MICROWORLD

by

GARY MICHAEL SMITHRUD

A THESIS

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

March 1990

An Abstract of the Thesis of

Gary Michael Smithrud for the degree of Master of Science

in the Department of Computer and Information Science

to be taken March 1990

Title: MICROWORLD

Approved: ______________________________

Sarah A. Douglas

MicroWorld is a tool for instructors to build computer-aided instructional lessons. Although MicroWorld's focus is a second (spoken) language tutoring system, the design allows the creation of several different types of lessons. MicroWorld produces lessons involving manipulation of graphical objects on a screen, using a direct-manipulation environment and its own programming language, which is designed for instructors who have little experience with computer programming.

This paper is divided into two major sections. The first section describes the high-level concepts of MicroWorld and includes a comparison with other similar environments. The second section describes MicroWorld's design and implementation, which includes MicroWorld's class structure, its user interface and programming language implementations.

An Abstract of the Thesis of

Gary Michael Smithrud for the degree of Master of Science in the Department of Computer and Information Science to be taken March 1990

Title: MICROWORLD

Approved: ______________________________
Sarah A. Douglas

MicroWorld is a tool for instructors to build computer-aided instructional lessons. Although MicroWorld's focus is a second (spoken) language tutoring system, the design allows the creation of several different types of lessons. MicroWorld produces lessons involving manipulation of graphical objects on a screen, using a direct-manipulation environment and its own programming language, which is designed for instructors who have little experience with computer programming.

This paper is divided into two major sections. The first section describes the high-level concepts of MicroWorld and includes a comparison with other similar environments. The second section describes MicroWorld's design and implementation, which includes MicroWorld's class structure, its user interface and programming language implementations.

VITA

NAME OF AUTHOR: Gary Michael Smithrud

PLACE OF BIRTH: Seattle, Washington

DATE OF BIRTH: August 22, 1961

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
University of Washington

DEGREES AWARDED:

Master of Science, 1990, University of Oregon
Bachelor of Science, 1986, University of Oregon

AREAS OF SPECIAL INTEREST:

Object-Oriented Programming Languages
Operating Systems

PROFESSIONAL EXPERIENCE:

Member of Technical Staff, Worldwide Computer Services, Inc., Boulder, Colorado, 1989-present

Summer Intern, US West Advanced Technologies, Boulder, Colorado, 1989

Research Assistant, Department of Computer and Information Science, University of Oregon, Eugene, Oregon, 1986-88

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to Sarah Douglas and Russell Tomlin, who led the team effort in designing and developing MicroWorld and who were willing to make me a part of this project. Furthermore, Sarah Douglas deserves a special thanks for helping me write and finish this thesis. Without her help, I doubt that I would have completed it. Finally, I wish to thank David Novick for his contributions to the project--I could not have done it on my own.

DEDICATION

I would like to dedicate this thesis to the following four people: my mother, Norma Smithrud, and my grandmother, Grace Miles, without whose support and financial assistance I would not have completed school; my brother, David Smithrud, for being my best friend for all these years; and most of all, my fiancée, Carolyn Avila. Without her love and understanding, the most frustrating and difficult parts of thesis writing would have been unbearable. Thanks to all of you.

TABLE OF CONTENTS

Chapter Page

1. INTRODUCTION 1

1.1 Introduction to Object-Oriented Languages 1
1.2 MicroWorld's Introduction 5
1.3 MicroWorld's Goal 7
1.4 Scope of the Paper 11

2. PREVIOUS WORK 12

2.1 Programming by Rehearsal 12
2.1.1 Programming by Rehearsal's Metaphor . . 14
2.1.2 User Interface 16
2.1.3 Problems with Programming by Rehearsal 21
2.2 HyperCard 28
2.2.1 Hierarchy of HyperCard's Objects . . . 32
2.2.2 HyperCard Scripts 34
2.3 Other Works 44
2.3.1 Building User Interfaces by Direct Manipulation 45
2.3.2 A Substrate for Object-Oriented Interface Design 48

3. MICROWORLD 52

3.1 Things, the World, and the Tutor . . . 53
3.1.1 Spatial Relations Between Things . . . 56
3.2 User Modification of MicroWorld's Objects 66
3.3 Instructor's View vs. Student's View . 68
3.4 Introduction to the MicroWorld Environment 70
3.5 User Interface 92
3.5.1 The World Menu 92
3.5.2 The Thing Menu 95
3.5.3 The Tutor and Window Menus 96
3.5.4 Programming Interface 96
3.6 MicroWorld's Programming Language . . . 104
3.7 Conclusion 104

4. COMPARISON BETWEEN PROGRAMMING BY REHEARSAL, HYPERCARD, AND MICROWORLD 110

Chapter Page

4.1 Programming by Rehearsal vs. MicroWorld 110
4.2 HyperCard vs. MicroWorld 114
4.3 Conclusion 118

5. MICROWORLD'S DESIGN 120

5.1 Introduction to the Object-Oriented Approach 120
5.2 The Effect of the Object-Oriented Approach on the Implementation . . . 127
5.3 Languages Used in MicroWorld 130
5.4 MicroWorld's Class Hierarchy 138
5.5 MicroWorld's Control Structure 153
5.6 User's Manipulation of Objects 156
5.7 Providing Multiple Worlds 161
5.8 Object Programming 163
5.9 Conclusion 169

6. MICROWORLD'S IMPLEMENTATION 170

6.1 User Interface Implementation 171
6.1.1 Dialogs 171
6.1.2 Menus 178
5.1.3 Windows 180
6.2 Implementation of User's Programming Language 185
6.2.1 UW-Method's Subclasses 188
6.2.2 Converting Between User's Source Code and Lisp Code 189
6.2.3 Inserting New Code into the Method's Definition 197
6.2.4 Propagating Changes Through Active Dialogs 201
6.2.5 UW-Method's Problems 202
6.3 Information Storage 204
6.4 Implementation Conclusion 209

7. CONCLUSION AND FURTHER WORK 210

BIBLIOGRAPHY 214

LIST OF FIGURES

Figure Page

1. Lifeboat Lesson 8
2. Flatland 10
3. Rehearsal World 14
4. HyperCard's Home Stack 31
5. HyperCard's Object Hierarchy 32
6. First Spatial Relationship Test 57
7. Describing Positions Around an Object 59
8. Global Relationship Between Objects 60
9. Position with Respect to Irregular-Shaped Objects 61
10. Object with Components 63
11. Lifeboat 63
12. World Menu 71
13. New World's Name 71
14. Backgrounds Defined in the New World 72
15. World Contains Backgrounds 72
16. Selecting Background File 74
17. New Background's Name 74
18. Newly Created World 75
19. Thing Menu 75
20. New Thing's Name 77
21. New Thing's Image and Mask 77
22. New Thing's Icon 78
23. New Thing 78

Figure Page

24. Thing's Menu When a Thing is Selected 79
25. Movement Command 79
26. Set Movement Path's Dialog 81
27. Actions Command 81
28. Editing Attributes 82
29. Editing Local Actions 82
30. Editing Landing Action 84
31. Editing Landing Action's Move Command 84
32. Global Actions 86
33. Editing the Arrange Command 86
34. Editing the If Command 88
35. Editing the Sound List 88
36. Viewing the Arrange Command 89
37. Tutor Menu 89
38. How the Tutor Teaches 91
39. What the Tutor Teaches 91
40. Tutor Menu when Tutor Exists 92
41. Saving a Copy of the World 94
42. UWorld's and Base-Worlds' Pointers to Things . . 151
43. Method Definition's List Format 188

LIST OF TABLES

Table Page

1. MicroWorld's Programming Language 106

2. MicroWorld's Class Hierarchy 139

3. Results Based on Image, Mask, and Destination Bitmaps 174

CHAPTER 1

INTRODUCTION

MicroWorld is a direct-manipulation programming environment used to build lessons for students. The instructor builds lessons by manipulating graphical objects on the screen and programming the response to student-created events by a menu-driven editor. MicroWorld is the next step in the programming environment of the Programming by Rehearsal research project.

Two important aspects are discussed in this paper, the concepts of object-oriented programming languages and MicroWorld itself. The object-oriented approach is an important tool for conceptualizing problems. MicroWorld is based on both the work in the Programming by Rehearsal project and the concepts of object-oriented languages.

1.1 Introduction to Object-Oriented Languages

A wide variety of programming languages exists today and more are being developed. The design emphasis has moved from a way to program the machine to a way of conceptualizing a problem. A prime example is object-oriented programming languages. MicroWorld uses the object-oriented concept as its

foundation.

An object-oriented languages is either fully object oriented or is an extension of another language. Smalltalk is an example of a fully object-oriented environment. The only program entities are objects and messages. Flavors and Object Lisp are examples of an object-oriented package for the Lisp language. Program entities are objects, Lisp data structures, messages and function calls. In other words, an object-oriented package causes the environment to be a mixture of both the extension and the base language.

In an object-oriented environment, everything is considered an "object." An object is an encapsulation of data and methods. Data is stored in an object through the use of instance variables which are unique to the object. Changing the information in one object does not change the information in any other object. Therefore, each object is a unique entity in the environment.

Executing a program involves sending messages between objects. When a message is received by an object, the method that corresponds to the message is performed. A method may access or set the values of the object's instance variables, send messages to other objects, or both.

Each object belongs to at least one class that defines the structure of the object and its relationship with other objects. The class contains the definition for both the in-

stance variables and the methods. When an object of a class is created, the instance variable definition of the class is used to determine the object's size. When an object receives a message, its class method definition is used to perform the message.

Some object-oriented languages have class variables (e.g., Smalltalk). Because a class variable is contained within the class itself, it is global to all instances of the class. By using class variables, information is shared between the instances of the class without the use of instance variables. Multiple copies of information create the need for a large number of messages to update that information.

Classes are arranged in a hierarchy. A class inherits both the variables and the methods defined in the classes above it in the hierarchy. Some languages, like Smalltalk, only allow a single inheritance between classes. A class is a subclass of only one other class. Other languages, such as Flavors, CLOS, and Object Lisp allow multiple inheritance between classes.

As an example, let Physical-Object be a class for physical objects. A large number of variations exists in the real world, but several features are common to all physical objects; they have mass, dimension, etc. Therefore, variables for these shared attributes are defined in the class Physical-Object.

The class Physical-Object can be subdivided in several ways. One possible division is between animate and inanimate objects. Since animate and inanimate objects are subdivisions of Physical-Object and both have the attributes of Physical-Object, it makes sense to have the classes Animate and Inanimate as subclasses of Physical-Object.

Both the Animate and Inanimate classes have unique attributes for their class. For instance, Inanimate objects have a purpose while Animate objects have goals. One method for handling goals and purposes is to store these attributes in instance variables. Another method, if multiple inheritance is allowed, is to make the Animate and Inanimate classes a subclass of a second class that defines the goals or purposes. Both classes inherit the attributes from the Physical-Object class, therefore, Animate and Inanimate have mass, dimensions, etc.

When an object receives a message, the methods defined for its class are checked first for the corresponding method. If the method is not defined for the class, its superclass (the next class higher in the hierarchy) is checked. The search continues up the hierarchy until either the method is found or the top of the hierarchy is reached and an error has occurred. Methods defined in the class of the object have the highest priority, therefore, the class can specialize the methods of its superclasses.

For example, most physical objects react the same to being pushed by another object. A possible method for "push" determines if the force is large enough to overcome inertia and if the direction of movement isn't blocked. If these conditions are met, the method starts moving the object. On the other hand, if an animate object is pushed, the reaction will be different. For example, the animate object will push the other object in return. Therefore, the method for "push" is different between the Animate class and the Physical-Object class.

Writing programs in an object-oriented system is the process of defining the most abstract, generic classes and working towards defining the more concrete, specialized classes. In some cases, subclasses of existing classes are tailored to meet the needs of the program. For example, the class Window creates a generic window on the screen. Subclasses of Window can be used to create special windows, such as Graphic-Window graphics, Text-Window for text, etc. In each case, the subclasses are more specialized than the generic class.

1.2 MicroWorld's Introduction

MicroWorld is based on the object-oriented concept, but is slightly different. Objects belong to one of a small number of classes (discussed later in this paper). These

classes are not part of a hierarchy, with respect to each other. Furthermore, the MicroWorld designer can only use the predefined classes and not create new classes. To specialize an object, two major attributes are modifiable, its image and its reaction to events.

A unique graphical image is associated with each object. The image provides the means for the user to interact with the object. Using the mouse, the user manipulates the graphical representation and the object is sent the appropriate event messages. For example, the user can send a "click event" message to an object by placing the mouse arrow over the graphical image of the object and clicking the mouse button.

Providing unique graphical images models the real world. No two objects look exactly the same in the real world. Therefore, in MicroWorld, any image, as long as it fits on the screen, is possible. Objects are allowed to look exactly the same by choice of the MicroWorld designer.

Another similarity between objects in the physical world and in MicroWorld is the object's reaction to events. Some objects in the physical world react the same to a given event. For example, two different balls fall when dropped from a height. Their response to the event is the same.

On the other hand, some objects respond differently to the same event. For example, one person responds differently

from another person to being hit. In order to model the real world, the environment needs to be flexible enough that objects of the same class can respond differently. The object's responses to events are each individually programmable in MicroWorld. An object can respond the same or differently to other objects in the same class by the choice of the designer.

Objects in MicroWorld interact with each other on a name basis. Each object has a unique name given by the MicroWorld designer. Sending a message to an object only requires its name. Matching the name to the object is performed automatically by MicroWorld.

1.3 MicroWorld's Goal

The goal of MicroWorld is to provide the designer with an environment to develop graphical and manipulative type lessons that model both the real world and the way we think. Instead of writing code in a standard high-level language, the programs are created by a combination of building the objects interactively and programming their interactions with other objects and with the user. This allows designers with little or no experience in programming languages to develop these types of programs.

The application currently under development using this concept is a second (spoken) language tutoring system and

Figure 1. Lifeboat Lesson

provides a good example for the type of programs MicroWorld can create. The instructor builds lessons consisting of tasks for the student to perform. These tasks, given in the language being taught, involve manipulating graphic objects on the screen. An evaluation of the student's performance is used to determine the next task.

One lesson, called "the lifeboat," was developed in MicroWorld (see Figure 1). The student is presented with a sinking ship. On this ship is a lifeboat, a man, a woman, and several items needed for survival. The student is asked to place each item into the lifeboat. When the student has successfully placed all of the items into the lifeboat, the

lesson is completed.

In this lesson, the lifeboat, the man, the woman, and each item to put in the lifeboat are all program objects. The process for defining the objects is straightforward. Each object is given a name, a graphical image that matches the physical object, and the object's attributes (including event responses) are defined. For example, one of the attributes can indicate that the object is movable by the student. In "the lifeboat," all objects, but the lifeboat itself, are movable.

Another part of the lesson creation process involves programming the interaction between the objects. An interaction in "the lifeboat" lesson occurs when an object is dragged by the student. The dragged object checks its position with the lifeboat. A response of "yes" or "no" is given, depending on if the object is "in" the lifeboat.

Another lesson, called "Flatland" was also developed in MicroWorld (see Figure 2). Instead of teaching object's names and positions with respect to a lifeboat, Flatland teaches the concepts of shape, size, and color. Three objects are moved into a line in the window's center. One object will have a unique characteristic from the other two objects, which is the concept being taught. For example, the square shape is being taught in Figure 2. The student is asked to select the unique object (e.g. the square in Figure

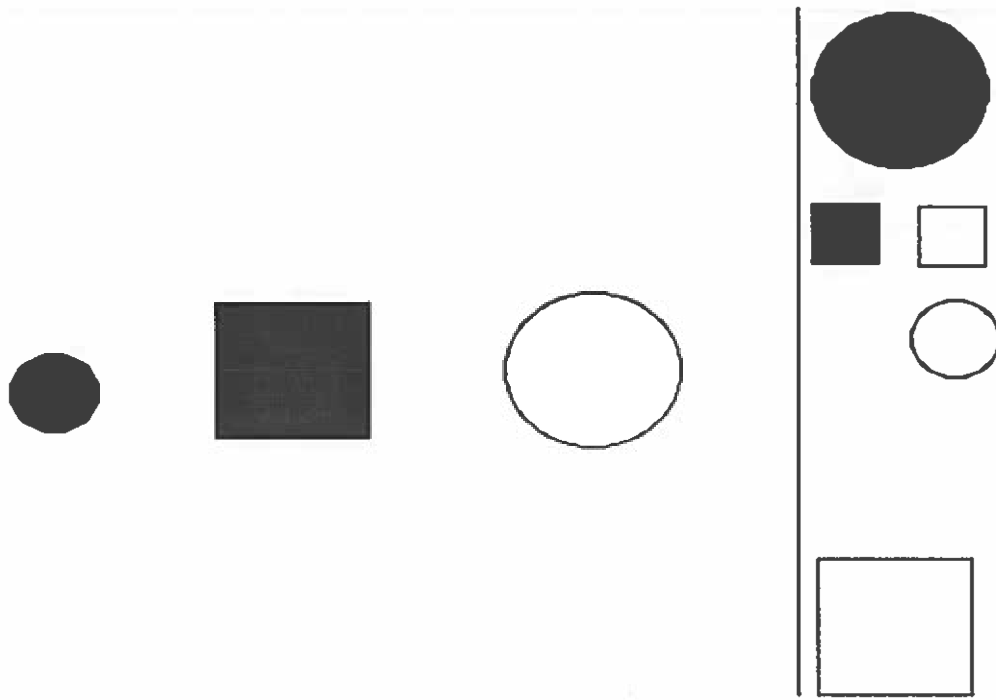

Figure 2. Flatland

2) in the language the student is learning. Once the student grasps the first concept, the lesson precedes with the next concept.

MicroWorld provides the tools and a suitable approach for this type of program. The programmer does not worry about bit-maps, event processing, opening windows, etc. Instead, the programmer thinks about objects and their interactions. This approach is closer to the way we think about physical objects in the real world.

1.4 Scope of this Paper

This paper consists of two major sections, the high-level concepts of MicroWorld and the design of MicroWorld. In the first section, Chapter 2 examines the research that lead to MicroWorld and work that occurred in parallel. Chapter 3 looks at the MicroWorld environment. Chapter 4 is a comparison between MicroWorld, Programming by Rehearsal, and HyperCard. The latter two systems are presented in Chapter 2.

In the second section, Chapter 5 describes the design of MicroWorld. Chapter 6 deals with the implementation of MicroWorld at the time this paper was started. This section may only be of interest to people working with object-oriented languages, or who are working on the next phase of MicroWorld.

Chapter 7 is the conclusion with areas for further research and problems with the current MicroWorld environment.

CHAPTER 2

PREVIOUS WORK

Two projects are of special interest with respect to MicroWorld. The first project, Programming by Rehearsal (Gould & Finzer, 1984), is the basis for this project. It was an early attempt to provide a direct-manipulation environment to create instructional programs. MicroWorld is a more robust environment, allowing a larger variety of lessons, and corrects some of the problems found in Programming by Rehearsal.

The second project is HyperCard™ (Goodman, 1987). Work on HyperCard occurred in parallel with MicroWorld. Although Hypercard was designed to be a general programming tool, it has similar features to MicroWorld and is included in this paper for that reason.

Other research projects and papers have influenced the design of MicroWorld. Some of these are discussed briefly at the end of this chapter.

2.1 Programming by Rehearsal

The purpose of Programming by Rehearsal was to construct an environment for teachers to build complex graphical

productions without the necessity of becoming programmers. It attempted to combine the design and programming phases into a single interactive system. The domain of the Programming by Rehearsal project is restricted to the design of interactive graphical lessons.

The design and programming phases represent a feedback loop. The designer gives the programmer a specification. The programmer, after a possibly long period of time, returns with an implementation of the design, which is checked and changes are requested. The loop continues until the implementation is sufficient.

Considerable time may pass for each cycle of the loop. During this time, the designer may have lost ideas, lost interest, or developed a better design. The amount of work involve may restrain the designer from implementing the better design.

One solution is to reduce the cycle time by combining the designing and programming phases in a system that does not require the designer to learn a programming language. As the designer specifies the system, the environment implements the specification and provides immediate feedback. The designer can then easily modify the design, without waiting for a programmer to implement the previous design.

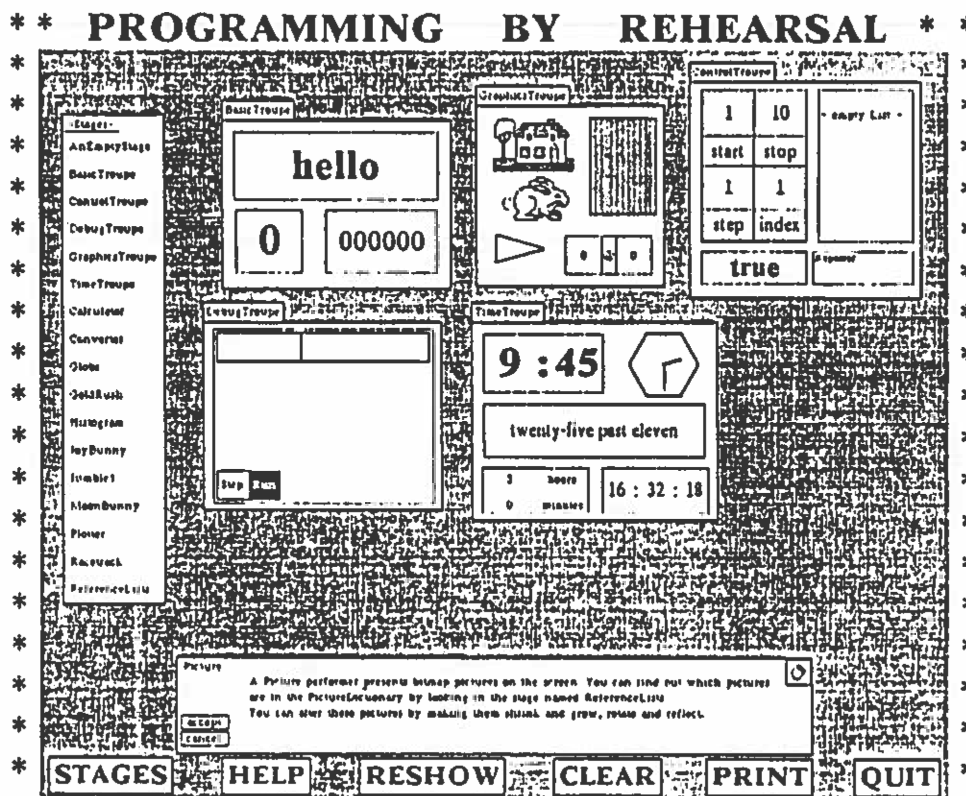

Figure 3. Rehearsal World

2.1.1 Programming by Rehearsal's Metaphor

Programming by Rehearsal uses a theater metaphor, which gives the designers a framework to conceptualize programs with a familiar, real-world reference. The environment used by the designers is called Rehearsal World (Figure 3).

Rehearsal World is written in Smalltalk-80, and is closely tied to the language. As mentioned earlier, Smalltalk is an object-oriented language. In Rehearsal World, the Smalltalk objects are considered *performers* on a *stage* in some *production*. The messages between Smalltalk

objects are consider *cues* for the performers.

A collection of primitive performers is provided by Rehearsal World. The designer is free to *audition* the performers by sending them cues and observing the responses. A production is created by copying the performers onto a stage, and setting their responses to cues.

It was recognized by Gould and Finzer that Rehearsal World would never provide an adequate number of primitive performers and that the performers would not have a sufficient set of cues. Therefore, the designer can create composite performers, by combining previously defined performers. New cues can be taught to both old and new performers.

The classes that defines the performers do not belong to an hierarchy from the instructor's view[1], therefore he or she cannot augment the performers' behaviors by creating new subclasses without programming in Smalltalk. Instead, the performers' behaviors are defined by setting instance variables to Smalltalk scripts. The class forms a prototype for the performer, which restricts the instructor to the behaviors allowed by the prototype. Although less flexible than a class hierarchy that allows subclass creation, prototype class structure provides a level of protection for the instructor.

---

1. The classes do belong to the Smalltalk class hierarchy.

2.1.2 User Interface

The user interface consists of a bitmap graphic screen and a mouse. The screen displays bitmap representations of the performers. The mouse is the primary input device. An on-screen cursor follows the movement of the mouse and clicking the mouse button sends a message to the performer under the cursor.

The screen initially contains a control panel, which holds six items, and a help area. Stages and performers are added to the screen during the creation of a production.

The mouse is the main device for interactions with Rehearsal World. The keyboard is used only occasionally for supplying parameters. A "cursor" follows the mouse on the screen as the mouse moves. The shape of the cursor provides feedback for the operation Rehearsal World is performing. For example, the cursor changes from an arrow to an hourglass shape when the desired operation takes time.

The mouse has three buttons used to interact with the objects on the screen. The *Perform Button* (left button) causes the selected item to perform its action, which depends on the type of the item. For example, if the selected item is the control button labeled *Quit*, then the session ends.

The *Name Button* (middle button) is used to retrieve the name of the performer. The cursor changes to display the name and follows the mouse until the user clicks the button

again. If the mouse is clicked in an area that accepts names, the name is pasted to that area. Pasting in the help area causes the display of a text description of the item.

The *Menu Button* (right button) produces a pop-up menu related to the selected item. The menu appears at the cursor point, and remains until the next mouse click. If the cursor is over a menu item when the button is clicked, the command associated with the item is performed.

2.1.2.1 Rehearsal World's Theater

When starting Rehearsal World, the designer is presented with an empty theater. The control panel and help area are the only visible items on the screen.

The control panel contains six sub-items, *STAGES*, *HELP*, *RESHOW*, *CLEAR*, *PRINT*, and *QUIT*. These commands are not for individual items, but work on all items in the theater. For example, *RESHOW* redraws all the items on the screen, *CLEAR* erases the current production, and *QUIT* exits Rehearsal World.

2.1.2.2 Rehearsal World's Stage

The *STAGES* button presents a menu of available stages. The first entry is *AnEmptyStage*, which creates a new stage. Performers from other stages are then copied into the new stage.

Performers are found in stages with names ending in *Troupe* and are grouped by similar functionality. For example, the *GraphicsTroupe* contains graphical oriented performers, such as a *Traveler* that moves along a defined path on the screen. To place a performer into the new stage, the performer's *Troupe* is opened, and the performer is copied to the new stage.

A performer responds to cues sent to it by other performers or by the user. Some cues are specific to the performer, while others are common among all performers. The *Menu Button* presents a pop-up menu to the user containing two parts. The top part is the cues common to all performers, such as *move*, *resize*, *copy*, and *erase*. The bottom part is the categories of cues for that particular performer.

When the user selects an item from this menu, a *cue sheet* containing the cues of the selected category is displayed. The performer responds immediately and appropriately to any cue selected from the cue sheet. Copies of the performer respond the same to a given cue.

Some cues require parameters to provide more information. The parameters are set in the cue sheets for the performer, by either typing the appropriate Smalltalk expression or by the use of an *eye* icon. While the *eye* is opened (by clicking the icon), Rehearsal World watches the actions performed by the user. When the *eye* is closed, a

Smalltalk script is made to repeat the actions performed by the user.

Associated with each performer is a *default action*, a *button action*, and a *change action*. The *default action* is only defined for some performers. When the user selects the performer, the *default action* is performed, unless the action is overridden.

This action cannot be redefined, but can be inhibited, by the designer. Sending the cue *inhibitAction* disables the *default action*, while the cue *enableDefaultAction* enables it.

A performer may become a button by sending a *becomeAButton* cue. When the performer is selected, the *button action* is executed instead of the *default action*.

If a performer's state changes, either by the user or the actions of another performer, then the *change action* is executed. What constitutes a *change* is defined in the description of the performer's type. For example, if a *Number* changes its value, then the *change action* displays the new number.

The designer may create scripts for both the *button action* and the *change action* by sending the cues *scriptForButtonAction*: and *scriptForChangeAction*:. A script of Smalltalk statements is sent with the cue. For instance, sending the message: *scriptForButtonAction: [x <- x + y]* causes the performer to add two number and store the results

when the user selects the performer.

Some performers include other specialized actions. For example, the List performer executes the action set by the *scriptForSelectionAction*: each time the List is told to iterate. The specialized actions are also programmable by the designer.

A *production* consists of the stage, its wings, and the performers contained within the stage and wings. Performers invisible to the student (either because they are waiting to appear or perform their functions backstage) reside in the wings. A production is different from a troupe, since the performers interact with each other using scripts and cues. The performers in a troupe are not programmed.

Each performer has a name assigned by the system upon creation. The name is created by appending a number to the type of the performer. For example, two *Picture* performers names are *Picture1* and *Picture2*. The designer may change the name by the *setName*: cue.

Names are unique within a stage. Different stages do not share the same context. Therefore, the script of one performer can only send cues to other performers within the stage. Cues are sent by giving the name of the performer, followed by the cue and parameters.

A new performer class is created by first creating a stage with performers and interactions as described above.

The stage is then saved under a name using the *storeWithName*: cue. The name becomes the type of the new performer.

To create a performer of the new type, the *makePerformerOfType*: cue is sent to the stage. The new performer type is added to the menu of available troupes, if its name ends with *Troupe* and is sent the *makePerformerOfType*: cue.

Cues are created and deleted by selecting the *CUES* category in the stage menu. The *scriptForCueNamed*: cue takes a name of a new cue and creates an empty cue template. Colons in the name are used as parameter markers. For example, a cue named *setX:Y:* takes two parameters, one is the value for X, the other is the value for Y.

The cue template contains two fields, *inCategory*: and *withScript*:. The *inCategory*: field defines the category of the new cue. The *withScript*: field is the Smalltalk code to execute when a performer of this type is sent the new cue.

2.1.3 Problems with Programming by Rehearsal

Programming by Rehearsal World was an attempt to provide a direct-manipulation, graphical environment for the building of lessons by designers with little programming experience. It succeeded in providing some isolation from such programming details as bitmaps, event handling, etc. Unfortunately, it does not isolate the designer from the Smalltalk

language.

Only simple scripts can be built using the watching mode (open *eye* icon) of Rehearsal World. Commands more complicated than selecting or moving objects on the screen are written in Smalltalk. Therefore, the designer is required to learn at least a subset of Smalltalk.

The Smalltalk environment is both large and complex. The designer needs to learn about classes, objects, and messages, before creating more than the simplest lessons.

For example a designer created, during a case study, the following script, which illustrates the problem (the '^' character represents the return arrow in Smalltalk):

```
[^ Picture1 getCenter getX * maxX / Joystick getWidth].
```

The designer must know the following to write the above statement.

- *getCenter* is a message to a picture and returns a point.
- *getX* is a message to a point and returns an integer.
- the order of the messages are *getCenter* then *getX* and not *getX*, *getCenter*.
- the multiplication is performed before the division.
- *getWidth* is a message to the Joystick object that returns an integer.
- the *getWidth* message is performed before the division operation.

Two items from the list should be noted. First, a point is not defined as a *Troupe*, but as a Smalltalk class. Therefore, even this simple statement is outside the designer's

environment. The situation becomes worse has scripts of higher complexity are created.

Second, Smalltalk's evaluation order of mathematical operations are in a left to right order, unless surrounded by parenthesis. Most other programming languages have an order of precedence. For example, the precedence order between multiplication and addition in Smalltalk is the leftmost operation first. In a language like C, the multiplication comes first. A designer familiar to other programming languages may become confused by this difference.

2.1.3.1 Multiple Processes

The use of multiple process in Rehearsal World is another problem area. Three approaches were examined, one process per performer, one process only, and one process per user action. Rehearsal World uses one process per user action, which is not transparent to the designer, and, thus requires the designer to program a multiprocessing systems.

The following example was given for the problem with one process per performer:

> On stage are three performers, a button named JumpButton, a Traveler named Rabbit, and a Number named Height. JumpButton's ButtonAction is first to tell Height to increment its value by a fixed amount and then to tell Rabbit to jump to Height's value. How high will Rabbit jump? If the three performers are totally independent, then we have no way of knowing what the value of Height will be when Rabbit starts its jump. Just because JumpButton is told first to change the value of

> Height does not mean that that will happen before Rabbit jumps again (Gould & Finzer, page 53).

Based on the problem above, the builder's of Rehearsal World decided against the one process per performer model.

The second approach uses a single process for the entire environment. Two reason against this approach were given. First, the designer must explicitly program each step of a multiple-object animation sequence. For example, Traveler1 and Traveler2 are to move simultaneously on the screen. The designer has to tell each Traveler to move a step in a loop instead of just telling the Travelers to move along their assigned paths.

Second, debugging is easier with multiple processes. One process can display information about another running process. This allows the designer to watch the actions of a performer, while the actions are in progress.

Rehearsal World uses one process per user action. In most cases, a process is spawned when the user or designer interacts directly with a performer. In the example given above, one process is created when the user selects the JumpButton. JumpButton changes the value of Height and then causes the Rabbit to jump, which is the desired behavior.

The problem with this approach is the creation of multiple processes using the same performer. For example, if the JumpButton is clicked twice in rapid succession, two processes are created. The second process may change the

Height before the first process starts jumping, or the two jumps could interfere by one jump starting before the finish of the other jump.

Rehearsal World's solution to this problem is to inhibit the button, with the *inhibitAction* cue, at the start of the action. Upon completion, the button is re-enabled with the *becomeAButton* cue. In the example, JumpButton's first command in its ButtonAction is to inhibit itself and the last command then re-enables itself. This solution forces the designer to handle inhibiting and reinstating the button - a person unlikely to understand the necessity.

Some observations on both the problems and the solution. The difficulty is that their system does not guarantee that messages are received in the same order that they were sent. In the example, the Height object receives a message from the JumpButton object and then from the Rabbit object (i.e. the request for the height value). The Rabbit will jump the incorrect height, if the second message is received first.

The underlying system can easily guarantee that messages are received in the correct order, since Rehearsal World is on a single processor computer. Only on a distributed environment does this become a hard problem. In a distributed environment, the system must handle lost, delayed, or duplicated messages, and also establish a global time. These problems are still being studied.

If multiple threads of control are allowed for each process, then more work is needed. With one thread, the process finishes processing the first message, before starting the second message. In the example, the Height object processes the *change value* message from the JumpButton object before starting on the *get value* message from the Rabbit object. The Rabbit object gets the value only after the update is finished, which is the correct behavior.

With multiple threads, the object must block all messages that access a value being updated, until the update is complete. Furthermore, the update message must be blocked until all threads that access the value are completed. This restricts the update "thread" from changing the value while it is in use by other "threads" and read messages received after the update message from accessing the value before the update. The simplest solution is to wait for all previous messages to complete and to block later messages until the update is complete.

From the example, the Height process receives a message to update its value. Since this is a write event, the process immediately blocks other messages that access its variables. The update "thread" waits for the completion of previous messages, updates the value, and then unblocks the messages.

When the Rabbit receives the message to jump, it requests the height from the Height object. If Height has received the update message, the request from Rabbit is blocked and, therefore, the Rabbit process is blocked. Height finishes the update, then processes Rabbit's request for information. Rabbit receives the reply, becomes unblocked, and jumps.

The problem with Rehearsal World's solution to multiple processes is that the designer is responsible for provide mutual exclusion. Providing mutual exclusion in a multiple processes environment is difficult for experienced programmers, especially if the production is large. Rehearsal World purpose was to provide an environment for designers with little programming experience.

In general, the goal of Rehearsal World was not met. In all but the simplest productions, the designer is faced with programming in Smalltalk and providing mutual exclusion between performers in the system. Two modifications would help the environment.

First, either restrict the designer to the subset of the Smalltalk language or provide a new language that is applicable to the creation of productions. Currently, the entire Smalltalk environment is accessible to the designer, although, most of it is not needed. The designer can easily crash the entire system with a mistake. Providing this power

to inexperienced designers is not wise.

Second, guarantee that messages are received in the send order. Rehearsal World is close to the level of Smalltalk, but this closeness isn't required. The designer's code is easily checked for such things as accessing or setting of instance variables. When a message is received that sets the value of the instance variable, then all messages that use the variable should be blocked.

The designer's scripts can easily be surrounded by Smalltalk code to handle correct message order. A performer's process can be designed to not allow the processing of a message until the previous message of the same type is completed. These extensions are invisible and makes a better environment for designers with little programming experience.

2.2 Hypercard

HyperCard is an authoring and information tool for the Macintosh. It shares many similarities with MicroWorld, although the work was done in parallel and without knowledge of each other.

HyperCard consists of five components, stacks, backgrounds, cards, fields, and buttons. A stack is similar to an application; it is created to solve a certain set of problems, or provide a certain service. For example, the

address book and the calender provided by HyperCard are both stacks.

A stack contains a combination of the other components of HyperCard. A stack may either be homogeneous or heterogeneous. A homogeneous stack appears to have the same background for each card, although, several backgrounds may be used. A heterogeneous stack contains more than one background with a different look.

Cards in the stack hold the user's information, with each card as a single unit. For example, an address in the address book is a card. The background's graphic defines the common look of each card in the stack.

A stack has several layers that are important for scripts and event handling (see below). An important division exists between the background and card layers. The card layer is always higher than the background layer.

When a field or button is created, it is placed on the top layer. The user can change the layer of a field or button by moving the item up or down in the layers. For example, moving an item down one layer moves the next lower item up one layer.

The background is the lowest layer. Buttons and fields created before a card is created are placed above the background layer and below the card layer. These buttons and fields belong to the background.

After creating the card layer, newly created buttons and fields are above the card layer, unless HyperCard is told specifically to place the new items above the background. Buttons and fields above the card layer are unique to the current card. If another card is selected from the stack, it does not have the buttons or fields defined for the previous card.

Both the background and card layers may contain graphical images. The background defines the base image; each card's information is drawn above the background. If the card layer contains an image, it is unique to the card.

Buttons contain an icon and are either visible or invisible. Invisible buttons are useful when the graphical image at a lower level provides the need information. For example, a stack to teach a student the different parts of a car can use invisible buttons over different areas of the graphical representation of the car. The student is then able to select a car part by clicking on the representation (for example, a tire), and the invisible button handles the click.

Fields are used for displaying text, and are also either visible or invisible. Invisible fields serve two purposes. One, hide text to be displayed later. For example, the answer for a guessing game is in a hidden field; only displayed when the student gets it right, or gives up.

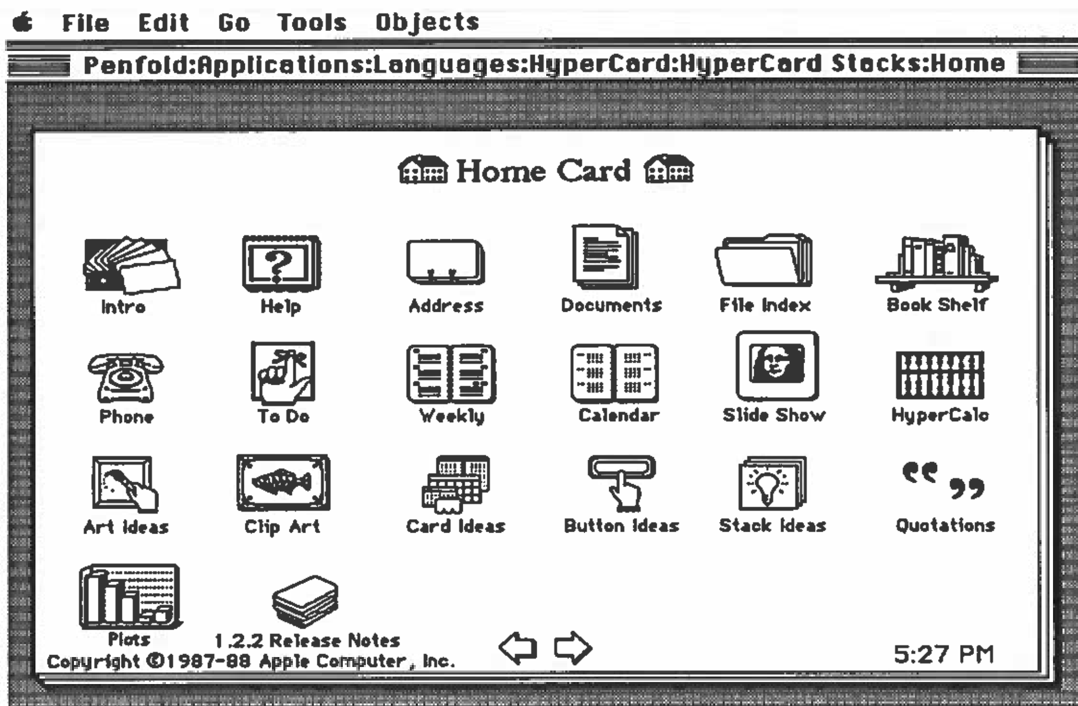

Figure 4. HyperCard's Home Stack

Second, a place holder for the card's state information.

Figure 4 shows HyperCard's home stack. The large icons (such as "Address" or "Calendar") are in the card level and an invisible button exists above each icon. Pressing the button causes HyperCard to load the appropriate stack. The two arrow buttons are in the background level. Pressing the right button displays the next card and pressing the left button displays the previous card. Since these arrows are in the background layer, they appear in each active card, therefore, the arrows allow the user to move through all cards in the stack. The current card in Figure 4 does not

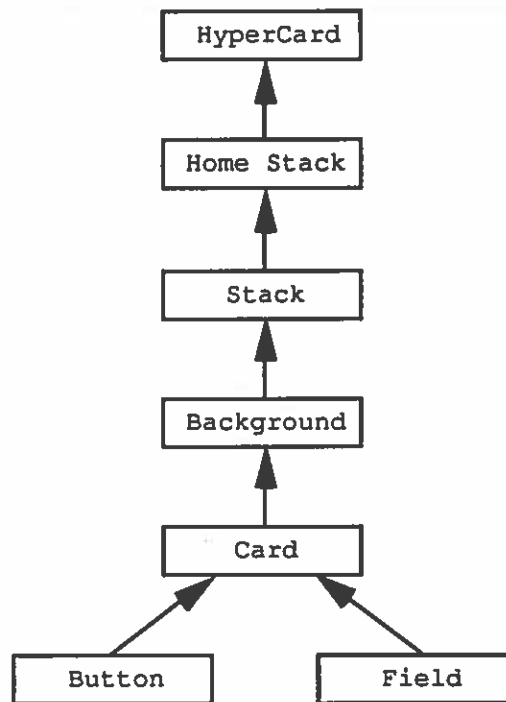

Figure 5. HyperCard's Object Hierarchy

contain any fields.

2.2.1 Hierarchy of HyperCard's Objects

The hierarchy of HyperCard is similar to object-oriented languages, although HyperCard does not have a class structure. HyperCard's hierarchy is shown in Figure 5.

Two entries in Figure 5 need mentioning. HyperCard refers to the system itself. If a message is received at this level, and is either not a primitive nor external command, then an error has occurred.

The Home Stack is the stack in charge of other stacks. For example, it allows the user to select between stacks in the system, create new stacks or delete old stacks.

When a message is sent, the hierarchy is searched for the script that corresponds with the message. The entry point in the hierarchy depends on two things, how the message was created and the position of the mouse.

A message is either generated in a script or by the system. If the message was generated by a script, then the search begins at the same level as the script. For example, if a script at the background level sends a message, the search begins at the background level and moves upwards until found.

Messages generated by the system have three possible entry points, depending both on the message and the position of the mouse. If the message is either *mouseDown*, *mouseStillDown*, *mouseUp*, *mouseEnter*, *mouseWithin*, or *mouseLeave*, then the message is sent to the top button or field under the current mouse position. Otherwise, the message is sent to the current card.

The entry point for other messages is based upon which item uses the message. For example, the messages *newButton* and *deleteButton* are sent to the button level, and *newField*, *openField*, *closeField*, and *deleteField* are sent to the field level. All other system generated messages are sent to the

current card.

2.2.2 HyperCard Scripts

Scripts can send messages to the current object or to other objects. The commands *on <command>* and *end <command>* are the entrance and exit of a script. *<Command>* is the name of the message this script performs. For example, *on mouseUp…end mouseUp* defines a script to handle mouse up events.

Information is stored in areas called containers. The most common containers are text fields and variables. Three specialized containers, the *It* variable, the *Message Box*, and *Selection*, are provided and maintained by HyperCard. Some commands implicitly use these containers.

All containers store information as strings. If the information is used in another form, such as a numeric value, the string is converted before use. Results are converted back into strings before storage. Uniform format allows the same treatment of all containers.

Fields hold information for a card. This information is unique whether the field is in the background or card layer. A different card holds different information in all fields.

Several ways exists for accessing a field in a card. The domain (background or card), field's name or number, and the card's name or number gives a complete path to a field.

The domain indicates which area the field is defined, either above the background layer or above the card layer. HyperCard defaults to the background layer, if the domain is missing.

A field is referenced either by name, local ID, or global ID. The field's name is set by the user. Local ID is relative to the field's layer level within the card, with respect to other fields. For example, a field one layer above the background has a local ID of one.

Global IDs are uniquely assigned by HyperCard on the creation of any object in the system. HyperCard automatically saves information to disk, therefore, these numbers are unique between sessions and stacks.

Cards are also accessed through name, local ID or global ID. As with fields, the name is set by the user and the global ID is defined by HyperCard. The local ID is the relative position of the card within the stack. For example, the second card in the stack can be referenced as *card 2* or as *second card*. If the card designation is missing, HyperCard uses the current card.

For example, the command *card field ID 418 of card Cari* accesses the information from the field with the global ID of 418 from the card named *Cari*. The keyword *ID* indicates a global ID. The field is above the card layer, as specified by the *card* keyword. If the local ID of the *field* is three,

and *Cari* has a global ID of 132, then the command *card field 3 of card ID 132* provides the same information.

Unlike other containers, information in a field is saved between sessions. This fact allows invisible fields to be used as instance variables for the cards; the information is retained between scripts and between sessions. Global variables are volatile, and are global to HyperCard. Therefore, a global variable is a poor choice for use as instance variable.

Local variables are created by first reference and are not declared before use. Further references within the script accesses the same variable.

Global variables are declared before use. Other scripts must declare the variable before accessing it. The variables are global to all stacks allowing stacks to share information. Global variables retain their values when exiting a script, but not when exiting HyperCard.

The special local variable called *It* is used as the storage point for many HyperCard commands. For example, the *get* command places the value into *It*. The user is free to access the *It* variable the same as any other variable, but must remember that a command may change the value.

The Message Box is a window in HyperCard and is convenient for sending messages to the user. Furthermore, the user is able to type commands and execute them from the

Message Box. This container differs from other containers in one aspect. Only one line of text is allowed in the Message Box.

The Selection container contains the currently selected text. The contents of this container is treated as any other container, which includes replacing the contents with a new value. The new information is automatically placed in the field of the currently selected text.

All containers hold strings, which may include many words and lines of text. HyperCard provides commands to subdivide the text into components. The text can be divided into characters, words, lines, and items. Items are defined as the text between commas.

The keywords *character* or *char*, *word*, *line*, and *item* are used to subdivide the text. If the keyword *to* is added, then a range is selected. For example, the command *word 1 to 4 of field 1* returns the first four words of the first field.

Further subdivision is accomplished by nesting the component expressions. For example, the command *character 2 of word 3 of field 1* returns the second character of the third word of field one. The expressions are defined from the narrowest to the broadest subdivision.

Commands are typed by the user, with one command per line. If the command is too long, a *soft* return is used to continue the command on the next line. The format of a

command is the name of the command followed by the parameters and are separated by commas.

Commands may contain optional keywords as part of the command. These keywords are ignored, but provide a more readable source in some cases. For example, the *go* command has an optional keyword of *to*. Therefore, the command can either be *go <destination>* or *go to <destination>*.

Parameters may also have keywords. In some cases, the keyword provides more information to the command. Other times, the keyword makes the command more readable. For example, the *put* command is *put <source> [into | after | before <container>]*. The keywords *into*, *after*, and *before* tells the command the location to put the information with respect to the container. The keywords in the command *click at <location> [with <modifier key>]* are used for readability.

Functions are defined differently than class definitions. The format *function <function name>…end <function name>* is used, instead of the *on <command name>…end <command name>* format used for commands. Values are returned by the *return* command. If the script ends before a return is reached, then the empty string is returned.

Parameters are also passed differently. The parameters to a function are separated by commas and surrounded by parenthesis. Parameters to commands are separated by a space

and are not surrounded by a special character.

Commands may be grouped into the following categories:

- Navigation
- Action
- Arithmetic
- Object Manipulation
- Screen Manipulation
- Sound
- File Manipulation

These commands are too numerous to name individual, but the following is a general outline of the commands.

Navigation commands move the user from one card or stack to a different card or stack. Included in this group are the *go* and *find* commands. *Go* makes the destination the current card or stack. *Find* searches the current stack for the occurrence of a string. The search may be further restricted to a specific field within a card. If the string is found, the card containing the string becomes the current card.

Action commands modify the user's information. This category contains the commands *put* and *get* used to store and retrieve information from a container; the *send* command for sending messages to another object; the *open* command to start another application; and the *doMenu*, *click*, *drag*, and *type* commands to allow the script to input commands into HyperCard as though the user had performed the action. For example, the *click* command at the given location appears exactly the same as the user clicking the mouse at the same location.

Most math functions are defined as functions in HyperCard. The four arithmetic functions, *add*, *subtract*, *multiply*, and *divide* are also defined as commands. The commands take a source and a container, performs the math function on the two values, and store the results into the container. Alternatively, the *put* command could be used, therefore these commands are shorthand for the longer *put* command. For example, the command *add 5 to it* is equivalent to *put 5 + it into it*.

Object manipulation commands are for accessing and changing information about an object. In most cases, the objects are fields and buttons. Commands in this category include *hide* and *show*, which makes the object invisible or visible, and *get* and *set*, which gets and sets the properties of an object. The properties are further subdivided into global, window, stack, background, card, field, button, and painting, depending on the area of the system affected by the property. The same property name may affect more than one area. For example, the property *name* is applicable to the stack, background, card, field, and button.

Three commands, *visual*, *answer*, and *ask* manipulate the screen, but are not associated with a particular object. *Visual* produces a visual effect on the screen. The effects are defined by HyperCard, and the user is only allowed to choose one from the set, but not define a new effect. *Answer*

and *ask* presents dialog boxes to retrieve information from the user.

HyperCard allows the production of two types of sound, a system beep and sound resources. The resources are digitized from an external source and are played back as recorded, or with a variation of speed and pitch. Varying these parameters plays the sound resource as notes. In this case, the resource is used as the base for the note.

The standard file manipulation commands are provided by HyperCard, which includes *open*, *close*, *read*, and *write*. Only the text file type is supported and the user is responsible for the delimitations between fields and cards.

HyperCard also includes a set of functions that may be grouped into the following categories:

- Time and Date
- Keyboard and Mouse
- Text
- Math
- Miscellaneous

The HyperCard function returns one value, formatted as a string. To return more than one value, a string is built with the values, and separated by commas. Individual values are retrieved using the "item" subdivision commands.

The time and date functions return the value of the internal clock in the Macintosh in several different formats. The smallest time increment available is 1/60th of a second.

The keyboard and mouse functions return the current state of the user interface, such as the current mouse position, the state of the mouse button, the state of the modifier keys, and the location of the last mouse click. These functions allow special interaction between the script and the user.

The text functions provide information about the contents of containers. These functions give the length of a container, find the position of one string inside another string, and find the number of components within a container.

HyperCard also has the standard math functions, such as *abs*, *annuity*, *atan*, *average*, *compound*, *cos*, etc. These functions cover both business and scientific calculation, as shown in the example. Furthermore, the function *the value of <container or expression>* processes the string as a mathematical expression and returns the results. This allows the evaluation of text within a field, useful for teaching.

Several useful functions are grouped under the miscellaneous category. Functions for accessing parameters (described below), retrieving the results of the last *go* or *find* command, and retrieving the ID of the object the received the last message (called *the target*) are in this category.

The target function is similar to the reference to *self* in an object-oriented language. A script defined higher in

the object hierarchy can send messages to the original object that received the message using this function. *The target* and the command *pass* are similar to the reference of *self* and *super* in Smalltalk.

The values of parameters are accessed by two possible methods. First, parameters can be declared as part of the function or command definition. The parameters are bound to the given names. If less parameters are passed than are defined, the excess parameter names are bound to the empty string. If more parameters are passed, the excess parameters are only accessible by the following method.

The second method for accessing parameters are the three functions, *the param of <parameter number>*, *the paramcount*, and *the params*. The function *the param of <parameter number>* returns the value of the parameter at the given index. *The paramcount* returns the total number of parameters passed into the script and *the params* returns the list of the parameters. Any parameter passed to a function or command are accessible by these functions, including the named parameters.

For example, observe the following definition for *foo*:

```
on foo name, date
    :
end foo
```

The named parameters to *foo* are *name* and *date*. The statement *foo Cari* binds *name* to Cari and *date* to the empty string. On the other hand, the statement *foo Cari, "8/22/88", bar* binds

name to *Cari*, *date* to *"8/22/88,"* and *bar* is only retrieved by the statement *the param of 3*.

Parameters in HyperCard are flexible, but debugging the scripts is more difficult than other programming languages. Required parameters must be tested individually to insure they contain values. The programmer may forget to test a parameter and create bugs that are hard to trace. An improvement is a command, such as *require <parameters>*, that automatically test each parameter for a non-nil value.

In general, HyperCard is a good environment for building user interfaces and information processing packages. HyperCard fits in well with the Macintosh's operating system and event handling, although it does not use the Macintosh's standard interface At least two problems exist with HyperCard, the difficulty in debugging caused by using the empty string by default, instead of reporting an error (see above), and the scripts are interpreted, which slows the system.

2.3 Other Works

Two papers, Building User Interfaces by Direct Manipulation (Cardelli, 1987) and A Substrate for Object-Oriented Interface Design (Smith, et al, 1987) influenced the design of MicroWorld. A brief description of each paper is given below.

2.3.1 Building User Interfaces by Direct Manipulation

Building User Interfaces by Direct Manipulation describes a system for building the user interface's dialog boxes separate from the application. A set of interaction primitives, called *interactors*, is provided by the system. An editor is used to assemble them into a dialog box.

The editor is similar to HyperCard and MacDraw. The interactors are placed and positioned in the dialog editor similar to HyperCard's method for buttons and fields. Moving the sides of the bounding rectangle resizes the interactors, similar to MacDraw's resizing of objects.

Interactors can be grouped together on the screen or as logical units, called *interactor groups*. The interactors of a screen group are moved and resized as a single unit, same as object groups in MacDraw. The interactors in a logical group work together to perform a function.

For example, a radio button group is a collection of independently operated on-off buttons. Only one button is "on" at a time, so the previous "on" button must be turned "off," when a new button is selected. A radio button is a logical group, since they only have one function, namely to select one item.

New interactors may be composed of other interactors, or created entirely from scratch. Building a new interactor requires programming in Modula2+. The interactor belongs to

the *Interactor* class, and has the following four methods:

- *GetProperty*: gets a property value of the interactor.
- *SetProperty*: sets a property value.
- *GetDescription*: provides a standard data structure to describe the interactor.
- *SetDescription*: conforms the interactor to the provided description.

Interactors have several properties. Some attributes are apparent to the user, such as the look of the interactor and its "reactivity" to events. Other attributes are internal, such as the interactor's type and the event produced when activated (see below). Individual properties can be changed with the editor without using the *SetProperty:* method.

The appearance of the interactors is specified by several data structures, called *looks*. A *look* is used for each state of activation. This information includes the look of the frame, the alignment of the interactor with respects to the dialog, and the margins.

Looks do not contain size information. The *look* must adapt to the geometry of the interactor. For example, a *look* may specify color of a border, but the size of the border is set and changed by the size and resizing of the interactor.

In general, *looks* are composed of an image and a frame. The image is either a pattern, a bitmap, or a text field. The frame has a background, a border, and an overlay. The overlay is used for translucence effects.

Five levels of activation exists for each interactor. The meaning for the activation levels are:

Active: Functions normally.
ReadOnly: Functions, but not modifiable.
Protected: Needs special action to become active.
Passive: Does not function.
Dormant: Does not function, and looks "dim."

The interactor may have a different look for each activation level, or the same look for several levels.

When an interactor is activated, such as clicking a button, an event is posted. The event's name is contained within an attribute of the interactor and is set by the user.

A procedure is registered with the dialog for each type of event by the application. When an event occurs, the procedure associated with the event is called with a single user-defined parameter and four standard parameters: the dialog, event name, event value, and event time. The same information is passed for each type of event.

Several interactors are provided by the system, including buttons, pulldown menus, scroll bars, browsers, fatbits, and text. A browser is used to select one or more items from a variable-sized set. Fatbits presents a magnified view of a bitmap. It can be used in a graphic editor. The other interactors are typical user interface objects.

This system is intended for a different class of user than in MicroWorld, which is designed for people with little

programming experience to build graphical-oriented lessons. This system is designed as a tool for programmers to easily build the user interface portion of the project. The internals are still written in a standard programming language, in this case, Modula2+. Still, it provides a model for using objects in the user interface.

2.3.2 A Substrate for Object-Oriented Interface Design

A Substrate for Object-Oriented Interface Design describes the GROW (GRaphical Object Workbench) system, that supports the development of graphical interfaces. As with the system describe in Building User Interfaces by Direct Manipulation, GROW separates the interface from the application. The interface built for one application is modifiable and reusable in other applications.

GROW is built upon a set of interrelated graphical objects, arranged in a hierarchical class system. Each object contains the attributes and methods necessary for interaction with other objects, displaying the object within a window, moving the object, etc. Multiple inheritance is allowed, therefore, an object may share common attributes and methods of several objects.

Graphical objects can be grouped into a single, complex object. Operations performed on a complex object are applied to all of its components. Furthermore, a composite object

may consist of other composite objects.

Composite objects are handled by slots containing the name of the components. The components also have a slot that points back to the composite object. When an instance of a composite object is created, the components are created recursively, and these links are established.

The first step in creating a graphical object is to define the class for the object. The programmer is prompted for the name, its superclass, and its composite relationship to the other objects in the system. Objects provided by the system are not used directly.

The next step is to define the graphical dependency. The attribute for one object may depend on the values of other attributes, both within the object and from other objects. For example, the size of a name box is dependent both on the size of the font and on the number of characters within the name.

When an attribute is updated, all dependent attributes are updated. A simple and efficient updating algorithm is used to propagate the changes throughout the dependencies. This algorithm only allows an acyclic dependency graph.

Each attribute contains a list of the other attributes on which it depends. If the other attributes are in other objects, then a path is given from the first object to the other object along with the needed attribute. For example,

if the needed attribute is *DisplayRegion* of the *ModuleBox* of the object's *Module*, then the path is: *(DisplayRegion of (ModuleBox of (My Module)))*.

The attribute's dependencies can be declared either in the class or in an instance of the object. If declared in the class, then all instances of the class have the same dependency.

Creating an instance of a graphical object involves four steps. First, the object itself is created. If the object is a composite, then the components are recursively created and the component links are set. Second, the dependency links are constructed. Third, the display attributes are computed. The dependency links are traversed for each attribute needed for the display. Four, the object is displayed.

As stated earlier, the interface and application are separate. The GROW environment responds to high-level messages from the application. Each object is created with an application-defined key, which the application uses to identify the object. The key and graphical instance are saved in an association table.

Both the application and the interface have access to the association table. Messages are sent from the application to the instances associated with the key. The key can be used as an argument in calls from the interface to

the application. The functions to retrieve instances and send messages via a key are provided by GROW.

The separation of application from the interface allows the easy reuse and modification of the user's interface. The application only knowledge of the interface is the names and keys of the objects. This separation allows the use of interface components in different applications, including applications written in other languages.

The goal of GROW is similar to the project outlined in Building User Interfaces by Direct Manipulation, which is to provide interface building tools for programmers. Although MicroWorld's goal is different, useful information was provided by studying GROW.

MicroWorld was influenced the most by Programming by Rehearsal, which was the starting place for MicroWorld. The papers Building User Interfaces by Direct Manipulation and A Substrate for Object-Oriented Interface Design served more as a way to conceptualize the use of objects in a user interface, than as a blueprint for MicroWorld. HyperCard was introduced too late to have a direct influence in our design. Chapter 4 gives a comparison between MicroWorld, Rehearsal World, and HyperCard.

CHAPTER 3

MICROWORLD

Computer systems are increasingly more powerful as new designs in hardware are implemented. Today, desktop computers provide more power than some mainframes built a few years ago. Unfortunately, software development has not kept pace with hardware development, especially in the area of computer-aided instruction (CAI).

In most cases, CAI systems are built from one of the standard programming languages, such as C or Pascal. These languages are designed for general purpose programming and are not well suited for building CAI systems. As a result, the program is designed to teach one particular area and is not flexible enough to teach a different area. For example, a CAI system to teach math is not easily modified to teach reading.

Programming by Rehearsal is one of the first attempts to provide an environment for the development of CAI systems. It provided a set of classes, called Troupes, to create object used in the lesson. These classes were specialized for certain functions (see Section 2.1).

MicroWorld uses an approach similar to Programming by Rehearsal's approach, but contains several modifications, which improves the development of CAI systems. Both systems, use the prototype model instead of a class hierarchy for their interface objects. In Programming by Rehearsal, each object is created from one of the Troupes defined in the environment. All of MicroWorld's objects are created from the Thing class, which is more generic and flexible than the objects created from Programming by Rehearsal's Troupes. The instructor programs the object's responses to events, thus the instructor has complete control over the lesson.

Although MicroWorld's focus is a second language tutoring system, the design allows the creation of several different types of lessons. MicroWorld provides the tools for producing lessons involving manipulation of graphical objects on a screen. Future design plans include handling text and numeric representations.

3.1 Things, the World, and the Tutor

A world metaphor is used by MicroWorld. A lesson consists of a *World* and the objects contained within it. The *World* provides a structure for objects to exist, similar to the Earth providing a structure for humans to exist. A familiar metaphor helps the instructor work with MicroWorld.

Each *World* has one or more backgrounds, which defines the lesson's setting. The background can change as the student progresses through the lesson. For example, a lesson can begin on a city street, with an appropriate city background. The student enters a store (possibly by clicking the mouse on the door) and the background changes to the interior of the store.

Objects, both visible and invisible, reside in the world. The base object is called a *Thing* and is similar to, but not restricted to, a physical object in the real world. For example, some physical objects are rocks, computers, cars, men, and women. These objects can be represented in MicroWorld.

Associated with each *Thing* is a graphical image and programmed responses to events. The students uses the graphical image to manipulate the object during lessons. No restriction is placed on the size or shape of the image, although, an image larger than the world is not entirely shown.

When an user-created event occurs, the *Thing's* programmed response to the event is performed. The set of programmed responses defines the behavior of the *Thing*.

Several events are possible. A *click* event is sent to the *Thing* under the mouse cursor when the user presses and releases the mouse button. Clicking twice rapidly sends a *double click* message to the *Thing*. Holding the mouse button

down and moving the mouse moves the *Thing's* image on the screen. During the move, the *Thing* receives *dragging* events. When the mouse button is released after a *dragging* event, the *Thing* is sent a *landing* event.

The responses are programmable by the instructor and are unique to each Thing. For example, the response to a "click" event for one Thing is to say its name, while the response for another Thing is to "flash" its image. Unique responses provides a more flexible environment for the instructor in designing a lesson.

Several other features uniquely define a Thing. A movement path can be defined for each Thing. The path can be followed in response to an event, a message from another Thing, or a message from the Tutor. The path is unique to the Thing.

The recording of history and the evaluation of performance are also unique to each Thing and programmable by the instructor. For example, the task of a lesson is to move Things into a landmark Thing. Part of the history kept by the landmark could be the names of the Things that were moved and landed correctly. Performance evaluation could be based on whether the Thing was suppose to move and if it was moved to the landmark. Programmable history and performance evaluation creates a more flexible environment for the instructor and a larger variety of lessons is possible.

Some objects do not have a physical form and reside in an aether associated with the world. Currently, the only object that exists in the aether is the *Tutor*, which controls the lesson.

The *Tutor* acts as a supreme being over the *World*. It gives a task for the student to perform, evaluates the performance, and decides on the next task. Some possible tasks are: repeat the previous task, teach a similar concept, teach a different concept, or quit the lesson.

The *Tutor* communicates with *Things* by sending messages using the *Thing's* name. The *Tutor* may ask the *Thing* to set some instance variables, or to provide information. For example, the *Tutor* could ask the *Thing* to become stationary (i.e. can not be dragged by the user) or perform an evaluation of the student's performance of a task.

Things relate to other *Things* in two respects, through communications and spatial relations on the screen. A *Thing* can ask another Thing by name to perform a command or function. The more difficult relationship between Things is their spatial relations on the screen.

3.1.1 Spatial Relations Between Things

Determining the spatial relations between Things is a more difficult problem than it appears at first glance. One way to determine if a Thing is to the left of another Thing

Figure 6. First Spatial Relationship Test

is to compare the two Things' positions and the distance separating them. This calculation is easy, but does not match how humans perceive spatial relations.

When humans are given a choice, why do we say that an object is to the left of another object instead of above, for example? In several cases the choice is easy, but other cases are not so clear cut. In order to have a good system for building graphical interaction applications, the choices that the system makes must match the choices a human would make.

The difficulties involved were first noticed when we presented the diagram in Figure 6 to a small number of subjects. Several questions were asked about the spatial relations between the objects. For example, one of the question was, "Which block is above C?" The typical answer

was B, although both A and B are above C.

The interesting section of this diagram contains the squares C, D, and E. When asked which square was to the left of the square E the answer was C, with only one exception. If the simple method of comparing the horizontal position and the separation distance was used, then the answer would be D. Therefore, the simple method does not determine what we, as humans, consider to be the correct answer.

This observation lead to the corridor theory, which states that prototypical locations exist for an object based on its dimensions. Another object's position is compared to the prototypical locations to determine the spatial relationship with the first object.

In Figure 7, A is the position considered the prototypical position for an object to the left of the square and C is the prototypical position for above the square. The line at B is the point of the most ambiguity between left and above. As an object's position moves from A to B and then to C, the description of the object's location changes from being left to being above. It is called the corridor theory because the horizontal and vertical lines makes corridors for the prototypical locations.

When an object is near the line at B in Figure 7, most people use both directions to describe the location of the object. In other words, a person will describe a circle

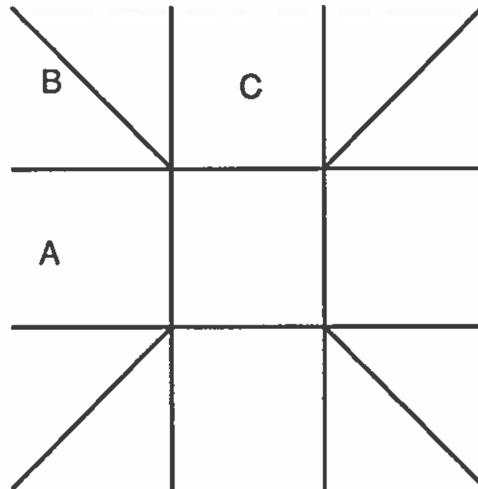

Figure 7. Describing Positions Around an Object

(assuming a circle exists at line B) as being above and to the left of the square. An interesting study is whether a person describes a location using the spatial relation closer to the prototypical position first, or just by a random selection. In other words, if an object is between A and B, would a person describe it as being to the left and above the square, and if it was between B and C, describe it as being above and to the left?

As to the question of why people chose the rectangle B over square A when asked which object was above the square C, (Figure 6) two factors may play a role. First, the rectangle B is larger than the square A. Because of size, rectangle B may be more prominent in the mind of the observer, and thus its selection over square A. Second, the square A is on top

Figure 8. Global Relationship Between Objects

and contained in rectangle B, which is a three-dimensional relationship. Since the objects are on a two dimensional plane, some people may rule out that possibility. Other issues may also be involved.

One problem with the corridor method is its locality in terms of describing spatial relations between objects. An experiment, which illustrates the difficulty, is shown in Figure 8.

The task was to describe the location of the black square next to the white square. The question, "Describe the location of this square," was asked of each subject. Based on the corridor theory and the fact that each object has a unique feature (i.e. only one circle and one white object exists), the predicted response was either, "the square to the right of the (black) circle" or "the square to the left of the white square." Instead, the answers were in the form, "the second square from the left (or right)" or "the square between the (black) circle and the white square."

The corridor method uses a very local relationship between objects. The expected answers were derived from

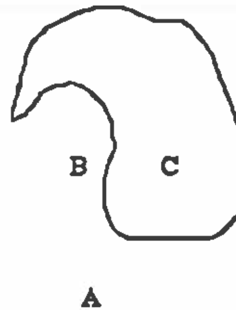

Figure 9. Position with Respects to Irregular-Shaped Objects

taking the objects in the most prototypical positions and using those as the point of reference. In the diagram (Figure 8), the objects are in line. The observer groups these objects together, and thus creates a frame of reference for that group. The corridor method does not handle this situation.

Note the response of, "the square between the black circle and the white square." The possible reason for this response is based on the question. If the query was less formal, such as, "tell me…", instead of "describe the…", then less information may have been provided. The more formal query implies a more informative answer. A language tutoring system needs to take this into consideration.

The second problem with the corridor theory is with irregular shapes. Figure 9 shows an example of the problem. An object at point C is described as being above the object A. On the other hand, an object at point B is described as

being to the left of object C, although, object B is in the vertical corridor more than the horizontal corridor. These two problems lead to the centroid theory, which was combined with the corridor method to form the present method for determining spatial relations.

A centroid is the center point of the image. The calculation is based on a modification of the center of mass calculation. The object is consider to have uniform density, since the object is on a two dimensional plane. The centroid is used to determine where the object is in the corridor. For example, in Figure 9 the centroid of object C is in the right corridor of the object B.

Another observed phenomena is the breaking of an image into segments. For example, the standard description of an object at location A in Figure 10 is a variation of "the object is below the horizontal bar and to the right of the left vertical bar." It appears that the object is segmented into three sub-objects, and the description is based on the segments.

When an object has a more complex image, another problem arises. If the object has a perceived notion of a front and back, such as a man or woman, then determining the objects to the left or right is more difficult. For example, in Figure 11 (from Chapter 1), the rope is to the woman's left, based on their positions on a two dimensional plane. Since the

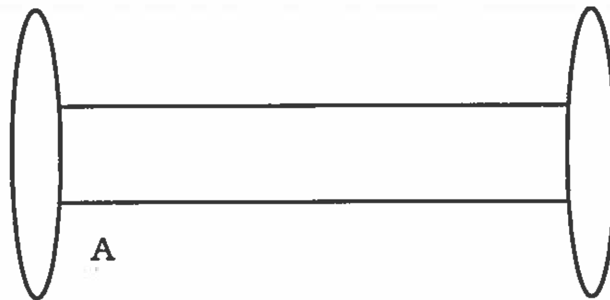

Figure 10. Object with Components

Figure 11. Lifeboat

woman is facing the viewer, the object is also described as being to the woman's right. In order to determine the correct spatial relationship, this case must be handled.

The algorithm to determine if an object is in a spatial relationship with another object (for example, "Is the man to the right of the woman?") is not difficult. The centroid of the objects and the position of the first object's centroid in the corridor of the second object are compared. If the object is closest to the prototypical location of the spatial relationship in question, then the answer is yes.

Determining a prototypical location to place an object into a spatial relation is also relatively easy. MicroWorld aligns the centroids in the direction of interest. For example, if "left" is the desired spatial relation, then the object is placed to the left, with the two object's centroid on the same vertical line. The two objects are separated by a distance equal to one-tenth of the larger object, or by a minimum of 10 spaces. This distance was defined arbitrarily and more study is needed to determine a more accurate separation value.

On the other hand, determining which object is the closest to the prototypical location is very difficult. Two indicators need consideration. One indicator is the relationship of the centroids. An object whose centroid is in line with the comparison object's centroid on the axis of interest (either horizontal or vertical) is closer to the prototypical location than an object not on that line. The second indicator is the distance between the objects. If two

objects are on the same line, then the closer object is nearer the prototypical location.

The difficulty is determining the relationship between these two indicators. If one object is not as close in one indicator, but is closer in the other, which object is in the more prototypical location? Does one indicator have a higher priority than the other indicator? If one indicator does have a higher priority, then how close does the other indicator have to be before it becomes the deciding factor? Is this relationship concrete, or does it vary between the viewers of the objects and their locations?

It appears that the relationship is subjective, varying from person to person. We have no way of measuring the relationship either, so designing an algorithm to perform correctly appears difficult, if not impossible. MicroWorld makes no attempt at determining which object is closest to the prototypical location. If the user asks for the objects in a certain spatial relationship with another object, a list is returned of all the objects that fit the relationship.

Dividing an object into segments is also a difficult task. A simple object, like the one in Figure 10, is relatively easy. The more complex an object becomes, the more difficult the task. Also, the segmentation may be along functional units and not geometrical units. MicroWorld has no concept of the items' functions. Again, the determination

is subjective, therefore, no algorithm was designed to handle this problem.

MicroWorld can determine if an object is in a certain spatial relationship with another object, a good prototypical location to put an object, and all objects that are in a certain spatial relationship with another object. Unfortunately, the problems of determining which object is in the most prototypical location and the segmentation of the objects were too great for us to design an algorithm to handle these cases.

3.2 User Modification of MicroWorld's Objects

MicroWorld has two levels of modifications to the environment. In authoring mode, the instructor can create the world; create, modify, and delete Things and Tutors; change the size and position of the world; change the background, and change the lexicon file used to generate sounds. In running mode, the internal state of the Things and the current background are changeable via the instructor's program.

The student can create four different events in MicroWorld. A Thing may be clicked, double clicked, or dragged from one location to another. Clicking and double clicking sends the event directly to the Thing. On the other hand, when a Thing is dragged from one location to another,

two event types are sent, dragging events during the movement and a landing event when the object is released. For example, when objects in the "Lifeboat" lesson (see Chapter 1) receives a click event, it says its name. When an object, such as the rope, receives a landing event, it ask the lifeboat if it's within the lifeboat's boundaries and it either says "The rope is in the lifeboat" or "The rope is not in the lifeboat" depending on the object's position with respect to the lifeboat's position.

Two internal actions, history and evaluation, are also programmable by the instructor. In a tutoring system, recording and evaluating the student actions are important for determining future tasks in the lesson. The responses to events are programmable, therefore, it is difficult to determine the type of information to record and how to evaluate it. A comprehensive history and evaluation mechanism will not handle all possible cases. Our solution is to provide the instructor with these two programmable actions.

Things have a unique program for each action. Changing the program for one Thing does not affect the programming of other Things. This approach allows specialized Things to exist in the world, but it is too time consuming to program several Things to perform the same task. Therefore, two mechanisms are provided to make programming responses easier.

When a Thing is copied, the programs for the Thing's responses, history, and evaluation are also copied. The instructor can make copies of one Thing, change their images and other information, and make a collection of Things with identical responses.

New commands and functions can be created by the instructor, using a set of primitives and any previously defined commands and functions. Instead of several Things using the same response action, one command can be created by the instructor and the action just calls the new command.

Dialogs and pop-up menus are used extensively throughout the programming phase. Only rarely does the instructor use the keyboard. Pop-up menus provide lists of available commands, functions, and items, depending on the position of the mouse. For example, if the mouse points to a field that expects a command, then the pop-up menu contains the list of available commands. Selecting a menu item places the command in that field.

3.3 Instructor's View vs. Student's View

MicroWorld operates in two modes, authoring and running. The instructor works in the authoring mode. The lesson is built by a combination of creating Things in the World, programming the responses, and creating the Tutor.

The student uses the running mode of MicroWorld. The lesson built by the instructor is given to the student to perform. The two modes differ in the available commands and the Thing's responses to events.

All menu commands described in Section 3.5 are available to the instructor. Most commands are used for creation and modifications of the World and Things that it contains. The only command available to the student is *Quit*. This command signals the Tutor that the student wants to quit the lesson prematurely.

In authoring mode, the Thing's responses are programmed by the instructor. When the instructor sends an event to a Thing (for example, clicking it), the programmed response is not performed, unless the instructor is testing the lesson. When the student is taking the lesson, the Thing's programmed responses are performed for the events.

All Things that are declared invisible are still visible to the instructor. This allows the instructor to interact with the Thing. During the lesson, the student does not see either the invisible Things, nor the Aether window. Only the World and visible Things are presented.

This paper is primarily concerned with the authoring mode of MicroWorld. Therefore, the term "user" is equivalent with "instructor." "Student" refers to a person performing a lesson created by the instructor. The student does not have

access to the authoring system.

3.4 Introduction to the MicroWorld Environment

This section walks through a session to create a new world in MicroWorld. The final product is not complete, but this description will give an idea of the steps involved in using MicroWorld. The next section describes the user interface in greater detail.

When the instructor first begins a session, only the *World* menu is activate (see Figure 12) and the available choices are to create a new world, open an existing world, or quit MicroWorld. The text of active menu and menu items are printed in black, and the rest is printed in gray. Other menu and menu items become available when certain conditions are true. For example, the other three menus become available when a world exist within the environment.

Since we are creating a new world, the *New* menu item is selected from the *World* menu. MicroWorld first requests the new world's name with a default setting of "Default World" (see Figure 13). The world will be saved in a file with the same name as appears in the box when the *OK* button is pressed.

Next, the world's current background is set. Figure 14 shows a list of backgrounds defined in the current world. Since the world is newly created, it contains no backgrounds

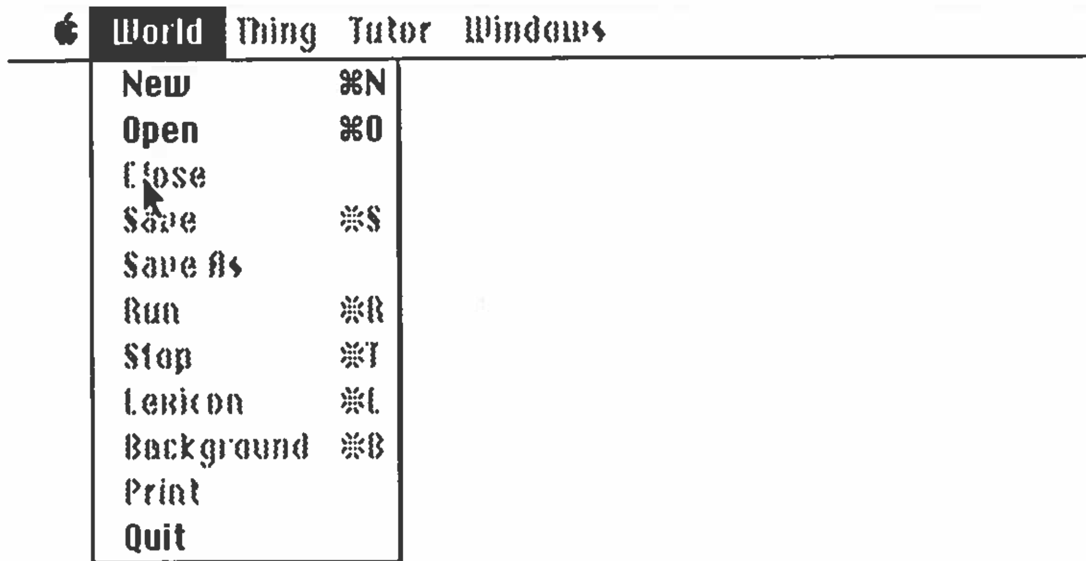

Figure 12. World Menu

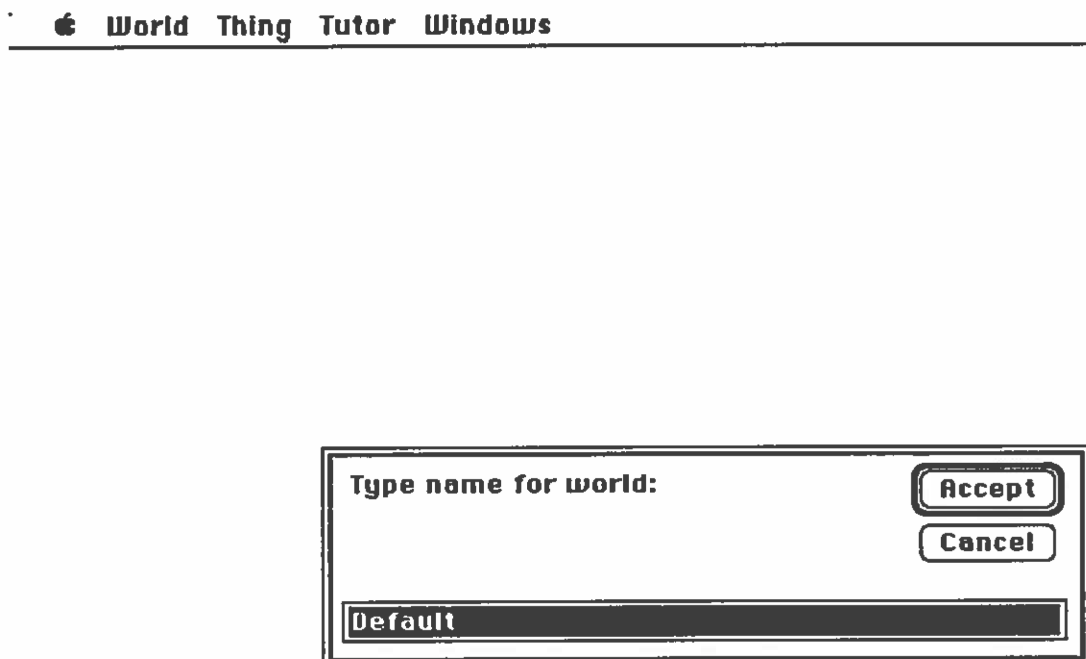

Figure 13. New World's Name

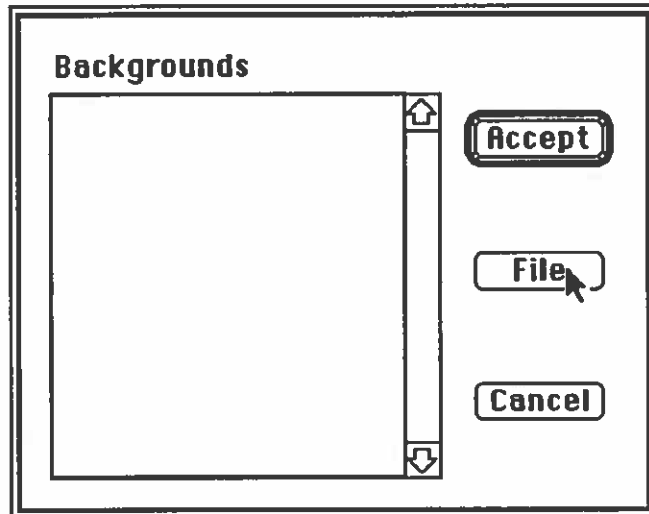

Figure 14. Backgrounds Defined in the New World

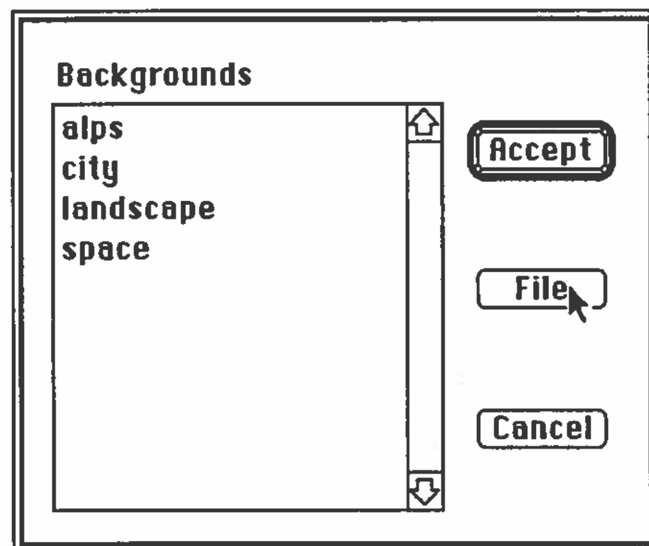

Figure 15. World Contains Backgrounds

and the list is empty (Figure 15 shows a non-empty background list). Selecting the *File* button allows the user to load a background from a file into the current world.

Figure 16 is a file list dialog (provided by the Macintosh Operating System), which shows the available files from which to select the background. Selecting an item (in this case "LBBackground") and pressing the *Open* button loads that background into the new world. Pressing the *Cancel* button exits this dialog without loading a background.

MicroWorld requests the new background's name when the *Open* button is pressed (see Figure 17). The file name is used as the default name for the background. The background's name is added to the list of available backgrounds (see Figure 15) when the user accepts the name (in this case, the default name was accepted). After this step, the bare minimum world is completed and is shown in Figure 18.

The next two major steps are to populate the world with *Things* and assign the world a *Tutor*. These objects can be created in any order, so we will start by creating a new *Thing*.

The only available *Thing* menu item is *Create* (see Figure 19). The other menu items become activate when at least one *Thing* exists in the world. Selecting *Create* begins the process of adding a new *Thing* to the world.

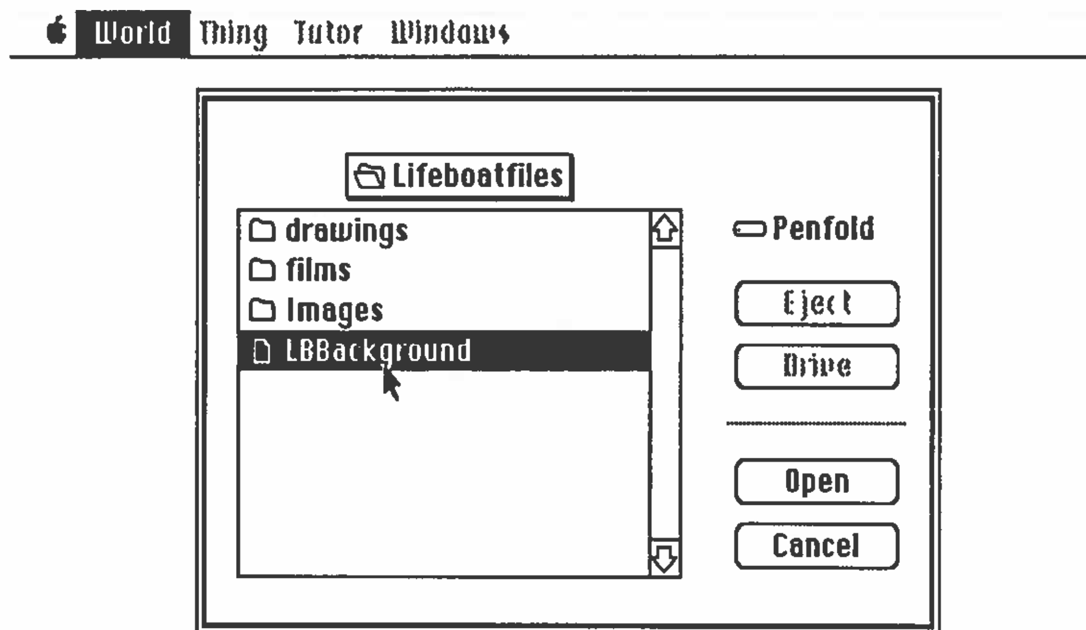

Figure 16. Selecting Background File

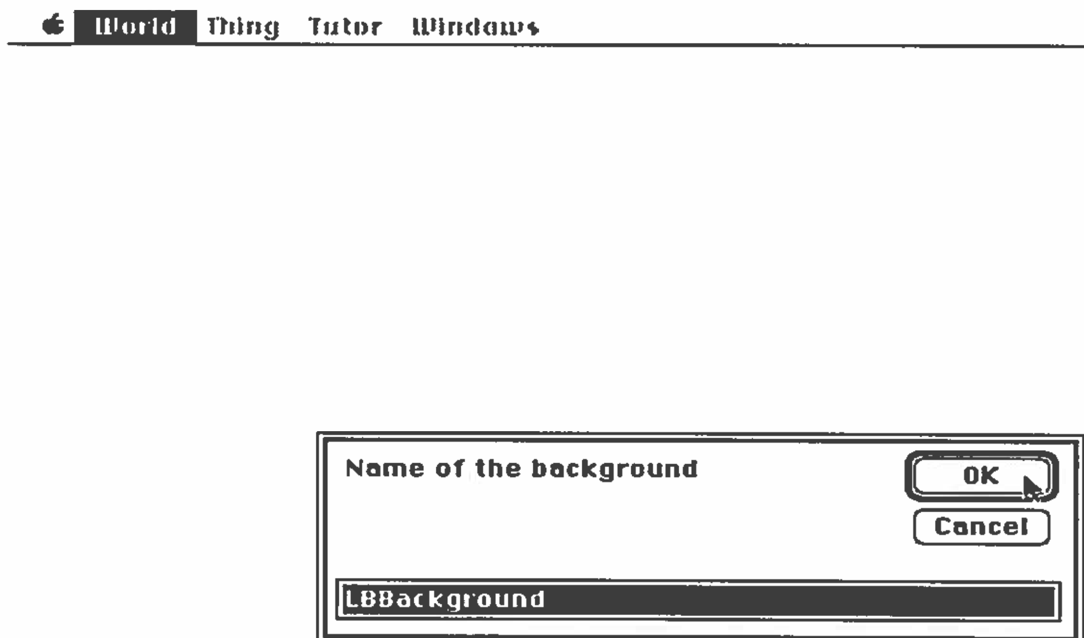

Figure 17. New Background's Name

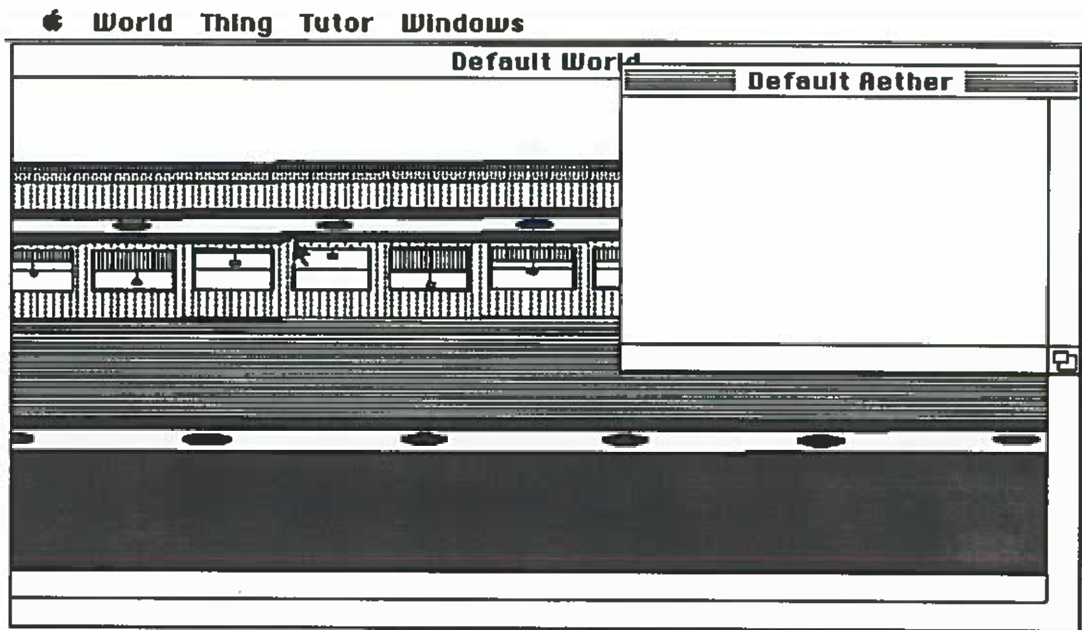

Figure 18. Newly Created World

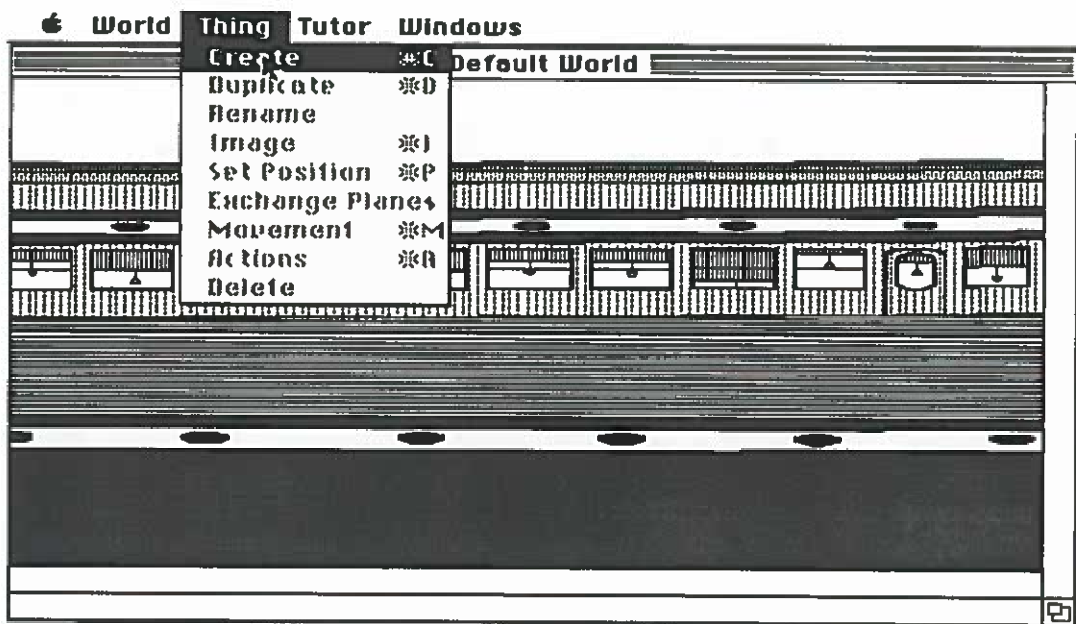

Figure 19. Thing Menu

The next task is to give the new *Thing* a name (Figure 20), which must be unique amongst all *Things* in the world. Information about the *Thing* is saved using its name and the name provides the means for other objects (*Things* and the *Tutor*) to communicate with this *Thing*, therefore the name is checked for duplication before allowing the user to accept it. Pressing the *Cancel* button aborts creating the new *Thing*.

After accepting the name, the new *Thing's* image and mask are defined (Figure 21). The mask gives the graphical object its shape and the image gives the object its color. The scroll bar below the image and mask boxes allows the user to scroll from one picture to the next. Figure 21 shows the image and mask for the lantern.

After accepting the *Thing's* image and mask, its icon is chosen next (Figure 22). The icon provides an alternative way of graphically representing the *Thing*, plus allows the user to defer setting the *Thing's* image and mask until later. Each *Thing* is required to have at least an icon, therefore acceptance is the only choice. Figure 23 shows the newly created *Thing*.

The next step is to program the *Thing's* behavior (although this task can be deferred until any time later). The *Thing* must be selected (as shown in Figure 24 after it was moved to a new location) before its attributes and

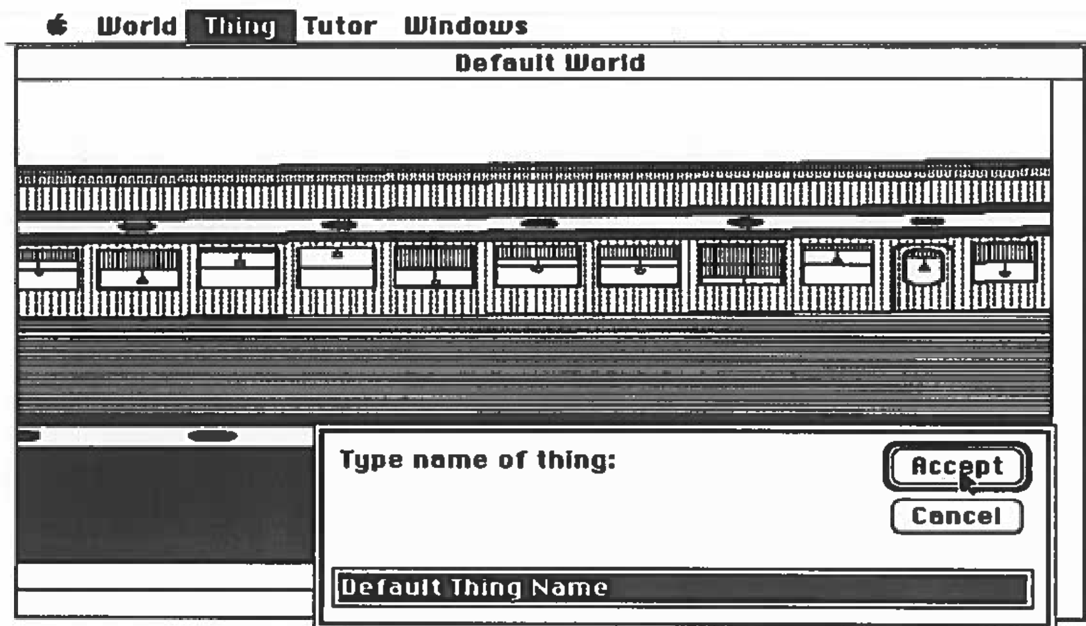

Figure 20. New Thing's Name

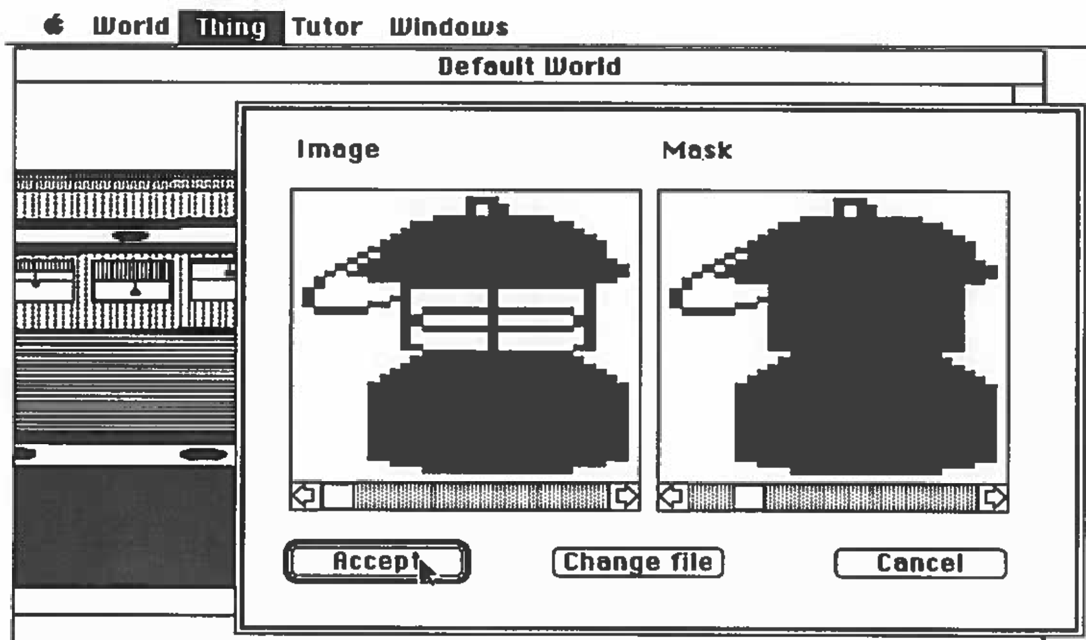

Figure 21. New Thing's Image and Mask

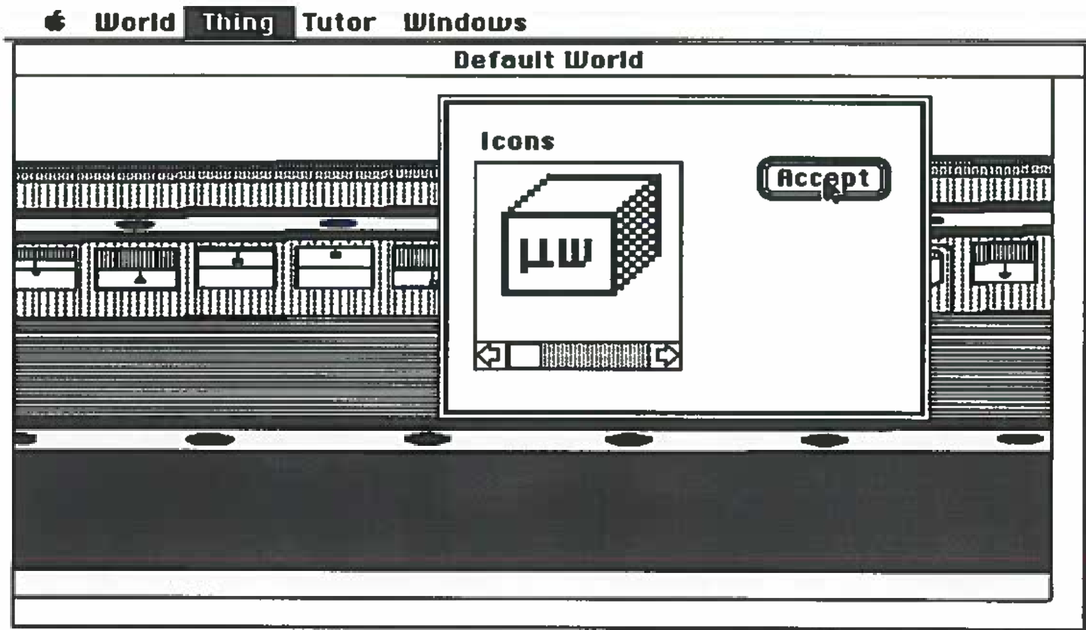

Figure 22. New Thing's Icon

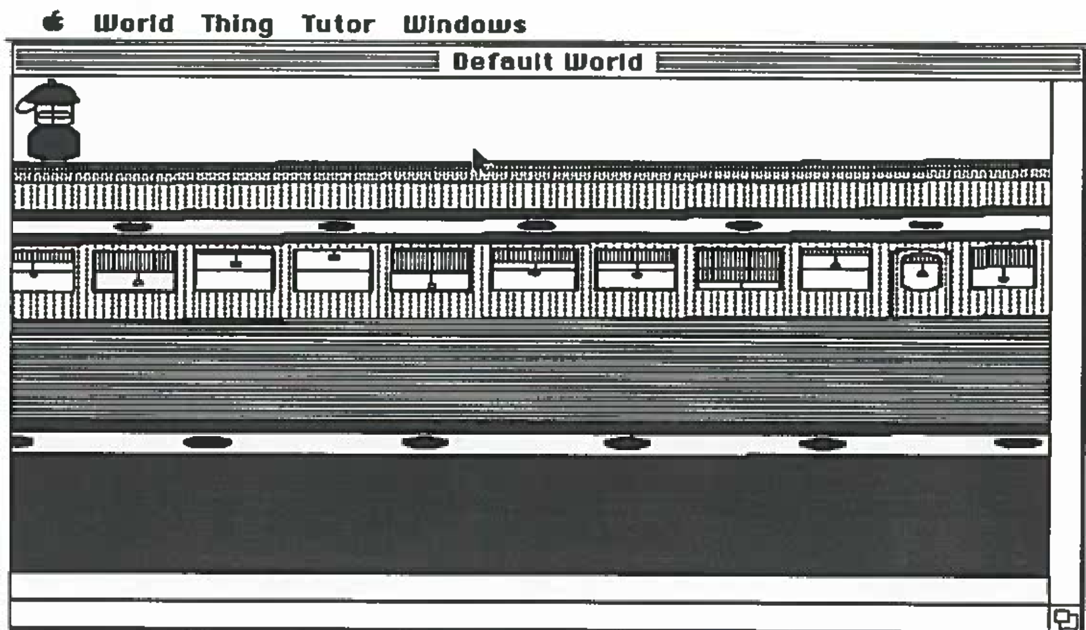

Figure 23. New Thing

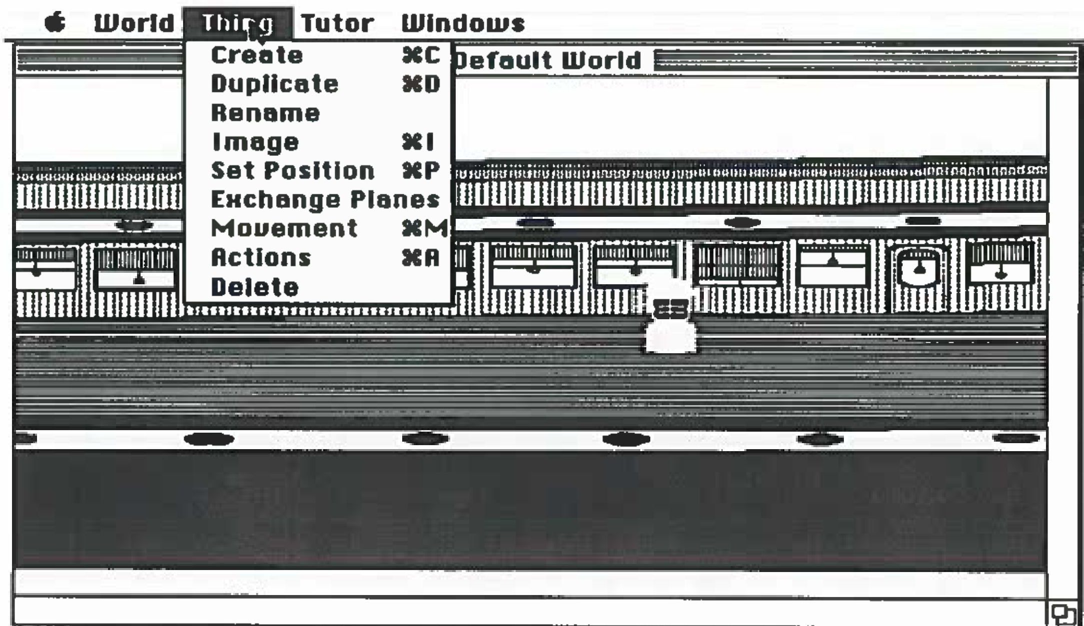

Figure 24. Thing's Menu When a Thing is Selected

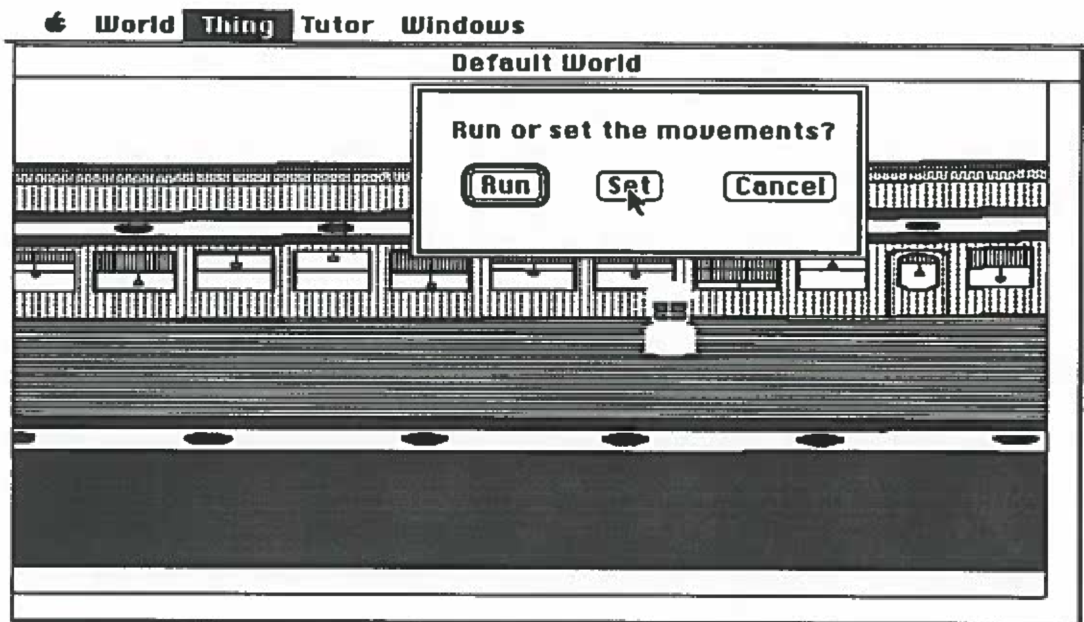

Figure 25. Movement Command

behaviors are accessible. In this example, only the *Movement* and *Actions* menu items are used.

The *Movement* menu item defines the *Thing's* movement path. The user can either watch the *Thing* move through its currently defined path by selecting *Run* or modify its path by selecting *Set* (see Figure 25). Since the new *Thing's* path is undefined, the *Set* button is pressed.

Figure 26 shows the dialog box used for modifying the *Thing's* path. Currently, the path is partially defined and the next position (offset in pixels from the upper left corner) is about to be added to the end of the list using the *Add* button. Each position along the path can be typed in or the path can be defined by following the mouse (using the *Follow mouse* button), which is quicker, but less accurate. Once the path is ready, the *Accept* button changes the *Thing's* path to its new value.

The *Actions* menu item allows the user to modify the *Thing's* attributes and actions. The dialog box in Figure 27 allows the user to select between the *Thing's* attributes, local actions, and global actions. In this example, we will first examine the *Thing's* attributes, then its local action, and, finally, the actions global to all *Things*, although this order is arbitrary.

Figure 28 shows the current *Thing's* attributes. Using the *Replace Value* button, the *animate?* attribute was set to

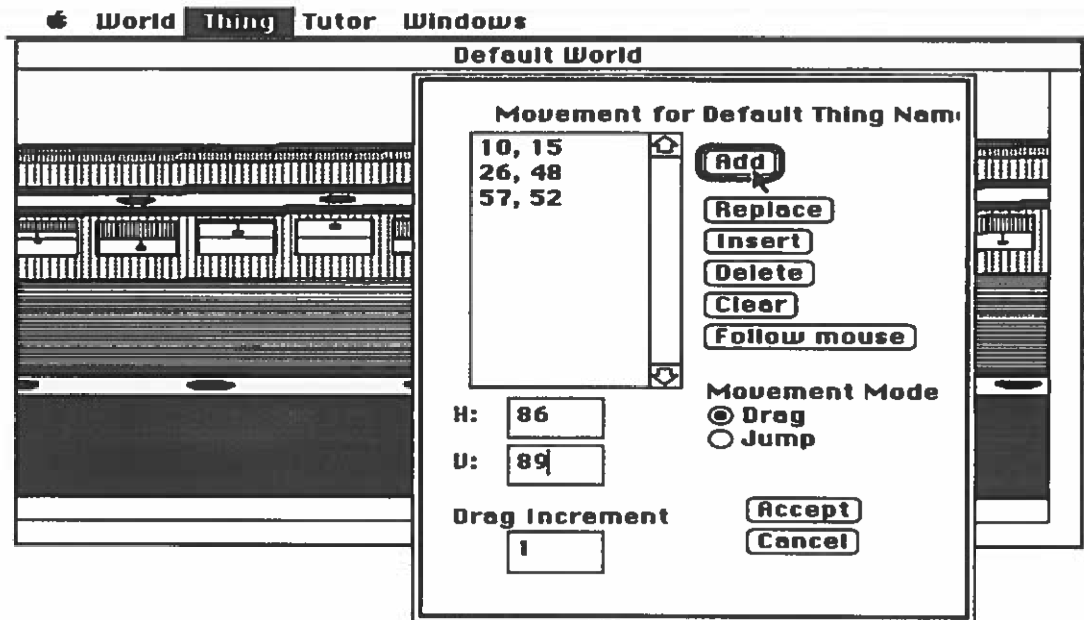

Figure 26. Set Movement Path's Dialog

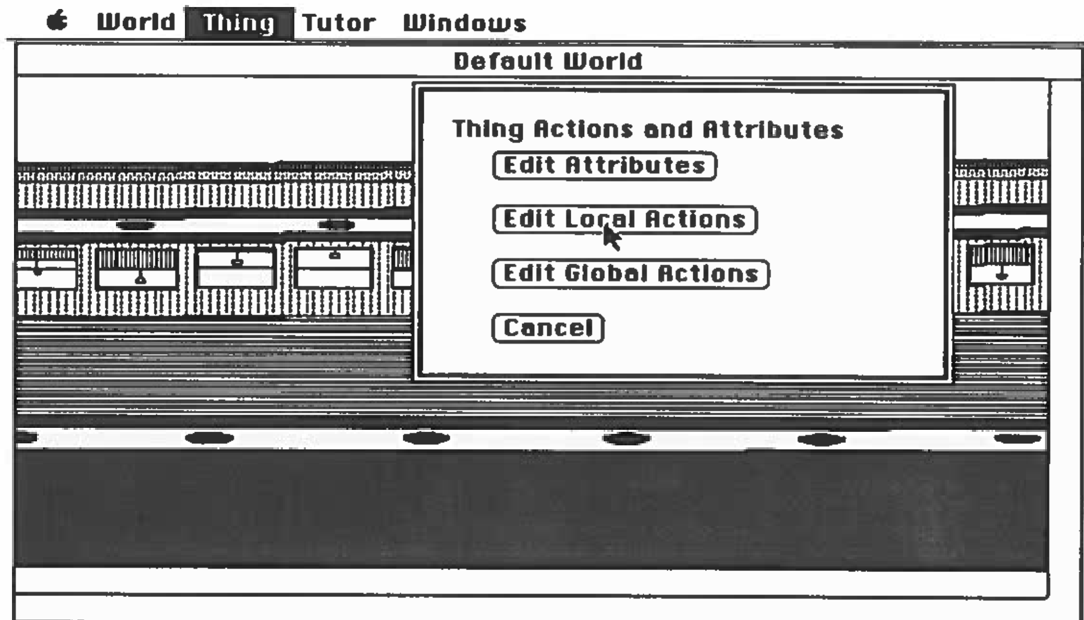

Figure 27. Actions Command

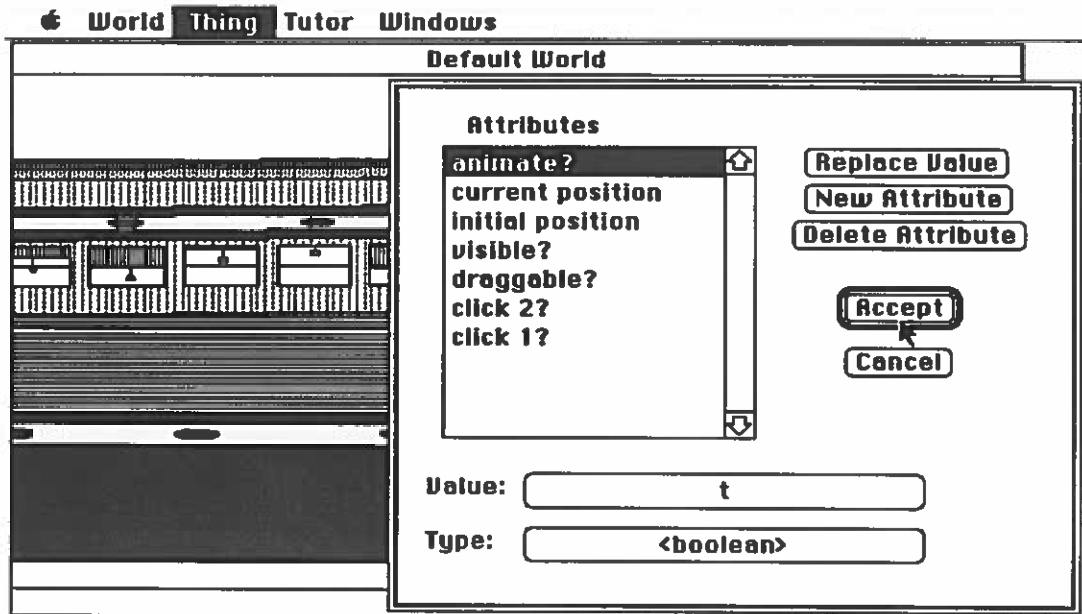

Figure 28. Editing Attributes

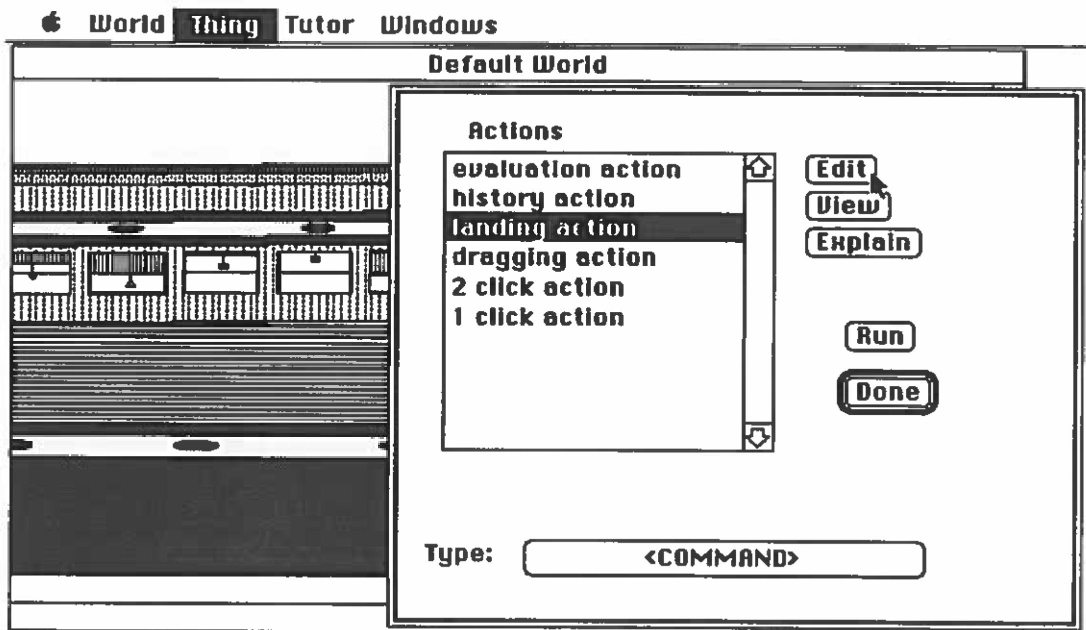

Figure 29. Editing Local Actions

true, which indicates that the *Thing* is an animate object. From this dialog box, new attributes can be added and old ones removed, although built-in attributes are not removable. These attributes are accessible from the local and global actions.

Figure 29 shows the *Thing's* list of local actions. By default, these actions do nothing when called. Therefore, in order to define the *Thing's* behavior, the appropriate action(s) needs editing. In this example, the *landing action*, which the *Thing* performs upon completion of being drag from one location to the next, is the action that will be editing when the *Edit* button is pressed.

The dialog box in Figure 30 contains the outer-most layer definition for this *Thing's* landing action, which is empty since the *Thing* performs no action upon landing. Placing the cursor over the single blank line in the *Code:* box and pressing the mouse button activates the pop-up menu shown in Figure 30. Since this line expects a command, only the commands defined for this world are shown[2]. Selecting the move command and releasing the mouse button places the command in the code.

The move command expects two parameters, therefore another dialog box is created above the current dialog box to set the parameters' values (Figure 31). Both parameters must

---

2. **Actually, the commands shown are a small subset of MicroWorld's commands.**

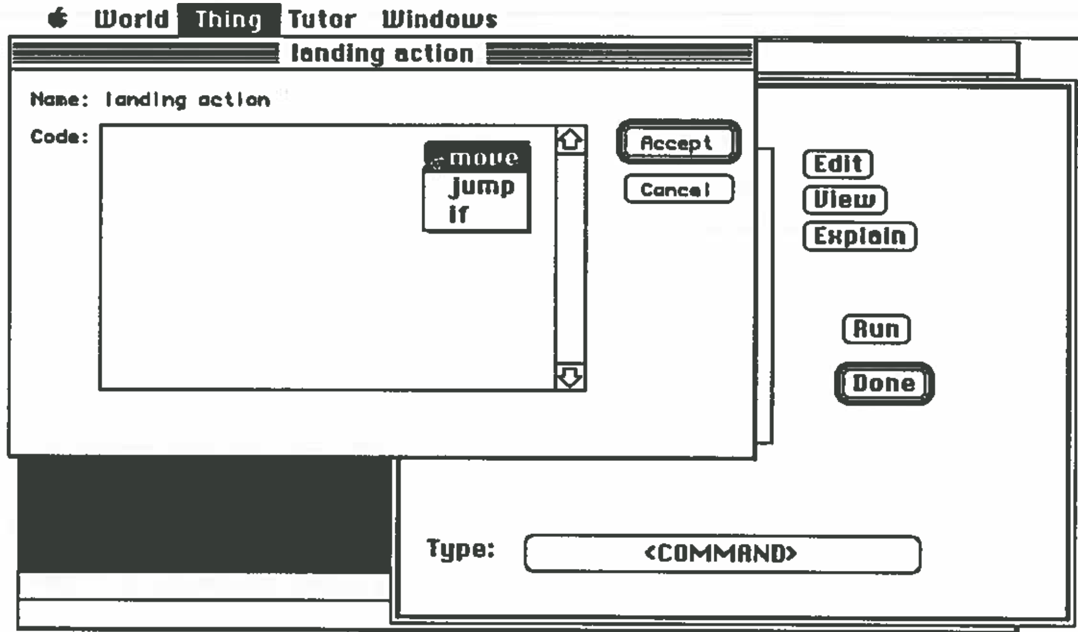

Figure 30. Editing Landing Action

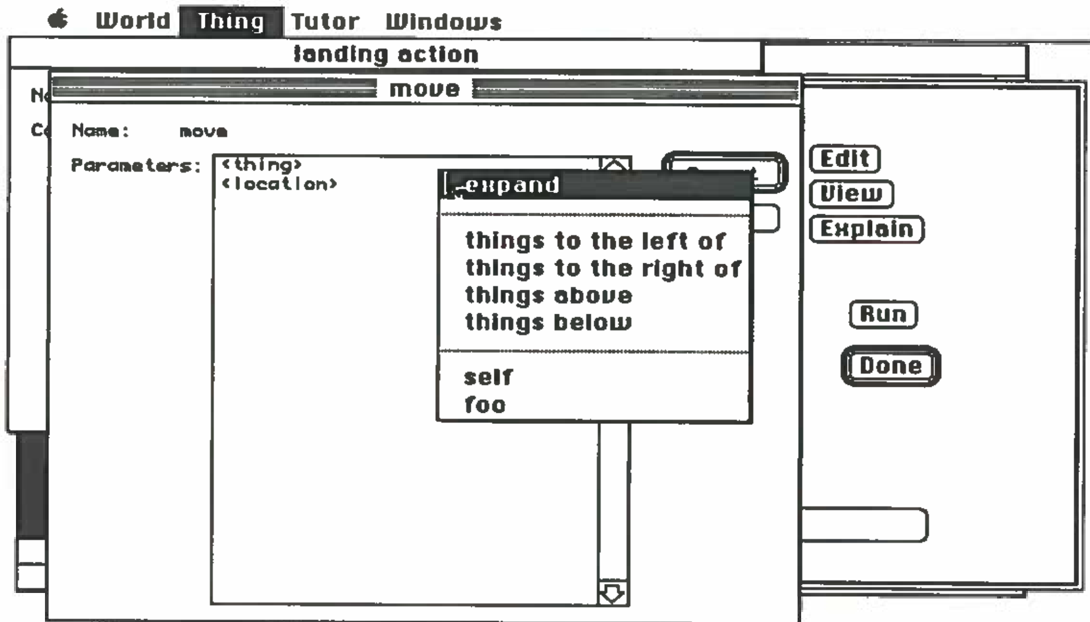

Figure 31. Editing Landing Action's Move Command

be set to appropriate values before the command is accepted. In the place of actual values are the parameters' types (in this case, *<thing>* and *<location>*), which lets the user know the expected values' types.

Pressing the mouse button while the cursor is over one of the parameter activates a pop-up menu that is appropriate to the parameter's type. In Figure 31, the cursor was over the *<thing>* parameter, therefore the menu contains all the functions that return a *Thing* plus the name of all *Things* defined in the world. The *expand* menu item creates another dialog box to show a function call, if the parameter had been set to a *Thing*-returning function. After setting both parameters to appropriate values and pressing the *Accept* button, the previous dialog is activated and updated to show the changes.

Global actions are commands and functions that can be called by any *Thing* defined in the world. Pressing the *Edit Global Actions* button in the dialog box shown in Figure 27 creates the dialog box that shows the list of the available global commands and functions (Figure 32). *Arrange*, which is currently selected, is a user-defined command that was created earlier. Similar to editing a local action, pressing the *Edit* button creates a dialog box that shows the command's definition (Figure 33). Unlike the dialog box for local actions, this dialog box contains an extra field for the

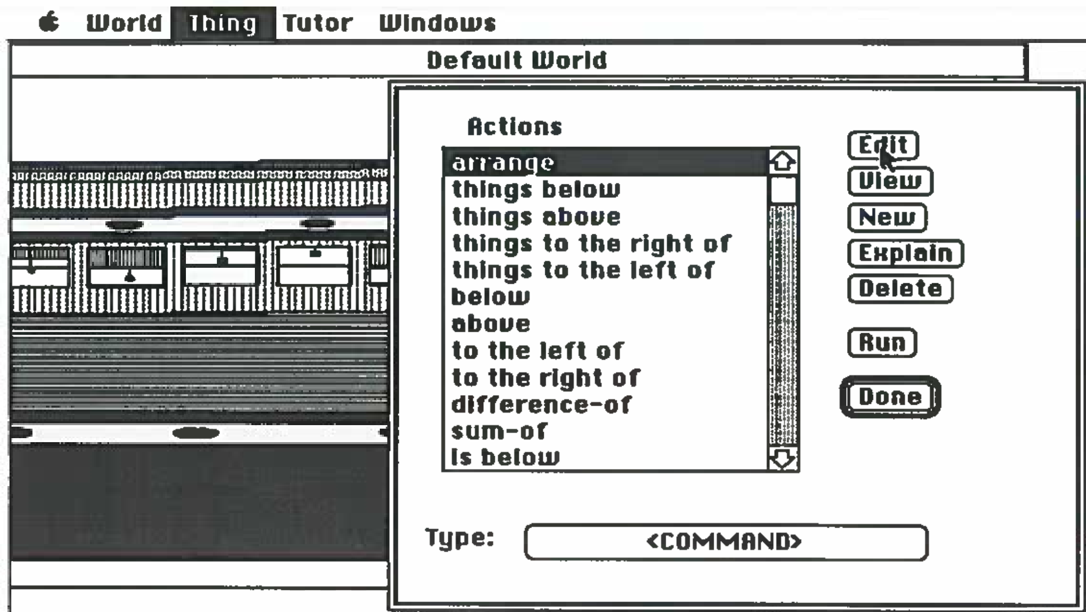

Figure 32. Global Actions

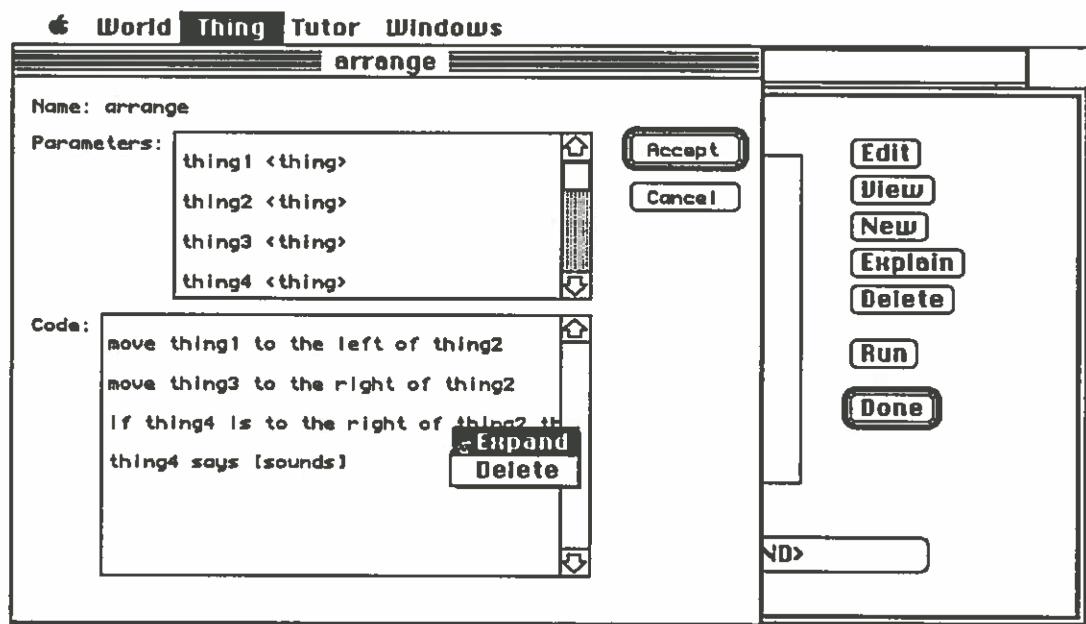

Figure 33. Editing the Arrange Command

parameter list. The *Arrange* command has four parameters of type *Thing*. A blank line exists between each parameter to allow the insertion of a new parameter at any position.

Arrange command's body contains four commands: two move commands, an if statement, and a says command. Expanding the if statement creates the dialog box shown in Figure 34. The user can set the conditional function (first field), the commands to perform if the conditional result is true (second field), and the commands to perform if the conditional result is false (third field). Currently the conditional test is whether *thing4* is to the right of *thing2*. If it is, then *thing4* is moved to the right of *thing1*, otherwise it is moved to the left of *thing3*.

Figure 35 shows the dialog box for editing the list of sounds that *thing4* says during the last command in *Arrange's* code body (Figure 33). Again, blank lines exists between each list item to enable the user to insert a new item at any position. Currently, the *Arrange* command moves two *Things*, moves another based on a test's result, and says "Good morning students" when called by a global or local action. Pressing the *Accept* button adds the command to the command pop-up menu shown in Figure 30.

The same *Arrange* command can also be viewed, which shows the command's definition, but does not allow the user to make modifications (Figure 36). In this mode, blank lines do not

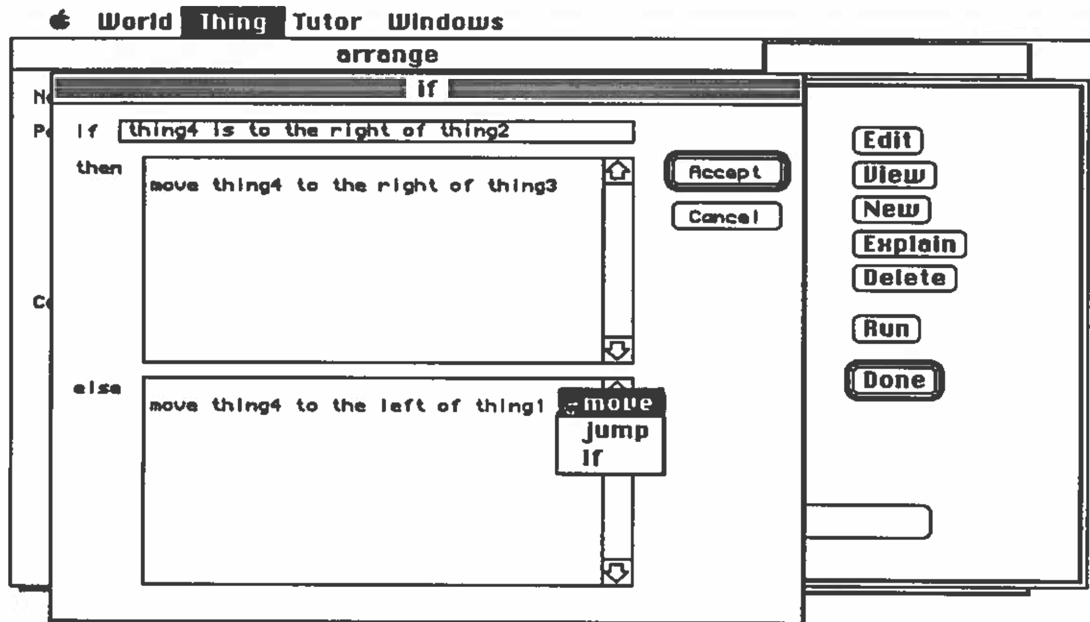

Figure 34. Editing the If Command

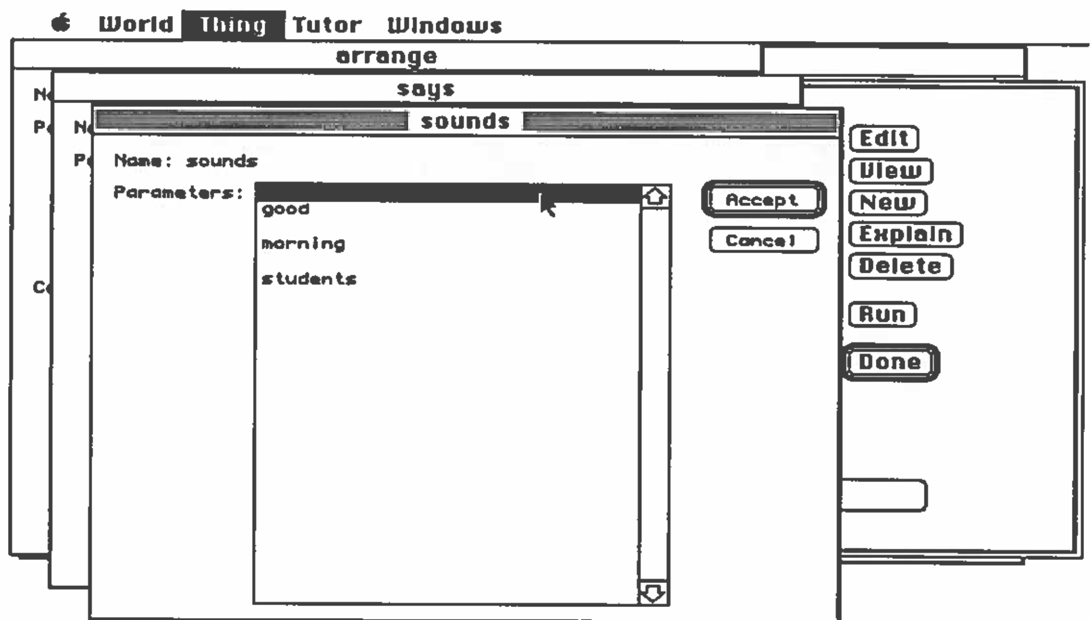

Figure 35. Editing the Sound List

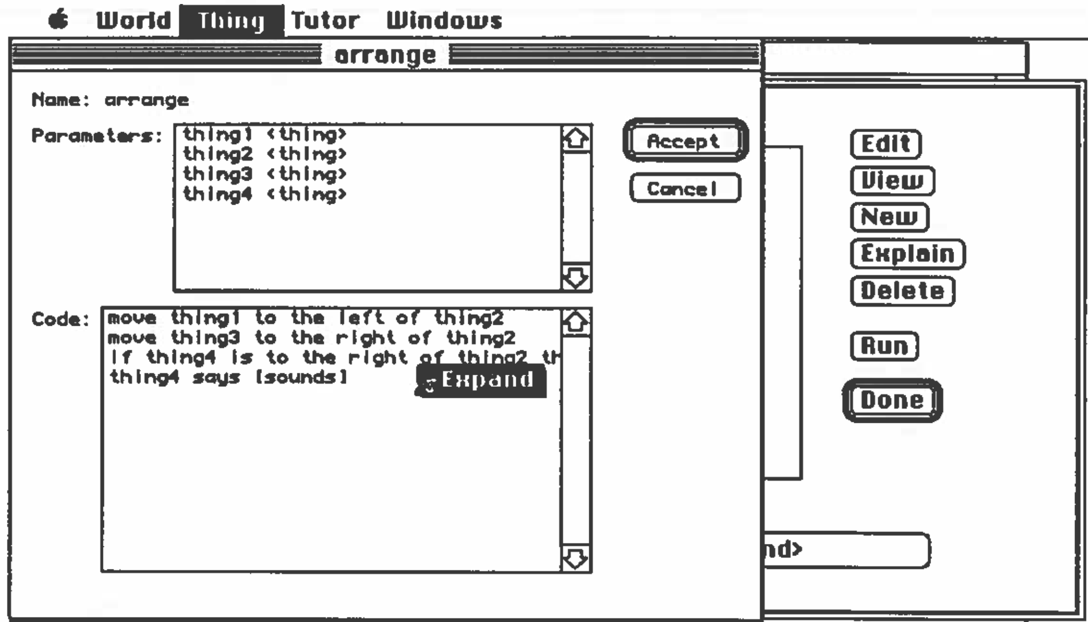

Figure 36. Viewing the Arrange Command

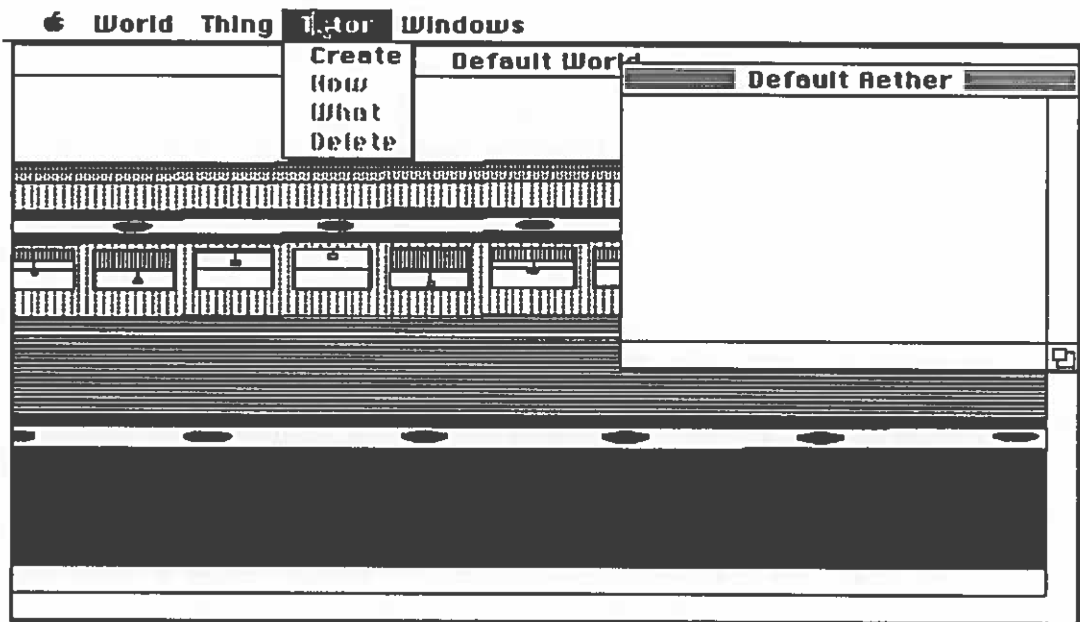

Figure 37. Tutor Menu

exist between each line of code, therefore the user cannot insert new lines. Furthermore, the only command available in the pop-up menu is to expand (for viewing) the selected line. Viewing code protects the user from making accidental changes.

The other major step (besides populating the world with *Things*) is assigning a *Tutor* to the world. A *Tutor* is created using the *Create* menu item under the *Tutor* menu (Figure 37). Since the *Tutor* is an invisible object that resides in the *Aether* window, it does not have a graphical image and is represented by its icon. The *Tutor's* icon is set using the same dialog box for setting the *Thing's* icon (Figure 22).

How and what the *Tutor* teaches are set using the dialog boxes in Figures 38 and 39. In this example world, the *Tutor* teaches by allowing the student to explore the environment, which means that the *Tutor* does not control the lesson, therefore the selection made in Figure 38 does not change the *Tutor's* behavior. Only one *Tutor* can exist in the world, therefore the *Create* menu item is deactivated when the *Tutor* is defined (Figure 39).

The above description gives a general idea of the steps needed for building a lesson, but is incomplete. The following sections give greater details about each menu item, the programming interface, and MicroWorld's programming

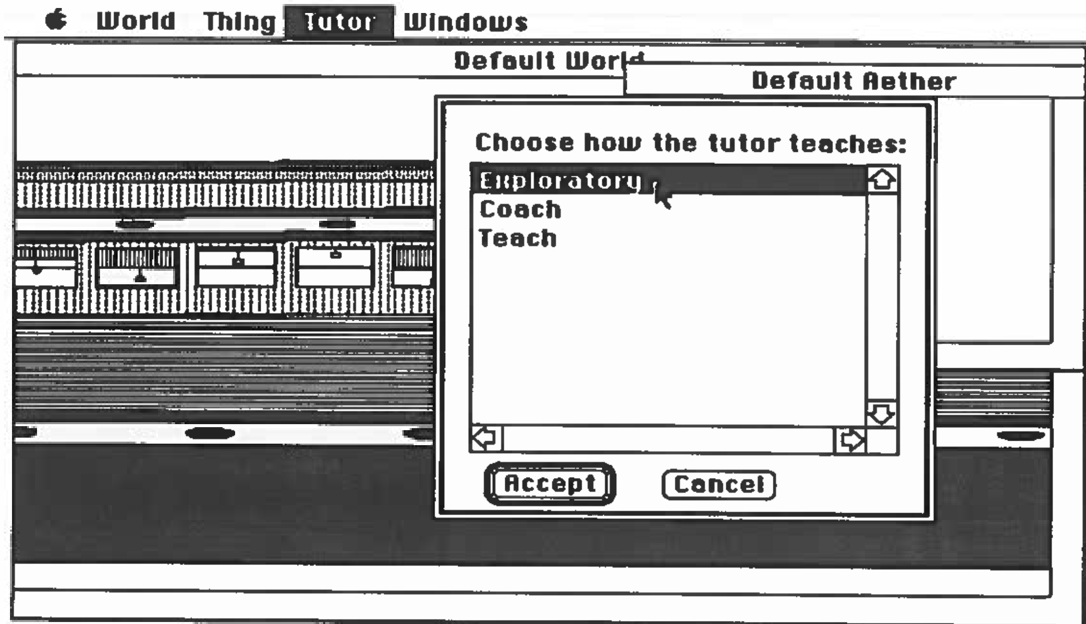

Figure 38. How the Tutor Teaches

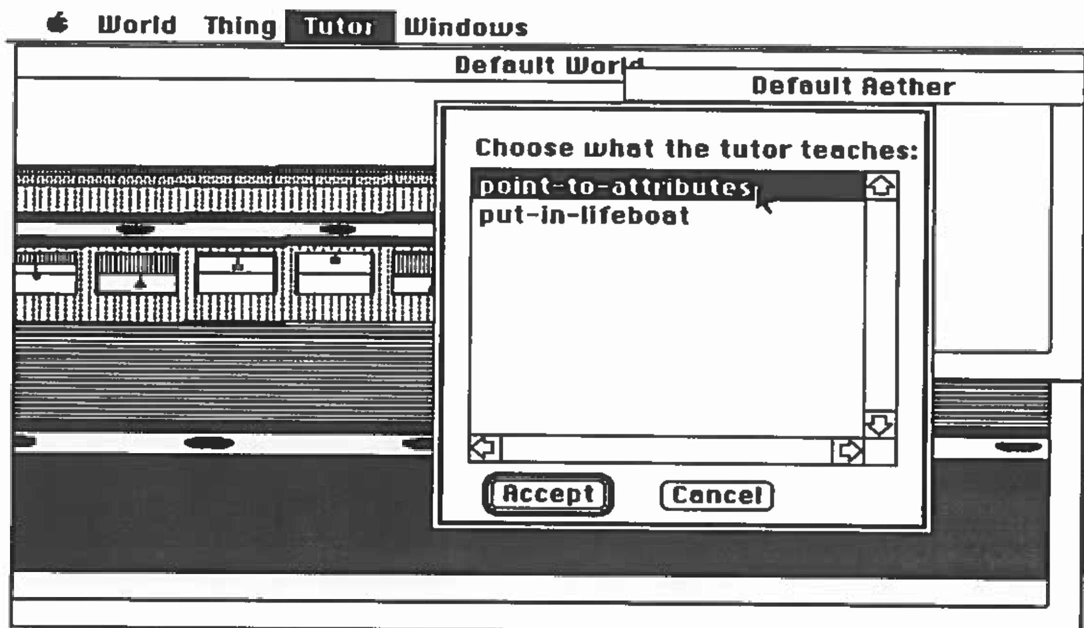

Figure 39. What the Tutor Teaches

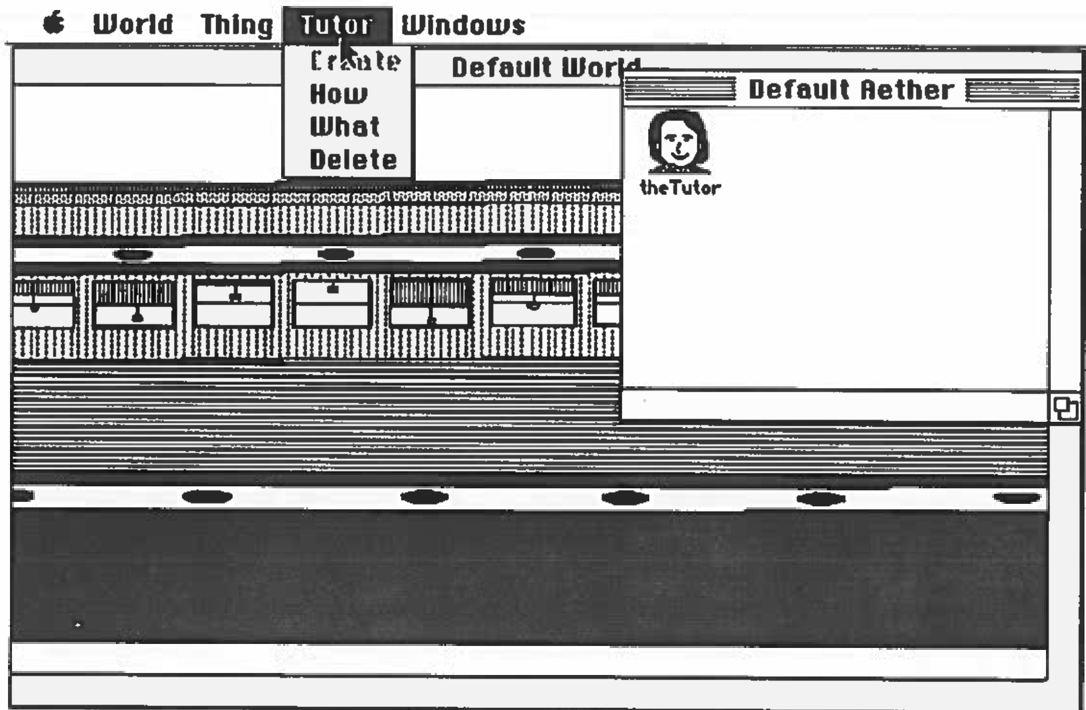

Figure 40. Tutor Menu when Tutor Exists

language.

3.5 User Interface

3.5.1 The World Menu

Three commands are available from the *World* menu when MicroWorld is first started: *Open, New,* and *Quit*. *Open* uses a file list dialog similar to the file list dialog shown in Figure 16, except files with a different type are displayed. Selecting a file loads the previously saved world. The *New* command was describe above. *Quit* exits MicroWorld, but checks for unsaved changes before proceeding. An alert is

displayed asking the user to either save or discard the changes.

When a world is loaded into MicroWorld (using either the *Open* or *New* commands) The rest of the *World* menu commands become available, plus the *Create* commands in the *Thing* and *Tutor* menus. Since MicroWorld only allows a single world loaded at a time, the *Open* and *New* commands are deactivated.

The *Close* command closes the current world. If the world was modified, then the user is asked whether to save the changes, or cancel the close operation. If the user selects yes, then the *Save* command is performed, before closing. The menus are returned to the previous state (Figure 12).

The *Save* command saves the file under its current name. Similarly, the *Save as* command saves the file, but under a different name. A file list dialog is present for the new file name (Figure 41). The old file is copied to the new file and the old file remains unchanged.

Both the *Run* and *Stop* commands are used to test the lesson. *Run* starts the lesson as though the instructor is a student. *Stop* returns the system to authoring mood.

Sound files used by MicroWorld are kept in separate files from the world files. A sound file may be shared between several different worlds. The *Lexicon* command presents a file list (same as Figure 16, except a different

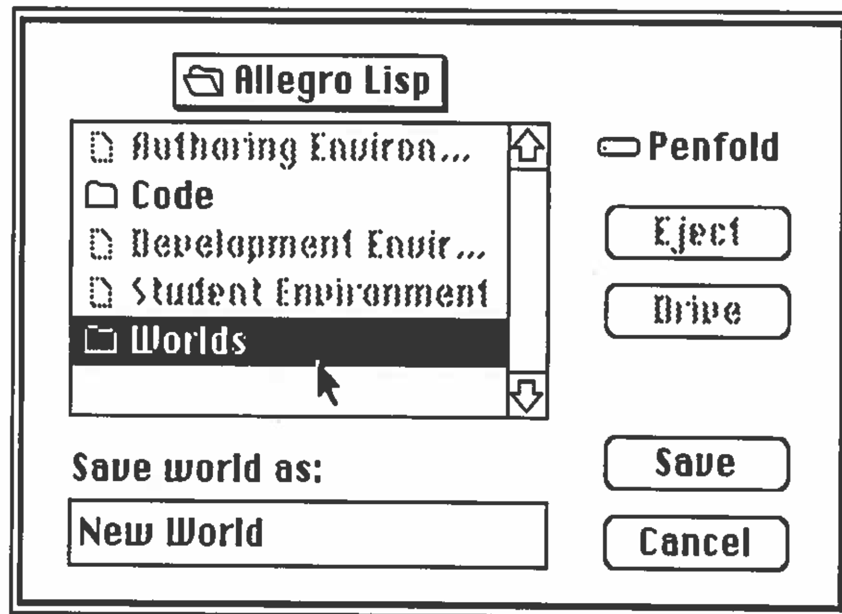

Figure 41. Saving a Copy of the World

type of files is shown) of available sound files, and sets the current lexicon to the file chosen by the user.

The *Background* command changes the current background. The same steps are performed as setting the background for the first time (see Section 3.4). *Print* prints useful information about the world to the printer.

If a world is currently loaded, the *Quit* command performs a *Close* command before exiting to the Finder. The *Close* command, as described earlier, checks for unsaved changes and gets confirmation from the user.

3.5.2 The Thing Menu

The *Create*, *Movement*, and *Actions* commands' description are given in Section 3.4 and are not repeated here. The *Duplicate* command creates a copy of the selected Thing with a new name. The user types the name (Figure 20) and it is checked for a name conflict. If a conflict exists, the name is not allowed and the user must try a different name. If the user cancels the command, the copy is removed.

The *Rename* command also requests a new name from the user, but it just changes the name of the selected Thing. If the user cancels this command, the name remains the same.

The *Image* command sets the image of the selected Thing, as described above (Figure 21). If the user cancels the command, then the image remains the same. The Thing's icon is not set by this command.

Set Position sets the initial position of the Thing. The user determines the position by moving the Thing in the environment of the lesson, instead of typing a numeric value for the position. Although this method is not as exact, direct visual feedback makes the task easier.

Each *Thing* is assigned the next highest plane number upon creation. This number determines the stacking order of the images on the screen. *Exchange Planes* switches the plane numbers between two *Things*. Unlike the other commands, only two Things can be selected during this command.

The *Delete* command removes, after user confirmation, the selected *Thing* from MicroWorld. If all *Things* are removed from MicroWorld, the *Thing* menu returns to its previous state (Figure 19).

3.5.3 The Tutor and Windows Menus

The *Create* command's description is given above and is not repeated here. The *How* command resets how the *Tutor* teaches the lesson (see Figure 38). Similarly, the *What* command resets what the *Tutor* teaches (Figure 39). *Delete* removes the *Tutor* from the aether. The *How*, *What*, and *Delete* commands become active when the *Tutor* is created and become unavailable when the *Tutor* is deleted. The *Create* command becomes unavailable while a *Tutor* exists.

The *Windows* menu selects between the different windows. Currently, only two windows are defined, *world* and *aether*. If more windows are defined later, the names are added to this menu.

3.5.4 Programming Interface

This sections describes, in greater detail, the dialog boxes used for the *Movement* command and for editing a *thing's* attributes, local actions, and global actions. The dialog boxes were describe briefly in the world-building example given above.

The dialog in Figure 26 sets the movement path for the selected Thing. The box on the left presents the currently defined path in horizontal and vertical components, separated by a comma. The *Add* button adds the values in the *H:* and *V:* fields to the end of the movement path.

The *Replace* button replaces the selected point in the defined path with the values in the *H:* and *V:* fields. *Insert* adds the values in the "H:" and "V:" fields before the selected item.

The *Delete* button removes the selected point from the defined path. *Clear* removes all of the points in the path. The *Follow Mouse* button defines the path by following the mouse as the user drags the *Thing*. Currently, this command is unavailable.

The movement mode selects the method used to moves from one location to another location. If the mode is *Drag*, then the *Thing* moves smoothly from one point to the next point. The size of the movement is determined by the *Drag Increment* field. If the mode is *Jump*, then the Thing disappears at the current location and reappears at the next position.

The dialog box in Figure 28 presents the attributes to the user. The box on the left shows the Thing's defined attributes. The *Value*: and *Type*: fields display the value and type of the selected attribute. In Figure 28, the selected attribute is *animate*? and its current value and type

are *t* (for true) and *<boolean>*.

The *Replace Value* button sets the current value of the attribute. The type of the attribute is set upon creation and can not change during its lifetime. The attribute must be deleted and a new one created to change the type.

New Attribute creates a new attribute under the name given by the user and a type selected from a list. *Delete Attribute* removes the attribute from the list, if it was defined by the user. The attributes shown in Figure 28 are defined for the Thing by the system, and can not be deleted.

The dialog in Figure 29 is used to editing local actions, which are shown in the left-side box. These actions are defined by MicroWorld only; the user can not add or delete local actions.

The return type is shown in the *Type*: field for the selected action. All local actions are of type *<command>*. Global Actions (described below) can also be functions, with return types of either *<boolean>*, *<integer>*, *<location>*, *<sound>*, or *<thing>*.

After selecting an action, the user can either edit, view, explain, or run the action. Figure 30 shows the user editing the local action called *landing action*.

In Figure 30, the script for *landing action* has not been written, therefore, no action is performed when the Thing lands. The *Code*: field contains the commands for the

action, if defined.

The pop-up menu in Figure 30 was created when the mouse button was pressed on the first line of the *Code*: field. The menu presents the list of available commands to insert at that position. The menu in Figure 30 is an example of the commands available in MicroWorld, but it is not a complete set.

When a choice is made from the menu the command, in this case *move*, it is inserted at the selected line. Another dialog is created for the user to set the parameters of the command. As the user programs an action, the dialogs are stacked to give visual feedback on the current position in the code. Figure 31 is an example for the move command.

Two fields, *Name*: and *Parameters*:, are used to present the call of a command. The user can view or edit the definition of the global command (if user defined) by selecting its name with the mouse, and choosing *Definition* from the pop-up menu (see below). In the example, selecting *move* creates a new dialog with the definition for move, if move was user defined.

In Figure 31, the parameters to the *move* command are not defined, instead the expected value's type is shown for each parameter. In this example, the *move* command expects a Thing and a location. Pressing the mouse button when a line is highlighted creates a pop-up menu containing the allowed

items for the selected parameter. The selected line in Figure 31 is the *<Thing>* parameter.

Values have several different sources, therefore, the pop-up menu is divided into fields, separated by lines. The first field contains the command to expand the current definition, only if the parameter is a function call. Expanding the function creates the same type of dialog as shown in Figure 31.

The second field contains the functions that return a value of the appropriate type. This example contains some of the *Thing*-returning primitive functions defined in MicroWorld. If the user defines a global function that returns a *Thing*, then the name of the defined function is added to this field.

The third field contains the names of the *Things* defined in the system. *Things* are referenced by name, therefore, the name can be used anywhere a parameter of type *<thing>* is expected. The name of *self* refers to the current Thing. In this case, it is the *Thing* with this *landing action*. *Foo* is the name of another *Thing* defined in MicroWorld.

A fourth field (not shown) is used to display the attributes and the parameters of the definition of type *<thing>*. Currently, no attributes are defined as type *<thing>*. Also, since the current definition is a local action, no parameters are passed. If the definition was a

global action (see below), then all parameters of type *<thing>* would be in the fourth field.

The pop-up menus for the other types only contain three fields, the first, second, and fourth fields defined above. Instead of the list of *Things* defined in the world, the first field contains another menu item to define a constant for that type.

If the *Accept* button is clicked, a test is performed to determine if all of the parameters are set. In the example, the user can not accept the changes until both the *<thing>* and *<location>* are replaced in the *Parameters*: field. If the *Cancel* button is clicked, then the newly added line is removed from the previous dialog. In both cases, the previous dialog is updated and becomes active.

When viewing the code, the user can not make changes to the definitions. The only commands available are to expand a command or function, or to expand the definition of another command or function. Blanks line are not inserted between lines in the *Code*: field, as these lines exists for the insertion of new commands. See below for an example of the differences between editing and viewing code.

The *Explain* button gives an explanation to the user about the action. For example, the explanation of *landing action* is: "This user-definable action occurs after the user drags the thing." This mechanism provides the means to

explain the purpose of built-in primitives and actions, and for the user to comment their definitions.

The *Run* button performs the command. This allows the command to be tested without the need to return to the windows, selecting *Run* from the *World* menu, and then performing the action on the Thing.

Editing global actions is similar to editing local actions. A list of available global commands and functions is presented to the user (Figure 32). Two new buttons are added to the dialog. The *Delete* button removes the selected definition, unless it is a primitive command or function.

The *New* button creates a new global action. First, the name and the return type are determined by the user. Second, the appropriate dialog is presented to display the new action.

Figure 33 is an example of the top level definition of a command in the editing process. The *Parameters*: field displays the name and type of each parameter used in a call to this command. In this case, all parameters are of type *<thing>*. Blank lines exists between the parameters to allow the user to insert a new parameter at any position.

The *Code*: field contains the commands to perform the *arrange* command. The blank lines are used as described above, to allow insertion of new lines of code. If the user selects a blank line in the *Code*: field, then a pop-up of

menu of available commands is presented (see Figure 30).

The code for this definition contains four commands, two *moves*, an *if*, and a *says* command. The currently selected line is the *if* command and the menu selection of *Expand* creates the dialog in Figure 34.

The format of an *if* statement is: *if <boolean> then <commands> else <commands>*. In the example, the *if* field contains a function that returns a Boolean value. Selecting this field creates a pop-up menu that contains all constants, functions, and variables of type Boolean. The *then* field and *else* field contain commands to perform if the test is true or not true. The same commands are performed on these fields as the *Code*: field for the *arrange* command (Figure 33).

Figure 35 shows the editing of the list of sounds from the fourth line of code in the *arrange* command (Figure 33). A list is presented in the outer definitions as the type surrounded by the '*[*' and '*]*' characters (in the example, the list is *[sounds]*). The dialog in Figure 35 was created by expanding the *says* command and expanding the *[sounds]* parameter.

When editing, the lists contain blank lines between each item. An item can be inserted at any position. When the item is inserted, blank lines are added to allow for more insertions. Pressing the mouse button at the current position presents a pop-up menu with all available sounds.

Figure 36 is the *arrange* command when the user is viewing the code. Since the user can not insert new items, no blank lines exists between each line. Also, the pop-up menu contains only the commands that do not modify the code.

3.6 MicroWorld's Programming Language

MicroWorld's programming language is divided into five categories: commands, functions, arithmetic, assignment, and flow of control. The user is allowed to define new function and commands, which are accessible by all objects in the world.

Table 1 is a list of primitives currently defined in MicroWorld. Parameter types are surrounded by '<' and '>', such as <thing> for a parameter of type Thing. If the parameter's type is plural (e.g. sounds or things), then a list of items is expected.

3.7 Conclusion

MicroWorld provides an environment for instructors to generate lessons easily. Its programming language is well suited for direct-manipulation type lessons and is less generic than most other languages. The instructor is directing the objects, instead of programming the host machine.

Providing an easy-to-use and intelligent environment is important for building good instructional lessons. The in-

structor will develop better lessons, if he is provided with an easy-to-use environment, which gives direct visual feedback. Furthermore, the instructor will create more lessons, if the less work is required.

MicroWorld attempts to provide the instructor with the necessary tools for creating direct-manipulation type lessons. How well it succeeds is unknown, since the environment has not been tested with actual instructors. Through limited in use, it appears to be a good environment overall.

Table 1. MicroWorld's Programming Language

Commands

move <object> <location>
moves (movement shown on screen) the object from its current location to <new location>.

jump <object> <location>
jumps (movement not shown on screen) the object from its current location to <new location>.

move <object> along <path>
the object moves to each point in the path.

jump <object> along <path>
the object jumps to each point in the path.

say <sounds>
each sound is played in order.

animate <objects>
the objects are moved along their path simultaneously.

highlight <object>
inverts the image of the object. An object appears selected when highlighted.

un-highlight <object>
restores object to the original image.

flash <object>
highlights and then un-highlights the object.

set <attribute> of <object> to <value>
Sets the object's attribute to value. The type of the attribute determines the type of the value. The value of the attribute is retrieved by using the name of the attribute, instead of an explicit *get* function.

Flow of Control

repeat until <boolean> <commands>
the commands are repeated until the boolean test is true.

Table 1 (continued).

repeat while <boolean> <commands>
the commands are repeated while the boolean test is true.

repeat for <count> <commands>
repeats commands for n number of times, where n is an integer great than or equal to zero.

if <boolean> then <commands> else <commands>
if the boolean test is true, then the commands following the "then" keyword, but before the "else" keyword are performed. If the test was false, the commands after the "else" keyword are performed.

Arithmetic

sum-of <integer> <integer>
returns an integer that is the sum of two integers.

difference-of <integer> <integer>
returns an integer that is the result of subtracting the second integer from the first integer.

multiplication-of <integer> <integer>
returns an integer that is the multiplication of two integer.

quotation-of <integer> <integer> - returns the integer result of dividing the first integer by the second integer. A zero is returned if the user tries to divide by zero.

Functions

to the left of <object>
returns the prototypical "left" location of the object. Calculation is performed in the context of another object, such as *move object1 to the left of object2*. The prototypical location is based on both the center of mass of the second object, and on the vertical size of both objects.

Table 1 (continued).

to the right of <object>
returns the prototypical "right" location of the object. The calculations are performed in the same manner as the "to the left of <object>" function.

above <object>
returns the prototypical "above" location in relationship to the object.

below <object>
returns the prototypical "below" location in relationship to the object.

<object> is to the right of <object>
returns true if the first object is to the right of the second object. Both the centroid and corridor methods are used to determine the relationship of the two objects. Information about the object determines which side of the screen is the right side. For example, an object that represents a human facing out from the screen (i.e. towards the user) has the right side opposite of an object representing a box. When viewing an image of a human, the right side is determined by the right hand on the image and not by the view on the screen.

<object> is to the left of <object>
returns true if the first object is to the left of the second object. Again, information about the object determines which side of the screen is the left side.

<object> is above <object>
returns true if the first object is above the second object. The concept of "above" is based on the two dimensional plane of the screen and not on what the objects represents. In other words, an object is above another object if it is closer to the top of the screen. This test would return false if one object was above another object, but the view was from above looking downwards.

Table 1 (continued).

<object> is below <object>
returns true if the first object is below the second object. Again, the position on the screen determines the results, not what the object represents.

<object> is on <object>
returns true if the first object is on the second object. This tests whether the image of the two objects overlap and the first object is on a higher plane than the first object. If the viewpoint is overhead, then this function tests whether the first object is above the second object in the real-world representation.

<object> is under <object>
returns true if the first object is under the second object. This is equivalent to *<object2> is on <object1>*.

things to the left of <object>
things to the right of <object>
things above <object>
things below <object>
things on <object>
things under <object>
each function returns a list of things for which the corresponding test is true. For example, *things to the left of <object>* returns a list of things for which the test *is to the left of <object>* is true.

the location of <object>
returns the centroid location of the object.

CHAPTER 4

COMPARISON BETWEEN PROGRAMMING BY REHEARSAL, HYPERCARD, AND MICROWORLD

This chapter discusses the similarities and differences between MicroWorld and Programming by Rehearsal (see Section 2.1), and between MicroWorld and HyperCard (see Section 2.2). MicroWorld shares many similarities with Programming by Rehearsal, but also corrects many design flaws. MicroWorld shares less similarities with HyperCard, but HyperCard is also suited for building instructional lessons.

4.1 Programming by Rehearsal vs MicroWorld

Programming by Rehearsal forms the basis for the MicroWorld project, therefore, they share many features. Both environments are built upon the object-oriented concept. A screen image corresponds to an object within the environment and these objects communicate through messages. From another view point, the screen's graphical images forms a bridge between the user and the internal objects.

Not all objects are visible to the student in the environments, but both provides means for the instructor to communicate with the invisible objects. Programming by Re-

hearsal's invisible objects are placed in the wings, which are offstage in the theater metaphor. The wings are visible to the instructor, but invisible while the student studies the lesson. MicroWorld's invisible objects reside in one of two locations. If the object is normally invisible (e.g. the tutor), then it resides in the aether *Base-World*. Otherwise, it resides in the world *Base-World*, but stays invisible until made visible while the student studies the lesson. While the instructor builds the lesson, all objects are visible.

The instructor positions screen objects by dragging the object to its location. Rehearsal World automatically sets this location as the object's initial position. MicroWorld requires a menu selection before setting the object's initial position, which allows the instructor to view the objects at different locations without changing the initial position.

Furthermore, the object's movement path can be created by following the mouse's movement. Rehearsal World uses only this method for path definition, but the user can also type location constants in MicroWorld, which makes editing the path definition easier.

Object programming is primarily performed by pop-up menu selections. Rehearsal World's pop-up menu contains the methods to which the selected object responds. Programming is performed through a cue sheet and interacting directly with the object. The instructor is not required to use the pop-up

menu and can program the objects with the keyboard, thus allowing errors which could crash the system (e.g. sending an inappropriate message to a Smalltalk global variable). Furthermore, errors are not caught until run time. The error could occur while the student is using the lesson, which is less than ideal.

MicroWorld uses a structure editor, which helps the novice user with programming the objects. MicroWorld separates the commands and functions into separate pop-up menus. The functions are further separated by return value's type. Only the appropriate menu is displayed when the instructor presses the mouse button. MicroWorld's approach provides strong type-checking during programming, which avoids runtime errors.

From within the Programming by Rehearsal environment the instructor can test a lesson by using two different methods. First, the instructor can send a single message to an object and watch its behavior. Second, the instructor can place the environment into "student" mode, which runs the lesson. The instructor can view the lesson for possible problems.

The lesson types that Rehearsal World can build is restricted for four reasons: event types are limited, no spatial relationship between objects, no lesson history or evaluation, and no sound. Rehearsal World's graphical images represent a simple user interface object that the user can

only select, which limits the environment to a single mouse click. Although Rehearsal World increases the flexibility of object's behavior by evaluating two separate methods for this event, which depends on the object receiving the *becomeAButton* message, it restricts the lesson to a simple point-and-click type, i.e. exploratory.

MicroWorld's graphical images represent real-world objects and, therefore, the objects can receive more event types, which includes the user dragging it from one location to another. Smalltalk does provide a method for following the mouse (usable by Programming by Rehearsal's objects), but it does not send events to the object during motion nor sends an event when the user releases the mouse button. The instructor is required to program in Smalltalk to provide this functionality.

MicroWorld uses four event types, mouse click, mouse double click, dragging, and landing. Although MicroWorld does not allow runtime modifications to the object's behavior, which Rehearsal World performs by the *becomeAButton* message, individual event responses can be turned on or off during the lesson.

Programming by Rehearsal's objects cannot determine their locations with respect to other objects, which makes lesson based on spatial relationship impossible. MicroWorld's objects can calculate their spatial relationship

with other objects, although the results are not guaranteed correct. For example, MicroWorld cannot find the object which is closest to another object's prototypical location.

Programming by Rehearsal neither records nor evaluates students' actions, which makes it a less effective teaching tool. MicroWorld provides instructor-programmable history and evaluation actions, which tailor the lesson to the student's particular needs.

Sound is another important feature missing in Programming by Rehearsal. Digitized sound is important for teaching spoken language, but it also increases the interaction between the student and the lesson.

The instructor creates simple lessons easily using Programming by Rehearsal, but as the lesson's complexity increases, the instructor must write more Smalltalk code (see Chapter 2). Smalltalk is a large and complex environment, which increases the instructor's learning curve and increases the potential for errors. MicroWorld shields the instructor from its development language (Lisp) by providing its own programming language, which is specially designed for building lessons.

4.2 HyperCard vs. MicroWorld

HyperCard is a Macintosh generic programming environment and is not a specific environment for developing instruction-

al lessons, although several lessons have been built using HyperCard, which leads to the following comparison. HyperCard shares some features with both MicroWorld and Programming by Rehearsal, but, in general, its approach is far different.

The user creates and positions graphical images slightly different in HyperCard than either MicroWorld or Programming by Rehearsal. The three environments position objects using the mouse, but HyperCard includes a built-in paint program for creating the images. A separate program creates MicroWorld's and Programming by Rehearsal's images.

The user can test a program by returning HyperCard to browsing mode, which is similar to MicroWorld's and Programming by Rehearsal's testing mode. HyperCard sends events to the objects beneath the cursor and the objects evaluate the user-created scripts, until the user enters edit mode. Any user (including students) can activate edit mode, which makes HyperCard susceptible to malicious modifications.

HyperCard's and MicroWorld's programming language syntax are similar. Objects receive direct commands to perform an action. For example, HyperCard's statement "go to <card>" is syntactically similar to MicroWorld's statement "move <object> to <location>." The first statement activates the former object and the second statement moves the latter object to a new location.

Although the languages are syntactically similar, The method for entering the program is quite different. MicroWorld uses pop-up menu exclusively, except for numerical constants and variable names. HyperCard uses the keyboard exclusively. Furthermore, HyperCard's does not have a structured editor and errors occur during runtime (not during compilation nor editing), therefore it is less helpful to the novice user and allows errors to occur while the student is using the lesson.

Furthermore, HyperCard has two distinct features not found in MicroWorld. First, HyperCard's has untyped and variable-numbered parameters. The programmer must explicitly test the parameter's type and test for the correct number of parameters, otherwise runtime errors can occur. To protect novice programmers, MicroWorld checks the parameters' types and only allows the correct number of parameters. Second, HyperCard has separate syntax for commands and functions, which can confuse the novice programmer. MicroWorld keeps its syntax consistent between commands and functions.

HyperCard makes a distinction between objects' types. Some message are receivable by all objects and other messages are only receivable by objects with a particular type. Objects can not change their types during their existence, but objects can be created and deleted during the program's execution. MicroWorld's objects belong to a single type and

contains instance variables and event responses, which individualize the objects.

A one-to-one correspondence does not exist between graphical images and HyperCard's objects. The programmer can place an invisible button above the graphical image, but, in most cases, the button won't completely overlap the image, therefore the user can select the image without touching it with the cursor. A graphical image is a MicroWorld object's attribute.

MicroWorld's and HyperCard's event types are different. HyperCard using the Macintosh's operating system event types, whereas MicroWorld uses a higher conceptual event model. HyperCard is a tool for programming the Macintosh, therefore, it provides the same event types. MicroWorld's focus is building lessons that model real-world object, therefore, it groups events based on conceptual actions, such as selection, movement, etc.

HyperCard's "stack of cards" concept is missing completely from MicroWorld. Each card is similar to a separate world in MicroWorld, which only contains a single world. Furthermore, HyperCard allows the selection of a word or phrase to activate another card, which could contain more information about the word or phrase and is a great feature for teaching and browsing textual information. No mechanism exists in MicroWorld that allows a Thing to activate a

different window, which is a limitation.

4.3 Conclusion

MicroWorld provides the tools for creating a larger variety of lessons than Programming by Rehearsal by having uniform objects with more event responses and a programming language that shields the user from the development environment. MicroWorld's environment can create lessons based on spatial relationships and lessons that use sound production. MicroWorld's built-in language avoids run-time errors, plus protects the development environment from corruption by an error in the user's program. The user can modify the Smalltalk environment from Programming by Rehearsal, which could easily corrupt the environment.

MicroWorld's environment is more suited for building lessons than HyperCard's environment. HyperCard is good at creating button and text interfaces, but its separation of graphical images from the objects and inability to drag objects on the screen during run-time limits the types of lesson that can be created. Furthermore, HyperCard's programming language does not adequately protect the user from run-time errors, nor does it help the novice user.

MicroWorld can generate several different lesson types, which includes selecting objects on the screen (easily created in both Programming by Rehearsal and HyperCard) to moving

objects in relationship to each other (hard to create in Programming by Rehearsal and HyperCard). Furthermore, a lesson can contain digitized sound, which is unavailable in Programming by Rehearsal.

MicroWorld was especially designed for user with little programming experience. MicroWorld provides a direct-manipulation environment for lesson creations and the user can view or edit objects' behaviors at any time. The programming language is type-checked during editing and avoids run-time errors. Novice users do not receive the same level of support from either Rehearsal World's or HyperCard's programming environment.

CHAPTER 5

MICROWORLD'S DESIGN

The design of any large system goes through many revisions and MicroWorld is no exception. The following is the design of MicroWorld, with suggested changes and improvements, at the time this paper was started. The work on MicroWorld continues, therefore, some information is outdated. Still, it is worthwhile to examine the design at this point.

5.1 Introduction to the Object-Oriented Approach

The development language for a project influences the overall design. In a procedural language, such as C, a global approach is used. Data important to the system are either kept in global variables, or are passed as parameters. Functions are written to perform particular tasks on only certain types of data.

Object-oriented languages take a different approach. An object is an encapsulation of data and methods into a logical unit. For example, a graphical object's data could be its bitmap representation and position on the screen. A possible method is for it to draw itself on the screen.

Sending a message is the only access to an object. An outside object can not change the internal state of an object, only the methods of the object itself can modify its data. The object's representation of data can change without affecting other objects that communicate with it.

Take, for example, an object that represents a point as a pair of integers for the X and Y coordinate. One of the object's methods returns the current X and Y values. Changing the object's representation of a point only requires changing the object's method to return the same values. If the point were represented as an angle and distance, then the previous method would need to calculate the X and Y coordinate. Other objects would still receive the same values, therefore, no modification are needed.

In contrast, changing the data's representation in the global approach is not so clean. The data can be accessed by several functions in different files. If even one function expects the data in the wrong format, then the program will give incorrect results or crash. Finding every use of the data is an error prone and time consuming task. The larger the project and the number of people working on it increase the difficulty of the task.

Objects provide more than a modular approach to a problem. They are a method for conceptualizing a problem through the use of class hierarchy and inheritances.

A class defines the structure and methods of an object. Several objects can belong to the same class and are called instances of that class. Each instance has its own set of data (in instance variables), but shares the same methods as other instances in the class. Therefore, each instance responds to the same messages, but it keeps its own version of the data.

Classes belong to a hierarchical structure. A class inherits the instance variables and methods of the class above it on the hierarchy, until the root class is reached.

When a message is sent to an object, its class is checked for the associated method. If the method isn't found, then the class above it on the hierarchy (i.e. superclass) is checked. This search continues up the hierarchy until the method is found, or until the root object is reached. If the method isn't found, then an error occurs.

This search mechanism allows the class to override the methods of its superclass. The method can also invoke the superclass method during its execution. The procedure for calling the superclass' method is dependent on the language. In Smalltalk, for example, the method sends a message "super." Therefore, the methods of a class can specialize or augment the methods of its superclass.

Three important areas need consideration when designing an object-oriented system, the structure of the class

hierarchy, the relationship of the classes and the data, and the relationship of the classes and the methods. The class hierarchy depends upon the conceptual model of the system and the problems it was designed to solve. For example, a system for modeling transportation vehicles has the classes Car and PickupTruck as subclasses of PersonalTransportation, which is a subclass of Transportation.

The class hierarchy contains the most abstract classes at the top of the hierarchical tree and the most specific classes at the leaves. The superclass should provide at least some support to the subclass. If a class uses a minimal set of instance variables and methods provided by the superclass, then the class probably shouldn't be a subclass of its superclass.

For example, the *Tutor* class is a subclass of the *Thing* class in MicroWorld (see Section 5.4). *Tutor* uses a very small set of the instance variables and methods defined in *Thing*. The *Tutor* is considered an invisible *Thing*, therefore it was defined as a subclass of *Thing*. A better design has two subclasses, *Aether-Thing* and *World-Thing*, of the class *Thing*. *Aether-Thing* represents the invisible and *World-Thing* the visible objects in the world. *Tutor* would be a subclass of *Aether-Thing*.

Separating the *Thing* class is not solely based on the object's visibility. Several instance variables are used for

handling the user's events. For example, an instance variable contains the method for a *click* event. The *Tutor* does not handle any events directly, therefore, the *Tutor* should not have instance variables for handling these events.

In the future, the *Tutor* will handle events posted to a queue from several different sources. Although both classes will then handle events, the processing involved is different. The *Tutor* will accept events that *Things* do not use. For example, the *Tutor* will handle a null event, which is not used by *Things*. Separating the classes into *Aether-Thing* and *World-Thing* is still valid.

As the example shows, in which class the instance variables and methods are defined is important for good system design. An instance variable should be defined in the highest class that uses it. Methods, on the other hand, should be defined in the lowest class possible. If two classes use the same method, then the method should be defined in the immediate superclass.

Using the example above, the instance variable for the *Thing's* image should be defined in *World-Thing*, which is the highest class that uses it. On the other hand, both *World-Thing* and *Aether-Thing* have an icon, therefore, the icon instance variable should be defined in *Thing*.

Both the image and the icon representation of *Thing* are bitmaps and the methods for drawing them are the same. The

method can be defined using one of three alternatives. First, since *World-Thing* is either drawn as an image or an icon and *Aether-Thing* is only drawn as an icon, a method for drawing either the image or the icon is defined for *World-Thing* and a method for drawing the icon is defined for *Aether-Thing*.

Second, the method for drawing an image or icon is defined in *Thing*. The subclasses then provide information for which item to draw, either by passing the information with the message or the draw method request the information from the subclass.

Third, an instance variable in class *Thing* holds the current representation. The subclass is responsible for setting the instance variable to the correct representation, either an image or an icon.

The second and third alternatives are preferable over the first alternative as less code is duplicated between the objects. The best choice between the second and third alternative is based on the usage of the current representation. If the representation is used often and set seldom, then the third alternative is preferred since the information is easily accessible.

This illustrates another design decision, using an instance variable versus sending a message. Accessing the contents of an instance variable is faster than a message

send, but uses more memory. Two factors are involved in the decision, the frequency of use and the difficulty in finding the information. If the information is used often, or several messages are needed to get the information, then an instance variable should be used. Otherwise, a message should be sent for the information.

Another problem is getting information to the object that needs it. Two alternative are available, send the information with the message, or have the object send a return message requesting the information. If the information is passed to several objects that don't use it before it gets to the final object, it is better to have the final object request the information. On the other hand, the possible difficulty in determining which object to request the information from may make passing the information in a message the better approach. Again, it depends on the design of the entire system.

Another design problem is determining which object or objects should control the system. Global control uses one object to control the entire system, while local control uses several objects. The project and the operating system of the host machine determines the best approach. If the operating system is event driven, then local control is better.

5.2 The Effect of the Object-Oriented Approach on the Implementation

Using the object-oriented approach affects the implementation as well as the design. Sending a message between two objects is expensive in comparison to keeping the information local in terms of speed of execution, but is less expensive in terms of space. This decision is made throughout the implementation phase of any project.

For example, a *Thing* object has two instance variables for the *UWorld* and *Base-World* objects. Furthermore, the *Base-World* and *UWorld* objects both contain an instance variable that makes reference to the other object. Therefore, *Thing* contains redundant information by using two instance variables. The *Thing* object can communicate indirectly with one object by using the other object as an intermediary. The Thing object could access the UWorld object through the Base-World object or vice versa. Although it requires more space, *Thing* sufficiently accesses the *UWorld* and *Base-World* objects to warrant having an instance variable for both objects.

Determining the information contained within a class is another design choice. If one object makes frequent request for information from another object, then duplicating the information increases the performance of the program. Duplication of information is contrary to the object-oriented design approach. When several objects contain the same in-

formation, especially instances of different classes, the locality of information is lost. Changes must propagate to all objects that contains a copy of the information. If an object often uses the information defined in a different class, then the information probably belongs to the first object.

In MicroWorld, two classes in frequent conversations contain an instance variable that points to the other object. Duplication of object pointers are restricted to three classes. *UWorld* points to all *Things*, all *Base-Worlds*, and all instance of *UW-Method's* subclasses. *Base-World* also points to *Thing's* instances contained within the *Base-World's* boundaries and the *UWorld* object. A *Thing* object points the *Base-World* and *UWorld* objects. The increased performance outweighs the complex connections between objects created by this design. Other information is not duplicated between objects.

In an object-oriented system, another trade-off exists between sending information with a message or letting the receiver request the information from the sender. For example, an object obtains the user's selection from a list of items. Completing the selection sends a message to another object, which indicates this event. Two choices exists, either sending the selected item with the message or saving the item and let the receiving object request it.

The best choice depends on two factors, the number of times objects forward the selection until the value is used and the ease in which the receiving object can locate the sending object. The object should send the selection with the message, if the receiving object makes use of the information. If the message goes through several objects before the information is used, the choice is less obvious. For example, object A sends a message to Object B, which forwards it to Object C, and finally reaches Object D. If Object A sends a value to Object D, the Object B and C pass unused information. If the final object requires several messages to determine the original object, the sending the information with the message is the best solution, even with a long path of messages.

A case exists in MicroWorld where unused information is sent to an object. When a dialog item pops up a menu, the dialog item sends a reference to itself with the menu's message. The menu forwards the information to the current *UW-Method* instance, which uses it to request the code position of the user's selection. The pop-up menu, on the other hand, does not use this information, but *UW-Method* can not easily determine the dialog item.

5.3 Languages Used in MicroWorld

Three criteria were used in choosing MicroWorld's development language. First, the language had to be available for the Macintosh computer, which was chosen for its user interface and its use in the education field. If someone wants to use MicroWorld, chances are good that the equipment is available and the person has experience using the Macintosh.

An object-oriented language is the second criteria. The combination of modularity, inheritance, and overall design approach, makes an object-oriented language well suited for the building of large programs. A true object-oriented language was not a necessity; it could have object-oriented extensions. For example, Lisp with the Flavors package is a suitable language.

The third criteria is a good development environment. It is essential when building a large system, even with the modular approach of an object-oriented language. This environment should provide:

- error detection,
- a backtrace of the current objects and methods leading to the error,
- inspection of the objects and instance variables, and
- step and trace functions.

Error detection is an obvious choice. A backtrace provides the activation stack of objects and messages. Any object can

send the same message to an individual object, therefore the activation stack is important for determining the message's path. Inspection allows the programmer to examine the object's state. Coupled with the backtrace ability, the programmer can also view the object's state within the current activation. Step and trace functions allow the programmer to watch the dynamic behavior of the program.

At the start of MicroWorld's development, the Neon programming language met the most criteria. Neon was used for early prototypes to explore the feasibility of object-oriented languages. Neon is a version of Forth (stack-based, threaded-interpeted language) with object-oriented extensions. Forth's syntax has the parameter list before the function or procedure call (i.e. reverse polish notation). Neon uses the same syntax for messages. The parameter list comes first, followed by the message, and finally the object.

Unfortunately, Neon is lacking in two important areas. First, its extension to Forth is not truly object-oriented. Second, its total lack of a development environment.

Neon has one global object called the stack, which is used throughout the system, including the object package. This leads to difficulty in several respects. One of object-oriented concepts is modularity; objects only interact through messages. In Neon's case, objects interact through the stack. Although message's parameters are pushed on the

stack, methods can use any stack value, including values that are not part of the message. In other words, a message with too few or too many parameters is hard to detect.

Determining when the program uses a stack value is also difficult. Many messages and function calls could exist between placing the value on the stack and using the value, which increases the difficultly maintaining the program.

Stack overflow and underflow are the two major bugs sources in Neon. Stack overflow occurs when used values are not removed from the stack. Neon provides a function to test the stack's depth of the stack, but determining the responsible function or method is difficult.

Stack underflow occurs when too few values exists on the stack, caused by either not enough parameters for a function call or message, or not enough return values. Neon is very inconsistent about catching stack underflow errors. Only a subset of available functions check for stack underflow, thus the error isn't shown at the correct position, which increases the difficulty in debugging the problem.

The lack of a good development environment is compounded by bugs in the language itself. Several functions provided are not documented nor perform correctly. For example, the SIGN function is defined to return either -1, 0, or 1, based on the parameter's sign. Instead, the function returns nothing, causing a very difficult to find stack underflow

error.

Another difficulty exists with Neon's local variables and parameters. The programmer can only define names for six parameters and local variables. For example, if four parameters are given names, then only two local variables can be given names. This limitation causes the programmer to write complex stack manipulation to access the parameters and local variables above the limit.

Neon's source code is very difficult to read. Even with adequate documentation, determining how a function behaves is difficult. Stack manipulations, which are unavoidable, makes it harder to establish the parameters' values and their order on the stack. For example, MicroWorld contains a number of instances where one parameter (or more) stays on the stack through several calculations before its use in a message. These code segments are harder to read, especially for some-one besides the author.

As shown above, Neon is a poor choice for building large systems. The global stack, lack of a development environment, and the lack of sufficient named parameters and local variables, increases the development time when compared to other languages. Debugging the program takes a larger percentage of the programming cycle. MicroWorld would be nearer completion, if a better language was available at the project's inception.

Fortunately, Allegro Common Lisp became available in the project's second year. Lisp does not have the problems found in Neon. Unlike Neon's global stack, Lisp allows, but doesn't require, the use of global variables. Furthermore, the parameters follows, in order, the function call, and thus the programmer can easily determine the parameters' values. Lisp does not restrict the number of parameters and local variables with names.

Lisp's most important feature is a good development environment. Allegro Common Lisp provides the four essential tools outlined above. Building a large system using Lisp is easier than Neon, since it requires less debug time.

Allegro Common Lisp provides an object-oriented package called Object Lisp, which is different from Flavors and CLOS[3]. In Flavors and CLOS, objects are sent messages, which activate the corresponding methods. On the other hand, Object Lisp's objects are sent functions to evaluate within the receiving objects' environment.

For example, the message:

```
(ask a-window (move-to 5 6)
              (let ((x (+ 5 6))
                    (y (* 5 6)))
              (move x y))
              (print "I am here!")))
```

asks a window (called "a-window") to evaluate three statements. The first statement is a method defined for the

3. Two common object-oriented packages used with Lisp.

Window class. The second statement is a let statement, which allocates two local variables and then performs another method defined in the **Window** class. The third statement is a Lisp function, which is performed within a-window's environment.

The first statement has the following equivalent statement in Flavors:

```
(send a-window :move-to 5 6)
```

Only messages specified in the object's class and superclasses can be sent to the object, therefore the object's behavior is well defined. Objects' behaviors are not well defined in Object Lisp since any Lisp function can be included in the message, including functions that modify the object's internal state. No equivalent statement exists in Flavors for the second statement above.

Accessing instance variables is similar to accessing local variables within the object's environment. For example, suppose the class *Window* has "next-window" as an instance variable. The message:

```
(ask a-window next-window)
```

returns next-window's contents. Within *Window's* method definitions, using the atom "next-window" returns the same value as the previous message.

An equivalent statement in Flavors is:

```
(send a-window :get-next-window),
```

which is a method either created automatically by the compiler (when the variable is declared "gettable") or written by the user. The user determines which instance variables are accessible from outside the object in Flavors, which is not the case in Object Lisp. All instance variables are visible and can be set outside the object.

Object Lisp matches Lisp's semantics. A message is a request for the object to evaluate one or more functions within its environment. On the other hand, Flavors and CLOS distinguish between a function call and a message. Objects created in Flavors or CLOS can only receive messages defined for its class and its superclasses. Objects created in Object Lisp can also receive functions defined in the Lisp environment.

Object Lisp's message technique causes three problems. First, methods can not have the same names as functions defined within the Lisp environment. The conflict does not occur between methods defined in different classes, only between methods and Lisp functions. A method calling a function is exactly the same as the method calling another method defined within its class. The compiler can not determine which definition to use if the method and function share the same name, therefore, the names must be unique.

Second, instance variables are not hidden from functions and methods defined outside of the object's class. Any

function or method that knows both the object and the instance variable's name can get and set the instance variable's value. From the previous example, the message:

```
(ask a-window next-window)
```

returns next-window's value, even if the programmer desires restricted access to the data. In a large system, unrestricted access to an object's instance variables increases the probability of errors.

Third, messages can not directly include instance variables' values. For example, suppose a-variable is an instance variable defined for object-1. A method defined for object-1 can not send a message to object-2 of the form:

```
(ask object-2 (perform-method a-variable))
```

because "perform-method" executes in object-2's environment. A-variable is only bounded within object-1's environment and is either unbounded or bounded to a different value within object-2's environment. Binding the instance variable to a local variable (normally within a let statement) and sending the local variable with the message is the solution for this problem.

The first two problems are contrary to the object-oriented concept. Method definitions within a class are local to the class, which include the methods' names. By conforming to Lisp's semantics, naming conflicts are possible and thus, the class locality is lost.

Information hiding is an important concept in object-oriented languages, which Object Lisp lacks. In a true object-oriented system, all interactions between objects are through messages. Access to an object's instance variables is only achieved by sending a message to the object, therefore the value's internal format can be different from the external view. Furthermore, the value's representation can change without affecting other objects. Object Lisp's method evaluation within the receiving object's environment makes the object's instance variables visible to all outside objects.

Although Allegro Common Lisp has the two problems outlined above, its environment is a vast improvement over Neon. MicroWorld was rewritten in Lisp and the following sections describe the Lisp version.

5.4 MicroWorld's Class Hierarchy

An important design area in an object-oriented system is the class structure. The class determines the relationship between instances of different objects, and the inheritance of an object. The class structure is a tree with the most abstract classes at the top, and the leaves are the most specialized class.

Table 2 shows the class hierarchy in MicroWorld. Subclasses of a class are indented to the right one level.

Table 2. MicroWorld's Class Hierarchy

```
Object
   Category
   Concept
   *Dialog*
      Code-Dialog
   *Dialog-Item*
      *Button-Dialog-Item*
      *Static-Text-Dialog-Item*
         Code-Name-Dialog-Item
      *Table-Dialog-Item*
         *Sequence-Dialog-Item*
            Default-Choice-Sequence-Dialog-Item
            Image-Sequence-Dialog-Item
            Icon-Sequence-Dialog-Item
            Code-Sequence-Dialog-Item
   Function-Popup-Menus
   *Menu*
      Popup-Menu
         List-Popup-Menu
   *Menu-Item*
      UWorld-Menu-Item
   Movement
   Sound
   Thing
      Tutor
   UW-Bitmap
      UW-Background
   UW-Image
      UW-Icon
   UW-Method
      UW-Command
         Global-UW-Command
         Local-UW-Command
      UW-Function
         Boolean-UW-Function
         Integer-UW-Function
         Location-UW-Function
         Sound-UW-Function
         Thing-UW-Function
   UWorld
   UWorld-Author
   *Window*
      Base-World
```

For example, *Tutor* is a subclass of *Thing* and *Thing* is *Tutor's* superclass. Both *Thing* and *Tutor* are subclasses of *Object*.

Allegro Common Lisp, the language MicroWorld is written in, provides an object package that interfaces with the Macintosh Toolbox. The names of the classes start and end with the '*' character. Three classes, **Table-Sequence-Item**, **Sequence-Dialog-Item**, and **Window** are used as superclasses to classes defined in MicroWorld, but no instances are created. MicroWorld uses instances of the other classes defined by Allegro.

At the top of the class hierarchy is the class called *Object*, which is the most abstract class in an object oriented system. All other classes are subclasses of this class.

Object provides the most generic methods, usable by all instances. Some examples are the function *self*, which returns the current object, and *have*, which creates instance variables. In general, *Object* is responsible for creating and maintaining objects.

Class *Category* and class *Concept* are related. *Concept* defines the concept which the associated *Thing* represents. For example, a large black square has the concepts of "large", "black", and "square." *Category* groups the concepts of the same type. For example, the category for color

contains the concept of "black" and "white."

A *Thing* can be selected to teach a concept to a student by using objects from the *Category* and *Concept* classes. For example, if the student is learning the concept of shape, then the *Things* with unique shapes are selected from the category of "shape." Therefore, a circle and two squares could be selected to teach the shape of a circle.

Dialog provides an object-oriented interface into the Macintosh's Dialog Manager. A dialog is a type of window for displaying information and retrieving the user's responses. The **Dialog** class uses a list of **Dialog-Item** (or subclasses) instances and sends user-created events to the appropriate item. For example, if the button is pressed while the mouse is within the bounds of a **Dialog-Item**, it is sent a *dialog-item-click-event-handler* message.

Dialogs are either modal or modeless. A modal dialog retains control until the dialog is closed or until the dialog passes control to a menu, window, or another dialog. The user can change control to a different window or dialog when using a modeless dialog.

Code-Dialog specializes the *Dialog* class by sending null event messages to a list of dialog items. The dialog items' type determine the response to this event. *Code-Sequence-Dialog-Item* uses null events for highlighting the item under the cursor (see below).

The abstract class **Dialog-Item** provides the generic support for any dialog item. For example, the instance variables for text, position, and size are defined in this class. Event processing is also handled by **Dialog-Item**. When an a mouse up event occurs, for example, **Dialog-Item** evaluates the expression contained in the *dialog-item-action* instance variable.

Button-Dialog-Item supports the buttons used in a dialog. Clicking the button evaluates the *dialog-item-action*. One **Button-Dialog-Item** can be designated as the default button and is drawn with a three-pixel deep rounded rectangle border. Pressing the return key is the same as clicking the default button with the mouse.

Static-Text-Dialog-Item presents un-edittable text to the user in the dialog. **Static-Text-Dialog-Item** overrides the **Dialog-Item** class' mouse down event handler. **Static-Text-Dialog-Item** evaluates its *dialog-item-action* for a mouse down event, instead of a mouse up event.

Table-Sequence-Item displays a sequence of items, which are listed in column format with the current selection highlighted. The selection can either be only one item, a consecutive group of items, or any group of items, depending on the setting of an instance variable. Scroll bars are supported for lists that are too large to display at once. **Sequence-Dialog-Item** is used when the sequence is a list.

Another subclass of **Table-Sequence-Item** is used for arrays.

The four classes, *Code-Sequence-Dialog-Item*, *Default-Choice-Sequence-Dialog-Item*, *Icon-Sequence-Dialog-Item*, and *Image-Sequence-Dialog-Item* are subclasses of **Sequence-Dialog-Item**. *Code-Sequence-Dialog-Item* uses null events to highlight the list item under the cursor. A pop-up menu associated with the list item is activated when the mouse button is pressed.

Code-Sequence-Dialog-Items can be used in either edit or view mode. The code is presented on every other line in edit mode, but on every line in view mode. Each *Code-Sequence-Dialog-Item* has two pop-up menus. One menu is for the odd-numbered lines and the other menu is for the even-numbered lines. Instead of using an instance variable to keep track of which mode is in use, the menus determine the mode. When editing, the two menus are different, since the even-numbered lines have a different function than odd-numbered lines. When viewing, the two menus are the same.

Multiple items can be selected at the same time from a **Sequence-Dialog-Item**, therefore the selection is returned as a list. *Default-Choice-Sequence-Dialog-Item* provides a shorthand method for returning the first selected item. Other then this minor difference, *Default-Choice-Sequence-Dialog-Item* is the same as **Sequence-Dialog-Item**.

Icon-Sequence-Dialog-Item and *Image-Sequence-Dialog-Item* have the same functionality, but on different graphical formats. Each class allows the user to scroll through a group of pictures. When the accept button is selected, the current picture is converted into a bitmap. *Icon-Sequence-Dialog-Item* handles 'ICN#' resources and *Image-Sequence-Dialog-Item* uses 'PICT' resources.

The *Function-Popup-Menus* class has a pop-up menu for each function type (i.e. *boolean*, *integer*, *location*, *sound*, and *thing)* in MicroWorld. When the user selects a dialog item in a code dialog that needs a function (e.g. a parameter field or the return value), this class determines the function's type and displays the appropriate menu.

Menu is an interface with the menu manager in the Macintosh Toolbox. An instance of **Menu** is created for each menu in the menubar. When a menu-type event occurs, such as a mouse-down event in the menubar, the appropriate **Menu** instance is sent the event. If an item from a menu is selected, the menu determines which instance of **Menu-Item** (or subclass) that corresponds with the selection and sends the instance a *menu-item-action* message.

Popup-Menu provides the same functionality as **Menu**, except the menu "pops up" at the position of the mouse and not from the menubar. The user's previous selection is the highlighted item and is the default choice.

List-Popup-Menu's items are not instances of **Menu-Item**. Instead, each item is a list. The first element is the display string for the menu. The second element is the item's type. **Menu** and *Popup-Menu* items are instances of **Menu-Item** (or subclass).

Selecting an item from a *List-Popup-Menu* returns the corresponding list. Dynamic pop-up menus are easily created and updated. MicroWorld uses *List-Popup-Menus* for selecting an item from a list in a menu format instead of a dialog format. For example, the list of all available commands is an instance of this class.

The *UW-Method* class makes extensive use of pop-up menus, but the menus are created in the *UWorld* class (see below). Creating dynamic menus for each instance of *UW-Method* is expensive in memory and execution speed. Furthermore, *UWorld* contains the information needed to create the menus. Therefore, *UW-Method* retrieves the menus from *UWorld*, add items (if needed), before activating the dialogs.

An instance of **Menu-Item** is an item in a menu and contains the text of the item, active indicator, keyboard equivalent character, and the processing of the user's selection. In most cases, a message is sent to an object to perform the function corresponding with the menu item. For example, the menu item "New" sends a "create new world" message to the *UWorld-Author*.

UWorld-Menu-Item has an added instance variable that points to the current instance of *UWorld-Author*. The menu item can send messages directly to the *UWorld-Author*. In all other respects, *UWorld-Menu-Item* acts the same as **Menu-Item**.

Movement defines an on-screen movement path for a *Thing*. Currently, the user defines a path as a list of points in X, Y coordinates. Future plans include defining this path by following the movements of the mouse.

The class *Sound* provides an interface with the Macintosh's Sound Manager. An instance of *Sound* is sent a list of 'snd' resource names. The resources are loaded first, before any sound is produced, to minimize the delay between playing each sound resource.

Only one instance of *Sound* is created in the environment for two reasons. First, the information is the same for each instance. The Macintosh's Sound Manager is stored in ROM and each open call returns the same information. Second, the current version of the Sound Manager does not handle several simultaneously active sound channels. Therefore, one instance is created and is stored in the instance of *UWorld*. *Thing's* make requests for the *Sound* object through UWorld.

The class *Thing* defines the objects the student manipulates on the screen. A *Thing* can be clicked, double clicked, or dragged, depending on the setting of instance

variables, which the user can access.

Each *Thing* reacts uniquely to events with respect to other *Things*. Four instance variables contain *Local-UW-Command* objects (see below) which are evaluated when the *Thing* is clicked, double clicked, while it is being dragged, and when it is released.

Two more instance variables are used to record history of events and evaluate the performance of the student. It is impossible to determine the needs of the instructor, therefore, these instance variables contain instances of *Local-UW-Command*.

Each instance of *Thing* has several instances of other classes. The class *Concept* defines the concept of the object. *Movement* defines an on-screen movement path. *UW-Image* and *UW-Icon* defines the images used on the screen.

Things can be grouped together into a composite object. For example, a *Thing* modeling a calculator could be divided into several *Things*, each one is a button. A composite object can work as a unit or as individual *Things*, depending on the programming by the instructor.

Tutor, a subclass of *Thing*, assigns tasks to the student. The *Tutor* does not contain all instances used by *Thing*. The student can not manipulate the *Tutor*, therefore the classes, *Concept*, *Movement*, and *UW-Image* are not used. Two more instance variables were added to specify how and

what to teach.

The class *UW-Bitmap* is responsible for the bitmap images and provides low level graphic support for loading, saving, copying, and testing pixels of an image. Objects that contain instances of *UW-Bitmap* are responsible for drawing the bitmaps.

UW-Background is a subclass of *UW-Bitmap*. Backgrounds are saved in compressed format, therefore, *UW-Background* overrides *UW-Bitmap's* load and save methods to compress and uncompress the bitmap. Furthermore, backgrounds are not pre-loaded into memory to save space. The memory used by the old background is released when a new background is loaded into memory.

The class *UW-Image* uses two instances of *UW-Bitmap* to define a graphical image. One bitmap defines the object's color and the other bitmap defines its mask. If a pixel is "on" (a '1' in a black and white bitmap) in the mask, then the corresponding pixel in the image is drawn to the destination bitmap. If a pixel is off (a '0' in a black and white bitmap), then the destination bitmap remains the same.

UW-Icon, a subclass of *UW-Image*, converts a resource of type "ICN#" and the name of the *Thing* it represents into an image. The icons are similar to the icons used by the Macintosh Finder, except the name can not be edited using the name field. Implementing the Finder-type icons added

complexity to the *Base-World* and *UW-Icon* classes, therefore the Thing's name is changed with a menu item. *Base-World* just selects the object under the cursor with this approach, instead of handling events differently if the mouse event is in the icon's name field

UW-Method (described in greater detail in the Section 6.2) is the top level class for the user's programming interface. It handles the presentation, editing, compiling, loading, and saving of the code.

The two subclasses *UW-Function* and *UW-Command* subdivide *UW-Method* into code that returns a value and code that does not return a value. *UW-Function* displays its definition with an added field for the function, variable, or constant that determine the results.

Instances of *Global-UW-Command* are commands usable by all *Things*. Instances of *Local-UW-Command* are commands local to one *Thing*. Modifications to a *Thing's* local command does not change another *Thing's* local commands.

The five classes, *Boolean-UW-Function*, *Integer-UW-Function Location-UW-Function*, *Sound-UW-Function*, and *Thing-UW-Function* define functions for its return type. For example, an instance of Boolean-UW-Function defines a Boolean-returning function.

When a new method is created, it is either an instance of *Global-UW-Command*, *Boolean-UW-Function*, *Integer-UW-*

Function, *Location-UW-Function*, *Sound-UW-Function*, or *Thing-UW-Function*. *UW-Method*, *UW-Command*, and *UW-Function* are all abstract classes. *Local-UW-Command* are created automatically when a new *Thing* is created.

Figure 42 shows the structure between *UWorld*, *Base-World*, and *Thing* objects. The thick arrows shows the original design and the thin arrows were added later for better communications between *Things*. At first, the *Base-World* provided the communication link between two *Things*. When a *Thing* needed to send a message to another *Thing* the *Base-World* was given the name of the second *Thing* and it would return the object.

Under the original design, *Things* could not communicate between different *Base-Worlds*. Since the *Tutor* exists in a different *Base-World* from the visible *Things*, the *Tutor* could not direct the actions of the *Things*. Therefore, the *UWorld* handles both the *Base-Worlds* and the communication between the *Things*.

Not only do all *Things* need to communicate with each other, but they also need access to the global commands and functions. *UWorld* keeps a table of the *Global-UW-Command* and *UW-Function* objects. Furthermore, instances of *Function-Popup-Menus*, *Popup-Menu*, *List-Popup-Menu*, and *Sound* are kept in *UWorld* for efficient use of memory. *UWorld* contains the "global" knowledge for the environment.

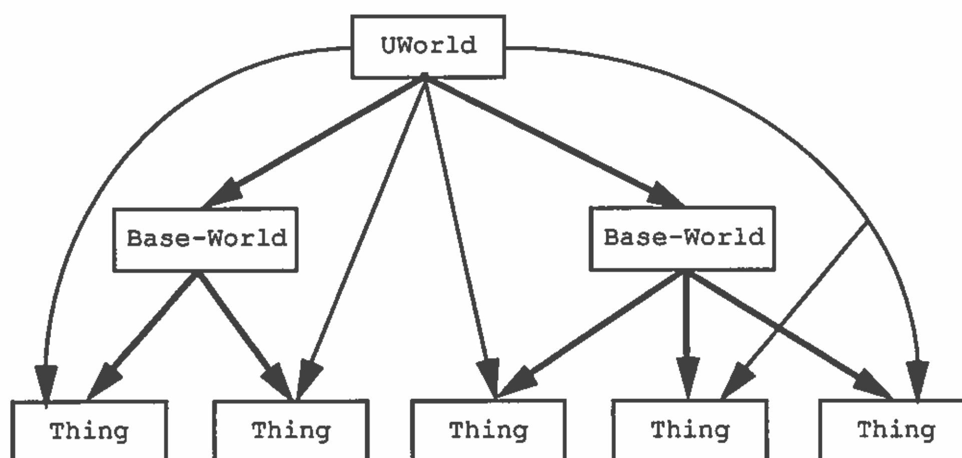

Figure 42. UWorld's and Base-Worlds' Pointers to Things

Currently, *UWorld* contains two *Base-Worlds*, called the World and the Aether. The World contains the objects of manipulation by the student and the backgrounds. It is the only visible *Base-World* when the student is running the lesson. The Aether contains the objects invisible to the student (i.e. the *Tutor*). The Aether provides the means for the instructor to manipulate the invisible objects when defining the lesson.

The *UWorld-Author* class provides the interface between the menu items and the *UWorld* class. A menu selection by the instructor executes the corresponding method in *UWorld-Author*, which send messages to the *UWorld* object. *UWorld-Author* provides the support for MicroWorld's authoring mode.

Another class will provide the support for MicroWorld's run mode and is responsible for the sequence of movies, sounds, and worlds defined by the instructor. For example, a possible sequence is an introductory speech, a movie, a second speech, another movie, and then a lesson. The lack of a foreign function interface in Allegro Common Lisp delayed the definition of this class.

The **Window** class provides the interface with both the Macintosh's window and event managers. An event sends a message to the instance of **Window** associated with the front window. A Macintosh window is divide into a title bar (with optional close and zoom boxes), a content region, and an optional grow region. **Window** handle events directed at the title bar and grow region. Events sent to the content region are the subclass responsibility.

Base-World, as mentioned earlier, is the subclass of **Window**, which handles the windows for *UWorld*. When a *Base-World* receives an event, a search from the highest to the lowest plane is made of the objects it contains. If two objects overlap and the cursor is over the area of intersection, then the top item is selected. If an object is not found, then the event is ignored.

Base-World forwards the event to the selected object for processing, except when the user is dragging the object. For efficiency, *Base-World* handles the actual movement of the ob-

ject. As the object moves, it receives "dragging events."

5.5 MicroWorld's Control Structure

At the core of any Macintosh application is an event loop that retrieves events from the operating system and starts processing it. The event loop is contained within Allegro Common Lisp environment and is not defined as a separate class. Instead, the event is sent to different objects depending on the cursor's position and the event type. Events associated with windows are sent to the active instance of **Window** (or subclass). Events directed at the menubar are sent to the appropriate **Menu** (or subclass) object.

Two approaches for event processing are possible in an object-oriented system. One centralized object could process all events (global control) or individual objects could be responsible for events directed at them (local control).

The difference between global and local control can be shown by using a mouse click as an example. Using global control, the centralized object sends a "highlight" message to the *Thing*. Using local control, a "click event" message is sent to the *Thing* and it decides to highlight itself. A different type of *Thing* could decide to behave differently for the same event.

MicroWorld uses local control as much as possible. *Thing* decide how to handle all the events it receives. For example, a mouse click sends either an *authoring-click-action* or a *perform-1-click-action* message, depending on if the *Base-World* is in authoring or running mode.

As mentioned above, the actual event messages *Things* receive depends on the mode of MicroWorld. In authoring mode, the methods *authoring-click-action* and *authoring-unclick-action* are used when the *Thing* is clicked and when another *Thing* is clicked. Dragging and landing events are not sent to the *Thing*, instead they are handled directly by *Base-World*. Furthermore, a double-click event is ignored. *Authoring-click-action* highlights the *Thing* and *authoring-un-click-action* unhighlights it.

In authoring mode, a mouse click selects the *Thing*. Holding the shift key during the mouse click adds the *Thing* to the list of selected *Things*. All the list items are affected by the commands of the "Thing" menu, with the exception of the command "switch planes".

In run mode, *Things* are sent the messages *perform-1-click-action*, *perform-2-click-action*, *dragging-action*, and *landing-action* for the mouse click event, mouse double click event, during the *Things* movement by the user, and when the *Thing* is released. Each method evaluates a different instance variable, which contains a *Local-UW-Command* object.

For example, the *perform-1-click-action* method evaluates the *1-click-action* instance variable.

Two instance variables, *history-action* and *evaluation-action*, are used in run mode. Each contains an instance of *Local-UW-Command* and is programmable by the instructor. The *Tutor* plans the student's next task by using *history-action* and *evaluation-action*. The *Tutor* uses *history-action* and *evaluation-action* to plan the student's next assignment.

The *Tutor's* "style" dictates the lesson's content and when it terminates. For example, in exploratory style, the *Tutor* waits until the student selects "Quit" from the "World" menu. Control resides with the *Thing* objects when the *Tutor's* style is exploratory.

The coaching or teaching styles give the *Tutor* more control over the environment. An evaluation of the student's performance determines the next assignment. Possible new assignments include repeating the previous assignment, teaching a new concept, teaching a concept related to the current concept, or exiting the current lesson.

Another class (not implemented) supports control between lessons. It regains control when the current *Tutor* terminates the lesson and activates the next phase, which includes starting another lesson, showing a movie, generating sounds, or exiting the program. The sequence can consist of any combinations of the first three items.

In general, two control levels exist in MicroWorld's "authoring" mode. *UWorld-Author* handles the menu commands and *Base-World* handles the other events. Three control levels exist in MicroWorld's "run" mode. *Things* handle the events, the *Tutor* handles the lessons, and the class mentioned above is responsible for the sequences of lessons, movies, and sounds.

5.6 User's Manipulation of Objects

In the original design of MicroWorld, pop-up menus were the primary input for manipulating objects. The pop-up menus contained only the commands appropriate to the selected object. For example, a pop-up menu contains the "change background" when the background is selected.

Unfortunately, the Macintosh did not support pop-up menus when this section of the user's interface was written. Instead, the Macintosh's menubar was used. Pop-up menus became available later, therefore, the Thing's programming interface makes extensive use of them (see Section 5.8).

The Macintosh's menubar user interface is poorly designed. It contains menu items whose contexts are outside the application, global to the application, global to the windows, and local to the front window. The "Apple" menu runs Desk Accessories and selects between different applications currently running under MultiFinder. Selecting

an "Apple" menu item places the current application into background processing. The application regains control only by choice of the user. "Apple" menu items are outside the application.

The "World" menu include commands to save, print, and exit the world, which are are global to the application. The "Window" menu activates the selected window, therefore, it is global to all windows.

The commands from the "Tutor" menu and the "Thing" menu, except for the "Create" command, are local to the active window. The "Create" makes a new item in the appropriate window, which does not necessarily affect the active window. Therefore, the "Create" command is global to the windows.

Mixing global and local commands in the same menubar makes the user's interface less clean and possibly more confusing to the user. A better approach is placing commands global to the application in the menubar and pop-up menus for all other commands. The global commands are selected outside the windows and local commands are selected in the window affected by the command.

The local commands are further subdivided into commands local to the window and commands local to an item within the window. The menu appears in the context that it effects. For example, the pop-up menu displayed when the user selects a *Thing* only affects the selected *Thing*.

The *UWorld-Author* class handles the menubar. The structure is similar between the class and the menubar. All menu items work through *UWorld-Author* to perform the commands, no matter if the commands are global to the application, global to the windows, or local to the windows.

A menu selection executes the function stored in an instance variable defined in the **Menu** class. In MicroWorld, the function sends a message to the *UWorld-Author* object. The actual processing of the menu selection can occur in an object far removed from the *UWorld-Author* object.

For example, the "Create" command in the "Thing" menu sends a message from the *UWorld-Author* object to the *UWorld* object, which sends a message to the appropriate *Base-World* and a new *Thing* is created. Selecting the "Set Position" command from the same menu sends a message from *UWorld-Author* to *UWorld*, which requests the *Thing* selection list from the appropriate *Base-World* and then sends a *set-position* message to each *Thing*. As these examples show, separating the object that first receives the menu selection and the object which processes the selection leads to a confusing mess.

In the ideal design, no separation exists between the object that receives the menu selection and the object that processes it. Furthermore, the position of the cursor gives the context in which the commands are performed. Pop-up menus vastly improve MicroWorld's user-interface design.

The menubar is now outside the context of the application and contains the "Apple" menu. The user either switches between applications or runs desk accessories from the "Apple" menu. Neither alternative directly affects MicroWorld, which stays suspended until it becomes the front-most application. The "Apple" menu is outside MicroWorld's context and, thus, resides in the menubar.

UWorld combines the "World" and "Window" menus into a single menu in the menubar. The commands found in these menus are global to the application and windows. *UWorld* has control over the *Base-Worlds* and all other objects in the environment, therefore, it is the ideal class for the new pop-up menu. Although the menu commands are not global to the application, conflicts with Multifinder[4] requires the menu to be a part of the menubar.

A *Base-World* is responsible for creating objects which belong to it, setting its background, and exchanging planes between two of its objects. The two currently defined *Base-Worlds* have different needs. The "aether" *Base-World* has neither background nor objects on planes[5], therefore, the only menu selection creates a new *Tutor*. The "world" *Base-World's* menu contains three items to create new *Things*, se-

4. A mouse click outside the application's window causes Multifinder to switch to a different application.

5. Not strictly true. Each object created in the "aether" *Base-World* is assigned a plane number, but the objects are invisible to the student. Therefore, it doesn't make sense to switch between the planes.

lect backgrounds, and exchange planes between two *Things*.

The *Thing* and *Tutor* classes each contain the menu items that directly affects their instances. A menu item selection executes the command on the object beneath the cursor. sends a message to the object beneath the cursor. The *Thing* class' pop-up menu contains the entries from the original "Thing" menu, except for the "Create" and "Exchange Planes" commands. Likewise, the *Tutor* class' pop-up menu contains the entries from the "Tutor" menu, except for the "Create" command.

This design is a large improvement over the current design, both from the standpoint of the user's interface and the object oriented approach. The cursor's position determines the available menu and the context for its commands. For example, the menu whose commands affect instances of the Thing class is only available when the cursor is above one of the instances.

A class handles the menu that affects only instances created from it or its subclasses. Control becomes more localized, since the object now controls the interaction with the user. The object determines the conditions necessary to activate the menu and handles the menu selection. Currently, the object indirectly control the menus by sending messages to the *UWorld-Author* object.

As mentioned when designing the pop-up menu structure above, the "world" *Base-World* is different from the "aether"

Base-World, therefore, they should be separate classes. The methods and instance variables for handling the backgrounds and planes, the methods for spatial relationships (e.g. things-to-the-left-of) and the methods for autodragging objects are not needed by "aether" *Base-World*. Furthermore, *Base-World's* method for dragging *Things* can be simplified. A better design has two subclasses of *Base-World*, one is the current "world" *Base-World*, the other is the "aether" *Base-World*.

5.7 Providing Multiple Worlds

More structural changes to the *UWorld* class are needed if windows are increased in MicroWorld. Instead of two instance variables for the "world" and "aether" *Base-World*, an instance variable would contain the window list. When a session continues, the same windows must exists as existed at the end of the previous session. Therefore, *UWorld* must save the number of existing windows and their titles in addition to sending the save message to each window.

The cleanest method for saving arbitrary number of objects is to save the information under the same type and with an unique name, using the resource manager provided by the Macintosh Toolbox. The window's title determines the name of its resource. To restore the windows, each window resource is loaded and the resource's name becomes the

window's title. *UWorld* does not remember the number of windows between sessions. Section 6.3 gives greater details about loading and saving worlds using the resource manager.

In a multi-world environment, *Things* can communicate with other *Things* across the world boundaries, because *UWorld* binds the *Thing's* name with its object and *UWorld* has a pointer to all *Things* in the environment. For example, "Fred" in "World One" can move "Julie" in "World Two" by sending the message, "move Julie to the left of Sam." This raises an interesting question of what does it really mean to have multiple windows in MicroWorld? Conceptually, is a window considered a separate "world" or is it a separate view into the same world? Under the current design, a window is a different view of the same world, because *Things* are allowed to refer to each other across window boundaries.

According to MicroWorld's metaphor, *Things* should only communicate within the boundaries of the "world." This implies that the *Base-World* class handles the references between Things. The difficulty with this approach is handling the reference to the *Tutor* as it resides in a different *Base-World*.

Two possible solutions are to include a pointer to the *Tutor* in each *Base-World* or *UWorld* handles references to the *Tutor*. The latter solution is superior for two reasons. First, it alleviates the need to propagate changes to several

objects, if the *Tutor* is replaced. The *UWorld* and *Tutor* objects communicate using the "aether" *Base-World* as mediatory. The first solution requires finding each *Base-World* that referenced the *Tutor* to update the pointer.

Second, *Base-World* can send unresolved references to *UWorld*, therefore allowing the creation of specialized object in the "aether" window without modifying either the *Base-World* or *UWorld* classes. Suppose, for example, a special help Tutor was created and a Thing tries to communicate with it. The Thing's Base-World can not find the new Tutor and forwards the request to its UWorld, which asks the "aether" *Base-World* for the Tutor and the connection between the *Thing* and the new *Tutor* is made. Creating a new type of *Tutor* only requires modifications to the "aether" *Base-World*.

5.8 Object Programming

The design of the user's interface for programming an object is vastly improved over the interface to manipulate the objects, but not without problems. MicroWorld makes extensive use of dialog boxes and pop-up menus for displaying and modifying user's programs. Creating a new dialog box places it on top and slightly below the previous dialog box, thus, giving an indication of the current code depth. Dialog boxes are created by expanding a line of code, asks for a method's definition, etc.

Once created, a dialog box stays active until the user creates a new dialog box on top of the current dialog box or until the user closes the current dialog box. Although the Macintosh Toolbox supports it, the user can not switch between dialog boxes. The potential for confusion is too great and, therefore, the user can not make a dialog box active without closing the boxes created later. Possible areas of confusion are:

- more than one dialog box opened on the same line of code.
- dialog boxes opened on different lines of code with the same appearance.
- dialog boxes closed between the original dialog box and the current box.
- accepting changes at a dialog with dialogs, created by the current dialog, still open.

The first three areas illustrate problems with determining the current position in the code. For example, if two dialog boxes contain code that looks the same, the user can not determine whether the code is the same, different, nor, if different, where the code belongs with respect to the method definitions. If the user changes one of the dialog boxes, the user does not know which method was changed until the change is accepted and the other dialogs are examined for the occurrences of the change.

If dialogs are closed along the path from the method definition (i.e. the first dialog) to the current dialog, then the presented code may not be visible in the other dialogs. For example, the "if" statement has a "then" code

segment and an "else" code segment. Unless both the test condition and the "then" code segment are very short, the "else" code segment is not visible when the "if" statement is presented on one line. If the user doesn't remember where the code is from in the current dialog, then an undesirable search is required to find the position again.

One solution to the problem is to provide feedback with the dialog box as to the current position in the code. Unfortunately, the further in depth the user goes (i.e. the more dialogs that are opened by expanding the previous dialog), the more information is needed to indicate the position. For example, if line numbers and positions within the line indicates the current position, several numbers may be needed for a large depth and will confuse the user.

When the user accepts a change in a dialog box, then all changes made from the dialog box are accepted, including changes made from dialog boxes opened by the current dialog box. If the dialog boxes are still open, the user can later cancel the changes that were already accepted. The user could be trying to cancel all changes made while the dialog was open, which is contrary to the previous acceptance of the changes, or the user could be trying to cancel the changes made since the previous accept command.

A solution to this problem restricts the user from closing a dialog until all dialogs created by this dialog are

closed. Unfortunately, this requires that dialogs know about other dialogs and their current states. Given the difficulties described above, it was decided to only allow the user to make the previous dialog active by closing the current dialog.

This solution isn't without problems. Currently, the user must close the current dialog to view the previous dialog's contents. The user may want to review a previous dialog while making changes to the current dialog, without accepting or canceling the changes. A possible solution has the dialog box becomes visible for a short period of time. For example, placing the cursor over the dialog box and pressing the mouse button makes the dialog visible until the button is released.

Pop-up menus are used to present choices to the user, depending on the location of the mouse when the button is pressed. Each instance of **Dialog-Item** (or subclass) that uses a pop-up menu contains the instance(s) of either *Popup-Menu*, *List-Popup-Menu*, or *Function-Popup-Menu*. When a **Dialog-Item** receives a mouse down event, the **Dialog** sends a *dialog-item-click-event-handler* message to the **Dialog-Item** and it displays the menu.

Each pop-up menu points to the instance of UW-Method (or subclass) currently in use by the instructor. A menu selection sends an appropriate message to the object contained

within the menu's instance variable. The dialog box regains control after executing the command.

Originally, *UW-Method* contained an active dialog stack. If the code changes, a message is sent to the top **Dialog** item with the name of the changed **Dialog-Item** and the update information. The **Dialog** searches the **Dialog-Item** list and forwards the information to the item it finds.

Each object has instance variables for the objects it communicates with directly. *UW-Method* has instance variables for the **Dialogs**, which have instance variables for *Popup-Menus*, *List-Popup-Menus*, and *Function-Popup-Menus*. Furthermore, each **Menu** subclass has an instance variable for the current *UW-Method*. UW-Method creates and updates the Dialogs, which are responsible for "popping up" the menus, and the menus send the user's choice to the UW-Method.

The design achieves the minimum number of connections between objects, but does not completely work. *UW-Method* can not determine which section needs modification, when two or more section of the same type exists within the same dialog box. For example, the Lisp function 'cond' has two areas to insert, delete, or expand a command, the section for the 'then' part and the section for the 'else' part. The method *add-method* (defined in *UW-Method*) does not know which section to add a new command, since it only receives the command and the commands position.

Providing more information in the **Dialog-Item** solves the problem. Upon creation, the **Dialog-Item** receives a type and an ID number, which is based on the order it was created within the group of **Dialog-Items**. Using the previous example, both *Code-Sequence-Dialog-Item* objects receive the same type (:commands-without-first) with different ID numbers. *UW-Method* requests the ID number to determine the section that needs modification.

UW-Method must communicate with the **Dialog-Item** that activated the pop-up menu for the correct ID number. Two possible approaches are to pass the **Dialog-Item** object with the pop-up menu message to *UW-Method* or the **Dialog** object saves the currently selected **Dialog-Item**. The first approach allows the *UW-Method* object to directly communicate with the **Dialog-Item** object, although the pop-up menu has unused information passed to it. The second approach forces the *UW-Method* object to use the **Dialog** object as intermediary, but useless information is not passed between the objects.

The first approach was used in MicroWorld, because of the frequency of use for the **Dialog-Item** in some of the methods. Upon further reflection, the second approach is better. The methods that make frequent use of the **Dialog-Item** object frequently should request the object from the **Dialog** object. Methods that seldom communicate with the

Dialog-Item objects should use the **Dialog** objects as intermediary.

5.9 Conclusion

MicroWorld's design is still evolving. For example, *Thing's* action instance variables for processing events are now contained within a single class. Also, user-defined instance variables are contained within their own class. Hopefully, the suggestions outlined above will be implemented in the near future.

CHAPTER 6

MICROWORLD'S IMPLEMENTATION

This chapter describes the major sections of MicroWorld's implementation. It is written for the people who are working with MicroWorld's source code, therefore it probably isn't interesting to the general public. The chapter assumes both a knowledge of Common Lisp and the Macintosh Toolbox, although it isn't required.

MicroWorld's first major version is near completion. The Tutor is incomplete and is the hardest class to design and implement. Therefore, completion of the first version may require large amount of time Furthermore, MicroWorld needs extensive testing and debugging before it is ready for use.

The implementation needs another major revision. Some methods are not well designed nor well written due to time constraints and poor design decisions. The classes used for programming *Thing's* responses and the *Tutor* class require the most work.

This chapter gives an overview of MicroWorld's implementation, which is still being written and debugged. Therefore, the description is not of the most current

implementation. However, examining the implementation, both past and present, provides useful information about the object oriented approach, the usefulness of different programming languages, the problems and areas of difficulties in the project, and improvements to the project.

6.1 User Interface Implementation

MicroWorld's user interface consist of three items, dialogs, menus, and windows. Allegro Common Lisp provides an interface between its class system and the Macintosh Toolbox (outlined in Chapter 5). Subclasses are used to specialize the characteristics of dialogs, menus, and windows.

6.1.1 Dialogs

Dialogs present information to the user and allows selection and/or modification of the information. MicroWorld uses dialogs in several areas, including retrieving file names, selecting a *Thing* object's image and icon bitmaps, and editing or viewing the user-defined methods. The next four sections outline MicroWorld's dialog types.

6.1.1.1 File List

Allegro Common Lisp provides two functions, *choose-file-dialog* and *choose-new-file-dialog*, which creates two file list dialogs using the Macintosh's Standard File Package.

Choose-file-dialog presents a list of available files of a specified file type and creator. Selecting a file and pressing the "open" button returns the file's name and its directory identification number.

Choose-new-file-dialog saves a file under a new name. For example, if the user selects "Save As..." from the File menu, *choose-new-file-dialog* displays the current directory and the default file name. From this dialog, the user can change the current director and the file's name. Upon pressing the "Save" button, *choose-new-file-dialog* tests the current directory for a file with the same name. The user must confirm writing over an existing file. Similar to *choose-file-dialog*, this function returns the file's name and directory id. Both functions use the Lisp statement, *(throw :cancel)*, when the user presses the "Cancel" button.

MicroWorld uses two file types, 'µWSV' and 'LEXI'. The 'µWSV' type is the main resource file, which contains a complete world's definition. The 'LEXI' type is the sound resource (or lexicon) file. The sound resources are separate from the main resource file, therefore, a lexicon file can be shared between several worlds. Both files' creators are the 'µWRD' type.

6.1.1.2 Image Dialog

Any Macintosh Paint program can create images for MicroWorld by copying the images to the scrapbook file. These images are saved using the *PICT* format, which consist of several Quickdraw commands that reconstruct the images. The image dialog uses the *PICT* format for displaying the images, but MicroWorld converts them to bitmaps before their use in the environment.

The dialog uses two regions for displaying the final image's components. The first region contains the "image," which defines the final image's color (black or white). The second region contains the "mask," which defines the final image's shape. The "image," "mask," and destination (normally the screen) bit patterns determines the resulting bit pattern using the formula in Table 3. Each region is an *Image-Sequence-Dialog-Item* object.

The dialog keeps a list of all *PICT* handles contained within the scrapbook file or another file selected by the user. When a scroll bar position changes, the appropriate *Image-Sequence-Dialog-Item* object receives a *draw-cell-contents* message, which includes the new image's list index. *Image-Sequence-Dialog-Item* uses this index to retrieve the picture's handle and draws the picture using the Toolbox call DrawPicture.

Table 3. Results Based on Image, Mask, and Destination Bitmaps

Image	Mask	Destination	Results
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

The dialog requests the current picture handle from each *Image-Sequence-Dialog-Item* when the user presses the "Accept" button. The unused picture handles are released to reclaim memory. The dialog returns the two picture handles, which are converted to bitmaps using a temporary, off-screen graf-port.

6.1.1.3 Icon Dialog

The Icon and Image dialogs uses similar techniques. MicroWorld provides a set of predefined icons. The user can not add to this set. The icon resource contains its image and mask, therefore, the Icon Dialog uses one region to display the icon.

The *ICN#* and *PICT* formats are different. The *ICN#* format contains two 32 x 32 bit patterns instead of the *PICT's*

Quickdraw commands format. The first bit pattern is the icon's "image" and the second bit pattern is its "mask." *Icon-Sequence-Dialog-Item* uses two temporary bitmap structures to draw the icons. These structures are shared between all MicroWorld's icons, instead of converting each icon to a bitmap. The temporary bitmap's base addresses are set to the dereferenced *ICN#* handle for the icon's image and the dereferenced handle plus 128 for the icon's mask. Switching between icons only requires updating the two base addresses.

Unlike the Image Dialog, Icon Dialog returns the icon's resource number when the user presses the "Accept" button. Several *Thing* objects can use the same icon, therefore, less disk space is used by saving the icon's id versus saving its bitmap representation.

A *Thing's* icon representation contains both the icon selected by the above method and its name. Therefore, the two sub-components are combined before displayed on the screen. A bounding rectangle is created, which contains both the icon and the name. The name's size is calculated by its length and the current font size. The smaller length sub-component is horizontally centered above or below the longer sub-component.

6.1.1.4 Code Dialog

Code-Dialog, a subclass of **Dialog**, is used for viewing and editing user-defined local and global actions. *Code-Dialog* uses "null events[6]" to track mouse positions, whereas, **Dialog** uses the same event to flash the active text field's caret. *Code-Dialog* keeps a list of objects (subclasses of **Dialog-Item**), which process null events. A *dialog-item-null-event-handler* message, which includes the current mouse position, is sent to each list item. One dialog item can not block another item from receiving null events. *Code-Dialog* does not process null events besides forwarding the message to each list item. Further processing is handled by the dialog items.

Normally, the item list is composed of *Code-Sequence-Dialog-Items*, which uses null events to highlight its list item currently under the cursor. *Code-Sequence-Dialog-Item* keeps the previous highlighted item to unhighlight it, if the cursor moves to a new item or moves outside the *Code-Sequence-Dialog-Item's* bounds. Interrupts are suspended until the *dialog-item-null-event-handler's* method completes, otherwise, another null event can occur before the items are correctly updated, causing incorrectly highlighted items.

---

6. The Macintosh's Operating System returns "null events" when *GetNextEvent* is called and no other event types are available.

Pressing the mouse button activates a pop-up menu, whose commands affect the currently selected item. Two factors determine which pop-up menu is activated, the current active dialog and whether the user is editing or viewing the code. These factors are outlined below.

Two dialog types are used in the presentation of user-defined code. The first dialog type displays method definitions. The second dialog type expands a single code statement into the method's name and its parameter list. The pop-up menus contain different commands for each dialog type.

When editing code, a blank line exists between each code statement, which allows the addition of new code statements from the pop-up menu created when the user presses the mouse button. Selecting an existing code statement activates a pop-up menu, which either expands or deletes the statement. Expanding a code statement creates a new dialog box for editing the statement. When viewing code, each lines contains an existing code statement and the pop-up menu contains a single command to expand the statement.

Code-Sequence-Dialog-Item does not contain an instance variable for the current mode (either editing or viewing). Instead, two instance variables contain pop-up menus. One menu activates when the user selects an even-numbered line (blank line in editing mode) and the other menu activates for odd-numbered lines (existing code statement lines). When ed-

iting code, the two instance variables contain different pop-up menus. When viewing code, the instance variables contain the same pop-up menu. *Code-Dialog* sets these instance variables based upon its mode.

6.1.2 Menus

Two Allegro Common Lisp classes, **Menu** and **Menu-Item** provide an interface to the Macintosh's Menu Manager. A menubar entry is an instance of **Menu** (or subclass) object and each item contained within a menu is an instance of **Menu-Item** (or subclass). A **Menu-Item** has two components, the item's name and a Lisp expression, which is evaluated when the user selects the item. MicroWorld added two methods, which activates and deactivates a list of menu items, to the standard **Menu** class. Furthermore, *UWorld-Menu-Item* (a subclass of **Menu-Item**) contains an instance variable, which points to the object that receives the menu selection's message. Besides these two minor modifications, MicroWorld's and Allegro's menus behave the same.

6.1.2.1 <u>Pop-up Menu</u>

On the other hand, Allegro Common Lisp does not support pop-up menus, therefore, the *Popup-Menu* class was created. *Popup-Menu* uses the NewMenu Toolbox call to create an empty menu structure, from which the menu is built dynamically.

The Macintosh's menu structure contains a pointer[7] to the *Popup-Menu* object associated with it, therefore, a random menu identification number is created and checked against the existing menu entries. A non-object oriented language (e.g. C or Pascal) uses this id number to determine which menu was selected.

The Macintosh's Menu Manager builds menus from strings, therefore, the menu item's names are combined into a Pascal string. Meta-characters can be appended to the menu items' name, which provides additional information to the AppendMenu Toolbox call. The meta-characters are used to set the command key equivalent character, enable flag, icon information, mark character, and character style for a menu item. *Popup-Menu* requests the previous information from each **Menu-Item** (or subclass) it contains and appends the appropriate meta-character to the item's name before adding the item to the Pascal string. For example, the string to define "Copy" with the keyboard-equivalence command-C is "Copy/C."

Instead of a **Menu-Item** object list, *List-Popup-Menu* takes a list consisting of menu's names and a Lisp symbol associated with each name. When selecting a menu item from *Popup-Menu*, the item's *menu-item-action* is evaluated. On the other hand, *List-Popup-Menu* returns the selection instead of

7. Actually, the menu structure contains a generic programmer-defined long integer called *RefCon*. Allegro object packages stores the object pointer in *RefCon* and later accesses it when a menu item is selected.

sending a message to the individual menu item, which is desirable when selecting an item from a list. For example, *List-Popup-Menu* handles the list of available fonts better than *Popup-Menu*. The Lisp symbol's defines the individual menu item's type.

6.1.3 Windows

Base-World, a subclass of **Window**, controls MicroWorld's windows. *Base-World* keeps an array of *Things* contained within its boundaries. The *Thing's* plane number provides an index into this array. When drawing the window, the background is drawn first and then all visible Things are drawn from the lowest to the highest plane, which overlaps the images correctly. An offscreen bitmap is used to compose the final image, before drawing it on the screen, to reduces flicker.

Base-World shadows three **Window** methods, *window-event*, *window-click-event-handler*, and *window-draw-contents*. Allegro's event manager sends the front window a *window-event* message for each event occurrence. The standard *window-event* method determines the event type and then sends itself the appropriate message. The front window receives the *window-click-event-handler* message from the standard *window-event* method during a mouse down event. *Window-draw-contents* draws the window's contents and is called for several reasons, in-

cluding window update events.

Base-World's window-event method determines if the cursor is in the window's grow box and the user is pressing the mouse button. If the cursor is not within the grow box's bounds, the method forwards the event to its superclass' *window-event* method. *Base-World's window-event* method also forwards the event to its superclass' method when the cursor is within the grow box, but it resizes *Base-World's* bitmaps upon return.

Base-World contains five window-sized bitmaps, which are used for temporary images while the user moves a screen object. When the window's size changes, *Base-World* deallocates memory the temporary bitmaps use and allocates memory for them at the new window size. Furthermore, the method reloads the background from storage and draws it to the background bitmap. Drawing the previous background's image contained in memory cause progressively worse images as error were compounded each time the window size was changed.

Changing the window's size is slow, but the alternative uses screen-sized bitmaps, which has a high memory cost, especially with the large monitors and eight-bit planes (or higher) color systems available today. Resizing the window is not a frequent operation, so the delay is more acceptable than the memory cost.

Base-World's window-click-event-handler method first determines which object[8], if any, is selected. The method checks the object array from the highest to the lowest plane number. Each object's mask is checked at the mouse's position when the button was pressed. The mask defines the object's shape. A value of one at the mouse's location in the mask's bit pattern selects the object. Performing the test from the highest to the lowest plane selects the top object when the images are stacked, which is the desired behavior.

When the user selects an object, *window-click-event-handler* updates three bitmaps in case the user moves the selected object. First, the objects are divided into two groups. One group contains objects on a higher plane than the selected object. The other group contains objects on a lower plane. The first group is drawn to two bitmaps, *upper-plane* (contains the images) and *upper-plane-mask* (contains the images' masks). The *lower-plane* bitmap contains both the background and the second object group.

After *window-click-event-handler* updates the three bitmaps, it tests the mouse button's state. If the user releases the button before the cursor moves outside a four-pixel radius, then the object receives a mouse click. Otherwise,

---

8. An object is either an instance of *Thing* or *Tutor*. The "world" contains *Thing* objects and the "aether" contains the *Tutor* object. The "world" and "aether" are both instances of *Base-World*.

the object receives a dragging event (details given below). The four-pixel radius allows the user to slowly click an object, without the intended click becoming a dragging event.

If the object receives a mouse click, *window-click-event-handler* performs another test to determine if the click is part of a double click. The distinction between a single and double click is the delay between the preceding mouse up event and the following mouse down event. In authoring mode, *window-click-event-handler* performs the same function for both a single click and double click events. In running mode, the two click event types perform separate actions.

The *drag-always* method[9] (called from *window-click-event-handler* when dragging an object) performs the following steps each time through a loop until the mouse button is released:

(1) Calculate the update rectangle, which is the union of the moving object's bounds rectangle at the old location and the new location.
(2) Copy the image contained within the update rectangle from the *lower-plane* bitmap to a temporary bitmap, using the CopyBits Toolbox call.
(3) Copy the object's image to the temporary bitmap, using the CopyMask Toolbox call.
(4) Combine the *upper-plane* and *upper-plane-mask* bitmaps and draw it to the temporary-bitmap, using CopyMask and the update rectangle.
(5) Copy the image contained within the update rectangle from the temporary bitmap to the screen bitmap, using CopyBits.

---

9. *Window-click-event-handler* calls *drag-always* when MicroWorld is in authoring mode. On the other hand, *window-click-event-handler* calls *drag* when MicroWorld is in running mode, which checks the object's *draggable?* instance variable before calling *drag-always*.

The update rectangle reduces the amount of data copied between bitmaps. The data outside update rectangle's bounds is correct, therefore, needless copying is avoided by using the update rectangle in steps 2, 4, and 5. The CopyBits commands in step 2 and 5 replaces the destination's bit pattern with the source's bit pattern alleviating the need to erase the bitmaps before making the copy. The above algorithm is memory intensive, but objects move smoothly.

The *window-draw-contents* method draws the window's contents. The method draws all images to an off-screen bitmap and then transfers the final image to the window, which reduces screen flickers. The objects are drawn, in order, from the lowest plane to the highest plane, which correctly overlaps stacked images.

Base-World contains several non-window related methods. *Base-World* performs the *things-to-the-left-of*, *things-to-the-right-of*, *things-above*, and *things-below* methods, which return a list of *Things* within that spatial relation. For example, the message *(things-to-the-left-of a-Thing)* returns each *Thing* in which the statement *(ask Thing (is-left-of? a-Thing)* returns true. Spatial relationships do not exist across window boundaries, therefore, *Base-World* is the logical class to contain these methods.

6.2 Implementation of User's Programming Language

UW-Method is an abstract class used for viewing or editing user-defined methods. Each instance defines a single method and contains the following information:

- Method's name.
- Method's creator type, either :user or :builtin.
- Format for displaying the method to the user.
- Syntax for calling the method either :send or :function.
- Method's explanation.
- Method's source code.

The user can edit or view code with the *:user* creator type. The :builtin type declares the method a MicroWorld primitive. Although the user can not edit or view these primitives, defining an *UW-Method* for them provides a uniform interface between primitives and user-defined code.

The display string provides formatting information about the method's source code and includes both the method's name and parameter types. The method's symbol name can be different from the name visible to the user. For example, the Lisp function 'cond' is presented as "if."

The display string includes the method's parameter types, which marks their positions. Although the actual type isn't necessary, it is included for convenience when loading and saving methods. The parameter types are replaced by the actual parameters' string representations.

For example, the Lisp S-expression:

```
(ask thing1 (move (to-the-left-of thing2)))
```

is displayed as:

move thing1 to the left of thing2.

The "move" command's display string is "move <thing> <location>" and the "to the left of" function's display string is "to the left of <thing>." The thing1 parameter is first converted to a string, then the (to-the-left-of thing2) parameter is converted next. Since it is also a list, a recursive call is made, which returns "to the left of thing2" and the strings are combined to make the final output.

The *convert-form-to-string* method, which converts lisp functions to strings, has six separate cases. If its parameter is an atom, a separate conditional tests for the Lisp symbols 't' (converted to "true") and 'nil' (converted to "false"), otherwise, it converts the symbol's name to a string.

If *convert-form-to-string*'s parameter is a list, it tests the first element for the Lisp symbols 'ask' and 'cond.' If the first element is 'ask, *convert-form-to-string* uses the template *(ask <parameter 1> (<method's symbol> <parameter 2> ...))* to convert the expression to a string. If the first element is 'cond,' it uses the template *(cond ((<boolean test>) <then statement>) (t <else statement>))*. If the first element is neither 'ask' nor 'cond,' *convert-form-to-string* uses the template *(<function's symbol> <parameter 1> <parameter 2>...)*. *Convert-form-to-string* is

recursively called for each *<parameter>*, *<boolean test>*, *<then statement>*, and *<else statement>* found in these templates.

The s-expression *(ask thing1 (move (to-the-left-of thing2)))* demonstrates the need for the :send and :function constants, which indicates the method's calling format. Both the *move* command and the to-the-left-of function are methods defined in the *Thing* class. The *to-the-left-of* function uses both *Things* to determine the second *Thing's* prototypical location with respects to the first *Thing*. Object Lisp evaluates functions in the receiving object's environment, therefore, formatting *to-the-left-of* as a function, instead of a message, causes the evaluation to occur within the first *Thing's* environment, which is the desired behavior.

UW-Method keeps the method's source code in the list structure shown in Figure 43, which is very similar to the Lisp syntax's for function definitions. Some modifications are necessary to compile the source code, which depends on whether the action is local or global. If the action is local, the method's symbol is replaced with 'lambda.' A *Thing's* instance variables points to the local action and the local action is not visible outside the object's boundaries, therefore, the action is not assigned a name. If the action is global, the 'defobfun[10]' symbol is inserted at the list's

---

10. *Defobfun* is Object Lisp's function for method definitions.

```
(<method's symbol> (<parameter list>)
     (let ((.self (self))
                 :
          )
         (<statement 1>)
         (<statement 2>)
               :
         (<final statement>)))
```

Figure 43. Method Definition's List Format

beginning and the method's symbol is replaced with *(<method's symbol> thing)*, which assigns the method to the *Thing* class.

6.2.1 UW-Method's Subclasses

The abstract class *UW-Method* has two subclasses, *UW-Command* (used for commands) and *UW-Function* (used for functions). *UW-Function's* dialog for displaying its source code includes an extra field, which contains the expression that calculates the return value (*<final statement>* in Figure 43). Each class creates its own dialog, although most editing functions occur in the *UW-Method* class. Furthermore, the *Local-UW-Command* (*UW-Command's* subclass) also creates its own dialog, which lacks the method's name and the parameter fields. *UW-Method* activates these dialogs and not the classes which created them, since *UW-Method* handles the source code modifications.

The abstract class *UW-Command* has two subclasses, *Global-UW-Command* and *Local-UW-Command*, which was mentioned

earlier. A *Global-UW-Command* object contains a command that is available to all *Thing* objects A *Thing* object's six user-programmable responses are instances of *Local-UW-Command*, which has neither a method name nor a parameter list. A method name makes the command accessible from other objects, which is not desirable. Furthermore, local actions are activated by messages received from the *Base-World* object, which is outside the user's domain, therefore, local actions can not receive parameters. *Local-UW-Command* creates a different dialog than the dialog created by *UW-Command*, because *Local-UW-Command* lacks a method name and a parameter list.

A *UW-Function* subclass exists for each function return type, which currently are *Boolean-UW-Function*, *Integer-UW-Function*, *Location-UW-Function*, *Sound-UW-Function*, and *Thing-UW-Function*. Each class defines two methods, *my-res-type*, which returns the method's resource type, and *get-action-type*, which returns the function's return type. Two instance variables defined in the *UW-Function* class would provide the same information, but this approach requires other objects to set these values.

6.2.2 Converting Between User's Source Code and Lisp Code

UW-Method converts user's source code into Lisp code as the user makes changes and not during a separate compilation step. Although this approach requires extra work to undo

user modifications, *UW-Method* handles editing and viewing code uniformly.

Two dialog types are used throughout the viewing or editing session. The first dialog type contains the method's formal parameters and definition. The second dialog type contains the method's name and actual parameters of a calling sequence. For example, *UW-Method* uses the first dialog type to edit or view the following function (in C syntax):

```
float foo (int x, float y)
{
    float results;
        :
    results = pow (x, y);
        :
    return (results);
}
```

and it uses the second dialog type to edit or view the pow function call.

When editing source code, the user can perform the following functions using the first dialog type:

- Change the method's name.
- Add formal parameters.
- Delete formal parameters.
- Change formal parameters' names.
- Insert source code.
- Delete source code.
- Expand source code.

When viewing source code, the user can only perform function seven from the same dialog.

Changing the method's name removes the old name and method definition from MicroWorld's environment. To avoid runtime errors, *UW-Method* updates all references found in

other method definitions to the new name. Currently, *UW-Method*, with *UWorld's* help, searches all user-defined global and local actions, therefore, changing the method's name is time consuming.

The user adds a formal parameter by selecting a blank line in the parameter list's display box (see Figure 33). The mouse down event activates a List-Popup-Menu object and the user selects the parameter's type from the menu. After making the selection, the user types the parameter's name.

Adding a parameters modifies three objects, *Code-Sequence-Dialog-Item*, *Function-Popup-Menu*, and *UW-Method*. The parameter display box in the dialog is an *Code-Sequence-Dialog-Item* object and its parameter list is updated with the new parameter's name and type. Furthermore, the parameter is selectable from the *Function-Popup-Menu*, therefore, it updates its menu items. *UW-Method* requires two steps when adding a parameter. First, it updates the display string (see above), the parameter type list (used for finding the parameter's type quickly), and the source code's parameter list (see Figure 43). Secondly, it informs *UWorld* of the parameter change and *UWorld* finds and edits each call to the modified method, which avoids runtime errors.

Deleting a parameter reverses the process outlined above. *Code-Sequence-Dialog-Item* and *Function-Popup-Menu* removes the entry from their lists. *UW-Method* removes the type

from the display string, the parameter type list, and the source code's parameter list. Currently, *UW-Method* does not correctly handle parameter references within the method's source code, which is a major bug. Two possible choices are that the user can not delete a parameter until all references are changed or change the references to the parameter's type needed for the call and the user must change the parameter's type to actual parameters before accepting the changes. *UW-Method* again informs *UWorld* of the parameter deletion and the other methods are updated.

The user can not cancel either operation once it begins for two reason, the original method is not copied and finding the methods that needs restoring is difficult. If the user cancels while editing a modified method, then *UW-Method* must undo all previous changes. Otherwise, some methods will have more parameters in the method call than other methods, which will cause a runtime error.

A possible solution uses a hash table, keyed by the method's name, and each entry contains the method's definition that calls the key method. When the user selects "cancel," *UW-Method* tells each entry to restore itself to its previous state. Unfortunately, this requires a changes stack, since the modified methods can be active in a different edit dialog. Canceling the changes should affect the modifications the added parameter creates and not other

changes that the user has neither accepted nor canceled. Therefore, providing cancellation will potentially use large amounts of memory.

Changing the parameter's name changes the entry in the source code's parameter list. Each parameter reference in the method's body is replaced with the new symbol.. Other changes are not necessary.

To insert a new command into the method's definition, the user starts by selecting the command from a menu that will return the command's name and symbol. The command's definition provides the calling format and the parameter type list, which are needed to generate the correct Lisp code.

If the calling format is :function, then *UW-Method* creates a Lisp expression from the template: *(<command's symbol> <parameter 1> <parameter 2> ...)*. On the other hand, if the calling format is :send, then *UW-Method* creates a Lisp expression from the template: *(ask <parameter 1> (<command's symbol> <parameter 2> ...))*. The new lisp expression is inserted into the method's definition using destructive Lisp operations, which reducing the memory used for list copies. This approach requires that only one pointer exists for each code definition.

Modifying the Lisp structure when the user inserts a new command instead of waiting until the user accepts the changes, allows *UW-Method* to use the same display code for viewing

both the new and existing Lisp expressions. Placing the parameters' types within the new Lisp expression provides the user with visual-feedback of the expected parameter types. Furthermore, *UW-Method* checks for actual parameters in the method's definition by testing for parameters that begin and end with the characters '<' and ">. The user can not accept the method's definition until the parameters are set to appropriate values, which avoid runtime errors.

Deleting a command destructively removes it from the method's source code. Since MicroWorld does not support the undo function, *UW-Method* should receive confirmation from the user before deleting the command. Currently, *UW-Method* deletes the command without confirmation.

Expanding a command or adding a new command creates an instance of the second dialog type, which presents the command's name and each actual parameter on a separate line. The user is allowed to do the following at this level:

- Expand the method's definition.
- Edit actual parameters.
- Expand actual parameters.

Expanding the method's definition creates another instance of the first dialog type, if it is a user-defined method.

The user edits a parameter by selecting an item from the pop-up menu displayed when the user presses the mouse button over the parameter. This pop-up menu contains only the allowed constants, parameters, variables, and functions for

the parameter's type. For example, the boolean pop-up menu contains the constants "true" and "false", the parameters and instance variables of type boolean, and the Boolean-returning functions.

Constants for the <integer>, <location>, and <string> types are entered from the keyboard by the user, since too many possible values exist. *UW-Method* checks the input for the correct value type. The user can not accept the change until the value is the correct type.

In edit mode, a blank exists between each *<locations>* or *<sounds>* list item, when *UW-Method* displays the list to the user. A blank line between each parameter allows the insertion of a new parameter at any list position. The list item's type is the parameter type's singular form, in this case *<location>* and *<sound>*.

Pop-up menus behave differently between the dialog items used for the *<locations>* and *<sounds>* parameter types and normal parameter types. A method or function call can contain many parameter types, therefore, *UW-Method* sets the *Code-Sequence-Dialog-Item's* pop-up menu to a *Function-Popup-Menus* instance, which contains the normal parameter types' pop-up menus. When the user activates the pop-up menu (by pressing the mouse button in an active field), *Function-Pop-up-Menus* requests the parameter's type from *UW-Method* and then displays the menu for that type. Since each item in the

<locations> and *<sounds>* list has the same type, the *Code-Sequence-Dialog-Item's* pop-up menu is set to the appropriate *Function-Popup-Menu* upon the dialog's creation.

Determining the <sound> parameter's value requires special processing. The sound *Function-Popup-Menu* contains the available sound resources, the sound-returning functions, the parameters and instance variables of type sound, and the "new sound" constant, which displays a list of SoundWave™ files. *UW-Method* converts the sound file into the Macintosh's 'snd' resource type and it is saved under a user-defined name. Normally, the name is the same as the sound (e.g. the name "hello" for the sound "hello").

Instantiating a <thing> parameter type also requires special processing. If the user selects a *Thing's* name from the *Function-Popup-Menu*, two modifications are made to the method's source code. First, *UW-Method* declares a local variable in the *let* statement (Figure 43) that holds the *Thing's* object pointer, if the local variable is not already defined. For example, selecting a *Thing* named "bike" creates the local variable declaration:

```
(.bike (find-thing "bike"))
```

The let statement binds the variable '.bike' to bike's object pointer found in *UWorld*. Secondly, each reference to the name "bike" is changed to the local variable '.bike.' With this approach, only one search is made for each *Thing* refer-

enced in the method's definition.

The user can only expand actual parameters that are function calls or the *<locations>* and *<sounds>* parameter types. Expanding a function calls creates another second-typed dialog. Expanding a *<locations>* or *<sounds>* parameter creates a third dialog type instance, which edits or views a list. The user can not expand symbols further, for obvious reasons.

6.2.3 Inserting New Code into the Method's Definition

When the user selects a pop-up menu's item, the menu returns a pointer to the dialog item and, in most cases, a line number, which marks the insertion point. The line number is not needed to change the method's name or expand a method's definition.

UW-Method's two instance variables, *code-level-stack* and *parameter-level-stack*, keep the code's current positions. *Code-level-stack's* top value is the current line number in the method's definition. For example, if the user expands the third code line from the top-level dialog, the *code-level-stack's* top value equals two (both stacks are zero-based).

UW-Method uses a stack for the code level, because the user can display the method's definition again, without closing the first dialog. For example, the user can display another method's definition that contains a call to the first

method. If the user expands the call and then expands the method's definition, two dialogs will contain the same definition. The stack separates the line selection between the two dialogs.

The *parameter-level-stack* keeps the current position within a line. A nil pointer in the *parameter-level-stack* separates the entries in the *code-level-stack*. A nil pointer also indicates an entire line is selected. Determining the current code position involves transversing, in reverse order, each item between the stack's top and the highest nil pointer on the stack. Each list item is applied to the Lisp nthcar function with the nthcar previous results.

The actual item's number depends on whether the list is a message (first list item is the 'ask' symbol) or a function call (first list item is not 'ask'). As stated earlier, the first parameter in a message is the second list item, and the other parameters are the third list item minus the first element.

For example, if the *parameter-level-stack* contains *(2 0 nil ...)*, then the current position is the third item within the first item at the code statement marked by the *code-level-stack's* top value. *Parameter-level-stack's* and *code-level-stack's* items are zero-based.

UW-Method determines the currently active position from the dialog item's type and index number. The dialog item's

types are :commands (command list), :commands-without-first (command list minus the first element), :item (single function), :item-at-first (use only function list's first element), :parameter (parameter list), and :return (calculates function's result). The dialog item's index number is its type's creation order.

The :commands dialog item's type specifies that the item contains a list of commands. The :parameter type specifies that the dialog item contains a function's or command's formal parameters. The :return type specifies that the dialog contains the code that calculates a function's return value. The :item type specifies that the dialog item contains a function's or command's actual parameters. *UW-Method* commonly uses these four dialog item types; the first dialog type uses the first three item types and the second dialog type uses the fourth item type.

UW-Method uses the :commands-without-first and :item-at-first dialog types for displaying the Lisp function 'cond.' The :commands-without-first type is the same as :commands, except the first element is not included. Also, :item-at-first is the same as :item, except the item is the first list element and not the entire list.

MicroWorld uses the Lisp 'cond' function's syntax:

```
(cond (boolean-test command ...)
      (t command ...))
```

for its if-then-else command. The dialog for display this command has three items for the boolean test, the commands to evaluate if the test returns true, and the commands to evaluate if the test returns false. The first dialog item's type is :item-at-first, since the boolean test is the first item of the second list element. Furthermore, the second and third dialog item's type are :commands-without-first, since the commands start at the second list element.

Selecting a pop-up menu's item sends a message to *UW-Method*, which includes the item's name, item's symbol, and the dialog item's selected line number. In most cases, this message provides *UW-Method* with enough information. The dialog item's type and index number are needed when the menu selection involves the method's body or a command.

In the method's body case, if the dialog item's type is :commands, then the line number gives the actual position[11]. If the dialog item's type is :return, then the actual position is the method's body last line, which calculates the function's return value.

Some commands contain one or more command blocks (e.g. repeat-until and if-then-else). Since the line number is local to the dialog item, its index number tells *UW-Method* the correct command block. Furthermore, the dialog item's type :commands and :commands-without-first indicate the

11. When editing, this line number is divided by two to compensate for the blank lines.

starting position within the command block.

As noted earlier, the :commands-without-first and :item-at-first types are used for the Lisp 'cond' function, which was a mistake as it lead to more complex code. Defining a macro that takes a boolean test and two command blocks is a better solution, since it removes the special types.

6.2.4 Propagating Changes Through Active Dialogs

When a change occurs in a dialog, the previous dialogs' information needs updating. *UW-Method* keeps a changes stack, which is a list of flags that are in a one-to-one correspondence with the active dialogs. When activating a dialog, *UW-Method* pushes false onto the stack, which marks the dialog's information as unchanged. If a change occurs, all stack entries are changed to true. When closing a dialog, *UW-Method* checks the stack and updates the previous dialog's information if the top value was true.

This approach causes several updates to occur simultaneously when reactivating the dialog, instead of propagating the changes, as they occur, along the current path[12]. Also, only the dialogs that need updating are marked. Dialogs created later, but not changed, are not updated.

---

12. A path is defined as the open dialogs from the top level definition to the current dialog.

6.2.5 UW-Method's Problems

UW-Method uses destructive list operations, which could cause problems. A large number of user-defined methods could exist within the system and copying a method each time a change occurs will increase the time spent garbage collecting and slow the system. Destructive list operations require that only one pointer points to a method, otherwise, changes occurring in one method will affect other methods.

For example, the original method for copying a *Thing* blindly set the copy's local action instance variables (which contain instances of *Local-UW-Command*) to the original values, which creates two pointers to the same list. Modifying one Thing's local action changes the other *Thing's* action. This problem was easily fixed by using the Lisp function "copy-tree", but it illustrates the potential trouble with destructive list operations.

UW-Method has a major design flaw when changes occurring to a method while the user has the method's definition in an inactive dialog. For example, the user first edits method1's definition, but does not accept or cancel the changes. Later, the user edits method2's definition, which method1 calls, and adds a new formal parameter. *UW-Method* creates a new dialog box for each method that calls method2, which includes method1. Therefore, another dialog contains method1's definition, which raises two questions. First, should the

dialog display the original definition or the definition with unaccepted modifications? If the second dialog displays the original definition, it becomes difficult to combine the changes when the user returns to the first dialog. On the other hand, the two dialog can use the same list structure and changes made to one dialog automatically updates the second dialog.

Second, what is the result if the user cancels method1's first dialog? If the second dialog displays the original definition, canceling just removes the modifications made in the first dialog. On the other hand, if the second dialog displays the unaccepted modifications and the user accepts the second dialog's changes, *UW-Method* does not know that the changes have previously been accept, therefore, canceling causes *UW-Method* to undo what it thought were the changes.

The best solution uses the modified method's definition in the second dialog and accepting the changes within the second dialog also accepts the changes in the first dialog. Currently, *UW-Method* uses the original definition and neither correctly updates the first dialog (and dialog created later) nor marks the changes as accepted in the first dialog.

In general, *UW-Method* naively assumes that changes made when opening a dialog can be undone when the cancel button closes the dialog by simply reversing the list operations. As shown above, this approach does not correctly handle

changes occurring in the dialog's children. Further study is necessary to develop a better approach.

6.3 Information Storage

MicroWorld uses the Macintosh's Resource Manager to save the world's information. The Resource Manager associates a resource type, an index number, and optionally, a name with a programmer-defined data structure and saves the unit as part of a disk file. MicroWorld retrieve information using the resource's type and name, which is the object's name.

MicroWorld uses three resource files, the main-resource file, the added-resource file, and the deleted-resource file, when a world is active.. The main-resource file contains the resources that defines the original world, which is empty when creating a new world. Creating a new object or modifying an existing object writes its resource to the added-resource file. Deleting an object moves its resource to the deleted-resource file.

Saving a modified world copies the resources from the added-resource file to the main-resource file, which overwrites the resources with the same name. Closing a modified world without saving copies the resources from the deleted-resource file to the main-resource file. In either case, the main-resource file contains all the world's current objects and the added-resource and deleted-resource files are re-

moved.

The classes that save information in resources are: *Movement*, *Thing*, *Tutor*, *UW-Bitmap*, *UW-Background*, *UW-Icon*, *UW-Method*, *UWorld*, and *Base-World*. *Movement* defines a path for a *Thing* to follow, which is a list of points. The first resource item is the *Thing's* step size as it moves from one location to the next. The second resource item is the path's size, followed by the path's points. *Movement's* resource type is 'MOVE' and its name is the *Thing's* name whose path the *Movement* defines.

Thing's first four resource entries are its centroid, its plane, and two boolean flags, which indicates that the *Thing* has an image and, if it does, that the image was the current representation. The user can save incomplete worlds and therefore, a *Thing's* might not be defined when it is saved. When restoring the *Thing* into the environment, it tests the first flag to decide whether an *UW-Image* instance should be created or not. The second flag indicates whether the Thing's image or icon is its current visual representation.

The next resource entry is the *Thing's Base-World*, which uses the *Base-World's* name, instead of a boolean flag, to allow for more than two *Base-Worlds* within a single world. Following the *Base-World's* name is the *Thing's* name in which this *Thing* is a subcomponent and a list of *Thing's* names,

which are this *Thing's* subcomponents. After the world is loaded, each *Thing* converts these names into object pointers using *UWorld's find-thing* method. A Thing resource's type is 'ACTR' and its name is the Thing's name.

Another resource type, 'ivar,' saves the instance variables accessible by the user. Currently, *Thing* saves the instance variable's symbol name, string name (used to display the variable to the user) and type as strings, followed by the instance variable's value, the storage of which depends on the variable's type. The <boolean>, <integer>, and <location> types are saved using their Lisp representation. *Thing* saves the <sound> and <thing> types using strings with the <sound> type using the 'snd' resource's name and the <thing> type using the *Thing's* name.

Thing saves the <sounds> and <locations> types as null-terminated list with each element saved as the singular type (<sound> or <location>). Fortunately, Lisp sets a point's high bit to one, which distinguishes it from a null terminator. On the other hand, the Macintosh Toolbox's (0, 0) location and a null terminator have the same value.

MicroWorld's *Tutor* class is not completely designed, and thus, it doesn't save information. Some information the *Tutor* needs to save is how and what to teach, which should be saved as strings to allow easy addition of new teaching methods.

UW-Bitmap saves its bounding rectangle, which defines its size and location, and its bit pattern. The other *UW-Bitmap* instance variables are recalculated from the saved information. *UW-Image* sends *UW-Bitmap* its resource type and name, which allows saving the image and mask under different resource types and under the same resource name. The image bitmap has the 'IMAG' resource type and the mask has the 'MASK' resource type. The two *UW-Bitmaps*, which creates the *Thing's* screen representation, use the same resource name as the *Thing*.

Since MicroWorld provides the icons used in the environment, *UW-Icon* saves only the 'ICN#' resource ID, which is Macintosh's standard resource type for icons, for its icon. When reloading a *UW-Icon*, it retrieves the 'ICN#' resource and converts the resource into an image and mask bitmaps. UW-Icon is saved under the Thing's name and 'µICN' type.

UW-Method's resource name and type depends on the object's subclass. If the object's class is *Local-UW-Command*, the resource's name is the *Thing's* name which created it and its type depends on the local action it performs. Currently *Local-UW-Command* resource's types are '1cla', '2cla', 'drga', 'lnda', 'hsta', and 'evla', which stands for "one click action", "two click action (or double-click action)", "dragging action", "landing action", "history action", and "evaluation action".

UW-Method's other subclasses use the method's name as the resource's name. The resource type depends on the method's return type. Global commands are saved under the 'µCmd' type and the functions are saved under the 'µBFn', 'µIFn', 'µLFn', 'µSFn', and 'µTFn' types, which stands for functions that return <boolean>, <integer>, <location>, <sound>, and <thing> types.

UW-Method saves its *message-send?* flag, display-string, and creator, which is either the constants :builtin or :user. *UW-Method* saves its creator as a string constant to allow for more than two creator types.

The last two entries are the method's explanation string and definition, which is converted from the Lisp's list structure into a string. Since the resulting string's size is arbitrarily long, it is broken into 255 character blocks and saved as Pascal strings and the block's end is null terminated.

UWorld saves the current lexicon file's name and the world and aether *Base-Worlds'* sizes and screen positions. *UWorld* uses the 'lexd' resource type and the resource id equals 128, since only one 'lexd' resource exists and it doesn't require a name. The 'µwnd' resource type is used to save the *Base-Worlds'* sizes and positions with the ID of 128 for the world and 129 for the aether. *UWorld* save this information since it is responsible for creating the *Base-*

Worlds.

6.4 Implementation Conclusion

As noted above, and in chapter five, several problems exist with MicroWorld's design and implementation. The user interface, with the exception of the user's programing environment, needs redesigning. The changes outlined in Chapter 5 provides a cleaner interface than the current design.

The programming environment's user interface and class structure are better designed, but the implementation is poorly done. Due to time constraints, several methods are hastily written and not well thought out. Also, solutions are needed for several problems caused by global changes to user-defined method and support is needed for undoing changes.

MicroWorld's design and implementation needs extensive work. The knowledge gained by implementing MicroWorld's current version will vastly improve the next version.

CHAPTER 7

CONCLUSION AND FURTHER WORK

Developments in Computer Aided Instruction are far behind other Computer Science areas for two reasons. First, instructors lack necessary tools for building lessons. Second, defining what makes a good instructor is very difficult, which makes developing a program that imitates an instructor equally difficult. MicroWorld attempts to solve both these problems.

The usual lesson building process has two entities, the instructor and the programmer, and, in most cases, the two entities are separate people, which lead to communication problems, misconceptions about the system's operation, and long turnaround time. Usually, the programmer and instructor have different areas and levels of expertise.

MicroWorld provides an environment in which the instructor programs the lesson, which has two major benefits. First, the instructor does not need to communicate ideas with another person whose expertise is in a completely different field, which could easily lead to misunderstandings. Instead, the ideas are given directly to the MicroWorld's environment.

Second, MicroWorld is programmed by direct manipulation. The instructor receives visual feedback as he or she builds lessons. At any time, the instructor can view the objects' behaviors, the objects' appearances, the lesson's behavior, and the lesson's appearance. With a short turn around, the instructor is more willing to experiment and is less frustrated, therefore, the results will be a better lesson for the student.

MicroWorld's environment is also beneficial for the students. An interesting learning environment keeps the student's attention longer and the student learns more. Restraint is necessary to keep the lesson from becoming too flashy, otherwise, the learning material will become lost in the glamour.

Defining what makes a good instructor and writing a program that imitates the instructor is more difficult and is currently unsolved. We do not know what makes a good instructor, although some information can be gathered from watching several instructors giving lessons. Unfortunately, this only provides techniques for teaching one lesson type, which may not be transferable to another lesson type.

MicroWorld provides programmable evaluation and history mechanisms for each object. The instructor provides the know-how on teaching a lesson to the student. We, as computer scientists, do not have the expertise to build lessons for

every field, therefore, we should provide tools to the people who do have the expertise, the instructors.

Unfortunately, MicroWorld's tutor is not programmable. Instead, the instructor selects between different tutor types, which limits the lesson types a student can receive, and, therefore, MicroWorld is not the ideal environment for building lessons. Providing a fully programmable tutor requires more research.

Another research area is spatial relationship. MicroWorld provides tests for determine if an object is within a certain spatial relationship, such as "to the left of," with another object, but this work is based on a small number of observation. If spatial relationships are the basis for a lesson, then MicroWorld's spatial relationship calculations and a human observer must reach the same conclusions.

A third research area is determining how much of the programming task can be done by direct manipulation? MicroWorld has a nice environment for programming the object's event responses, but it is still programming at the basic level. Is it possible that complicated lessons can be built without writing code, whether by typing or by menu selection? If it is possible, then how does the user make modifications to the program, without starting over at the beginning? This raises interesting questions on displaying and

modifying intermediate steps.

MicroWorld needs work in several areas, including reorganizing the class structure and the programming environment. Currently, some classes should be separated into smaller classes, and some classes should be made subclasses of other classes. Although this does not affect the user's environment, it is important as the research continues.

The programming environment's design and implementation requires extensive work, especially concerning consistency between changes made in separate dialogs on the same method. Currently, MicroWorld does not correctly handle accepting changes in the method's second dialog and then later cancel the changes in the its first dialog, which was open while the second dialog was active. Furthermore, MicroWorld does not correctly support undoing changes. The programming interface should be completely redesigned for MicroWorld's next version.

Although MicroWorld is incomplete and several difficult problems remain, the environment is functional. In fact, most of the MicroWorld project formed the basis for the QUICK project, which is a rapid prototyping tool for building user interfaces. The QUICK project contains few bugs and is quite solid. Interest in both projects are high, which increases the probability that they will become actual tools in the future.

BIBLIOGRAPHY

1. Cardelli, L., "Building User Interfaces by Direct Manipulation," Technical Report #22, DEC Systems Research Center, 1987.

2. Goodman, D., *The Complete HyperCard Handbook*, Bantam Books, New York, 1987.

3. Gould, L. and Finzer, W., "Programming by Rehearsal," Technical Report #SCL-84-1, Xerox PARC, Palo Alto, CA, 1984.

4. Smith, R.G., Barth, P.S., and Young, R.L., "A Substrate for Object-Oriented Interface Design," *Research Directions in Object-Oriented Programming* (B. Shriver & P. Wegner, eds.), MIT Press, Cambridge, MA, 1987