of rule possibilities appears, allowing only an embedded *if...then...* clause or another *SetState* effect. Select the latter, then select *'Crossed* to complete the rule.

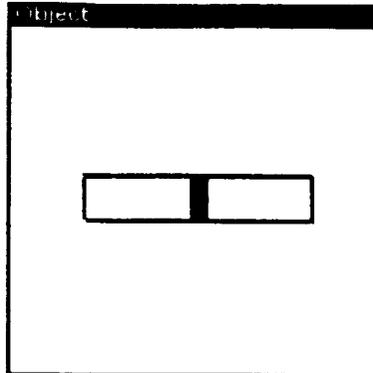
At this point, the entire rule has been composed by using the menu authoring option of the rule editor. The completed rule is displayed in the rule editing window, and is available for text-oriented editing. You can modify the rule by clicking at the point where you want to make a modification, and then backspace and/or type. If you make any editing changes, you should select the *Check Syntax* button (see the figure below). It is not necessary to use *Check Syntax* if you create a rule using the menu-based authoring and don't edit the rule by hand. All rules composed using the menu-based rule authoring system will be syntactically correct.



When you are happy with the rule, click on *Done*, and the editing window will close. The completed rule will be added to the list in the Rules window at the bottom of the display.

**Defining a Continuous Control**

Let's now define a different type of object a variable voltage source, called a *Slider Control*. In this example we will imagine that negative voltages are possible, and that our device is to output voltages between -25 volts and +25 volts. The appearance of the object is:

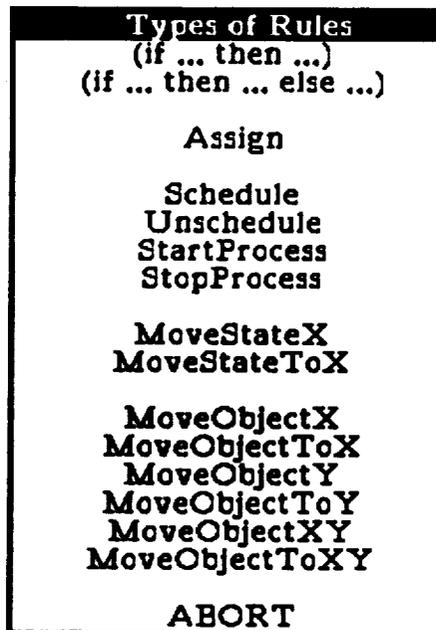


The solid black part in the middle of the Slider Control is a handle, which can be moved left and right by the mouse. Associated with the object is an attribute *StateLocX*, which takes values from 0.0 (when the handle is all the way left) to 1.0 (when the handle is all the way right). The rule we want will *Assign* to a new attribute *OutputVoltage* a value based on the position of the handle. A formula that converts handle position to the desired values is  $(50 \times (\text{StateLocX} - 0.5))$ .

The behavior of this object is that *OutputVoltage* should be set to such a value whenever the handle is manipulated. It is not necessary to use an *if...then* construct in such a rule. The form of the rule should simply be

```
(Assign OutputVoltage (50.0 x (StateLocX - 0.5)))
```

After selecting *Create rule by menus* from the top level menu, the *Types of Rules* menu appears.



The selection to be made from this menu is *Assign*. When it is selected the *Types of Rules* menu disappears, and the message windows prompts to enter the name for a new attribute, which we will call *OutputVoltage*

Next a menu appears for specifying the type of attribute that is being created.

```

Which type of attribute?
Integer
Real
Atom
String
Boolean

ABORT
    
```

From this we select *Real*. This menu disappears and is replaced by the *Numeric components menu*, which we have already seen. From this menu we first select *x*, the multiplication symbol. The same menu reappears, and this time we select *<number>* and type *50.0* in response to the prompt in the message window. Again the *Numeric components menu* appears; now we select *-*, the subtraction symbol. Next, from the same menu, we select *<attribute-of-this-object>*.

```

Numeric components
SIN
COS
TAN
LOG
ANTILOG
SQRT
ABS

+
-
x
/

POWER
MODULO
RandomNumber

MAX
MIN

XPositionToPercent
YPositionToPercent
PercentToXPosition
PercentToYPosition

MouseX
MouseY

Clock

<number>

<attribute-of-this-object>

ABORT
    
```

At last we get a new menu, which lists the existing numeric attributes of the Slider Control and *<new-attribute>*, in case the attribute we need does not yet exist.

```

StateLocX
ObjectLocX
ObjectLocY
OutputVoltage
<new-attribute>
ABORT

```

We select *StateLocX* , and one last time the *Numeric components menu* appears. We select *<number>* and type in *0.5* in response to the prompt. This completes the rule, which is reproduced in its final form in the *Rules* window after *Done* is selected from the final menu.

```

Rules
(if (CurrentState is 'Deenergized') then (Assign Pb B))
(if (CurrentState is 'Deenergized') then (Assign Pd 0))
(if (CurrentState is 'Crossed') then (Assign Pb Pc))
(if (CurrentState is 'Crossed') then (Assign Pd Pa))
(if (CurrentState is 'Straight') then (Assign Pb Pa))
(if (CurrentState is 'Straight') then (Assign Pd Pc))

```

---

## External Rules

We now want to write a rule that changes the state of the Valve we defined earlier, based on the output voltage of the Slider Control. To do this, we need to write an external rule, one that lets one object refer to the attributes of a different object. We will have to build a simple simulation using the scene editor. External rules are created when one enters the rule editor through the scene editor, rather than through the generic editor. Using the rule editor in the scene environment involves two differences, the ability to refer to any object in the scene when defining a rule, and the ability to connect two objects.

The next chapter in the preliminary version of the authoring manual describes how to use the scene editor. For the purposes of this discussion, we will assume that you have already created a simple scene that includes one instance of the valve defined in the generic editor, and one instance of the slider control.

In the generic editor, we defined the valve so that its state depends on whether the voltage coming into it is greater than zero or not, and we defined the Slider Control so that it can output voltages from -25 to + 25. What we want to do now is to add a rule to the Valve's definition, a rule that says that the valve receives its voltage from the Slider Control.

We begin by making sure that the Valve is the selected object in a scene which contains both of our objects. By selecting *Rule* in the scene editor, we get the top level menu.

The screenshot shows a software interface with several components:

- Menu:** Located at the top, with options: View, Object Ops (highlighted), Simulation, Write, and Exit.
- Object Info:** A panel on the left showing details for the selected object 'clock-display'.
 

ObjectLock	Int	0
ObjectLockY	Int	0
minutes	Int	3
hour	Int	0
day	Int	3
display?	Boolean	True
display-time	Int	4323
- VCR Control Panel:** A graphical representation of a VCR with various buttons like EJECT, PLAY, STOP, PAUSE, and a digital display showing '12:03'. It also includes a 'POWER' section with 'MUTE' and 'MAG' indicators.
- Object Operations:** A list of actions available for the object:
 

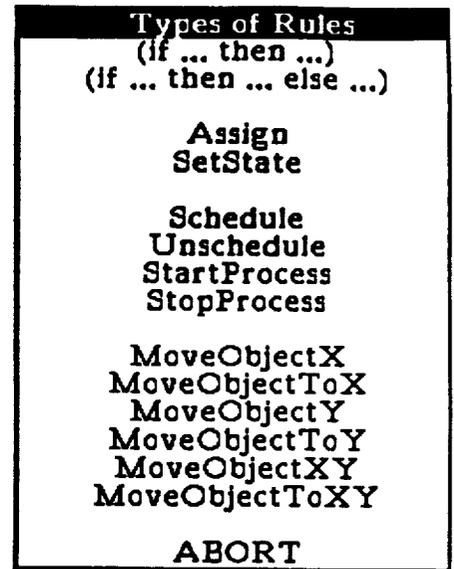
Create	Copy
Delete	Cycle
V-Flip	Label
<b>Make connection</b>	
New rule by menus	
New rule by typing	
Copy rule	
Edit rule	
Delete rule	
Done	
- Rules Editor:** A text area at the bottom containing logic rules:
 

```

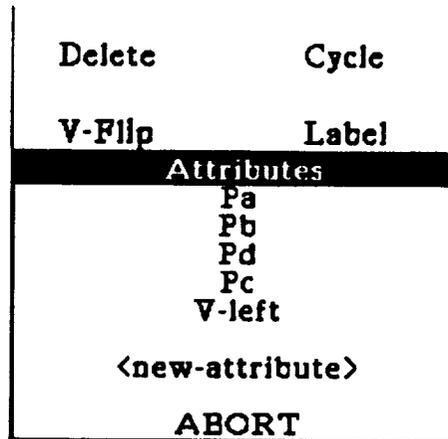
            (Assign flash? True)
            {if ((setting? of setting-clock on VCR-SCENE) is True) then (Assign flash? True) else
            {if ((setting? of setting-clock on VCR-SCENE) is True) then (Assign display-time (time
            (Assign start-time 0)
            (Assign start-reference 0)
            {if ((setting? of setting-clock on VCR-SCENE) is False) then (OneTime (Assign start-t1
            {if ((setting? of setting-clock on VCR-SCENE) is False) then (OneTime (Assign start-re
            (Assign display? I)
            {if {flash? is False} then (Assign display? True))
            
```

The only difference between this menu and the two-level menu in the generic editor environment is the existence of the *Make Connection* option in the menu.

We again select *Create rule by menus*, at which point the *Types of Rules* menu appears.



Selecting *Assign* gets rid of this menu and brings up the menu of attributes of the Valve.



We want to assign a value to *V-left*, so select that attribute. Since *V-left* is defined as type *Real*, our old friend the *Numeric components* menu appears again. This time it is slightly different; it includes the option *<attribute-of-different-object>*, which is what should be selected for this rule.

```

XPositionToPercent
YPositionToPercent
PercentToXPosition
PercentToYPosition

MouseX
MouseY

Clock

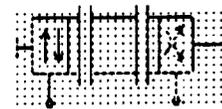
<number>
<attribute-of-this-object>
<attribute-of-different-object>

ABORT
    
```

When that selection is made, all the rule editor windows and menus disappear and the scene editor environment resurfaces. To select an attribute of a different object (that is, other than the valve), you hold down the *Shift* key while moving the mouse into the object you want to select, in this case the Slider Valve. Still holding the *Shift* key, click the left mouse button. This selects the object, and brings up a menu of the relevant (in this case, numeric) attributes of that object.

```

SliderControl0079
StateLocX Type: real Value: 0.505
ObjectLocX Type: int Value: 97
ObjectLocY Type: int Value: 317
OutputVoltage Type: real Value: ??
<new-attribute>
    
```



To complete the rule we select *Output Voltage*. The menu disappears, the rule windows reappear, and the new rule is printed in the *Rules* window, after *Done* is selected.

```

Rules
(Assign V-left (StateLocX of SliderControl0079 on SLIDERANDVALVE))
    
```

In this formula, *V-left* refers to an attribute of the selected object, the Valve. Since *StateLocX* is taken from a non-selected object, it is represented in a more complex way, specifying not only the attribute name, but also the specific object name, *SliderControl0079*, and the name of the scene which contains the Slider Control, *SliderAndValve*.

Given the three rules we have written, we could change the Valve to the Crossed state by moving the handle of the Slider Control to left of center, or to the Straight state by moving it to right of center.

---



---

## Rule Editor Features

**Check Syntax**      The *Check Syntax* button checks whether a selected rule is well-formed. (See the figure below.) All rules constructed using the menus of the rule editor are perform well-formed. This option is used to check the syntactic correctness of rules that have been edited in the *SEdit* window. The message window presents information about the well-formedness of a rule whose syntax has been checked.

The full syntax of RAPIDS-II rules is presented in the last section of this chapter.

**Elementary Rule Actions**      Rules may consist of a number of nested *if*-clauses, but eventually there must be an expression of some elementary action or actions that the rule is to perform.

The most basic (and, in most simulations, the most frequently utilized) kind of elementary action is the *assignment of a value to an attribute*. In the rule syntax, this action is represented by

(*Assign Attribute Value*)

where *Attribute* is the name of some attribute, and *Value* is either a constant or an expression that can be evaluated to produce a value. Attributes have types, such as *integer*, *real*, *atom*, *string* and *boolean*. The value that is assigned to an attribute must be of the same type. (One exception to this rule is that attributes of integer type — that is, whole numbers — can be assigned real values. Similarly, attributes of real type can be assigned integer values.)

Another commonly used rule action is *Set State*. This action is used to set an alternative-state object to one of its states. A spring-loaded positioner, for example, could have a rule that says

(*if (InputPressure > SpringForce) then (SetState Positioner 'Open')*)

where *InputPressure* and *SpringForce* are attributes of the spring-loaded Positioner.

**Graphic Rule Actions**      Many of the elementary actions of the RAPIDS-II rule syntax provide graphical control. These commands are used to move or rotate objects or states. These can be divided into two groups: those that apply to objects as a whole and those that apply to continuous states.

### Object-Level Actions

MoveObjectX  
MoveObjectToX  
MoveObjectY  
MoveObjectToY  
MoveObjectXY  
MoveObjectToXY

### State-Level Actions

MoveStateX  
MoveStateToX  
MoveStateY  
MoveStateToY  
MoveStateXY  
MoveStateToXY  
Rotate  
RotateTo

As the above table shows, only continuous states can be rotated under the control of rules, not objects as a whole. The *Move* commands for objects have parameters that refer to pixels. *Move* commands for continuous states have parameters that should be interpreted in terms of percent of the range of the state.

*Move* commands with *To* in their names are absolute moves, while those without *To* are relative to the current location.

#### Real-Time Rule Actions

Four elementary rule actions support real-time effects in RAPIDS-II. *Schedule* is used to post a future assignment or other elementary action. *Unschedule* can eliminate a scheduled action. *StartProcess* starts a continually updating assignment or other elementary action. *StopProcess* ends such an ongoing action short of its goal.

A *Schedule* action has two arguments: an elementary rule and a delay time for the execution of the rule. For example,

(Schedule (Assign Tank's Volume MaxCapacity) EndFillDelay)

or

(Schedule (SetState 'Exploded) DetonationDelay)

The delay time parameter is relative to the current time, that is, the time at which the schedule rule is carried out.

*Unschedule* takes only one argument, the rule that is to be unscheduled. If more than one scheduling of the rule has taken place, all schedulings are removed by the *Unschedule*. The actual unscheduling takes place as soon as the *Unschedule* is carried out.

A *StartProcess* elementary rule has three arguments: the attribute to be regularly updated, the rate at which to update it (expressed in units per second), and the destination value for the process. The way the simulator handles processes is to update each attribute that has been put on an *ongoing processes* list (by an invocation of *StartProcess*) according to its rate as often as possible. Once a destination value has been attained, that attribute is removed from the process list.

Another way to remove an item from the ongoing processes list is by the invocation of a *StopProcess* rule. *StopProcess* has only one argument, the attribute that is being regularly updated.

---

## Rule Syntax

The syntax of RAPIDS-II rules is presented below. For the most part, the generic editor and the surface editor have the same rule syntax. Exceptions are noted.

```

<rule>
  <if-clause>
  <effect>

<if-clause>
  (if <cond> then <if-clause>|[OneTime]<effect>
    else <if-clause>|[OneTime]<effect>)
  (if <cond> then <if-clause>|[OneTime]<effect>)

<cond>
  (<cond> AND|OR <cond>)
  (NOT <cond>)
  (<atomic> is <atomic>)
  (<boolean> is <boolean>)
  (<string> is <atomic>)
  (<numeric> <comp> <string>)
  <MouseFn>
  <attribute>                                     ;Of type Boolean

<comp>
  =
  <>
  >
  >=
  <
  <=

<MouseFn>
  (MouseDownInHandle <handle>)
  (MouseDownInObject <object>)
  (MouseDown)

<handle>
  (<handle-name> of <object-name> on <scene>) ;External rules only
  <handle-name>

<atomic>
  <attribute>
  <Lisp_atom>
  MouseState

```

```

<Boolean>
  (is <cond>)
  True
  False

<numeric>
  (<f1> <attribute>|<numeric>)
  (<f2> <attribute>|<numeric> <attribute>|<numeric>)
  (<f3> {<attribute>|<numeric>})
  MouseX
  MouseY
  XPositionToPercent
  YPositionToPercent
  PercentToXPosition
  PercentToYPosition
  Clock
  <number>
  <attribute> ;Of Lisp numeric type

<f1>
  sin
  cos
  tan
  log
  antilog
  sqrt
  abs

<f2>
  +
  -
  x
  /
  power
  modulo
  random-number

<f3>
  max
  min

<attribute>
  <attr-name> of <object-name> of <scene-name> ;Ext rules only
  <attr-name>

<string>
  <attribute> ;Of Lisp string type
  <Lisp_string>

```

```

<effect>
  <prim_effect>
    (Schedule <prim_effect> <numeric>)
    (Unschedule <partial_prim_effect>)
    (StartProcess <partial_prim_effect1> <numeric> <numeric>)
    (StopProcess <partial_prim_effect>)

  <prim_effect>
    (<Graphic1> <numeric>)
    (<Graphic2> <numeric> <numeric>)
    (SetState <atomic>)
    (Assign <attribute> <numeric>|<Boolean>|<atomic>|<string>)

  <partial_prim_effect>
    (<Graphic1>)
    (<Graphic2>)
    (SetState <atomic>)
    (Assign <attribute>)

  <Graphic1>
    Rotate
    RotateTo
    MoveStateX
    MoveStateY
    MoveStateToX
    MoveStateToY
    MoveObjectX
    MoveObjectY
    MoveObjectToX
    MoveObjectToY

  <Graphic2>
    MoveStateXY
    MoveStateToXY
    MoveObjectXY
    MoveObjectToXY

```

---

<sup>1</sup> The `prim_effect` of a `StartProcess` or a `StopProcess` cannot be an instance of `SetState`. It would not make sense to try to set up a process of object state changes. Also, only attributes of the Lisp numeric types can be the argument of `'Assign'` in a `partial_prim_effect` argument of `StartProcess` or `StopProcess`.

---

## Developing Simulations

The scene editor is the RAPIDS II authoring tool that is ordinarily used most in building simulations. It is an elaborate tool for composing and testing the scenes of interacting objects that comprise a simulation.

The scene editor can be used to create single-scene or multiple-scene simulations. In order to become familiar with the basic functions of the scene editor, read through the example below, *Building a Simple Simulation*, and then try it yourself. Then study the rest of the chapter, which briefly describes most of the features of the scene editor.

---

### The Role of the Simulation Scene

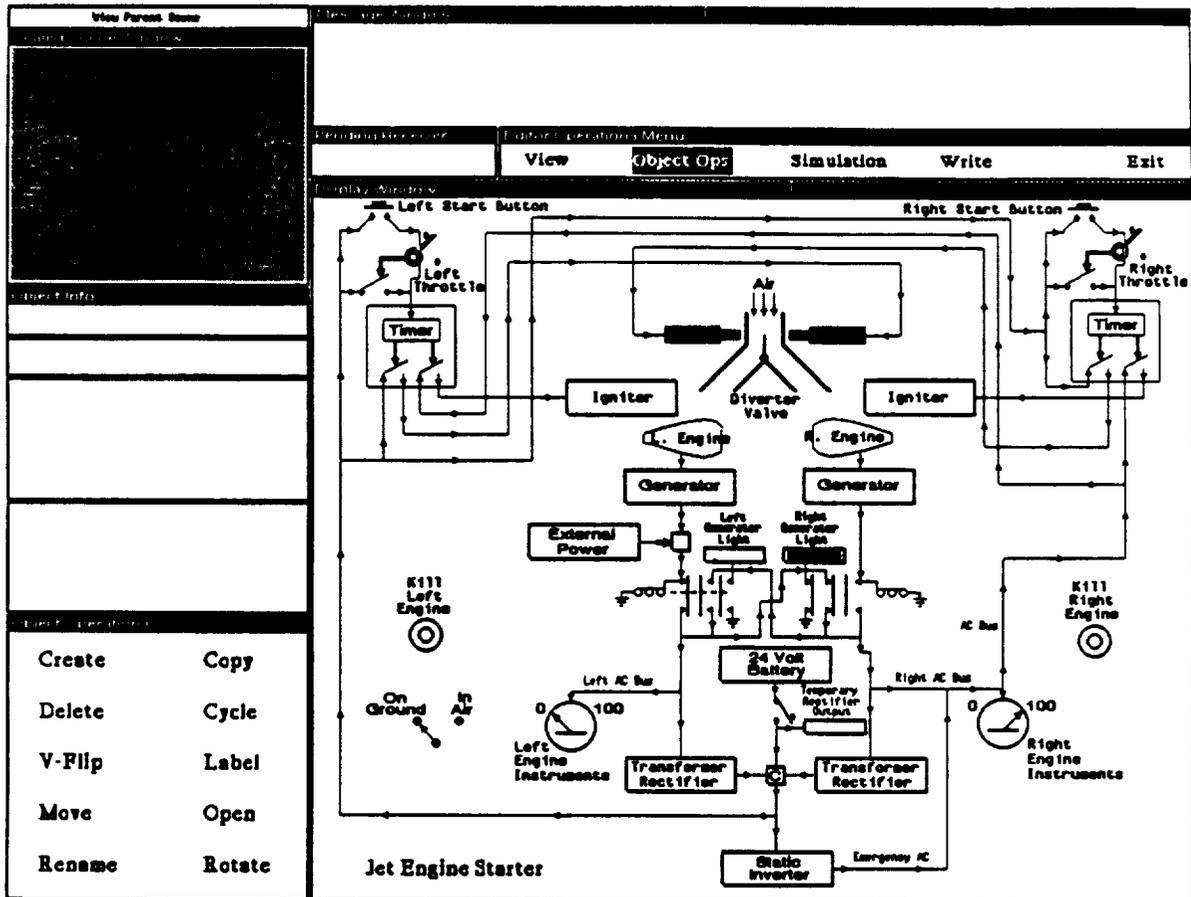
Simulations in RAPIDS II are divided into a number of interacting *scenes*. A scene is a fixed-size graphical view of a portion of a simulation. Simple simulations typically contain only one scene. More complex simulations may have a number of interacting scenes.

The form of the RAPIDS II student interface is influenced by the number of scenes in a simulation. If a simulation has more than one scene, the Options menu of graphical buttons in the upper left corner of the screen will include an item labeled *View*. Clicking on this button (available only in the student environment) brings up a tree of scene names that comprise the simulation. If a simulation has only one scene, the Options menu will not have a button labeled *View*.

When a simulation does have more than one scene, it can be navigated by students in either of two ways. One way of navigating is to use the *View* menu button to bring up a tree of available scenes, and then to click on a node labeled with the name of the destination scene. A second way of navigating is by means of the *scene icons* on a scene. A scene icon is a graphical object that serves as a gateway to another scene. When a student double-clicks on a

scene icon, the scene displayed in the scene window is replaced by the destination scene.

When you are first learning how to use RAPIDS II, you should work on single-scene simulations. Once you have mastered the basics of scene editing, you can progress to examples that include multiple scenes.



The Scene Editor Windows

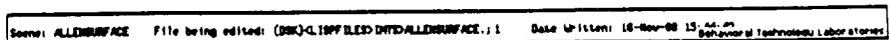
**The Scene Editor Windows**

The scene editor is a powerful and elaborate authoring tool, and it has many windows. As in the generic editor and the RAPIDS II simulation environment, the largest window is the *Display Window*. This window is used to build and display simulation scenes.

Immediately above the display window is the *Editor Operations Menu*, which is used to select global editor operations, such as saving and quitting. This menu is also used to move back and forth among the two major scene editor modes: object operations and simulation. When you begin a scene editor session, you will be in the object operations mode, which allows you to create, name, move, and otherwise modify the components that make up a simulation scene.

Above the editor operations menu is the *Message Window*, which, like the message window of the generic editor, is used to prompt you for data that must be entered with the keyboard, such as the names of objects.

Just below the display window is the *Scene Information Window*, which is not shown in the above figure. This window tells the name of the file containing the scene data that is currently being edited. It also tells when the file was saved. If changes have been made to the file since it was saved, the window will be inverted (white text on a black background).



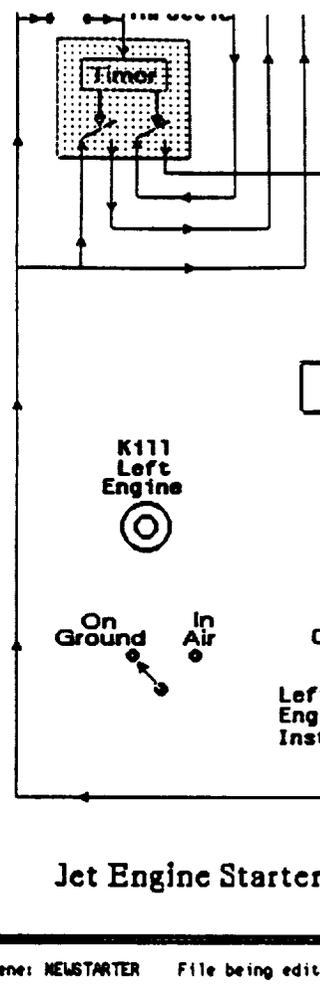
Scene: ALLENSURFACE File being edited: (DIR)-C:\RPP\FILES\DATA\ALLENSURFACE.J 1 Date Written: 16-Nov-88 15:44:07 Behavioral Technology Laboratories

To the left of the display window are a set of windows and menu items appropriate to the current editing mode. At the top of these is the *Scaled Scene Window* (gray in the screen picture on the previous page), which usually shows a miniature version of the scene that is parent to the scene in the display window. If the current scene has no parent, this window is filled with gray. The use of the scaled scene window is described in detail in the chapter section below called *Object Operations*.

Just below the Scaled scene window is an area called *Object Info*, where a variety of information about the currently selected object is displayed.

**Object Info in Object Operations**

Object Info		
<b>Left Timer</b>		
Open	(62 438 6 6)	Open
CurrentState	atom	Open
ObjectLocX	int	40
ObjectLocY	int	415
RightOutput	int	0
RightInput	int	100
TriggerVoltage	int	0
LeftOutput	int	0
(if ... then (Schedule (setState))) *no (if ... then (setState)) *no (if ... then (Assign RightOutput) else ...) *no (if ... then (Assign LeftOutput) else ...) *no (Assign RightInput) *no (Assign LeftInput) *no (Assign TriggerVoltage) *no		
Object Operations		
Create	Copy	
Delete	Cycle	
V-Flip	Label	
Move	Open	
Rename	Rotate	
Scale	Rules	



Scene: NEWSTARTER File being edit

Information about the currently selected object appears in a set of windows just above the object operations menu. From top to bottom, these windows contain:

- The name of the selected object
- A scrollable list of the handles of the object, with corresponding state names
- A scrollable list of the attributes of the object, together with their types and current values
- A scrollable list of the object's behavior rules, in an abbreviated form

The data shown in the object information windows are very useful for understanding the behavior of simulations during the authoring process.

**The Simulation Operations User Interface**

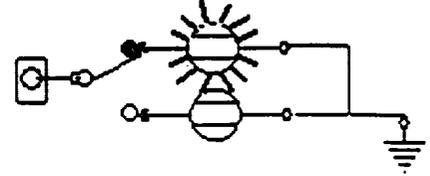
Pause/UnPause		Clock
<b>Paused</b>		40
Current Events		
Simulation Attributes		
Clock	int	40026
MouseX	int	54
MouseY	int	20
MouseState	atom	Up
Object Info		
<b>Left Timer</b>		
Open	(02 438 5 5)	Open
CurrentState	atom	Open
ObjectLocX	int	40
ObjectLocY	int	41.5
RightOutput	int	0
RightInput	int	100
TriggerVoltage	int	0
LeftOutput	int	0
<pre>(if ... then (Schedule (SetState))) *no (if ... then (SetState)) *no (if ... then (Assign RightOutput) else ...) *no (if ... then (Assign LeftOutput) else ...) *no (Assign RightInput) *no (Assign LeftInput) *no (Assign TriggerVoltage) *no</pre>		
Snap	Compile	
Save State	Restore State	
Pause Rules	Pause Attributes	
Trace Attributes	System Trace	

When an author puts the scene editor into its *Simulation Operations* mode, then the Object Info windows are shifted down. Above them appear windows with the current activity state of the simulation (Paused/Running), that show the state of the simulation clock, and that list the rules that are presently on the rule evaluation stack (in the *Current Events* window). There is also a list of *simulation attributes* and their values. Simulation attributes are not associated with any particular object, but rather with the simulation as a whole. Examples include the current simulation clock value and mouse information.

This chapter will show you how to interpret the data shown in these windows, and how the major modes of the scene editor are used to edit and test simulations.

## Building a Simple Simulation

In this exercise, you will create a simulation of a simple circuit that controls two lights. When the switch is put into the 'up' position, the upper light comes on. When the switch is in the lower position, the lower light comes on.



Follow the step-by-step process described below to build a simple simulation like this yourself.

### Starting the Scene Editor

Just as the generic editor can be started in three ways, so can the scene editor. You can start a new scene editor session by using the Lisp invocation:

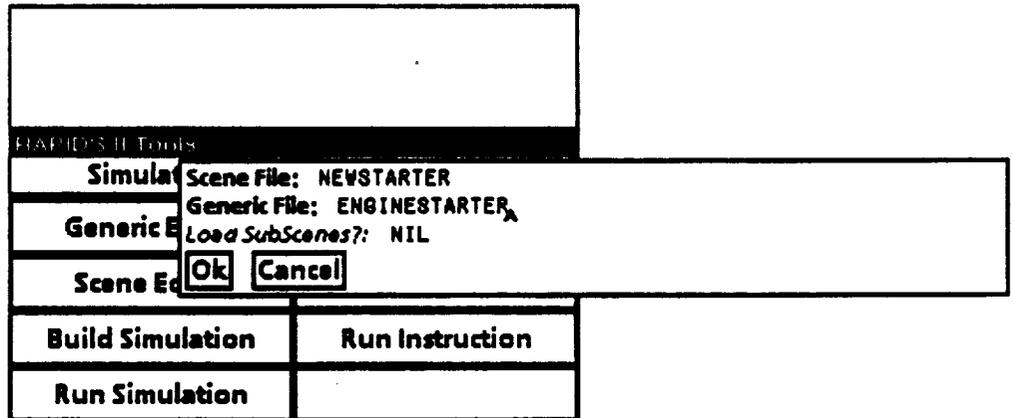
`(SceneEdReal 'YourSceneName 'ALibrary name)`

Where *YourSceneName* is the name of the scene you are building or editing and *ALibraryName* is the name of the library that will serve as the source of new objects. To create the simple scene described here, you need to use the library with the Bladefold objects. Type

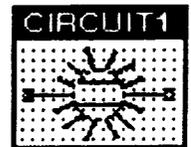
`(SceneEd 'CIRCUIT1 'SIMPBLADEFOLD)`

in the executive window. This will open the scene editor on an empty scene called CIRCUIT1. The Bladefold library of generic objects will be available.

The second way to invoke the scene editor is to use the RAPIDS II top-level menu. Click on the *Scene Editor* command, and a dialog box will open that asks for the names of the scene file and the generic file, as in the screen snapshot shown below.

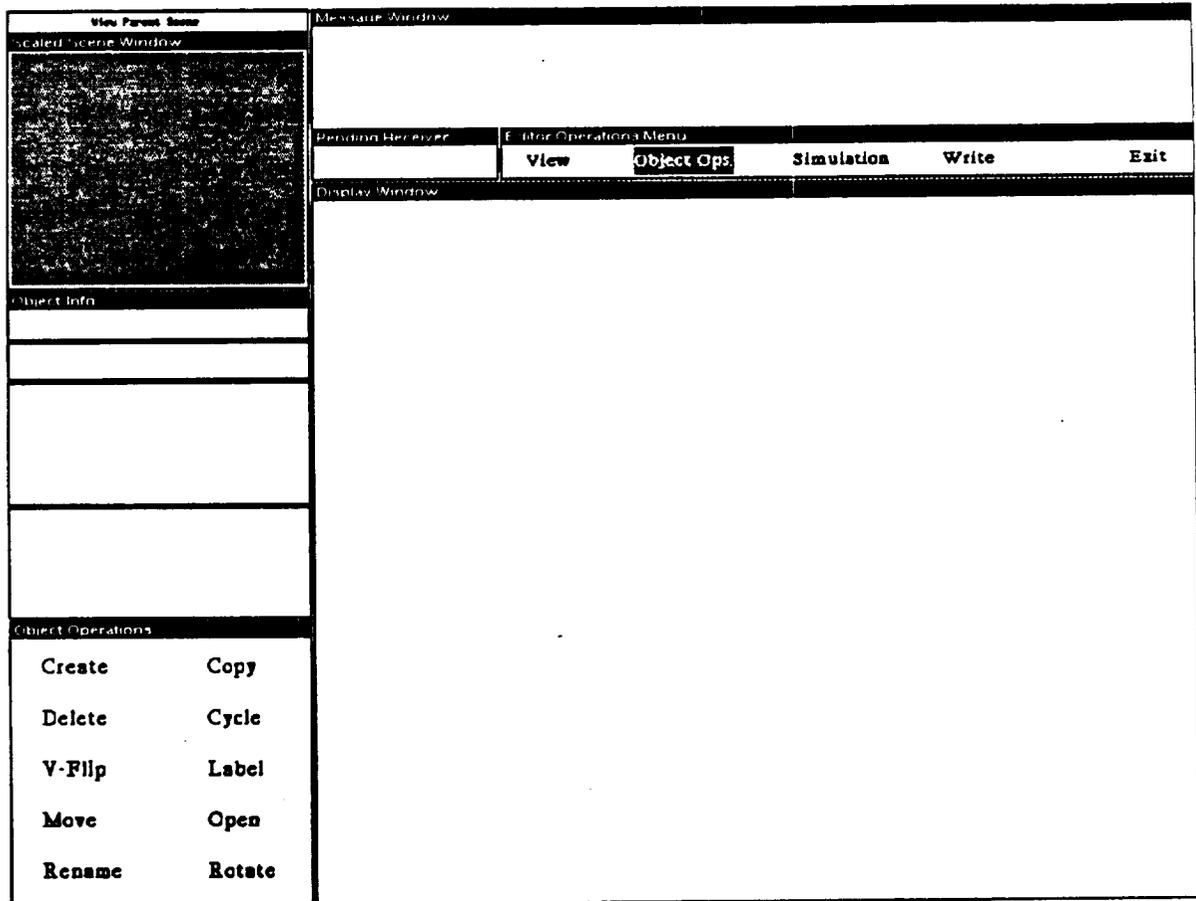


Finally, if a scene editor session was started earlier and suspended (by a method described later in this chapter), there will be a shrunken window on the screen. The window will show the object that was selected the first time that the scene was shrunk. The window title is the name of the scene.



You can resume that editing session either by choosing *Expand* from the right mouse button menu in the icon or by clicking on the icon with the middle button. Either action will open up the scene editor session in the state that it was left in.

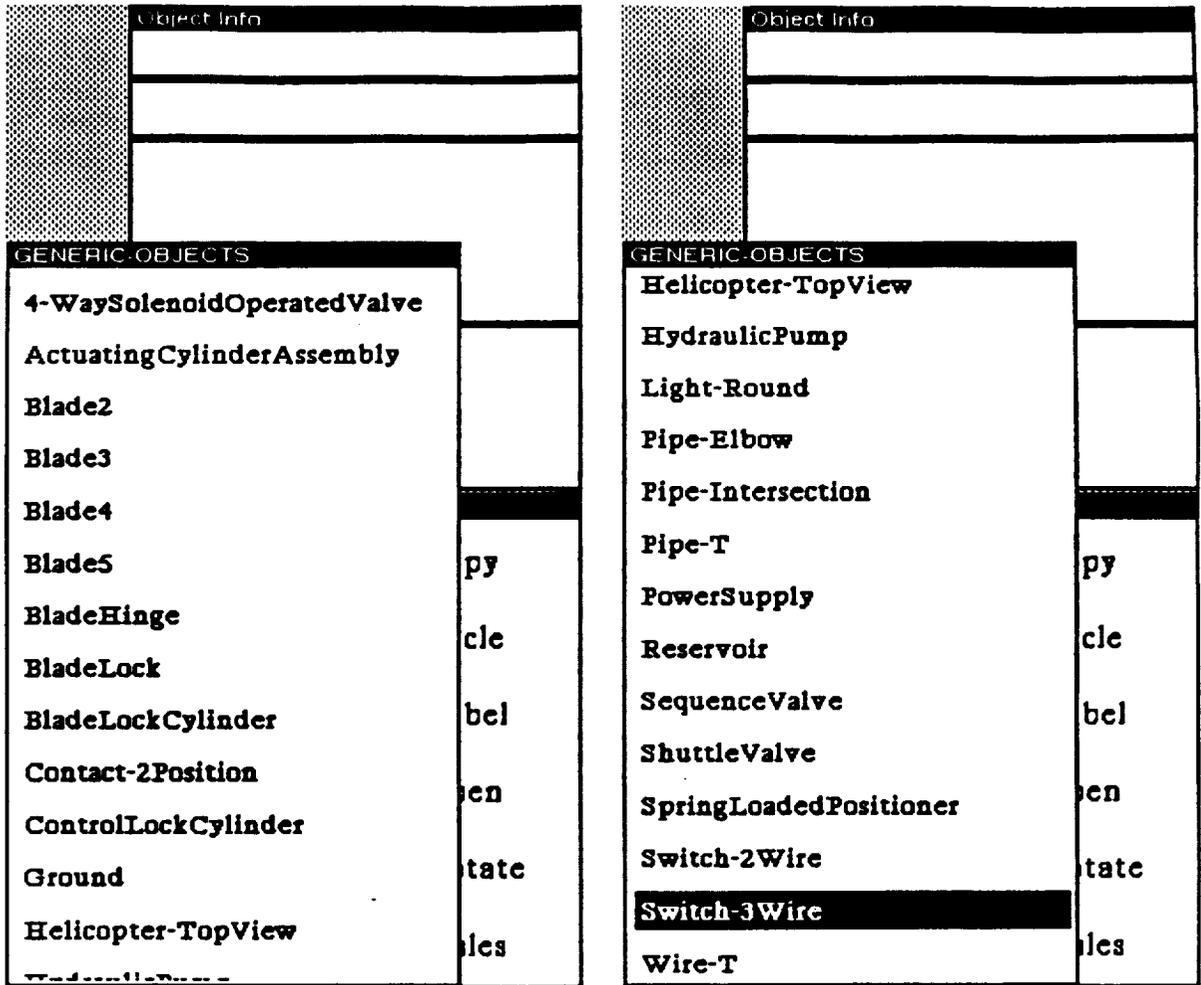
In this example, you are creating a brand-new simulation, so the scene editor will open with nothing in the display window.



You must create a number of specific objects on the simulation scene. Select the *Create* menu item from the *Object Operations* menu.

**The Generic Objects Menu**

*Create* is the command that lets you create a specific instance of some generic object. It brings up a list of the available objects — all those that are in the currently active library. In the figure below, the *Generic Objects Menu* for the 'SIMPBLADEFOLD library is shown. This is a scrollable menu with the names of the generic objects in the library given in alphabetical order. Scroll the menu until the item 'Switch-3Wire' appears (as in the right half of the figure below). Click the mouse on this item to select it.



*Switch-3Wire* is a simple two-position switch. A specific object based on the named generic object will now be drawn in the display window. The object will appear in the lower right corner of the window and will move with your mouse movements until you click the left button to drop it where you want it.

Object Info		
<b>Switch-3Wire0078</b>		
Up	(92 368 10 10)	Up
Down	(92 333 10 10)	Down
CurrentState	atom	Down
ObjectLocX	int	73
ObjectLocY	int	333



Position the switch on the scene and click the mouse button. The pattern of dots in the region of the switch shows that it is the currently selected object on the scene. The Object Info windows now show data about this specific object.

When new objects are created in the scene editor, unique names are constructed for them. These names are based on the names of their generic objects, with some appended numerals. It is a good idea to change the names of simulation objects to be suitable in the simulation context. To do this, click on the *Rename* command in the Object Operations menu. Type in a new name in response to the message window prompt.

```

Message Window
Please enter a new name for selected object >> Selector
    
```

Because the switch will be used to select between two lights, it would make sense to call it 'Selector.'

**Adding Objects**

Use *Create* again to add an instance of the *PowerSupply* to your simple simulation. Position it to the left of the switch and use *Rename* to give it a name like 'Main Power Supply.'

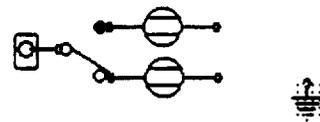
Object Info		
MainPowerSupply		
CurrentState	atom	Power
ObjectLocX	int	51
ObjectLocY	int	341



Because it is the currently selected item, the pattern of dots is now shown in its region, rather than in the region of the switch.

Use *Create* to add a couple of instances of the *Light-Round* object type and one of *Ground*. You should have a scene that looks something like that shown below.

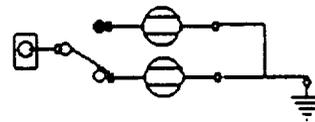
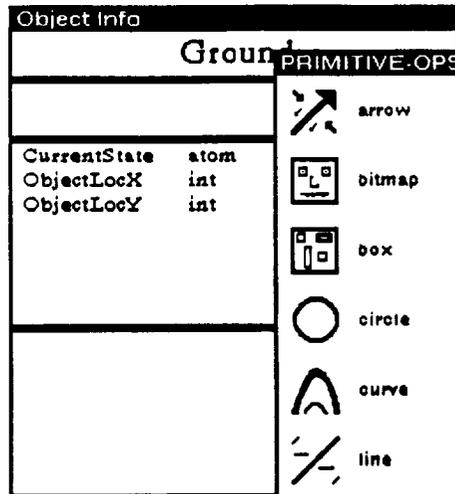
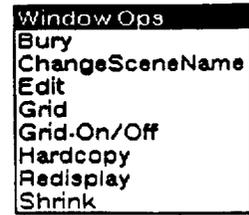
Object Info		
Ground		
CurrentState	atom	Grounded
ObjectLocX	int	197
ObjectLocY	int	313



**Drawing  
Background  
Elements**

A number of different techniques can be used to connect electrical objects, including creating new *Wire*-type objects. In many cases, however, it is sufficient to simply create background graphics that visually connect the objects.

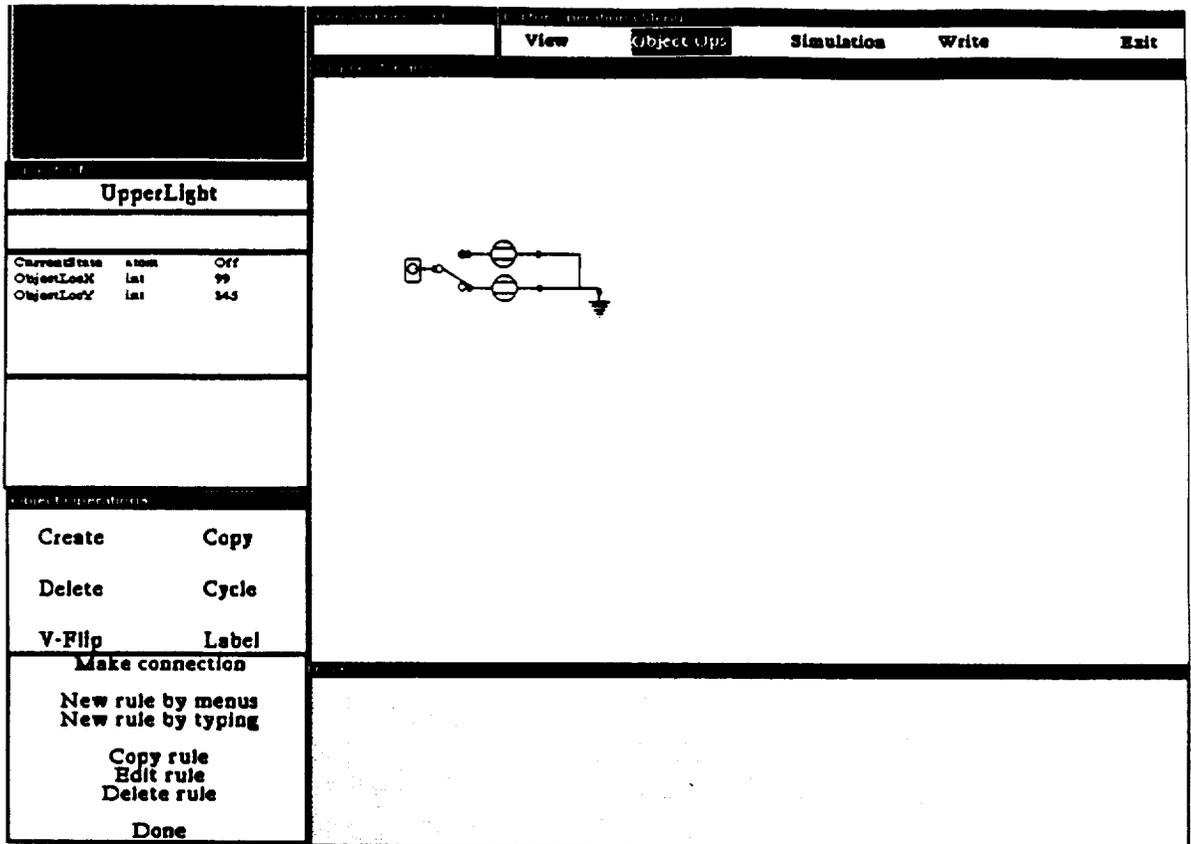
Hold down the right mouse button in the display window to bring up the *WindowOps* menu. Select the option called *Edit* to bring up the same menu of graphics authoring tools that is used in the generic editor. Any drawing elements you create now will belong, not to any of the objects on the scene, but to the scene as a whole. Such graphical elements are called *background graphics*.



Use the *line* tool to visually connect the two lights to the ground. When you have finished modifying the scene in this way, click on *Done* in the drawing menu to exit the background editing mode.

At this point, you have built a visual display that doesn't do much. Student users will be able to manipulate the switch (because of the way its generic object was defined), but such manipulations won't make the lights change. To get that effect, it is necessary to define some behavior for them. You must enter rules for the lights that tell them how to act, based on the state of the switch.

Select the upper light, then choose the *Rules* command. The rule editor, described in Chapter 4, will open.



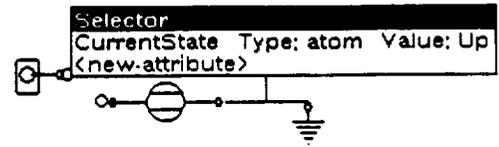
Using the menus, build a rule for the upper light that turns the light on when the switch is up. A rule like this

```
(if ((CurrentState of Selector on CIRCUIT1) SameAs 'Down)
  then (SetState 'On)
  else (SetState 'Off))
```

expresses this behavior. Note that the attribute reference for this external attribute specifies the object and the scene name as well as the name of the attribute.

In building the condition, you will specify that the condition is a comparison of the 'SameAs' type that refers to the <attribute-of-another-object>. When the rule editor is ready to be told what this attribute of another object is, you should specify the CurrentState of the switch (which we called Selector in this example). At this point the rule editor windows will be cleared from the display window, so that you can point to the 'other-object' that you want to specify. The way to indicate such an object is to hold down the shift key and click on the object. When you click on the switch, you will see a menu pop up that lists its attributes, as in the figure below:

Object Info		
UpperLight		
CurrentState	atom	Off
ObjectLocX	int	99
ObjectLocY	int	345



Here the only existing attribute of the switch is its CurrentState.

When you have finished building the rule for the upper light, build a similar one for the lower light.

```
(if ((CurrentState of Selector on CIRCUIT1) SameAs 'Down)
  then (SetState 'On)
  else (SetState 'Off))
```

This external rule specifies that the light is to go on when the switch is down, and to go off when the switch is up.

**Test Simulate**

Your simulation is now ready to test. You can perform test simulations without leaving the scene editor. Thus far, your work has all been done in the *object operations mode*. This is the default mode for the scene editor. To simulate, change to the *Simulation mode*. At the top of the display window is the *Editor Operations Menu*.

Pending Receiver	Editor Operations Menu				
	View	Object Ops.	<b>Simulation</b>	Write	Exit

Click on *Simulation* in this menu bar. You'll notice that the menu at the left of the display window changes. It now displays the *Simulator Operations Menu*. Click on the *PAUSE* button just to the left of the clock window. The clock will begin counting. If everything has been hooked up properly, you should see the light go on.

When you are in simulation mode in the scene editor and the simulator is not paused, you can manipulate switches the same way that students do in the RAPIDS II runtime environment. Play with your simulation for a while.

---



---

## How Simulation Works

It is possible to build effective simulation-based courses in RAPIDS II without understanding in detail how simulation works. This is particularly true if you are building simulations like the simple circuit presented above — simulations that do not schedule or unschedule events and that do not make use of processes.

In more complex simulations, however, authors often find that they have to debug their simulations. The scene editor's *simulation operations* mode makes it possible to view the values of attributes, to step through rule execution, and to perform many other detailed actions that help the debugging process. In order to carry out such actions, however, the author must have an understanding of how the simulation works.

### Overview

First consider the simple case of a simulation that does not have scheduled events or processes. The simulator spends most of its time waiting for the student to take an action (by clicking on a simulated switch). When a switch is thrown, its state is reset, and its *CurrentState* attribute is automatically changed.

Every attribute has an associated list of rules — the rules that refer to that attribute. For example, a 'PowerOn' light might have a rule something like this:

```
(if ((CurrentState of PowerSwitch) SameAs 'On)
    then (SetState 'Shining)
    else (SetState 'Off))
```

The *CurrentState* attribute of the PowerSwitch object will include this rule in its list of affected rules.

Whenever an attribute changes value, all of the rules that it affects must be run. In the case of the example above, the rule that sets the state of the PowerOn light should be run because the *CurrentState* attribute of the PowerSwitch changed. (The action portion of the rule — the *SetState* — will be carried out only if it actually changes the state of the PowerOn light.)

Affected rules are not run immediately when an attribute value changes. Instead, those rules are put on a list called *CurrentEvents*. Current events are all the things that are supposed to happen essentially simultaneously. After all the rules that refer to a just-changed attribute have been put on the Current Events list, then those rules are executed.

After the rules in the CurrentEvents list have been carried out, any graphical changes that are required are done. If a set of 'simultaneous' rule executions cause a number of changes in state and changes in the locations and rotations of objects or object states, then all those graphical effects are displayed at the same time at the end of the process of carrying out the CurrentEvents rules.

**The Clock**

The RAPIDS II simulator runs on computers that are not parallel machines. That is, they can do only one thing at a time. RAPIDS II simulates real-world events that may be simultaneous. The conflict (between the computer's ability to perform only one action at a time and our desire to simulate simultaneity) is resolved by using a *simulated* real time clock. When the simulation wants many things to happen at once, this clock freezes until they have all happened. Then simulated real time is set to the actual time, and the clock ticks on normally. The simulator does not fall further and further behind real time, because it jumps ahead to the current time whenever it has finished all the 'simultaneous' actions it was just working on.

**Scheduled Events**

Authors can write rules that schedule events. The simulator checks to see whether it is time for any scheduled events after it empties the CurrentEvents list. If scheduled events are due to take place, then they are added to the CurrentEvents list and the process begins again.

If the time for a scheduled event has already passed, then the clock is set back to the time at which the event was supposed to take place.

It is possible to write a (defective) simulation that never gets out of the current events list. For example, the two rules:

```
(Assign AttributeA AttributeB + 1)
(Assign AttributeB AttributeA - 1)
```

form a tight infinite loop. A simulation that includes these rules will never perform any scheduled events, because the CurrentEvents list will never be empty. The author has asked the simulator to simultaneously perform two operations that affect each other, and the simulator will keep working at those tasks to the exclusion of all else.

**Simulation Attributes**

There are a small number of attributes that belong to the simulation as a whole, rather than to particular objects. These attributes include the simulator clock, the position of the mouse, and the state of the mouse (whether there has been a click, for example).

After the CurrentEvents list has been emptied, the simulator checks to see if any of the simulation attributes have been changed. If so, any rules that refer to these attributes are added to the CurrentEvents list for execution during the next pass.

**Processes**

The simulator maintains a list of active processes. When a rule execution results in a *StartProcess* effect, the new process is added to this list. When the CurrentEvents list has been emptied, the simulator works through the list of active processes and carries them out.

The form of a *StartProcess* effect is

```
(StartProcess <partial-prim-effect> <numeric> <numeric> <numeric>)
```

An example of a rule that starts a process is

```
(if ((CurrentState of DrainValve) SameAs 'Open)
  then (StartProcess (Assign CurrentVolume) CurrentVolume 2 0)
  else (StopProcess (Assign CurrentVolume))
```

This is a rule that sets up a process that drains away a reservoir's volume at a constant rate (2 units per second) when the reservoir's drain valve is opened. The StartProcess specifies the effect that is to be carried out (an Assign to the CurrentVolume attribute. The next three values are the starting value for the assignment, the rate (in units per second), and the destination value. Here the starting value is the value at the time that the process is posted to the active processes list. The rate of change is 2 per second. The destination value is 0.

When a process reaches its destination value, it is removed from the active processes list. If the process would have advanced beyond the destination value, then the simulation clock is set back proportionately. In this way, the termination effects of every process are certain to be simulated.

### A Summary of RAPIDS II Simulation

In simplified pseudo-code, this is how the RAPIDS II simulation works:

```
Initialize simulation
Repeat
  if UserEvent (such as mouse action)
    then add affected rules to CurrentEvents
  if ScheduledEvent
    then for each scheduled event
      carry out the action and
      add affected rules to CurrentEvents
  if ongoing processes
    then for each process
      carry out the action and
      add affected rules to CurrentEvents
  While there is a rule in CurrentEvents
    carry out the action
    and remove the rule from the list
    add new affected rules to CurrentEvents
  Show all graphical effects of the events
Until user stops the simulation
```

It is useful for authors to understand enough about how the RAPIDS II simulator works that they can make effective use of the debugging tools that have been built into the *simulation operations* mode of the scene editor. The next section describes some specialized views of simulation elements that are provided in RAPIDS II. An understanding of the simulation at the level of detail described in this section should make it possible to use these views effectively.

### Simulation Data

Implicit in the above discussion were references to the major types of data used by the RAPIDS II simulator. These data elements include objects, handles, attributes, and rules.

**Object** Object data includes a list of handles, a list of attributes, a list of rules, and a set of possible graphical appearances, called *states*. For most purposes, the underlying simulation algorithms are much more concerned with attributes and with rules than with objects. Objects are natural for authors to deal with, however. They provide authors convenient access to attributes and to rules.

**Handle** A handle is a region (necessarily rectangular in RAPIDS II) that is sensitive to mouse clicks. It is handles that make user events

possible. There are three different kinds of handles: object handles, state handles, and attribute handles. Mouse actions in *object handles* have simulation effects if authors have written rules that refer to actions in those handles. Mouse actions in *state handles* automatically change states. Mouse actions in *attribute handles* create temporary assignment rules that send values to test equipment probes.

**Attribute** All values of interest in a simulation are stored in attributes. Attributes play a role similar to that of variables in programming languages. Attribute data includes the *type* of the attribute (such as integer or string), its current value, and a list of the rules that refer to the attribute.

**Rule** Rules provide the mechanism for changing values of attributes. A rule includes the literal rule expression that can be edited using the rule editor, a list of the *triggers* of the rule — the attributes that are referred to, and the *owner* of the rule — the attribute that may be changed by it.

In addition to these basic simulation data types (and many others that you need not be aware of to author effectively), RAPIDS II has a number of complex data types that you should be aware of. These include simulation attributes and the current events list, which are described above in this section.

---

---

## Viewing Simulation Data

The simulation data described above can be viewed in the *simulation operations* mode of the scene editor. This feature can be very useful for understanding the behavior of your simulation and for debugging it. This section previews these views of simulation data. The sections on Simulation Operations and on Simulation Debugging, below, give additional descriptions of how they are used.

**Object Data View** The *Object Info* windows provide essential object data. At the top is the name of the object. Just below is the list of handles associated with the object. In the box below the handle box is a list of the object's attributes. The last box presents a list of rules associated with the object.

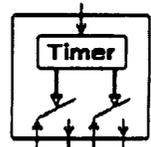
Object Info		
Left Timer		
Open	(82 438 6 6)	Open
CurrentState	atom	Open
ObjectLocX	int	40
ObjectLocY	int	415
RightOutput	int	0
RightInput	int	100
TriggerVoltage	int	0
LeftOutput	int	0
<pre>                     (if ... then (Schedule (SetState))) *no                     (if ... then (SetState)) *no                     (if ... then (Assign RightOutput) else ...) *no                     (if ... then (Assign LeftOutput) else ...) *no                     (Assign RightInput) *no                     (Assign LeftInput) *no                     (Assign TriggerVoltage) *no                 </pre>		

Objects have two types of attributes. One kind, called *system attributes*, are those that can be set by rule actions other than *Assign*. These attributes include *CurrentState* (which is set by *SetState*) and object and state location attributes (which can be set by a variety of movement functions). The second type of attribute is *author-defined* attributes.

System attributes are listed first in the attribute list, followed by the author-defined attributes. The first of the system attributes is *CurrentState*. The value of this attribute is the name of the currently displayed state in the display window.

**Object Graphic View**

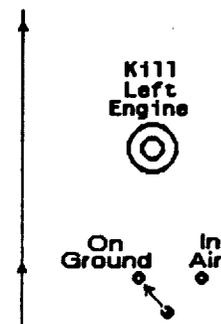
The simulation Display Window shows all the visible objects on a scene in their current states. Each object's graphic view is shown in this window.



**Handle Data View**

The box immediately below the object name in the *Object Info* windows provides a view of the object's handle data. One line of data describes each handle. A line of handle data includes, first, the name of the handle; second, the rectangular region of the handle; and, third, the name of the state associated with the handle. (Only state handles have associated state names. Object handles do not.) In the figure below, the *OnGround* handle has the rectangular area (59 130 34 32) for its region and the associated state named *OnGround*. It is often a good idea to give state handles names that are the same as or closely correspond to their associated states.

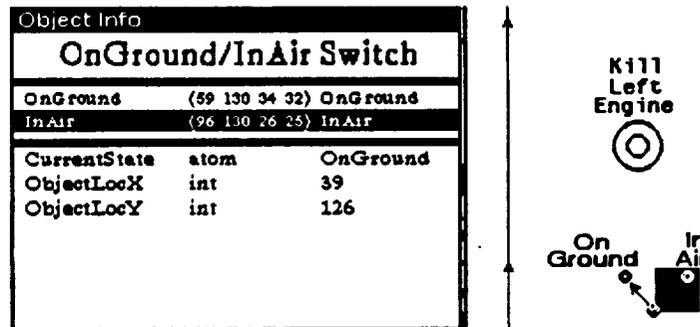
Object Info		
OnGround/InAir Switch		
OnGround	(59 130 34 32)	OnGround
InAir	(96 130 26 26)	InAir
CurrentState	atom	OnGround
ObjectLocX	int	39
ObjectLocY	int	126



**Handle Graphic View**

It is the author's responsibility to make handles graphically distinctive for students. The simulation appearance should make it reasonably clear to students which areas are touch sensitive.

In the scene editor, however, a special feature is available to show authors the location of handles. If an author holds down the middle mouse button on a line of handle data, then that line will be highlighted and the corresponding handle region will also be highlighted in the display window, as in the figure below.



**Attribute Data Views**

Attribute data can be viewed in a number of different windows. These include the object attribute list shown in the Object Info windows (as in the above figures), and a number of more specialized views of attribute data.

Each line of the object attribute list has three (sometimes four) elements. The first item on the line is the name of the attribute. The second item is the attribute's type, and the third item is the current value of that attribute. If the attribute has an associated *attribute handle*, then the letter *H* appears as the last item on the line.

In the figure above, the first line of the attribute list for the OnGround/InAir Switch presents data for its CurrentState system attribute. The name of this attribute is *CurrentState*, its type is *atom*, and its current value is *OnGround*.

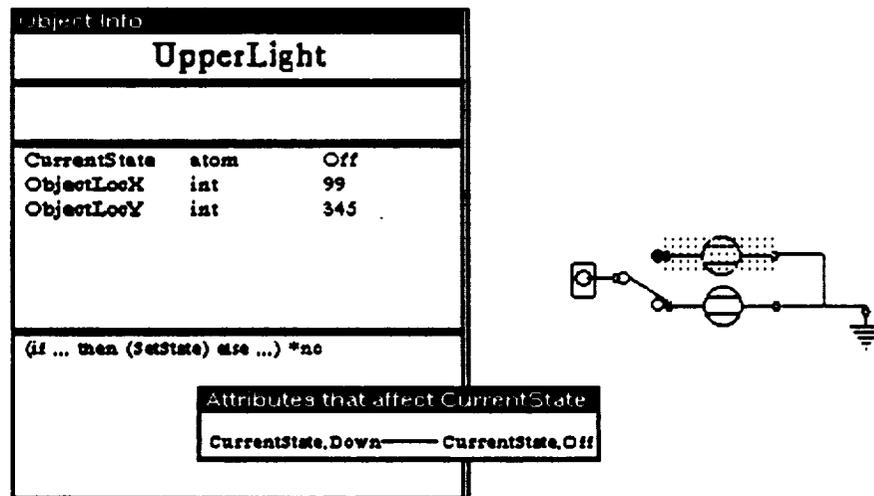
Attributes can also be viewed in windows that present the flow of effects in a simulation. When the left mouse button is clicked on an attribute name, a menu of attribute options appears. In the case of one of the attributes in an *Object Info* attribute list, a menu of options will appear as at the right.

- |  |
|--|
| <ul style="list-style-type: none"> <li>Edit Pause Condition</li> <li>Inspect</li> <li>Pause On/Off</li> <li>Set</li> <li>Trace On/Off</li> <li>Who Affects Me</li> <li>Whom Do I Affect</li> </ul> |
|--|

The details of this menu are described in the section on Debugging Simulations, below. For now, consider the last two options, *Who Affects Me* and *Whom Do I Affect*. When either of these options is selected, a window opens that shows a network of affects. In the case of *Whom Do I Affect*, the leftmost (or *root*) item is the selected attribute. In the case of *Who Affects Me*, the selected attribute will appear at the right, with one or more affecting attributes on the left. The size of the *Affects* window will depend on how large the network of affects is in the simulation. If the network of affects is large, the window will not be large enough to show all the effects. In that case, the window will be scrollable. Here are very simple examples of the two

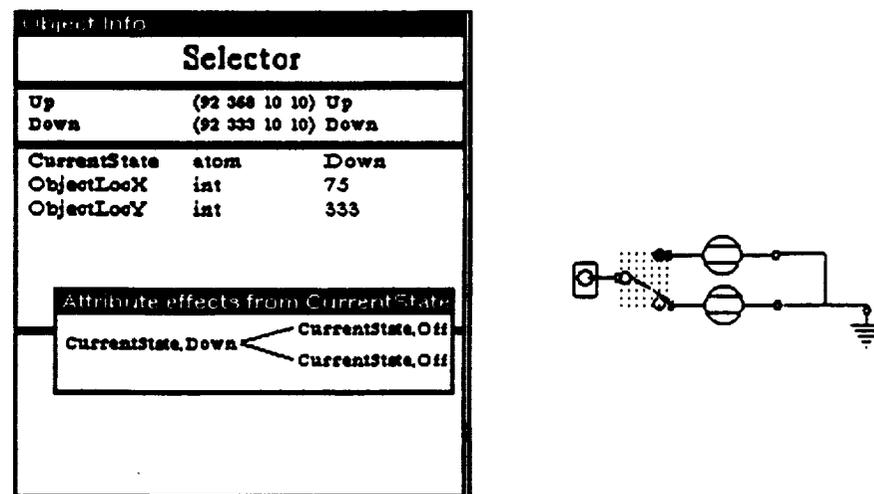
types of *Affects* displays. Both are taken from the simple circuit example at the beginning of this chapter.

In the figure below, an author has selected the UpperLight object, then clicked the left button on its CurrentState attribute in the attribute list (the third box in the *Object Info* window). When the menu of attribute options popped up, the author chose *Who Affects Me*. A small window appeared that shows that only one other attribute affects the CurrentState of UpperLight — an attribute that is also called CurrentState, and that has the value Down at this time. (See the figure.)



The textual objects that are shown in the Affects window are *attributes*, not objects. Each attribute in this window is represented by its name and its current state.

In the figure below, the author has selected the Selector switch, then clicked on its CurrentState attribute in the attribute list at the left, then chosen *Whom Do I Affect*.



In this example, two attributes (the CurrentState attributes of the two lights) are affected by the selected attribute. The Affects window can sometimes be confusing because only attribute names, not object names are used. In this example, all three different attributes shown in the Affects window have the same name. Authors can sometimes recognize which object's attribute is meant by observing the current value of the attribute. (In this example, only the Selector switch's CurrentState can have the value Down, so it is clearly the one at the left.)

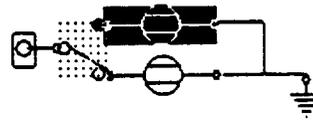
**Attribute Graphics View**

RAPIDS II provides another way to figure out what object is implicitly referred to by an attribute name in an Affects window. If the author clicks the middle mouse button on one of these names, the object that has that attribute will be highlighted in the Display Window, as in the figure below.

Object Info		
Selector		
Up	(92 368 10 10)	Up
Down	(92 333 10 10)	Down
CurrentState	atom	Down
ObjectLocX	int	75
ObjectLocY	int	333

Attribute effects from CurrentState	
CurrentState, Down	CurrentState, Off
	CurrentState, Off



In a sense, attributes have no graphics. Only objects have graphics. In order to highlight an attribute graphically, the scene editor highlights the object. If the object to be highlighted is on a different scene, that scene appears in the *scaled scene window*, with the object highlighted.

This technique for highlighting an attribute's object graphics — clicking with the middle button on the attribute data — is not restricted to use in the Affects window. An author can also middle-click on an attribute in the *Object Info* window's attribute list, and the corresponding object will be highlighted.

Some attributes are associated with particular locations on an object. These are the objects that were given *attribute handles* in the generic editor. If an author middle-clicks on the data of such an attribute, only the attribute handle is highlighted, not the entire object.

**Simulation Attribute Data Views**

Simulation attributes — the universal attributes associated with the clock and the mouse — can be viewed in much the same way as ordinary object attributes. In the Simulation Operations mode, this window appear just above the *Object Info* windows.

Simulation Attributes		
Clock	int	0
MouseX	int	40
MouseY	int	11
MouseState	atom	Up

Clicking on simulation attribute data with the left mouse pops up the standard menu that lets an author ask for an Affects window display for the attribute. Clicking on one of these attributes with the middle button has no effect, because they do not have an associated object that could be highlighted.

Simulation attributes may appear in Affects windows, just like object attributes. The attribute options menu works for them in such windows as well.

**Rule Data View**

Below an object's attribute list in the *Object Info* windows is its rules list. The rules in this list are shown in an abbreviated form. At the end of an abbreviated rule, there may be a tag that gives additional information about the form of the rule in the environment.

If an abbreviated rule has a trailing *\*nc*, it means that the rule has not been compiled in this environment. (All the rules of a scene can be compiled by using the *Compile* command in Simulation Operations mode.) Compiling the rules makes them run a little faster than they would otherwise.

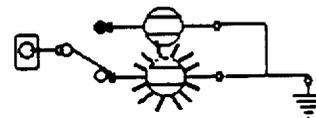
If an abbreviated rule is followed by *\*nw*, it means that the rule is not working in this environment. There are two reasons that a rule might not be working in an environment. The most common reason is that the rule refers to objects from another scene that has not been loaded in the current scene editor session. Such rules are inactive in the simulation operations mode, even though they might work perfectly in a complete simulation. One way to avoid this problem is to always require that the scene editor load all related scenes. This is done by giving the *Load SubScenes?* field of the dialog box the answer *T*. (See the figure below.)

RAPIDS II Tools	
Simulation	Scene File: NEWSTARTER
Generic	Generic File: ENGINESTARTER
Scene Editor	Load SubScenes?: NIL
Ok Cancel	
Build Simulation	Run Instruction
Run Simulation	

The second reason that a rule could be marked as *\*nw*, or not working, is that it could have a syntax error. This will not happen if the rule was built using menus, but can happen when rules are edited using the SEdit editor.

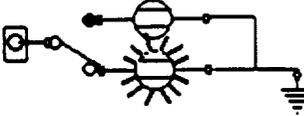
The abbreviated form of a rule in the rules list ordinarily makes it impossible to be read what the rule actually does. Fortunately, there is an easy way to expand the rule into a more readable form. If an author clicks the left mouse button on one of these rules a list of rule options appears in a menu, as in the figure below.

Simulation Attributes		
Clock	int	15445
MouseX	int	42
MouseY	int	14
MouseState	atom	Up
Object Info		
LowerLight		
CurrentState	atom	On
ObjectLocX	int	100
ObjectLocY	int	318
<div style="border: 1px solid black; padding: 2px; display: inline-block;">                     Edit Pause Condition                      Expand                      Inspect                      Pause On/Off                 </div>		
<small>Alt . then (Set) key /</small>		



The use of these options is explained in the section on Simulation Debugging later in this chapter. The second option, *Expand*, makes it possible to view the data of a rule. When *Expand* is selected, the standard Interlisp SEdit window opens and displays the selected rule in detail. The rule is shown as a conventional C-Lisp S-Expression, so it has many sets of nested parentheses, as can be seen in the next figure.

Simulation Attributes		
Clock	int	15445
MouseX	int	42
MouseY	int	14
MouseState	atom	Up
Object Info		
<b>LowerLight</b>		
Rule of object LowerLight on scene CIRCUIT1		
<pre>(if ((CurrentState of Selector on CIRCUIT1) SameAs 'Down) then (SetState 'On) else (SetState 'Off))</pre>		
(if ... then (SetState) else ...) *no		



In this expanded view of the rule, authors can take action to evaluate a portion of the rule. Clicking the middle mouse button on a part of the rule makes the value of that part of the rule appear in the message window. For example, clicking on the parenthesis in front of 'CurrentState of Selector on CIRCUIT1' makes the message window display *(CurrentState of Selector on CIRCUIT1) — Down*. Clicking on the next outermost level of parentheses evaluates the larger rule segment, '(CurrentState of Selector on CIRCUIT1) SameAs 'Down)'. The message window says that this rule segment is currently True.

**(CurrentState of Selector on CIRCUIT1) — Down**

Value of rule segment: True

**Rule of object UpperLight on scene CIRCUIT1**

```
(if ((CurrentState of Selector on CIRCUIT1) SameAs 'Down)
then (SetState 'On)
else (SetState 'Off))
```

Rule data are displayed in abbreviated form in the object info windows and in the CurrentEvents list. The same features just described for getting and evaluating expanded views of rules from an objects rule list can also be applied to rules in the CurrentEvents list.

RAPIDS II provides additional views of rule data in the rule editor itself, which is described in Chapter 4 of this manual.

**Rule Graphics View**

If the author clicks the middle mouse button on an abbreviated rule, the object that has that rule will be highlighted in the Display Window. This works in essentially the same way that the Attribute Graphics View (described above) does.

## Editor Operations

The highest level menu in the scene editor is the *Editor Operations Menu*. It is always displayed while the editor is running.

Pending Receiver	Editor Operations Menu				
	View	Object Ops.	Simulation	Write	Exit

This menu, which is positioned just above the display window, is used to change the modes of the scene editor. While carrying out the exercise of creating a simple simulation scene, you carried out most operations in the *object operations* mode. To test and debug your simulation, you used the *simulation* mode.

The *View* command is described below in the discussion of multi-scene simulations. This menu item brings up the parent scene of the current scene in the Display Window. An author can work up through a hierarchy of scenes by repeatedly clicking on *View*.

The *Object Ops.* command puts the scene editor into the object operations mode. In this mode authors can add objects to the simulation, move them, delete them, create and edit rules, and so on. This mode is used to build simulation scenes and link them together.

The *Simulation* command puts the scene editor into the simulation operations mode. In this mode, authors can run simulations interactively. Many special debugging windows are available for inspecting simulation data.

The *Write* command is used to save the currently displayed scene on your disk. Be sure to use it whenever you have made changes. Notice that some changes to a simulation scene are fairly subtle, and you have to work at it to remember to save. For example, suppose you delete an object on one scene that is connected to an object on another scene. You not only need to *Write* the scene with the deleted object, but also the scene that it was connected to. Be sure to bring up that scene and *Write* it to your disk, as well.

The small window below the display window will be inverted when changes have been made to the current scene but it has not yet been saved. After you do the *Write*, you'll see this window change back to the normal black text on white background.

The *Exit* option lets you leave the scene editor. If you have made changes that you haven't saved, the message window will name these altered but unsaved scenes. In this case, you'll be offered a menu of three choices

- Exit without writing files*
- Write altered files before exiting*
- Cancel exit*

The first option lets you leave the editor and abandon all changes that were not already explicitly saved with the *Write* command. The second option will automatically save all the altered files and then quit. The third option interrupts your *Exit* command so that you can continue editing. If you want to save the changes made to some scenes but not the changes made to other scenes, you can go to the scenes with changes you want to save and use the *Write* command there. Then select the *Exit* command and choose the *Exit without writing files* option. Ordinarily, of course, you will want to save all the changes you have made.

## Object Operations

If you worked through the example at the beginning of this chapter, then you are already familiar with many of the commands of the object operations menu. This section reviews those commands and presents others you may not have used yet.

Object Operations	
Create	Copy
Delete	Cycle
V-Flip	Label
Move	Open
Rename	Rotate
Scale	Rules

- Create** Choosing the *Create* option brings up the menu of generic objects. The *Generic Objects Menu* is a scrollable list of all the objects in the current library. Click on the name of the object you want to use as a template for a new specific object. If you change your mind and don't want to create a new specific object, just click on the title bar of the generic objects menu (the black bar with the word GENERIC-OBJECTS at the top).
- Sometimes it is difficult to remember what kind of object is meant by a particular name in the generic objects menu. If you are using one of the supplied libraries, you may want to use the Appendices to this manual to help you select objects by their names. The scene editor also has a built-in feature that helps you identify generic objects. If you point to an object name and hold down the middle button, a picture of the object type will appear above the menu.
- Copy** To create a new specific object of the same type as the selected object, choose the *Copy* command. The new object (the copy) will then be the selected object, and it will move with the mouse. Clicking the left mouse button will deposit the object at the location of the mouse pointer.
- Newly copied objects have the same behavior as the originals, because they have copies of the rules of the originals.
- If a scene icon object is copied, the new object will not point to the original scene (or any other). If you want it to function as a scene icon, you will have to *Open* it.
- Delete** The selected object is deleted. It disappears from the scene. If the object was labeled (see below), then its label will also be removed. Rules that referred to attributes of the deleted object should be edited appropriately (perhaps by replacing a deleted attribute with one that still exists).
- The rules of other objects that refer to deleted attributes will not work. In the *Object Info* windows, such rules in the abbreviated rules list will have a *\*nw* appended to indicate that they are not working.
- Cycle** The *Cycle* command is used to change the state of the selected object. Repeatedly choosing *Cycle* steps through the available states. You may need to use this option to put an object into a state that accords with its context in the scene. That is, you wouldn't want to build a scene in which the objects are in conflicting states. Use cycle to put objects into compatible states before going into the simulation mode.
- If an object with a continuous state is cycled, the state part will move through ten percent of its extent each time that the *Cycle* command is selected. (The increment of ten percent can be edited by the author.)
- V-Flip** *V-Flip* is used to make an object do a vertical flip on the scene. More formally, the object is displayed upside down and mirror imaged. By combining the *Rotate* and *V-Flip* commands, you can show an object in any orientation.

Objects that have text components may look a little strange after undergoing *V-Flip*. The text is not actually flipped, but is put into a different relative position. You may want to avoid flipping objects with text elements.

**Label** A specific object can have one or more associated labels on its scene. Labeling can mean much more than adding textual elements. Any of the graphic primitives of the *Primitive-Ops* menu can be added to an object.

Using the *Label* command, you can add static graphical elements to any object in a simulation. Some authors use this technique to build short graphical wires and pipes to visually connect neighboring objects.

**Move** If you choose *Move* when there is a selected object, then that object will move with the mouse. Clicking the left button will position the object again. If no object is selected, this menu command will have no effect.

**Open** Using *Open* converts the selected object into a scene icon. It will serve as a link to another scene. To open the scene that corresponds to a scene icon, the student must double-click on the scene icon object. Double-clicking means clicking twice in very close succession without moving the mouse. When a scene icon is opened this way, the scene in the Display Window is replaced by the scene associated with the scene icon.

After you click on *Open* you will be asked to click the left button if you want to type in the name of the scene that the object should represent. If you click the left button and type in a scene name, then the selected object will be linked to the named scene. If you click the right button, a new scene will be created and given the name of the specific object.

**Rename** This command lets you change the name of a specific object. When you build a simulation you may have scenes with many objects based on the generic object name plus a number. You can replace these names with more appropriate ones. Using meaningful names will help you in constructing your simulation and may help your students during simulation training.

**Rotate** The *Rotate* menu command rotates the selected specific object 90 degrees counterclockwise. The text elements of an object are not rotated, but are simply repositioned. You may prefer not to rotate objects based on generic types with text elements. It also usually works better to wait to label a specific object until after you have rotated it.

**Scale** Scaling has not yet been implemented. A message to that effect appears in the Message Window when the *Scale* command is selected.

**Rules** The *Rules* menu command opens the rule editor for creating and editing external rules. See Chapter 4 for a description of rule editing.

## Simulation Operations

The scene editor lets you test your simulations without using the run-time simulation driver. (That is, you don't have to quit the scene editor and go into the student simulation mode.) To get into simulation mode, click on *Simulation* in the editor operations menu.

In the simulation mode, the configuration of windows to the left of the display windows changes, and the object operations menu is replaced with the simulation operations menu.

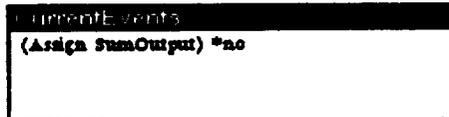
<b>Pause/UnPause</b>	<b>Clock</b>
<b>Paused</b>	<b>40</b>
<b>CurrentEvents</b>	
<b>Simulation Attributes</b>	
<b>Clock</b>	int 40026
<b>MouseX</b>	int 54
<b>MouseY</b>	int 20
<b>MouseState</b>	atom Up
<b>Object Info</b>	
<b>Left Timer</b>	
<b>Open</b>	(82 438 6 6) Open
<b>CurrentState</b>	atom Open
<b>ObjectLocX</b>	int 40
<b>ObjectLocY</b>	int 415
<b>RightOutput</b>	int 0
<b>RightInput</b>	int 100
<b>TriggerVoltage</b>	int 0
<b>LeftOutput</b>	int 0
<pre> (if ... then (schedule (setState))) *no (if ... then (setState)) *no (if ... then (Assign RightOutput) else ...) *no (if ... then (Assign LeftOutput) else ...) *no (Assign RightInput) *no (Assign LeftInput) *no (Assign TriggerVoltage) *no                     </pre>	
<b>Snap</b>	<b>Compile</b>
<b>Save State</b>	<b>Restore State</b>
<b>Pause Rules</b>	<b>Pause Attributes</b>
<b>Trace Attributes</b>	<b>System Trace</b>

**Mouse Actions in the Simulation Mode Windows**

<b>Pause/UnPause</b>
<b>Paused</b>

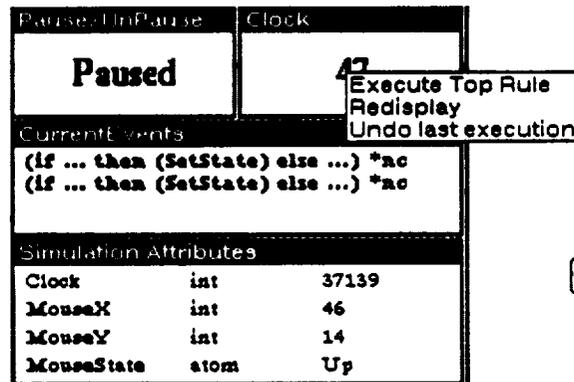
A simulation can be *Paused* or *Running*. A little window labeled *Pause/UnPause* displays a label that describes the current state of the simulation. Clicking in this window toggles the simulation between these two states.

To the right of the *Pause/UnPause* window is a window that displays the simulator clock, in seconds. See the section on *How Simulation Works*, above, for an explanation of the clock. Clicking in this window has no effect.



Just below the pause window and the clock window is the *CurrentEvents* window. This window shows a list of the rules in the *CurrentEvents* list.

The role of this list is explained above in *How Simulation Works*. In brief, the rules in *CurrentEvents* are the currently pending rules that must be carried out 'simultaneously.' The rules are shown in an abbreviated rule form in this list (See the section entitled *Viewing Simulation Data*.) Clicking with the left button on an abbreviate rule brings up the menu of rule operations, described below in *Debugging Simulations*. These rule operations are actions that authors can apply to rules to try to understand what the simulation is doing. Holding down the middle button on one of these rules will highlight the object that owns the rule in the display window. The right button anywhere in this window will bring up a special menu of *CurrentEvents Operations* that are explained in the section on *Debugging Simulations*.



Below the *CurrentEvents* window is a window labeled *SimulationAttributes*. This window displays a data view of the attributes that do not belong to any specific object.

A left mouse button click on one of these attribute data lines will pop up the *Attribute Operations* menu, which is discussed below in the section on debugging simulations. The right button menu is also useful in this window. If an attribute is not currently *traced*, then changes in the attributes value will not be posted in this data view. The *Redisplay* option in the right button menu can be used to update the attribute data display with the latest values.

Below the *Simulation Attributes* window is the set of *Object Info* windows, which are discussed earlier in this chapter. Some of the actions that can be taken in these windows are explained below in the *Simulation Debugging* section.

**Display Window  
Actions**

Clicking on an object in the scene editor's simulation mode will have different effects, depending on whether the simulation is paused or running. If the simulation is paused, clicking with the left button will select the object, just as in the object operations mode. The object will not be highlighted in the display window as it would be in the object operations mode, however. Its object data will be displayed in the *Object Info* windows at the lower left corner of the display.

If the simulation is running, then, if the object has state handles, a click on a handle puts the object into the state that corresponds with that handle. If the object has an object handle, then any rule that referred to *Mouse Down in Handle* of the object will be executed.

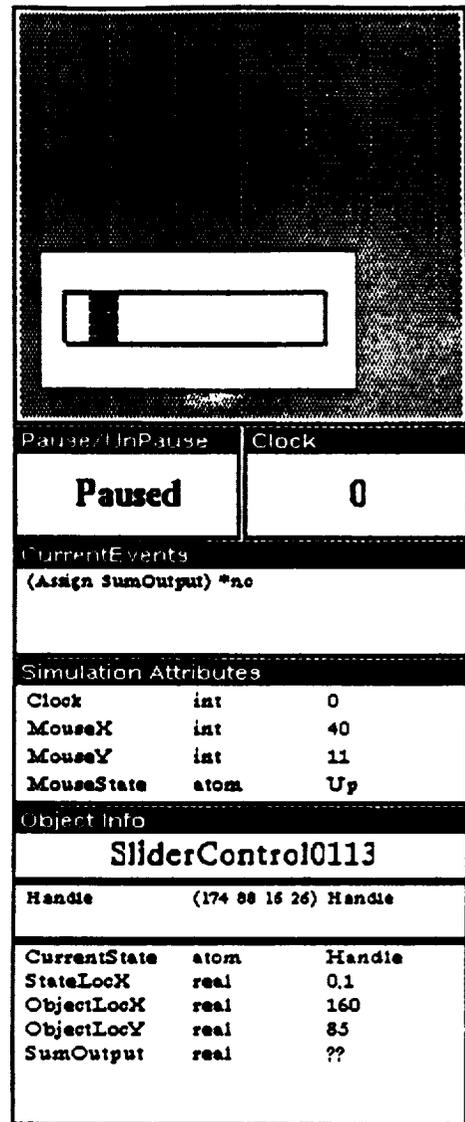
If the object has an attribute handle that has been designated as a *Probe*, then that attribute will be made the current probe attribute. If the attribute handle has not been designated as a probe, then the corresponding attribute will be connected to the currently designated probe attribute, if there is one. See the section on *Authoring and Using Test Equipment*, below.

**Snap**

The *Snap* command is used to create exact copies (called snaps) of portions of the display window. These snap windows can be positioned anywhere on the screen. A convenient place to put snap windows is on the scaled scene window in the upper left corner of the set of simulation editor windows.

In the figure at the right, a snap of a slider control object has been placed in this area. Snaps stay on the screen until the user closes them (using the *Close* command on the right button menu). When the user changes the scene in the Display Window, the snaps are still present.

Snaps are fully functional views of the snapped objects. When a switch is manipulated in the display window, any affect on the snapped object will be displayed in its snap window. Snapped controls can be manipulated as well, and will have exactly the same effects that they would if the object were manipulated when its scene is displayed in the Display Window.



When the *Snap* command is selected, the pointer will change shape (to the Interlisp Expanding Box cursor). The user can then drag out a rectangle to select one or more objects on a scene. When the mouse is released, the snapped window will appear.

Snap windows have their own right button menu. To get rid a subscene, use the *Close subscene* option from its right button menu. To put a subscene in a different location, use the *Move subscene* option from this menu.

**Compile**

When a simulation is built for use by students, all the rules in the simulation are automatically compiled to native machine code. In the scene editor environment, however, rules are not ordinarily compiled, in order to avoid long compilation delays when rules are edited.

Compiled rules run much faster than uncompiled rules. If a simulation seems to be running slowly, an author can often speed everything up by choosing the *Compile* command. After this command is selected, there will be a delay while compilation takes place. As the rules for an object are compiled, the name of that object appears in the message window.

**Save State** *Save State* makes a snapshot of the current state of the simulation, which can later be restored using *Restore State*. When you select *Save State*, a prompt in the message window asks for the name of the state to be saved. Type the name and the Return key.

**Restore State** *RestoreState* inserts a saved snapshot of a state of the simulation, one of those previously stored using the *Save State* command.

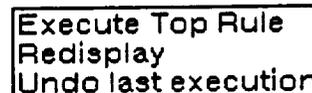


When this command is chosen, a menu of the saved states appears. Selecting the name of the desired state has the effect of restoring that state.

**Pause Rules** Developing RAPIDS II simulations, like computer programming, sometimes calls for a debugging phase. You may build a scene and find that it does not behave exactly as you expected it to. The scene editor includes a number of tools to help you figure out what you might have done wrong in constructing a scene. These tools include the ability to set and remove pauses at particular objects, the ability to artificially assign attributes certain values, and a trace facility for studying the sequence of effects during simulation on your scene.

The details of simulation debugging are treated in the *Simulation Debugging* section of this chapter, below. Briefly, both rules and attributes may be paused. When a rule is paused, the simulation stops when the rule is about to be executed.

In order to get through the paused rule, it must be executed 'by hand,' which is done using the *Execute Top Rule* command of the *Current Events Operations* menu.

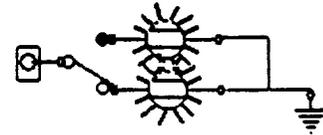


The *Pause Rules* command has the effect of marking every rule as paused, so that the simulator will pause before executing any rule.

**Pause Attributes** Attributes can also be paused. When an attribute is paused, the simulator stops immediately after changing an attribute's value. The *Pause Attributes* command makes the simulator pause after changing the value of any object attribute in the simulation.

Pauses can result in apparently incongruous simulation appearances. As the figure below demonstrates, a pause may intervene between the execution of two rules that are supposed to be simultaneous. In the figure below the lower light has been put into its 'On' state, but the upper light has not yet been put into its 'Up' state.

Pause/UnPause		Clock
<b>Paused</b>		<b>236</b>
Current Events		
(If ... then (SetState) else ...) *no		
Simulation Attributes		
Clock	int	37139
MouseX	int	46
MouseY	int	14
MouseState	atom	Up



**Trace Attributes** When an attribute is being traced, its data view is refreshed whenever the attribute changes. The *Trace Attributes* command causes all the object attributes in a simulation to be traced. Tracing attributes slightly decreases the responsiveness of a simulation, because time is spent rewriting data views.

**System Trace** This command is not yet implemented. A message to that effect appears in the Message window when the *System Trace* command is selected from the menu in the Simulation Mode.

In the future, this command will be used to trace the system attributes.

---

---

## Run-Time Corrections

If an author builds every behavior rule using menus, then the simulation will be *syntactically correct*. This means that every rule will adhere to the requirements for rule structure. Unfortunately, syntactic correctness alone will not guarantee that the simulation can run (much less that it will run as the author expects). In order to run, a simulation's behavior rules must also be *semantically correct*. Rules can fail during execution if they have undefined attributes values or if they apply operations to attributes that have values outside of the domain of the operation. This section describes how the simulator responds to undefined-attribute errors and to out-of-bounds errors.

### Undefined Attributes

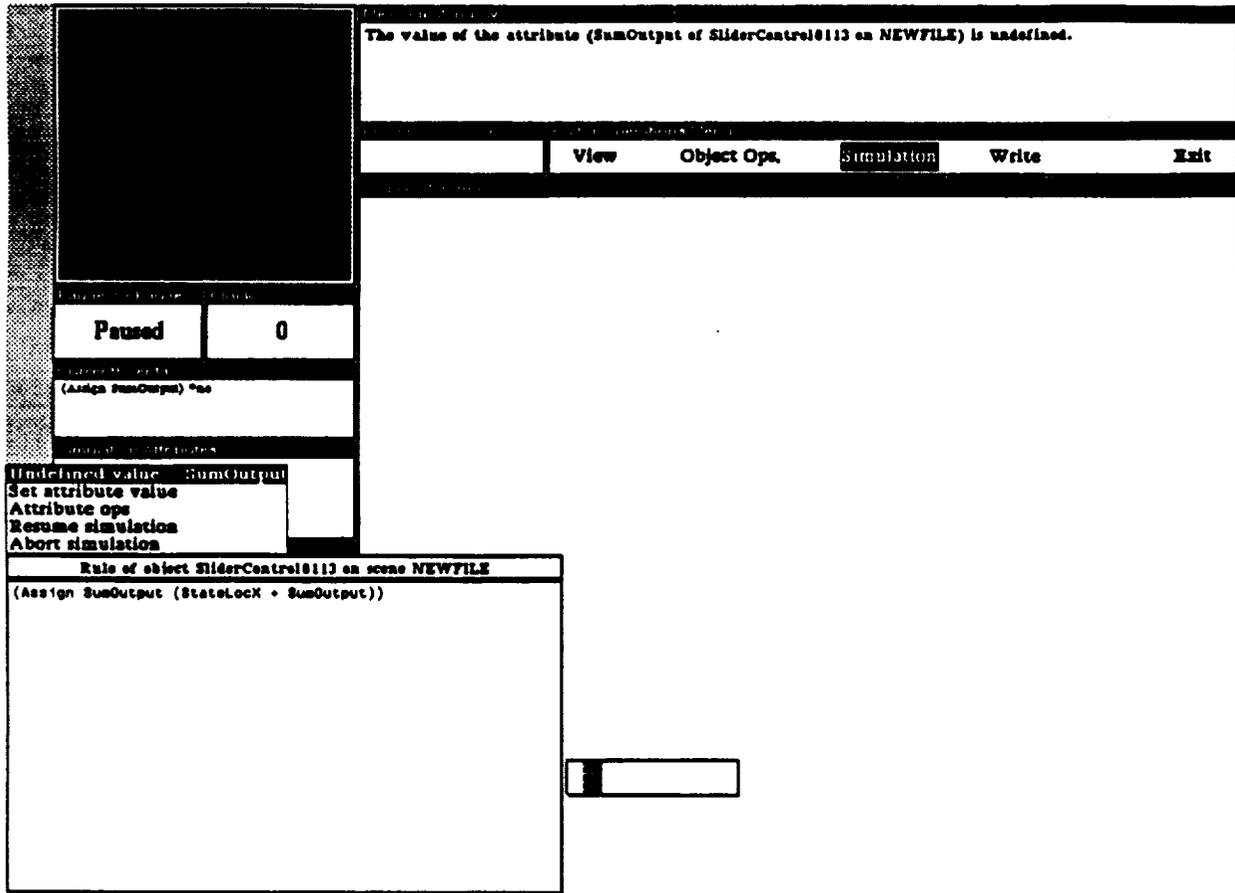
When a simulation is loaded, all of its attributes are undefined. Then, when the simulation is started (as, for example, when the author clicks on the *Pause* button in the simulation operations mode in the scene editor), all of the constant assignment rules are run once. A *constant assignment rule* is a rule that assigns to an attribute a constant value — a particular number or atom or string — rather than assigning some function of other attributes.

This initial assignment of constants will affect all the rules that refer to the attributes that just received values. Those rules will be placed on the *CurrentEvents* list, and their execution may result in the propagation of effects to still other rules. In this way, most of the attributes of a simulation will lose their undefined status and will acquire values at the time that the simulation is started.

Sometimes, however, an attribute will be referred to (by an executing rule) when it does not have a value. RAPIDS II is ordinarily able to handle this situation by postponing execution of the rule. It carries out other pending rules first. One of these rules may assign a value to the undefined attribute. This method will work if the author has designed a simulation so that certain attributes, those that function as *sources* (such as electric power supplies and hydraulic pumps), are given initial values by constant assignment rules. Other assignment rules in the simulation propagate effects from these sources through the simulation.

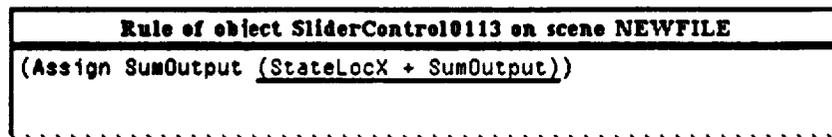
Sometimes a list of object attributes (in the *Object Info* window, for example) includes undefined attributes. The value field of such attributes will be displayed as ??.

If an author's rules don't provide a value for an attribute that serves as a source, then the simulator's strategy of postponing rules will not succeed. The simulator will detect a *semantic error*, called *undefined attribute*. When this happens, the simulator pauses and it re-posts the rule to the *Current Events* list. It then opens a special *Undefined Attribute Window*. See the figure below.



The window at the lower left corner of this figure is an *undefined attribute window*. It presents the text of the rule that encountered the undefined value in an expanded form. If the author holds down the middle button in the window's title bar (the area that says 'Rule of object SliderControl0113 on scene NEWFILE' in the above figure), then the object that owns the rule will be highlighted in the display window. (If the object is not in the display window, then a scaled version of the scene that it is in will appear in the *Scaled Scene Window* at the top left corner of the screen, and the object will be highlighted there.)

The *undefined attribute window* is a rule evaluation window, just like the expanded rule discussed above in the subsection on *Rule Data Views* in the section on *Viewing Simulation Data*. This means that you can click on portions of the rule to underline rule segments.



These segments will be evaluated and the results printed in the Message Window above.

```

Message Window
SumOutput — undefined
StateLock — 0.1

Value of rule segment: undefined
    
```

**The Undefined value Menu**

A menu at the top of the *undefined attribute window* gives a number of options for dealing with undefined attribute error. The title of this menu begins with 'Undefined value' followed by the name of the attribute that is undefined. In this example 'SumOutput' is not defined.

```

Undefined value -- SumOutput
Set attribute value
Attribute ops
Resume simulation
Abort simulation
    
```

If the author chooses *Set attribute value*, the first option in the menu, then the message window will prompt for a value that should be used in this execution of the simulation. (See the figure below.)

```

Message Window
The value of the attribute (SumOutput of SliderControl0113 on NEWFILE) is undefined.
Type the value to be assigned to the attribute SumOutput.
The value must be a number. >>
    
```

Note that the value entered will not be permanently assigned to the attribute. That is, the next time the simulator is initialized, the same problem will occur again. Nonetheless, this is often a good choice for an author to make in order to test the behavior of the simulation when a certain value is used for the simulation. If the simulation behaves appropriately, the author can later add a constant assignment rule that gives the value to the attribute.

When the author sets an attribute value, the scene editor will offer to build a constant assignment rule that gives that value to the attribute. If the author agrees, that value will serve as an initialization value for the attribute when the simulation is run again. (If automatic rule creation is carried out during a scene editor session, the author must be sure to *Write* the changed file in order to save the change.)

The second choice on the menu, *Attribute ops*, gives authors a standard menu of attribute operations. This menu applies to the attribute named in the menu title — in this case, 'SumOutput.' Two of these options, *Who Affects Me* and *Whom Do I Affect* are described above in the section on *Viewing Simulation Data*. The other options are discussed below in the *Simulation Debugging* section.

```

Edit Pause Condition
Inspect
Object Bundle
Pause On/Off
Set
Trace On/Off
Who Affects Me
Whom Do I Affect
    
```

The third option on the *Undefined value* menu is *Resume simulation*. This command closes the undefined attribute window and attempts to continue with the simulation. Since the simulation will resume with the rule that was paused, this choice will be successful only if the author has taken steps to give the undefined attribute a value while the simulation was paused. This can be done by using the *Set attribute value* option in the menu, or by using *Set* from the *Attribute Operations* menu (shown immediately above).

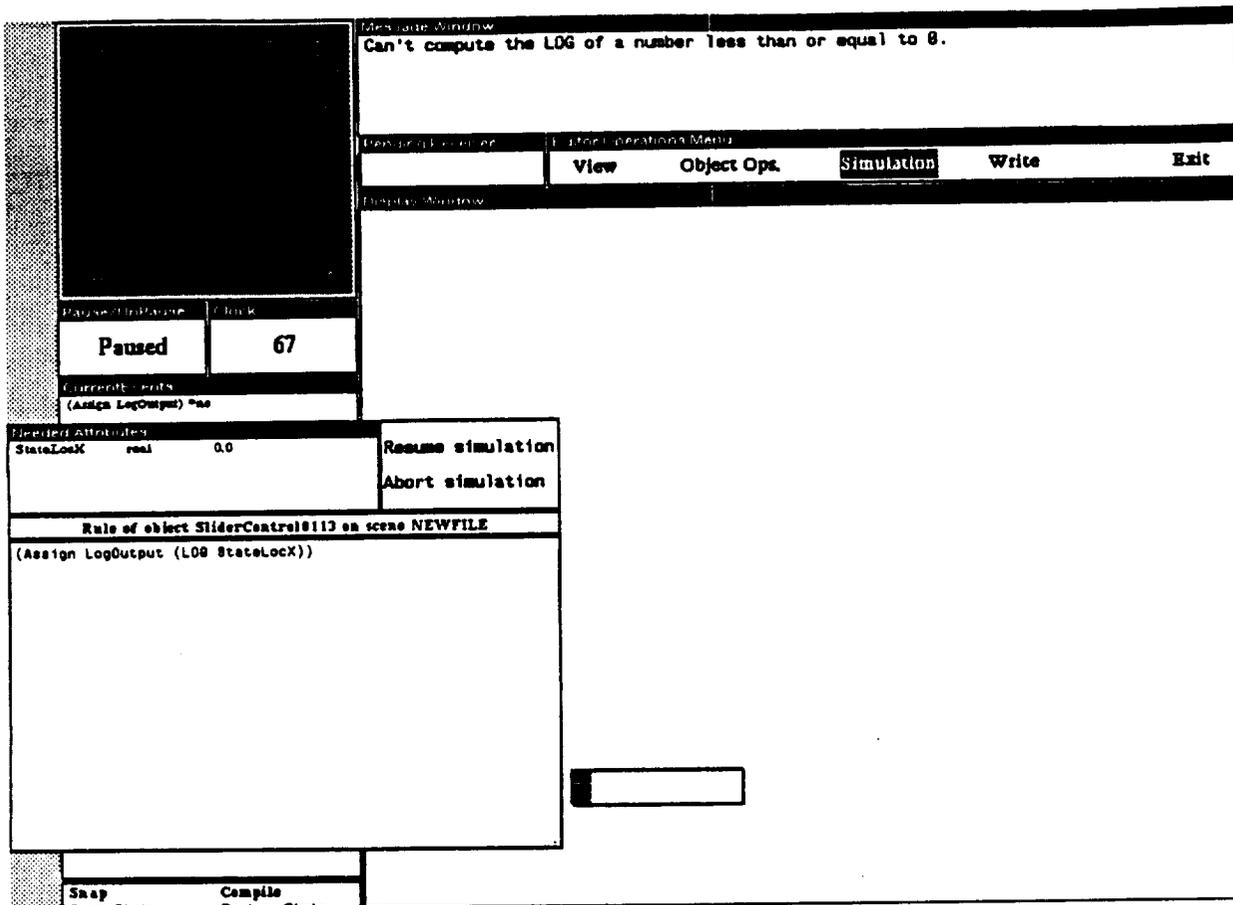
The fourth choice on the *Undefined value* menu is *Abort simulation*. This command closes the undefined attribute window but does not continue with the simulation. This is an appropriate choice if the author wants to carry out actions using the object operations or simulation operations menus, rather than to continue simulating at this time.

**Out of Bounds Values**

The second kind of semantic error that RAPIDS II can detect is *out of bounds* errors. Rules can include calls to arithmetic functions that appropriately apply to only a limited domain of values. Here are the domain restrictions that currently apply to such function calls in these rules.

Function	Domain Error Condition
LOG n	n ≤ 0
SQRT n	n < 0
x/n	n = 0
x MODULO n	n = 0
ANTILOG n	n > 87
SetState x n	n not in the set of states of x

If any of these domain error conditions are detected when the simulator executes a rule, the simulator pauses and it re-posts the rule to the CurrentEvents list. It then opens the *Out of Bounds Attributes* windows, as shown in the figure below.



The largest of the new windows, which is at the bottom of the group of windows, is an executable rule window, similar to the one that is opened in the case of an undefined attribute error. It presents the text of the rule that encountered the out of bounds value in an expanded form. If the author holds down the middle button in the window's title bar (the area that says 'Rule of object SliderControl10113 on scene NEWFILE in the above figure), then the object that owns the rule will be highlighted in the display window. (If the object is not in the display window, then a scaled version of the scene that it is in will appear in the *Scaled Scene Window* at the top left corner of the screen, and the object will be highlighted there.)

You can click on rule segments in the executable rule window and they will be underlined in the expanded rule view.

<b>Needed Attributes</b>		Resume simulation Abort simulation
StateLocX	real 0.0	
<b>Rule of object SliderControl0113 on scene NEWFILE</b>		
<b>(Assign LogOutput (LOG StateLocX))</b>		

At the same time, the selected segment will be evaluated, and the results of its evaluation will appear in the message window, as shown below.

<b>Message Window</b>
StateLocX -- 0.0
<b>Value of rule segment:</b> Can't compute the LOG of a number less than or equal to 0.

When an out-of-bounds error is encountered, the message window describes the nature of the domain constraint violation.

**Needed Attributes**

A window above the executable rule window is the *Needed Attributes* window. It lists the attributes with values that violate the domain constraints of the operations that are applied to them in the rule. In the case of this example, only one attribute is a problem, *StateLocX*.

If the author clicks on an attribute in this list, the menu of attribute operations pops up. The same attribute operations discussed in the sections *Viewing Simulation Data* and *Simulation Debugging* are available.

<ul style="list-style-type: none"> <li>Edit Pause Condition</li> <li>Inspect</li> <li>Object Bundle</li> <li>Pause On/Off</li> <li>Set</li> <li>Trace On/Off</li> <li>Who Affects Me</li> <li>Whom Do I Affect</li> </ul>	<b>67</b>	Resume simulation Abort simulation
	StateLocX 0.0	
<b>Rule of object SliderControl0113 on scene NEWFILE</b>		
<b>(Assign LogOutput (LOG StateLocX))</b>		

One authoring strategy is to use the menu to set an appropriate value for the attribute and to then continue, testing that the corrected value works. If it does, the author can edit rules to ensure that the attribute will have an appropriate value in the future.

Another strategy for dealing with Out of Bounds errors is to change the rule so that it tests for the out-of-bounds condition and has a different effect when that condition holds true.

**Resume/Abort Simulation**

To the right of the *Needed Attributes* window is a menu with two options, *Resume simulation* and *Abort simulation*. If *Resume simulation* is selected, the *Out of Bounds Attributes* windows close and the simulator resumes its work. It again attempts to execute the rule that made the simulation pause. If *Abort simulation* is selected, the windows close but the simulation does not continue.

**Other Rule Errors**

Undefined attributes and out-of-bounds attribute values are not the only kinds of errors that can occur in a simulation's rules. They are the two types of semantic errors that the simulator knows how to detect.

The most common errors are those that tell the simulation to behave in ways that are different from the way the actual device behaves. Sometimes these authoring errors are difficult to detect. The next section presents the tools that help authors find and fix such problems.

## Simulation Debugging

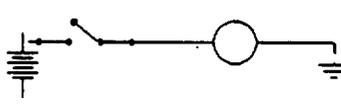
Complex simulations don't always work exactly as their authors expected when they are first tested. The *Simulation Operations* mode of the scene editor has many features that assist the simulation debugging process.

### Debugging an Example Simulation

Consider the simple circuit shown below. A battery is connected to a light by a switch. The author clicks on *Paused* to start the simulation.

File Edit Simulation Menu
View Object Ops **Simulation** Write Exit

<b>Paused</b>	<b>0</b>
(Assign ValueOut) *no	
(Assign ValueOut) *no	
Simulation Attributes	
Clock	int 0
MoneyK	int 194
MoneyY	int 18
MoneyRate	atom Up
Object Info	
<b>battery</b>	
CurrentState	atom 41
ObjectLock	int 158
ObjectLockY	int 308
ValueOut	int ??
(Assign ValueOut) *no	
Snap	Compile



When the author clicks the switch into its closed position, the light fails to come on! (See the figure below.) The simulation needs to be debugged.

Request to Pause		Clock
<b>Running</b>		<b>19</b>
Current Events		
Simulation Attributes		
Clock	int	0
MessageK	int	194
MessageY	int	18
MessageRate	atom	Up
Object Info		
<b>battery</b>		
CurrentState	atom	on
ObjectLocX	int	138
ObjectLocY	int	308
Voltage	int	77
(Assign Values) ...		
Save	Compile	

The simulation is running, but the light is not in the correct state. There are a number of strategies that the author can apply to determine how the simulation should be revised to solve this problem.

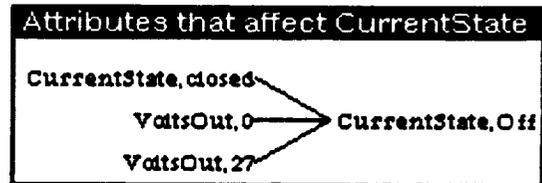
First, clicking on the light in the Display Window puts the light's object data into the *Object Info* window.

Object Info		
<b>light</b>		
CurrentState	atom	Off
ObjectLocX	int	296
ObjectLocY	int	321
(if ... then (if ... then (SetState) else ...) else ...)		

Then, clicking on the *CurrentState* attribute in the *light*'s object attribute list brings up the *Attribute Operations* menu. The author might select *Who Affects Me* in order to find out what attributes in the simulation have control of the state of the light.

Object Info		Edit Pause Condition	
		Inspect	
		Pause On/Off	
		Set	
		Trace On/Off	
		Who Affects Me	
		Whom Do I Affect	
<b>CurrentState</b>			
ObjectLook	int	296	
ObjectLocY	int	321	
(if ... then (if ... then (SetState) else ...) else ...)			
Snap		Compile	
Save State		Restore State	
Pause Rules		Pause Attributes	
Trace Attributes		System Trace	

A window that shows the attributes that affect the *CurrentState* attribute of the light opens on the screen.



The author can see that the state of the light depends on three other attributes: the state of the switch, and the *VoltsOut* attributes of two objects. (To find out what objects are the owners of these attributes, the author could middle-mouse click on any attribute name in the *Affects* window and the object would be highlighted in the Display Window.)

The nature of the problem is still not clear, so the author clicks on the abbreviated rule in *Object Info* in order to bring up the *Rule Operations* menu.

Object Info		
light		
CurrentState	atom	Off
ObjectLocX	int	296
ObjectLocY	int	321
Edit Pause Condition Expand Inspect Pause On/Off		
(f... men, ll... men (Object)		
Snap	Compile	
Save State	Restore State	
Pause Rules	Pause Attributes	
Trace Attributes	System Trace	

Clicking on *Expand* brings up an executable rule window for the rule, as in the figure below. The rule specifies that the light should go on if the battery's *VoltsOut* attribute is 28 and the ground's *VoltsOut* attribute is 0 and the switch's *CurrentState* attribute is 'closed.

The author decides to check on the status of each of these preconditions for the light coming on. One way to carry out this check is to select the corresponding rule segments in the expanded rule window. The rule segment will be evaluated and the result printed in the message window.

(VoltsOut of battery on MOJO) == 27  
 Value of rule segment: False

View
Object Ops
Simulation
Write
Exit

**Running**

**1673**

Global Attributes	
Clock	int
Message	int
MessageY	int
MessageState	enum

with index that affects current state	
CurrentState, enum	CurrentState, Out
VoltsOut, int	VoltsOut, 27

**light**

CurrentState	enum	Off
ObjectLock	int	296
ObjectLockY	int	

**Rule of object light on scene MOJO**

```

                (If ((VoltsOut of battery on MOJO) = 28)
                AND
                ((VoltsOut of ground on MOJO) = 0))
                then (If ((CurrentState of switch on MOJO) SameAs 'closed)
                then (SetState 'On)
                else (SetState 'Off))
                else (SetState 'Off))
            
```

**light**

**Snap**

Selecting the first rule segment

((VoltsOut of battery on MOJO) = 28)  
 produces an evaluation result of *False*. (See the message window above.) The battery's *VoltsOut* attribute has the value 27, not 28 as the rule requires.

The author now checks on the battery by clicking on the battery in the Display Window. The *Object Info* windows change to display an object data view of the battery. Looking over the object attribute list, it is clear that the *VoltsOut* attribute does indeed have the value 27.

To find out how the battery got this value, the author decides to look at the battery's only rule. Clicking on the abbreviated rule brings up the *Rule Operations* menu.

Object Info		
battery		
CurrentState	atom	s1
ObjectLocX	int	153
ObjectLocY	int	308
VoltsOut	int	27

(Assign VoltsOut) *rc...	Edit Pause Condition Expand Inspect Pause On/Off
--------------------------	---

Snap	Compile
Save State	Restore State
Pause Rules	Pause Attributes
Trace Attributes	System Trace

Choosing *Expand* from this menu brings up an expanded rule window for the battery's rule:

Rule of object battery on scene MOJO
(Assign VoltsOut 27)

At this point the author can clearly see that the problem is an erroneous constant assignment rule for the battery's *VoltsOut* attribute. The most direct route to solving the problem now is to edit the rule in the rule editor, changing the 27 to 28. If the problem were less clear, it might be desirable to use the *Set* feature of the *Attribute Operations* menu to test the simulation's behavior with the value set to 28. After the correct behavior was observed, the author would use the rule editor to change the above constant assignment rule.

**Attribute and Rule Operations**

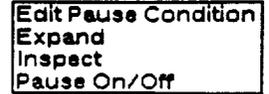
In the remainder of this section, the major debugging features of the scene editor are presented. Two of the most important ways of accessing the debugging features require making use of the *Attribute Operations* menu and the *Rule Operations* menu.

The attribute operations menu (shown at right) is accessed by clicking the left mouse button on attribute data. Attribute data can be found in the following windows:

Edit Pause Condition Inspect Object Bundle Pause On/Off Set Trace On/Off Who Affects Me Whom Do I Affect
---

The object attributes window (in the *Object Info* windows)  
 The Affects windows  
 The Out of Bounds Attributes window  
 The simulation attributes window

The rule operations menu (shown at right) is accessed by clicking the left mouse button on rule data. Rule data can be found in the following windows:



The Object Rules window (in the *Object Info* windows)  
 The Current Events window

The sections below describe the features offered by these menus.

**Pauses**

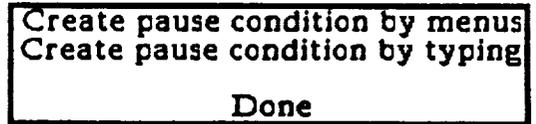
Authors can instruct the simulator to pause under specified conditions. During a pause, the *CurrentEvents* list can be inspected, object attributes and simulation attributes can be examined, and individual rules can be executed. Pauses can be associated with attributes or with rules.

Pausing a rule means that whenever the rule is about to be executed, the simulation pauses. Pausing an attribute means that just after the attribute's value changes, the simulation is paused. The simulation pauses just as it would if the author had clicked on the *Running* button to pause the simulation. After browsing through the simulation data, the author can resume the simulation by clicking the same button, which reads *Paused*.

Authors can edit a *Pause Condition* for any attribute or rule. A pause condition determines whether or not the 'paused' rule or attribute will actually make the simulation stop running. If an author does not create a pause condition for a rule or an attribute, then its pause condition is considered to be 'True.' This means that if pausing is turned on, the simulator will stop running when that rule is to be executed or when that attribute is about to be assigned a value.

**Edit Pause Condition**

Whether an author creates a pause condition for an attribute or for a rule, after choosing the *Edit Pause Condition* command, a new menu appears at the lower left corner of the screen.



If the option *Create pause condition by menus* is selected, then a series of menu choices are presented to help the author build an expression that will determine whether the attribute or rule will be paused if pausing is turned on.

The screenshot displays a complex simulation diagram of a jet engine starter system. The diagram includes components such as 'Timer', 'Igniter', 'Starter Valve', 'L. Engine', 'R. Engine', and 'Generator'. It shows the electrical and mechanical connections between these parts, including 'Left Starter' and 'Right Starter' units.

On the left side, there is a 'Paused' status window showing '389' and a 'Current State' section with a rule: `(If ... then (Schedule (SelfState)))`. Below this is a table of simulation attributes:

Clock	int	0
MouseX	int	174
MouseY	int	20
MouseState	atom	Up

The 'Object Info' window for the 'Left Start Button' shows its 'Pressed' state and a table of attributes:

CurrentState	atom	Pressed
ObjectLock	int	20
ObjectLocY	int	538
OutputVoltage	int	28
InputVoltage	int	28

The 'SEdit' window is open, showing a menu-based editor for the pause condition. The menu options include: 'Types of Conditions', 'Comparison', and 'Mouse'. The current selection is 'Mouse', and the condition expression is `((PauseCondition))`. The title bar of the SEdit window reads: 'SEdit \EditableRule\ Package: INTERLISP'. At the bottom, a status bar shows: 'Name: NEWSTARTER File being edited: (OSK)\LISPFILES\SOURCE\RAPIDS II\NEWSTARTER.L1 Date Written: 15-Feb-90 15:11:00'.

The menu-based pause-condition editor lets you build a conditional expression similar to the condition expressions that can be created to fill the <condition> part of a rule. This condition expression is understood to refer implicitly to the object to which the attribute or rule belongs. The figure below shows the appearance of the SEdit window just after the author has completed a condition expression for the pause condition for a rule of a push-button object.

The close-up screenshot shows the SEdit window with the following text in the main editing area:

```
SEdit \EditableRule\ Package: INTERLISP
(CurrentState SameAs 'Pressed')
```

The title bar of the window is 'SEdit \EditableRule\ Package: INTERLISP'.

The effect of this condition expression is to pause the rule just before its execution if the *CurrentState* attribute of the object is 'Pressed' and if pausing has been turned on. The pause will take place whether pausing was set for the individual attribute or rule — using one of the above menus — or whether it was set by the one of the two global commands, *Pause Rules* or *Pause Attributes*.

**Pause On/Off**

Choosing Pause On/Off has the effect of marking an individual attribute or rule for pausing. The simulation will actually pause only if the Pause Condition of the rule or attribute is true. (The default Pause Condition is True.) When rules or attributes are paused individually, they appear in bold face in any visible rules lists of abbreviated rules, as in the figure at the left below.

Pause/UnPause	Clock	
<b>Running</b>	<b>319</b>	
<b>CurrentEvents</b>		
<b>Simulation Attributes</b>		
<b>Clock</b>	int	<b>0</b>
<b>MouseX</b>	int	<b>174</b>
<b>MouseY</b>	int	<b>20</b>
<b>MouseState</b>	atom	<b>Up</b>
<b>Object Info</b>		
<b>Left Start Button</b>		
<b>Pressed</b>	<b>(32 838 34 27) Pressed</b>	
<b>CurrentState</b>	atom	<b>Pressed</b>
<b>ObjectLocX</b>	int	<b>20</b>
<b>ObjectLocY</b>	int	<b>538</b>
<b>OutputVoltage</b>	int	<b>28</b>
<b>InputVoltage</b>	int	<b>28</b>
<b>(if ... then (Schedule (SetState))) (if ... then (Assign OutputVoltage) else ...) (Assign InputVoltage)</b>		
<b>Snap</b>	<b>Compile</b>	
<b>Save State</b>	<b>Restore State</b>	
<b>Pause Rules</b>	<b>Pause Attributes</b>	
<b>Trace Attributes</b>	<b>System Trace</b>	

Pause/UnPause	Clock	
<b>Paused</b>	<b>387</b>	
<b>CurrentEvents</b>		
<b>(if ... then (Schedule (SetState))) (if ... then (Assign OutputVoltage) else ...)</b>		
<b>Simulation Attributes</b>		
<b>Clock</b>	int	<b>0</b>
<b>MouseX</b>	int	<b>174</b>
<b>MouseY</b>	int	<b>20</b>
<b>MouseState</b>	atom	<b>Up</b>
<b>Object Info</b>		
<b>Left Start Button</b>		
<b>Pressed</b>	<b>(32 838 34 27) Pressed</b>	
<b>CurrentState</b>	atom	<b>Pressed</b>
<b>ObjectLocX</b>	int	<b>20</b>
<b>ObjectLocY</b>	int	<b>538</b>
<b>OutputVoltage</b>	int	<b>28</b>
<b>InputVoltage</b>	int	<b>28</b>
<b>(if ... then (Schedule (SetState))) (if ... then (Assign OutputVoltage) else ...) (Assign InputVoltage)</b>		
<b>Snap</b>	<b>Compile</b>	
<b>Save State</b>	<b>Restore State</b>	
<b>Pause Rules</b>	<b>Pause Attributes</b>	
<b>Trace Attributes</b>	<b>System Trace</b>	

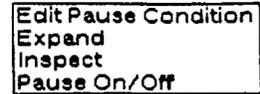
When the simulation pauses on encountering such a rule, as in the Figure at the right above, the *CurrentEvents* list will show the paused rule at the top of the list. Naturally, the rule data will also appear in bold face in this window.

**Inspect**

The Inspect option can also be found on both the Attribute Operations menu and the Rule Operations menu. This option opens an Interlisp-D data structure inspector for the selected data (the selected attribute or rule). This inspector is really a Lisp programmer's tool, rather than a simulation developer's tool. We recommend that you avoid using this feature, as it is both confusing and dangerous. Documentation on the inspector can be found in the Xerox or Envoy Interlisp-D documentation.

**Expand**

The Rule Operations menu's *Expand* feature opens an expanded structural view of a rule. Selecting rule elements in this window results in the evaluation of those elements, and the evaluation results are presented in the message window.



See the section *Viewing Simulation Data* for more on this feature.

**Object Bundle**

When an attribute data view that is not in a set of *Object Info* windows is selected, the command *Object Bundle* will be included in the Attribute Operations menu. This command will open a new set of *Object Info* windows for the object that owns the selected attribute. Authors can open a large number of such window sets to view the data of many objects at the same time.

It is also possible to open an object bundle from the Rule Operations menu, when you bring up this menu within the *CurrentEvents* window.

**Set**

The *Set* command in the Attribute Operations menu can be used to give a particular attribute a certain value. This command is useful for quickly testing the effects of certain values in the simulation.

**Trace On/Off**

The Attribute Operations menu's *Trace On/Off* command lets authors toggle the trace status of attributes. If an attribute is being traced, its visible data views will be updated as the simulation changes the attribute's values. If you want to know the current value of an untraced attribute during a simulation, you must use the right button command *Redisplay* in the attribute data window.

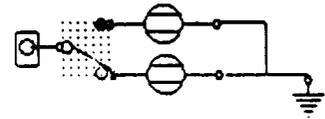
If an attribute is being traced, its data view in the attribute list of *Object Info* windows is overlaid with a light gray pattern. In the figure at the right, the *CurrentState* attribute is marked as having Tracing turned on.

Object Info		
Left Engine		
CurrentState	atom	Off
ObjectLocX	int	246
ObjectLocY	int	359
OutputVoltage	int	0

**The Affects Commands**

The last two commands on the Attribute Operations menu are *Who Affects Me* and *Whom Do I Affect*. These commands open a window that graphs the flow of effects among attributes.

Object Info		
Selector		
Up	(92 368 10 10)	Up
Down	(92 333 10 10)	Down
CurrentState	atom	Down
ObjectLookX	int	75
ObjectLookY	int	333
Attribute effects from CurrentState		
CurrentState, Down	CurrentState, Off	CurrentState, Off



It is possible to build a perfectly legal simulation that has circular effects. If an attribute name appears more than once in an Affects window, it will be boxed.

## Multi-Scene Simulations

Many complex simulations require a number of scenes. In RAPIDS II, the scenes of a simulation are organized hierarchically. You should organize the scenes to minimize the number of required scene changes. It often helps to include functionally related components on the same scene.

A parent scene in an RAPIDS II simulation is one that has one or more objects that represent or stand for other scenes. Clicking on such an object during a simulation will cause the display window to replace the current scene with the scene that the object represents. We call such objects *scene icons*. Any specific object can be made a scene icon.

To make an unconnected object into a scene icon, use the object operations menu to select it and then choose the *Open* command. You will be asked to click the left button if you want to type in the name of the scene that the object should represent. If you click the left button and type in a scene name, then the selected object will be linked to the named scene. (That scene will then appear in the display window for scene editing.) If you click the right button, a new scene will be created and given the name of the specific object.

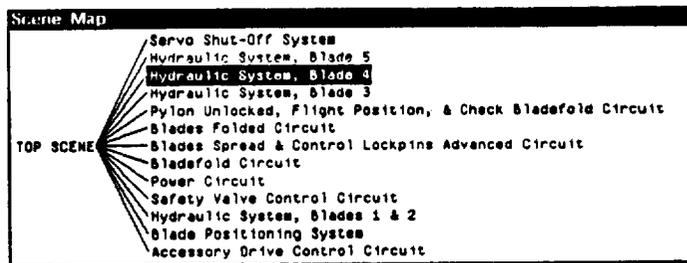
The simplest structure for a multi-scene simulation is to have one parent scene with a scene icon for every other scene in the simulation. This is a nearly flat scene structure. The parent scene, in a sense, merely replicates the scene map.

In more complex scene hierarchies, some of the scenes that can be accessed from the highest parent scene have scene icons themselves. Parent scenes are not constrained to contain only scene icons. They can have ordinary objects as well.

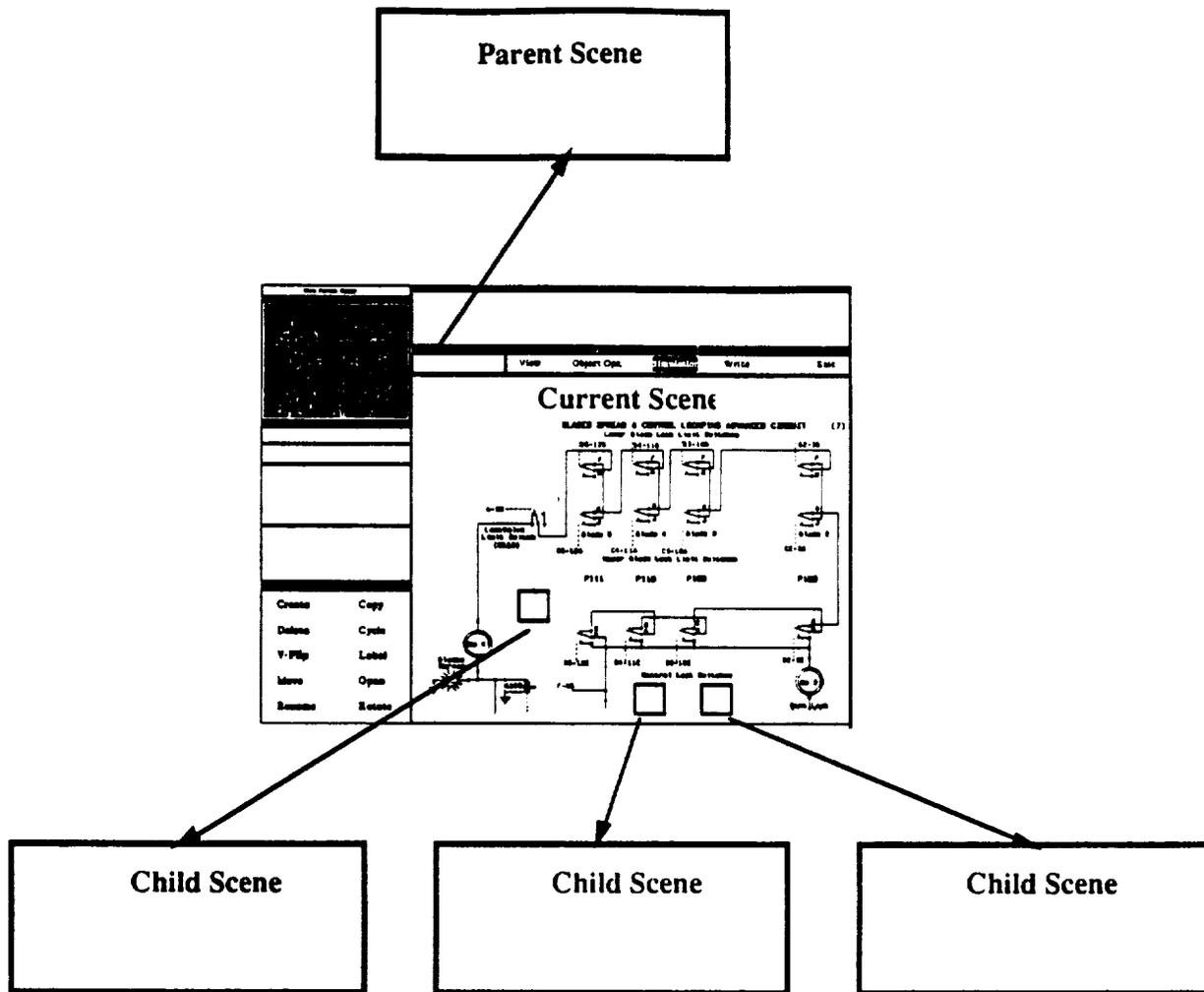
If an ordinary object is made a scene icon (by *Opening* it), it does not lose its active characteristics. Values are still propagated through its attributes and its rules will be invoked normally. Users can manipulate controls (switches) that are turned into screen icons, using normal mouse actions. Double-clicking on a screen icon only means to go to the scene the icon represents; it does not also mean that the switch should be manipulated.

**Scene Navigation**

In the scene editor you navigate using scene icons and the *View* menu command. This contrasts with navigation during student simulation, which lets students use the automatic *scene map* to go to any of the scenes in a simulation hierarchy in one step.



*A Scene Map*



*Display Window Navigation in the Scene Editor*

When you are using the scene editor, there is no way to bring up the scene map. All navigation is carried out using scene icons and the *View* menu command. Clicking on a scene icon brings up the scene it represents; clicking on *View* brings up the parent of the scene in the display window.

When a scene is brought up during an editing session there may be a significant delay while the editor reads data from your disk. Ordinarily, the first time delay is much greater than scene-changing delays during run-time simulation. Subsequent access to the scene during an editing session also will be quicker, in most cases.

**The Scaled Scene Window**

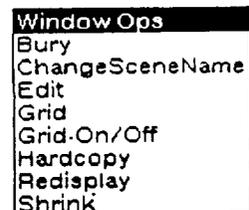
The *Scaled Scene Window*, the small window at the upper left corner of the scene editor, displays a miniature copy of a scene. As far as scene navigation goes, it is controlled a lot like the display window. If you click on a scene icon in the parent scene window, the contents of that window will change to those of the scene represented by the scene icon. By clicking on the menu button just above the window (labeled *View Parent Scene*), you can display the parent scene of the scene currently displayed there.

In addition to controlling the parent scene window this way, authors automatically change it when they change the scene in the display window. When the scene displayed there changes, the parent scene window shows the parent scene of the scene in the display window. (If there is no parent scene for the scene in the display window — that is, if the top scene is displayed — then the parent scene window will display a gray background.)

The parent scene window is sometimes displayed inverted — black for white, as in a photographic negative. This means that the miniature image of the scene is not entirely accurate because it hasn't yet been updated to reflect recent changes you made to the scene. To update the miniature view of the scene, simply put the mouse pointer in the parent scene window and wait a moment for it to be redrawn.

## Display-Window Operations

The right mouse button brings up a window operations menu, similar to the one discussed in Chapter 3.



**Bury**

This command works just like the *Bury* command of the generic editor. It brings up the windows that are hidden below the scene editor windows and puts them on top, where you can manipulate them. To restore a scene editor window to the fore, click on any portion of the window.

**Change Scene Name**

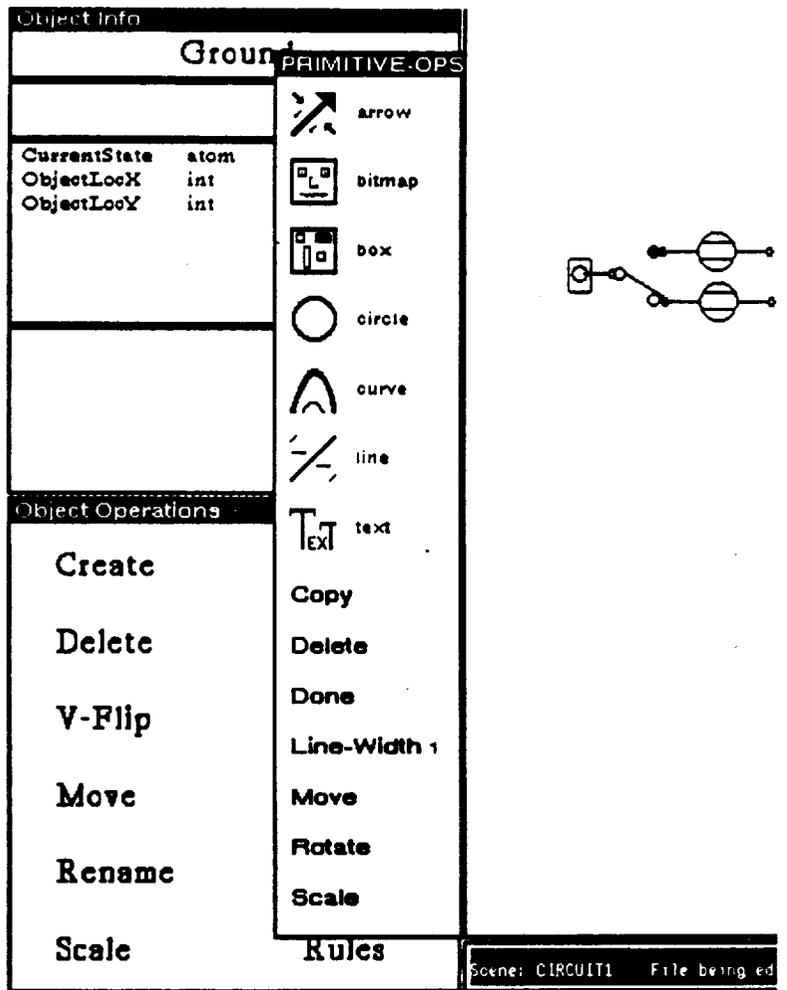
The *Change Scene Name* command on the right button menu lets you change the name of the scene currently displayed in the display window. The message window prompts you for a new name. Scene names can have spaces in them. A scene name is not necessarily the same as the name of the disk file that represents the scene. The name first given to a scene (or something close to it) will be used as the file name. Later changes to the scene name will not make the file name change. It is an error to change the name of any scene file (for example, using the Filebrowser utility) if the scene it contains is referred to by any other scene.

It is usually best to change the names of any scenes that are created automatically when you open specific objects without specifying a name.

**Edit**

Sometimes you want a scene to contain elements that don't really have to be specific objects — they are primarily decorative rather than functional from the point of view of the simulation. Examples include fixed mechanical elements such as brackets and fasteners, and labels that apply to the scene as a whole rather than to particular objects on the scene.

The scene editor lets you draw such graphical elements directly on the scene that needs them. When you choose *Edit* from the window operations (right button) menu, a palette of drawing tools appears at the left edge of the display window. If you have already used the generic editor drawing tools, this tool menu should look familiar. It is the standard *primitive operations menu*. You can use it to draw graphic elements and to add scene-level textual elements. (See Chapter 3 for a detailed description of the use of this menu.) The menu is shown in the figure below, overlaying the object operations menu.



- Grid**                   The *Grid* command has the same effect as in the generic editor. It allows you to specify the grid intervals (in pixels) for the purposes of positioning objects. It is often helpful to use an appropriate grid size in the scene editor when laying out a scene. An active grid allows you to place objects only at grid locations, not at one of the pixels in between. For example, many of the objects in the Bladefold library were drawn using a grid size of 6. You may find it easier to line up specific objects with each other in your scene if you choose the same grid size when you use this library
- Grid - On/Off**       As in the generic editor, this feature toggles the visual appearance of a grid in the display window.
- Hardcopy**            If your computer has been configured with the appropriate printer drivers and is connected to a printer, the *Hardcopy* command will print a copy of the display window on your connected printer.
- Redisplay**           The *Redisplay* command repaints the display window. On rare occasions, graphic operations may leave bits of meaningless garbage on the screen that can't be selected or otherwise dealt with normally. These graphic artifacts can be removed by repainting the scene using *Redisplay*.
- Shrink**                You can suspend a scene editing session by using the *Shrink* command. The scene editor windows will shrink to a tiny window that displays only the currently selected object.
- Always do a *Write* before shrinking your scene editing session and going on to something else. Otherwise, if something damages your Lisp environment, anything that you haven't saved may be lost.

## Using Attribute Handles

---

An *Attribute Handle* is a region that is associated with particular a attribute of an object. Attributes don't have to have handles, and most don't. There are two major uses for attribute handles

- connecting attributes while authoring scenes, and
- creating test equipment (such as multimeters, pressure guages, etc.

This chapter describes how attribute handles are used to make connections and test equipment. If your simulations don't require test equipment, and you don't plan on connecting attributes using the mouse, you don't have to read this chapter. The first part of the chapter describes how attributes can be connected using attribute handles. The second part of the chapter describes test equipment authoring.

### Connecting Attributes — Overview

In RAPIDS II, authors must explicitly connect objects to each other wherever they want values to be passed. Any values that are associated with objects are found in the *attributes* of the object. 'Making a connection' between two objects means ensuring that a value will flow from one object to another. If an author wants the value of a power supply's *OutputVoltage* attribute to flow to the *InputVoltage* attribute of a power switch, then he or she must ensure that the switch's *InputVoltage* is *assigned* the *OutputVoltage* of the power supply.

There are three different ways to connect attributes in RAPIDS II.

- Write a rule that assigns the value of one attribute to another
- Use the middle mouse button to link the attribute handles of objects
- Use the *Make Connection* option of the rule editor

No matter which of these connection methods is used by the author, the underlying effect is the same. A new rule is created that has the form (Assign *InputVoltage* of *PowerSwitch* *OutputVoltage* of *PowerSupply*). Every connection is underlyingly represented as an assignment rule.

---

In this chapter, the latter two methods for making connections are outlined and demonstrated in the context of an elementary simulation, which is described below. The two shortcut approaches to making connections are:

- (1) linking attributes using the middle button of the mouse, and
- (2) using *Make Connection* in the rule editor.

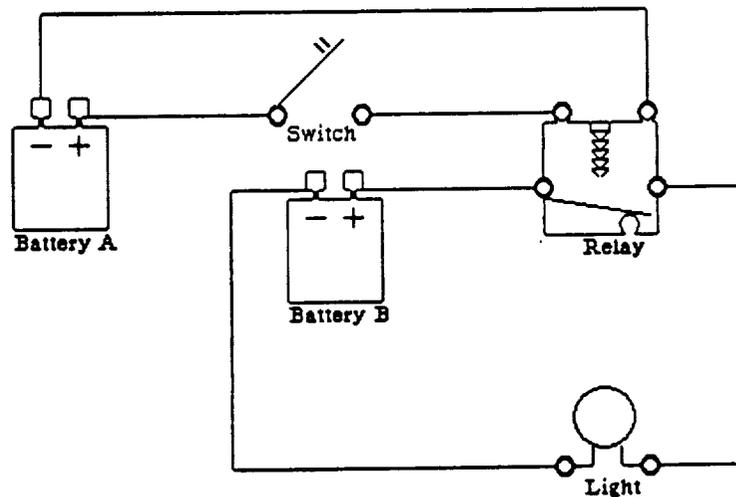
Chapter 4, on rule editing, describes the creation of assignment rules using the ordinary features of the rule editor.

Here we describe only the simple case of direct connections between the attributes of two objects. Keep in mind that it is sometimes necessary to link attributes in more complicated ways. For example, *conditional assignments* of values are sometimes required. (That is, an assignment is to take place only if some condition holds true.) These types of connections must always be authored by writing a rule that prescribes the flow of effects.

## An Example Simulation: A Simple Electrical Relay

Imagine that you want to create a course to teach elementary electrical component functionality. In such a course, you might have a scene that demonstrates the behavior of an electrical *relay*.

A Simple Circuit with a Relay



This simple simulation can be used to demonstrate how a relay behaves in a circuit. When the switch is closed, the power provided by Battery A energizes the coil in the Relay. This closes the relay's internal contacts, so that the power provided by Battery B will turn on the light. If the switch is opened again, the coil will be de-energized, the contact will open again, and the light will go out.

There are a number of different ways that this simulation could be written. In the example presented here, an intermediate level of behavior modeling is used. If we had taken a completely high-level 'surface' approach to modeling the circuit's behavior, we would not need to connect object attributes at all. If we had taken an approach that represented electrical phenomena in a more detailed, 'deeper' way, the examples would be significantly more complex.

The first step in building the course on the behavior of the relay was to create the generic objects that serve as the templates for the specific objects in the simulation scene shown above. In the course of building these objects, we sometimes used the ordinary rule-building approach to connecting attributes in a single object. For example, the switch needs a rule that describes when voltages should be passed from its input to its output attribute (called *VoltsIn* and *VoltsOut*, respectively). In the figure below, such a rule is in the process of being built.

Select an expression to replace <Effect1>.  
 Select an expression to replace <Attribute>.  
 Select an expression to replace <numeric>.  
 Select an expression to replace <Effect2>.

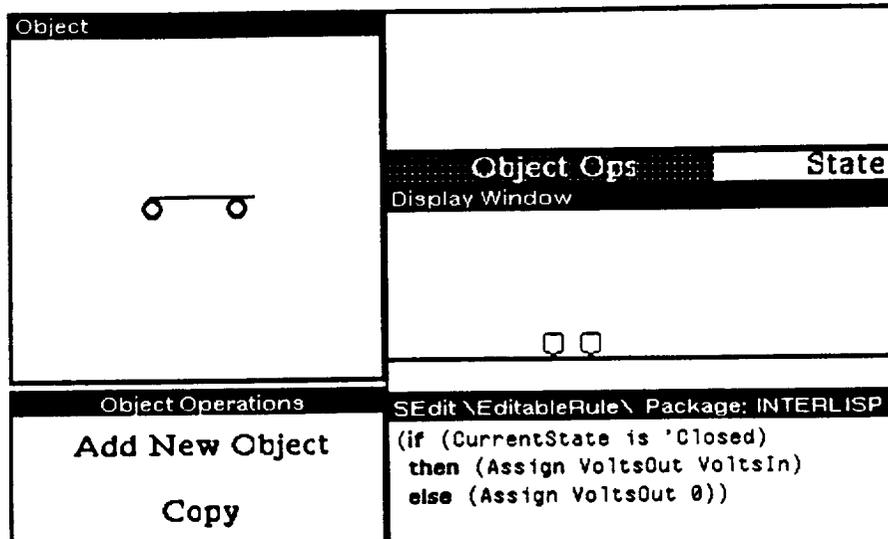
Object Ops State Ops  
 Display Window

Object Operations  
 Add New Object  
 Copy

```
SEdit \EditableRule\ Package: INTERLISP
(if (CurrentState is 'Closed)
    then (Assign VoltsOut VoltsIn)
    else <Effect2>)
```

The sequence of directions to the author ('Select and expression to replace...') shown in the message window at the top indicates that this rule is being built using the menu-based rule creation option.

When the rule has been completed, it appears as shown in the figure below.



This rule could be viewed as an instance of a *conditional connection*, in the sense that it connects the value of the *VoltsIn* attribute to the *VoltsOut* attribute of the switch. Such attribute-connecting rules can be created either in the generic editor (for connecting two attributes of one generic object) or in the specific editor (in order to connect the attributes of different objects).

The two shortcut methods that are the subject of this chapter (mouse-based connections and *Make connection* connections) are available only in the scene editor. They can therefore only be used to connect the attributes of specific objects. When two generic attributes (of a single generic object) are to be connected, the author must create an assignment rule using the rule editor in the generic editor. (It is, however, possible to connect two attributes of a single specific object using either the mouse-based or the *Make connection* methods.)

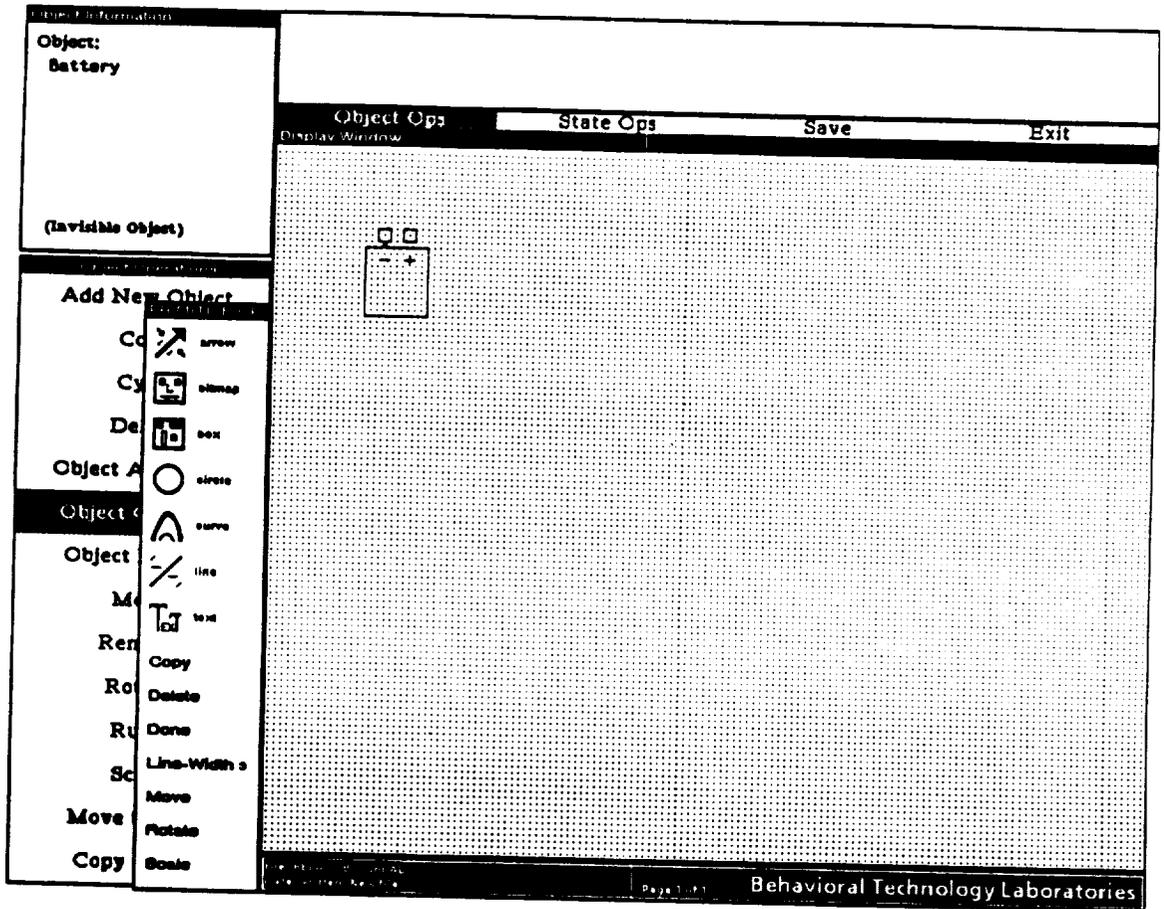
## Connecting with the Mouse

The easiest way to connect two attributes (of two different objects in a simulation) is to use the middle mouse button. The simulation author clicks first on the attribute that is to receive a value, and then on the attribute that is to provide the value. A new assignment rule is automatically created for the first attribute, with the form

(Assign AttributeWhatever of FirstObject SomeAttribute of SecondObject)  
 This process is shown in detail later in this section.

**Attribute Handles**

In order to connect attributes using the mouse, there has to be something to click the mouse on that represents the attributes. This is the function of *attribute handles*. An attribute handle is a rectangular area that represents the location of some attribute of an object. Most attributes don't have attribute handles. There is no particular part of an object that should be associated with its *CurrentState*, for example. For some other attributes, however, especially those that are associated with values that are *input* to or *output* from an object, it makes sense to associate a particular part of the appearance of the object with that attribute. An author sets up such an association by creating an attribute handle for the attribute. All attribute handles must be created in the generic editor.



In the figure above, the author is completing the appearance of the *Battery* generic object in the generic editor. The two terminals at the top of the battery are good contenders for the locations of attribute handles for attributes that will be used to distribute power from the simulated battery.

**Creating Attribute Handles**

After the drawing is finished, the author in this example chooses *Object Attributes* from the object operations menu (to the left of the display window) and adds a couple of attributes, which he chooses to call *PosVolts* and *NegVolts*. Following the procedures described in Chapter 3, he names the attributes and assigns them the type integer. When attributes are first created, their entries in the *Handle Region* column of the Attribute Operations window (see the figure below) appear as **undefined**.

Object Attributes		
Attribute Name	Type	Handle Region
<i>CurrentState</i>	<i>Atom</i>	
<i>ObjectLocX</i>	<i>Integer</i>	
<i>ObjectLocY</i>	<i>Integer</i>	
<i>PosVolts</i>	<i>Integer</i>	<b>undefined</b>

In order to create an attribute handle for an attribute, the author clicks on the word 'undefined,' and a menu pops up with a choice of ways to create a handle for the attribute.

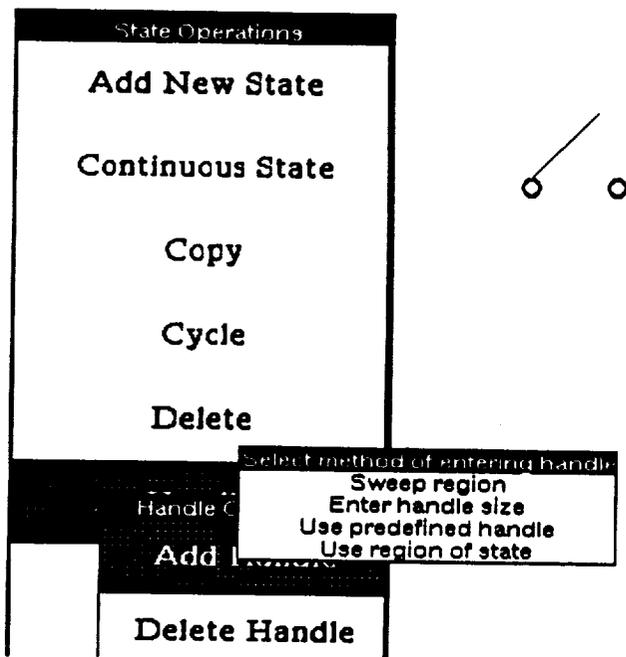
Attribute Operations			
Add	Delete	Done	
Object Attributes			
Attribute Name	Type	Handle Region	Select method of entering handle
<i>ObjectLocX</i>	<i>Integer</i>		Sweep region
<i>ObjectLocY</i>	<i>Integer</i>		Enter handle size
<i>PosVolts</i>	<i>Integer</i>	(100 490 10 10)	Use predefined handle
<i>NegVolts</i>	<i>Integer</i>	(80 490 10 10)	Use region of object

In the case of these attributes, the author wants to create handles about the size of the terminals at the top of the battery, so he chooses the *Use predefined handle* option, which creates a handle that is 10 by 10 pixels. The mouse pointer turns into a black rectangle, and the author positions it on top of the appropriate part of the appearance of the object. In this example, one attribute handle is associated with the *PosVolts* attribute by being placed on the terminal labeled with a + in the battery's appearance. Another attribute handle is created for the *NegVolts* attribute and is placed on the terminal graphic just above the -.

Creating such attribute handles in the generic editor is all that an author has to do in order to be able to select an attribute with the right mouse button when working in the scene editor.

**State Handles  
and Attribute  
Handles**

It is important to keep in mind the distinction between attribute handles and the state handles that are associated with switches and other controls. Authors create attribute handles from the Attribute Operations window, as shown in the figure above. State handles — the regions of a control object that cause the object to change between states — are created in the State Operations mode of the generic editor, as shown in the figure below.



Control-type objects can have both types of handles — state handles and attribute handles. The switch used in this simple example has both types of handles. In the figure below, the Attribute Operations window is open for the same switch. Here the author has created attributes called *VoltsIn* and *VoltsOut* and has associated attribute handles with each. The handles are located at the two circles at either end of the switch. See the figure below.

<b>Object Operations</b>		
Add New Object		
Copy		
Cycle		
Delete		
<b>Object Attributes</b>		
Object Graphics		

<b>Attribute Operations</b>		
Add	Delete	Done
<b>Object Attributes</b>		
<b>Attribute Name</b>	<b>Type</b>	<b>Handle Region</b>
<i>CurrentState</i>	<i>Atom</i>	
<i>ObjectLocX</i>	<i>Integer</i>	
<i>ObjectLocY</i>	<i>Integer</i>	
<b>VoltsIn</b>	<b>Integer</b>	<b>(65 380 10 10)</b>
<b>VoltsOut</b>	<b>Integer</b>	<b>(110 380 10 10)</b>

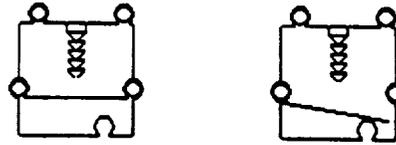


In a similar fashion, the author then creates a *Light* object for the simulation, adds new attributes (here called *VoltsIn* and *GroundSide*, and makes attribute handles for the new attributes.

<b>Attribute Operations</b>		
Add	Delete	Done
<b>Object Attributes</b>		
<b>Attribute Name</b>	<b>Type</b>	<b>Handle Region</b>
<i>ObjectLocX</i>	<i>Integer</i>	
<i>ObjectLocY</i>	<i>Integer</i>	
<b>VoltsIn</b>	<b>Integer</b>	<b>(65 305 10 10)</b>
<b>GroundSide</b>	<b>Integer</b>	<b>(110 305 10 10)</b>



The next step in creating the example simulation to teach about relays is to create the Relay generic object. It should have the two states shown here:



The Relay has four electrical ports: two for the coil and two for the voltage path that is controlled by the relay.

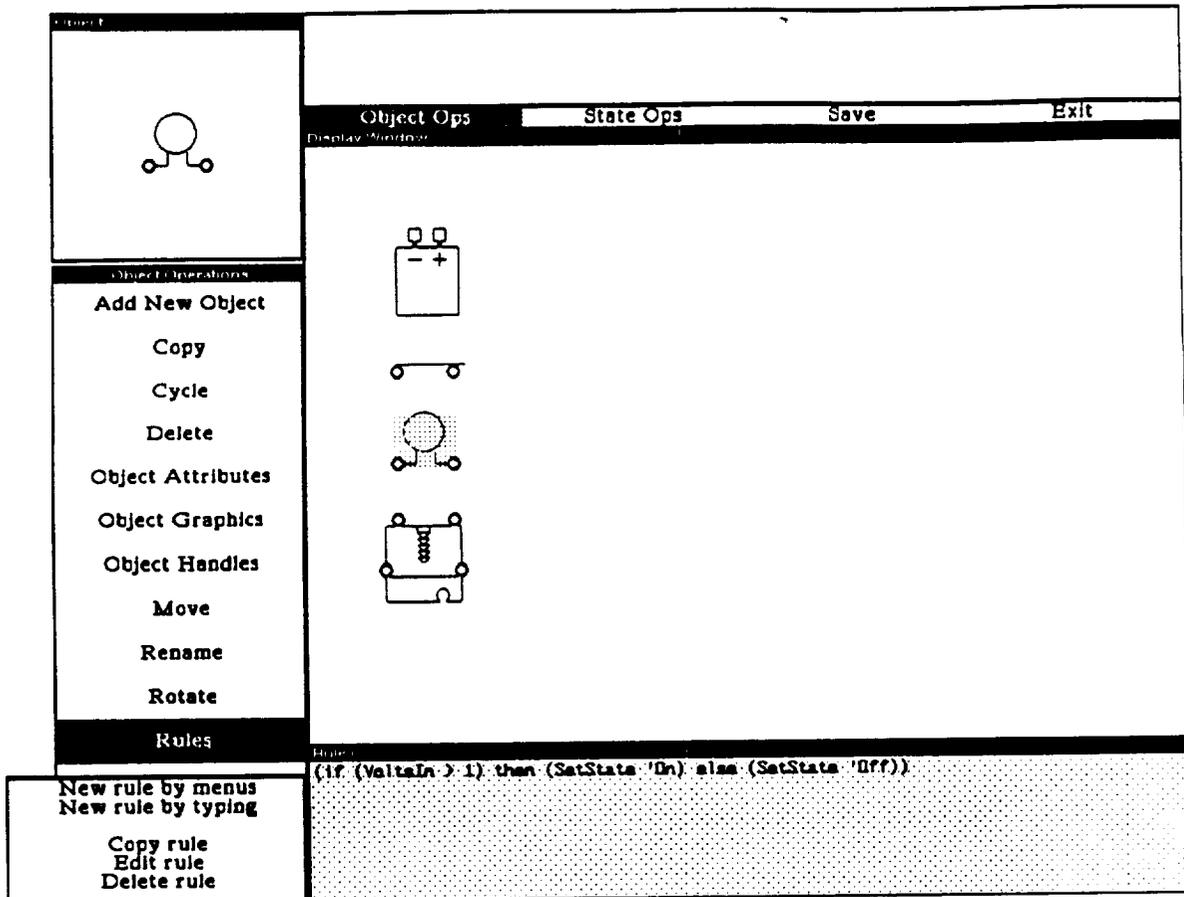
Object Attributes		
Object Graphics		
Attribute Operations		
Add	Delete	Done
Object Attributes		
Attribute Name	Type	Handle Region
<i>CurrentState</i>	<i>Atom</i>	
<i>ObjectLock</i>	<i>Integer</i>	
<i>ObjectLocY</i>	<i>Integer</i>	
<i>CoilVoltsIn</i>	<i>Integer</i>	(65 260 10 10)
<i>CoilVoltsOut</i>	<i>Integer</i>	(110 260 10 10)
<i>SignalVoltsIn</i>	<i>Integer</i>	(55 220 10 10)
<i>SignalVoltsOut</i>	<i>Integer</i>	(115 220 10 10)

Using the Attribute Operations window of the generic editor, the author creates these new attributes and assigns them attribute handles, as shown in the above figure.

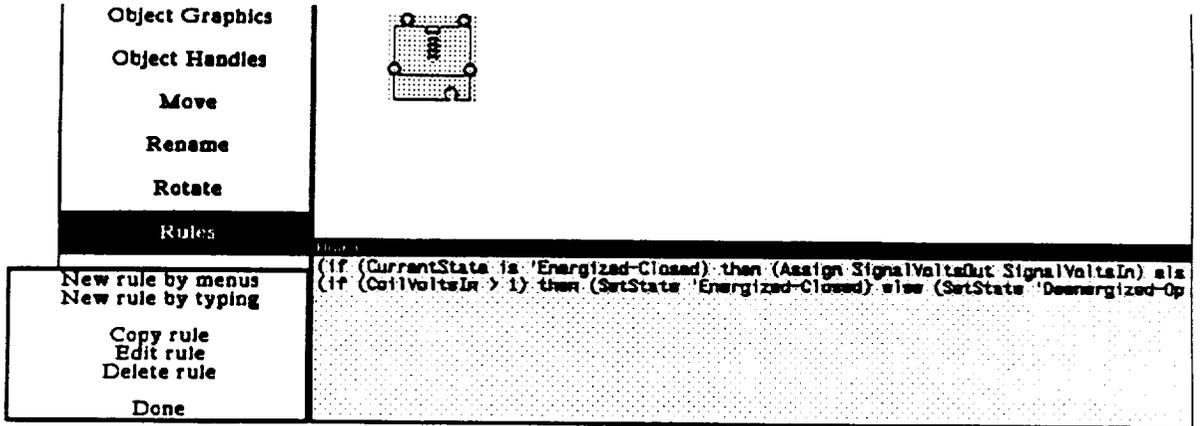
**Defining Generic Behavior**

At this point, all the necessary attribute handles have been created. The author can build whatever rules are required at the generic level. It usually makes sense to write generic rules that govern the internally-determined aspects of an object's behavior. One example is the rule for the switch presented near the beginning of this chapter, which says that values are passed from one voltage attribute to another if the switch is closed.

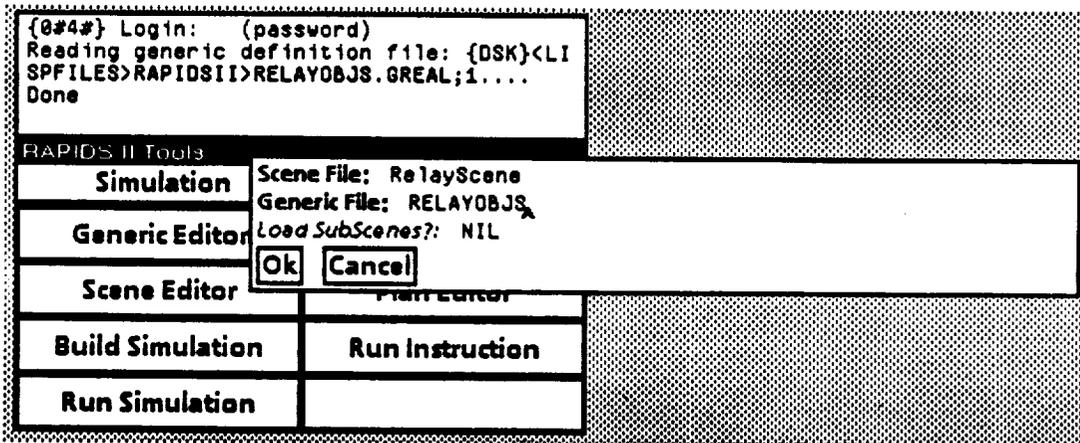
Since the behavior of the light can be said to depend on the values of its voltage attributes, it also makes sense to create a generic object rule that sets the state of the light. The oversimplified rule shown below works for this simulation, but an author could write a more sophisticated rule that would give the object wider applicability.



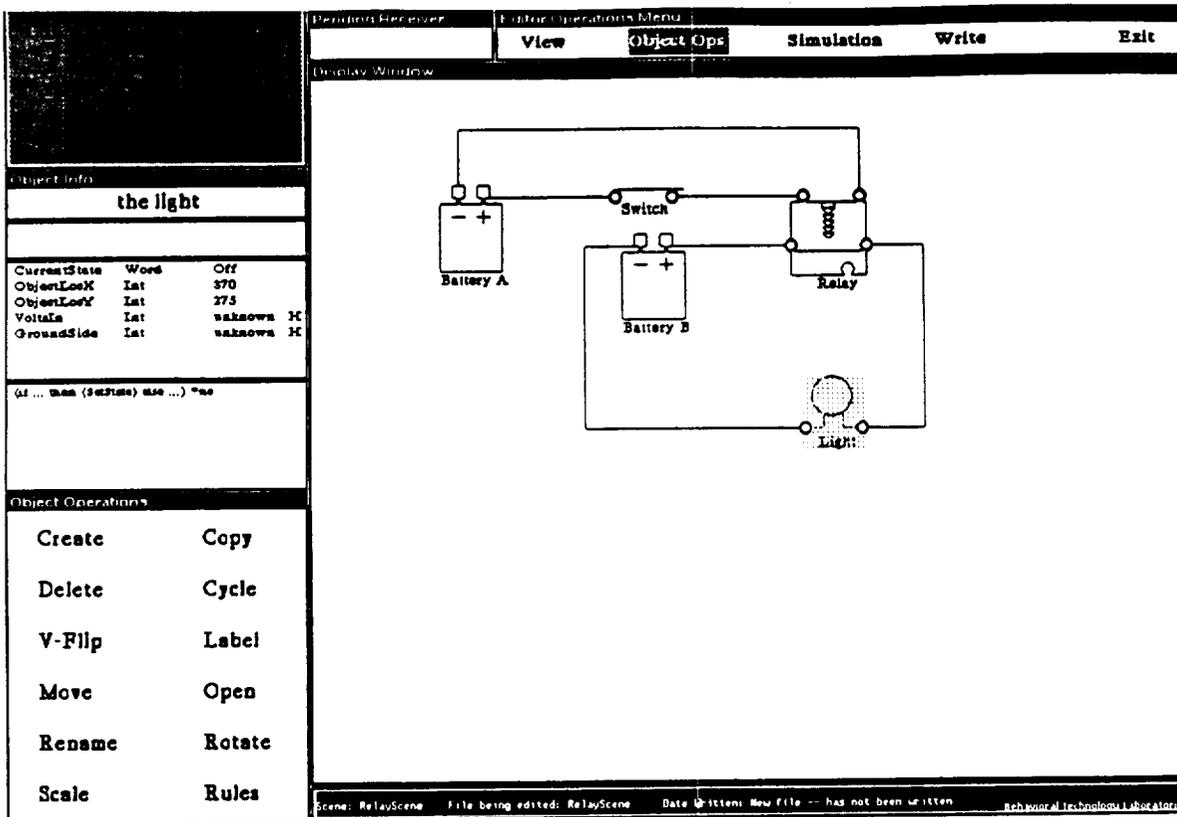
The behavior of the generic Relay can also be defined in the generic editor. Two rules are called for. The first says that signal voltage should be propagated when the relay is in its energized (closed) state. The second says that the state of the relay should be closed when power is supplied to the coil and that the relay should otherwise be open. See the figure below.



**Building a Scene** At this point, the generic library required for the simulation is complete. The author must Save the library before opening the scene editor. The author activates the scene editor and creates the scene shown on the on the second page of this chapter. If you have the files used in this example, and are following along on your machine, use the name *RelayScene* for the scene file and *RELAYOBSJS* for the library file.



The appearance of the scene during its construction in the scene editor is shown in the figure below. The *Create* menu command is used to create new instances of the generic types previously defined and to place them on the scene. The lines ('wires') in the scene were simply drawn in using the background drawing tools of the scene editor. (The author chooses *Edit* from the right button menu in the display window. A drawing menu appears at the left of the Display Window. When the line drawing is complete, the author selects *Done*, and the drawing menu disappears. See Chapters 3 and 5 for additional information on using the drawing tools.)

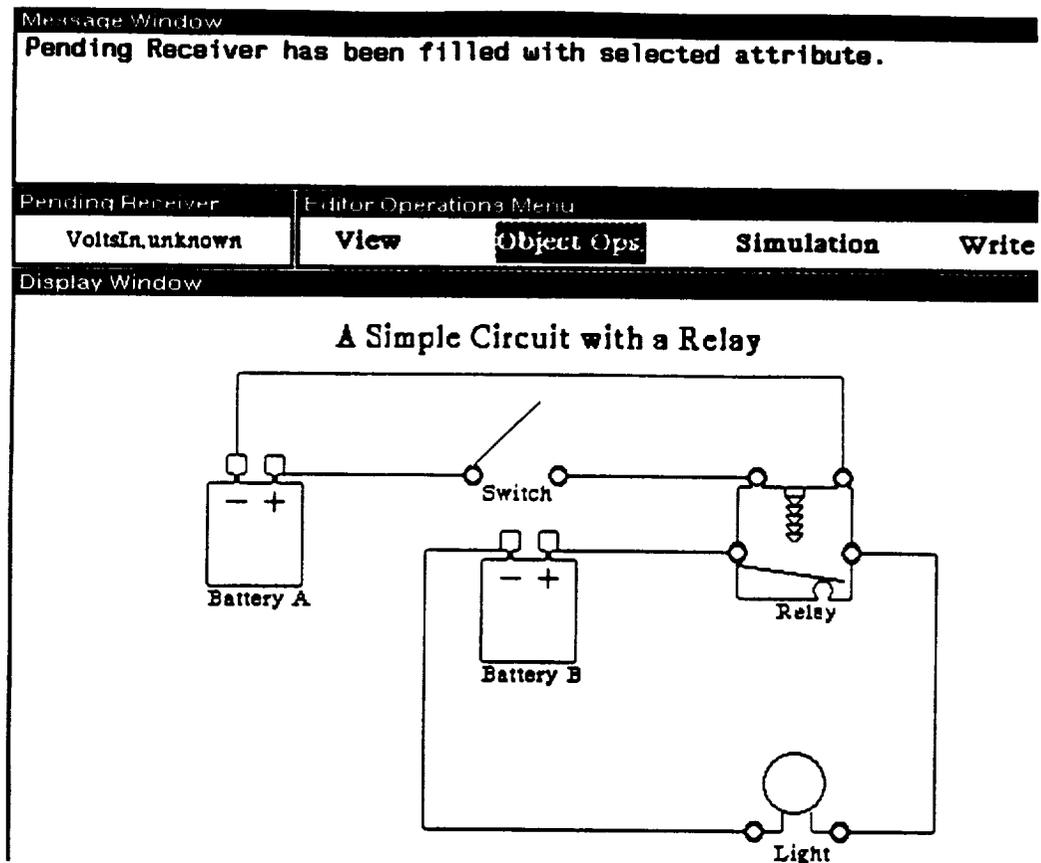


Simply placing the objects on the scene and drawing some lines to make them look attached to each other has not resulted in their becoming functionally connected. At this point the author is ready to begin making connections between objects in the scene. Connections are made in the Object Operations mode of the scene editor.

**Mouse Connections**

Making connections with the mouse is a two-step process. The author first designates the attribute that will be the *receiver* (the one that will be *assigned* a value in the rule that is created) and then designates the attribute that will be the *source* of the assigned value.

An author might choose to start by connecting the input voltage attribute of the switch in the above scene (an attribute called *VoltsIn*) to the voltage attribute associated with the positive terminal of the battery. The first step is to designate the receiving attribute, by clicking with the middle button of the mouse in the circle at the left of the switch. Because that area was previously marked as an attribute handle for the *VoltsIn* attribute, the scene editor recognizes that an attribute has been chosen as the target of a new assignment. It gives feedback to that effect by presenting the message "Pending Receiver has been filled with the selected attribute" in the message window.



Just to the left of the menu bar, above the top left corner of the Display Window is a small window labeled Pending Receiver. This window is ordinarily empty. After the author clicks on the *VoltsIn* attribute handle, however, the name of the attribute, 'VoltsIn' appears in this window. The current value of the attribute is also displayed. Because the simulation has never been run at this point, the value of *VoltsIn* is *unknown*.

When a Pending Receiver has been designated, the next click of the middle button on an attribute handle is interpreted by the scene editor as the source of the attribute value that should be assigned to the attribute named in the Pending Receiver window.

Reading about the mouse-based connection process is much more laborious than simply doing it. The author clicks the middle mouse button once on the handle of the receiving attribute and then on the handle of the sending attribute. Presto! The connection has been made.

Once a connection has been made, a new rule is immediately created that constitutes the functional implementation of the connection. If the author now selects the switch, its object information window (as at the right) will show that there is a rule that assigns a value to *VoltsIn*. (The rule is shown in an abbreviated form — Assign *VoltsIn*. The *\*nc* next to the rule simply indicates that the rule has not yet been compiled for execution efficiency.)

Object Info			
the switch			
Closed	(280 465 10 10)	Closed	
Open	(242 483 62 31)	Open	
CurrentState	Word	Open	
ObjectLocX	Int	235	
ObjectLocY	Int	465	
VoltsIn	Int	unknown	H
VoltsOut	Int	unknown	H
(Assign VoltsIn) *nc			

An author can make most of the required connection in a small scene such as the relay simulation in two or three minutes. In the figure below, the rules of the relay object are examined in the Rules Window immediately after its *SignalVoltsIn* and *CoilVoltsIn* attributes have been connected using the mouse-based connection method. The lowest rules shown in this list are the rules of the generic Relay, while the higher rules — in this case the two rules that begin with the word *Assign* — are rules of the specific relay object on this particular scene.

```

Rules
(Assign SignalVoltsIn (PosVolts of battery for light on RelayScene))
(Assign CoilVoltsIn (VoltsOut of the switch on RelayScene))
(if (CurrentState is 'Energized-Closed) then (Assign SignalVoltsOut SignalVoltsIn) els
(if (CoilVoltsIn > 1) then (SetState 'Energized-Closed) else (SetState 'Deenergized-Up
    
```

Making connections with the mouse is the easiest way to provide for the correct assignment of attribute values between objects in a simulation. It can only be used, however, if *attribute handles* were defined for the generic objects. The next section describes another way to make connections.

## Using Make Connection

The second easy way to make connections is to use the *Make Connection* option in the RAPIDS II rule editor. This feature does not require attribute handles, so it can be used to connect any attributes of specific objects.

CurrentState	Word	Off
ObjectLock	Int	370
ObjectLockY	Int	373
VoltsIn	Int	unknown M
GroundSide	Int	unknown M

(if ... then (SetState 'On') else ... ) \*as

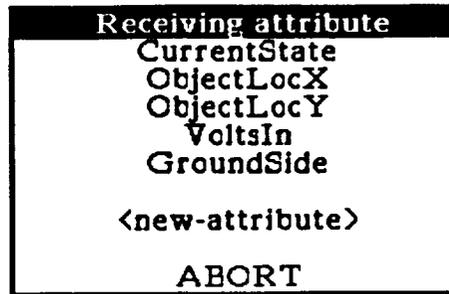
  

<p>Operations</p> <p>Create      Copy</p> <p>Delete      Cycle</p> <p>V-Flip      Label</p> <p>Make connection</p> <p>New rule by menus</p> <p>New rule by typing</p> <p>Copy rule</p> <p>Edit rule</p> <p>Delete rule</p> <p>Done</p>	<p>Rules</p> <pre>(if (VoltsIn &gt; 1) then (SetState 'On') else (SetState 'Off'))</pre>
--	--

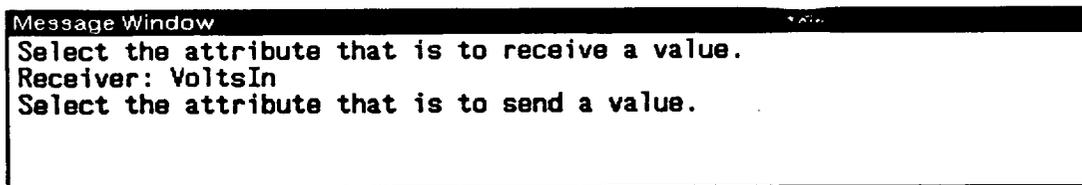
The figure above shows the rule editor being invoked on the specific light in the relay simulation. One rule already exists for the light, a rule inherited from the generic Light object. Here the light's *VoltsIn* attribute must be connected to the *SignalVoltsOut* attribute of the coil above. The author chooses *Make connection* from the top-level rule menu at the left. The message window (near the top of the screen) prompts the author to select an attribute of the light that is to *receive* a value. (See the message window appearance below.)

<p>Message Window</p> <p>Select the attribute that is to receive a value.</p>
---



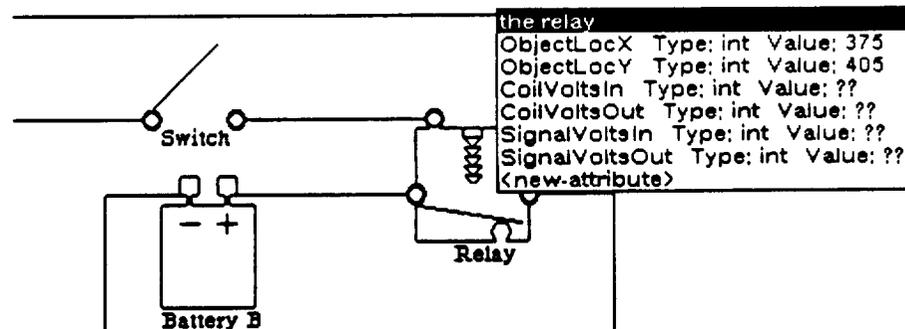
At the same time, the top-level rule menu is replaced with a new menu titled 'Receiving attribute.' This menu lists those attributes that have already been defined and gives the author the option of defining a new attribute at this time.

The author wants to create a rule that assigns a value to *VoltsIn* (that is, one that connects the light's input voltage to a source). The message window shows that the *VoltsIn* attribute has been designated as the receiver and asks the author to pick what attribute is the source of the value.



The easiest way to designate the source (that is, the sending attribute) is to choose it from a list of the attributes of another object. The author holds down the shift key and clicks on the object that has the source attribute. In this case, the relay should be Shift-clicked.

### A Simple Circuit with a Relay



A pop-up menu of the selected object's attributes appears, as in the figure above. Here all the defined attributes of the relay (that are of the appropriate type — integer or real, in this case) appear in the menu, along with an option to define a new attribute. The type and value of each attribute is also displayed in the menu. (If attribute names were not well-chosen, it is sometimes very useful to be able to see their current values.)

Since the light should get its value from the relay's *SignalVoltsOut* attribute, that is the attribute that the author chooses from the pop-up menu. At this

point, the connection has been established, and a new assignment rule appears in the list of rules. See the figure below.

the light		
CurrentState	Word	Off
ObjectLocX	Int	370
ObjectLocY	Int	275
VoltsIn	Int	unknown X
GroundSide	Int	unknown X
(if ... then (SetState) else ...) *no (Assign VoltsIn) *no		
Other Operations		
Create	Copy	
Delete	Cycle	
V-Flip	Label	
Make connection		
New rule by menus		
New rule by typing		
Copy rule		
Edit rule		
Delete rule		
Done		
(Assign VoltsIn (SignalVoltsOut of the relay on RelayScene)) (if (VoltsIn > 1) then (SetState 'On) else (SetState 'Off'))		

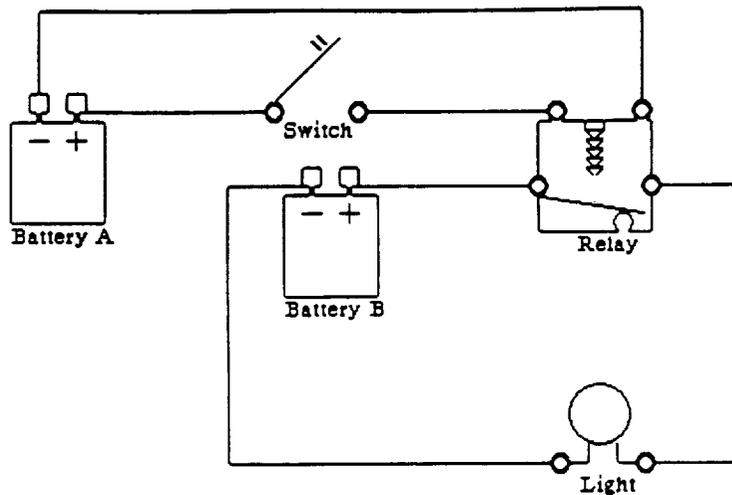
The author can then make another connection using this method, or use one of the conventional (menu-based or structure-editor) approaches to building another rule. As with any other rule, a rule created using *Make connection* can be deleted or edited.

**Testing the Simulation**

After a few minutes of making connections on this scene (using either the mouse connection option or the *Make connection* feature of the rules editor), the author can begin testing the finished simulation.

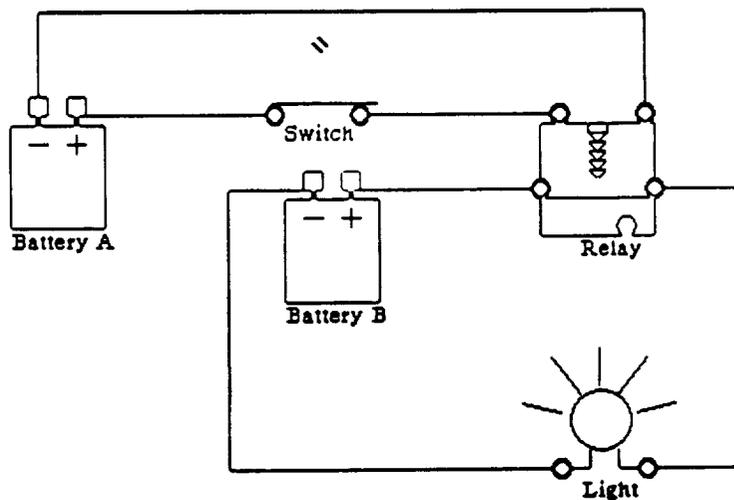
The two figures below show two states of the finished simulation. In the first one, the switch is open, so the relay coil is unenergized, the relay contact is open, and there is no power to the light. In the second figure, the switch has been closed, so the relay coil is energized, its contact is closed, power is available to the light, and it is shining.

**A Simple Circuit with a Relay**



*Figure A. Switch Open, Relay Coil Unenergized*

**A Simple Circuit with a Relay**



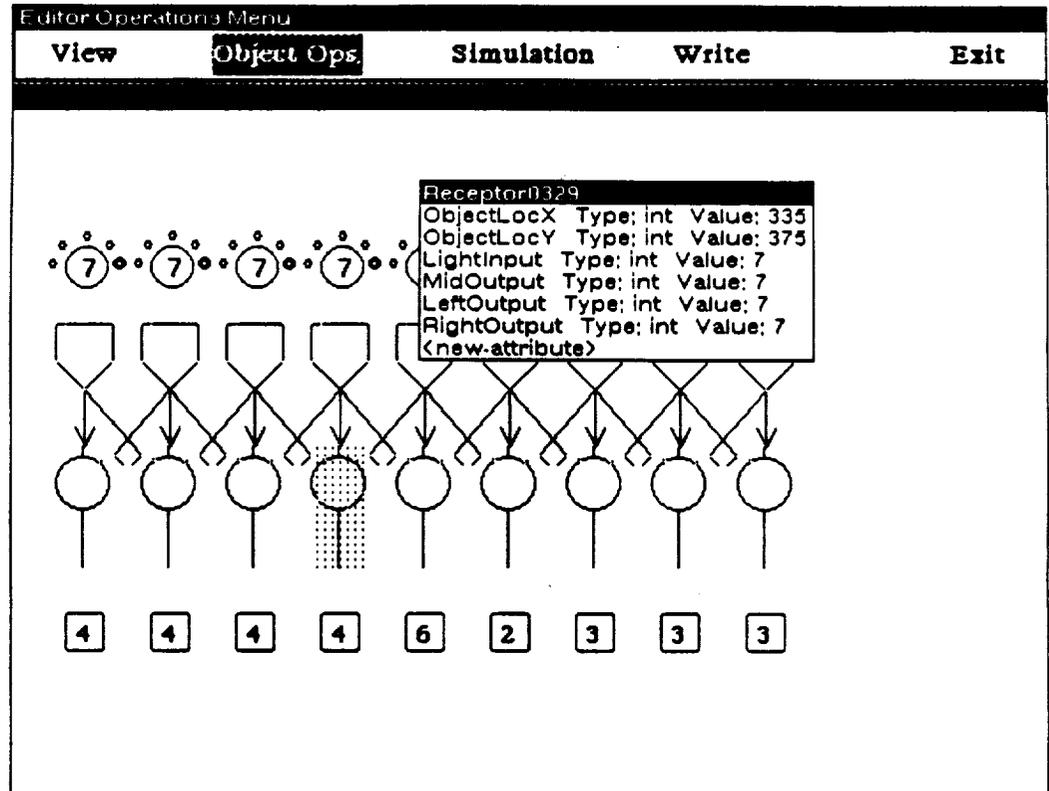
*Figure B. Switch Closed, Relay Coil Energized*

**When to Use  
Make Connection**

This simulation scene is now ready for use as part of a course on the purpose and functioning of simple electrical relays.

Both mouse-based connection and the *Make Connection* option from the Rule Editor are easy to use. If attribute handles have not been defined for a generic object, it is not possible to use mouse-based connection. In such cases, the *Make connection* option is the only one available other than simply building the rule (either by menus or with the structure editor).

Sometimes you will not be able to use mouse-based connection even when attribute handles have been defined. In particular, when two different specific objects with handles have overlapping regions it may not be possible to choose attribute handles from both. This is the case in the simulation of neural connections on the retina, shown below.



Here the retinal ganglion cell objects partially overlap the receptor cell objects from which they receive input. This makes it impossible to connect a ganglion cell's input attribute to the corresponding output attribute of the receptor cell. Fortunately, it was easy to use *Make connection* to establish the proper assignments. (In the figure above, the author is about to assign the *MidOutput* of a receptor to the *MidInput* of the retinal ganglion cell immediately below it.)

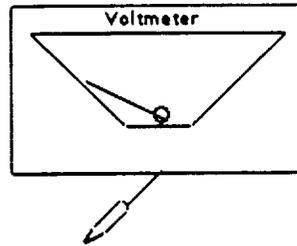
When mouse-based connection fails to work because the objects are so close together that their regions overlap, simply select the object that owns the receiving attribute, choose *Rules* from the object operations menu, and then use *Make connection* to establish the assignment rule that links the attributes appropriately.

**A Note on Productivity**

This description of attribute connection uses quite a few pages to describe two techniques that are really very simple to use. The relay simulation took less than one and a half hours to complete, including building the generic library from scratch, together with testing and debugging the simulation. Part of this productivity was due to the ease with which connections could be made.

## Creating Test Equipment

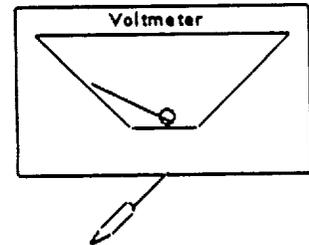
Attribute handles also play an important role in creating pieces of test equipment. To make an item of test equipment, such as the voltmeter shown below, simply requires taking certain attribute actions in the generic editor and then building a special type of rule in the scene editor.



**Attributes for Test Equipment**

In the generic editor, an attribute called *VoltsAtProbe* was created and assigned an attribute handle. See the figure below. This attribute will represent the test probe for the simulated voltmeter. Its handle is located at the stylus in the voltmeter's graphics.

Object Attributes		
Object Graphics		
Attribute Operations		
Add	Delete	Done
Object Attributes		
Attribute Name	Type	Handle Region
<i>StateRotation</i>	<i>Real</i>	
<i>CurrentState</i>	<i>Atom</i>	
<i>ObjectLocX</i>	<i>Integer</i>	
<i>ObjectLocY</i>	<i>Integer</i>	
<b>VoltsAtProbe</b>	<b>Real</b>	<b>(72 55 31 25)</b>
<i>DisconnectValue</i>	<i>Real</i>	<i>undefined</i>



The attribute that will be associated with a test equipment probe can be given any name. The attribute name *DisconnectValue*, however, is a special name that serves a special function in simulated test equipment. Whenever a piece of test equipment is disconnected, the value in its *DisconnectValue* attribute is placed in its test probe attribute or attributes.

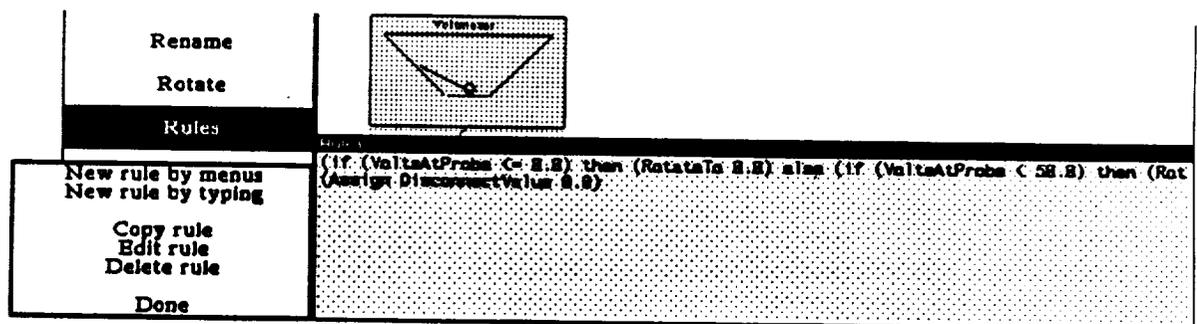
**Generic Behavior** In the case of the voltmeter, if the author wants a value of 0 to register when the meter is disconnected, then the generic object should be given a constant assignment rule that gives *DisconnectValue* that value:

```
(Assign DisconnectValue 0.0)
```

The visual behavior of the generic object must also be authored in the generic editor. The voltmeter's needle should rotate to a position that is determined by the value at the *VoltsAtProbe* attribute. The following rule achieves this effect:

```
(if (VoltsAtProbe <= 0.0)
  then (RotateTo 0.0)
  else (if (VoltsAtProbe < 50.0)
    then (RotateTo (VoltsAtProbe / 50.0))
    else (RotateTo 1.0)))
```

This rule says that if *VoltsAtProbe* is greater than 0, then the needle should rotate to a proportion of its extent determined by the value of *VoltsAtProbe*. This rule creates a voltmeter that measures voltages between 0 and 50.



When these two rules have been created, a simple voltmeter has been defined. (See the above picture.) Naturally, it is possible to build more complex items of test equipment, as well. For complex test equipment, it is usually best to create the controls and indicators as separate generic objects. The indicators should have the test probe attributes and the special *DisconnectValue* attribute.

**Defining Specific Test Equipments** To make a specific test equipment indicator, you must create a special rule that assigns the reserved attribute indicator *PROBE* to your test probe attribute. The rule editor supports the authoring of such rules by menu in the scene editor environment.

In the case of the simple voltmeter, the specific voltmeter object needs a rule that assigns *PROBE* to its *VoltsAtProbe* attribute:

(Assign VoltsAtProbe PROBE)

In the figure below, such a voltmeter is shown in the simulation operations mode of the scene editor. It has been connected to the output voltage of the relay.

The screenshot displays the scene editor interface. At the top, a menu bar includes 'View', 'Object Ops.', 'Simulation', 'Write', and 'Exit'. Below the menu is a 'Display Window' titled 'A Simple Circuit with a Relay'. The circuit diagram shows Battery A, a Switch, Battery B, a Relay, and a Light. A voltmeter is connected to the relay's output. To the right of the circuit is a separate window labeled 'Voltmeter' showing a needle deflected from zero.

On the left side of the interface, there are several panels:

- Pause/InPause**: A button labeled 'Running'.
- Clock**: A digital clock showing '0:0:44'.
- Current Events**: An empty list.
- Simulation Attributes**: A table with the following data:
 

Clock	Int	0
MouseX	Int	157
MouseY	Int	4
MouseState	Word	Up
- Object Info**: A table for 'Voltmeter0469':
 

StateRotation	Real	0.0
CurrentState	Word	MediaRotati
ObjectLock	Int	515
ObjectLockY	Int	410
VoltsAtProbe	Real	5
DisconnectValue	Real	unknown
- Scripting**: A list of rules:
  - (If ... then (RotateTo) size ...) \*no
  - (Assign DisconnectValue) \*no
  - (Assign VoltsAtProbe) \*no
- Tools**: A list of actions:
  - Snap
  - Save State
  - Pause Rules
  - Trace Attributes
  - Compile
  - Restore State
  - Pause Attributes
  - System Trace

At the bottom of the interface, a status bar shows: 'Scene: Relayscene File being edited: (D:\A)\FILES\RAPIDSII\RELAYSCEM.E File Date Written: 10/11/90 15:02:15'.

The needle has deflected about 10% from its 0 value in this figure. It is measuring five volts, so its needle has rotate 10% of the extent between its 0- and 50-volt values.

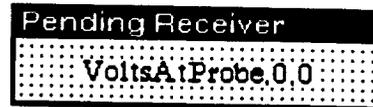
**How Test Equipment is Used**

Clicking on a test probe with the middle button during a simulation makes that probe the *currently pending* probe. The little window labeled *Pending Receiver* (at the top left of the Display Window) will then show the name of the test probe attribute. (Its value is also shown here. When the test probe is first designated, its value will be *unknown*.) The test probe will remain the pending receiver until a different test probe is selected.

To hook the chosen test probe up to a test point on the target equipment, the user must click the middle button of the mouse on some test point. A test

point is interpreted here as any attribute handle in the simulation. When this attribute handle is middle-buttoned, the test probe attribute is temporarily connected to it, until the probe is connected elsewhere or is disconnected.

An active test probe can be disconnected using the right button menu of the Pending Receiver window. After such a disconnect takes place, the value at the test point will be the value of its *DisconnectValue* attribute. See the updated (disconnected) attribute value in the figure below.



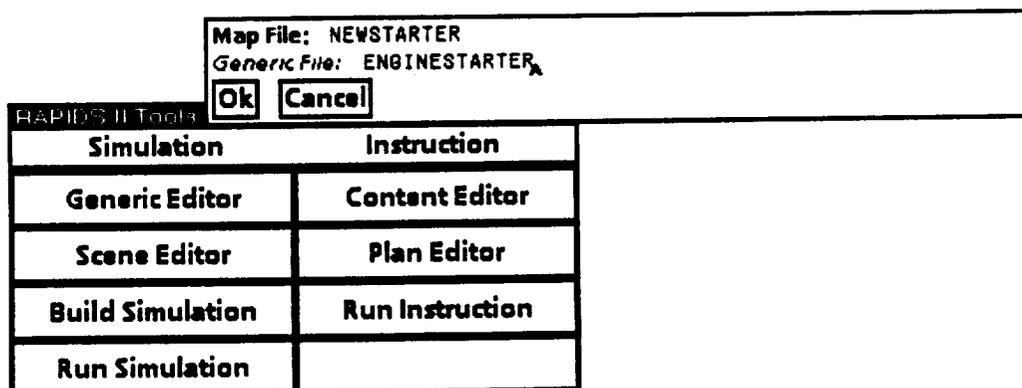
So long as a test point is connected to another object's test point (attribute handle), the behavior of the test equipment will reflect changes in the test point's attribute value. This approach to test equipment permits very flexible and realistic simulation behavior in RAPIDS II.

## Authoring Instructional Content

The content unit editor is used to create and edit a course's *content units*. A content unit is a fragment of a lesson that is based on a RAPIDS II simulation. Complete courses are constructed using the instructional plan editor, which is described in the next chapter. Each content unit (lesson fragment) includes one or more *content items*, which are also created in the content unit editor. Every content item has a *student action*. The *expositions* associated with content units and content items are composed in the content unit editor. In sequence, this chapter describes the editing of content units, content items, student actions, and expositions. The examples in this chapter are based on a course about the jet aircraft engine starter system described in Chapter 2. You can examine this course using the RAPIDS II authoring tools that you loaded earlier.

### Building a Simulation

The content editor can only be used to build course materials for the most recently *built simulation* in the environment. Here we use the term *built simulation* in a special way to mean a special simulation that has been built by choosing the *Build Simulation* menu item in the *RAPIDS II Tools* menu. When you click on this option, the dialog box shown below appears.



The *Map File* referred to in this dialog is the highest-level scene in the simulation. (If the simulation has only one scene, that is the name to insert.)

The *Generic File* is, of course, the library of generic objects used to build the simulation.

After the *OK* button is clicked, several minutes will be required to build a run-time, compiled version of the simulation. As they are processed, names of the specific objects in the simulation will appear in the window at the top of the *RAPIDS II Tools* menu. When the simulation build process is complete a message to that effect will appear above the menu:

Right Start Button Left Start Button The simulator is now built.	
RAPIDS II Tools	
Simulation	Instruction
Generic Editor	Content Editor
Scene Editor	Plan Editor
Build Simulation	Run Instruction
Run Simulation	

The content unit editor always works on the last *built simulation*. Even if another simulation has been edited since, the course development will apply to the older built simulation.

**Starting the Content Unit Editor**

Normally, the content editor is started by using the RAPIDS II top-level menu, as shown below. After clicking on the *Content Editor* button in the Instruction column, a dialog box opens, asking for the name of the content file. The RAPIDS II environment supports only one active simulation at a time, so there is no need to name the simulation that the instructional content will be based on — it must be based on the current simulation.

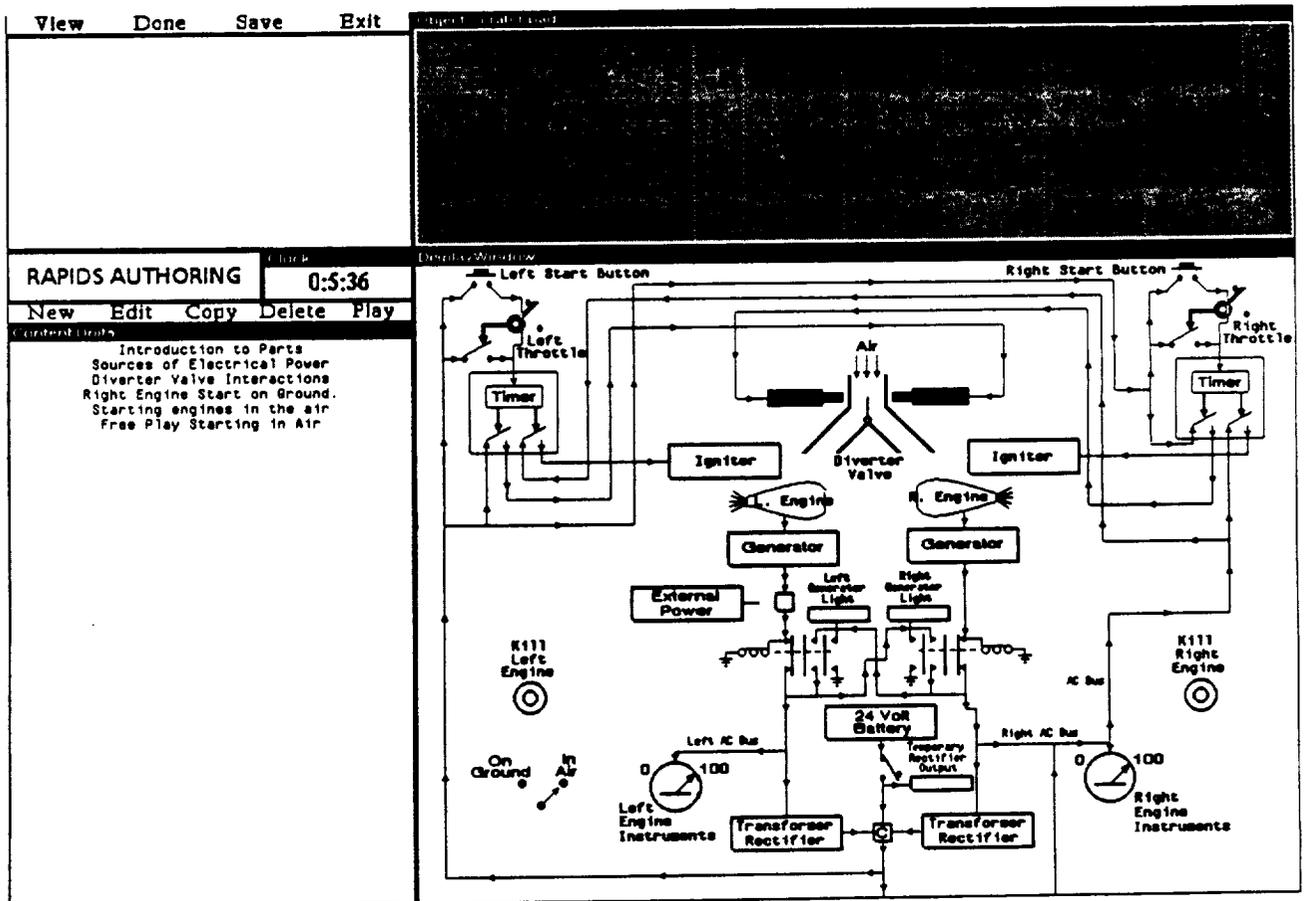
Content File: STARTERTASKS	
Ok Cancel	
RAPIDS II Tools	
Simulation	Instruction
Generic Editor	Content Editor
Scene Editor	Plan Editor
Build Simulation	Run Instruction
Run Simulation	

To edit the EngineStarter course used in the examples in this chapter, specify the content file called *STARTERTASKS*. After a brief delay, a set of windows similar to those shown on the next page will open on your screen.

Another way to start the content unit editor is by invoking the Lisp function call *ContentUnitEd*, as in  
 (ContentUnitEd 'CourseName)  
 where 'CourseName is the name of the content unit file to be edited. Most authors will prefer to use the *RAPIDS II Tools* menu, as shown above.

## Content Units

When you first open a content editing session, you will see a display like that shown below. The simulation window will contain the top scene from the simulation that has been built in your Lisp partition. Here we see a new course for the NewStarter simulation, just after it has been opened in the content unit editor.



The window at the left lists the content units that have already been defined for the course, if there are any. Just above that window is a set of commands that apply to content units as a whole. These commands are New, Edit, Copy, Delete, and Play. They allow you to create a new content unit, or to edit, copy (and edit the copy), delete, or play an existing unit.

**New   Edit   Copy   Delete   Play**

### New

At this level of the content unit editor, you can choose to create a new content unit, by clicking on New in the area above the list of content units. Doing so will open a new *Unit Editor Window*, shown below. The unit editor lists the data fields of a content unit, along with the default values that are assigned to certain fields.

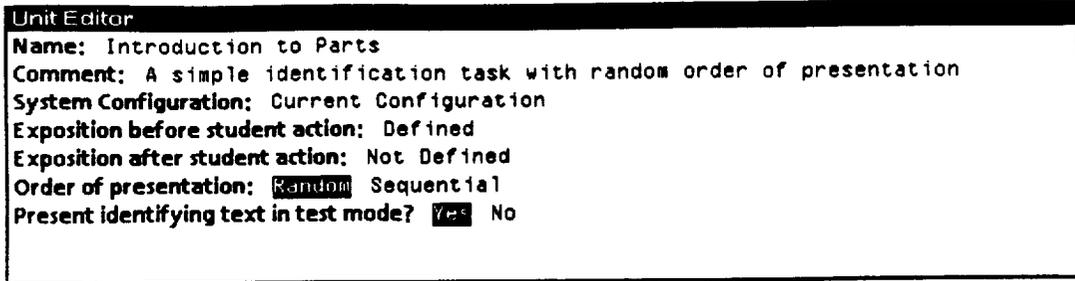
Unit Editor	
Name:	
Comment:	
System Configuration:	Current Configuration
Exposition before student action:	Not Defined
Exposition after student action:	Not Defined
Order of presentation:	Random <input checked="" type="checkbox"/> Sequential
Present identifying text in test mode?	<input checked="" type="checkbox"/> No

### Edit

To acquaint yourself with the editor, you should first use the unit editor to edit an existing content unit. To edit a unit, you must first *select* it, by clicking on its name in the list of content units. The selected unit name will then appear inversed (that is, as white text on a black background).

Content Units	
Introduction to Parts	
Diverter Valve Interactions	
Left Engine Start on Ground	
Right Engine Start on Ground	
Start when engines die in the air	
EngineStarter Test	
Identify sources	

If, after selecting the unit called 'Introduction to Parts,' you click on Edit, you will see the unit editor window open at the top of the simulation window. It will display the values of the data fields associated with that unit, as shown below.



The meanings of each of the data fields for a content unit will be described later in this section.

**Copy** It is often useful to model one content unit on another that has already been defined. You can do this by selecting the unit that you want to base a new unit on and then choosing **Copy** from the menu of content-unit-level commands. The new copy of the unit will immediately be opened for editing. Since it will have exactly the same data as did the original unit, the first thing you should do is to change the name of the unit.

**Delete** Choosing the **Delete** command from the menu of content-unit-level commands will cause the currently selected content unit to be deleted. The editor will ask you to confirm that the unit should actually be deleted by clicking the left mouse button. If you click the right mouse button, the delete command will be aborted.

**Play** You can see how a content unit will appear to students by choosing the **Play** command, which applies to the currently selected content unit. A menu will appear asking whether you want to go through the unit in Drill or Test mode. After you make one of these selections, the screen will change its appearance. The command buttons in the upper left corner of the screen (Menu, Replace, Find Object, and Indicator) will be replaced with the Options Menu of the student user interface (Quit, Don't Know, Test Equipment, and View). In effect, you are now a student being presented with that content unit.

When you finish the unit, the student interface will disappear, and the content unit editor will be restored. The unit you just played will still be the selected unit.

## Editing Content Unit Data

The data fields of content units are described in the remainder of this section. To follow the examples on your own computer, begin by selecting the 'Introduction to Parts' unit and then clicking on the Edit command. The major windows will look something like the figure below.

View	Done	Save	Exit	<p><b>Name:</b> Introduction to Parts  <b>Comment:</b> A simple identification task with random presentation order  <b>System Configuration:</b> Initial State  <b>Exposition before student action:</b> Defined  <b>Exposition after student action:</b> Not Defined  <b>Order of presentation:</b> <input type="checkbox"/> Random <input checked="" type="checkbox"/> Sequential  <b>Present identifying text in test mode?</b> <input type="checkbox"/> Yes <input checked="" type="checkbox"/> No</p>
CONTENT UNIT				
		0:5:36		
New		Edit	Move Delete	
<p>Find the Left Start Button.          Find the Left Throttle.          Find the Left Timer.          Find the Left Igniter.          Find the Left Engine.          Find the Left Generator.          Find the Left Relay.          Find the Emergency Power Switch.          Find the Emergency Power Light.          Find the Right Start Button.          Find the Right Throttle.          Find the Right Timer.          Find the Right Igniter.          Find the Right Engine.          Find the Right Generator.          Find the Right Relay.</p>				

The list of content units that was on the left of the simulation window has now been replaced with a list of the defined *content items* for the content unit that is being edited. Above this list of content items is a menu with three choices: New, Edit, Move, and Delete. These commands apply not to the content unit as a whole, but rather to content items. The next section will discuss content item editing.

In the remainder of this section, the other data elements of a content unit are described, along with how they are edited. Those elements are:

- Name
- Comment
- System Configuration
- Exposition before content unit
- Exposition after content unit
- Order of presentation
- Present identifying text in test mode?

In most cases, you will find that you can learn how to enter or edit these data elements simply by trying to do so. The first step is to click in the field.

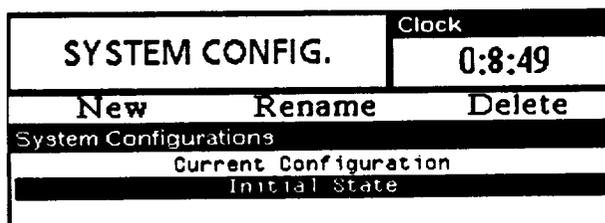
**Name** Click on **Name** in the content unit window, or click anywhere in the name if one has already been defined. The typing cursor appears at the point that you've clicked. You can delete letters by backspacing and type a new name. The name of a content unit can include spaces.



If you fail to give a content unit a name and choose *Done* in the top menu, the name will be shown in the list of content units as '---'.

**Comment** The comment associated with a content unit is edited in just the same way that a name is. Click where you want to enter or delete material and type normally. This data field is optional. The comment is never seen by students; it is meant only as an aid to courseware documentation for authors.

**System Configuration** When you click on the **System Configuration** command, the content unit window disappears, so the simulation window has nothing overlaying it. The list of content items at the left is replaced with a list of defined system configurations.



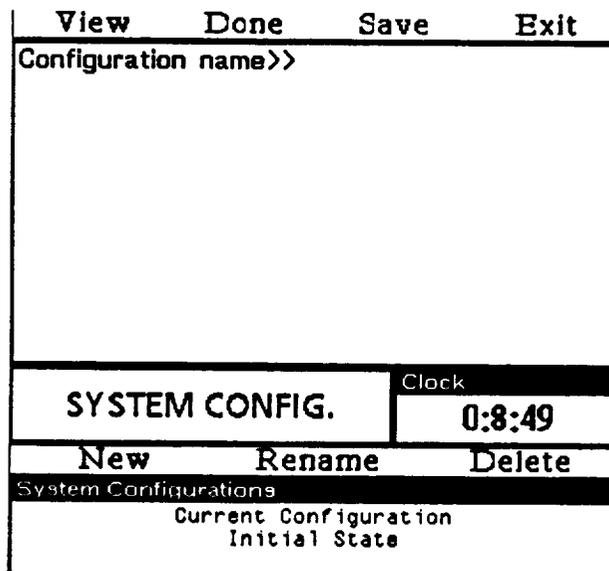
If one of these is a defined state of the simulation that you want to have installed when the content unit begins, you can simply click on it to select it (a system configuration called 'Initial State' has been selected in the above figure), and then click **Done** on the menu bar above the simulation window. At this point the content unit window will reappear, and the list of defined system configurations at the left will be replaced once again by the list of content items for the unit.

*Initial State* is a special pre-defined state. It is the state that the simulation was left in when it was last saved in the scene editor. (Hence, this is the state that the simulation will be in after *Build Simulation* is carried out.)

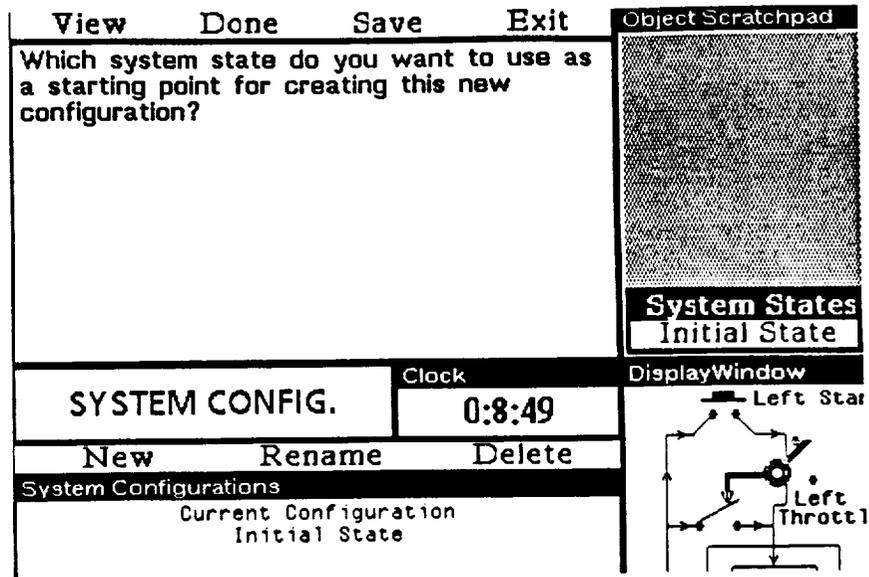
*Current Configuration* has a different meaning from all the other configuration names that can appear in the list of system configurations. When an author selects *Current Configuration*, it means that it doesn't matter what configuration is installed when the unit is started; the author is indifferent. Be careful not to use this option if students will be required to manipulate switches or if the lesson fragment discusses any aspect of the displayed system configuration. Students may see a quite different simulation state, depending on what happened in the previous content unit.

**Defining  
a New  
Configuration**

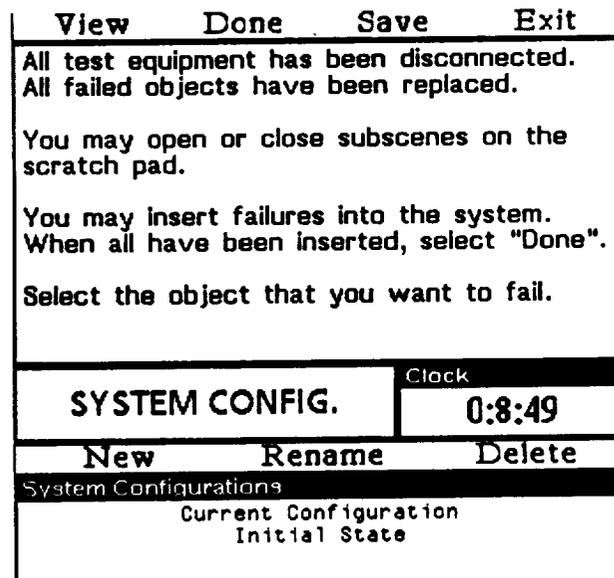
Sometimes you may want to begin a content unit in a system configuration that has not yet been defined. If so, you must define the new system configuration. Note that the menu bar above the list of defined system configurations has three commands relevant to system configurations. Choose the **New** command to begin defining a new configuration. You will be asked to name the new configuration.



The new configuration can be based on an existing configuration, including the Initial State. This means that a previously defined configuration can be loaded in to serve as the starting point for defining a new configuration. Choose from the list by clicking on the desired configuration in the menu labeled *System States*. This menu is shown in the figure below, just above the upper left corner of the Display Window.



A system configuration can include failure states for components of the device. This means that before a content unit is started, the selected component failures are entered into the simulation. Most system configurations don't include failures, so the author ordinarily simply clicks on *Done* in the menu above at this point. See the figure below.

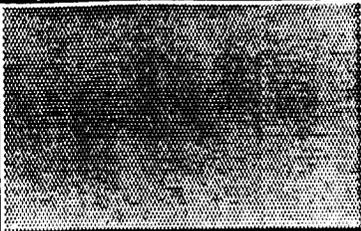
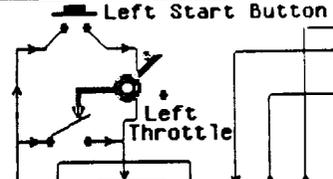


Then the author manipulates the simulation switches, just as the student would. When the desired simulation state is achieved, clicking on *Done* marks that state as the new system configuration. The figure below shows the prompts that appear at this point in the process of defining the system configuration.

During this phase authors can change scenes normally so that they can manipulate switches on other scenes. They can also — using the commands on the right button menu — open and position scratchpad scenes. (For more information on the Object Scratchpad, see Chapter 5.) As you click on switches to set them, a textual transcript of your actions will appear in the Message Window. In the example shown above, a new system configuration is being defined that begins with the Right Start Button being put into the Pressed position. When you have finished setting up your new configuration, click on Done on the menu bar above the simulation window.

View		Done		Save		Exit	
<p>You may insert failures into the system. When all have been inserted, select "Done".</p> <p>Select the object that you want to fail.</p> <p>Set the switches to put the system into the appropriate configuration.</p> <p>When all have been set, select "Done".</p>							
SYSTEM CONFIG.						Clock	
						0:0:0	
New		Rename		Delete			
System Configurations							
Current Configuration							
Initial State							

At this point the author is asked whether to save the whole configuration or simply the sequence of steps gone through to set up the simulation. A menu appears in the object scratchpad area presenting these two choices, along with the option to abort the definition of this system configuration. (See below.) Either saving the whole configuration or the switch sequence will work. If the switch sequence is short, that is usually a better choice, because it requires that less data be stored and retrieved. On the other hand, when such a system configuration is reinstalled, it actually goes through the process of simulating each switch throw in turn, so students may observe a good deal of possibly mysterious simulated activity at the beginning of a unit as the configuration is installed.

View Done Save Exit		Object Scratchpad
Left Start Button CurrentState: Pressed Do you want to save the entire system configuration or just the sequence of switch settings?  If you abort, the system configuration will revert to what it was before you set the switches and nothing will be saved.		
SYSTEM CONFIG. <span style="float: right;">Clock 0:1:56</span>		DisplayWindow
New Rename Delete		
System Configurations Current Configuration Initial State		

After one of the two *Save* options is selected, the new system configuration name appears in the list of configurations. Here a system configuration called *Left Running on Ground* has been defined.

SYSTEM CONFIG.		Clock
		0:1:56
New	Rename	Delete
System Configurations		
Current Configuration		
Initial State		
Left Running On Ground		

**Other Configuration Options**

When you have selected **System Configuration** in the content unit window, you can also perform two other system configuration options — renaming and deleting defined configurations. To exercise either capability, you must first select the configuration by clicking on its name in the list of system configurations. Then choose the command you want from the menu above — **Rename** or **Delete**.

Before using the **Delete** command, however, be aware that it may be dangerous. Other units may call for the deleted configuration. If so, they will thereafter install the 'Initial State' on startup. You will be warned of this danger if you begin to delete a configuration that is used by another unit.

**Rename** is not similarly hazardous. If you rename a system configuration, all the content units that use it will automatically refer to it by its new name.

**Content Unit Expositions**

There are two exposition fields for a content unit, **Exposition before content unit** and **Exposition after content unit**. The expositions you create for these fields will be presented to students at the beginning and end of the content unit, respectively.

Clicking on one of these fields in the content unit window will bring up the exposition editor, which always appears to the left of the simulation window. Because the exposition editor offers a rich set of options for creating presentations to the student, it is described in a separate section later in this chapter.

Once an exposition has been created, the word 'Defined' appears near its field name in the content unit window. See, for example, the Exposition before content unit field below.

Unit Editor	
Name:	Introduction to Parts
Comment:	A simple identification task with random order of presentation
System Configuration:	Current Configuration
Exposition before student action:	Defined
Exposition after student action:	Not Defined
Order of presentation:	<input checked="" type="checkbox"/> Random <input type="checkbox"/> Sequential
Present identifying text in test mode?	<input checked="" type="checkbox"/> Yes <input type="checkbox"/> No

#### Order of Presentation

You can decide whether you want the content items of the current unit to always be presented in a fixed order (the order in which they are listed in the list of content items) or randomly. Simply click on the option you want for this field.

In a newly created unit, the default ordering of content items is **sequential**.

#### Identifying Text in Test Mode?

As you will see in the next section, each content item has an identifying text that helps specify what student action is required. In most cases, this text should be used in constructing a prompt for the student. **Yes** is the default value of this field of a new content unit. In rare cases, an author wants to require that a student perform a series of steps without any prompting when the content unit is presented in test mode. In this circumstance, the option **No** must be selected.

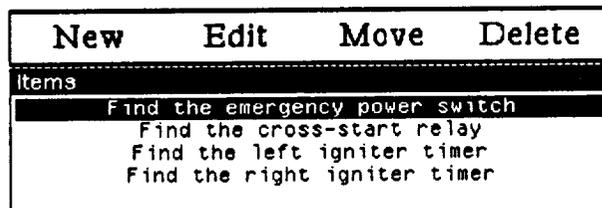
In the other two modes, **instruct mode** and **drill mode**, identifying text will always be presented. This option allows you to eliminate identifying text in test mode only.

## The Content Items Menu

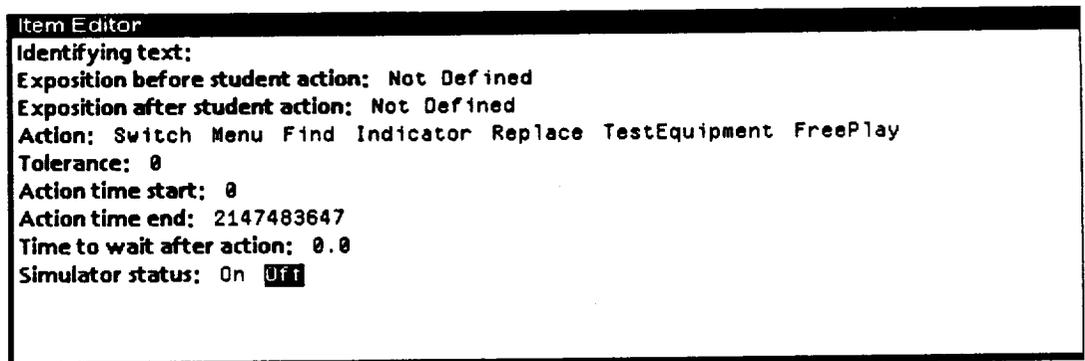
Content items are usually the most important parts of their content units. They prescribe what actions a student must take while working through the content unit. Each content item may have associated expository material, as well.

Content items appear in a list to the left of the simulation window while you are editing a content unit. At the top of this list is a menu bar with four commands that apply to menu items: New, Edit, Move, and Delete.

- New** The New command will create a new item and open an item editor window with default values for the four data elements.
- Edit** When you are editing a content unit, the list of its content items appears in the window to the left of the simulation window. To edit one of these items, first select the item by clicking on it. In the figure below, an item called 'Find the emergency power switch' has been selected. Then click on Edit in the menu above.



An *Item Editor Window*, such as the one shown below, will then open at the top of the simulation window.



*The Content Item Editor Window*

**Move**

Clicking on **Move** in the menu bar above the list of content items signals that you want to move the selected item to a different point in the list. It doesn't make much sense to move an item if the unit uses random presentation, but **Move** is commonly used to reorder items in sequential units.

When you issue the **Move** command, a prompt will appear that asks you to click on the item that the selected item should appear after.

Text Input Window

Select location to move item after (to move to beginning of list select first item and choose the appropriate menu response)

If you pick the first item in the list of content items, a menu will appear that asks you whether the originally selected item should appear before or after the first item.

Select location relative to selection
Before
After

**Delete**

The **Delete** command is used to remove the selected content item. When you choose **Delete**, the content item editing window appears with the item to be deleted displayed in it. You are asked to confirm the deletion by clicking the left mouse button. When the deletion is confirmed, the editing window is closed and the item is removed from the item list.

---



---

## Content Item Data

Content units contain four data elements: identifying text, optional pre- and post-expositions, and a student action. The authoring of student actions is covered in some detail in the next section.

Item Editor	
<b>Identifying text:</b>	
<b>Exposition before student action:</b>	Not Defined
<b>Exposition after student action:</b>	Not Defined
<b>Action:</b>	Switch Menu Find Indicator Replace TestEquipment FreePlay
<b>Tolerance:</b>	0
<b>Action time start:</b>	0
<b>Action time end:</b>	2147483647
<b>Time to wait after action:</b>	0.0
<b>Simulator status:</b>	On <input type="checkbox"/>

- Identifying Text** You can edit Identifying Text in much the same way that content unit names are edited. The next section of this chapter, *Student Actions*, describes how the identifying text of an item can be automatically generated by the RAPIDS II editor, based on the student action.
- Item Expositions** The optional pre- and post-exposition elements of a content item are created and edited with the Exposition Editor, which is described later in this chapter. If you click on one of these fields ('Exposition before student action' or 'Exposition after student action') in the item editor window, then the exposition editor will open at the left of the simulation window.
- Action** This field lists the seven types of student action that can be required in a content item. Each one of these action types (Switch, Menu, Find, Indicator, Replace, Test Equipment, and Freeplay) has its own ways of being authored. So far as possible, each such action is authored by simply carrying out the action, just as the student would. The details of these authoring processes are described in the next section of this chapter.
- Done** When you have finished editing a content item, click on **Done** in the menu bar above the simulation window. The item editor window will close and be replaced with the content unit editor window for the content unit you are working on. The content item list will still be displayed on the left. You can edit another item by selecting it and clicking **Edit** again.

---



---

## Student Actions

There are seven kinds of student actions that can be prescribed in a content item. They are:

- manipulating one or more switches into specified states
- make one or more selections from a menu of text items
- 'Finding' by either
  - clicking on one or more objects in the simulation, or
  - clicking in a region on a scene
- noting an indicator value
- replacing a simulated object
- performing a specified test using simulated test equipment
- interacting in a free play fashion with the simulation

This section walks through the actions required to author each of these student action types. You should be able to carry out the process on your own machine.

**Finding Objects** If you have been following this text by carrying out the illustrated operations, you are now in the unit called 'Introduction to Parts.' Try recreating one of the content items in this unit. Delete the item called 'Find the emergency power switch.' Then select New from the menu bar over the item list. The item editor window will open.

Item Editor

**Identifying text:**

Exposition before student action: Not Defined

Exposition after student action: Not Defined

Action: Switch Menu Find Indicator Replace TestEquipment FreePlay

Tolerance: 0

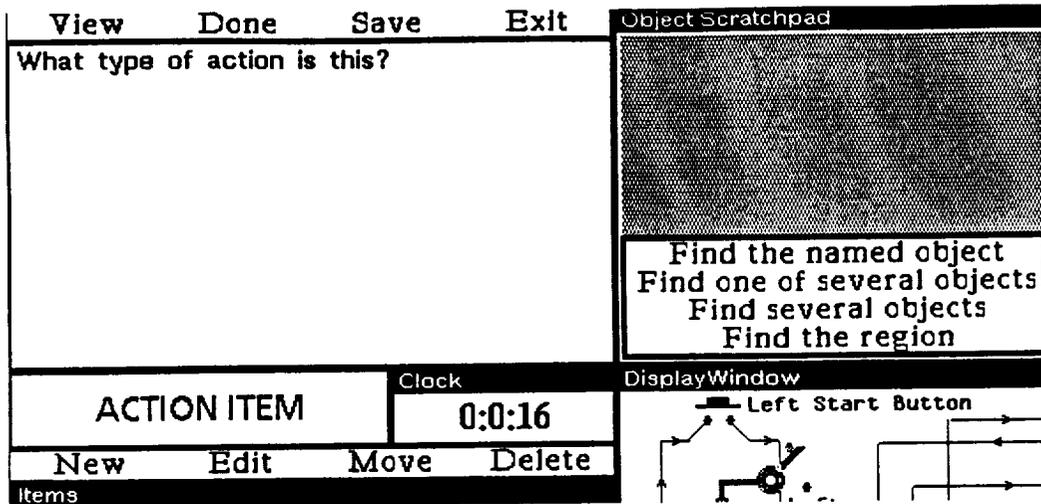
Action time start: 0

Action time end: 2147483647

Time to wait after action: 0.0

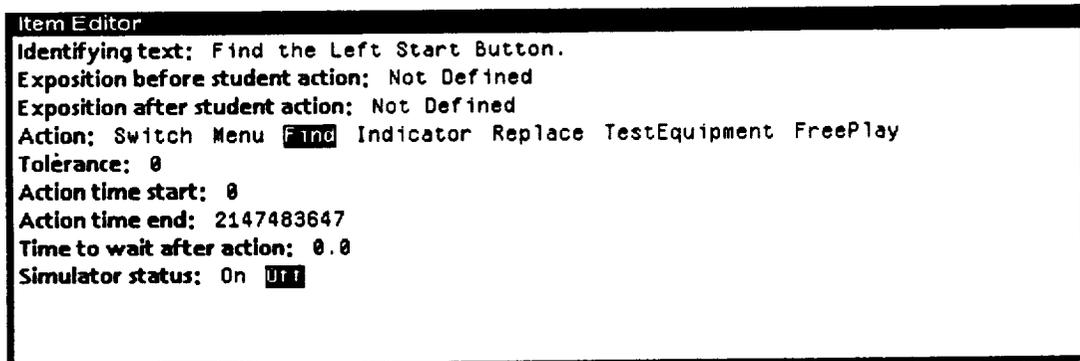
Simulator status: On

Leave the *Identifying text* field undefined for now. Click on the *Action* field. The item editor window will disappear and a message will appear in the left window asking you to select from the options menu or to throw a switch. Click on **Find Object** in the options menu, as shown below. A menu will pop up asking what type of action this is: finding one named object, finding several objects, or finding a region.



Click on **Find the named object**. The message window will then ask you to 'Select the object that the student must find.' The *left start button* is the switch at the top left of the simulation scene (see the figure above). Click on this switch.

The item editor window will now reappear. The identifying text field will have in it the phrase 'Find the Left Start Button.' You can now click in the identifying text field and edit this phrase to say whatever seems appropriate, such as 'Find the switch that starts the left engine' Very often, authors are happy with the automatically generated phrase, and no editing is necessary.



**Finding One of Several Objects**

Sometimes you may want to ask the student a question for which there is more than one correct selection. Suppose, for example, that you want to ask the student to click on one of the possible sources of electrical power in the EngineStarter simulation. RAPIDS II lets you author a *Find Object* type student action that accepts any of a set of objects as the correct answer.

To begin authoring your 'Find a source of electric power' item, choose the New command to create a new item. Click on the *Action* field in the item editor window, and choose **Find Object** in the option menu, just as you did

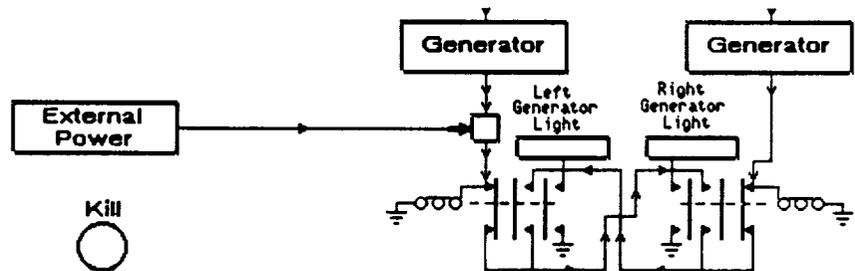
for the single-object find action above. When the *type of action* menu (shown at right) pops up, choose the second option.

- |   |
|---|
| Find the named object<br>Find one of several objects<br>Find several objects<br>Find the region |
|---|

You will then be prompted to select all the objects that would constitute correct student actions for this content item. This prompt appears in the message window, as shown below.

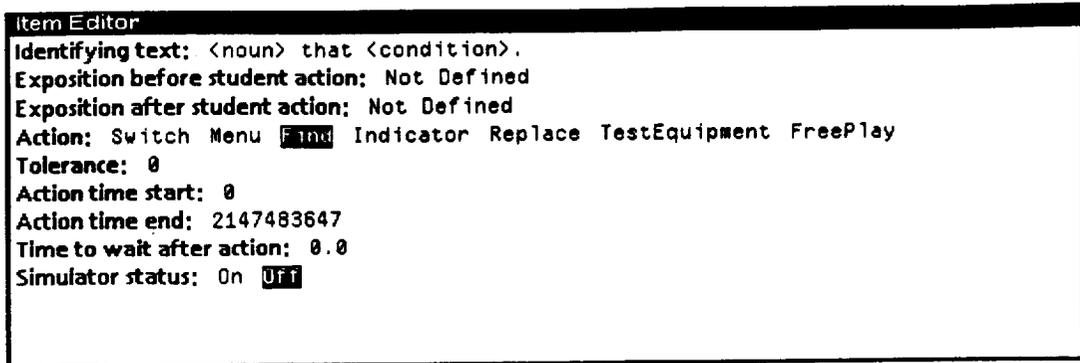
View	Done	Save	Exit
What type of action is this?  Select all the objects that would be correct responses for the student. When all objects have been selected, click on "Done".  Left Generator Right Generator External Power Emergency Power			

There are three possible sources of electrical power in the EngineStarter system: the two generators and the external power unit. Click on each of these in the simulator window.



As you click on these objects, their names will appear in the message window.

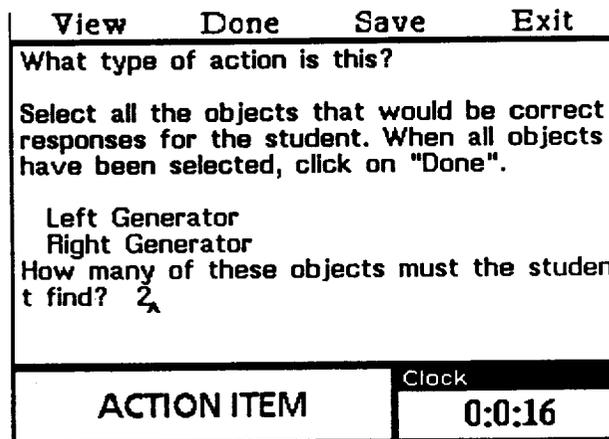
After you have designated all the correct choices, click on the Done command in the menu bar above. You have finished authoring that student action, so the item editor window reappears. The identifying text shown for all the *Find one of* types of student action is '<noun> that <condition>.' This prompt reminds you that you should edit this field so that RAPIDS II will ask the student to pick one of the objects that meets some constraint.



In this case, you should change this identifying text to read 'the sources of electrical power.'

**Finding Several Objects**

You can also create a student action that requires that several object designations be made. The **Find several objects** choice makes it possible to require that the student select all or some specified number of a set of objects in a simulation. When you choose this type of **Find** action, the message window asks you to select all the objects that meet the condition of interest. (See the figure below.)



Click on all the objects in the simulation window that meet the condition. (If you are working with a multi-scene simulation, use **View** to change scenes, if necessary.) When you have selected all the objects that would constitute correct responses to the **find** instruction, click on **Done**. A message in the window above the simulation window will ask how many of these objects are required in this content item. In other words, how many object selections that meet the specification should RAPIDS II ask the student to perform?

Type in a number and press the Enter key. You have finished authoring that student action, so the item editor window reappears. The identifying text shown will be '<noun> that <condition>.' Replace this with text that would make sense when preceded by the instruction "Find *n* \_\_\_", where *n* is the number of required identifications. If your identifying text is "parts that

provide electrical power under normal flying conditions” and *n* for a *Find several objects* content item is 2, then RAPIDS II will ask students to

**Find 2 parts that provide electrical power under normal flying conditions**

Using *Find several objects*, you can require identifying all of a class (without constraining the order in which students make their selections), simply by setting *n* to the number of objects in the class. You can also require only that a certain number of the objects described be selected by the students. RAPIDS II will permit these selections to be made in any order.

**Finding a Region** It is also possible in RAPIDS II to ask the student to click anywhere in a region on the simulation window. The third option in the *type of action* menu is **Find the region**. When you make this choice, the mouse pointer changes shape and you are asked to drag out a rectangle that surrounds the region in which you want the student to click.

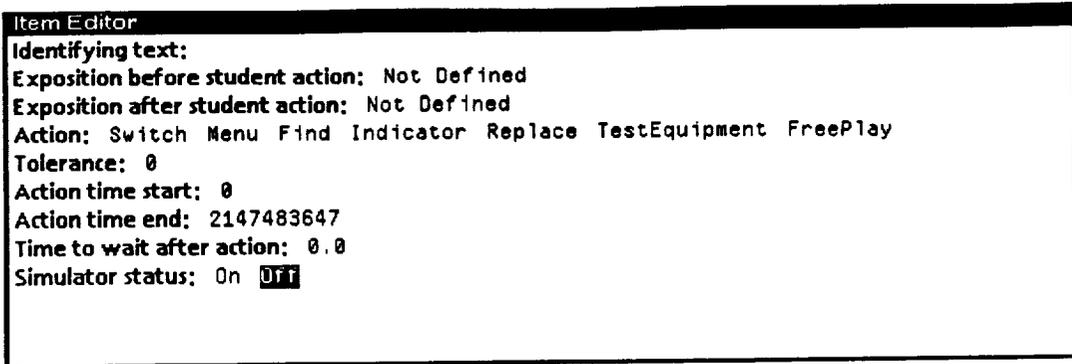
When you finish dragging out the window, the item editor window returns. The generated identifying text specifies the region that is to be selected. (See the figure below.) You can edit this text to create a more interpretable prompt for the student.

**Performing Switch Changes** Your lessons can also require specific switch manipulations by the student. Unlike a *Find Object* action, a switch manipulation will actually affect the state of the simulation, both during authoring and in the student environment. It is therefore easy to author a sequence of related simulation actions in a content unit.

Try authoring the actions of a ‘Left engine start on ground’ unit for the EngineStarter course. To carry out this example, begin by selecting *New* to create a new content unit.

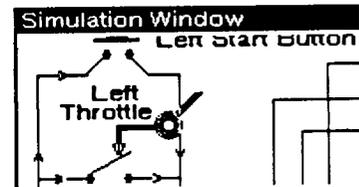
<b>Unit Editor</b>	
<b>Name:</b>	
<b>Comment:</b>	
<b>System Configuration:</b>	Initial State
<b>Exposition before student action:</b>	Not Defined
<b>Exposition after student action:</b>	Not Defined
<b>Order of presentation:</b>	Random <input checked="" type="checkbox"/> Sequential
<b>Present identifying text in test mode?</b>	<input checked="" type="checkbox"/> Yes <input type="checkbox"/> No

Give the new unit the name ‘Left Start On Ground’ and then choose *New* from the menu bar again. This time the item editor window will open, as in the figure below.

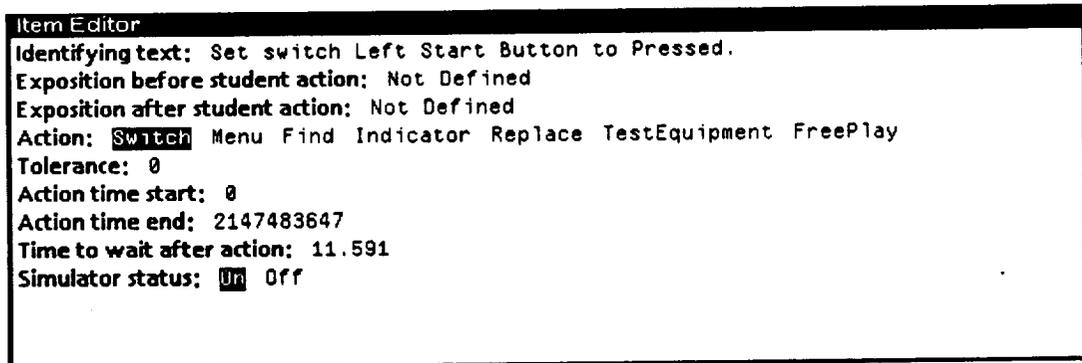


To author the required student action, first click on the *Switch* option in the *Action* field. The message window will prompt you to 'throw the desired switch'.

One of the strengths of the RAPIDS II approach to authoring is that it lets you specify many student actions by performing them. All you have to do to author this student action is to click on the left start button (the object at the top left corner of the simulation window).



After the effects of the switch throw have occurred, click in the clock window to stop the simulation. This marks the end of the required student action.



Just as with *Find Object* authoring, authoring of simulation actions also generates identifying text automatically. Naturally, you can edit these phrases if you want to. Click *Done* in the top menu bar to end the creation of this switch-based content item.

**Replacing an Object**

Your RAPIDS II lessons can require that the student replace a specified object. This feature is used in maintenance training sessions that call for the student to follow a sequence of troubleshooting actions that includes one or more replacements. Authoring such a student action is straightforward. First select **Replace** from the options menu.

Then, in response to the prompt in the message window, click on the object in the simulation window that the student should replace. When the simulated effects of the replacement have occurred, click in the clock window to stop the simulation. Then the item editor window will reappear and you can edit the identifying text.

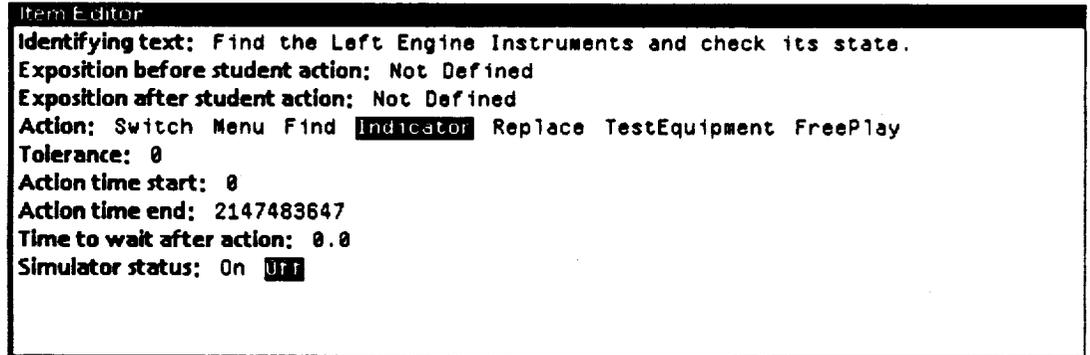
**Indicator Tests**

You can require that a student identify the value displayed by an indicator. RAPIDS II will help you compose a menu of choices for the student, including the value actually displayed.

You can experiment with this authoring capability using the EngineStarter system. Create a new item in an existing content unit. Click on the **Indicator** button in the actions menu. You will then be instructed to

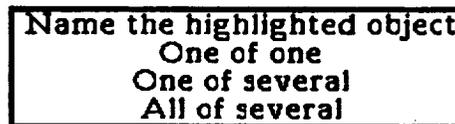
Select the indicator.

After you make an indicator selection (by clicking on the object), identifying text will appear in the top field of the Item Editor window. This text can be edited.



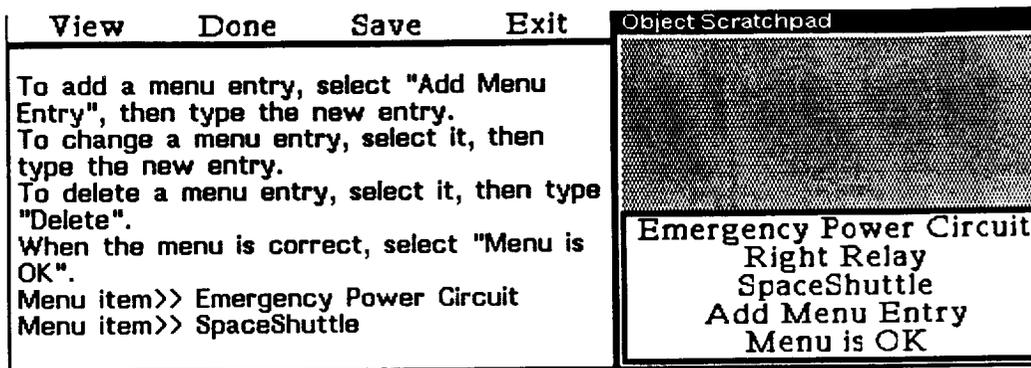
**Menus: Multiple-Choice Questions**

In RAPIDS II, you can present multiple-choice questions to the student. The choices are always presented in the form of a menu. RAPIDS II supports four types of multiple-choice questions. When you begin authoring a student action and pick the **Menu** command from the options menu, you will see a menu of four choices appear in the scratchpad area:



Whichever of these four types of menu-response student action you choose, you will be asked to build a menu of possible choices. The student interface for building these menus is the same as that used for authoring test equipment values, as described above.

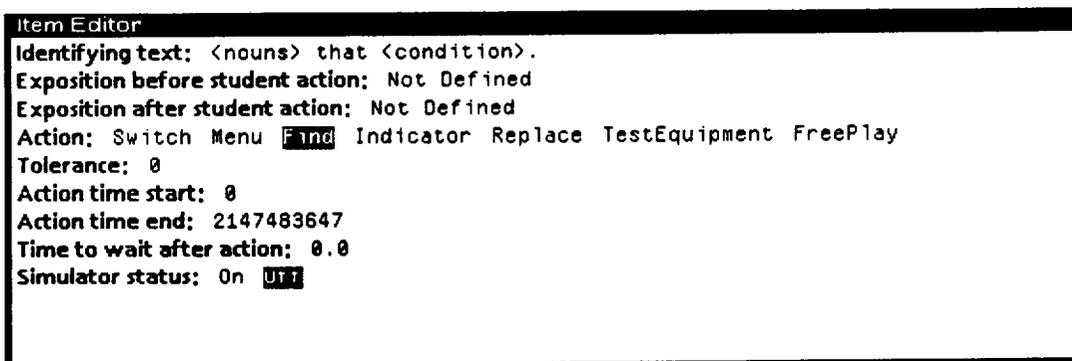
**One highlighted object.** After you choose the option, you will be asked to select the object that should be highlighted. Indicate the object by clicking on it. You can then edit the menu of choices, which will include the name of the item you indicated. You won't be able to exit menu editing until you provide at least one other choice.



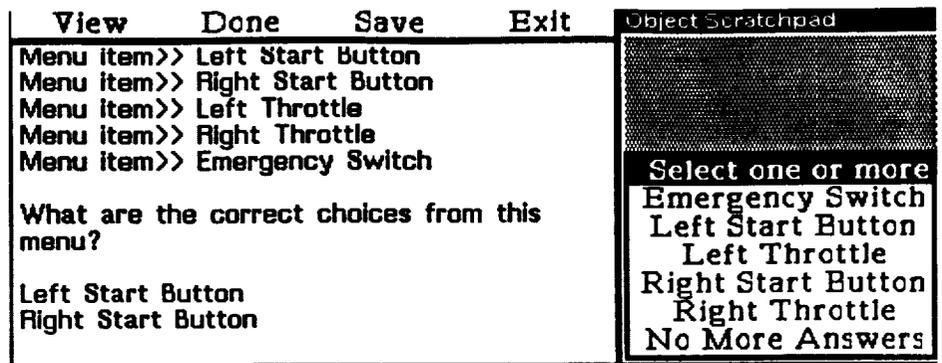
The editor will next ask you to indicate the correct choice in your menu. The item editor window will then reappear with the identifying text, "Name the highlighted object." You might want to change this to be more contextually appropriate.

**One of one.** This type of menu offers a set of text choices, with only one correct answer. When you choose this option, you are given the opportunity to construct a menu of text items. When you indicated that Menu is OK, you will be asked to pick the right answer from the menu.

The identifying text constructed for multiple choice menus should be edited to something more readable. It often makes sense to make use of the pre-exposition in defining a menu-type content item, as well.



**One of several.** This option is to be used when you are happy to take any one of several correct menu responses from the student. You will use the standard menu-construction options to build up your menu. Then, after you have chosen Menu is OK, you will be asked to select all the correct answers. When you have picked them all, click on No More Answers, at the bottom of the menu.



As in the case of single-answer menus, the composed identifying text will need to be edited.

**All of several.** This type of menu-based student action is one that requires that the student click on more than one answer in the menu. After you compose the menu using the same approach outlined above, you will be asked to designate all the choices that will be required for this menu.

Such a menu might be used to provide answer choices to a question such as "What are the possible power sources for a normal (non-emergency) engine start operation?"

## Expositions

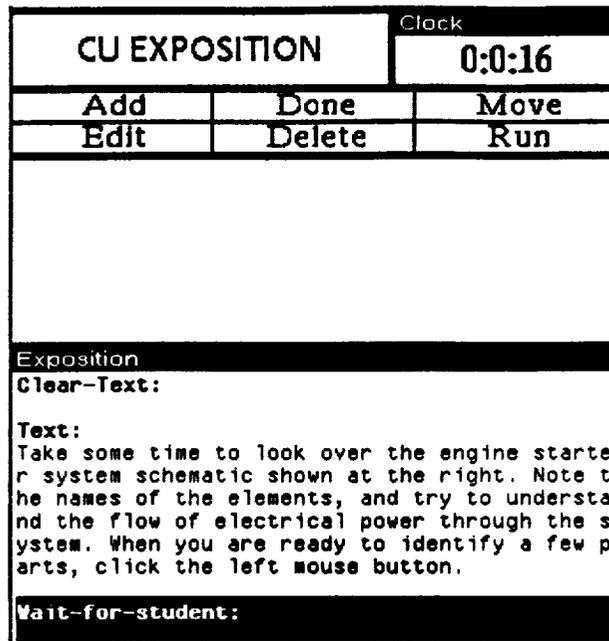
The RAPIDS II exposition editor is automatically invoked whenever you click on an exposition field in the unit editor window or in the item editor window. The exposition editor appears as a set of windows and menus to the left of the simulation window, in the same area that is used (in the student environment) by the message window.

<b>CU EXPOSITION</b>		<b>Clock</b> 0:0:16	<b>DisplayWindow</b> 
Add	Done	Move	
Edit	Delete	Run	
Text: Take some time to look over the engine starter system schematic shown at the right. Note the names of the elements, and <b>A</b>			
Exposition Clear-Text:			

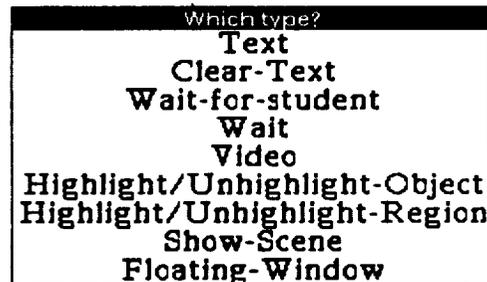
Using this editor, you can create expositions that

- present text in the message window
- clear the message window
- wait for a student click
- wait for a specified amount of time
- play a videodisc segment
- highlight an object in the simulation window
- highlight an arbitrary region in the simulation window
- change the scene displayed
- perform a *floating window* operation

Expositions consist of sequences of exposition elements. You can select an element by clicking on it. In the exposition shown below, a floating window element (one that opens a window) has been selected. When you choose the **Add** menu item, a pop up menu with two choices: *Before* and *After* appears. The new exposition item you are about to create can be placed either before or after the selected item.



After you choose *Before* or *After*, you will see a menu that offers the choice of all the different types of exposition items, as shown below.



If you choose **Text**, you will be invited to type your text element in the area just under the exposition window menu bar. The text word wraps automatically, so you don't have to use the return key. Typing the return key ends the entry of the text element. At this point the text you have entered will appear as an element in the list of exposition elements in the exposition editor. It will be preceded by **Text:** in bold, as you can see in the accompanying figures.

If you choose **Clear-Text**, the element label **Clear-Text:** will appear in the list of exposition elements. In the student environment, a Clear-Text element will have the effect of erasing the message window.

**Wait-for-student**, will also simply add a label to the list of exposition elements. At run-time, this element will make the cursor shape change to a mouse with the left button highlighted, signalling that the student must click to go on.

If you choose **Wait**, you will be asked how many seconds the exposition should wait before preceding to the next exposition element. Type a number and the return key.

The **Video** option in the *Which Type?* menu is used to specify an exposition element that will play a video segment to the student. When you choose this option, a menu will pop up that lets you specify which video segment you want to play.



If you select **Highlight/Unhighlight-Object**, then you will be prompted to click on the object that you want to highlight or unhighlight.

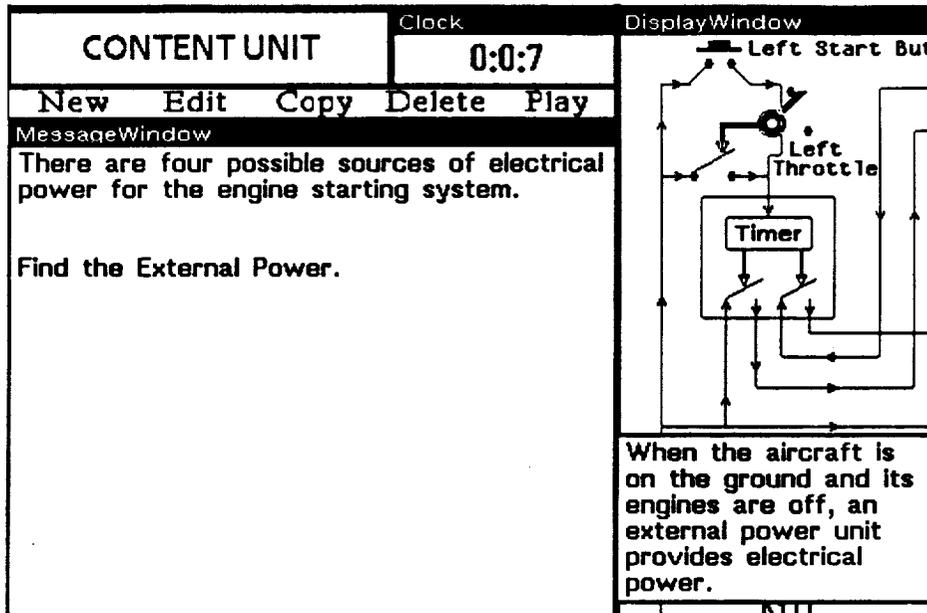
If you choose **Highlight/Unhighlight-Region**, then you will be asked to drag the mouse pointer from the top left to the bottom right of the rectangular area that you want to have highlighted.

The **Show-Scene** command on the *Which Type?* menu has the effect of changing the simulation scene displayed (both in the authoring and in the student environments). Use this command to change scenes under the control of an exposition.

If you select **Floating-Window** from the *Which Type?* menu, you will be presented with another menu that offers you the choice of floating menu operations. See the figure below.

<b>CU EXPOSITION</b>		<b>Clock</b> 0:0:4
Add	Done	Move
Edit	Delete	Run
Exposition		
<b>Clear-Text:</b>		
<b>Text:</b> There are four possible sources of electrical power for the engine starting system.		
<b>Wait-for-student:</b>		
<b>Highlight/Unhighlight-Object:</b> External Power		
<b>Floating-Window:</b> clear window		
<b>Floating-Window: reshape:</b> (-1 265 169 92)		
<b>Floating-Window: show text:</b> When the aircraft is on the ground and its engines are off, an external power unit provides electrical power.		

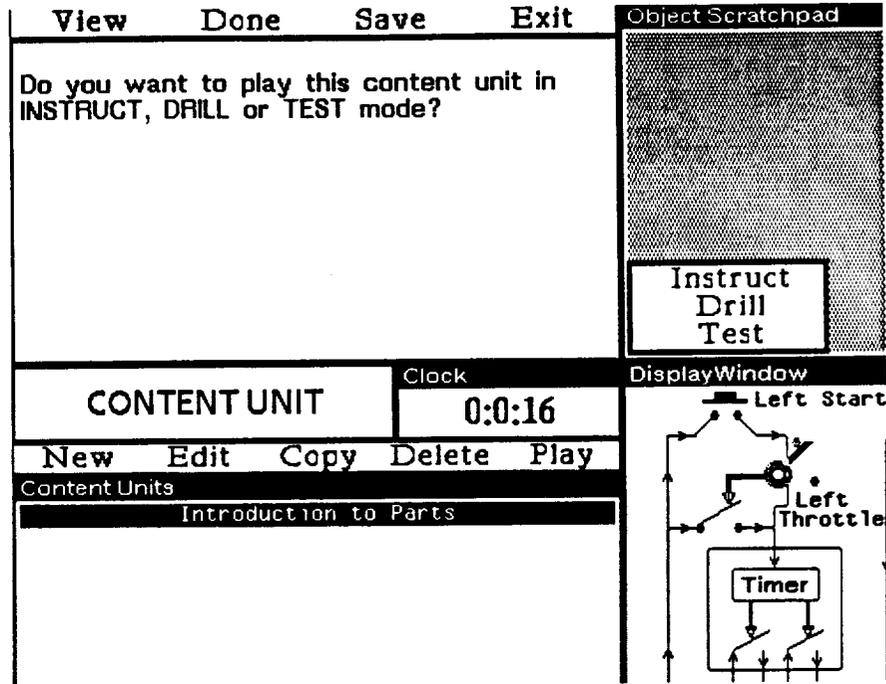
A floating window is a window of a shape and size determined by the author that 'floats' above the simulation scene. When you want to create and use a new floating window, your exposition will typically contain a sequence of Floating-Window operations: first a **Shape**, then an **Open**, followed by a **Text** action. If you want to, you can re-use the floating window, clearing it, moving it, and sending text to it again. Eventually, you will want to **Close** the floating window.



*A Floating Window with Text*

**Using *Play* to Test a Unit**

After building a content unit, it is a good idea to *Play* the unit in the content editor. After you select the *Play* option, you will be asked what mode you want to use. See Chapter 2 for a discussion of the three instructional modes.



## The Global Editor Commands

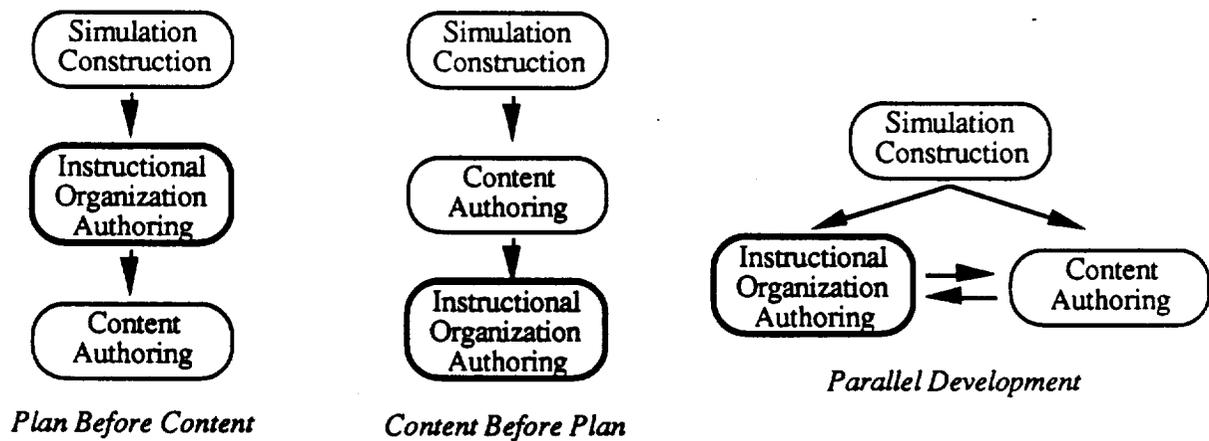
The menu bar at the top of the simulation window contains three commands, **Done**, **Save**, and **Exit**. As you have already seen, **Done** means that you are done defining a content item when the item editor window is open, and it means that you are done defining a content unit when the unit editor window is open.

- Save** Save will save the current set of content units, preserving any changes you made since the last save.
- Exit** Exit will close the content editor, ending the current authoring session. You will be asked whether you want to save changes.

## Instructional Organization

This chapter deals with creating and editing instructional plans. A training course must have an *instructional plan* (or *instructional organization*), which determines under what conditions each content unit will be presented to a student. A course's instructional organization also determines in what mode a content unit will be presented: instruct mode, drill mode, or test mode.

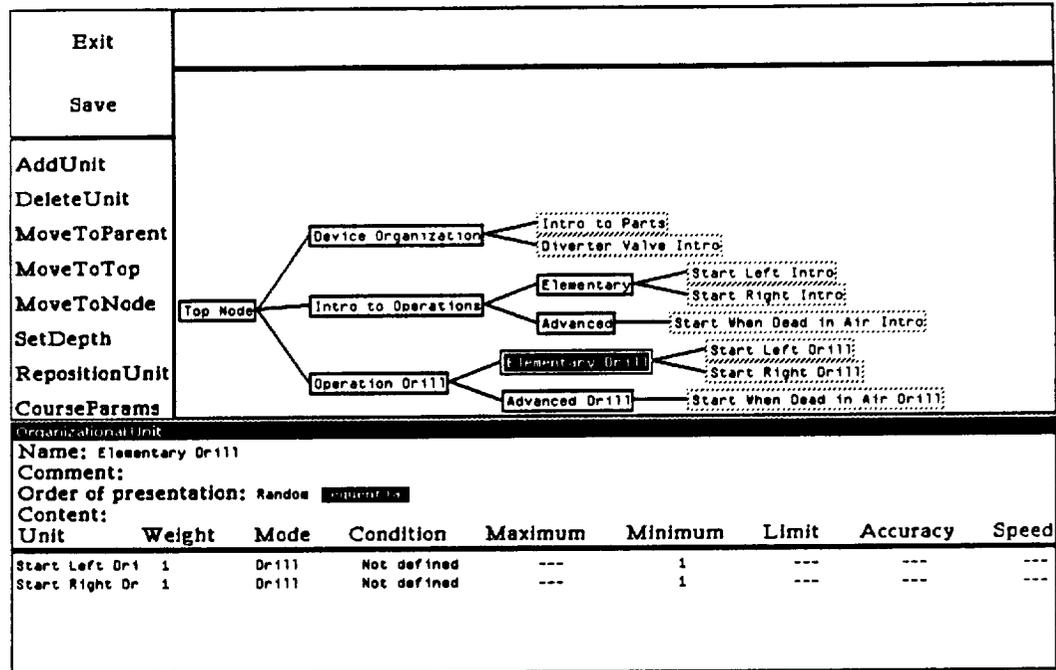
As Chapter 1 pointed out, instructional plans can be authored before, after, or in step with content unit editing.



In the figure above, the darker boxed items represent the process of building instructional plans.

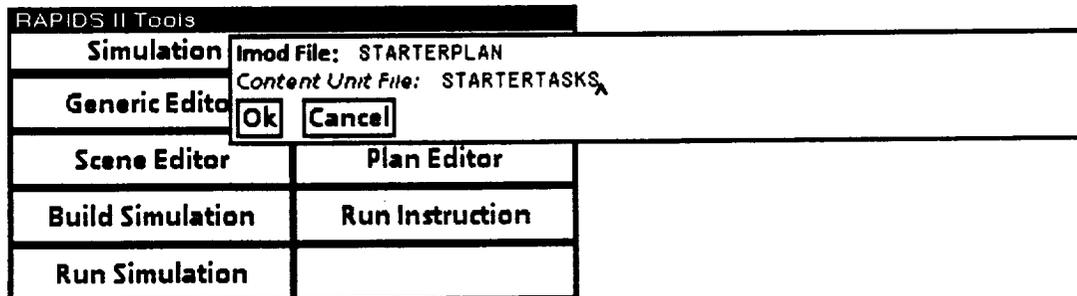
## A Sample Instructional Organization

The figure below displays an example of an instructional plan that takes advantage of defined content units created for a course called ENGINESTARTER.



### Starting the Plan Editor

To start the instructional organization editor, use the *Plan Editor* option in the RAPIDS II tools menu. You will have to provide two parameters, *PlanFile* and *ContentFile*. *PlanFile* is the name of the file that has the plan for the course; *ContentFile* is the name of the file containing the content units that the course should contain. *ContentFile* does not have to be specified if it has the same name as *PlanFile*.



Alternatively, one can start the organization editor for the simple EngineStarter course, by typing  
 (ImodEditor 'STARTERPLAN 'STARTERTASKS)  
 in an Exec window. The same name can be used for both files. The actual file names on disk have appended three-letter extensions that serve to specify which type of data they contain — instructional plan or content units. Most authors will prefer to use the *RAPIDS II Tools* menu.

**The Editor's Windows**

The largest window of the instructional plan editor is its tree window. This window displays a tree of organizational units that organize the content of a course in a hierarchical fashion.

To the left of the tree window is the instructional plan editor's menu. The top two commands (**Exit** and **Save**) apply to the planning session as a whole. The other commands have effects on particular nodes in the plan.

Above the tree window is a message window for the display and entry of text.

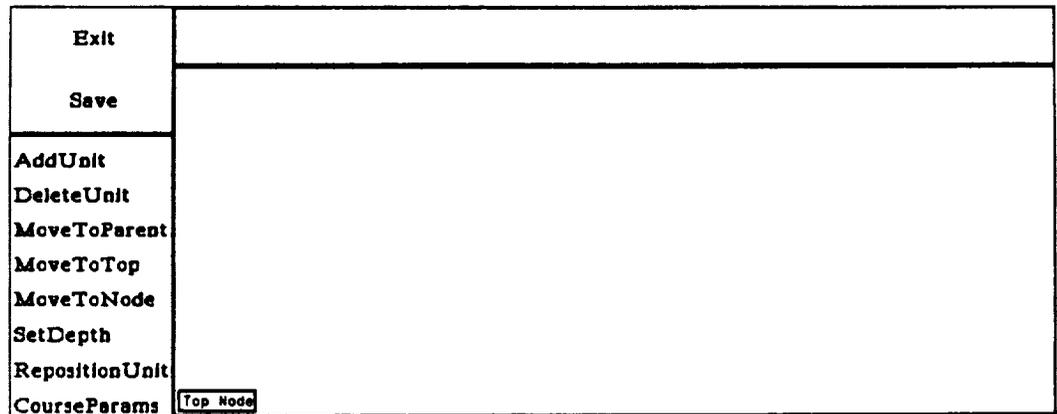
Below these windows is an organizational unit editor window. Here data associated with particular nodes can be entered and edited.

## Creating a New Instructional Organization

One way to get acquainted with the instructional plan editor is to build a plan from scratch. Begin by starting the editor with a new PlanFile name. For example:

```
(ImodEditor 'MYSTARTERCOURSE 'ENGINESTARTER)
```

The editor window will open with a display that includes only the top node of the new instructional organization, as in the figure below.



If you click on the node, a new window will open below, showing the data associated with the selected node. As the figure below shows, the top node

will be highlighted to show that it is selected. The organizational unit editor window below lists the data fields associated with an instructional unit.

Exit	
Save	
AddUnit	
DeleteUnit	
MoveToParent	
MoveToTop	
MoveToNode	
SetDepth	
RepositionUnit	
CourseParams	Top Node
<b>Organizational Unit</b>	
Name: Top Node	
Comment:	
Order of presentation: Random	
Content:	
Unit	Weight Mode Condition Maximum Minimum Limit Accuracy Speed

Associated with each unit called in a plan are these data fields:

- weight: the importance of the called unit (relative to the others in the list)
- mode: whether to execute a called content unit in Instruct, Drill, or Test mode
- condition: an optional expression that controls whether to present the unit
- maximum: the maximum number of times to present the unit
- minimum: the minimum number of times to present the unit
- limit: the time limit for the unit, in minutes
- accuracy: the accuracy score (%) required to complete the content unit successfully
- speed: the speed score required to complete the content unit successfully

Certain of these fields apply only to content units. The mode, accuracy, and speed fields have undefined values for organizational units. The data fields used to control the presentation of organizational units are condition, maximum, minimum, and limit. (Although accuracy requirements for organizational units are not authored as field values, accuracy scores computed and returned in the student environment.)

**Menu Commands  
in the  
Instructional Plan  
Editor**

<b>AddUnit</b>
<b>DeleteUnit</b>
<b>MoveToParent</b>
<b>MoveToTop</b>
<b>MoveToNode</b>
<b>SetDepth</b>
<b>RepositionUnit</b>
<b>CourseParams</b>

The menu to the left of the organizational tree window lists the top level commands of the instructional plan editor. The effects of most of these commands are with reference to the currently selected node. **Add Unit**, for example, adds a new descendent of the currently selected unit. **Delete Unit** deletes the selected unit.

Try adding a unit to the top node in your otherwise-empty course. When you click on **Add Unit**, you will be asked whether the new unit is to be an organizational unit or a content unit. Click the left button if you want an organizational unit; click the right button if your want a content unit.

<p><b>Exit</b></p> <p><b>Save</b></p>	<p>Click left button for an Organizational Unit-- click right button for a Content Unit.</p>
---------------------------------------	--

Choose the left button now, to specify that you want a new organizational unit. You will be asked to name the unit. Type the Return key when you have finished entering the name.

<p><b>Exit</b></p>	<p>Please name the new Organizational Unit &gt;&gt;_</p>
--------------------	--

Type in the name 'Device Organization' now. Notice that the unit editor window for the top node now displays data for the new node you have created, as is shown in the figure below.

Exit	
Save	
AddUnit	
DeleteUnit	
MoveToParent	
MoveToTop	
MoveToNode	
SetDepth	
RepositionUnit	Top Node — Device Organization
CourseParams	
<b>Organizational Unit</b>	
Name: Top Node	
Comment:	
Order of presentation: Random <input type="checkbox"/> Sequential <input type="checkbox"/>	
Content:	
Unit	Weight Mode Condition Maximum Minimum Limit Accuracy Speed
Device Organiz	1 Dr111 Not defined --- 1 --- N/A N/A

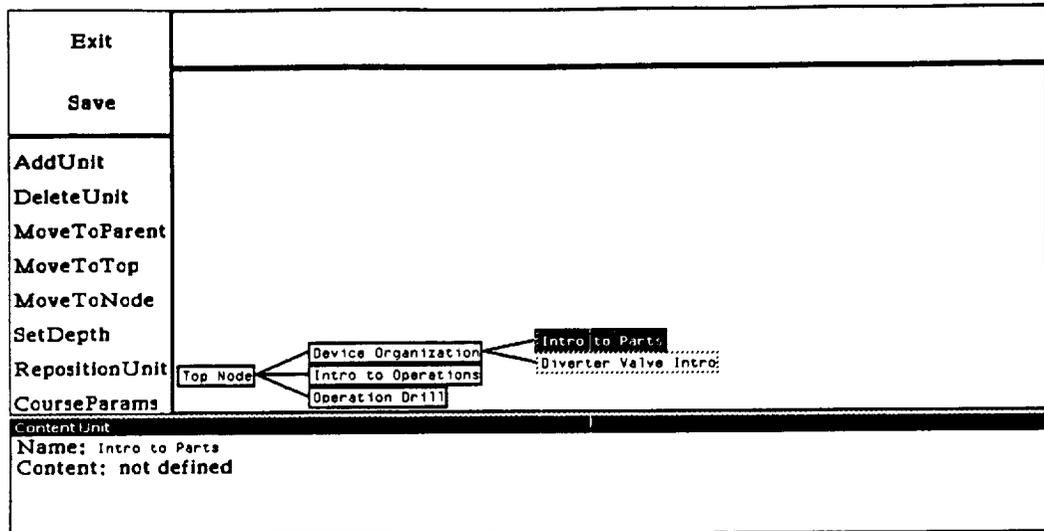
Use the same actions to create two new organizational nodes, one called 'Intro to Operations' and the other 'Operation Drill.' Information about these units will also appear in the organizational unit editor window for the top node, as in the bottom of the figure below.

Exit	
Save	
AddUnit	
DeleteUnit	
MoveToParent	
MoveToTop	
MoveToNode	
SetDepth	
RepositionUnit	Top Node — Device Organization Intro to Operations Operation Drill
CourseParams	
<b>Organizational Unit</b>	
Name: Top Node	
Comment:	
Order of presentation: Random <input type="checkbox"/> Sequential <input type="checkbox"/>	
Content:	
Unit	Weight Mode Condition Maximum Minimum Limit Accuracy Speed
Device Organiz	1 Dr111 Not defined --- 1 ---
Intro to Oper	1 Dr111 Not defined --- 1 ---
Operation Drill	1 Dr111 Not defined --- 1 ---

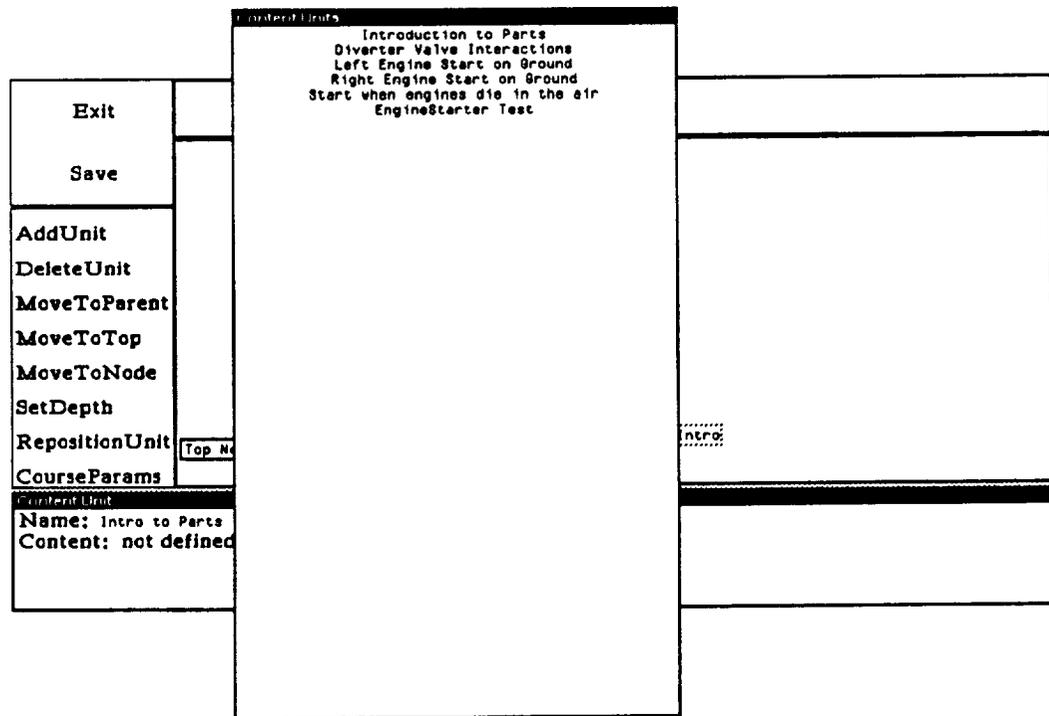
All three of these nodes created thus far were descendents of the top node. To create a descendent of the 'Device Organization' node, first click on that node to select it. Then choose the Add Unit command. This time, click on the right button to specify that the new node will be a content unit. Give the node a name like 'Intro to Parts.' At this point, your screen should show something like the figure below.

Exit	
Save	
AddUnit	
DeleteUnit	
MoveToParent	
MoveToTop	
MoveToNode	
SetDepth	
RepositionUnit	
CourseParams	
<b>Name:</b> Device Organization <b>Comment:</b> <b>Order of presentation:</b> Random <input type="checkbox"/> Sequential <input checked="" type="checkbox"/> <b>Content:</b>	
<b>Unit</b>	<b>Weight</b> <b>Mode</b> <b>Condition</b> <b>Maximum</b> <b>Minimum</b> <b>Limit</b> <b>Accuracy</b> <b>Speed</b>
Intro to Parts	1 Dr111 Not defined --- 1 ---

Create another new content unit called 'Diverter Valve Intro' and then select the 'Intro to Parts' content unit in the organizational tree. A window with content unit information will open below the tree window, in place of the organizational unit editing window. As is shown in figure below, the actual content of the 'Intro to Parts' node is not yet defined.



To define the content, you must associate the content unit node in the tree with one of the content units defined with the content unit editor (see Chapter 3). The instructional plan editor knows about the content units in the file referred to by the *ContentFile* parameter that was specified when the editor was invoked. (In this example, the file of content units is called 'ENGINESTARTER.') If you click the mouse in the area to the right of the Content label (where it says 'not defined' in this example), a list of the content units in that content file will appear, as shown below.



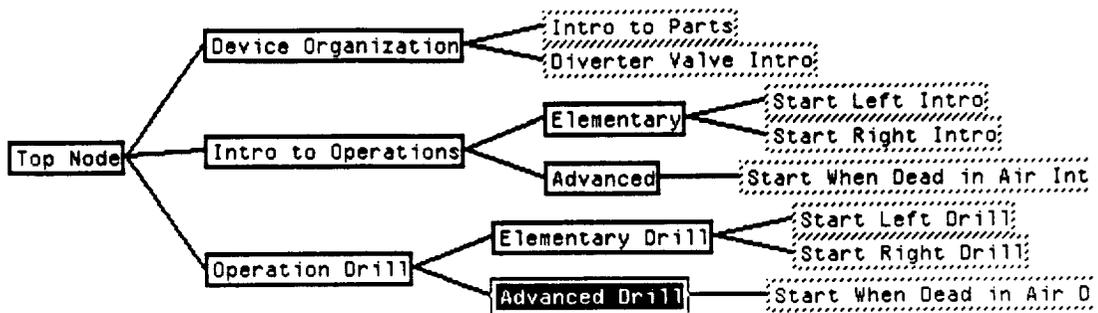
This list is a menu for the content of the selected unit in the tree window. In this case, click on the first item in the menu 'Introduction to Parts.' The 'not defined' in the bottom window will be replaced by the name you have chosen from the menu, as shown below.

Exit											
Save											
AddUnit											
DeleteUnit											
MoveToParent											
MoveToTop											
MoveToNode											
SetDepth											
RepositionUnit	<table border="1"> <tr> <td>Top Node</td> <td> <table border="1"> <tr> <td>Device Organization</td> <td> <table border="1"> <tr> <td>Intro to Part</td> </tr> <tr> <td>Diverter Valve Intro</td> </tr> </table> </td> </tr> <tr> <td>Intro to Operations</td> <td></td> </tr> <tr> <td>Operation Dr111</td> <td></td> </tr> </table> </td> </tr> </table>	Top Node	<table border="1"> <tr> <td>Device Organization</td> <td> <table border="1"> <tr> <td>Intro to Part</td> </tr> <tr> <td>Diverter Valve Intro</td> </tr> </table> </td> </tr> <tr> <td>Intro to Operations</td> <td></td> </tr> <tr> <td>Operation Dr111</td> <td></td> </tr> </table>	Device Organization	<table border="1"> <tr> <td>Intro to Part</td> </tr> <tr> <td>Diverter Valve Intro</td> </tr> </table>	Intro to Part	Diverter Valve Intro	Intro to Operations		Operation Dr111	
Top Node	<table border="1"> <tr> <td>Device Organization</td> <td> <table border="1"> <tr> <td>Intro to Part</td> </tr> <tr> <td>Diverter Valve Intro</td> </tr> </table> </td> </tr> <tr> <td>Intro to Operations</td> <td></td> </tr> <tr> <td>Operation Dr111</td> <td></td> </tr> </table>	Device Organization	<table border="1"> <tr> <td>Intro to Part</td> </tr> <tr> <td>Diverter Valve Intro</td> </tr> </table>	Intro to Part	Diverter Valve Intro	Intro to Operations		Operation Dr111			
Device Organization	<table border="1"> <tr> <td>Intro to Part</td> </tr> <tr> <td>Diverter Valve Intro</td> </tr> </table>	Intro to Part	Diverter Valve Intro								
Intro to Part											
Diverter Valve Intro											
Intro to Operations											
Operation Dr111											
CourseParams											
<b>Name:</b> Intro to Parts <b>Content:</b> Introduction to Parts											

Carry out a similar sequence of actions to assign the content unit called 'Diverter Valve Interactions' to the instructional plan node of the same name:

SetDepth											
RepositionUnit	<table border="1"> <tr> <td>Top Node</td> <td> <table border="1"> <tr> <td>Device Organization</td> <td> <table border="1"> <tr> <td>Intro to Parts</td> </tr> <tr> <td>Diverter Valve Intro</td> </tr> </table> </td> </tr> <tr> <td>Intro to Operations</td> <td></td> </tr> <tr> <td>Operation Dr111</td> <td></td> </tr> </table> </td> </tr> </table>	Top Node	<table border="1"> <tr> <td>Device Organization</td> <td> <table border="1"> <tr> <td>Intro to Parts</td> </tr> <tr> <td>Diverter Valve Intro</td> </tr> </table> </td> </tr> <tr> <td>Intro to Operations</td> <td></td> </tr> <tr> <td>Operation Dr111</td> <td></td> </tr> </table>	Device Organization	<table border="1"> <tr> <td>Intro to Parts</td> </tr> <tr> <td>Diverter Valve Intro</td> </tr> </table>	Intro to Parts	Diverter Valve Intro	Intro to Operations		Operation Dr111	
Top Node	<table border="1"> <tr> <td>Device Organization</td> <td> <table border="1"> <tr> <td>Intro to Parts</td> </tr> <tr> <td>Diverter Valve Intro</td> </tr> </table> </td> </tr> <tr> <td>Intro to Operations</td> <td></td> </tr> <tr> <td>Operation Dr111</td> <td></td> </tr> </table>	Device Organization	<table border="1"> <tr> <td>Intro to Parts</td> </tr> <tr> <td>Diverter Valve Intro</td> </tr> </table>	Intro to Parts	Diverter Valve Intro	Intro to Operations		Operation Dr111			
Device Organization	<table border="1"> <tr> <td>Intro to Parts</td> </tr> <tr> <td>Diverter Valve Intro</td> </tr> </table>	Intro to Parts	Diverter Valve Intro								
Intro to Parts											
Diverter Valve Intro											
Intro to Operations											
Operation Dr111											
CourseParams											
<b>Name:</b> Diverter Valve Intro <b>Content:</b> Diverter Valve Interactions											

As you continue to add nodes to the instructional plan, you will observe that the tree automatically spreads out to accommodate the newly defined elements. A course something like the one shown below is included in the release, with the name 'ENGINESTARTER.'



In this course, the same content units often serve as the content of different nodes. For example, a content unit called 'Left Engine Start on Ground' is used as the content of two nodes in the instructional plan above. Both 'Start Left Intro' and 'Start Left Drill' have this content unit as their content. The difference between them is the *mode* in which the content unit is to be played.

**Mode**

There are three modes that a content unit can be played in: Instruct, Drill, and Test. You can choose the mode that you want by clicking the mouse on the current value shown in the *Mode* column of data in the organizational unit editor window. A menu of the three mode choices will pop up. In the figure below, the default value of *Drill* has been changed to *Instruct* for one of the two content units.

MoveToTop

MoveToNode

SetDepth

RepositionUnit

CourseParams

---

**Organizational Unit**

Name: Elementary

Comment:

Order of presentation: Random **Sequential**

Content:

Unit	Weight	Mode	Condition	Maximum	Minimum
Start Left Inti	1	Instruct	Not defined	---	1
Start Right In:	1	Drill	Not defined	---	1

---



---

## Student Evaluation in RAPIDS II

As a student works through a RAPIDS II course, a number of data on his performance are recorded. The scoring of content items and content units is automatic. As an author, you can determine what should be presented to a student, based on his or her performance.

### Item Scoring

When an item is completed in Drill or Test mode, it returns two values:

1. an accuracy score: 1 if the student got the item correct on the first attempt; 0.5 if the item was correct on the second attempt; else 0
2. a speed score: the time required for the student to complete the item. (The completion time of an item is the time at which the student either gets the item right, gets it wrong twice, asks for the answer by clicking on **Don't Know**, or exceeds the time limit of the item.

If an item is not attempted within a certain amount of time, then RAPIDS II performs the action and presents the next item. The time limit for an item is two times the time limit for its parent content unit, divided by the number of items in the unit.

### Content Unit Scoring

The accuracy score for a content unit is the **percent correct**, computed as the number of items performed correctly, divided by the number of items in the content unit (thus items not attempted due to time limits count as incorrect). For a content unit that is attempted multiple times, the score is the higher of a) the last complete performance of the unit, and b) the score of the final, incomplete attempt. This algorithm recognizes a learner that nearly finished a final round, and performed well on it, even though the previous attempt was poorly done. It also does not unfairly charge a learner who has a low score on a final round that was just begun; instead it uses the score on the last complete round if it is higher.

The speed score for a content unit is the **total response time of all the items in the unit**. If a content unit is not completed, the items not attempted are essentially assigned the average time of the completed items. If a unit is attempted multiple times, and the final round is not complete, then the speed score for the unit is the speed score for whichever round was used in counting the accuracy score.

### Starting, Repeating, and Scoring Units

Since the instructional plan might require that a unit in progress be terminated prior to completion, due to a time limit at some level, RAPIDS II must guard against starting or repeating a unit when there is almost no time available. If a unit has no time limit, or the time limit is less than 3 minutes, then it is started unless a condition causes it to be skipped or unless the time available is less than the time limit. Thus short units are not started unless there is time to complete them. If a unit has a time limit greater than 3 minutes then the unit is started as long as there are at least 3 minutes available. Thus we avoid starting a unit and then ending it just minutes later because time has expired.

The following process is followed if 1) there are no Conditions that control initiation of the unit, and 2) there are at least 3 minutes available for the unit, or it has no time limit specified:

A unit presents all of the member items or units at least one time, unless its time limit (or a higher-level unit's time limit) is reached prior to completing the first presentation. If the minimum number of presentations is greater than 1, then the unit is repeated until the minimum is reached or time expires. At the conclusion of the minimum number of presentations, the student's performance is compared to the criteria to determine if the unit should be repeated further. An accuracy score is computed for all units as the weighted average of the accuracy scores of the completed units. A speed score is computed for content units as the total response times of all the called items in the unit.

The Unit is passed successfully and not repeated if:

- the accuracy score meets or exceeds the criterion, and
- the total response time is equal to or less than the criterion

If this test fails, then the Unit is repeated (and the student scores are recomputed) if:

- the unit has not been presented the maximum number of times, and
- the time limit for the unit, or a higher-level unit, has not been reached

Otherwise, the Unit is terminated, returning either the score on the previous repetition or the score on the last, incomplete, presentation, whichever is higher.

Example:

Here is the body of a particular organizational unit. It lists the member units and the parameters which control presentation of those units:

unit	wt	mode	cond	max	min	limit	accuracy	speed
1.1.2	-	I	-	-	1	20	-	-
1.1.2	1	D	-	4	1	15	85	7
1.2	2	-	-	1	1	20	-	-

This unit first presents Unit 1.1.2 in Instruct (I) mode. The student may study for up to 20 minutes, but, because the maximum number of repetitions was not specified, it will be presented only once. Then the same unit is presented in Drill (D) mode as many as 4 times. When the student can get 85% of the items correct, completing the unit in under seven minutes, the Drill is not repeated further. If the student cannot attain this proficiency in 15 minutes, or after 4 repetitions, then the Drill is ended.

Finally, unit 1.2 is presented. It is an organizational unit, and it is presented just once over the course of 20 minutes. It might consist of several content units or even more organizational units. If the student can learn the material and get through any drills or tests presented within the 20 minutes, then unit 1.2 ends successfully.

The student's score on this unit is a weighted average of the score on unit 1.1.2 (weight 1) and unit 1.2 (weight 2). Notice that the sum of the time limits of the parts of this example unit is 55 minutes. A planner might allocate 55 minutes to this unit, therefore. If s/he did, then the strategy would be to work through each unit until passed or until time runs out on that unit. But the time limit of a parent unit need not be the sum of its parts. By assigning *more* time to the unit than the sum of its component units, the planner can execute the following instructional strategy:

Work on each part for the specified time and number of repetitions. If the student can't reach proficiency on a unit in the time allocated to the unit, go on to the next unit. When all units have been presented, see if there is time remaining for the parent unit. If so, go back and work on the units that were not passed.
--

Thus a relatively rich set of instructional alternatives are provided, all achieved by the setting of a few values, rather than by involving programming-like skills.

## Authoring Conditional Course Sequences

Your RAPIDS II course can be authored so that certain content units are presented only to those students who need additional exposure to the material that those units deliver. This control over course sequence is determined by the conditions you place on units in the instructional plan.

**Conditions** You can author *conditions* that determine whether or not a unit will be delivered. A condition is an expression that will evaluate to either 'True' or 'False.' A simple RAPIDS II condition might be  
*Accuracy of Intro to Parts < 0.8*

Here we follow the steps required to build such a condition. The author has decided that the unit called 'Start Left Intro' can be skipped unless the student was less than 80% accurate on the earlier 'Intro to Parts' unit. The condition is created entirely by making selections in pop-up menus. When the 'Not Defined' condition is selected, the first menu pops up, as shown below.

Unit	Weight	Mode	Condition	Maximum	Minimum
Start Left Intro	1	Instruct	Not defined	---	1
Start Right Intro	1	Instruct	Not defined	---	1

This menu lets the user begin forming a condition. The simple condition to be formed here (*Accuracy of Intro to Parts is less than 0.8*) begins with a reference to a parameter associated with another unit, the *Accuracy* that the student exhibited in that unit.

Select next symbol

NOT

(

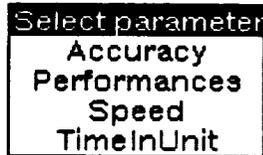
.

Number

Parameter

Unit

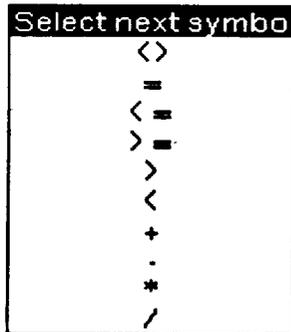
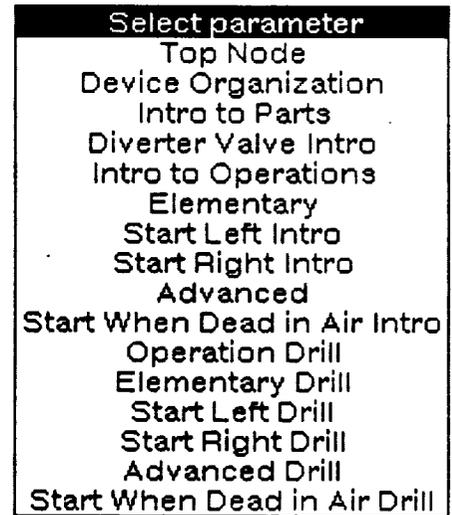
As soon as *Parameter* has been selected, the menu titled Select next symbol disappears, and a Select parameter menu appears in its place.



*Accuracy* is just one of four built in parameters that conditions can refer to. Accuracy on a unit is always a number between 0 and 1. It reflects the ratio of the number of items correct to the total number of items. *Performances* is a count of the number of times that a

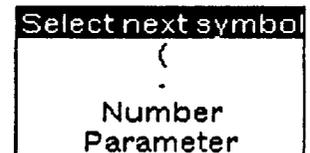
unit has been attempted. Speed is the time it took to complete the unit the last time that it was completed. TimeInUnit is a measure of the total amount of time the student spent in the unit in this session.

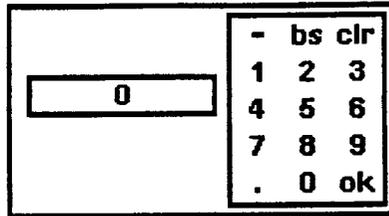
When the *Accuracy* parameter is selected, its menu goes away and another one appears that lists all the units in the course. The author chooses the one that the parameter —*Accuracy*, in this case — is to refer to. Choose 'Intro to Parts' in this menu. (As you complete portions of the condition, you will see them appear in the top window.)



The next symbol that this condition can contain is one of those in the menu that is then presented. Choose the *less than* symbol, <.

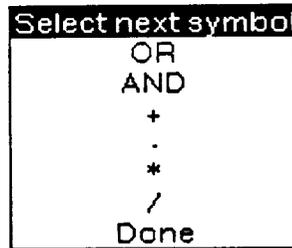
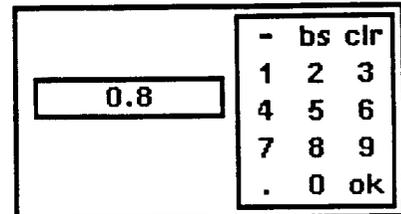
The next step in building this condition is to specify what the accuracy of the Intro to Parts is to be compared to. We could compare it with the value of a parameter of some other unit, by choosing *Parameter* from this menu. Since we want to compare it with 0.8, we should choose *Number*.





This number pad will appear. Its initial value is 0. You can click on '.' and '8' to enter the number. If you accidentally click on an unwanted digit, you can use 'bs' to backspace or 'clr' to clear the display.

When you have composed the number in the keypad display, click on 'ok' on the pad to accept it. The pad will disappear, and the next menu will pop up.



At this point, you could begin to build a complex condition — one with several condition parts, separated by 'OR' or 'AND.' For this simple condition, however, you can simply choose the menu item *Done*. The completed condition will be displayed in the top window of the editor.

Exit	Accuracy of Intro to Parts < 0.8
Save	

Organizational Unit			
Name: Elementary			
Comment:			
Order of presentation: Random <b>Sequential</b>			
Content:			
Unit	Weight	Mode	Condition
Start Left Int:	1	Instruct	Defined
Start Right In:	1	Instruct	Not defined

The condition data element for the 'Start Left Intro' unit will now be marked as 'Defined.'

**Course  
Parameters**

You may have noticed that there is no way to specify parameters for the top node in the tree in the organizational unit editor window. When you select the top node, the organizational units displayed in the editor window are its descendents. Of course, not all the options in that window would make sense for the top node. For example, it wouldn't make sense to define a condition for doing the course at all, since the condition could only refer to the values of performances in parts of the course. It may make sense, however, to be able to set a time limit for the course as a whole, and to determine how many times the course must be (may be) repeated. The **Course Params** menu command provides these features.

When you click on **Course Params**, you are presented with a menu that asks which parameters you want to set. You can determine the maximum and minimum number of repetitions for the course and its time limit.

<b>Which Parameter?</b>
Maximum Repetitions
Minimum Repetitions
Time Limit

---



---

## Local Editing in Large Trees

In large RAPIDS II courses, tree structures in the instructional plan editor may be very large. Naturally, you can use the scroll bars to move to any point that you want to work on in the tree, but this can take some time. Furthermore, it may be that you would rather not view large amounts of instructional plan material that are not relevant to the portion that you are editing. Every time you add, delete, or move a unit in a tree, the entire tree is redrawn. This drawing time can be noticeable and annoying in a large course.

<b>MoveToParent</b>
<b>MoveToTop</b>
<b>MoveToNode</b>
<b>SetDepth</b>

The solution offered by the instructional plan editor is to permit local editing of trees. Several of the the editor commands are related to these local editing options.

### Set Depth

The **SetDepth** command determines how much of the tree will be drawn during editing. The default setting is 1000. This means that 1000 levels of the tree will be displayed. You can greatly reduce the time it takes to redraw after a change is made by setting the depth to 3 or 4. Only that many levels of the tree will then be displayed in the window.

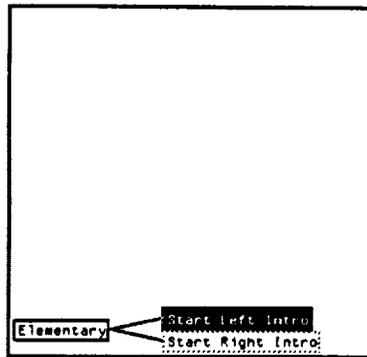
### Focus

If only a few levels of the tree are being drawn, you need to be able to change the focus of the editor. That is, you need a way to describe what part of the tree you want to work on. You do this by making a node the current focus. This node will serve as the root in the displayed portion of the tree. Focus is not the same as selection. The selected node is the one you last clicked on and it is highlighted in the tree window. The focused node is the one that is displayed as the root of the tree in the tree window. You *select* a node in order to do something to it or to examine data associated with its descendents. You *focus* on a node in order to change what part of the tree is being displayed.

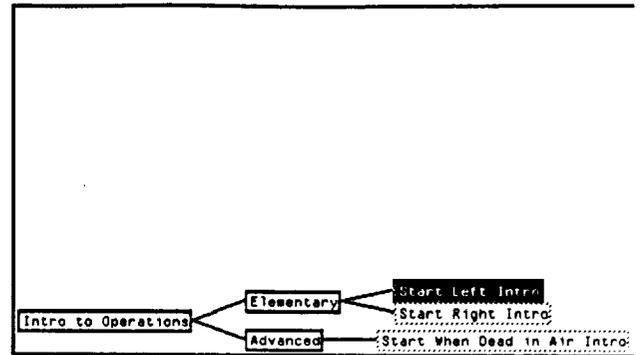
Focus is changed by using one of the three menu commands **Move to Parent**, **Move To Top**, and **Move To Node**. These commands only change what part of the tree is being displayed in the tree window of the editor; they don't have any effect on the structure of the course.

### Move To Parent

This command has the effect of changing the focus to the parent of the node that is currently the focus. Notice that it does not change the focus to the parent of the currently *selected* node.



*Before MoveToParent*



*After MoveToParent*

- Move To Top**      The **Move To Top** command has the effect of making the top or root node of the tree the focus of the editor's tree window. It does not change the depth, so portions of the tree furthest from the top node might not be displayed.
- Move To Node**      Choosing **Move To Node** has the effect of making the currently selected node the focus of the tree window. In other words, that node will be shown at the left edge of the window, as though it were the root of the currently editable tree. Remember that you can always move above this focus by using the commands **Move To Parent** or **Move to Top**.

## Instructor Utilities

---

---

In the process of building a course, you will test the content units you are constructing by using the **Play** command in the content editor. In order to test the instructional plan, you'll want to run the course in the student environment. Finally, when the entire course is complete, you'll need to install a turnkey instructional environment on the computers that will be used by students. This chapter describes how to test a course and how to build a turnkey training environment.

One feature of the RAPIDS runtime environment is specifically designed for instructors. By starting a session in a certain way, an instructor can examine the performance of selected students. This performance report is cast in terms of the structure of the course that is specified in the instructional plan. This feature is described later in this chapter.

---

---

### Testing a Course

Course contents can be tested in isolation in the content editor, using the **Play** command described in Chapter 3. Testing an entire course, however, observing the sequence of unit presentations under a variety of actual performance conditions, must be done in the RAPIDS student environment.

You can test a course by clicking on the *Run Instruction* command on the *RAPIDS II Tools Menu*.

---

RAPIDS II Tools	
<b>Simulation</b>	<b>Instruction</b>
<b>Generic Editor</b>	<b>Content Editor</b>
<b>Scene Editor</b>	<b>Plan Editor</b>
<b>Build Simulation</b>	<b>Run Instruction</b>
<b>Run Simulation</b>	

This will set up a training environment based on the simulation that has been built in your system. The executive and prompt windows will disappear from the screen. Then a number pad opens so that a student number can be entered. (See the figure below.)

<b>0</b>		
<b>1</b>	<b>2</b>	<b>3</b>
<b>4</b>	<b>5</b>	<b>6</b>
<b>7</b>	<b>8</b>	<b>9</b>
<b>ok</b>	<b>0</b>	<b>clr</b>

Enter Student Number.

Click on '0' and then on 'ok.' The student number 0 has the special characteristic that it inhibits the recording of student data. You can carry out a student exercise to test a course without preparing a student disk. RAPIDS will prompt you to type the name of the course. In many cases, there will be only one course associated with the simulation.

The set of windows that constitutes the student training environment (discussed in Chapter 2) will then appear on your screen. You can work through the course in the same way that a normal student would.

You can stop the course either by selecting **Quit** from the options menu or by picking the **Stop Session** choice from the menu that is presented after each unit. The student environment windows will close and you will then be presented with the number pad for the next student.

You can resume editing a course or a content unit file by typing in the appropriate command. In order to do this you will have to type in an exec window. The easiest way to do this is to get back your old exec window by typing a control-E, killing the RunRapids session. An alternative approach is to open a new exec window by using the right button menu in the screen background.

---

## Building a Turnkey Training Environment

Once your course is completely debugged, you will want to create turnkey environments that cannot be disturbed by ordinary students. A different command (Rapids, not RunRapids) is used to set up such an environment.

The command  
(Rapids)

works much like RunRapids (see above) except that it does not require a parameter. It builds the simulation and creates a clean student environment, without exec or prompt windows, and then it opens the student number pad. In addition, it locks the machine environment so that it cannot be altered. (For those of you familiar with Interlisp-D, it does a (SAVEVM) and a (VMEM.PURE.STATE T).)

Once the student environment has been locked in this way, it is impossible for students to do anything that will have a long-term negative effect on the environment. Even if a student somehow manages to open a Break window, you can always resort to hitting the Reset button on the computer to restore the environment to its initial state.

Student data is saved to a file with the name 'STUDENT#' where # represents the student number. This file is saved to a directory specified by the global variable StudentFileDirectory. If you do not set this variable, student data files will be saved to the current directory. If you want to store student data on a floppy, you must first set the directory variable, as in  
(SETQ StudentFileDirectory 'FLOPPY)

## Examining Student Data

Course authors and instructors can examine student data in the RAPIDS training environment. Begin by signing on as student 99. You will then be asked to select the instructor activity that you would like to carry out.

```

Select Operation
Examine Student Data
Run Rapids Session
Done
    
```

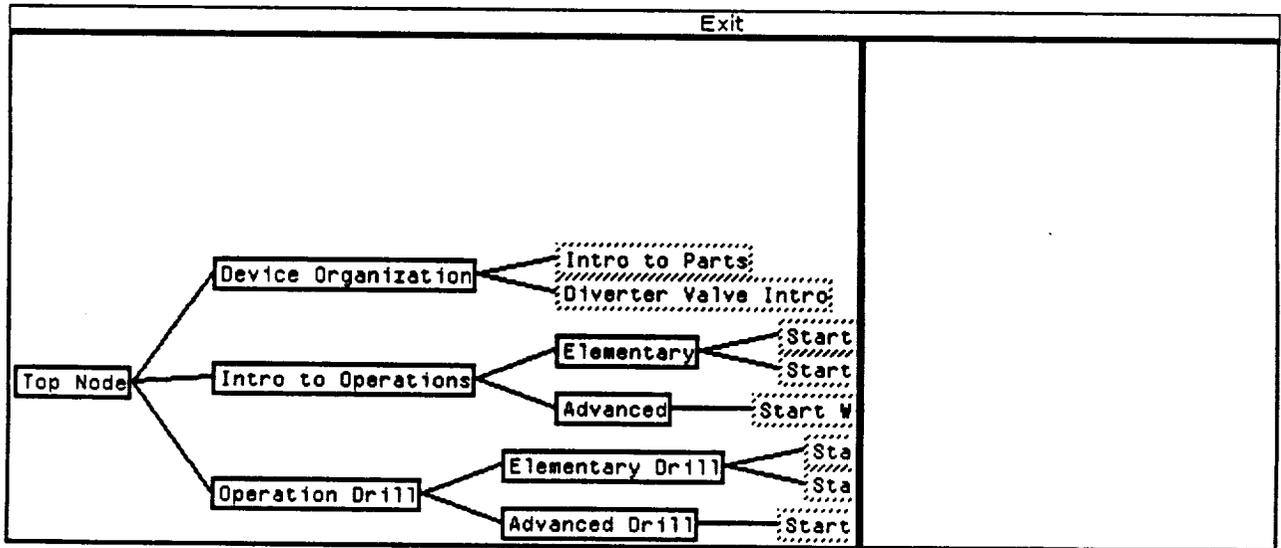
Choose Examine Student Data. You must then specify which student's data you would like to examine. The student number pad will appear again. Enter the number of the student whose performance data you would like to see. You will then be asked for which course you want to examine the data. Select the appropriate course name from the menu.

```

Select course
ENGINESTARTER <suspended>
    
```

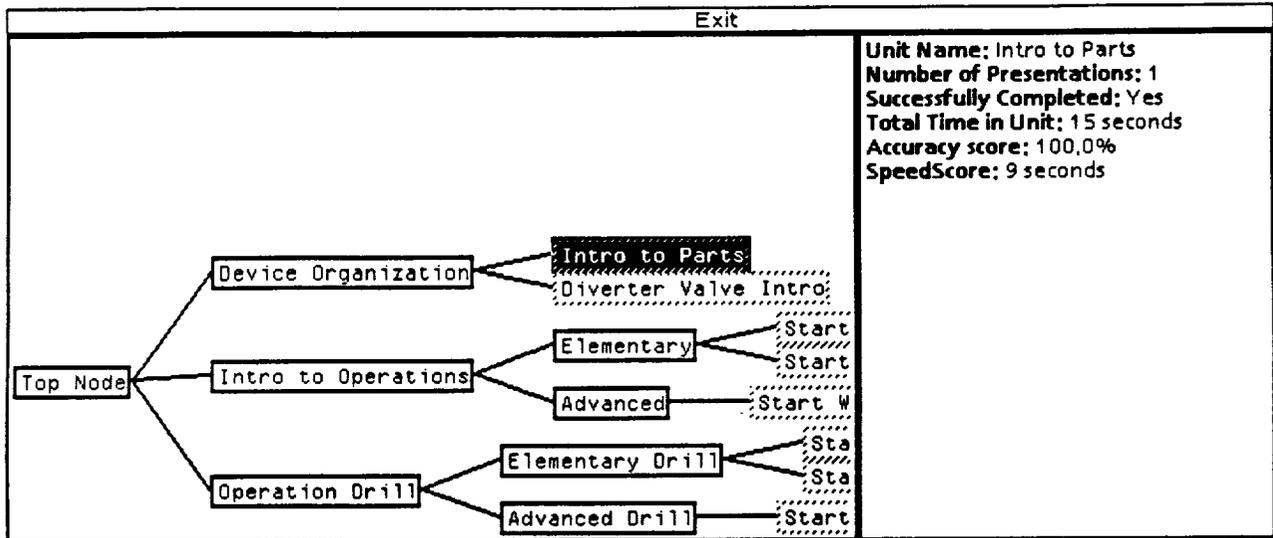
In the case shown above, the ENGINESTARTER course was suspended by the student.

After you select the course you want, a set of windows will open that are reminiscent of the instructional plan editor. In fact, the tree displayed is the instructional plan of the course.

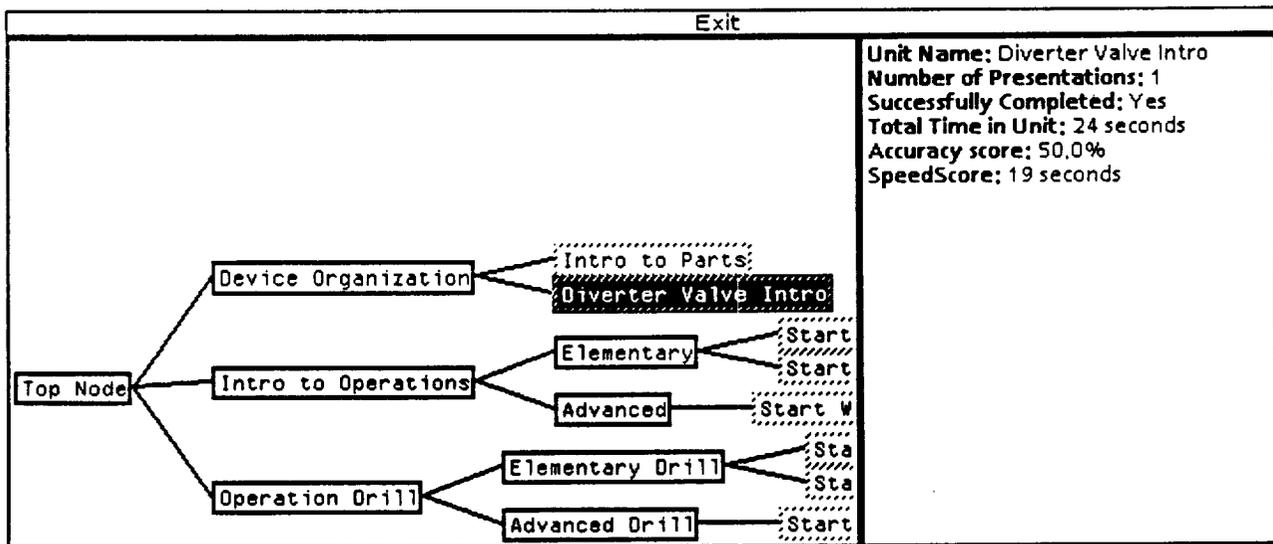


When you click on one of the nodes in this course tree, the node is highlighted (see the figure below) and the performance data associated with that node (for the specified student) is displayed in the window just to the right of the tree window.

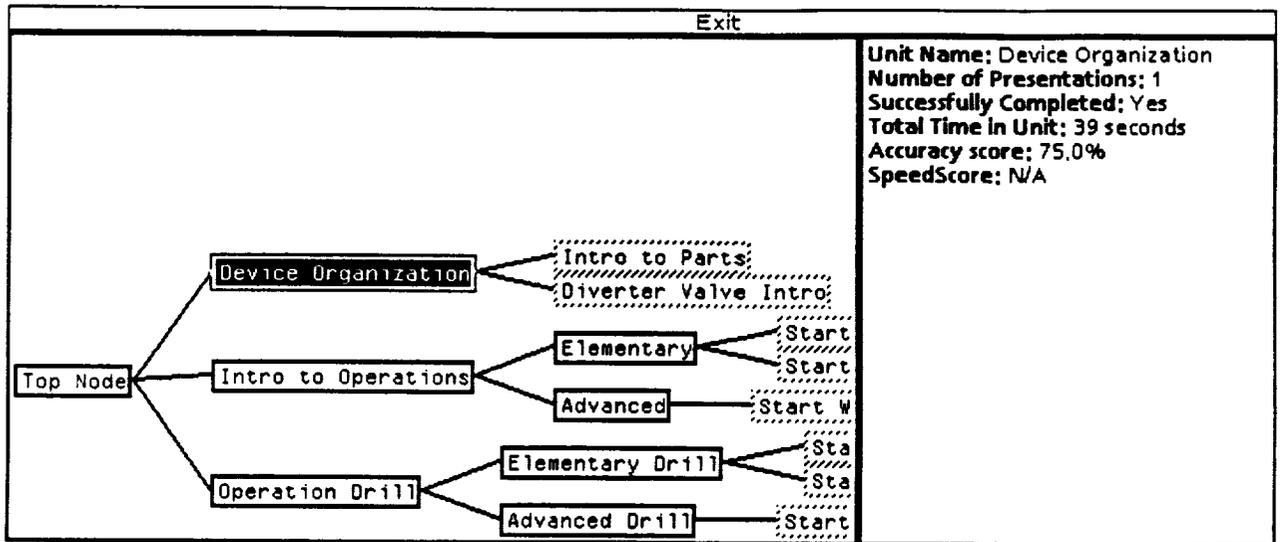
In this figure, the 'Intro to Parts' unit has been presented once. It was successfully completed in fifteen seconds with 100% accuracy.



The sibling node to 'Intro to Parts' is 'Diverter Valve Intro.' The depicted data shows that a student got only an accuracy of only 50% on this node.



Clicking on the parent node to these two nodes shows the performance measures interpreted for that node. Time is the sum of the times spent in the descendent nodes. Accuracy is the weighted average of the accuracy scores of the descendent nodes.



Experiment with this student data browser to learn more about how RAPIDS scores and evaluates student performance.

---



---

## References

- Hollan, J. D. 1983. STEAMER: An Overview with Implications for AI Applications in Other Domains. Presented at the Joint Services Workshop on Artificial Intelligence in Maintenance, Institute of Cognitive Science, Boulder, CO: October 4-6, 1983.
- Hollan, J. D., Hutchins, E. L., and Weitzman, L. 1984. STEAMER: An Interactive Inspectable Simulation-based Training System, *The AI Magazine*, 1984, 2.
- Norman, D. A. and Draper S. W. (Eds.) *User-centered system design: New perspectives on human-computer interaction..* Hillsdale, NJ: Lawrence Erlbaum Associates, 1986.
- Kieras, D. E. What mental model should be taught: Choosing instructional content for complex engineered systems. In J. Psotka, L. D. Massey & S. Mutter, (Eds.) *Intelligent Tutoring Systems: Lessons Learned*. 1988, Hillsdale, NJ: Lawrence Erlbaum Associates.
- Towne, D. M. A generalized model of fault-isolation performance. *Proceedings, Artificial Intelligence in Maintenance: Joint Services Workshop*, 1984.
- Towne, D. M. A generic expert diagnostician. In *The Proceedings of the Air Force Workshop on Artificial Intelligence Applications for Integrated Diagnostics*, 1986.
- Towne, D. M. The generalized maintenance trainer: Evolution and revolution. In W. B. Rouse (Ed.), *Advances in man-machine systems research, Vol 3*, JAI Press, 1986.
- Towne, D. M. and Johnson, M. C. *Research on computer-aided design for maintainability* (Technical Report No. 109). Los Angeles: Behavioral Technology Laboratories, University of Southern California, February 1987.
- Towne, D. M. and Munro, A. *Generalized maintenance trainer simulator: Development of hardware and software*. (Technical Report No. 81-9) San Diego: Navy Personnel Research and Development Center, 1981.
- Towne, D. M. and Munro, A. *Preliminary design of the advanced ESAS System*. (Technical Report No. 105) Los Angeles: Behavioral Technology Laboratories, University of Southern California, December 1984.
- Towne, D. M. and Munro, A. The Intelligent Maintenance Training System. In J. Psotka, L. D. Massey & S. Mutter, (Eds.) *Intelligent Tutoring Systems: Lessons Learned*. 1988, Hillsdale, NJ: Lawrence Erlbaum Associates.
- Towne, D. M. & Munro, A. Artificial intelligence in training diagnostic skills. In D. Bierman, J. Breuker, & J. Sandberg (Eds.) *The Proceedings of the Fourth International Conference on Artificial Intelligence and Education*. Amsterdam: IOS, 1989a.
- Towne, D. M. & Munro, A. RAPIDS: A simulation-based instructional authoring system for technical training. Technical Report No. 112, Los Angeles: Behavioral Technology Laboratories, University of Southern California, 1989b.
-

- Towne, D. M. & Munro, A. Tools for Simulation-Based Training. Technical Report No. 113, Los Angeles: Behavioral Technology Laboratories, University of Southern California, September 1989c.
- Towne, D. M. & Munro, A. Two approaches to simulation composition for training. In J. Psotka and M. Farr (Eds.), *Intelligent instruction by computer: From theory to practice*. London: Taylor and Francis, in press.
- Towne, D. M., Munro, A., Johnson, M. C. Generalized maintenance trainer simulator: Test and evaluation. (Technical Report No. 98) Los Angeles: University of Southern California, Behavioral Technology Laboratories, 1982.
- Towne, D. M., Munro, A., Pizzini, Q. A., & Surmon, D. S. Development of intelligent maintenance training technology: Design study. Technical Report No. 106, Los Angeles: Behavioral Technology Laboratories, University of Southern California, May 1985.
- Towne, D. M., Munro, A., Pizzini, Q. A., & Surmon, D. S. Representing system behaviors and expert behaviors for intelligent tutoring. Technical Report No. 108, Los Angeles: Behavioral Technology Laboratories, University of Southern California, February 1987.
- Towne, D. M., Munro, A., Pizzini, Q. A., Surmon, D. S., Collier, L. D., & Wogulis, J. L. Model-building tools for simulation-based training. *Interactive Learning Environments*, 1990, 1, 33-50.
- Towne, D. M., Munro, A., Pizzini, Q. A., Surmon, D. S., & Wogulis, J. L. ONR Final Report: Intelligent maintenance training technology. Technical Report No. 110, Los Angeles: Behavioral Technology Laboratories, University of Southern California, March 1988.
- Towne, D. M., Munro, A., Johnson, M. C., and Lahey, G. F. *Generalized Maintenance Trainer Simulator: Test and Evaluation in the Laboratory Environment*. (NPRDC TR 83-28) San Diego: Navy Personnel Research and Development Center, August 1983.

---

## Index

- accuracy 206, 215
  - of Instructional unit 14, 195
- accuracy score 202
- Action 177 - 186
- Add New Object (generic editor) 37
- Add New State 48
- Add Unit 196
- adding objects in the scene editor 91
- advantages of RAPIDS II 2
- Affects Commands 133
- All of several (menu) 186
- appearances 29
- arrow drawing primitive 57
- Attribute vii, 64
  - author-defined 100
- attribute and rule operations 129
- Attribute Data View 101
- Attribute Graphics View 103
- Attribute Handle 101, 140-162
- Attribute Operations menu 119, 129
- attributes for test equipment 159
- Background (generic editor Window Op) 61
- background elements, in scene editor 92
- bitmap drawing primitive 57
- box drawing primitive 58
- Build Simulation 17, 163
- BuildRapidsSimulation 15
- Bury
  - generic editor Window Op 61
  - scene editor Window Op 137
- Change Scene Name
  - scene editor Window Op 137
- ChangeGridSize 62
- Check Syntax 71, 78
- circle drawing primitive 59
- Clear-Text (in expositions) 188
- clock (simulation) 97
- Comment (content unit) 169
- Compile
  - scene editor 114
- computer aided instruction 1
- Condition (instructional unit) 13, 195, 205
- conditional connection 143
- conditional course sequences 205-208
- configuration options 173
- connecting attributes 140
  - with the mouse 143-153
- constant assignment rule 117
- Content Editor 164
- content item 9, 168, 177
  - Content Items menu 175
- content unit 8, 165-174
  - exposition 173
  - editor 163, 164
  - scoring 202
- continuous appearances vii, 49-52
- continuous controls — rules 72
- Continuous State 49-52
- Copy
  - content unit editor 167
  - drawing primitive 60
  - generic editor 38, 52
  - scene editor 109
- Copy to File (generic editor) 47
- corrective feedback 28
- course 17
- course authoring 3
- Course Params 208
- Create
  - scene editor 109
- Create Page
  - generic editor Window Op 62
- Crosshairs 62
- curve drawing primitive 59
- Cycle
  - generic editor 38, 52
  - scene editor 109
- deep simulation vi
- Delete
  - generic editor 38, 52
  - scene editor 109
- Delete
  - content item 167, 176
  - drawing primitive 60
  - unit 196
- Display Window Actions
  - scene editor 113
- Display-Window Operations 137-139
- DisplayGrid 62
- Don't Know 27
- Done
  - generic editor 63

- drawing primitive 60
- drawing operations 57-60
- drill mode 28
- Edit*
  - content item 175-177
  - content unit 166-174
  - scene editor *Window Op* 138
- Edit Pause Condition* 130
- Editor Operations Menu*
  - scene editor 84, 107-108
- elementary rule actions 78
- emphasis in authored presentations 24
- EngineStarter* 19
- error feedback 23
- examining student data 214
- Expand* 133
- exposition 9-10, 187-191
- exposition types 188
- external rules 74-78
- Find Object* 178
- Find One* 179
- Find several objects* 181
- Find the named object* 179
- Find the region* 182
- floating window 10, 24, 189-190
- generic behavior rules 30, 66-74
- generic editor 29-63
- generic objects 29-56
- Generic Objects menu* 89
- graphic rule actions 78
- Graphic Utilities*
  - generic editor *Window Op* 62
- Grid*
  - generic editor *Window Op* 61
  - scene editor *Window Op* 139
- Grid - On/Off*
  - generic editor *Window Op* 61
  - scene editor *Window Op* 139
- Handle*
  - generic editor 52
- handle data view 100
- handle graphic view 101
- Handle Operations*
  - generic editor 44, 54
- handles 4
- Hardcopy*
  - generic editor *Window Op* 62
  - scene editor *Window Op* 139
- hiding objects
  - generic editor 37
- Highlight/Unhighlight-Object* 189
- Highlight/Unhighlight-Region* 189
- highlighting 24
- how simulation works 96-99
- identifying text 9, 174, 177
- ImodEditor* 194
- IMTS vi*
- indicator 184
- Inspect* 132
- installing RAPIDS II 14-15
- Instruct mode* 28
- instructional content editor 8, 163-191
- instructional organization 11, 192-210
- instructional plan 4, 11, 192-210
- instructor utilities 211-216
- intelligent tutoring system 1
- internal rules 66
- invisible object 48
- item editor window 175
- item expositions 177
- item scoring 202
- Label*
  - scene editor 110
- levels of representation 7
- library of generic objects 6
- limit (instructional unit) 14, 195
- line drawing primitive* 59
- line-width drawing primitive* 60
- local editing in large plans 209-210
- locality of effect *vi*
- Make Connection* 154-159
- map file 163
- maximum (instructional unit) 14, 195
- menu selection 25
- message window 18
- minimum (instructional unit) 14, 195
- mode 13, 192, 195, 201
- modeling at the element level 4
- modes of instruction 28
- mouse actions, simulation mode 111
- mouse connections 151
- Move*
  - generic editor 44, 55
  - generic editor *Window Op* 62
  - scene editor 110
  - content item editor 176

- move drawing primitive* 60
- Move To Node* 210
- Move to Page* 47
- Move to Parent* 209
- Move To Top* 210
- multi-scene simulation 134-137
- multi-state object 33
- multiple-choice questions 26, 184
- Name* (content unit editor) 169
- needed attributes 122
- New*
  - Content Item 175
  - Content Unit 166
- new configuration 170
- Next Page*
  - generic editor *Window Op* 62
- object attributes
  - generic editor 38
- object bundle 133
- object data view 100
- object designation 20
- object graphic view 100
- Object Graphics*
  - Generic Editor 40
- Object Handles*
  - generic editor 41
- Object Info* 100
  - object info window 86
- Object Operations*
  - generic editor 36-48
  - scene editor 108-110
- object scratchpad 18
- One highlighted object* (menu) 184
- One of one* (menu) 185
- One of several* (menu) 185
- Open*
  - scene editor 110
- options menu 27
- organizational unit 12
- out of bounds values 120
- pause attributes 115
- Pause On/Off* 132
- Pause Rules* 115
- Pause/UnPause* 112
- pauses in the Scene Editor 130
- performances 206
- Play* (Content Unit) 167
- presentation order 174
- Previous Page*
  - Generic Editor *Window Op* 62
- Primitive-Ops* Menu 57
- Probe* 113, 160
- processes viii, 65, 97
- propagation of effect 4
- range of rotation 50
- range of translation 50
- real-time rule actions 79
- Redisplay*
  - generic editor *Window Op* 62
  - scene editor *Window Op* 139
- Rename*
  - generic editor 44, 55
  - scene editor 110
- Replace* (content item action) 183
- Restore State*
  - scene editor 115
- Resume simulation* 120
- Rotate*
  - drawing primitive 60
  - generic editor 45, 55
  - scene editor 110
- rule authoring 64-79
  - by menu 66-77
- rule data view 104
- rule graphics view 107
- rule operations menu 130
- rule syntax 80-82
- Rules*
  - generic editor 45
  - scene editor 110
- run-time corrections 117-123
- Save State*
  - scene editor 115
- Scale* 46
  - drawing primitive 60
  - generic editor command 56
  - scene editor command 110
- scaled scene window 121, 137
- scene 83
- Scene Editor* command 88
- scene editor windows 84
- scene icons 83, 134
- scene map 135
- scene navigation 135
- Schedule* 79
- scheduled events 97

- scoring units 202
- script (exposition) 11
- SEdit editor 105
- Set 133
- Set attribute value 119
- SetDepth 209
- Show-Scene (in expositions) 189
- Shrink
  - generic editor 63
  - generic editor *Window Op* 62
  - scene editor *Window Op* 139
- simulation attribute data view 104
- simulation attributes 97
- simulation construction 83-137
- simulation data 98-106
- simulation debugging 124-134
- simulation mode, scene editor 95, 111-116
- Simulation Operations* 87
  - scene editor 111-116
- simulation window 18
- Snap
  - scene editor 114
- specific test equipment 160
- speed (instructional unit) 14, 195
- speed score 202
- StartProcess* 79
- State Graphics* 30, 56
- State Operations* (generic editor) 48-56
- STEAMER 4
- StopProcess* 79
- student action 10, 178-186
- student evaluation 202-204
- student interface 17-28
- student number 212
- surface simulation *vi*
- switch changes 182
- switch sequence 172
- switch settings 22
- system attributes 100
- system configuration 9, 169
- System Trace* 116
- test equipment 159-162
- test mode 28
- Text* (in expositions) 188
- text drawing primitive* 59
- time limit for an item 202
- Trace Attributes* 116
- Trace On/Off* 133
- turnkey training environment 213
- undefined attribute window 117
- undefined attributes 117
- undefined value 119
- Unschedule* 79
- V-Flip
  - scene editor 109
- Video* (in expositions) 189
- View* 27
- viewing simulation data 99-107
- Wait* (in expositions) 189
- Wait-for-student* (in expositions) 188
- weight (instructional unit) 13, 195
- Who Affects Me* 102, 133
- Whom Do I Affect* 102, 133
- window operations
  - generic editor 61-63
  - scene editor 137-139



---

Copies of this publication have been deposited with the Texas State Library in compliance with the State Depository Law.

---