DOCUMENT RESUME

ED 290 438 IR 012 986

AUTHOR Cunningham, Robert E.; And Others
TITLE Chips: A Tool for Developing Software Interfaces Interactively.
INSTITUTION Pittsburgh Univ., Pa. Learning Research and Development Center.
SPONS AGENCY Office of Naval Research, Arlington, Va.
REPORT NO TR-LSP-4
PUB DATE Oct 87
CONTRACT N00014-83-6-0148; N00014-83-K-0655
NOTE 63p.
PUB TYPE Reports - Research/Technical (143)

EDRS PRICE MF01/PC03 Plus Postage.
DESCRIPTORS *Computer Graphics; *Man Machine Systems; Menu Driven Software; Programing; *Programing Languages
IDENTIFIERS Direct Manipulation Interface; Interface Design Theory; *Learning Research and Development Center; LISP Programing Language; Object Oriented Programing

ABSTRACT

This report provides a detailed description of Chips, an interactive tool for developing software employing graphical/computer interfaces on Xerox Lisp machines. It is noted that Chips, which is implemented as a collection of customizable classes, provides the programmer with a rich graphical interface for the creation of rich graphical interfaces, and the end-user with classes for modeling the graphical relationships of objects on the screen and maintaining constraints between them. This description of the system is divided into five main sections: (1) the introduction, which provides background material and a general description of the system; (2) a brief overview of the report; (3) detailed explanations of the major features of Chips; (4) an in-depth discussion of the interactive aspects of the Chips development environment; and (5) an example session using Chips to develop and modify a small portion of an interface. Appended materials include descriptions of four programming techniques that have been found useful in the development of Chips; descriptions of several systems developed at the Learning Research and Development Center using Chips; and a glossary of key terms used in the report. (EW)

University of Pittsburgh
LEARNING RESEARCH AND DEVELOPMENT CENTER

Chips: A Tool for Developing Software Interfaces Interactively

Robert E. Cunningham
John D. Corbett
and
Jeffrey G. Bonar

October, 1987

Technical Report No. LSP-4

This work was supported by the Office of Naval Research, under Contract No. N00014-83-6-0148 and N00014-83-K-0655. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the authors, and do not necessarily reflect the views of the U.S. Government.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	Approved for public release; distribution unlimited.
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
UPITT/LRDC/ONR/LSP-4	
6a NAME OF PERFORMING ORGANIZATION: Learning Research & Development Center, Univ. of Pittsburgh; 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION: Personnel & Training Research Programs Office of Naval Research (Code 1142PT)
6c ADDRESS (City, State, and ZIP Code): 3939 O'Hara Street Pittsburgh, PA 15260	7b ADDRESS (City, State, and ZIP Code): 800 North Quincy Street Arlington, VA 22217-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-83-K-0655
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
61153N	RR04206	RR04206-00	NR442c524

11 TITLE (Include Security Classification)

CHIPS: A tool for Developing Software Interface Interactively

12 PERSONAL AUTHOR(S)

Robert E. Cunningham, John D. Corbett and Jeffrey G. Bonar

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Technical	FROM _____ TO _____	1987, October 23	65

16 SUPPLEMENTARY NOTATION

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Human/computer interfaces; Graphical interface; Direct manipulation interface; Visual programming; Object-oriented programming; User interface management systems; Programming environments.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Chips is an interactive tool for developing software employing graphical human/computer interfaces on Xerox Lisp machines. For the programmer, Chips provides a rich graphical interface for the creation of rich graphical interfaces. In the service of an end user, Chips provides classes for modeling the graphical relationships of objects on the screen and maintaining constraints between them. Several large applications, including tutors for programming and electricity, have been developed with Chips.

Chips is implemented as a collection of customizable classes in Loops, the objected-oriented extension to Interlisp-D. The three fundamental classes provided by Chips are DomainObject, DisplayObject, and Substrate. DomainObject defines objects of the application domain, DisplayObject defines mouse-sensitive graphical objects, and Substrate defines specialized windows for displaying and storing collections of instances of DisplayObject.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION: Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL: Susan M. Chipman	22b TELEPHONE (Include Area Code): (202)696-4318; 22c OFFICE SYMBOL: ONR 1142PT

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted; All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Chips: A Tool for Developing Software Interfaces Interactively

Robert E. Cunningham, John D. Corbett, and Jeffrey G. Bonar

Learning Research and Development Center
3939 O'Hara Street
University of Pittsburgh
Pittsburgh, Pennsylvania 15260

Technical Report No. LSP-4

This work was supported by the Office of Naval Research, under Contract No. N00014-83-6-0148 andN00014-83-K-0655. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the authors, and do not necessarily reflect the views of the U.S. Government.

Abstract

Chips is an interactive tool for developing software employing graphical human/computer interfaces on Xerox Lisp machines. For the programmer, Chips provides a rich graphical interface for the creation of rich graphical interfaces. In the service of an end user, Chips provides classes for modeling the graphical relationships of objects on the screen and maintaining constraints between them Several large applications have been developed with Chips including intelligent tutors for programming and electricity.

Chips is implemented as a collection of customizable classes in the Loops object-oriented extensions to Interlisp-D. The three fundamental classes provided by Chips are:

DomainObject which defines objects of the application domain — the domain for which the interface is being built — and ties together the various functionalites provided by the Chips system,

DisplayObject which defines mouse-sensitive graphical objects, and

Substrate which defines specialized windows for displaying and storing collections of instances of **DisplayObject**.

A programmer creates an interface by specializing existing **DomainObjects** and drawing new **DisplayObjects** with a graphics editor. Instances of **DisplayObject** and **Substrate** are assembled on screen to form the interface. Once the interface has been sketched in this manner, the programmer can "build inward," creating all other parts of the application through the objects on the screen. Chips makes this easy by supplying simple and direct access to the source code and data structures of an application. Chips not only allows one to build powerful graphical interfaces, but provides the same sort of powerful graphical interface to the programmer building the interface.

Keywords: human/computer interfaces, graphical interface, direct manipulation interface, visual programming, object-oriented programming, user interface management systems, programming environments.

Typographic Conventions

Technical terms appearing in the glossary are italicized and underlined upon first use, i.e *display*. Menu selections are printed in a sans-serif font, i.e. Edit Mechanism. Class names are printed in a bold faced sans-serif font, i.e. **Substrate**.

Acknowledgments

We would like to thank several people who provided important help and support during the development of Chips.

We would like to acknowledge several people who developed applications using Chips, often putting up with and working around deficiencies and bugs in early versions of the system. They provided many helpful suggestions that were incorporated into the design, discovered problems that we had missed, and provided much needed evidence that our ideas were on the right track. These diligent folks are Andrew Bowen, Joyce Friel, Dan Jones, Steve Kalinowski, Debra Logan, Bob Merchant, and Jamie Schultz.

Stewart Nickolas contributed important ideas to the project as well as providing inspiration for what can be done with Xerox Lisp machines.

Arlene Weiner tried her best to teach us how to write. In the brief time she had to work with us, she helped us decide upon our audience and present our ideas coherently.

Doug Roesch did early work with the latest version of Chips and created a lab for beginning Chips users.

Marty Kent worked on FlowChips, a precursor to Chips, and left us with many good ideas.

Joyce Friel, Stewart Nickolas, and Doug Roesch read and commented on earlier drafts of this report.

Dr. Alan Lesgold, the associate director of the Learning Research and Development Center, is responsible for the resources we use in our work and responsible that those who grant us resources are satisfied with all our work in the Intelligent Tutoring Systems Group. Chips could have easily been written off as a fruitless digression without his faith in us and his vision. Alan provided much needed support and guidance, both professionally and personally, as well as creating an environment in which we could do our work. This project could not have happened without him.

Jeff Bonar started and managed the Chips project. The Chips system itself was designed and implemented in collaboration between John Corbett and Bob Cunningham. We gratefully acknowledge, however, that Chips was built using as much of Interlisp-D and Loops as we could rationally incorporate. Bob Cunningham did the lion's share of the writing of this document. John Corbett wrote earlier drafts, and participated in the writing.

Table of Contents

1. Introduction

Creating good human/computer interfaces is a notoriously difficult task. Furthermore, our current best estimates indicate that interface design consumes 50% of the time on a large programming project. Even with that large time budget, the interfaces produced are usually difficult to debug and modify. This problem is compounded by the lack of any theory, or even consistent design guidelines, that could guide the development of interfaces. Even the most carefully thought out interface is likely to need some redesign when tried with real users.

Chips has been created to simplify the development of sophisticated interfaces. In particular, Chips can cut the time needed to implement a prototype interface by a factor of ten. Chips allows interfaces to be designed, thoroughly tested, and then discarded for more effective designs. With the extended amount of empirical experience afforded by the use of Chips, there is the possibility for a comprehensive theory of interface to emerge.

To create an interface in Chips, the programmer uses graphic editors to mock up interface designs by drawing and arranging objects that appear on the computer's display. The application underneath the interface is created by building inward from this mock-up. Typically, a Chips user is building a direct manipulation interface (see, for example, Hutchins, et al. [1986].) A direct manipulation interface allows the user to command the computer by moving and selecting icons designed to behave like the objects they represent.

1.1 The Contribution of Chips

Chips supports the development of direct manipulation interfaces directly. Chips objects can be created, displayed, and manipulated directly. All the difficult algorithms for smoothly dragging an icon across the screen, having that icon interact correctly with other icons it moves over and near, and connecting mouse or keyboard behavior to underlying functionality are provided in Chips. Chips provides extensive support for editing the properties and behavior of an application interactively, through the interface itself. Finally, Chips allows an interface to be simply saved and restored. In summary, Chips allows the programmer to treat an interface as a object for inspection, manipulation, and design.

The key potential of Chips is that it provides a sufficiently high level interface design language that a theory of interface design can emerge. In particular, Chips supports a rich set of syntactic relationships for objects in a diagram. Although the key difficulty in a theory of interface design is relating the syntax of the diagram to the underlying semantics of the domain being represented in the diagram, the systematic syntax of Chips' diagrams allows for a direct attack on representing semantics. The phrase "syntactic relationships in a diagram" is meant to refer to the 2-D physical relationships between the graphic elements in a diagram. For example, one icon may be above another icon, connected to another icon with a line, or have various mouse-sensitive areas (places where a user can button and invoke a program).

1.2 Current Approaches to Interface Development

Most interfaces are written in traditional programming languages. These languages supply primitive elements, such as commands for drawing lines and printing text, leaving the programmer to construct more sophisticated objects such as menus. This is time consuming and often leads to complex and idiosyncratic interfaces.

User Interface Management Systems (UIMS) [italicized underlined words also appear in the glossary] improve this situation by packaging common elements of interfaces so they can be reused. In addition, if the UIMS itself has an interactive interface, it may be possible to create entire application interfaces without programming.

For many applications, a good UIMS is sufficient, however it is not clear what belongs in a good UIMS. So rather than providing several specific interface elements, Chips provides two generic interface elements and tools for specializing them.

1.3 The Chips Approach to Interface Development

To create an interface in Chips, the programmer uses graphics editors to mock up interface designs by drawing and arranging the objects that appear on the computer's *display*. It is only a slight over-simplification to say the application is created by drawing it on the display and adding functionality by "building inward."

1.3.1 Exploring and Testing Interface Designs

Because so little is known about interface design, it is useful to try out various designs, especially with potential users [Rosson, et. al., 1987]. The cost of this exploratory approach is prohibitive with traditional programming languages; this is true even with systems designed for exploratory programming such as the Interlisp-D programming environment [Sheil, 1983] unless the programmer is a real master of interface design. Using Chips, one can experiment with many interface designs in the time it takes to build a single interface without Chips.

For example, for the programming tutor Bridge, we designed and implemented six versions of a visual programming language in three months. We estimate that this would have taken at least a year and a half without Chips.

1.3.2 Object-Oriented Interface Design

Objects on the display are more than pictures; they are objects that respond to the user's actions, such as selection with the mouse, and interact with one another. For a complete introduction to object-oriented programming see *Smalltalk-80: the Language and Its Implementation* [Goldberg and Robson, 1983].

Object-oriented programming is based on the notion of objects interacting by sending *messages* to one another. An object is a semi-autonomous combination of a data structure and procedures for responding to messages. Message names, unlike procedure names in most programming languages, need not be unique, thus objects of different classes can use different *methods* to respond to a particular message.

A new *class* of objects can be defined by specifying only how it differs from an existing class of objects. The new class is said to *inherit* everything that it does not specifically define. The new class is called a *specialization* of those classes used to define it.

The generic interface elements referred to above are classes of objects; they are named **DomainObject**, **DisplayObject**, and **Substrate**. Objects which are *instances* of the class **DomainObject** or any specialization of the class **DomainObject** are called *domain objects*. Objects which are *instances* of the class **DisplayObject** or any specialization of the class **DisplayObject** are called *display objects*. Similarly, objects which are instances of class **Substrate** are called *substrates*. The terms domain object, display object, and substrate are used to refer to instances of these classes. Classes will be referred to explicitly and printed in a bold sans-serif font, i.e. **DomainObject**.

Substrates are objects which appear as rectangular regions on the display. They are specialized windows used to create domain objects and display their associated display objects. Domain objects are focal points that allow the combination of the various behaviors of the the Chips system. Display objects are mouse-sensitive objects with arbitrary pictures. **DomainObject**, **DisplayObject**, and **Substrate** are the basic classes for objects in interfaces. They inherit many of the common aspects of

graphical interfaces, yet the programmer can specialize any aspect of them and thus is not locked into the existing ways of doing things.

1.3.3 Controlling Programs by Manipulating Pictures

Chips is especially useful for constructing *direct manipulation interfaces* (DMI). These allow the user to command the computer by moving and selecting cartoon-like icons designed to behave somewhat like the objects they represent. The Apple Macintosh employs direct manipulation extensively and is widely considered one of the easiest computers to use for people who are not necessarily computer specialists. The advantages of direct manipulation are widely recognized, [Hutchins, et. al., 1986].

Unfortunately, DMI are often difficult to construct and difficult to modify once they are constructed The programmer needs to write programs to create the pictures, move the pictures around the screen, determine what picture the mouse is pointing to, what pictures on the screen represent, what to do when an icon is selected, and so forth. Although programming languages provide commands for drawing geometric figures and ways of sensing the mouse, these basic capabilities are far removed from the task of directly manipulating graphical objects.

Chips classes provide these aspects of DMI automatically. Once objects from Chips are created and displayed, they can be manipulated directly; selecting an object with the mouse cursor causes that object to animate and follow the cursor around the display or causes that object to display a menu of operations to be performed on it or on objects related to it.

Chips provides extensive support for editing the properties and behavior of an application interactively, through the interface itself. Every object of an interface that appears on the screen can be edited by selecting the object and choosing the aspect of the object to edit. This behavior is useful throughout the development process, so usually the programmer makes the application interface by adding behavior and only disables the default behavior when it might confuse unsuspecting users by allowing them to stumble into the program code and data structures.

Thus it is easy to assemble objects of the application interface on the display and having done that, to use these objects to access relevant portions of the application program code and data structures. This feature of Chips facilitates the entire software development process by providing convenient access to the program code and data structures; "What You See Is What You Get" moreover, "What You See You Can Edit."

1.3.4 Mocking Up the Interface

Using direct manipulation, a user creates an interface by drawing pictures of the interface objects and arranging them on the screen in appropriate places. The interface can then be saved to a file and recreated simply by loading the file. This allows a user to effectively mock up an application interface without programming.

1.3.5 Establishing Relationships Between Application Objects

Chips provides explicit means for establishing connections between domain objects. Chips defines *connections* between objects to reflect relationships between those objects both on the display and in the computer. Chips also defines *mechanisms* which allow aspects of a domain object to be implemented with a collection of domain objects, like the clockwork inside a clock.

1.4 Interlisp-D/Loops Implementation

Chips is an integrated extension to the Interlisp-D/Loops programming environment. Loops [Bobrow and Stefik, 1981] provides object-oriented programming with *multiple inheritance*. Both Loops and Interlisp-D [Sannella, 1985] provide a very sophisticated programming environment including graphical browsers and program inspection facilities. They run on Xerox 1100 Series workstations.

Chips performs well on the Xerox 1186, which is one of the least expensive and least powerful of workstation class computers. In light of this, we feel the concepts demonstrated by Chips are practical for almost any *workstation*.

2. Overview

The remainder of this report discusses Chips from several different perspectives. Section 3, **Chips Structures**, gives a detailed explanation of the major features of Chips. Section 4, **Chips Interactive Environment**, provides an in depth discussion of the interactive aspects of the Chips development environment. Section 5, **A Session with Chips**, presents an example session using Chips to develop and modify a small portion of an interface. Appendix A, **Special Programming Techniques**, describes four programming techniques that we have found useful in the development of Chips: self-inspecting code, a general purpose caching scheme, a fast bitmap intersection algorithm, and the EditWhen macro. Appendix B, **Applications**, describes several systems developed at the Learning Research and Development Center using Chips. The final section, **Glossary**, describes key terms used in this report.

3. Chips Structures

In this section the major components of the Chips system are presented: Domain Objects, Display Objects, Picture Specifications, Substrates, Event Queues, Connections, Mechanisms, Event Streams, and Display Streams. Finally, the strategy used for saving Chips objects is presented.

3.1 Domain Objects as Instances

Domain objects are instances of subclasses of the class **DomainObject** that combine the functionality provided by Chips through inheritance, including: displaying themselves on the screen, animating themselves, connecting themselves to other domain objects, defining their behavior in terms of other domain objects, saving themselves to a file, and editing their behavior and properties interactively

3.1.1 Display Objects' Graphical Data Structure

Each instance of a subclass of **DomainObject** defines one or more instances of the class **DisplayObject** that determine how the domain object is to be displayed. The domain object itself corresponds to an object in the application domain, while its display object determines how the domain object will display itself on the screen. For example, in our digital circuit editor, there is a class of domain object called **LightBulb**. It has display objects associated with it that determine how it will show up on the screen, but the domain object instance itself determines the object's behavior. It determines how to process inputs, controls its display objects in response to inputs, and connects to other domain objects.

Each class of domain object defines one or more instances of display object. These display objects are stored on the domain object class's IV, **displayObjects** as an association list of the form:

$$((tag_1\ displayObjectInstance_1) \ldots (tag_n\ displayObjectInstance_n))$$

Each instance of a domain object class stores one or more display objects in its **displayObjects** IV. These display object instances are copies of those stored on its class's **displayObjects** IV. Each display object stored with a particular domain object instance is currently displayed on the screen. When a display object is removed from the screen, it is also removed from its corresponding domain object.

Each display object defines a figure, mask, and map, stored in the IVs **figure**, **mask**, and **map**, respectively. The figure and mask are used for displaying instances in a substrate and the map is used for determining what part of an instance is located where, typically to see what part of a display object has been selected with the mouse cursor.

Each display object also defines several other IVs including:

object — the domain object that the display object represents

host — the substrate instance that contains the display object

displayStream — the display stream the object is displayed on (usually whe window of its host)

position — the position in the display stream that the display object is located

editor — the editor that is used to modify the display object; usually an instance of the class **DisplayEditor**

responsesToSelection — a form that determines the display object's response to being selected with the mouse cursor

physicalConnectors — a list of the physical connectors associated with the display object

The figure, mask, and each element of the map of a display object is stored as an instance of PictureSpecification or some subclass of PictureSpecification. Each instance of PictureSpecification has three IVs:

displayRepresentation — the represen that is used to display the picture on the screen; the default display representation is a bitm lowing for fast display using BITBLT

editRepresentation — a representation that allows the picture to be edited, presumably in a more convenient manner that the bitmap; the default edit representation is a list of vector graphic commands in a format that is recognized by the Display Editor

offset — a position that describes the location of the picture specification relative to the lower left corner of the display object it is stored in

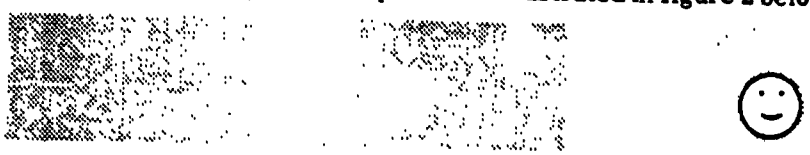
Thus, the actual representations of the pictures are separated from the operations necessary for displaying and manipulating objects on the screen. Each identical copy of a display object points to the same picture specification instances; new picture specification instances are only created as required due to local modifications made to a particular display object.

Because Interlisp-D bitmaps are rectangular and have only one bit per pixel, it takes two bitmaps to represent a figure with an arbitrary shape. One instance of PictureSpecification, the figure, defines the way the display object will appear on the display; a second instance, the mask of the display object, defines which areas of the display object are to be opaque and which transparent. The mask is black only where the corresponding location in the figure is considered opaque. For example, a SmileFace display object might have the following figure and mask (see figure 1 below):

(a) (b)

Figure 1. (a) The figure and (b) the mask of class SmileFace

Using this scheme, it is possible to display a figure of arbitrary shape on an arbitrary background. A simplified version of the display procedure is to erase the area where the figure is to be placed using the mask and then paint the figure. This process is illustrated in figure 2 below.



(a) initial background ( ) after erasing the mask (b) after painting the figure

Figure 2. Procedure for displaying an display object

Chips does its painting and erasing on a separate bitmap and then paints the result on the screen to avoid the flicker associated with erasing from the screen. This technique is called double buffering.

Note that this procedure does not constrain the figure to be closed nor composed of a single part. Figure 3 shows the possible combinations of figure and mask and what will be displayed on the screen with each combination.

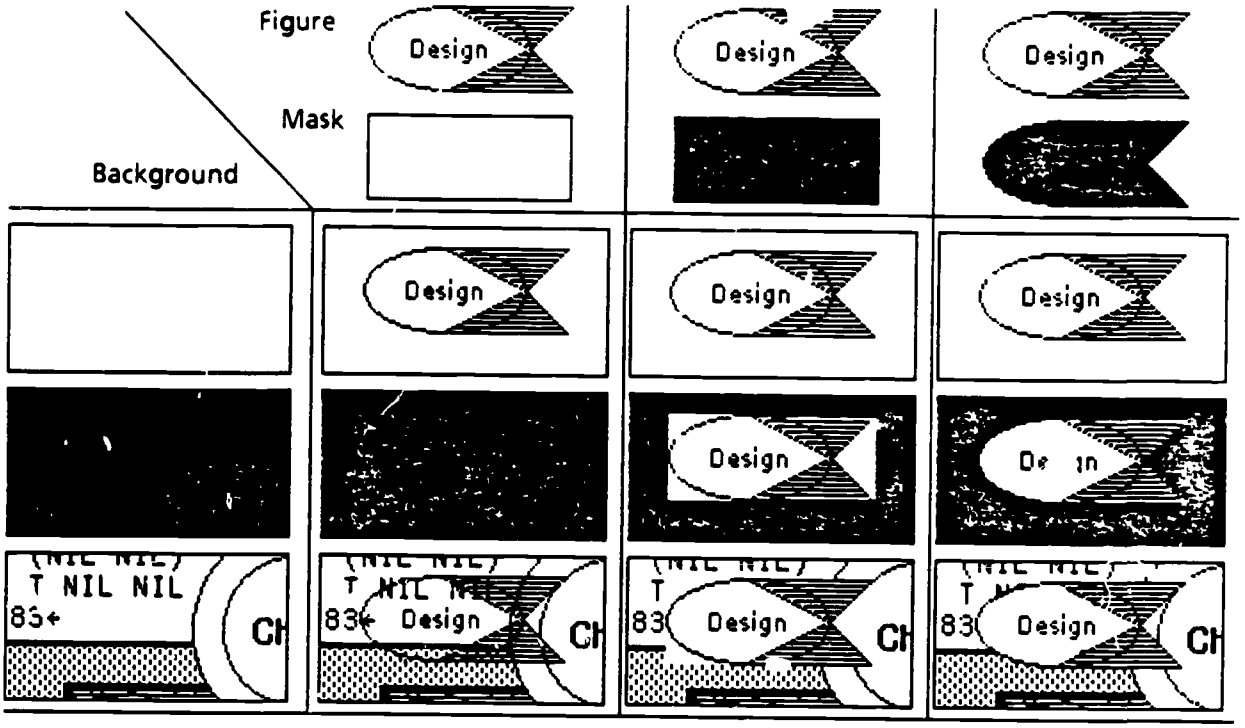


Figure 3. Displaying a figure with various masks on various backgrounds

The map is a list of elements that name the mouse-sensitive parts of the display object. Each element contains an instance of **PictureSpecification** and a tag, a mnemonic way to refer to the map element. The map is a list of the form:

$$(tag_1\ PictSpecInst_1\ (tag_2\ PictSpecInst_2)\ (tag_3\ PictSpecInst_3\ (tag_4\ PictSpecInst_4 \ldots]$$

The map is treated as a tree. The root contains the region that bounds the entire display object. The root is followed by *subregions* that may in turn have subregions, and so on, that distinguish different parts of the display object. To determine if a display object has been selected and what part was selected, a depth-first search is performed on the map. The subregions are considered to be contained in their region.

3.1.2 Multiple Display Objects and Multiple Picture Specifications

In Chips, there are two ways of representing various kinds of multiple display representations with a particular domain object: multiple display objects and multiple picture specifications.

A domain object may have more than one instance of **DisplayObject** associated with it, providing more than one view onto that domain object. This could be used, for example, with a business graphics application, with a domain object representing gross receipts having a display object that displays a number in one window, and a barchart representation of the value displayed in another window.

A display object may also have more than one set of picture specification instances associated with it. Each of the following IVs of display object have a property, tagList, which stores information concerning alternate picture specification sets and the display object's corresponding behavior when a

particular set is used: figure, mask, map, and physicalConnectors The tagList property stores an association list of the form:

$$((tag_1\ form_1) \ldots (tag_n\ form_n))$$

that associates certain forms with corresponding tags. Each display object also has an IV, tag, which stores the current tag being used.

3.1.3 Physical Connectors

Elements of a display object's map may be physical connectors, establishing the subregion they define to have special significance to another display object landing on that subregion. This can be used, for example, to establish physical attachment between display objects. In our digital circuit editor, the display object for the **ANDGate** domain object (see figure 4) has three physical connectors, two representing the input leads of the and gate and one representing the and gate's output. When one end of a wire is placed on top of one of these physical connectors, the wire attaches to the associated lead.

Physical Connectors

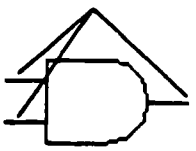


Figure 4. Physical connectors for the display object of an ANDGate

Physical connectors are stored in the physicalConnectors IV of a display object in the form:

$$((PictSpecInst_1\ position_1) \ldots (PictSpecInst_n\ position_n))$$

where position is the position of the connector relative to the lower left corner of the display object. This position is used to line up the display objects when establishing physical attachment between two display objects.

3.1.4 Graphical Relationships

Chips provides several methods to determine graphical relationships between display objects and their parts. These include methods to determine if a display object or one of its parts is above, below, to the left of, or to the right of another display object or one of its parts. There are also methods to determine if a display object or one of its parts intersects, is inside of, occludes, or obscures another display object or one of its parts.

3.2 The Substrate

The class **Substrate** defines instances that create and manage windows for displaying and manipulating display objects. A substrate senses mouse cursor activity within its substrate window and determines what messages to send to itself or to the instances it contains based on the location of the mouse cursor and the buttons that are pressed.

Figure 5 below illustrates what a substrate looks like. There are two windows, a substrate window and a prompt window. The substrate window has the title, "Substrate without a name." The substrate contains two display objects, one an abstract face, and the other, a text display object with the word "Foo" contained in a box. Each display object that appears in a substrate represents some domain object.

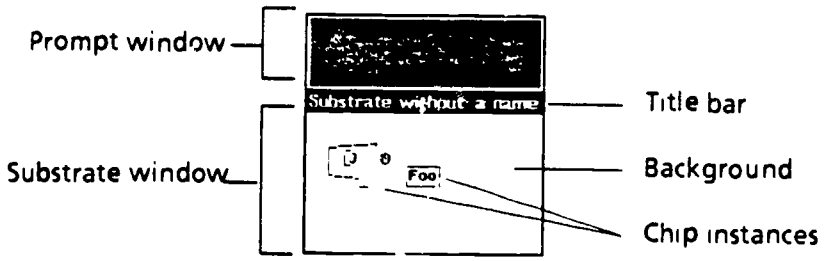


Figure 5. A default substrate instance containing two display objects

Substrates define several IVs including:

fileComs — the name of the file variable that describes the file that the substrate instance is stored on

fileName — the name of the file the substrate instance is stored on

window — the window that the substrate instance uses to display instances of DisplayObject

contents — a list of instances of DisplayObject that are displayed by the substrate

responsesToSelection — a form that describes the response to pressing a button while the mouse cursor is inside the window

A substrate's window stores its Substrate instance on its window property, LoopsInstance.

Substrates keep a list of the display objects they contain. This list is used to to redisplay the window, to find the display object under the mouse cursor, and to save the display objects and their associated domain objects to a file.

The substrate instance can also save a description of itself to a file that will create a window with the same attributes when a file containing the description is loaded into the environment. Figure 6 shows a substrate instance for which several of the parameters, such as the border size, background shade, and title, were changed from their default values. Modifications made to the substrate instance interactively can be saved and reproduced.

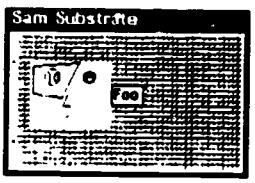


Figure 6. A substrate instance with parameters different from their default values

3.3 The Event Queue

In Chips, communication between individual domain objects is handled via an event queue. Each communication is considered an event and is posted on the queue along with a time when the event is to occur. The events are then processed in the order of the times declared. This allows events to be handled asynchronously by a separate process. The event queue was initially developed to avoid the problems of recursive function calling in complicated simulations [Duisberg, 1986].

Event queues are implemented by the class ChipsAnima. Each instance of the class ChipsAnima has two IVs:

eventQueue — a list of instances of the record type, queueEvent, with associated time stamps, stored as a skew heap

eventQueueProcess — a process that continually polls the eventQueue IV to see if there is an event whose time stamp indicates that it is time to be processed

When an event queue is established, a process is created that checks the eventQueue IV and sees if the event on the front of the queue, if there is one, has a time stamp that has expired. If there is such an event, the process sends the event queue the message ProcessEventQueue that removes the event from the event queue and sends the message ChangeOccurred to the instance stored in the participant field of the queueEvent record with the associated parameters. The default event queue, Anima, is created when Chips is loaded. When Anima is first used, a process is created, called Anima's Queue Handler.

The record queueEvent has several fields, including:

participant — an instance of the class DomainObject to whom communication is to be propagated

author — an instance of the class DomainObject that initiated the communication

name — an arbitrary tag that is the name of the communication; used to establish different communication types and to communicate information relevant to the communication

value — a value associated with a particular communication

3.4 Connections

Broadly speaking, a workstation screen normally displays a diagram consisting of windows and icons. Inside the windows are diagrams and text. Certain relationships are implied through what is displayed. A facility in Chips, called a connection, can be used to make an implied relationship on the display explicit for the computer. For example, if a window contains a road map, a line connecting two dots might indicate that there is a road between the two cities indicated by the dots. The fact that a road, displayed as a line, leads to a city, displayed as a dot, can be recorded in a connection between the road instance and the city instance. When the user makes a connection explicit for an application program, Chips causes the key relationships depicted graphically to be represented internally. Thus diagrams on the screen can have a syntax and semantics that both the user and the program share, and that both can manipulate.

Chips provides a class, Connection, whose instances represent relationships between instances of subclasses of DomainObject. Each instance of Connection has three IVs:

participant\nameList — a list of the form:

$$((\text{participant}_1 . \text{name}_1) \ldots (\text{participant}_n . \text{name}_n))$$

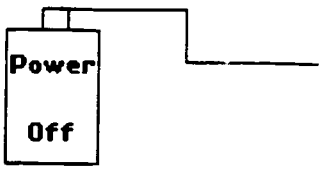
where name is some arbitrary tag used for establishing some connection type, or storing information useful for the participants in a connection, or both; participants are instances of some subclass of DomainObject.

responsibleObject — an instance that is responsible for propagating the communication from a domain object to the participants in a connection; the default responsible object is the Anima

timeDelay — an integer which establishes a time delay in the propagation of the connection; if non-NIL it is added to the current time before the event is placed on the event queue, thereby causing the event to wait in the event queue until its time arrives; the time delay is expressed in milliseconds

Figure 7 shows the list of participants and names for an instance of PowerSource, from our digital circuit editor. This power source is connected to an instance of the class Wire. In this example, the

name is used to determine which physical connectors of the two participants are connected, the output of the power source and one end of the wire.



((#$Wire0079 Output . endPoint2))

Figure 7. Connection between a power source and a wire

Connections store an object that is responsible for informing participants in the connection that some change has occurred that is relevant to the connection, the defaul s the event queue. A time delay, useful for simulations, may also be established for a connection and causes a delay before the propagation of the change to the participant. Connections can be used to represent many kinds of relationships between domain objects, such as physical attachment or containment.

The class **ConnectionMixin** provides the capability of connections to a class of domain object. Connections are established between a domain object and other domain objects. Each instance of **DomainObject** with connection capability stores a list of instances of the class **Connection** in an IV called **connections**. When a connection is established for a particular domain object, an instance of **Connection** is created and stored with that domain object.

When a domain object wants to propagate a connection, it sends the connection instance the message **AnnounceChange**, either directly or by sending itself the message **PutValueWithConnection** or **AnnounceChange**. The connection then sends the message **ChangeOccurred** to the instance stored in its **responsibleObject** IV for each participant in the **participant/nameList** IV of the connection. The message **ChangeOccurred** typically takes the parameters **author** (the domain object initiating the communication), **participant**, **name**, **value** (the value that has changed), and **time** (the time that the propagation is to happen, calculated by adding the value of the **timeDelay** IV of the connection to the current time).

Figure 8 shows a simple circuit containing a power source and a light bulb. Note, in our digital circuit editor, grounding is implicit.

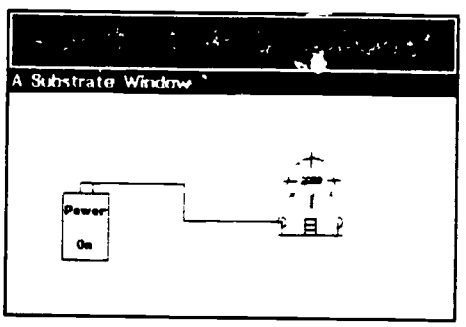


Figure 8. A simple circuit showing connections

In this example, a connection has been established between the output of the power source and one end of the wire. Another connection has been established between the other end of the wire and the input of the light bulb. Whenever a change is made to the output of the power source, in this case turning it on, the change is automatically propagated, through the wire, to the input of the light bulb, which responds to the change by lighting up.

The responsible object of a connection is, by default, the event queue. Another kind of responsible object provided by Chips is a *Spy*. Instances of the class Spy may be installed as the connection's responsible object and may be used to redirect connection changes or to do recording. By default, they just beep when a connection announces a change, and then pass the message to the event queue.

When an instance of the class Spy is installed in a particular connection, the old value of the responsibleObject IV of that connection is pushed on a stack on the IV property previousValues of the responsibleObject IV of the connection instance. Removing a spy pops the stack, re-installing the old responsible object. This provides an easy way to turn recording on and off during an application, for example.

3.5 Mechanisms

It is also useful to represent the relationship between an object and its parts. The mechanism of a domain object is a collection of instances of DomainObject, usually connected together, representing that domain object's internal mechanism. Through the connections, the collection of domain objects can act as "the clockwork inside the clock."

The class MechanismMixin provides the ability for a domain object to have a mechanism. It provides IVs to domain objects including:

mechanism — a list of instances of subclasses of the class DomainObject which define this domain object's behavior.

mechanismEditor — an instance of the class MechanismEditor, used to define and modify the mechanism of a domain object

If a domain object class has a mechanism defined for it, whenever an instance of that class is to be created, an isomorphic copy of the mechanism must be created, with all connections maintained.

Chips provides a *Mechanism Editor* to define and modify the mechanism associated with a particular subclass of DomainObject. The class MechanismEditor is a specialization of substrate with behavior that supports the definition of mechanisms. When the Mechanism Editor is opened, the mechanism of the selected domain object is displayed along with an internal connector for each physical connector defined for the domain object. Physical connectors provide access to the domain object's internal mechanism for other domain objects. These physical connectors are represented by instances of the class InternalConnector. These instances set up a connection between the domain object's external connectors and its internal mechanism.

When a domain object with a mechanism is sent the message ChangeOccurred, it forwards the message to the appropriate instance of InternalConnector, which in turn sends it to the domain object's that define the mechanism.

An example of the use of mechanisms is the class NANDGate, which was defined for our digital circuit editor. Its display object is shown in figure 9.

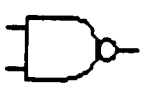


Figure 9. The display object of the class NANDGate

Display objects of the class NANDGate have three physical connectors, two on the left for input and one on the right for output. The class's behavior can be defined in terms of instances of two other classes, ANDGate and NOTGate. Figure 10 shows the mechanism of the class NANDGate.

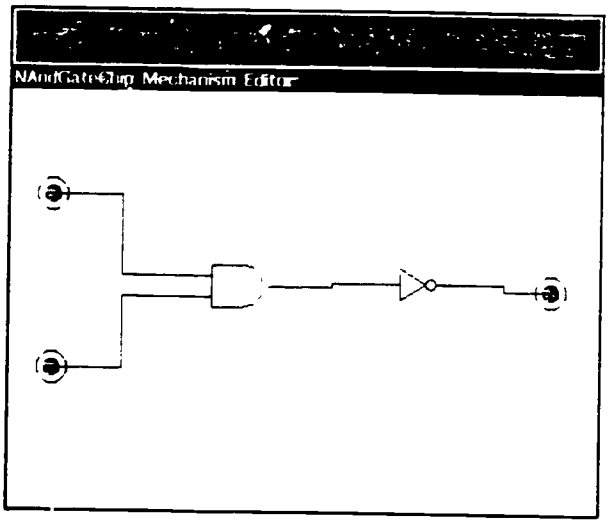


Figure 10. The mechanism of the class **NANDGate**

Each physical connector of the class **NANDGate** is represented by an instance of **InternalConnector**, shown in figure 10. The **Mechanism Editor** automatically positions the instances relative to where the physical connector appears on the domain object's display object.

The user creates the mechanism for the selected domain object class by selecting instances of the classes of domain objects that are to be included in its definition, dragging their display objects to an appropriate position, and connecting them with wires. The mechanism may then be saved to the domain object's class by selecting the **Save Class Mechanism** option from the substrate menu.

When an instance of **NANDGate** is used in a circuit, it processes signals sent to it by sending them to the instances defining its internal mechanism, via its internal connections. Figure 11 shows a **NANDGate** domain object in action.

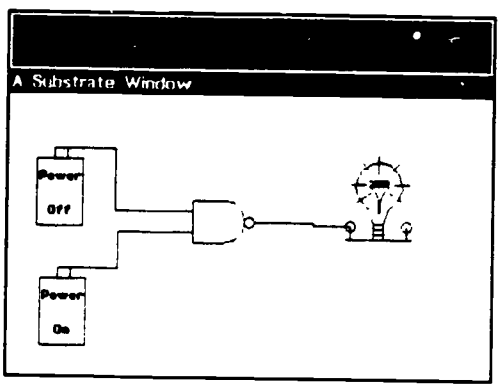


Figure 11. Example using the NAND Gate

3.6 Event Streams and Display Streams

Chips generalizes the input and output facilities of Interlisp-D to include object-oriented event streams and display streams, providing a straightforward way of performing I/O redirection.

Instances of the class **EventStream** may be passed to some methods expecting input from the mouse or keyboard, such as the method for dragging a display object around the screen, providing direct control

of the input from either the mouse or the keyboard. The default event stream is an instance of the class EventStream, called Mouse, which polls the mouse each time it is asked to update itself. This class can be specialized to get coordinates from a file, calculate coordinates based on some pre-determined path, poll the keyboard, etc.

Instances of the class DisplayStream, likewise, may be passed to certain methods that expect a display stream on which to perform output. One useful example of this is the class BufferedDisplayStream, which, instead of doing output directly to the screen, does its output to a scratch bitmap and displays on the screen when sent the message, Update.

Note: we have not developed display streams very much. They are included as a point of departure for further exploration.

3.7 Saving Chips Classes

When a file that contains Chips classes is saved, certain values of instance variables and class variables may need to be specially saved. Values such as bitmaps, instances, user-datatypes, arrays, hash tables, windows, and circular list structures will not be saved correctly without special handling. Chips defines several methods and functions that enable these kinds of values to be saved correctly.

For one of these values to be saved correctly, the instance or class variable that they are stored on must have a property that designates them as special. The property name may be either Instances, Ugly, or Horrible. If the property name is Instances, it designates some value of the instance or class variable that it is stored on as an instance or a list structure containing instances. If the property is Ugly or Horrible, it designates that some value of the instance or class variable that it is stored on is some other structure, such as a bitmap, user-datatype, array, or hash array, needs to be treated specially. If a value is marked as Horrible, it may contain a circular structure; if it is marked Ugly, it may not. Marking some value as Ugly results in a large speed and internal-storage advantage over marking it as Horrible.

Each of these properties, Instances, Ugly, or Horrible, may have values that designate which values of their instance or class variable are to be treated specially. If the value is Value, then the instance or class variable value is treated specially. If the value is All or Any, the instance or class variable value, as well as any properties of the instance or class variable, are treated specially. If the value is some other atom, it is treated as a property name, and that property of the instance or class variable is treated specially. The value may also be a list containing any of the above values.

When a file containing Loops classes is saved, each class is sent the message FileOut to save itself to file. Chips specializes this method, in the metaclass UglyMeta, so that it checks each instance and class variable to determine if any of its values are to be treated specially. When a Chips class (any class which has ChipMeta as its metaclass) is sent the message FileOut, the message is intercepted by UglyMeta (a super class of ChipMeta). This method calls the function AddInstancesToFilevar, which saves all values designated by the Instances property to the file variable of the file being saved. It then encodes all values marked by the Ugly or Horrible property by printing their values to a core file, using HPRINT, and reading them back in, using BIN, and constructing a string representation, which is then saved to the file.

When these files are loaded, the values marked as Ugly or Horrible must be converted back to their original representation. This is done by printing the values to a core file, using BOUT, and read from the core file using HREAD.

4. Chips Interactive Environment

Chips provides a powerful environment for interactively creating and modifying direct manipulation interfaces. There are two paths for developing applications that use Chips. They can be used interchangeably as convenience suggests. The first is through a Chips Browser This browser provides: access to the class definitions, editors for specific properties of classes, and access to the taxonomic hierarchy of the classes of an application. The second is through the application's own interface. There are a number of features that support direct access through the interface to underlying data structures, functionality, and specific properties of an interface. This section summarizes the features of the Chips interactive environment.

4.1 Chips Icon

Both paths of interaction are accessible through the Chips Icon. When Chips is loaded, the Chips Icon appears on the screen (see figure 1).

Left button: "Drag the icon"
Middle button: "Chips options"
Right button: "Window options"

Figure 1. The Chips icon and its mouse button options

Selecting the Chips Icon with the middle button presents a menu of Chips options, Create a substrate, Browse a file, Browse Saving Options, and Edit Chips Icon.

Selecting Create a substrate creates a new instance of the class **Substrate** and sends it the message Initialize which prompts for a region of the screen to display the new substrate.

Selecting Browse a file presents a menu of all the files on the system variable FILELST. Selecting a file name from this menu creates an instance of the class **ChipsBrowser** that shows all of the classes defined by that file. This browser may then be accessed interactively. This option has a submenu associated with it with one selection, Browse object dependencies. Selecting this option presents a menu of all files on the system variable FILELST. If a file name is selected, a browser of that file is created, displaying the file name and all objects that are stored on that file's variable (see figure 2)

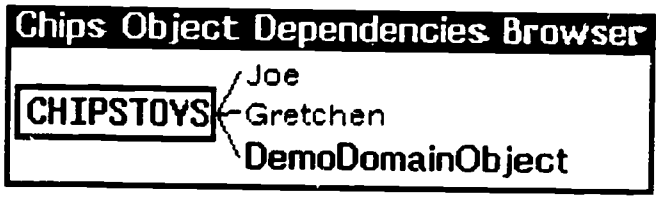


Figure 2. A browser showing the objects pointed to by the file CHIPSTOYS

In this browser, nodes representing file name are display in bold font with a two pixel border around the name, class names are displayed in bold font without a border, and instances are displayed in a regular font. Each node has several options available by selecting the node with the middle mouse button pressed. These options are shown in figure 3.

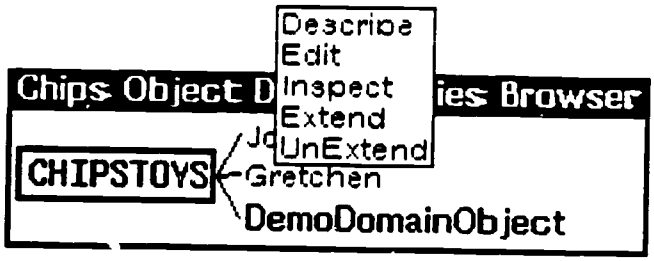


Figure 3. Options available from the object dependencies browser

Selecting Describe from this menu prints information about the selected node including what kind of object it is and what files it is stored on.

Selecting Edit from this menu invokes the Interlisp-D editor DEdit on the definition of the object associated with the selected node.

Selecting Inspect from this menu creates an Interlisp-D inspector, inspecting the object associated with the selected node.

Selecting Extend from this menu extends the browser to include objects pointed to by the selected node.

Selecting UnExtend from this menu removes all objects pointed to by the selected node from the browser.

Selecting Browse Saving Options from the Chips Icon middle button menu presents a browser of saving options that controls what actions are to occur when certain events occur during the use of Chips. This browser is shown in Figure 4.

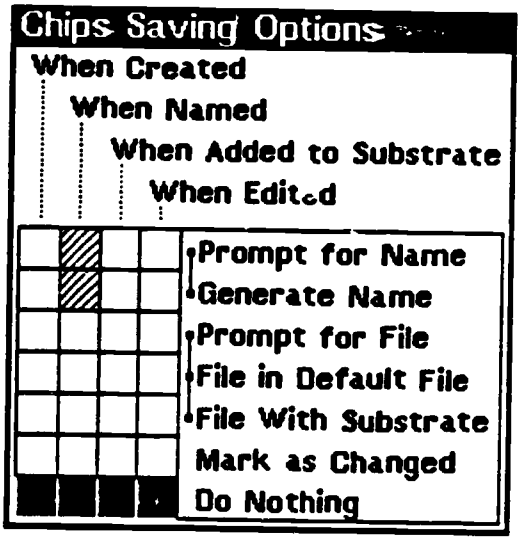


Figure 4. The Chips Saving Options Browser

The grid in the browser allows the user to control what actions are to occur at specified events during the use of Chips. The events are listed, horizontally, at the top of the browser while the actions to take in response to these events are displayed vertically, to the right of the grid. Responses that are mutually exclusive are grouped with a vertical bar connecting the mutually exclusive responses.

The four events that are controlled with this browser are: When Created, When Named, When Added to Substrate, and When Edited

When Created — whenever an instance of the class DomainObject, DisplayObject, Substrate or any of their subclasses is created and initialized, the selected responses occur

When Named — whenever a instance is named while using Chips, the selected responses occur

When Added to Substrate — whenever a display object instance is added to a substrate, the selected responses occur.

When Edited — whenever an instance is edited through a Chips menu, the selected responses occur

The responses that are controlled from this browser are Prompt for Name, Generate Name, Prompt for File, File in Default File, File With Substrate, Mark as Changed, and Do Nothing.

Prompt for Name — asks the user to enter a name for an object.

Generate Name — generates a name for an object using GENSYM and the class name of the object.

Prompt for File — asks the user to select a file in which to store the object.

File in Default File — stores the object in the default file; if the object is a class, it is stored in the file CHIPSCLASSES, if it is an instance, it is stored in the file CHIPSINSTANCES.

File With Substrate — stores the object, usually a display object instance, in the same file as the substrate it is displayed in.

Mark as Changed — marks an object as changed so that it will be recognized by the file package

Do Nothing — does nothing in response to the selected event.

Selecting the option Edit Chips Icon invokes the Interlisp-D editor, DEdit on the class ChipsIcon

4.2 The Chips Browser

Chips provides a graphical browser for a class hierarchy of Chips classes that supports the creation and management of Chips files. It is called the Chips Browser (see figure 5).

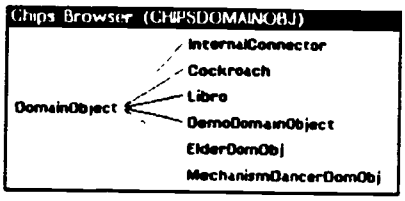


Figure 5. A Chips Browser

This browser provides a graphical display of the portion of the class inheritance lattice that is defined by a particular file. Selecting the name of a class with the mouse produces a menu for editing different aspects of the selected class.

This browser is a specialization of the Loops class FileBrowser. The Loops browser provides options that allow the interactive creation, modification, and examination of classes (see figure 6).

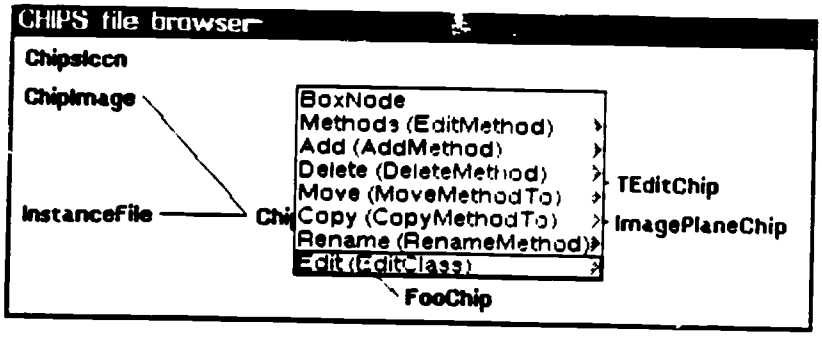


Figure 6. Browser options provided by the Loops file browser

In addition to these, we have added options specific to chips classes. These options are shown in figures 7 and 8.

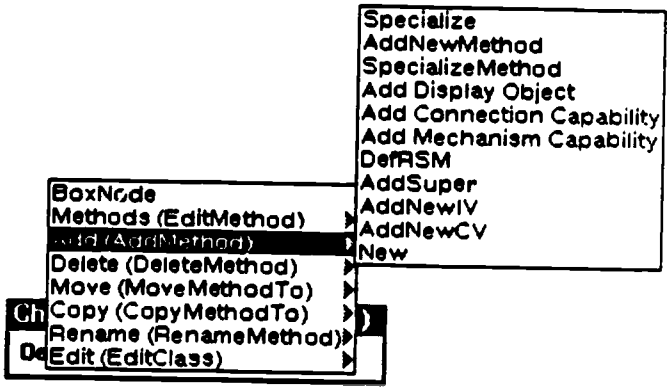


Figure 7. Options available from the Add (AddMethod) submenu

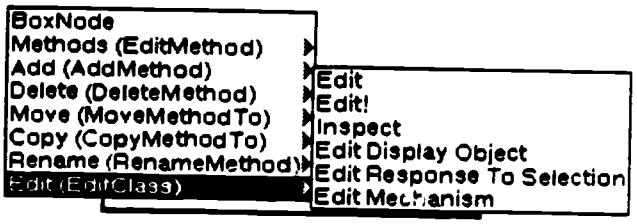


Figure 8. Options available from the Edit (EditClass) submenu

There are several options that are specific to Chips, all to be used with subclasses of the class **DomainObject**, including: Add Display Object, Add Connection Capability, Add Mechanism Capability, Edit Display Object, Edit Response To Selection, and Edit Mechanism.

Selecting Add Display Object creates an inspector that allows the user to define the new display object that will be added to the selected domain object class. This browser is shown in figure 9.

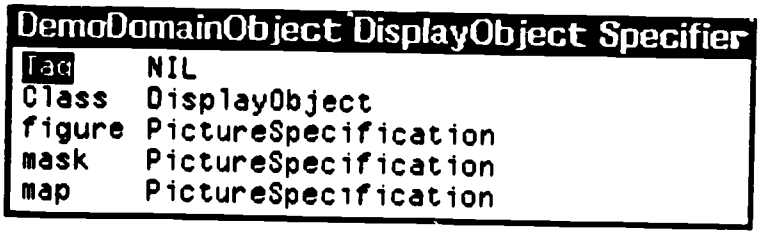


Figure 9. The Display Object Specifier

Using this inspector, the user may specify the class of display object that is to be added to the selected domain object class along with the tag that will be used to refer to that display object and the classes of picture specification that are to be used for the display object's figure, mask, and map When the display object is specified, it may be installed in the domain object class by selecting the inspector's title bar with the middle button pressed and selecting Install from the menu that appears.

Selecting the option Add Connection Ability establishes connection capability for the selected domain object class. When this option is selected, Chips attempts to add the class ConnectionMixin to the supers list of the class. This is done by sending the class the message InstallSuper, which is defined by the metaclass **AddSuperMeta**. This method expects that either the class or one of its super classes has a CV that has the same name as the super class to be added. This CV should have two properties: fileName, which stores the file that the super class is stored on, and selectors, which stores a list of the messages that the super class implements. InstallSuper asks to make sure that the user wants to add the super to the selected class. If so, it checks to see if the file that implements the super is loaded, by sending the class the message FileLoaded?. If the file isn't loaded, it will load it. It then installs the super in the super list of the class, copying any IVs with a property copyDown that has a non-NIL value.

In addition to explicitly requesting that a capability be added, if any message is sent to a domain object instance that it does not understand, Chips checks to see if the message is one that would be understood if a certain super were added to the supers list of the domain object This is accomplished with the **AddSuperMeta** class and the method DomainObject.MessageNotUnderstood. If a message is sent to a domain object that it does not understand, the message MessageNotUnderstood is sent by Loops to the object, which, in turn is intercepted by DomainObject.MessageNotUnderstood. MessageNotUnderstood sends the message NewSuperSelector? to the class of the domain object. This message is implemented by the class **AddSuperMeta** and looks at the domain object's class for a CV that has a selector on its selectors property that matches the message that was sent to the domain object. If such a CV exists, the message InstallSuper is sent to the Domain object's class.

Selecting the option Add Mechanism Ability establishes mechanism capability for a class of domain objects. When this option is selected, Chips attempts to add the class **MechanismMixin** to the supers list of the domain object's class. If the class **MechanismMixin** is not loaded, Chips will ask the user whether to load the file MECHANISMS which defines the classes, instances, methods, etc., that are required to establish a mechanism. **MechanismMixin** is then added to the supers list. This is done following the same procedure described above for **ConnectionMixin**.

Selecting the option Edit Display Object allows the user to define how instances of a domain object class will display themselves in a substrate. When this option is selected, it presents a menu of all dispiay objects defined for the selected domain object class. If a display object is selected the display object is then edited, using the Display Editor. If the selected display object has an instance of the class **DisplayEditor** stored in its editor IV, that display editor is opened. If not, a new instance of the class **DisplayEditor** is created, stored in the editor IV of the display object, and opened. In this case, selecting Exit while using the Display Editor updates the display object associated with the domain object class, so instances created from this class will subsequently reflect the changes made during editing. The Display Editor is discussed in detail at the end of this section.

Selecting the option Edit Response To Selection allows the user to define the response to selecting a particular display object with the mouse cursor while that display object is displayed in a substrate. When this option is selected, a menu of all display objects defined for the selected domain object is

presented. If one is selected, the Interlisp-D editor DEdit is invoked on the form that describes that display object's response to selection.

Selecting the option Edit Mechanism allows the user to edit the mechanism associated with the selected class of domain object. Mechanisms provide a way to describe the behavior of a class of domain objects in terms of instances of other classes of domain objects, as described above.

When this option is selected, the user is asked to sweep out a region of the screen to display a substrate, called the Mechanism Editor. The Mechanism Editor will contain the class's mechanism, if one is defined.

An option has been added to the title bar menu of the Loops FileBrowser: Add New Class. Selecting Add New Class and sliding to the right (see Figure 10) presents a menu of Chips classes that will frequently need to be specialized, providing a straightforward way of creating new specializations and associating them with a particular file. When a class is selected, the user is asked to type in a name for the new specialization, which is then created, having the selected Chips class in its supers list. When a Chips class is specialized, all IVs of the specialized class that have a property copyDown set to a non-NIL value are copied along with their values to the new class. This is accomplished using the metaclass CopyOnSpecialize with the method Specialize. This method is a specialization of Class.Specialize.

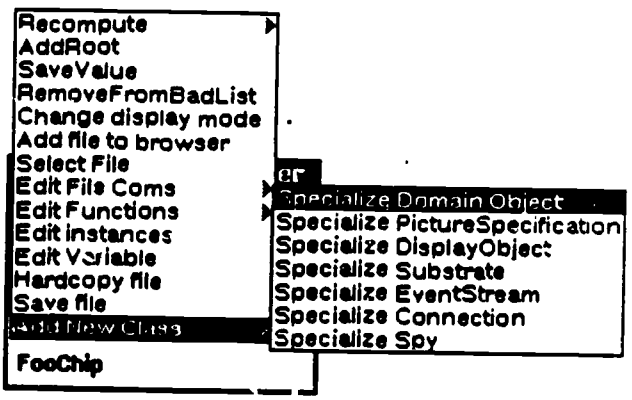


Figure 10. Creating a specialization of a Chips class from the file browser.

4.3 Modifying an application through the development interface

This section will discuss a) how a substrate manages the display of multiple, overlapping display objects and b) the editing options available by selecting substrates and display objects.

4.3.1 Displaying overlapping display objects

To support the display of multiple, arbitrarily shaped display objects in a substrate, Chips creates the illusion that display objects overlap one another, as though the screen had depth and some display objects were closer to the viewer than others. This overlapping is essentially 2 1/2 dimensional. That is, there is no sense of absolute distance between the display object and the viewer, only that certain display objects are closer to the viewer than those that they overlap. Chips provides a sense of relative depth, not absolute depth.

Each substrate instance stores a list of the display object instances it contains in the IV contents. They are stored in order, so that the topmost display object is on the front of the list. Each display

object stores an ordered list of the display objects that it overlaps in an IV, occludedByMe, and an ordered list of the display objects that overlap it, in an IV, called occludesMe. When a substrate instance redisplays its window, it clears the window, and traverses its contents in reverse order, sending each display object the message Draw. As mentioned in Chapter 3, display objects can be irregularly shaped and may have holes in them.

When a display object is to move, it is sent the message PrepareToMove which, in turn, sends the message DrawUnder, drawing all display objects that overlap the display object to a scratch bitmap. It then removes itself from all occludesMe and occludedByMe IVs of the overlapping display objects, and finally removes everything from its own occludesMe and occludedPyMe IVs.

When a display object is placed in a substrate, it checks to see which display objects it overlaps and updates itself and them accordingly, with the message InformThoseILandedOn. It also puts itself on the front of the substrate's contents IV, sending the substrate instance the message AddInFront.

Occlusion is maintained with respect to selection of a display object with the mouse cursor. When a mouse button is pressed while the mouse cursor is in a substrate's window, the window's BUTTONEVENTFN is called. The default BUTTONEVENTFN in Chips is ChipsEventFn. This function sends the window's substrate instance the message GetObjectAt, which traverses the contents IV of the substrate, in order, sending each display object the message OnYou? with the coordinates of the mouse cursor selection. If a display object was under the cursor, it is returned, otherwise the substrate instance itself is returned. The instance that is returned is sent the message RespondToSelection. The RespondToSelection method sends the selected instance the message GetPartAt with the coordinates of selection. The method GetPartAt traverses the object's map and returns a tag, indicating what the cursor was over when the mouse button was pressed. The RespondToSelection message then looks at the eventResponses IV of the object to determine what to do in response to the selection. The eventResponses IV stores a list of triples of the form:

(part howSelected whatToDo)

part is the name of a part of the instance, howSelected indicates the type of selection and is usually a type of button, such as LEFT or MIDDLE, whatToDo is either an atom in which case it is treated as a message name and is sent to the selected instance, or it is a form that is evaluated.

In addition to being arbitrarily shaped, display objects do not have to be entirely solid. It is possible to define holes in the middle of a display object. This is also supported both visually and with respect to selection with the mouse cursor.

Figure 11 shows a substrate with three display objects: the display object of the class ChocolateChip which looks like a chocolate chip cookie, the display object of the class FooChip which looks sort of like the man in the moon, and the display object of WasherChip which has a hole in the middle.

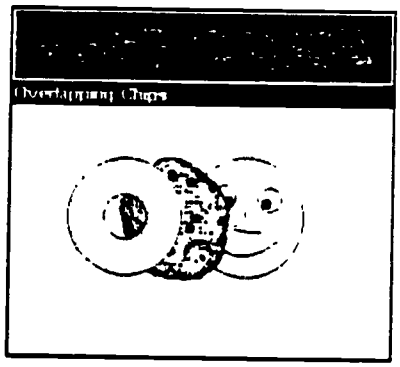


Figure 11. A substrate with three overlapping display objects

In this figure, the WasherChip overlaps the ChocolateChip which in turn overlaps the FooChip. The ChocolateChip is partially occluded by the WasherChip but can be seen through the hole in the WasherChip. Selection of these display objects with the mouse cursor exactly corresponds to their visual representation in the substrate. Selecting the part of the FooChip that is not occluded selects this display object. Selecting any part of the WasherChip's display object selects it. Selecting any part of the ChocolateChip that can be seen, including the part that is seen through the hole in the WasherChip, selects it.

Substrates keep a list of the display object instances they contain. This list is ordered by depth; the front-most display object instance is first. To redisplay the substrate window, the list is traversed in reverse order so that the front most display object is displayed last. Thus, overlapping display object instances give the illusion of depth as display object instances closer to the front occlude display objects behind them. To determine which display object instance the mouse cursor is pointing to, the list is searched in order. Thus, if display object instances overlap one another, the one closest to the front is found first.

4.3.2 Interactive editing of display object instances

By default, Chips provides a number of options available through a display object on the screen. To perform some operation on a display object or its associated domain object, the user merely selects that display object with the mouse cursor. This section will discuss the default options that are available for interacting with display objects and domain objects through their pictures on the screen.

The default response to left button mouse selection of a display object is to send that display object the message Animate, which picks it up, attaches it to the mouse cursor and allows it to be dragged around the screen. When a display object is picked up, it first comes to the top of whatever display objects may have been overlapping it. It then follows the mouse cursor around the screen until another mouse button is pressed. When a display object is put down, it will, by default, overlap any display objects that are occupying the region it is placed in. Display objects may be dragged from one place in a substrate to another or placed in any open substrate on the screen

Dragging maintains the illusion that the user is actually manipulating the objects represented by a particular display object. The dragging animation is very smooth with no flicker and does not obliterate the screen.

The method that implements dragging is called Animate. Animate provides hooks for redefining what happens when dragging a display object. To use these hooks, the user needs to specialize one or more methods for a new class of display object.

Chips provides several options for editing the properties and behavior of a display object and its associated domain object. These are available by selecting a display object with the middle button and

choosing the editing option from a menu. When the middle button is pressed, the display object is sent the message OfferEditOptions, which presents the menu of options. These options are acquired by appending the results of sending the display object and its associated domain object the message GetEditOptions. These options are roughly grouped into four categories: operations involving the display object's properties and behavior, operations involving the associated domain object's properties and behavior, operations involving the connections of a domain object, and operations involving the domain object's mechanism. Figure 12 shows the menu of editing options available for the display object of an instance of the class **MechanismDancerDomObj**.

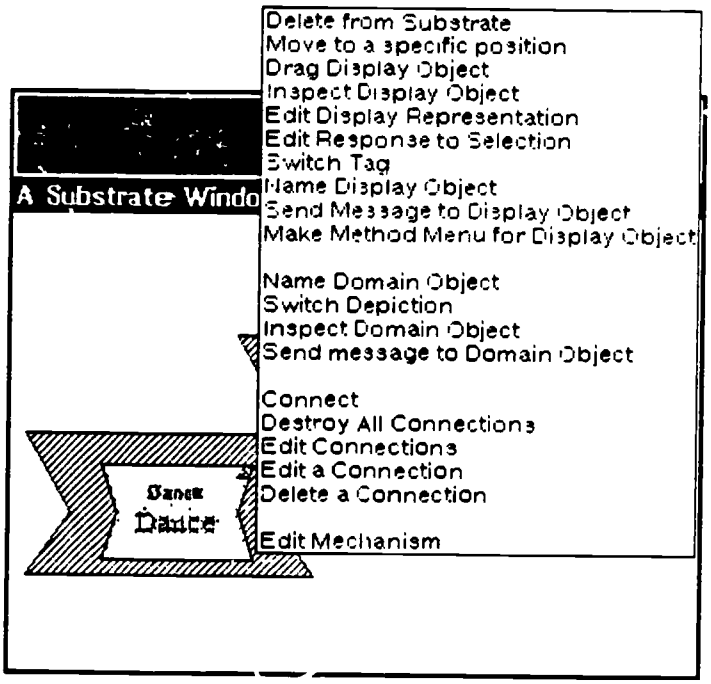


Figure 12. The editing options menu for an instance of **MechanismDancerDomObj**

Editing options involving display objects

There are ten options that support editing the properties and behavior of display objects: Delete from Substrate, Move to a specific position, Drag Display Object, Inspect Display Object, Edit Display Representation, Edit Response to Selection, Switch Tag, Name Display Object, Send Message to Display Object, and Make Method Menu for Display Object.

Selecting the option Delete from Substrate deletes a display object from the substrate in which it is displayed, by sending the display object the messa DeleteFromSubstrate Deleting a display object erases it from the screen, displaying any display objects that it overlapped, maintaining, in turn, their overlapping with other display objects in the substrate. It also removes it from the displayObjects IV of its associated domain object instance.

Selecting the option Move to a specific position allows the user to specify coordinates within the same window where the display object is to be moved. The user is prompted to enter the x and y coordinates for the move, using the Interlisp-D function RNUMBER, and the display object then removes itself from its current position and relocates in the position indicated by the entered coordinates, sending itself the message Move.

Selecting the option Drag Display Object sends the message Animate to the display object, allowing it to be picked up and dragged around the screen. Dragging is described in detail above. Selecting this option is the same as selecting the display object with the left button.

Selecting the option Inspect Display Object invokes the Interlisp-D inspector on the selected display object instance. The inspector is window-based and allows the user to examine and modify the properties of a particular instance of a display object class. Figure 13 shows an inspector for the display object of an instance of the class SquareChip.

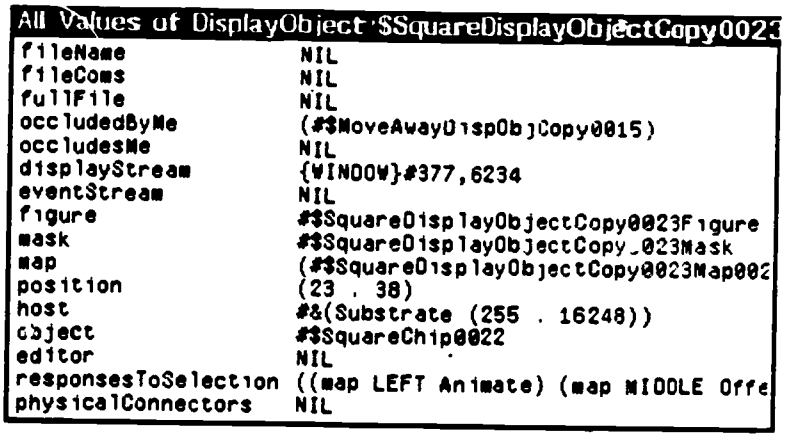


Figure 13. The inspector

Selecting the option Edit Display Representation invokes the Display Editor on the selected display object. The Display Editor will be discussed in detail below.

Selecting the option Edit Response to Selection invokes the Interlisp-D editor DEdit on a form that defines the display object's response to selection with the mouse cursor. The form is a list of triples, each consisting of the name of a map element of the display object, a type of mouse selection (usually either LEFT or MIDDLE), and the action to take in response to selecting the particular map element with the particular type of mouse selection. If the action is an atom, it is treated as a message name that is sent in reponse to the particular combination; otherwise it is treated as a form to be evaluated. The user may alter elements, add new elements, or delete existing elements from the list, altering the display object's response to selection with the mouse cursor. Figure 14 below shows the response description form for the display object of a MoveAwayChip.

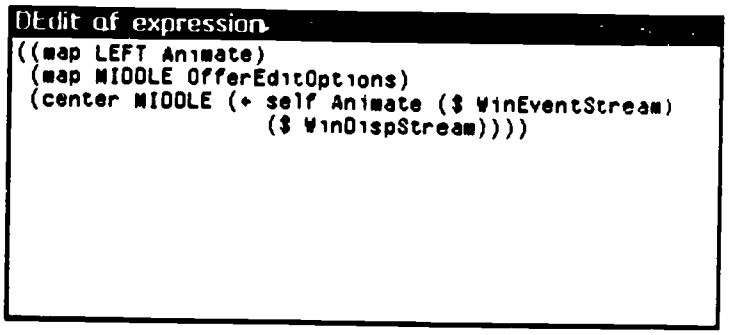


Figure 14. The response description for a display object.

Selecting the option Switch Tag allows the user to switch the set of picture specification instances that are used to display the selected display object. When this option is selected, a menu of all tags associated with the selected display object is presented. If a tag is selected from this menu, the new

picture specification instances are swapped in, becoming the new values of the figure, mask, and map IVs of that display object.

Selecting the option Name Display Object allows the user to give some easily remembered name to a particular display object instance.

Selecting the option Send Message to Display Object allows the user to send the selected display object a message. When this option is selected, the user is prompted to enter the name of a message in the substrate's prompt window. This message is then sent to the display object, executing the associated method.

Selecting the option Make Method Menu creates a menu of the methods associated with the selected display object's class. This menu may then be used to edit particular methods with the Interlisp-D editor.

Editing options involving domain objects

There are four options that support editing the properties and behavior of a display object's associated domain object: Name Domain Object, Switch Depiction, Inspect Domain Object, and Send Message to Domain Object.

Selecting the option Name Domain Object allows the user to give some easily remembered name to a particular domain object instance.

Selecting the option Switch Depiction allows the user to switch display objects for a particular domain object. When this option is selected, a menu of the display objects associated with the selected domain object's class is presented, by sending the domain object the message AskDepiction. Selecting one of these sends the display object the message ReplaceDepiction, deleting the current display object and substituting the selected display object in the substrate at the same position. Figure 15, shows a sequence of three substrates that demonstrates changing the display object of an instance of **DemoDomainObject**.

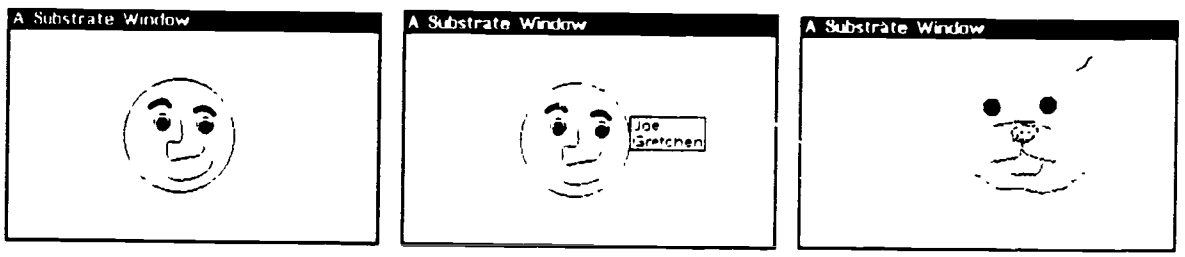


Figure 15. Changing the display object of **DemoDomainObject**

Selecting the option Inspect Domain Object invokes the Interlisp-D inspector on the domain object instance associated with the selected display object. The inspector is window-based and allows the user to examine and modify the properties of a particular instance of a domain object class.

Selecting the option Send Message to Domain Object allows the user to send a message to the domain object instance associated with the selected display object. When this option is selected, the user is prompted to enter the name of a message in the substrate's prompt window. This message is then sent to the domain object, executing the associated method.

Editing options involving Connections

If the domain object associated with the selected display object has connection capability, five options are available from the editing options menu that support creating and maintaining connections

between domain objects: Connect, Destroy All Connections, Edit Connections, Edit a Connection, and Delete a Connection.

Selecting the option Connect allows the user to interactively add a new connection for the domain object associated with the selected display object. When this is selected, the user is prompted to enter the name of the new connection and to select the participant in the connection. The connection is then established.

Selecting the option Destroy All Connections deletes all connections currently established for the domain object associated with the selected display object.

Selecting the option Edit Connections allows the user to edit the connections of the domain object associated with the selected display object. When this is selected, the Interlisp-D inspector is invoked on the instances of Connection currently defined for the domain object.

Selecting the option Edit a Connection allows the user to specify a particular instance of Connection to be edited. When this option is selected, a menu is presented of all participants involved in connections with the selected domain object. If one is selected, another menu of the names of all connections that the selected domain object and the selected participant are involved in. If both participant and name are specified, the Interlisp-D inspector is invoked on the instance of Connection indicated.

Selecting the option Delete a Connection allows the user to interactively specify a particular connection to be deleted. Specifying the connection is done as described above for Edit a Connection. Once a connection has been specified, this connection is deleted from the domain object associated with the selected display object.

Editing options involving Mechanisms

If the domain object associated with the selected display object has mechanism capability, an option is available from the editing options menu that supports creating and maintaining the domain object's mechanism: Edit Mechanism. Selecting the option Edit Mechanism enables the user to edit the mechanism that determines the selected domain object's behavior.

4.3.3 Options available by selecting a substrate

Chips provides a number of options that are available by selecting a substrate window. These options allow the user to interactively examine and modify important properties of substrates.

New instances of domain object classes can be created and their display objects displayed in a substrate by pressing a mouse button while the mouse cursor is in the background of a substrate window. When the background is selected the message OfferNewDomainObject is sent to the substrate instance. This method presents a menu, by sending the substrate instance the message AskDomainObjectClass, which contains the names of all the classes of domain object currently defined in the environment. If one is selected, an instance of that class is created and sent the message Initialize. If there is more than one display object for the selected domain object, a menu of the display objects is presented. If there is only one display object for the selected domain object, that one is used. The display object is then displayed in the substrate's window. When new classes of domain object are defined, they are automatically added to the substrate's background menu. Figure 16 shows the response to selecting in the background of a substrate.

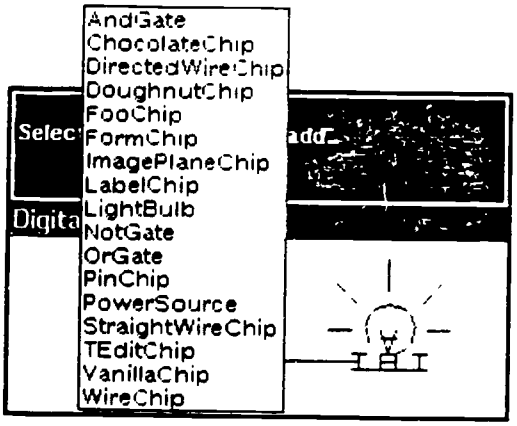


Figure 16. The background menu of a substrate

Pressing the left mouse button while the mouse cursor is in the title bar of a window, sends the associated substrate instance the message OfferEditOptions, presenting a menu of editing options that allow the user to examine and modify important properties of the selected substrate. These options are shown in figure 17.

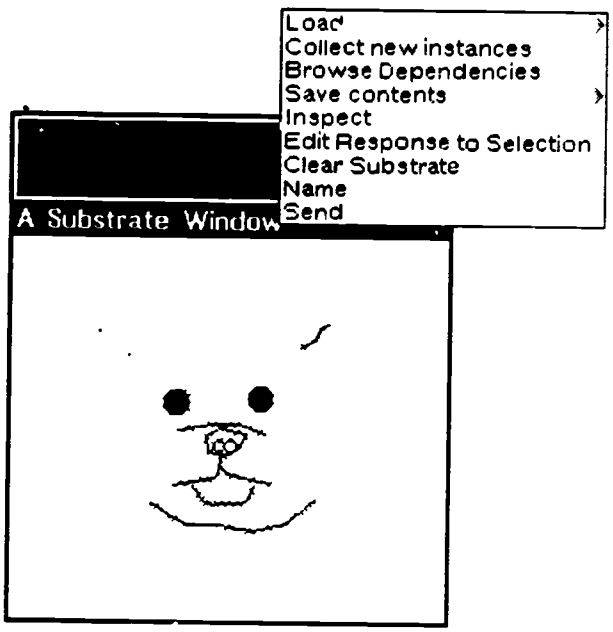


Figure 17. Title bar options of a substrate

Selecting the option Load allows the user to load a group of display objects from a file into the selected substrate. When this option is selected, the substrate is sent the message Load, which prompts the user to enter a file name. If a file name is entered, the file is loaded into the environment and display object instances stored on the file are displayed in the substrate's window.

When an instance is loaded that has a value that is marked as Ugly or Horrible, such as a bitmap, it is necessary to convert this from the form that was used to save it. When an instance is loaded from a

file, the function DEFINST is called to create the instance. DEFINST sends the message OldInstance to the instance after it is defined. Classes whose instances may have such values stored in some IV or IV property have a super called UglyMixin. UglyMixin specializes the method OldInstance to check the new instance for values marked as Horrible or Ugly.

To designate that a particular value is Horrible or Ugly, the IV containing the value should have a property Horrible or Ugly, which may have as its value one of the following: Value, which designates the IV value as the ugly or horrible structure, All or Any, which designates that the IV value and all of the IV's properties have a structure that is horrible or ugly, a property name, which designates a specific property as horrible or ugly, or a list containing any of the above values. If a value is designated as Ugly, it is assumed to not have circular structures; a value that is marked as Horrible may have circular structures. Marking something Ugly resits in a large speed and internal-storage advantage over marking it as Horrible. When a horrible or ugly value is encountered, the method UglyMixin.OldInstance decodes the value by using BOUT to write the value to a core file and then reading it from the core file using HREAD.

After each instance is read, its host IV is set to the substrate it is loaded into, its displayStream IV is set to the substrate's window, and each instance is added to the contents IV of the substrate. The window is then redisplayed.

Selecting the option Collect new instances allows the user to associate the substrate and all display objects it contains with a particular file, placing all instances on the files file variable. When this option is selected, the user is asked to specify a file to save the substrate and its display objects on. These instances are then added to the file variable of the specified file.

Selecting the option Browse Dependencies creates a browser window with one node representing the substrate. This node can then be expanded further to examine the objects pointed to by the substrate. This can be useful to discover exactly what will be saved to a file when the substrate is saved.

Selecting the option Save contents allows the user to save the display objects contained in the selected substrate and their associated domain objects to a file. When this option is chosen, the user is prompted to enter a file name to save to. If one is specified, the contents are saved to a file. They may be loaded into another substrate later using the Load option. Saving display object instances means that picture specification instances must be saved as well. Since picture specification instances typically have bitmaps as values of their instance variables, these values will need to be encoded before saving them to a file. This is accomplished by the method UglyMixin.FileOut. FileOut is a specialization of Object.FileOut which encodes values that are marked as Ugly or Horrible. It does this by writing the values to a core file with HPRINT and reading them in using BIN and converting them to a string before it prints them to a file. Classes whose instances may store these values, such as PictureSpecification, have UglyMixin as a super class.

The Save contents and substrate option is available by selecting the Save contents option, sliding to the right, and selecting it from the submenu that appears. The user will be prompted to enter a file name. If one is specified, the substrate and all of its contents will be saved to the file. When this option is selected, a description of the substrate's window is also saved to the file so that the window can be recreated with all its properties intact.

Selecting the option Inspect invokes the Interlisp-D inspector on the selected substrate's instance.

Selecting the option Edit Response to Selection invokes the Interlisp-D editor DEdit on a form that defines the substrate's response to selection with the mouse cursor. The form is identical to the form

described above for display objects. Figure 18 below shows a sample response description form for a substrate.

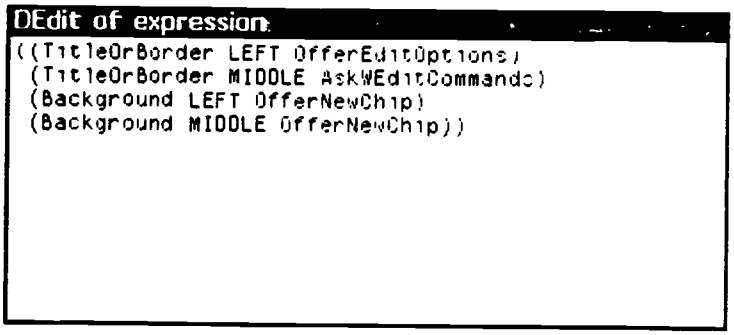


Figure 18. The response description for a substrate.

Selecting the option Clear Substrate deletes all the display objects from a substrate and updates the display.

Selecting the option Name allows the user to give some easily remembered name to a particular substrate instance.

Selecting the option Send allows the user to send a message to the selected substrate. When this option is selected the user is prompted to enter a message name. If one is specified, that message is sent to the substrate and the corresponding method executed.

There are two additional options available by selecting the window's title bar with the middle mouse button pressed: Edit Window and Edit Button Event Function.

Selecting Edit Window invokes the Window Description Editor, a modified version of the Interlisp-D Inspector, allowing the properties of the substrate's window to be interactively modified. This inspector allows window properties to be interactively changed and the results seen immediately The Window Description Editor is shown in Figure 19.

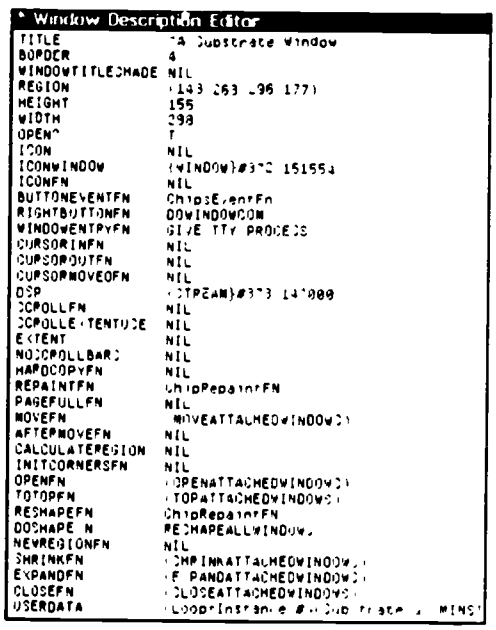


Figure 19. The Window Description Editor

Selecting Edit Button Event Function invokes the Interlisp-D editor DEdit on the function that determines the window's response to selection with the mouse cursor. The default button event function, ChipsEventFn, merely sends the message RespondToSelection to the object that was selected with the mouse cursor. This enables the user to control responses to selection through the menu option, Edit Response to Selection, provided for display objects and substrates. Editing the button event function directly may disable this ability but is provided to allow for more flexible determination of a window's response to selection.

4.3.4 The Display Editor

The Display Editor allows the user to interactively design a display object. This is done by using a modified version of the Interlisp-D graphical editor, Sketch, to draw what the display object should look like when displayed on the screen. Using the Display Editor, the user can define both what the display object will look like and its mouse-sensitive areas. It also provides a way to define alternate sets of picture specification instances for the display object and to establish a mouse-sensitive subregion as a physical connector. The Display Editor is shown in figure 20.

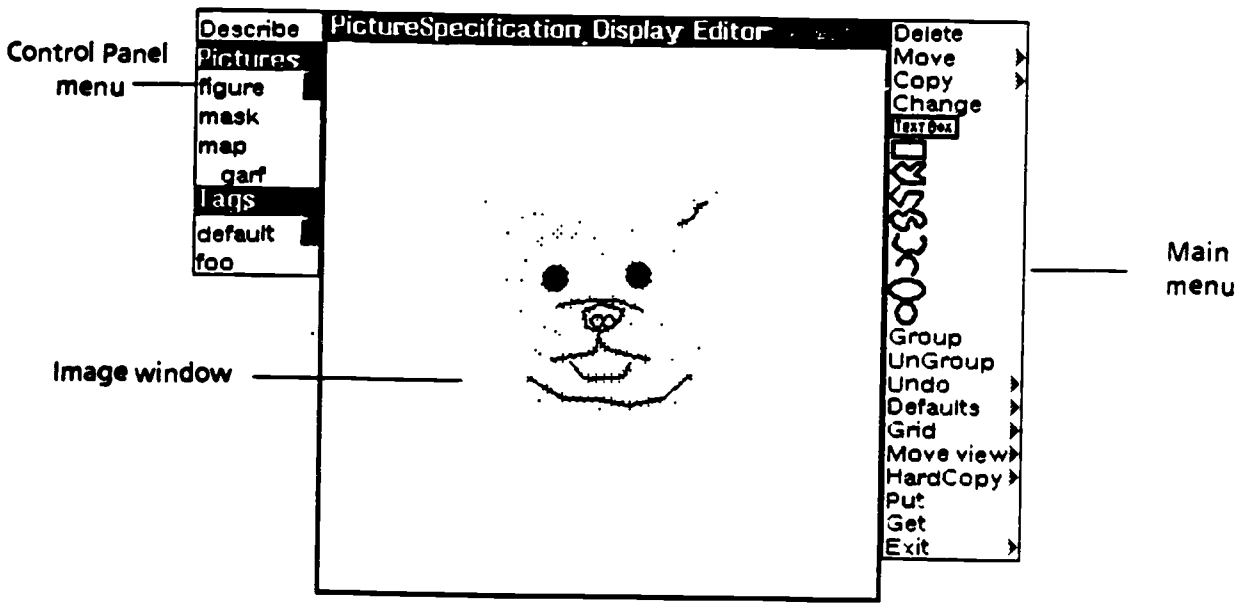


Figure 20. The Display Editor

Instances of the class DisplayEditor are cached on the editor IV of the display object that is edited.

Each display object has at least three pictures associated with it: the figure, mask, and map. The figure picture describes what the display object will look like on the screen. The mask picture describes which parts of the display object are opaque. The map picture describes what part of a display object may be selected with the mouse cursor.

The main menu of the Display Editor provides graphical primitives such as circles, polygons, curves, and closed curves plus simple operations for manipulating these graphical objects. The main menu can be seen to the right of the main window in figure 20.

The Display Editor adds three options to the main Sketch menu: Move to picture and Copy to picture, both available in the submenu of their corresponding main menu selections, and Exit. When one of Move to picture or Copy to picture is selected, a menu of all the pictures currently defined for the

display object is presented. If one is selected, the user is then asked to select the elements to copy or move. If one or more are selected, they are then moved or copied to the selected picture.

The exiting options are Exit and Quit. Exit saves the sketches to the editRepresentation IV of the picture specifications of the display object being edited, creates a bitmap from the sketches, and updates the offset of the picture specification to represent the offset of the region occupied by the picture's sketch from the largest region occupied by the sketches of all pictures. Quit stops the editing, leaving the picture specifications as they were before editing

The Display Editor also provides a control panel for moving between the various pictures of a display object, for creating new mouse-sensitive subregions, for switching between various sets of pictures defined for the display object, and for adding new sets of pictures to the display object. This is shown in figure 20 to the left of the main window.

The control panel menu is split into three parts. The top part is the option Describe which prints information in the User Exec window describing the display object and where it came from.

The next part of the control panel menu is the Pictures Menu. This allows the user to switch between pictures by selecting a name with the left button. Selecting a picture from the control panel with the middle mouse button pressed presents several other options. Each picture has two options: Display picture and Edit picture. Selecting Display picture displays the selected picture in the background of the picture being edited, in gray. This is often useful for lining up parts of two separate planes. Selecting Edit picture makes the selected picture the picture being edited.

The user can define new pictures, representing mouse-sensitive subregions, for a display object. The map and subregions are stored in the tree form used by the map of a display object. Their position in the tree is represented in the control panel by indentation, those things indented further to the right indicate that they are at a lower level of the tree. To add a new mouse-sensitive subregion, the user selects the map or an existing subregion from the control panel with the middle mouse button pressed. This presents a menu with several options, including Subdivide picture. If this option is selected, the user is prompted to enter a name for the new subregion, and a new subregion picture is created, nested within the selected region.

A subregion picture can be deleted by selecting it with the middle mouse button pressed and selecting Delete picture from the menu that appears.

A subregion plane can be established as a physical connector by selecting Label Position from the middle button menu. When this option is selected, the user is prompted to select a position in the Sketch that will serve as physical connector position for this picture. This will add the picture's name to the physicalConnectors IV of the display object being edited when the Display Editor is exited.

The next part of the control panel menu is the Tags Menu. This menu allows the user to switch between editing different sets of pictures that are defined for the display object. To select a particular set of pictures for editing, its tag is selected from the Tags Menu with the left button.

There are three options available for tags by selecting a particular tag with the middle mouse button pressed: Add a tag, Delete, and Copy Tag. Selecting Add a tag prompts the user to enter a name for a new tag and then creates a new set of pictures for the display object. Selecting Delete deletes the selected tag from the display object's definition. Copy Tag allows the user to copy an entire set of pictures to another set all at once. This can be useful if two sets of pictures are to be mostly the same with only a few differences.

Pictures are drawn for each display object by selecting graphical primitives from the main menu and then describing their sizes and where they are to be placed using the mouse cursor.

5. A Session with Chips

In this section, we will describe a sample interaction with Chips. We will go through the creation of a simple class of domain object, called a **FaceDomainObj**, to demonstrate the interactive facilities for creating and modifying part of an interface.

5.1 Creating a new domain object

To create a new domain object class a programmer first specializes the class **DomainObject**. This is done from a Chips **Browser**, selecting **Add New Class** from the title bar menu, sliding to the right, and selecting **Specialize Domain Object** from the menu that appears. The selection is shown in figure 1.

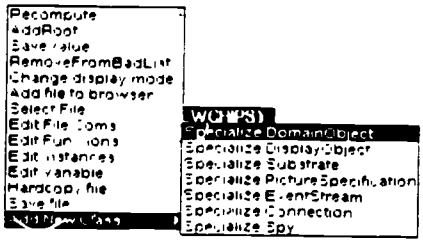


Figure 1. Specializing the class **DomainObject** from the browser

This creates the new class, **FaceDomainObj**. This class will inherit the functionality and properties needed by objects with graphical images.

5.2 Editing the display object of a class of domain object

Once this is done, a display object can be defined for this class of domain object and edited using the Display Editor. First, to create the new display object, we select Add Domain Object from the Chips browser (see figure 2).

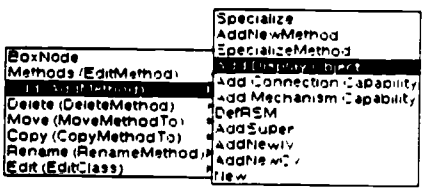


Figure 2. Adding a new display object

This creates an inspector that we will use to define our new display object. Since this new display object will use the default properties, we need only declare the tag that we will use to refer to the display object and then install the new display object by selecting Install from the title bar menu of the inspector (see figure 3).

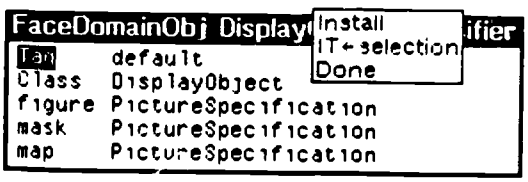


Figure 3. Installing a new display object with the Display Object Specifier

Next, we will edit the way our new display object looks using the Display Editor. To do this, we select Edit Display Object from the Chips Browser (figure 4).

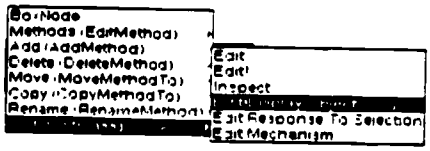


Figure 4. Invoking the Display Editor from the browser

5.2.1 Using the Display Editor

The Display Editor is a modified version of the Interlisp-D graphical editor, Sketch. It provides an interactive way to draw and edit pictures (figure 5).

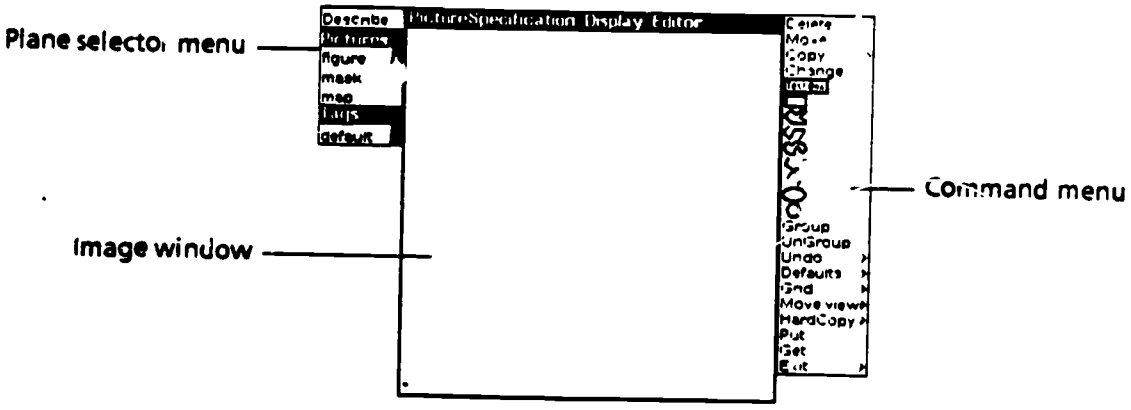


Figure 5. The Display Editor

5.2.2 Defining the figure picture of a display object

Using the graphical editor, we will first draw a figure for the display object of the class FaceDomainObj. This figure will consist of a circle for the outline of the face, two filled circles for the eyes, and two curves for the nose and mouth. The completed figure is shown in figure 6.

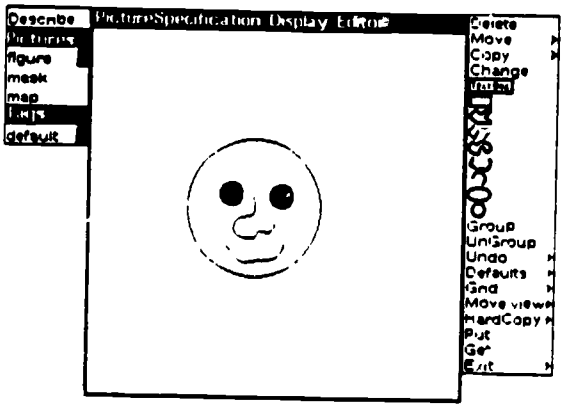


Figure 6. The figure picture of the display object of FaceDomainObj

5.2.3 Defining the mask picture of a display object

Next, we will define the mask picture of our display object. The easiest way to do this is to copy the outline from the figure picture to the mask picture. This is done by selecting the Copy option from the

main menu, sliding to the right and selecting Copy to picture from the submenu that appears. This presents a menu of pictures currently defined for this display object. Selecting mask from this menu establishes it as the picture to be copied to. We may then select any of the graphical primitives of our figure to be copied. We select the circle that defines the outline of our display object.

We now switch to editing the mask pictur by selecting mask from the control panel. The circle that we copied from the figure is the only thing currently defined for this picture. If this circle is filled in completely, the display object will be completely opaque. We'll just fill in part of the circle to demonstrate how to make part of a display object transparent. The completed mask picture is shown in figure 7.

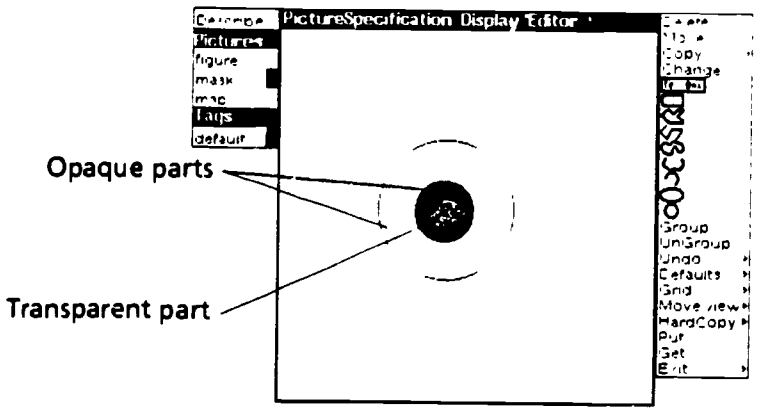


Figure 7. The mask picture of the display object of FaceDomainObj

5.2.4 Defining the map picture of a display object

Next we'll define the map picture of our display object. To do this, the outline is copied to the map picture using the procedure described above.

We switch to the map picture by selecting map from the control panel. Since we want to be able to button on the entire display object, the outline will be filled entirely. The completed map picture is shown in figure 8.

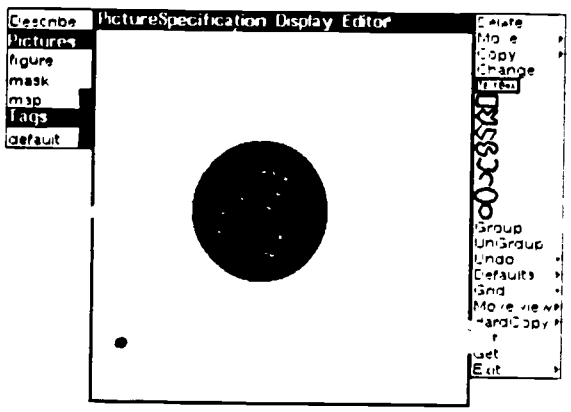


Figure 8. The map picture of the display object of FaceDomainObj

That completely defines our display object. Our display object is now available for use. To continue, we select Exit from the main menu, saving the definition of the display object to its PictureSpecification instances.

5.3 Using a domain object with a substrate

To use this domain object, we will need a substrate in which to place it. We can get a new substrate by selecting the Chips Icon with the middle button and selecting Create a substrate from the menu that appears (figure 9).

Figure 9. Creating a new substrate using the Chips Icon

Selecting in the background of this new substrate presents a menu of all domain objects currently defined in the environment (figure 10). You will notice that FaceDomainObj has been automatically added to this menu.

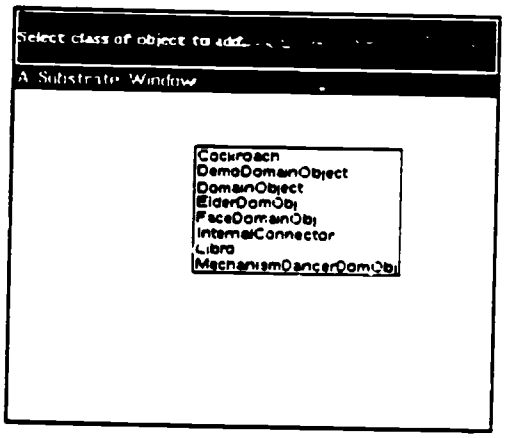


Figure 10. The default background menu of a substrate

Selecting FaceDomainObj from this menu creates a new instance of the FaceDomainObj class and places its display object in the substrate. We can create as many face chips as we want and place their display objects in the substrate. Three face chip display objects are shown in figure 11. In this figure, Face Domain Object Number 2 is overlapping Face Domain Object Number 3. Notice that Number 2 is partially transparent, revealing part of Number 3 around the edges. This is a result of how we defined our mask picture.

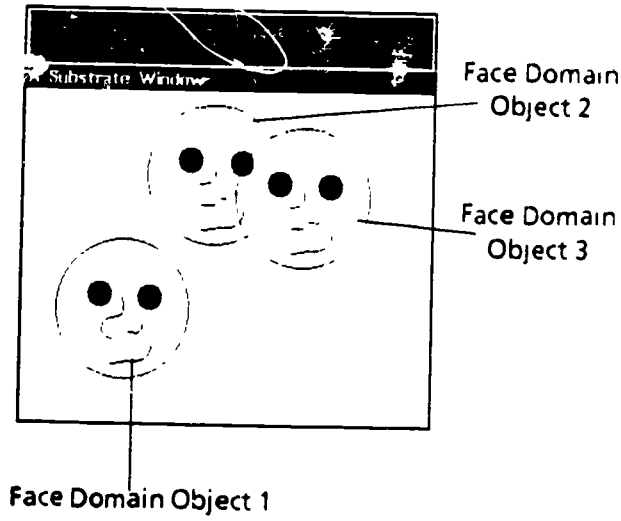


Figure 11. Three Faces in a substrate

Selecting a display object with the left button picks it up and drags it, following the cursor until another button is pressed. Display objects may be put down anywhere in the substrate or in any other open substrate on the screen. Selecting a display object with the middle button provides a menu of options that allow a user to edit various aspects of the display object and its associated domain object (figure 12). One option, Edit Display Representation allows the user to reenter the Display Editor, changing any of the existing pictures or adding pictures to the display object. We will add a picture that declares a different button response for part of the display object. We will declare that selecting one of the eyes of this display object sends the message Ouch to the domain object.

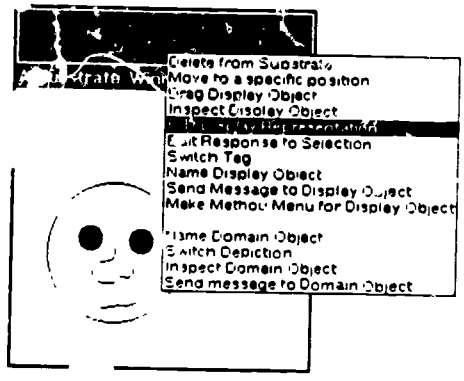


Figure 12. Selecting Edit Display Representation from the middle button menu of a display object

5.4 Interactively changing a display object

We want to define a new picture for this display object that will define an additional mouse-sensitive region. To add this picture, we select map from the control panel with the middle mouse button pressed and then select Subdivide picture from the menu that appears (figure 13)

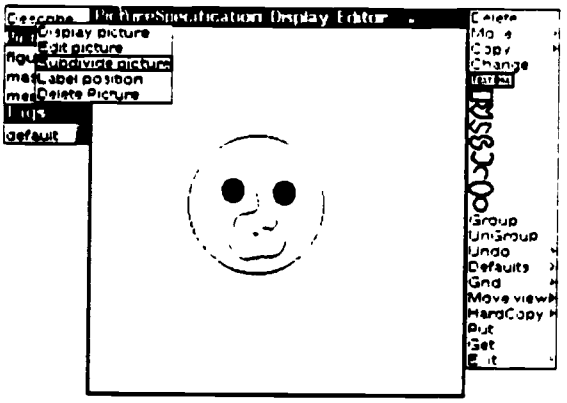


Figure 13. Adding a new picture to the display object of FaceDomainObj

The user is then prompted to enter a name for this picture. The Display Editor creates a new picture called eyes that will be used to draw the new mouse-sensitive region for the display object. It also adds the name eyes to the control panel. To define this picture, we need only copy the eyes from the figure picture to the eyes picture. Copying is done as described above. The completed eyes picture is shown in figure 14 below. Note that the eyes picture does not define the visual appearance of the eyes, which is done by the figure picture, but merely defines a new mouse-sensitive region.

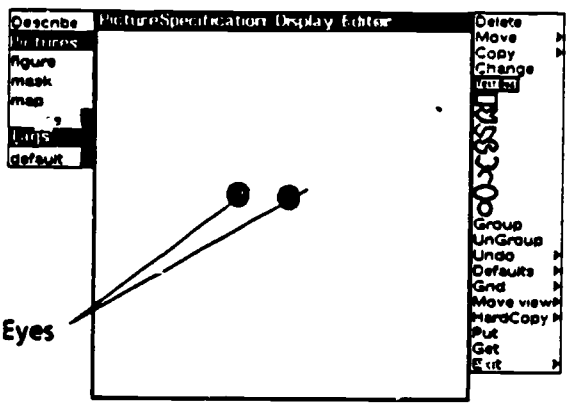


Figure 14. The eyes picture of the display object of FaceDomainObj.

Exiting the editor redefines our display object, defining a new mouse-sensitive region. To use this mouse-sensitive region, we must alter the display object's response to selection with the mouse cursor. This is done by selecting the display object with the middle mouse button pressed and selecting Edit Response to Selection from the menu that appears. This invokes the Interlisp-D editor DEdit on a form that describes how this display object is to respond to selection. We will add an expression to this form that tells the display object to send the message Ouch to its associated domain object whenever one of its eyes are selected. Figure 15 shows this form.

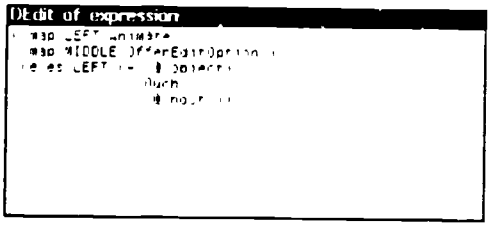


Figure 15. Editing the response to selection form for a display object

When an eye is selected from one of the faces in our substrate, the message Ouch is sent to that domain object and the corresponding method is executed, ringing bells and printing a message in the substrate window (figure 16).

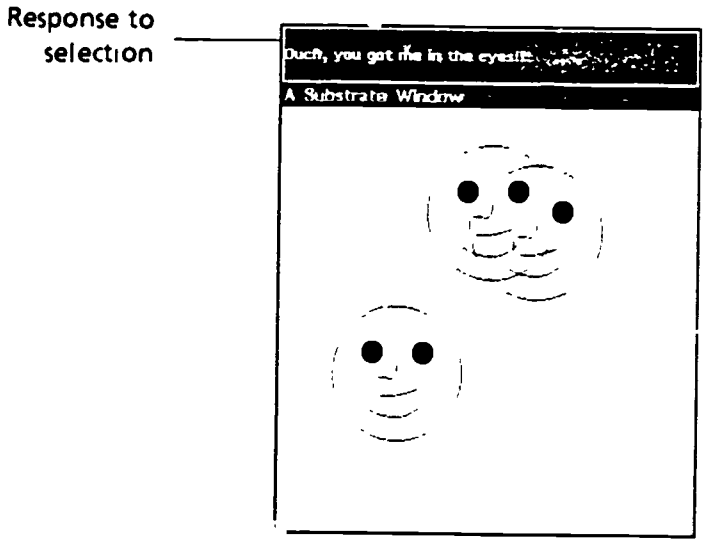


Figure 16. Selecting an eye of the display object of a face domain object

5.5 Conclusion

As you can see, using Chips, it is quite easy to define what a piece of your interface looks like and to determine its response to selection with the mouse cursor. Now that we have gone this far, it is easy to go ahead and develop the domain object's functionality more fully using the display object on the screen as the access point. The display object's selection response can be changed interactively The way the display object looks can be changed by changing the drawing. The internal data structures and the methods defining a domain object's behavior can be accessed interactively In short, the user interface can be quickly and easily modified.

References

Bobrow, D. G. and Stefik, M. The Loops manual. Tech. Note KB-VLSI-81-13. Xerox Palo Alto Research Center, Palo Alto, CA, 1981.

Bonar, J and Cunningham, R. Bridge: An Intelligent Tutor for Thinking about Programming In *New Horizons in Intelligent Tutoring*, edited by John Self, 1986.

Borning, A. The programming language aspects of ThingLab, a constraint oriented simulation laboratory. *ACM Trans. Program. Lang. Syst.* 3, 4 (Oct. 1981), 353-387.

Duisberg, R.. Animated graphical interfaces using temporal constraints. In *Human Factors in Computing Systems: CHI'86 Conference Proceedings* (Boston, MA). ACM, New York, 1986, pp. 83-96

Goldberg, A. J., and Robson, D. *Smalltalk-80: the Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

Hutchins, E., Hollan, J., and Norman, D. Direct Manipulation Interfaces, *User Centered Systems Design*, edited by Donald Norman an Stephen Draper, Lawrence Earlbaum Associates, Hillsdale, NJ, 1986.

Lesgold, A., Bonar, J., and Ivill, J. Toward Intelligent Systems for Testing University of Pittsburgh, Learning Research and Development Center Technical Report ONR/LSP-1, March 1987

Norman, D. Design principles for human-computer interfaces. In *Human Factors in Computing Systems: CHI'83 Conference Proceedings* (Boston, MA). ACM, New York, 1983, pp. 1-10.

Rosson, M. B., Maass, S., and Kellogg, W. A., Designing for Designers: An Analysis of Design Practice in the Real World. In *Human Factors in Computing Systems and Graphics Interface '87 Conference Proceedings* (Toronto, Canada). ACM, New York, 1987, pp. 137-142.

Sannella, M., Interlisp-D Reference Manual. Xerox Artificial Intelligence Systems, Pasadena, CA, Oct. 1985.

Schultz, J. *personal communication*, 1987.

Sheil, B. Power Tools for Programmers, *Datamation Magazine*, Feb. 1983

Stefik, M., Bobrow, D., Mittal, S., and Conway, L. Knowledge Programming in Loops: Report on an Experimental Course, *AI Magazine*, Vol 4, No. 3, Fall 1983, pp. 3-13.

Appendix A: Special Programming Techniques

Several aspects of the Chips program code take advantage of unique features of an open Lisp-based environment. While the techniques described in this section are not part of Chips, per se, they are interesting and generally useful.

A.1 A General Caching Function

Chips uses a function called CacheResults to be used with functions that have no side-effects (except perhaps storage allocation) and consume large amounts of time or space to compute. CacheResults takes a function and its arguments and returns the result of applying the function to its arguments However if the same function and arguments are supplied to CacheResults again, it simply return the same result it returned previously. This function is often used for pop up menus. Creating pop up menus is slow and it consumes large amounts of storage.

A.2 Self-Inspecting Code

When the programmer defines a new subclass of Chip, the menu whose items are all the subclasses of Chip becomes obsolete. There are three ways of dealing with this: one, ignore the problem and let the programmer fix this menu by hand, two, modify the method for defining a new subclass so that it updates the menu or in some way records the fact that the menu needs to be updated, or three, have the function that produces the menu check what classes are currently defined and if new ones have appeared, create a new menu, otherwise use the old one.

Using the cache and scheme three in the preceding paragraph, it is trivial to create menus that automatically update themselves only when necessary. When a menu is needed, the list of items that should be on the menu is used with CacheResults. A new menu will be created only if the list is different.

Schemes like this simplify the code. The programmer need not remember to update the list after defining a new class; the system notices automatically.

A.3 Fast Bitmap Intersection

Frequently direct manipulation interfaces need to determine whether irregularly shaped objects overlap. It is possible to take advantage of the fact that BITBLT is a very fast operation on Xerox 1100 Series workstations. The shadow bitmap and the relative displacement of one object from the other are used in a series of four BITBLT operations, and one BITBLT-like operation to a scratch bitmap Figure 1 illustrates the procedure.

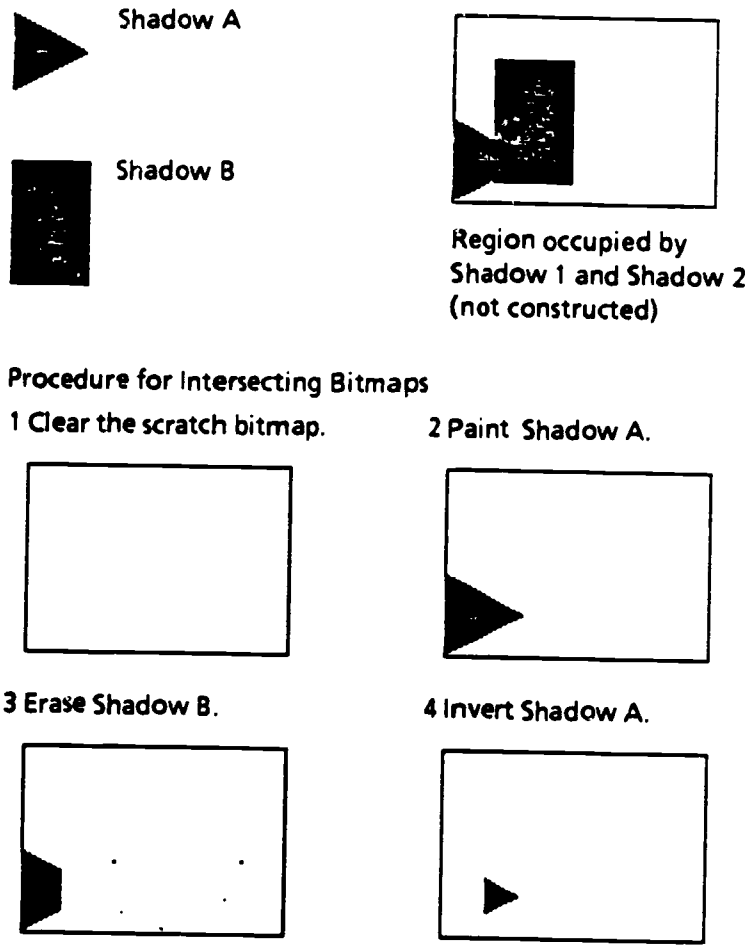


Figure 1. Intersecting bitmaps

If the scratch bitmap is blank after this procedure is executed then the two objects do not overlap. If the scratch bitmap is not blank then the black areas will be the areas that are common to both shadows with the given relative displacement.

It is important to note that it is not efficient to use the Interlisp-D function BITMAPBIT to scan for black pixels. Chips provides a function that does this scan efficiently called \BITMAPCLEARP.

A.4 The EditWhen Macro

The EditWhen macro is used throughout Chips to provide uniform access to the underlying code of the interface. The macro itself is very simple and is described below:

(EditWhen *keyNameorExpr who*) [Macro]

Parameters:

keyNameorExpr — either the name of a key (on the keyboard) or an expression

who — the name of a function or method

If *keyNameorExpr* is the name of a key, determines if the key is pressed or else *keyNameorExpr* is evaluated. If the key is pressed or the expression evaluates to a non-NIL value, whichever the case,

the function or method *who* is entered with all bindings set to their values during evaluation Upon exiting the editor, evaluation proceeds from the point of entry.

EditWhen basically allows a user to set up a conditional breakpoint in the code. We have used this macro to provide a uniform interface to the code of Chips. Throughout Chips we have strategically (we hope) placed calls to EditWhen that look like the following:

(**EditWhen** OPEN *functionOrMethodName*)

This allows a new user of Chips to find out about the code that is used to perform various interface functions by performing whatever action that he or she is interested in while holding down the OPEN key. So for example to examine the machinery behind figuring out how Chips determines what graphical objects are selected by pressing a mouse button, the user needs only hold down the OPEN key and then select a display object, a substrate, or whatever, with the mouse curor This will successively open each function or method as it is called, allowing the user to examine the function or method and then proceed by exiting the editor, continuing to hold down the OPEN key We hope this will help people become familiar with the underlying code of Chips.

Appendix B: Applications

Digital Circuit Editor and Simulator

A simple editor and simulator for digital circuits was created to help develop and demonstrate Chips See figure 1 below (readers familiar with electric circuits may notice that the ground is missing) Many common integrated circuit components are defined, including AND gates, OR gates, NOT gates, NAND gates, signal sources, wires, and switches. Creating the circuit editor was easy — once classes for circuit components were defined and their schematics were drawn — all that was required to build a circuit editor were a handful of methods for interactively connecting components with wires. The input/output behavior of the primitive gates are specified as a simple logical expression in Lisp The input/output behavior of the NAND gate is defined using a circuit consisting of an AND gate wired to a NOT gate. Thus demonstrating that new components, like the NAND gate, can be defined completely interactively without programming using the circuit editor and other editors provided by Chips. Signal propagation is implemented as a discrete event simulation. When a circuit component changes state, it recomputes its outputs and if they have changed, it signals the objects it is connected to. Each signal is considered an event, and is placed in a global event queue by sending a message to an event manager object. The event manager dispatches events in its own process so various control regimes can be implemented.

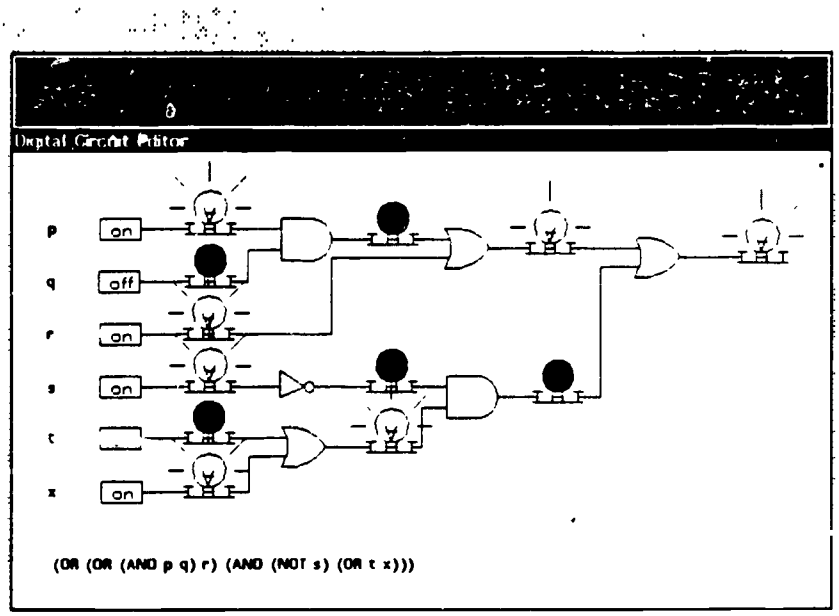


Figure 1. A digital circuit.

Bridge

Bridge [Bonar and Cunningham, 1986] is an Intelligent Tutoring System to teach introductory programming. Bridge teaches programming based on the idea of programming plans. Programming plans model the conceptual understanding which allows experienced programmers to combine several programming language constructs into common idioms. These plans are the same for any procedural programming language, though corresponding code would be slightly different. For example, when writing a program it is often necessary to keep a running count of something. The idea of keeping a count always has certain features associated with it, such as incrementing the counter and using the value of the counter.

Bridge teaches programming by "bridging the gap" between a student's understanding of specifying procedures in a natural language like English and the understanding needed to write a procedure in a programming language. The student works through three phases to specify a procedure in Bridge: a natural language phase, a programming plans phase, and a programming language phase. The student may request feedback about a proposed solution at any time.

In Phase 1 of Bridge, (see figure 2), the student constructs a solution to a programming problem by selecting and moving English phrases selected from a menu. Each phrase is a chip. These chips format themselves when the student moves them, highlight themselves at various times, and disappear when the student discards them.

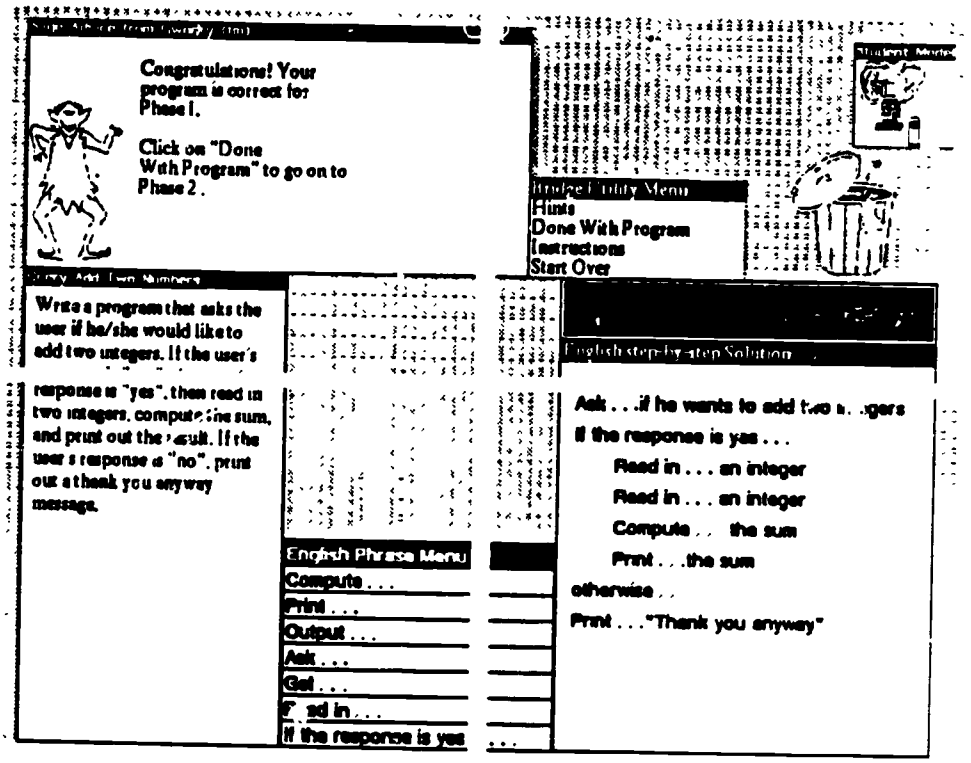


Figure 2. Phase 1 of Bridge

In Phase 2, (see figure 3), the student constructs a solution to the problem using a visual programming language (VPL). Each icon in the VPL corresponds to a programming plan. The student builds a solution by assembling the programming plans much like putting together a jigsaw puzzle.

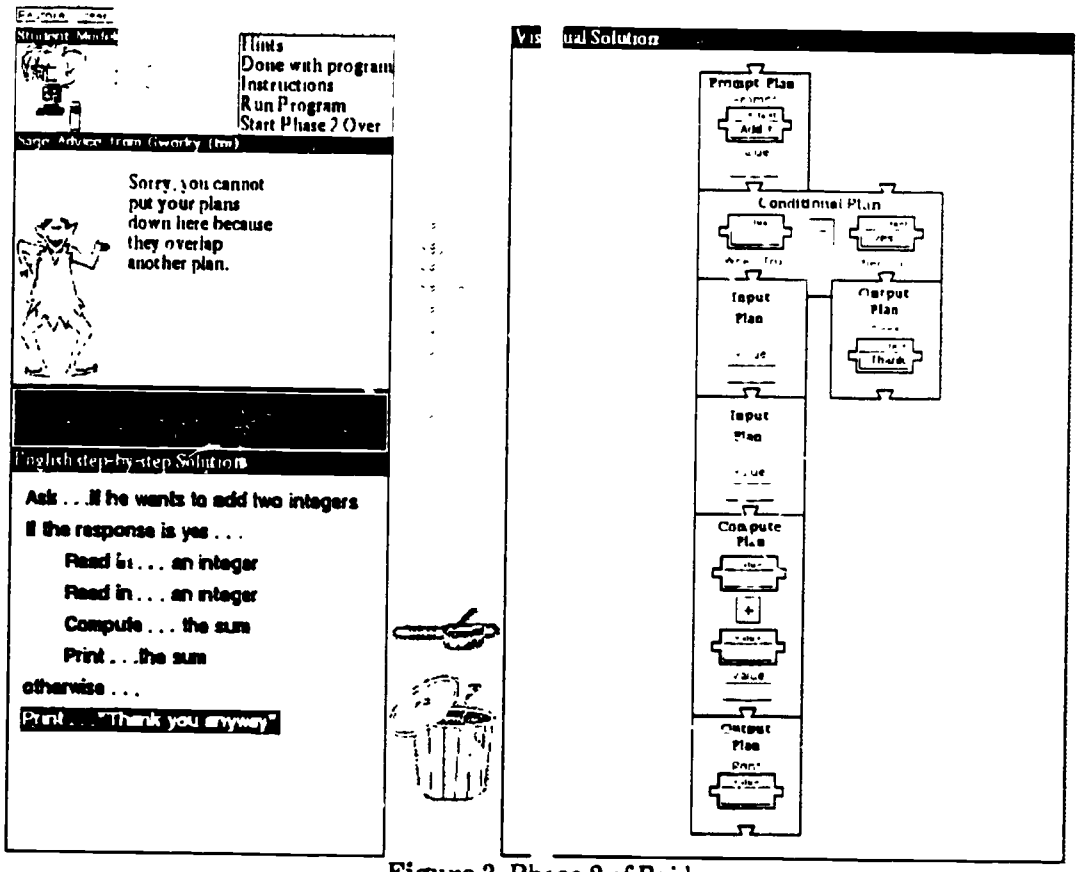


Figure 3. Phase 2 of Bridge

In Phase 3, (see figure 4), the student constructs a solution to the problem in Pascal, using a syntax directed editor.

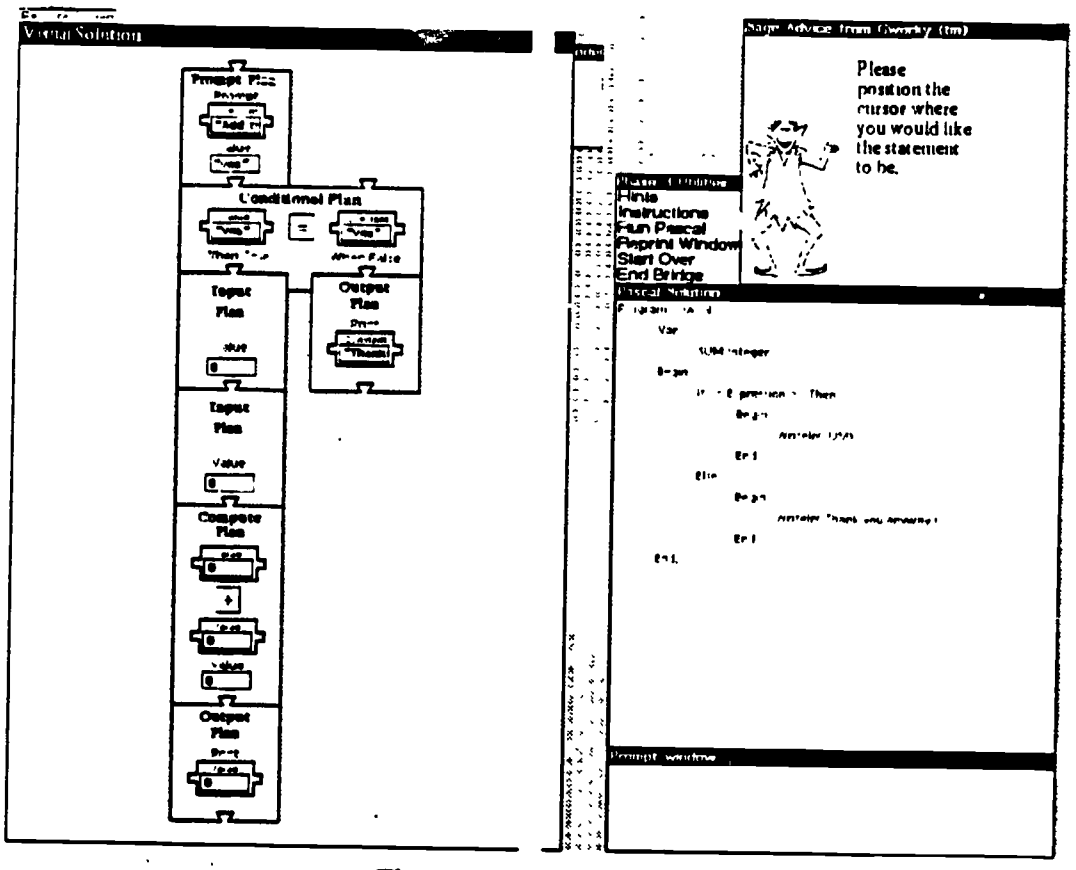


Figure 4. Phase 3 of Bridge

We used Chips to develop the visual programming language in Phase 2 of Bridge.

To construct a program using the VPL, the student moves the plans around on the screen and attaches them together by fitting a tab from one plan into a slot in another plan.

Different parts of the plans respond differently to selection by the mouse cursor. To use the value from one plan in another part of the program, the student selects the box marked Value. When this is done, an instance of the class ValueChip is created, attached to the cursor, and may then be placed inside another plan. Figure 5 shows a ValueChip instance that is about to be placed inside the Output Plan.

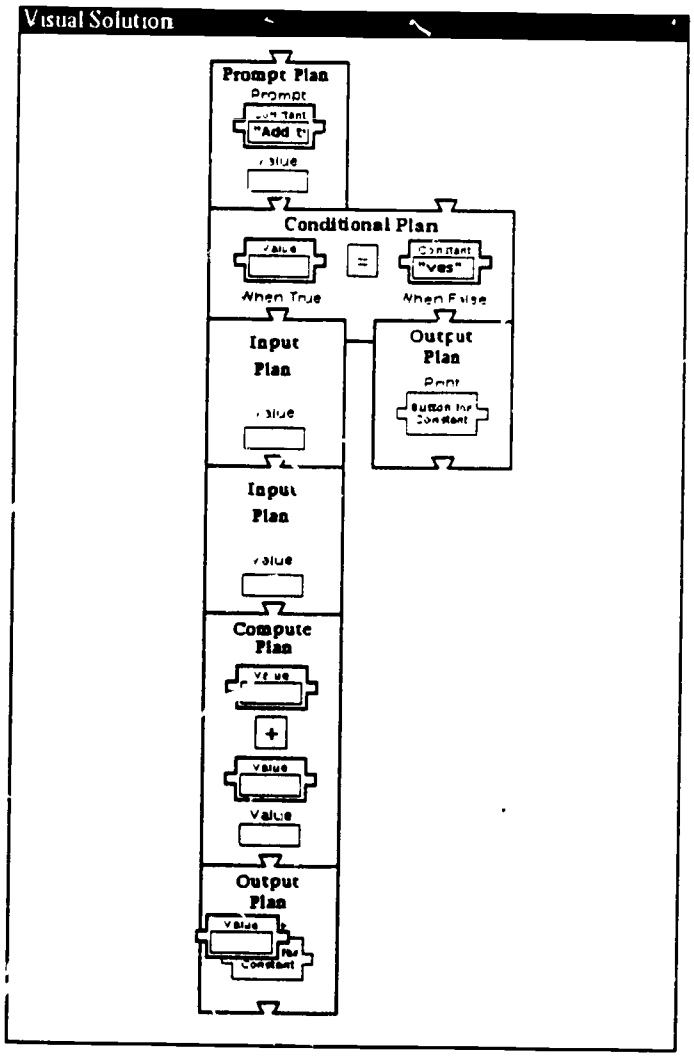


Figure 5. The Visual Programming Language from Phase 2 of Bridge

Once the student has constructed a proposed solution to the problem, the program may be executed. As each plan is executed, it inverts. Also, as values are updated, these values animate throughout the program to the location of their respective variables. Thus the VPL provides the student with an explicit view of both the control flow and the data flow during execution.

This application proved especially difficult because so little is known about the effective use of visual programming languages. Chips enabled us to do extensive iterative design of the language, developing six significantly different versions in three months.

MHO

MHO is an intelligent tutoring system for teaching basic direct current circuits that automatically generates problems for the student based on a model of what the student understands and dependencies among the domain concepts [Lesgold 87]. See figures 6-8. Circuits and meters are created with chips (instances, not integrated circuits). The circuit layouts are automatically generated. The student uses the meters to measure current, resistance and voltage between any two points in the circuit.

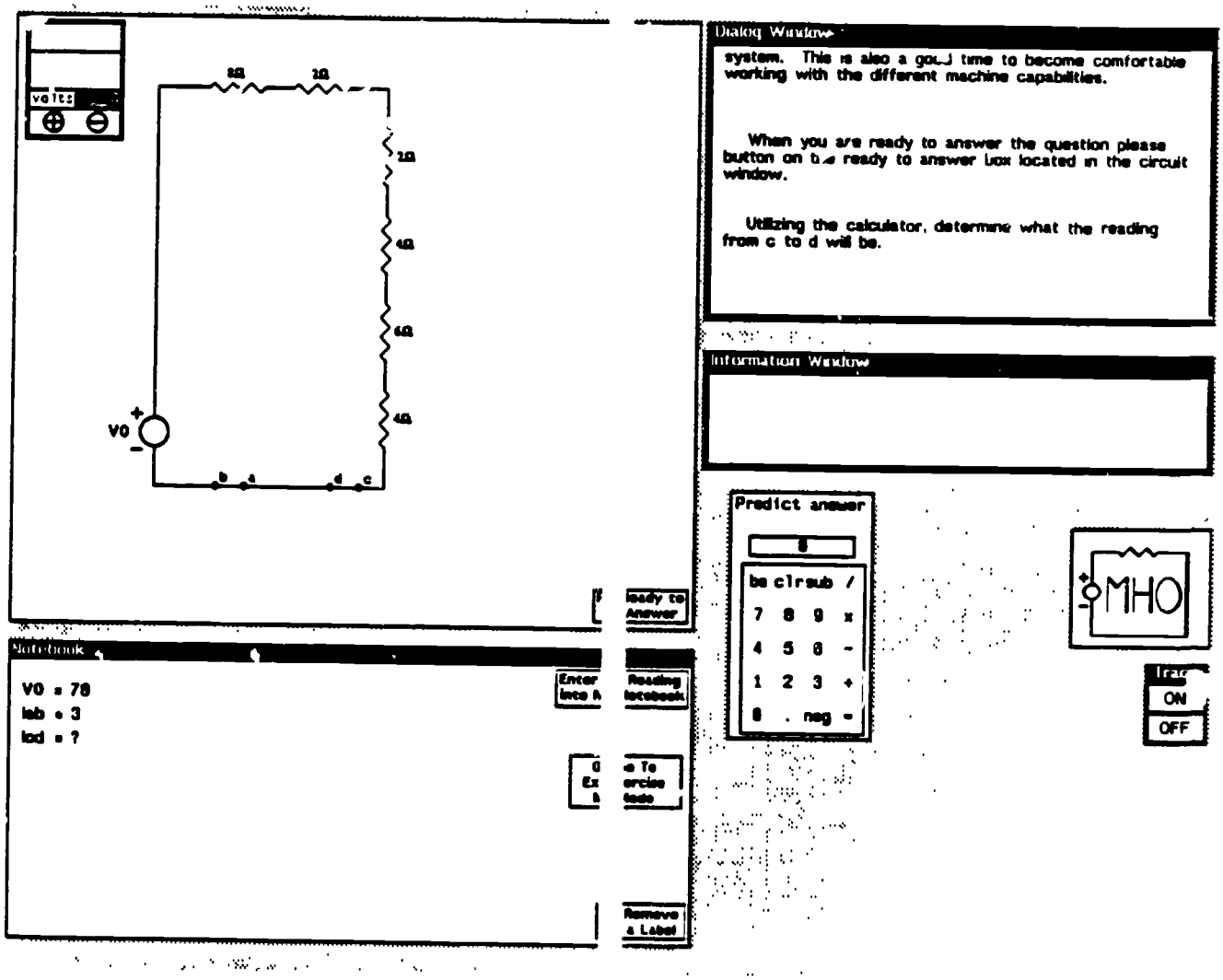


Figure 6. A screen from MHO, an intelligent tutoring system for DC circuits.

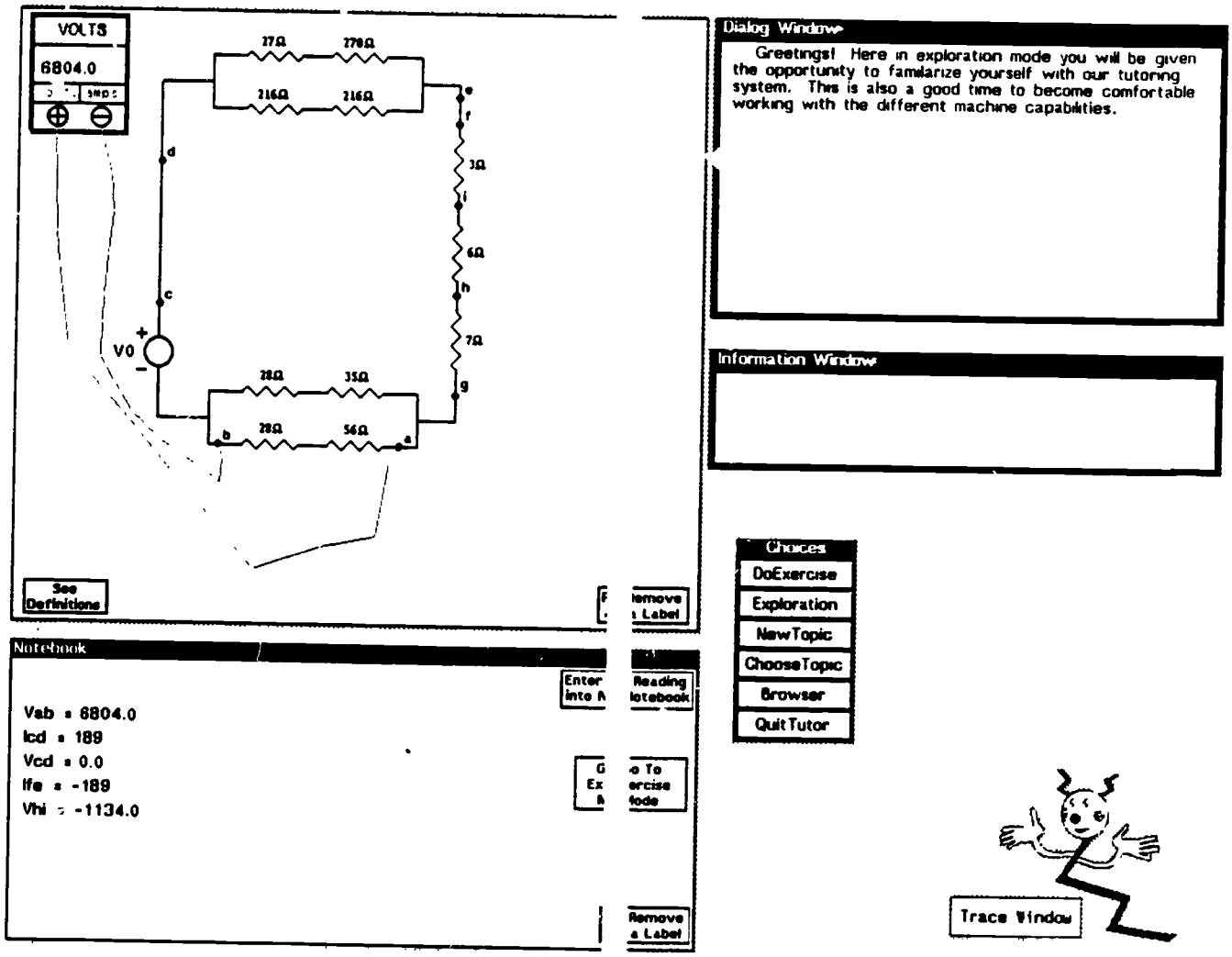


Figure 7. A screen from MHO, an intelligent tutoring system for DC circuits.

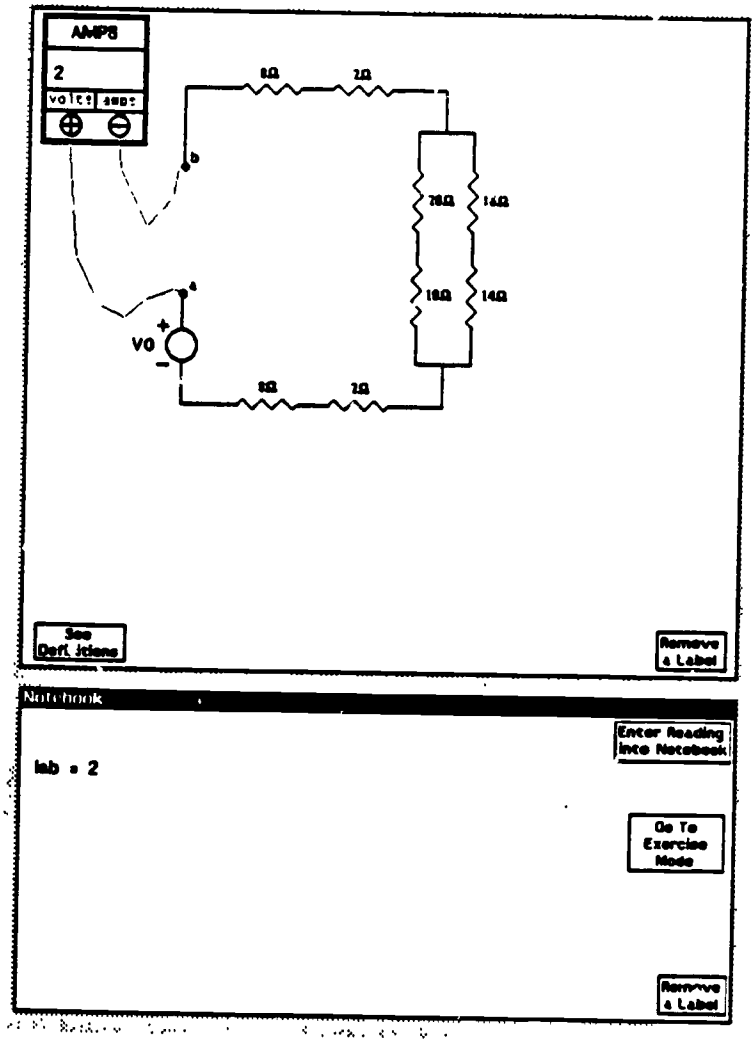


Figure 8. Using a meter to measure current between point a and point b.

Voltaville

Voltaville [Schultz, 1987] is a discovery world for students to learn about direct current circuits A circuit simulator and simple data collection and analysis tools are provided so that the student may explore electricity in a systematic manner. See figure 9. Voltaville watches to see whether or not the student actually is being systematic by searching for patterns in the student's behavior and by prompting the student to ı mulate and test hypotheses.

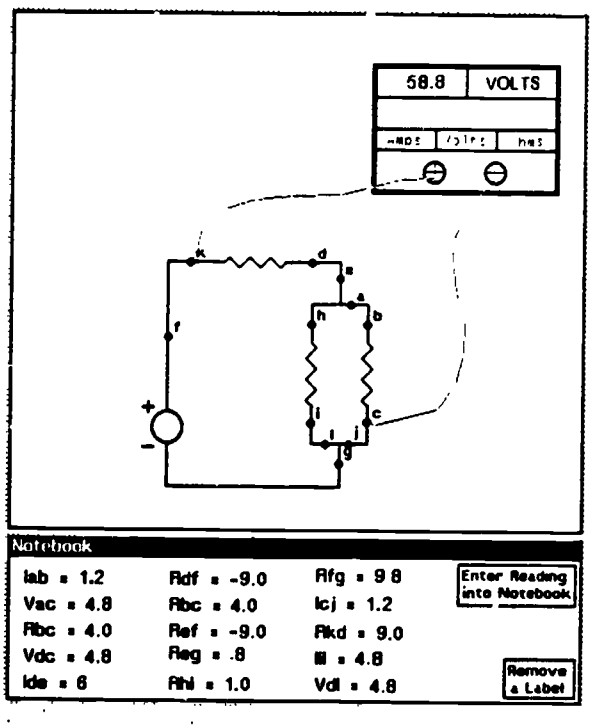


Figure 9. Measuring a circuit in the Voltaville discovery world.

Chips was used to build a circuit editor and simulator, and several animated simulations to illustrate concepts such as current flow. The animated simulations are part of a hypertext system of electricity concepts, which students can browse for background information, terms, and concepts. Students obtain a paragraph with illustrations by selecting a term from a menu, from a graph illustrating

relationships between concepts, or from a concept description that contains mouse-sensitive terms.

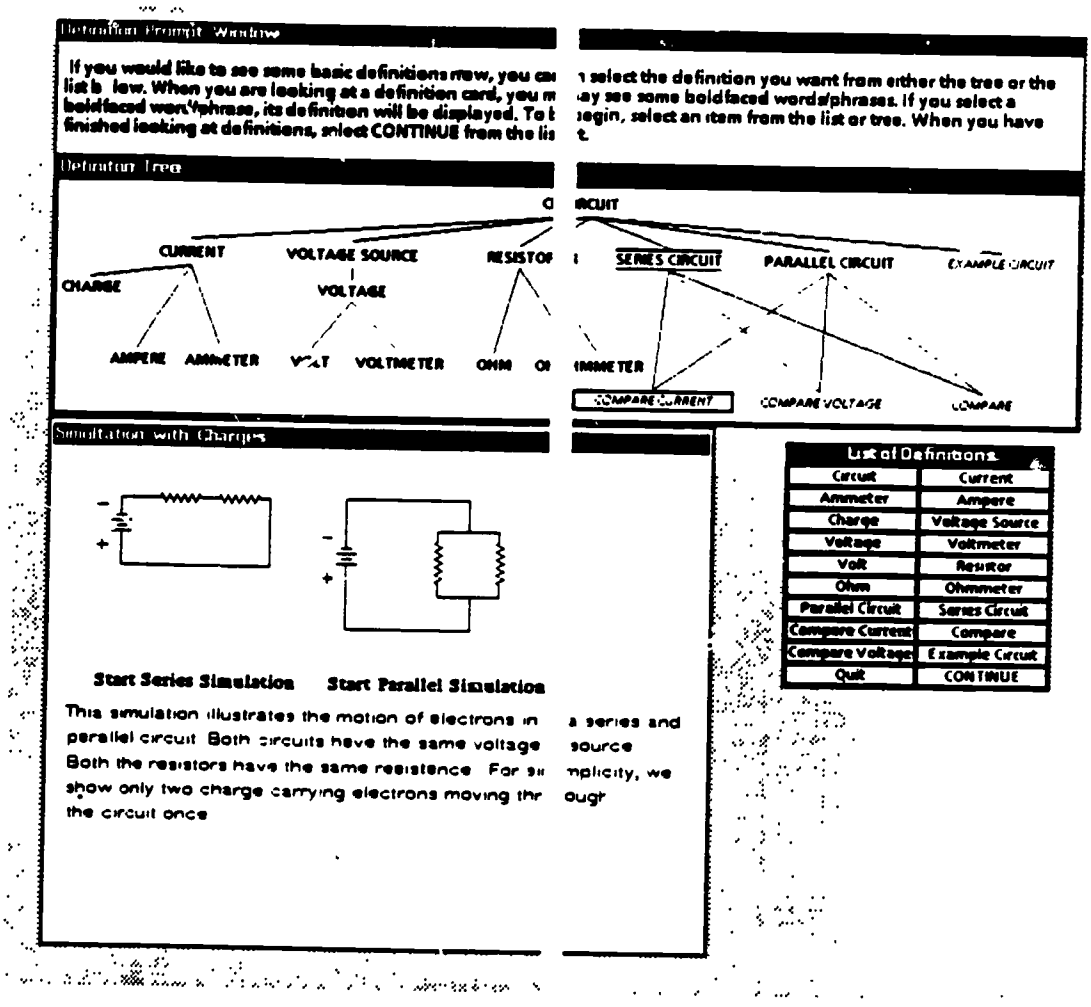


Figure 10. A screen from Voltaville, a discovery world for DC circuits.

The window labelled "Simulation with Charges" in figure 10, displays an animation sequence. Snap shots of the animation are shown in figure 11.

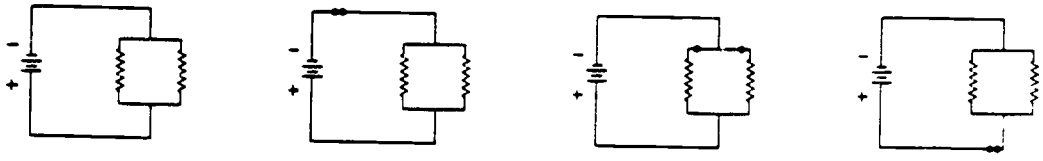


Figure 11. A simulation explaining current flow by animating electrons.

Glossary

AND gate	a component, a digital logic circuit with two inputs and one output; if both inputs are true then the output is true, otherwise the output is false; see also NOT gate
application interface	the human/computer interface to a particular application
button	*v.* to press one of the buttons of the mouse
Chips	a computer program for building graphical human/computer interfaces
class	a template for a particular kind of object including methods for responding to messages and variables
class browser	a tool for examining and modifying classes and their taxonomic relationships via a lattice diagram of classes in Loops
class library	a collection of class definitions designed for some common purpose
connection	a data structure provided by Chips for representing relationships between objects
development interface	the human/computer interface used to develop an application program
direct manipulation	a method for a person to control a computer program by manipulating pictures that represent objects of interest
direct manipulation interface	a human/computer interface that allows the user to command the computer by selecting and manipulating cartoon like icons, usually with a pointing device, such as a mouse (see direct manipulation)
display	*n.* the screen of the computer; *v.* to depict on the screen of the computer
display object	an instance of the class **DisplayObject** or one of its subclasses; determines how a domain object will be displayed on the screen
domain object	an instance of a subclass of the class **DomainObject**
DomainObject	a class of object that can be displayed as a mouse-sensitive picture
drag	to move a picture of an object on the display by animating it
editor	an interactive program for creating, displaying and modifying some entity of interest; usually maintains constraints that would be tedious to maintain by hand and provides a convenient interface to the entity
Event Queue	a queue of messages with time-stamps to be sent by an event queue process in an order consistent with the time-stamps
figure	a description of how a display object is displayed on the screen; stored as an instance of the class **PictureSpecification**
graphical primitives	programming language functions or menu options for drawing lines, curves, and text, etc. on the display
icon	a picture used in a human/computer interface to represent some object or concept in the world
Image Editor	a specialization of the Interlisp-D graphical editor, Sketch, which allows a user to interactively design the looks of a particular display object
inheritance	an aspect of object of object-oriented programming When a new class is created by specializing another class, it receives behavior from its super class
inspector	a tool for examining and modifying data structures in Interlisp-D
instance	an object in the computer produced by a class
Interlisp-D	a programming environment which provides sophisticated graphical programming tools for the interlisp programming language implemented on workstations
InternalConnector	a class of domain object that establishes a connection between the physical connectors of a domain object and its internal mechanism

IV	i.e., instance variable; a variable associated with an object whose value is local to that object
Loops	an object oriented programming language and tools for program development integrated with Interlisp-D
map	a list of elements, instances of the class PictureSpecification with mnemonic tags, that name different parts of a display object; determines the mouse-sensitive regions of a display object
mask	a description of which areas of a display object are to be opaque and which are transparent; stored as an instance of the class PictureSpecification
message	a command to an object
mechanism	a collection of domain object instances, usually connected together, representing a domain object's internal behavior
Mechanism Editor	a specialized substrate for editing a domain object's mechanism
method	a subroutine used by an object upon receipt of a particular message
mouse event	pressing or releasing one or more of the mouse buttons
mouse-sensitive	an area of the workstation's display which can be selected with the mouse to produce some effect
mouse-sensitive picture	a picture (usually associated with an object) which can be selected with the mouse to produce some effect
multiple inheritance	a capability provided by some object oriented systems, including Loops, which allows classes to inherit from more than one class
NAND gate	a component of a digital logic circuit with two inputs and one output; if both inputs are true then the output is false, otherwise the output is true; AND is an abbreviation for Not AND; see also AND gate, NOT gate
NOT gate	a component of a digital logic circuit with one input and one output; if the input is true then the output is false, otherwise the output is true; see also AND gate
object	an instance or class (see class, instance)
object-oriented programming	a programming methodology based on the metaphor of communicating objects, rather than procedures that operate on data types (see class, instance, message, method)
picture specification	an instance of the class PictureSpecification or one of its subclasses that defines the display and edit representations for part of a display object
physical connector	a mouse-sensitive region of a display object that has special significance to other display objects that may overlap it; used to establish physical attachment between display objects
plane	represents part of a display object in the Image Editor
select	to move the mouse cursor to something of interest and press one of the mouse buttons
Sketch	the Interlisp-D drawing editor; allows the user to interactively construct figures from graphical primitives
specialize	to define a new class or method in terms of an existing class or method
spy	an instance of the class Spy or one of its subclasses that may be used with a connection to redirect I/O or do recording of messages sent via connections
submenu	a menu that appear when the mouse cursor is slide out the right-hand edge of certain menu items indicated by a grey triangle (▶)
subregion	a region within a region; may be arbitrarily shaped
Substrate	a class of object appearing on the display as a rectangular window and used for displaying display objects, displaying prompts and processing mouse events

user interface management system	a computer program that provides a collection of interface elements. such as menus and dialog boxes; often include interactive tools for building prototype interfaces
workstation	a single-user computer with a large graphics display, several megabytes of memory, a processor capable of at least one million instructions per second, and a device for pointing to objects on the display, such as a mouse