AD-A285 977

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DTIC
ELECTE
NOV 07 1994
G D

THESIS

DESIGN AND IMPLEMENTATION OF VISUAL OBJECTED-ORIENTED LOGO USING PROGRAPH

by

Emily M. Black
and
Thierno Fall

September 1994

Thesis Advisor: C. Thomas Wu

Approved for public release; distribution is unlimited

94-34470

94 11 4 111

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503

1. AGENCY USE ONLY (Leave blank)
2. REPORT DATE: September 1994
3. REPORT TYPE AND DATES COVERED: Master's Thesis
4. TITLE AND SUBTITLE: Design and Implementation of Visual Object-Oriented LOGO Using Prograph (U)
5. FUNDING NUMBERS
6. AUTHOR(S): Emily M. Black and Thierno Fall
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES): Naval Postgraduate School, Monterey CA 93943-5000
8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES): None
10. SPONSORING/MONITORING AGENCY REPORT NUMBER: N/A
11. SUPPLEMENTARY NOTES: The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT: Approved for public release, distribution unlimited

12b. DISTRIBUTION CODE: A

13. ABSTRACT *(maximum 200 words)*

This thesis addresses the problem of how best to teach beginning programmers the necessary skills of object oriented programming. There is no established method of introducing object oriented concepts such as encapsulation, inheritance, and polymorphism, or providing an intuitive progression from simple programs to complex problem solving.

The approach was to use two commercially available programming languages which we consider exemplify good object oriented programming techniques, to teach beginners how to program. We selected LOGO, which has been used succcessfully in the past as a first programming language for children. Then we added the concepts of visual programming throught he use of Prograph, a language which provides a visual, object oriented, dataflow environment.

The main result of our research is the design and implementation of a prototype language called Visual Object Oriented LOGO (VOOL). VOOL is intended for use at all levels of education to teach problem solving, object oriented concepts, and fundamental programming skills. VOOL was implemented on a Macintosh in the pictorial, iconic language of Prograph and fully supports the goals of this thesis

14. SUBJECT TERMS: object oriented programming, visual languages, LOGO, Prograph, turtle graphics, classes, objects, inheritance, encapsualtion, polymorphism
15. NUMBER OF PAGES: 110
16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Ap[illegible]d for public release; distribution is unlimited

Design and Implementation of
Visual Object-Oriented LOGO using Prograph

Emily M. Black
Lieutenant Commander, United States Navy
B.A., Wellesley College, 1979

and

Thierno Fall
Captain, [illegible] Army
B.S., Lycee De[illegible] Senegal, 1979
Civil Engineering, Poly[illegible]ique Senegal, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September, 1994

Authors: Emily M. Black

Thierno Fall

Approved By: C. Thomas Wu, Thesis Advisor

Roger Stemp, Second Reader

Ted Lewis, Chairman,
Department of Computer Science

For
A-1

ABSTRACT

This thesis addresses the problem of how best to teach beginning programmers the necessary skills of object oriented programming. There is no established method of introducing object oriented concepts such as encapsulation, inheritance, and polymorphism or providing an intuitive progression from simple programs to complex problem solving.

The approach was to use two commercially available programming languages which we consider exemplify good object oriented programming techniques, to teach beginners how to program. We selected LOGO, which has been used successfully in the past as a first programming language for children. Then we added the concepts of visual programming through the use of Prograph, a language which provides a visual, object oriented, dataflow environment.

The main result of our research is the design and implementation of a prototype language called Visual Object Oriented LOGO (VOOL). VOOL is intended for use at all levels of education to teach problem solving, object oriented concepts, and fundamental programming skills. VOOL was implemented on a Macintosh in the pictorial, iconic language of Prograph and fully supports the goals of this thesis.

TABLE OF CONTENTS

I. INTRODUCTION 1
A. BACKGROUND 1
B. OBJECTIVES 2
C. ORGANIZATION OF THE THESIS 2
II. DESCRIPTION OF THE ORIGINAL LOGO 5
A. DESCRIPTION OF LOGO 5
1. Turtle Graphics 5
2. Sample Turtle Program 6
3. Functional Abstraction 8
B. LOGO OBJECT ORIENTED PROGRAMMING 10
1. Object LOGO 10
2. LOGO Inheritance 12
3. LOGO Multiple Inheritance 13
4. Method Override 14
III.STRENGTHS AND SHORTCOMINGS OF LOGO 17
A. STRENGTHS OF LOGO 17
1. LOGO is Modular 17
2. LOGO Supports Nesting 17
3. LOGO is Interactive 18
4. LOGO Supports Graphics 19
5. LOGO Supports List Processing 19
6. LOGO Develops Problem Solving Skills 19
7. LOGO is a General Purpose Programming Language 20
B. SHORTCOMINGS OF LOGO 20
1. LOGO's Functionality 21
2. LOGO's Object Oriented Concepts 21
a. Turtle's Abstract Data Type 22
b. Turtle's Object Class 22
c. Inheritance 23
d. Data Abstraction 23
IV. VISUAL PROGRAMMING 25
A. WHAT IS A VISUAL PROGRAMMING LANGUAGE? 25
1. **Definition** 25
2. **History** 26
B. ADVANTAGES OF VISUAL PROGRAMMING 26

C. GOAL OF VISUAL OBJECT ORIENTED LOGO (VOOL) 27
V. DESIGN OF VISUAL OBJECT LOGO 29
A. PROGRAPH AS A FOUNDATION 29
1. Prograph's Class System 30
2. Prograph's Attributes 32
3. Prograph's Methods 32
B. THE TURTLE AS A MODEL 34
1. Turtle Class Hierarchy 34
2. Turtle Attributes and Methods 36
VI. IMPLEMENTATION OF VOOL 39
A. OBJECTS AND CLASSES 39
1. The name: 40
2. The attributes: 40
3. The methods: 40
B. INHERITANCE 41
C. ENCAPSULATION/ABSTRACTION 44
D. POLYMORPHISM 50
E. ITERATION 51
F. SELECTION 53
VII. CONCLUSION AND RECOMMENDATIONS 59
A. VISUAL APPROACH: PROS AND CONS 59
B. BUILDING THE PROTOTYPE 60
C. ASSESSMENT 61
D. RECOMMENDATIONS 62
APPENDIX A-USER COMMAND AND METHOD DEFINITIONS 65
APPENDIX B - TURTLE GRAPHICS' SOURCE CODE 69
LIST OF REFERENCES 93
BIBLIOGRAPHY 95
INITIAL DISTRIBUTION LIST 97

LIST OF FIGURES

1. Turtle Graphics Commands for Square Process 8

2. Object Instances of Turtle CLIFF and JEFF 12

3. The Inheritance Hierarchy Relating HOPPER, FLYER, TURTLE 13

4. The Multiple Inheritance Hierarchy for BIRD 14

5. Prograph's Graphical Class Hierarchy 31

6. Prograph's Attribute Definitions 33

7. Prograph's Method Definitions 33

8. Turtles' Class Hierarchy 35

9. Main_Turtle's Attribute Definitions 37

10. Main_Turtle's Method Definitions 37

11. Turle_1's Method Definitions 38

12. Turtle_1's Attribute DEfinitions 38

13. Iteration For Circle........ 45

14. Turtle_1 / Polygon 45

15. Polygon 46

16. Square 46

17. Draw Train 49

18. Iteration / Polygon 52

19. Iteration For polygon........ 54

20. Rotator 54

21. Iteration / Circle 55

22. Circles 56

23. Compute Coord (Selection) 58

ACKNOWLEDGMENT

This thesis has an international flavor, bringing together minds from the shores of Hong Kong, Senegal Africa, and the tiny state of Rhode Island, (a foreign land in itself). We wish to thank Dr. Wu, the Emperor of Enlightenment, for his invaluable guidance during our months-long Quest for the Thesis Grail. Professor Stemp, our second reader, also deserves credit for his thoughtful suggestions for improvement, and for reading the finished product without going cross-eyed.

Of course, our spouses, John Black and Aissatou F. Sabara, deserve special recognition for admitting that they know us. Many thanks to both of them for their unquestioning support, and their blind faith that we knew what we were doing!

I. INTRODUCTION

A. BACKGROUND

In 1967 Seymour Papert introduced Turtle geometry with the development of the programming language Logo at Massachusetts Institute of Technology. The initial intent was to develop a computer language that would be both suitable for children, yet powerful enough for professional programmers. Logo is described as having "no threshold" -- preschool children can use it -- and "no ceiling" -- computer scientists can use it for their work. The turtle is simply a computer-controlled cybernetic animal. It exists within the cognitive Logo environment in which communication with the turtle takes place. Turtle geometry is a computational style of geometry in which the turtle has its own position and heading. It provides a straightforward meaning to attach to each individual procedure, namely a picture. Children can identify themselves with the turtle and are thus able to bring their knowledge about their bodies and how they move into the process of learning formal geometry. The basic foundation for turtle geometry lies with the idea that specific problems of interest to the novice can be tackled by simple programs.

Logo is a widely recognized programming language for children, designed to develop both their problem solving skills as well as their general programming skills. The language has met many of its original goals, but is somewhat outdated by today's computer standards, especially as it does not integrate the concepts of object-oriented programming.

B. OBJECTIVES

The purpose of this research is to study the benefits and shortcomings of the programming language Logo, and to update it by proposing a fully visual programming system which supports object oriented concepts. This system, titled Visual Object Oriented Logo (VOOL), has the primary objective of providing an intuitive environment in which students of all skill levels can learn the concepts of an object oriented language. The visual, object oriented, data flow programming environment of Prograph was used for the implementation of this prototype, because it provided the necessary base classes for interface design, as well as the primitive operations for graphics drawing functions.

To meet our objectives, this thesis addresses the following issues: first, it provides a description of the original language Logo; second, determines the merits and shortcomings of the language Logo, and then suggests possible solutions to the shortcomings through a prototype; third, evaluates the benefits of visual object oriented design; fourth, describes the design and the implementation of the proposed prototype; and finally, identifies the targeted users of the upgraded Logo language. Each specific issue will be developed in detail throughout later chapters.

C. ORGANIZATION OF THE THESIS

Chapter II of this thesis describes the original Logo language, followed by the objectives, strengths and shortcomings of Logo in Chapter III. Chapter IV contains the

goals of visual object-oriented Logo. Chapter V discusses the design of the prototype, then Chapter VI focuses on the prototype's implementation, including amplifying examples. Finally, Chapter VII summarizes the research and provides conclusions and recommendations.

II. DESCRIPTION OF THE ORIGINAL LOGO

Logo was developed at the Massachusetts Institute of Technology under the direction of Seymour Papert. Papert's intent was to create a "mathland" where students could actively explore mathematical concepts.

A. DESCRIPTION OF LOGO

Logo is a full-featured computer language derived from Lisp, the language of artificial intelligence. Logo was developed to be both powerful and easy to use, hence it is found in educational settings from kindergartens to universities. Although the language itself is not limited to any particular subject, it is most commonly used for exploring mathematics, since Logo's turtle graphics provide a natural mathematical environment. In particular, it is an ideal tool for studying geometry through manipulating the turtle to draw various shapes.

1. Turtle Graphics

Logo's best-known feature is the Turtle, a triangular cursor used to create graphics in a programming area called Turtle Geometry. Originally, the Turtle was viewed as a computer-controlled "cybernetic animal" that leaves a trace on the display screen and responds to Logo commands to make it move or rotate. The motion of a pen on a sheet of paper is simulated by commanding the Turtle to move FORWARD, or to turn LEFT or

RIGHT. The user specifies how many units to move ahead, or how many degrees to turn. As the Turtle moves (FORWARD, BACK) or rotates (RIGHT, LEFT), it leaves a trace of its path, and in this way can be used to make drawings on the display screen. Even young children can quickly learn to move and turn the Turtle using easily-remembered, intuitive commands, and can generate quite detailed pictures. Using these commands, a child can explore the properties of regular planar shapes ranging from simple squares, triangles, and hexagons to complex patterns reminiscent of the designs made by Spirograph drawing wheels.

Turtle space is defined by the dimensions of the screen on which the graphics will appear. Screen dimensions are generally stated as x and y axis coordinates representing horizontal and vertical positions respectively, and with the origin at $x = 0$, $y = 0$. The Turtle can then be moved around the screen using one of two possible frames of reference. In the first method, the user specifies an x-y position on the screen, and the Turtle moves directly to that point regardless its current location or heading. Alternatively, the user can specify distances to move and angles to turn, which are then executed based on the Turtle's current location and heading. Basically, the first method uses the grid origin as the reference point, and the second method uses the Turtle itself as the reference point.

2. Sample Turtle Program

The commands FORWARD and BACK require an input from the user to specify how many units the Turtle should move in the indicated direction. The commands RIGHT and LEFT require the number of degrees that the Turtle should rotate. For example,

typing FORWARD 50 moves the turtle forward 50 pixels (screen dots), and typing RIGHT 90 turns the Turtle 90 degrees clockwise

If we consider maneuvering the Turtle to draw a square, the following commands must be executed

1. FORWARD 50
2. RIGHT 90 that completes the left side of the square.
3. FORWARD 50
4. RIGHT 90 that completes the top side of the square.
5. FORWARD 50
6. RIGHT 90 that completes the right side of the square.
7. FORWARD 50
8. RIGHT 90 that completes the bottom side of the square.

By combining these steps with the command REPEAT, the same square can be drawn using only one line of code.

REPEAT 4 [**FORWARD** 50 **RIGHT** 90]

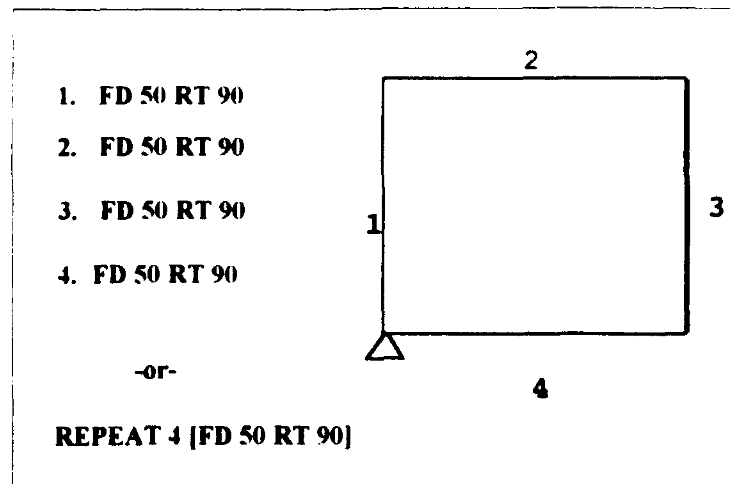


Figure 1: Turtle Graphics Commands for Square Process.

3. Functional Abstraction

Due to Logo's extensibility, the user can add new commands by creating short programs or sets of instructions called procedures. Some procedures can call other procedures as their helpers to solve a new general problem. Each procedure has a name which is used as the shorthand for calling all lines of code within it; this is a form of encapsulation. Once you have defined a procedure, it becomes part of the computer's working vocabulary and can be used as if it were a primitive. Building a library of procedures encourages code reusability and facilitates more time-efficient coding of future programs. Procedures can be viewed as the building blocks of larger programs.

A procedure is composed of three parts: a title line, which consists of the word TO followed by the name you chose for the procedure; a body, which is the sequence of commands that comprises the definition; and the word END which indicates completion of

the procedure Simply typing the name of the procedure tells Logo to automatically execute each line of the procedure in turn, just as if you had typed them individually on the keyboard

For example, here is the procedure that will draw our familiar square

```
TO SQUARE
    REPEAT 4 [FD 50 RT 90]
END
```

Now, to draw a square, you only need to type the single word SQUARE. Whenever you use this command, the turtle draws a square with side 50. You can use SQUARE just like any other Logo command, even including it in other procedures.

Generalizing a procedure can add to the command's power by giving the user more control and flexibility. You can change the definition of SQUARE to enable it to draw squares of all different sizes. The new SQUARE procedure takes an input that allows the user to specify the desired length of a side:

```
TO SQUARE :SIDE
   REPEAT 4 [FORWARD :SIDE RIGHT 90]
END
```

The procedure is evaluated just like any Logo command that takes an input, that is, to draw a square of side 25, you type SQUARE 25.

B. LOGO OBJECT ORIENTED PROGRAMMING

1. Object LOGO

The original Logo Turtle did not support object oriented programming concepts. It was designed as an educational tool of considerable power and wealth of expression, to be used easily at any level of the educational system.

As object oriented programming became an increasingly powerful and popular programming methodology, Object Logo was introduced as a type of object oriented system. Object Logo includes all the features found in original Logo, but goes beyond them in incorporating the concepts of object oriented programming.

Object Logo supports two of the most basic concepts in object oriented programming: objects and inheritance. An **object** is a collection of procedures and data that works together to implement some kind of behavior. An object's procedures represent the things that the object knows how to do. Once you have defined a type of object, you can make multiple copies, (also called multiple instances), of the object to produce multiple creatures that have the same behavior. Object Logo's data objects include not only numbers and character strings, but also compound structures. Since Object Logo

procedures can themselves be represented and manipulated as lists, users attain considerable direct control over the way commands are interpreted

The Turtle is an easily understood example of a Logo object, so we will focus primarily on Turtle objects and the methods inherited from them. The best way to show how objects behave in the Logo environment, is to create several Turtles and make them interact. To create a new object, you can use one of two Logo commands: KINDOF or ONEOF. The difference between the two commands is that, with KINDOF, you must explicitly ask the object to EXIST in order to initialize values for the object's variables; ONEOF will automatically make the object EXIST [Abel, pg. 99]. For example, you can create a Turtle named JEFF to be a KINDOF Turtle, and ask JEFF to EXIST. (To indicate a word in Logo, you type the character string prefixed by a quotation mark). At this point you can give commands to JEFF, using ASK to indicate which object you're asking:

MAKE "JEFF **KINDOF** TURTLE

ASK JEFF **[EXIST]**

ASK JEFF **[FORWARD** 100]

ASK JEFF **[LEFT** 45]

TALKTO is another way to issue commands to an individual object. Once you specify the particular object you're talking to, all commands go directly to that object until

you specify otherwise. The following example shows the interaction of two instances of Turtle named CLIFF and JEFF; each of them was asked to draw a square. (The two objects were created prior to the TALKTO commands, and CLIFF was forwarded 100 units, while JEFF was turned 45 degrees then forwarded 100 units).

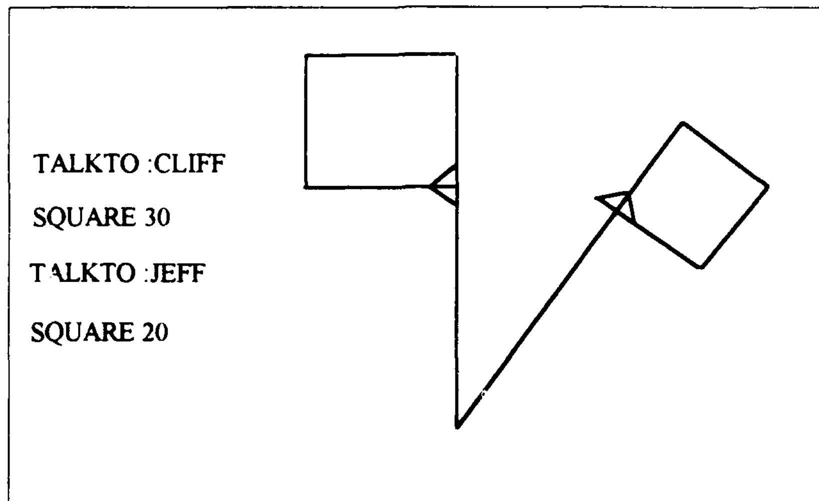


Figure 2 : Object Instances of Turtles CLIFF and JEFF.

2. LOGO Inheritance

Inheritance is the ability to define new kinds of objects in terms of previously-defined objects. Figure 3 shows an example of the inheritance hierarchy relating three objects named Hopper, Flyer and Turtle. Turtle is said to be the parent of Hopper and Flyer. Whenever you ask an object to perform a command, the object first checks to see if it has a command by that name. If none is defined, then it checks its parent, then its parent's parent, and so on. For example, a Flyer knows how to SWOOP because SWOOP was defined for Flyer, whereas a Flyer knows how to FORWARD because its parent, Turtle, knows how to FORWARD and the trait was inherited.

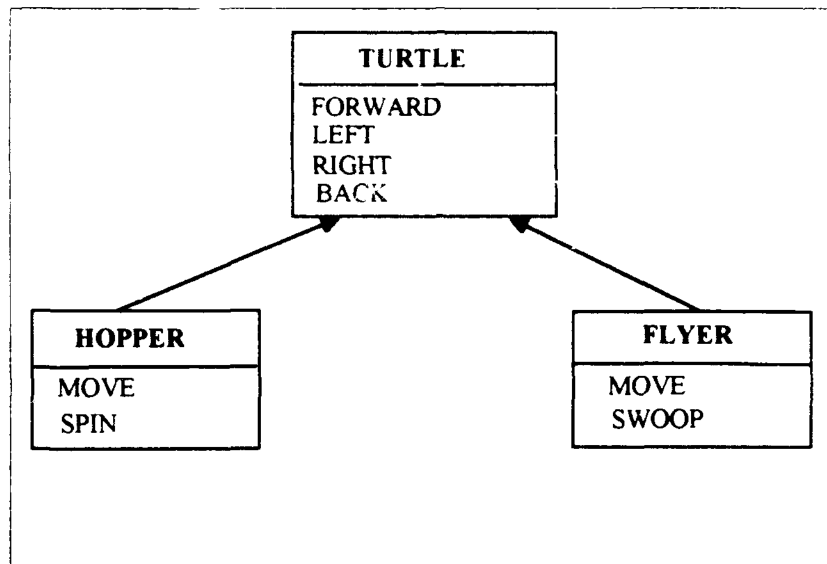


Figure 3 : The Inheritance Hierarchy Relating HOPPER, FLYER, TURTLE.

3. LOGO Multiple Inheritance

Inheritance rules can be more complex when an object has multiple parents. For instance, you can make a Bird that is both a kind of Hopper and a kind of Flyer.

MAKE "BIRD **KINDOF** (LIST :HOPPER :FLYER)

This example shows that you can make something with multiple parents by giving KINDOF a list of parents, rather than just a single parent. As you can see on Figure 4, a Bird can go FORWARD, BACK, LEFT, and RIGHT, (which it inherits from Turtle); a Bird can SPIN, (which it inherits from Hopper); and a Bird can SWOOP, (which it inherits from Flyer). A potential conflict exists, since MOVE is inherited from both Hopper and

Flyer and might contain disparate definitions. The conflict is resolved by a rule requiring Bird to MOVE like Hopper, because Hopper was the first parent named in the list.

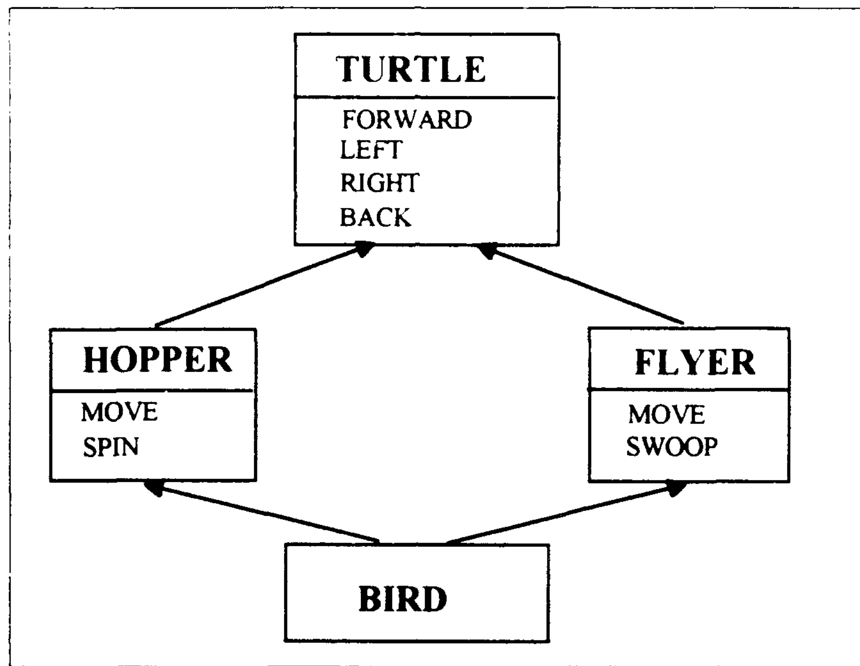


Figure 4: The Multiple Inheritance Hierarchy for Bird.

4. Method Override

An object and its parent might both have methods with the same name. In this case, the object will use its own method definition, rather than that of its parent. In Logo, if you need to explicitly refer to the parent's method vice the object's method, you use the dot notation with the prefix USUAL. For example, you can make a new object, Creeper, that is a kind of Turtle, and teach it to go FORWARD differently than a regular Turtle, as in the second line:

```
MAKE "CREEPER KINDOF TURTLE
ASK :CREEPER [TO FORWARD :DISTANCE]
REPEAT :DISTANCE [USUAL.FORWARD 1]
END
```

The notation **USUAL.FORWARD** in the third line indicates that the procedure definition for FORWARD should be used from a parent object. This concept of referring to a specific method definition using the dot notation is called method overriding.

III. STRENGTHS AND SHORTCOMINGS OF LOGO

Logo is a language that encourages students to explore, learn, and think. In the following sections, we'll discuss some characteristics that contribute to its success, then identify some of its limitations.

A. STRENGTHS OF LOGO

1. LOGO is Modular

A Logo language program is not necessarily written as one large unit. Rather, it can be divided into smaller pieces, and a separate procedure can written for each piece. A procedure is a group of one or more instructions to the computer that the computer can store to be executed at a later time. Logo users start with a vocabulary of primitives and use them to develop new procedures to add to the vocabulary. Procedures can communicate among themselves via inputs and outputs, and each new procedure becomes an extension of Logo. Figures 1 and 2 provide an illustration of building procedures and procedure calls using SQUARE.

2. LOGO Supports Nesting

We've already seen the use of the Logo REPEAT command to repeat a series of steps for a specified numbers of times. Another way to cause repetition in Logo is to define a procedure that includes a call to itself as the final line. This is called recursion and

is very useful because it allows an involved problem to be described in simple terms. For example, you can make the Turtle move in a square pattern over and over again until explicitly stopped.

```
TO SQUARE :SIZE
FORWARD :SIZE
RIGHT 90
SQUARE :SIZE
END
```

In this case, the definition of SQUARE is "go forward, turn right, and do SQUARE again." One disadvantage of this particular SQUARE, as opposed to REPEAT, is that it goes on indefinitely and so is not a good building block to use in making complex drawings.

When used properly with a stop condition, recursive procedures are a powerful tool which can be used to obtain complicated effects, but they are also much harder to understand and handle.

3. LOGO is Interactive

Any Logo command, whether built into the language or defined as a procedure, can be evaluated by simply typing the command at the keyboard. It is easy to change or

correct a procedure in Logo, because the editing process is designed as part of the language.

4. LOGO Supports Graphics

Turtle graphics allows users to order a Turtle to move forward or backward and turn left or right. As stated previously, the Turtle can leave a trace, or it can move without a trail. With simple commands, the user can "teach" the turtle to draw very complex drawings. These drawings can be created with or without reference to any coordinate system.

5. LOGO Supports List Processing

Logo provides operations for manipulating character strings that Logo calls "words." Logo also has the ability to combine data into structures called "Lists." These lists can be used to create very complex data structures.

6. LOGO Develops Problem Solving Skills

Logo's best known feature is the Turtle. By identifying themselves with this object, even children can use their knowledge of how a turtle behaves to move it across the screen. Children learn to program in Logo by experimenting with maneuvering the Turtle. Thanks to the turtle, Logo becomes very friendly and easy to understand, and offers immediate feedback through helpful messages. Many studies have been conducted in classrooms from kindergarten to elementary school, researching how Logo helps children develop their problem solving abilities. Results show that children using Logo exhibit statistical gains over non-Logo users.

Children who draw pictures with a Logo turtle unconsciously assimilate important mathematical concepts such as angles and estimation. In addition, by telling the Turtle to go forward 1000 units, forward 200 units, and left 30 degrees, many children are using number in a meaningful way for the first time; they receive feedback on the relative sizes of numbers that they never get doing school arithmetic [diSe 86, pg. 9].

7. LOGO is a General Purpose Programming Language

The primitives and concepts of the language are encountered in the context of working procedures. Completely predefined procedures are presented to the beginner, to be typed into the computer and used. In fact, experimenting these pre-written procedures is a good way of learning how they work. This approach is not prescriptive, in that students are allowed to develop their own way of grouping the primitives. The procedures presented are carefully chosen and deliberately ordered to illustrate particular concepts, increasing in complexity as the student progresses. By reading, using, modifying, and extending these procedures, the beginner can develop his or her own understanding and knowledge of the language and of programming concepts in general.

B. SHORTCOMINGS OF LOGO

Despite Logo's benefits and successes in the educational environment, we think there still exist some limitations of the language. This is primarily due to the fast development of today's computer standards, especially the emergence of object oriented programming.

1. LOGO's Functionality

We think that the power of Logo is limited by constraints on the amount of memory, and constraints of the processing power that, in part, forced Logo's designers to slight some areas. Logo is limited in building large structures and abstract objects that are difficult to manipulate. Logo is a list-based interpreted language where each line is processed as it is entered into a work window.

We also believe that Logo's syntax and semantics could be improved. For example, Logo includes only two limited conditional expressions for allowing users to write programs that perform test controls. The first expression is IF <condition> <action>: if the test condition is true, then the action is executed. A variation is IFELSE <condition> <action1> <action2>: if the test condition is true, then action1 is performed; otherwise, action2 is performed. The second expression is TEST <condition> IFTRUE <action1> IFFLASE <action2>: this tests a condition to be used in conjunction with IFTRUE and IFFALSE. These conditional expressions are only useful for basic test cases, but are not powerful enough for complex test operations.

The REPEAT command of Logo and the recursive procedures for executing multiple operations are also only useful for basic operations, due to the limitations of the control loop.

2. LOGO's Object Oriented Concepts

The original Logo was an early programming language, developed before the growth of object oriented programming languages. Object oriented programming (OOP) is

a data-centered view of programming in which data and behavior are strongly linked Data and behavior are conceived of as **classes** whose instances are **objects**. In the original Logo, the concepts of object-oriented programming are poorly defined in that there is no specific link between data and behavior, and there is no distinction between object and class.

a. Turtle's Abstract Data Type

In Logo, the user doesn't have access to the Turtle's attribute types nor to its predefined operations. This lack of access to the Turtle's data structure makes it difficult to explore the object oriented features. The user interacts with the Turtle only through the editor where he types a set of instructions and sees the result on the display screen. In our prototype, however, the user has the opportunity to check, and modify if necessary, the different attributes and methods that characterize the turtle. At run time, the user is not only able to see the data being input, but can also evaluate or modify it.

b. Turtle's Object Class

The concept of an object in Logo can lead to confusion, because Logo does not define the concept of a class. Although objects and classes are two different notions, there is a distinct relationship between them. In our model, a class defines a type of object and a set of operations associated with that object. An object of a class is referred to as an instance of that class. The object is defined in terms of its attributes and methods. Even though a system class with its inheritance hierarchy is clearly predefined in the Prograph

environment, the student should be able to define his or her own object class with its own characteristics, without being forced to link it to an existing object class

c. Inheritance

Although the concept of inheritance is defined in Logo, the type derivation mechanism is in a rigid form. The fundamental idea behind inheritance is code reuse, yet in Logo the user does not have any control over which components are inherited. This significantly limits the flexibility and practicality of reusing code. In addition, the language does not have a specific system class available to the user for modifications.

d. Data Abstraction

Logo has strong functional abstraction because all the low level implementation of commands and instructions is hidden from the user. The notion of data abstraction, however, is not specifically addressed, thus preventing the student from accessing beyond the command line level. This notion of data abstraction is clearly described in our system, allowing the user to access precisely what is needed to perform the job, and to modify the underlying code, if necessary.

IV. VISUAL PROGRAMMING

A. WHAT IS A VISUAL PROGRAMMING LANGUAGE?

1. Definition

Visual programming is a very general term with no consensus on exactly what it means. Visual programming languages have many forms and methodologies, depending on the application for which they will be used. Chang segments visual languages into the following four classifications:

- *languages that support visual interaction;*
- *visual programming languages;*
- *visual information processing languages, and*
- *iconic visual information processing languages.*

Visual Object Oriented Logo (VOOL), implemented using the Prograph environment, falls into the fourth category.

In the context of this thesis, a visual programming language is defined to be "a language which uses some visual representations (in addition to or in place of words and numbers), to accomplish what would otherwise have to be written in a traditional one-dimensional programming language." [SHU, pg. 138].

2. History

Text-based programming languages developed in parallel with computer hardware, and have been significantly influenced by the organization of this hardware. As a result, these languages are oriented towards simple, character-based input and output, and are generally sequential in structure. In addition, they tend to rely on a combination of mathematical formulas and natural language, resulting in a complex, inflexible syntax. The advent of sophisticated, high resolution graphics and user interfaces has made possible the direct use of pictures in programming. Pictorial programming has become a highly researched area, and visual languages are shaking free from the legacy of traditional programming approaches.

B. ADVANTAGES OF VISUAL PROGRAMMING

In many situations, people frequently prefer pictures to words. Pictures can be very powerful, conveying information succinctly without loosing the primary message in a padding of verbiage. Most importantly, pictures can bridge language barriers, or in the case of youngsters, can bridge reading ability barriers. In the arena of computer science, visual programming is an emerging new field for precisely these reasons. Its growing popularity is aided by the falling costs of graphics related hardware and software. Beginning programmers, and children in particular, usually face a large gap between the idea they want to implement and knowledge of the syntax and semantics of a text-based

programming language. It is difficult enough for beginners to analyze a problem, without getting bogged down in how to solve it using linear code. A visual language supports beginners with a premade vocabulary and system defined icons, which assist in organizing thoughts and structuring the program. For example, in linear programming, something as simple as incrementing a counter may require interspersing statements into three or four separate places in the program. Computer architecture is driving the code writing, rather than what is the natural way for a beginner to approach the problem. Intermingling code in this manner tends to blend the structure of the program, obscuring how different components of the problem solution interrelate with one another. Visual programming represents individual components with separate icons, thus enabling the beginner to actually see the interconnections between various parts of the program. The beginner can then manipulate and connect the various icons without confusing the two issues of structure and function. This is particularly beneficial when there are multiple connections to a given icon.

C. GOAL OF VISUAL OBJECT ORIENTED LOGO (VOOL)

Our goal is to design an environment suitable for all levels of computer literacy, in which a student can learn the concepts of object oriented programming in an intuitive manner. VOOL combines the advantages of object oriented languages with the benefits of visual programming described above. As a visual language, it is more intuitive for the beginner, and as an object oriented language, it incorporates the main OOP concepts of

inheritance, polymorphism, and encapsulation. VOOL supports code reuse and will ultimately reduce the time needed to implement solutions.

VOOL programs are built by pointing, moving and copying, with only minimal labeling. This reduces the typical syntactical errors which commonly consume the time, energy and patience of beginners. Once students are largely relieved of the mechanics, they can then concentrate on mastering the concepts of object oriented programming. To this end, VOOL provides a kernel class, in our case called Main_Turtle, which can be used to experiment with object oriented features. As the beginner progresses to more advanced levels of understanding, he can create his own classes and write his own methods.

V. DESIGN OF VISUAL OBJECT LOGO

This chapter addresses three issues: first, characteristics of the programming language Prograph, which we used for the design and implementation of our model; second, a description of the model; and third, examples of key object oriented programming concepts.

A. PROGRAPH AS A FOUNDATION

This section's intent is not to teach the concepts of the visual programming language Prograph, but rather to provide a basic understanding of its programming environment.

Prograph is a fully visual, object oriented dataflow language developed in the early 1980's and implemented on the Macintosh platform. It is an object-oriented language that defines objects and methods with icons, offering an option to text-based programming languages such as Logo. Prograph supports a highly visual programming system which has multiple windows for viewing program execution states, visual syntax editors for designing program data structures, and graphical expressions in the windows themselves. The major categories of operations are assigned icons, which are then connected to create the code; the result is code in the form of a diagram of icons.

The language interpreter allows the user to see the results of execution immediately, while the compiler builds complete stand-alone applications. This is in keeping with

Prograph's stated purpose of providing a programming environment which is intuitive and easy to use, but which also produces effective, useful programs. Prograph provides a mixture of high and low level routines, even including some Macintosh system calls for advanced interface programming. As a complete language, Prograph satisfies a wide range of different programming requirements.

1. Prograph's Class System

A class in Prograph is defined as a type of object and a set of operations associated with that object. The object is defined in terms of its *attributes*, while operations on that object are defined in *methods* of that class. An object of a class is referred to as an instance of that class.

Figure 5 shows the Prograph class system which provides a mean for constructing a new class from an existing one, via the mechanism of *inheritance*. Each Prograph class is represented by an hexagonal icon displayed in the class window. The lines connecting individual classes within the hierarchy represent the inheritance links between various classes. The left side of a class icon contains the attributes of that class, and the right side contains it associated methods. All object oriented programming applications start out with these minimum template classes.

A new class icon is created by clicking inside the class' window. The newly created class is given a unique name and is defined by adding the appropriate attributes and methods related to that class.

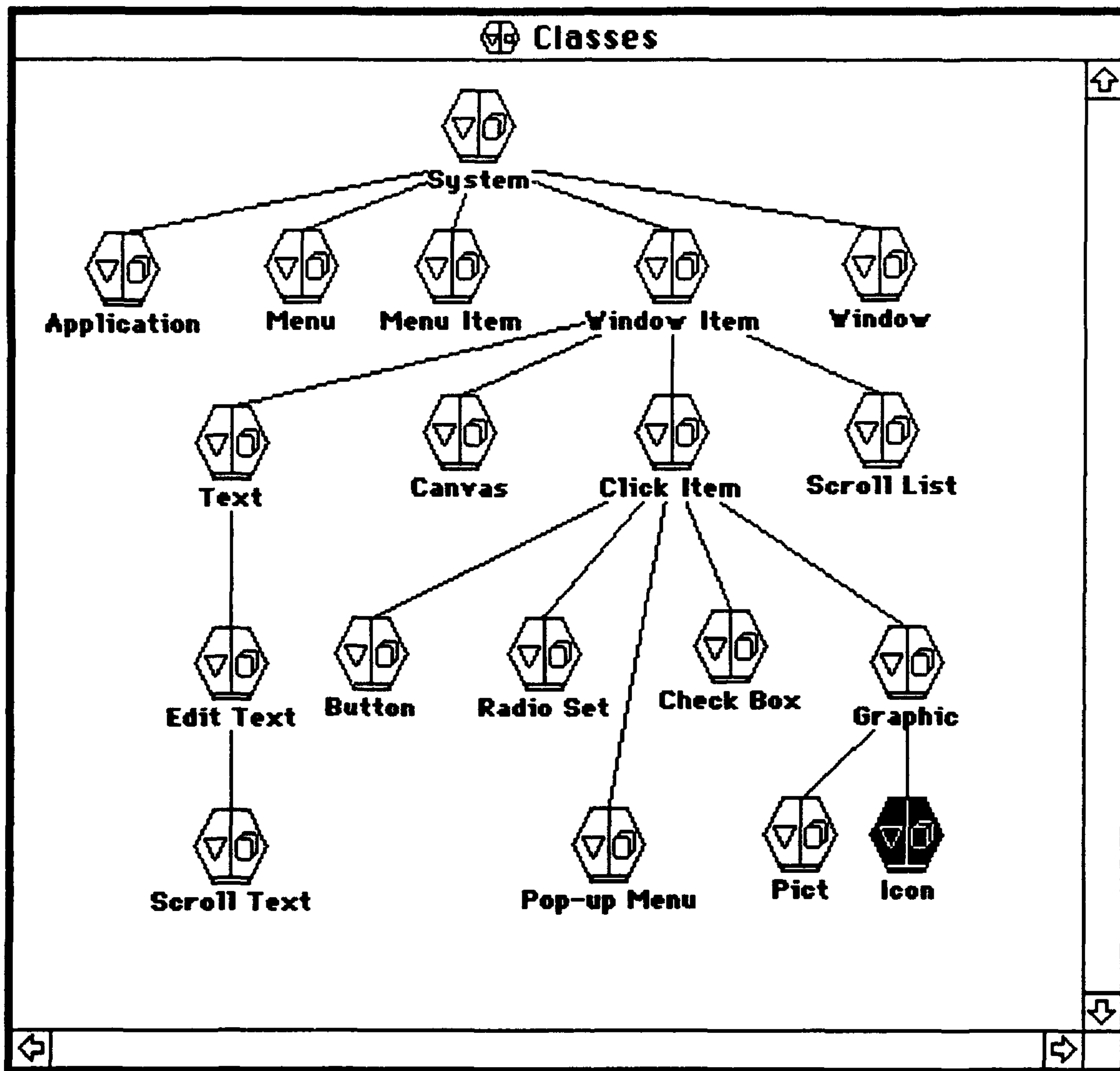


Figure 5: Prograph's Graphical Class Hierarchy

2. Prograph's Attributes

An attribute is a named slot for holding a value, and an attribute's name must be unique within the class. Figure 6 shows the results of double-clicking on the left half of the class icon. There are two types of attributes: an ***instance attribute*** is represented by an inverted triangle, and can have a different value for each instance of the class; a ***class attribute*** is represented by the hexagonal shaped icons above the thin line, and has one value which is shared by all the instances of the class. Inherited attributes have a downward pointing arrow inside the inverted triangle. Attributes can be assigned initial values by double-clicking on the icon and changing the value in the attribute editor. In addition to simple data types, attributes can also be instances of other classes that already exist as a composite object in Prograph.

3. Prograph's Methods

A method is a procedure or function associated with an instance of a class. As shown in Figure 7, methods are represented by an icon that contains a small dataflow diagram. A method is activated by double-clicking on the icon, at which point new code can be written, or existing code can be read. A methods name should be unique.

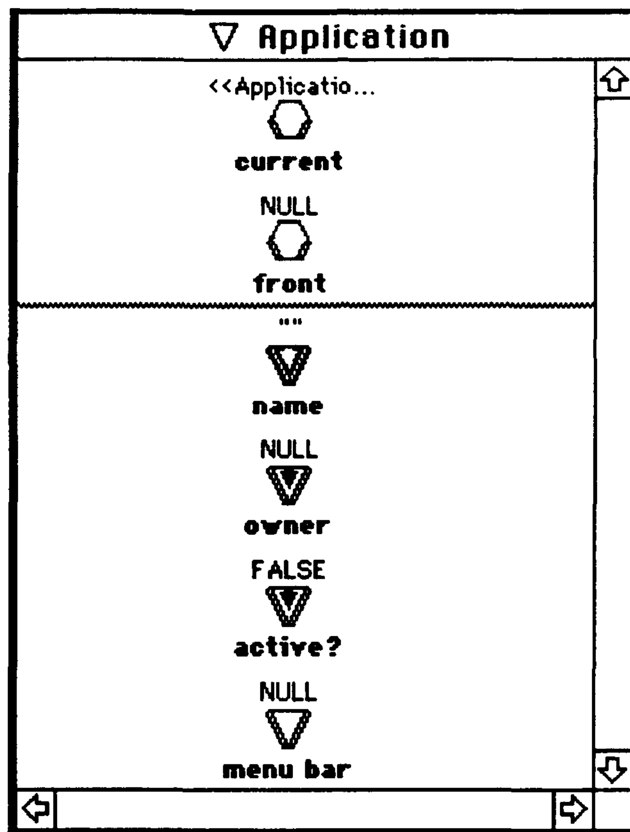


Figure 6: Prograph's Attribute Definitions

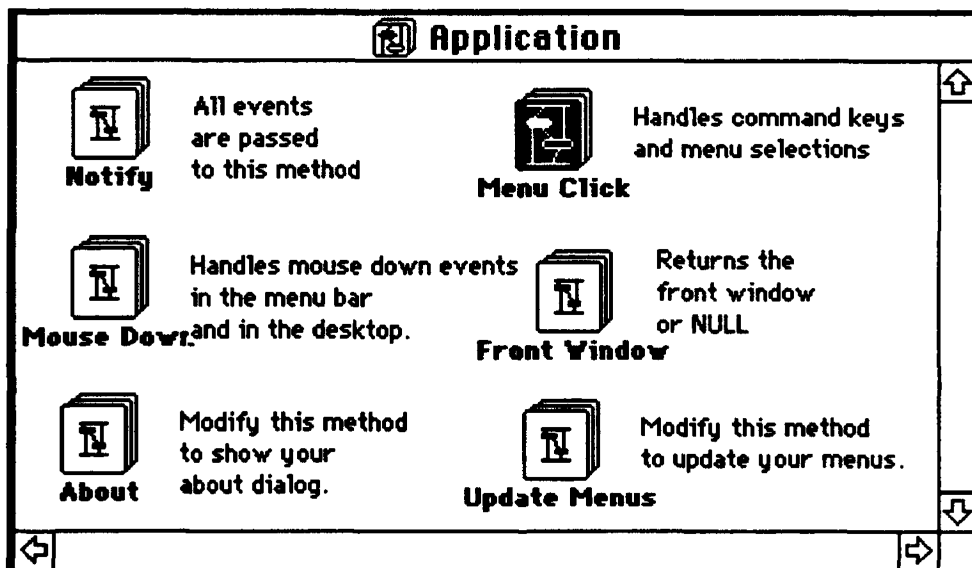


Figure 7: Prograph's Method Definitions

B. THE TURTLE AS A MODEL

The design and implementation of the prototype for this thesis is based on creating a Turtle object with specific attributes and specific messages, each of which can be defined by the user. The Turtle itself is split into two levels of complexity: first, the Main_Turtle has basic methods such as Move, GotoPos, TurnTo, PenUp, etc.; second, Turtle_1 contains additional behaviors such as making squares, polygons, circles etc. All the methods are fully defined in Appendix A.

A beginning programmer can relate to the idea of steering a miniature Turtle around on the screen, and this lays the groundwork for introducing the underlying concepts of object oriented programming.

1. Turtle Class Hierarchy

The design of the class hierarchy was based on the need to create Turtle objects that would have the necessary attributes and methods common to all subsequently created Turtle objects. This design includes the drawing window which is used by all existing Turtles for graphical display. Figure 8 shows the Turtle class hierarchy.

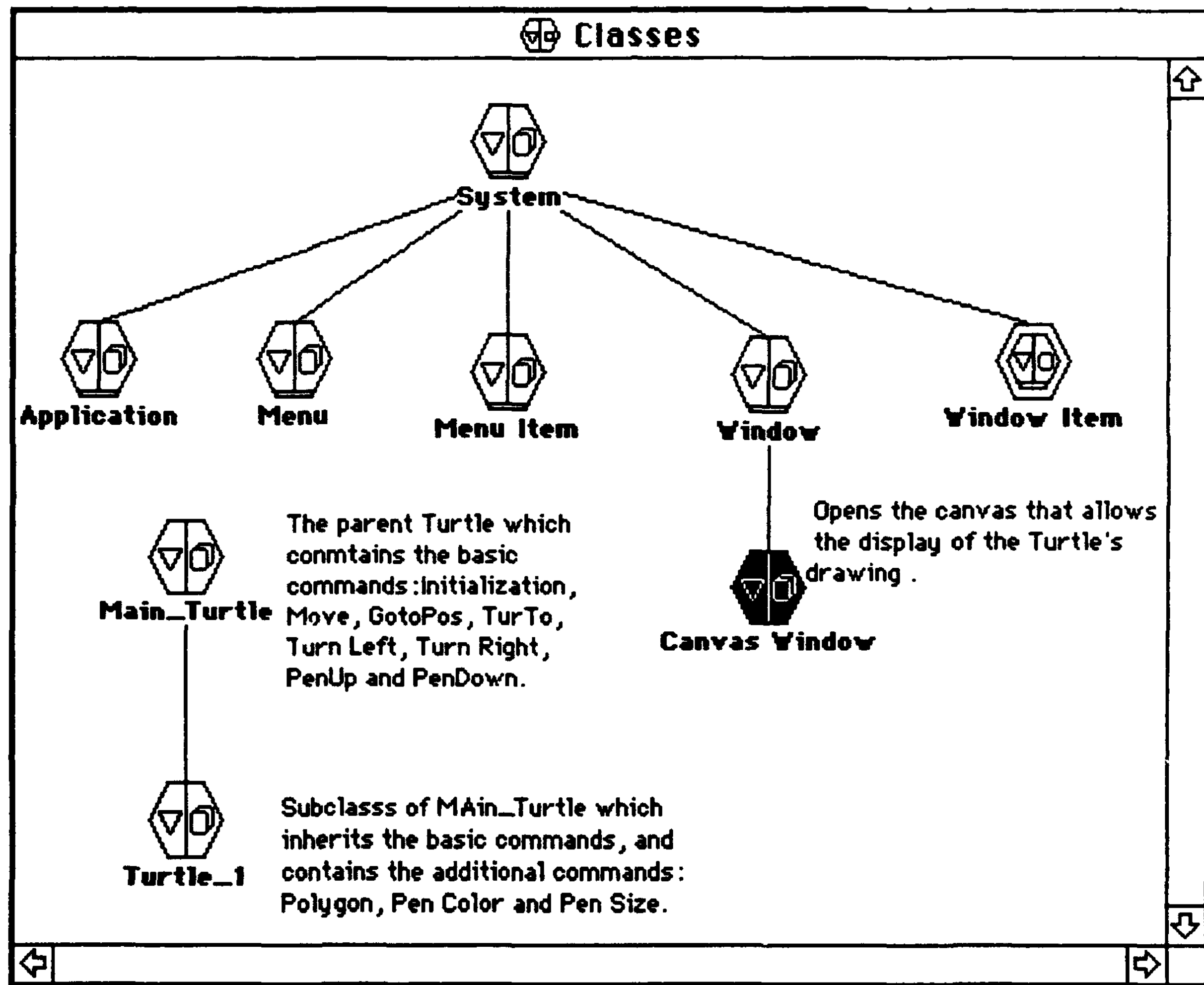


Figure 8: Turtle Class Hierarchy

2. Turtle Attributes and Methods

Figures 9 and 10 show the graphical representation of the Main_Turtle's class attributes and methods; they contain the necessary framework to define specific Turtle instances. In addition, Main_Turtle serves as the superclass for all subsequent Turtle classes, commencing with Turtle_1 which is a subclass with extended features. Each attribute type is initialized to a constant value for Main_Turtle, then set to a default value for Turtle_1. Turtle_1 inherits all of its superclass' methods, as well as providing additional ones of its own to support more complex needs. Figure 11 shows these additional methods.

Each Turtle object has some basic characteristics to support the drawing routines: the name that identifies it; its location on the drawing screen; and the direction it is heading. Turtles have other features such as setting the color of the pen, setting the width of the pen, and setting the pen on and off. These methods are completely defined in Appendix A.

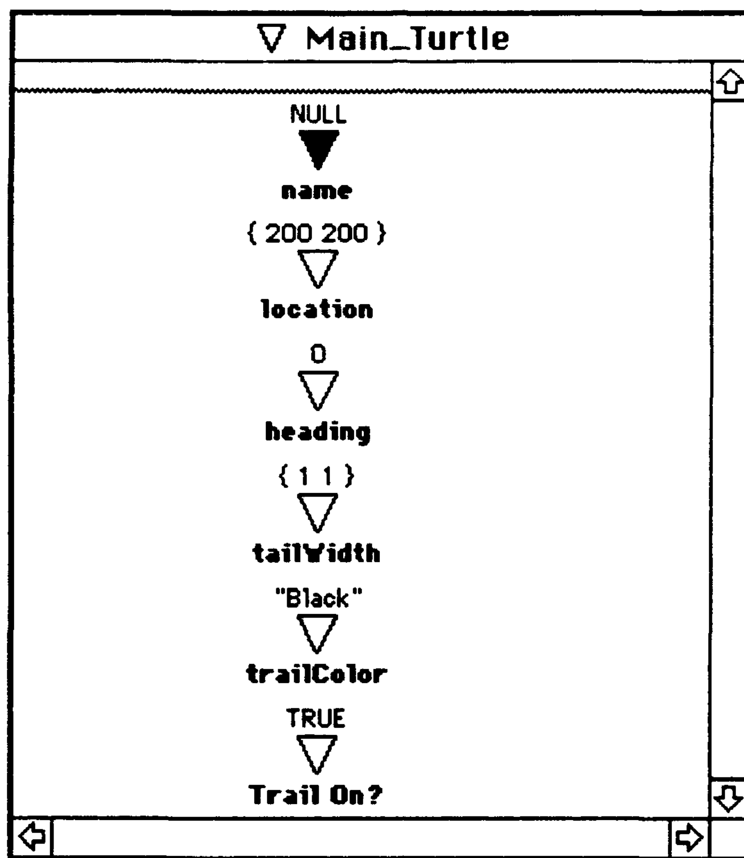


Figure 9: Main_Turtle's Attribute Definitions

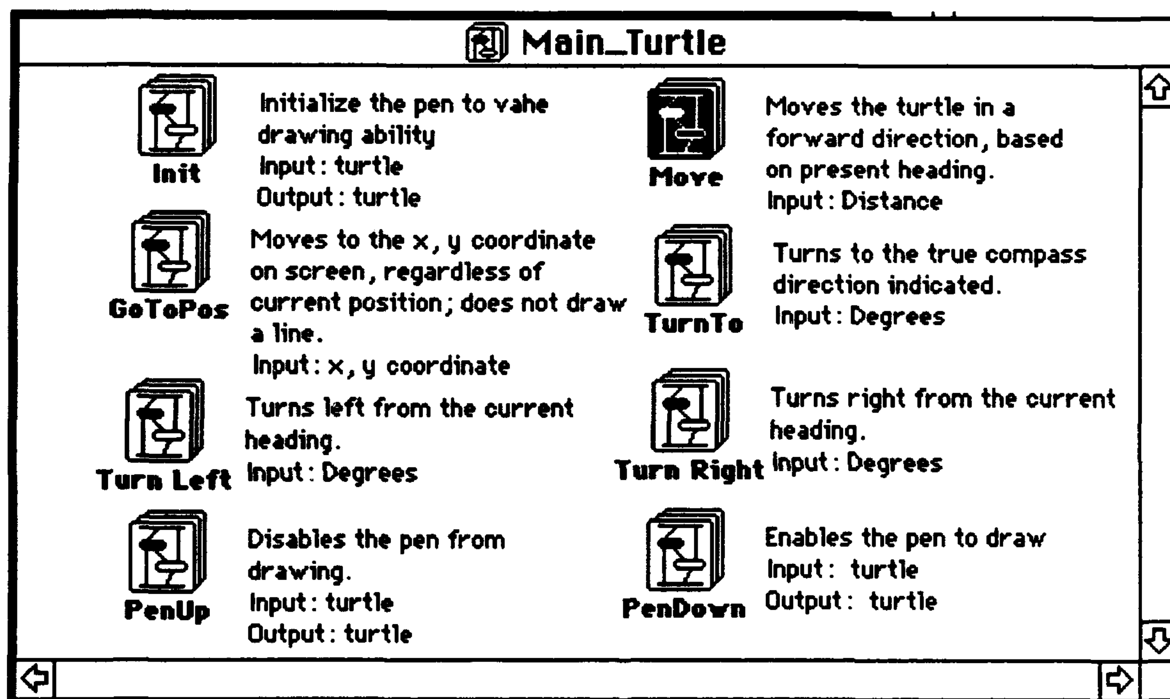


Figure 10: Main_Turtle's Method Definitons

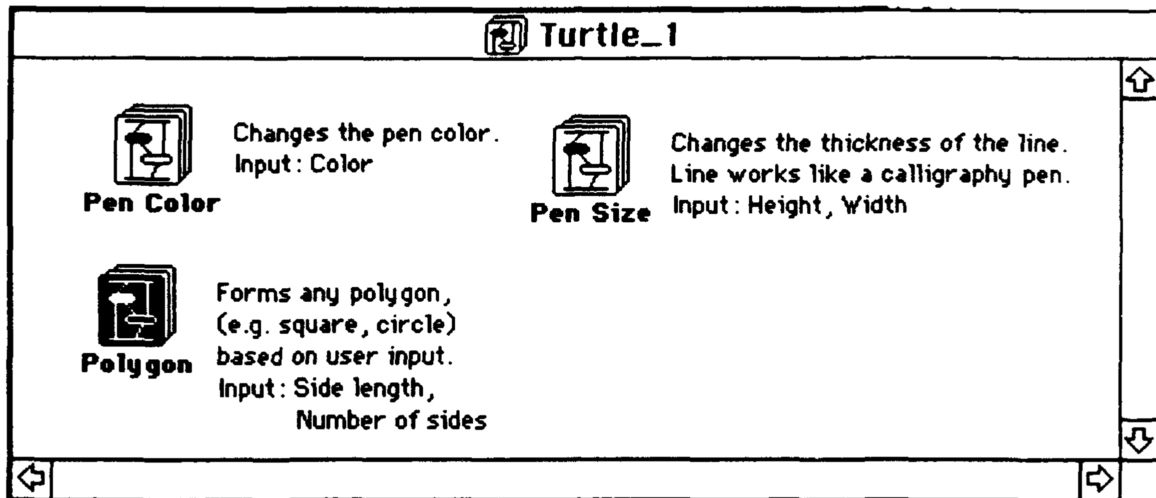


Figure 11: Turtle_1's Method Definitions

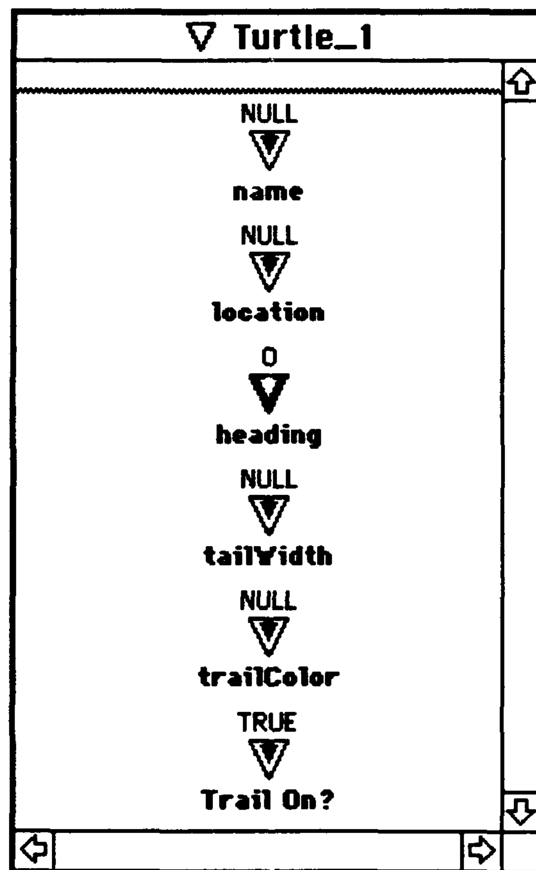


Figure 12: Turtle_1's Attribute Definitions

VI. IMPLEMENTATION OF VOOL

Object oriented programming (OOP), a data-centered view of programming in which data and behavior are strongly linked, is the programming methodology of choice in the 1990s. The goal for all OOP languages is to provide faster development, improved reliability and quality of end products, and easier maintenance and extension. The main objective of implementing Visual Object Oriented Logo (VOOL) is to teach the concepts of object-oriented programming: objects, classes, inheritance, abstraction/encapsulation and polymorphism.

A. OBJECTS AND CLASSES

It is natural to view the world as a collection of objects. Even children too young to program can see, feel, and differentiate between many objects every day. For the beginning programmer, objects provide an intuitive means of organizing thoughts and relating a real world problem to the program to be written. From prior every day experience, the beginner understands that objects have individual characteristics and behaviors, and that they can communicate with each other. From there, it is a natural step to cluster similar objects into groups, called classes.

A class is an abstraction of a specific group of objects which share common characteristics. As a minimum, a class includes the following:

1. The name:

Each class must have a unique name for reference purposes.

2. The attributes:

A set of features for the class are called attributes. Each object will acquire the same set of attributes, although the actual data values contained in these attributes can vary from object to object.

3. The methods:

A library of behaviors for the objects in the class are called methods. The methods are the operators which the class objects are capable of performing, and typically have a number of arguments, or parameters. Although each object has access to every method in its class library, it does not necessarily need them all for a given application.

The class serves as a template for objects which have not yet been instantiated, and identifies the skills and behaviors to be expected when an object is instantiated in the course of writing a program. Classes provide the ability to generalize from a few specific cases to a category of similar cases. In addition, attributes and methods common to all members of the class only need to be stored once, rather than repeated for each object.

Figure 8 shows the three features of a class in our implementation. We created a class and named it Main Turtle. The diagram represents its attributes with a triangle and its methods with a square. These attributes and methods are then further defined in Figures 9 and 10. Whenever the user wishes to create an instance of this class, he/she knows exactly what the attributes will be and which methods will be available for manipulation.

In our implementation, only one object per class may be displayed at a given time in the active window. Initially, this ensures that the programming environment is kept simple and the beginner is not overwhelmed with complex options. A more advanced user, however, might appreciate the ability to have multiple objects of the same class, e.g. two or three turtles, active in the window at the same time, or even multiple objects from different classes. This would provide the opportunity for objects to interact with each other.

B. INHERITANCE

Inheritance is a powerful tool in object oriented programming, enabling the construction of new classes based on the existing hierarchy. The necessity to redesign and recode is reduced, resulting in code reusability and timelier, more efficient programming. In addition, inheritance provides a logical structure for organizing information. Designing a well-defined inheritance hierarchy can assist the beginner in analyzing a problem and programming a solution.

New classes, called subclasses, inherit the methods and attributes of the class above it in the hierarchy. These subclasses can then be specialized by extending their behavior and representation, thus tailoring them to the application at hand. Inheritance means that methods defined in the parent class, (and all ancestor classes), are automatically part of the subclass without needing to repeat the code. One way to specialize a subclass, however, is to override an inherited method. This is achieved by using the same method name, but

writing different code, the new code applies only to that subclass (and any future descendants of it). For example, there could be a method named "Move_Shape", which is located fairly high in the inheritance hierarchy. Moving a square and moving a circle require slightly different code for execution, so Move_Shape could be specialized at subsequent levels in the heirarchy to fit the specific situation. A message to an object will execute the most recently defined method of that name. Only upon failing to find the appropriate method in a class, will the message search higher in the hierarchy until it finds the matching name.

So far, this discussion has dealt only with single inheritance, meaning that each subclass has one and only one immediate parent class. It is also possible for a class to inherit from two or more parent classes; this is called multiple inheritance. The resulting subclass contains the union of its parents methods and attributes, as well as any new methods and attributes defined for that subclass.

Although multiple inheritance may be desirable in certain rare cases, it significantly complicates the structure of a program and can lead to inheritance conflicts. For example, two methods with the same name but different code may inadvertently be combined in one subclass. To guard against such an occurrence, effort must be diverted from productive code writing and expended on conflict resolution strategies. It is our opinion that multiple inheritance has more drawbacks than benefits; it is better to restructure the program and avoid it completely.

Figure 8 shows the inheritance hierarchy for our implementation. Main Turtle, at the top of the hierarchy, contains what we considered to be the core capabilities and attributes needed for a turtle object, (Figures 9 & 10). With these features, a beginner has an object on the screen which he can then manipulate with simple visual code. Each piece of code has a direct effect on the object, for example turning it right or left, thus providing satisfying feedback to the beginner. When the beginner is ready for additional capabilities, he can progress to the next level in the hierarchy.

Turtle_1 is a direct descendant of Main Turtle, and inherits all of its attributes and methods. As Figure 12 indicates, however, Turtle_1 does not inherit the actual *values* of the attributes; rather, they are reset to null or a default setting. Turtle_1 does inherit the ability to perform all the methods defined in Main Turtle. As shown in Figure 11, these methods do not need to be redefined for descendants, which is a primary advantage of inheritance. Turtle_1 can then increase its capabilities by adding methods particular to that class, in this case Pen Color, Pen Width, and Polygon.

The relationship between Main Turtle and Turtle_1 demonstrates to the beginner the concept of inheritance. The beginner is able to play with a working model and observe inheritance in action, rather than just trying to grasp the idea from a book. For example, in order to execute the method Polygon, Turtle_1 must call on the methods Move and Turn Right which it inherited from Main Turtle. Based on user input for the parameters, the result in this case is a square, (Figures 13-16).

Once the beginner feels comfortable with the concept of inheritance through using the classes provided, he can then apply his new-found knowledge and create classes of his own. These classes could descend directly from Main Turtle, or could continue down the tree from Turtle_1.

C. ENCAPSULATION/ABSTRACTION

Encapsulation is the process of hiding all the details of an object that do not contribute to its essential characteristics. It is also referred to as abstraction or information hiding, where the object is a kind of black box. It provides the means by which the internal details of a specific method, and/or the different attributes of an object, are implemented and hidden from the outside objects. Encapsulation supports code reliability and extensibility, and also allows for integration.

The reliability of the end product is determined by reliable modules and reliable methods of integrating those modules together. In addition to the dependability of individual modules, there should be a proven means of putting individual modules together into a working whole with a smooth and easy integration.

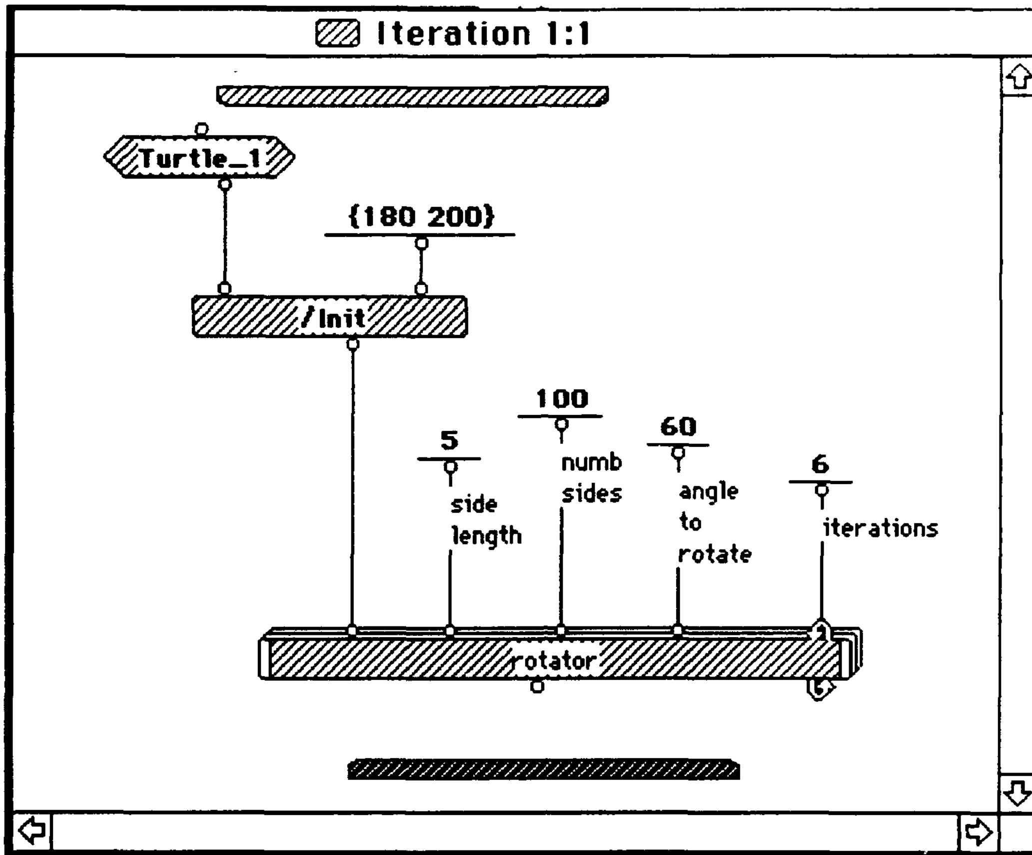


Figure 13: Iteration for Circle.

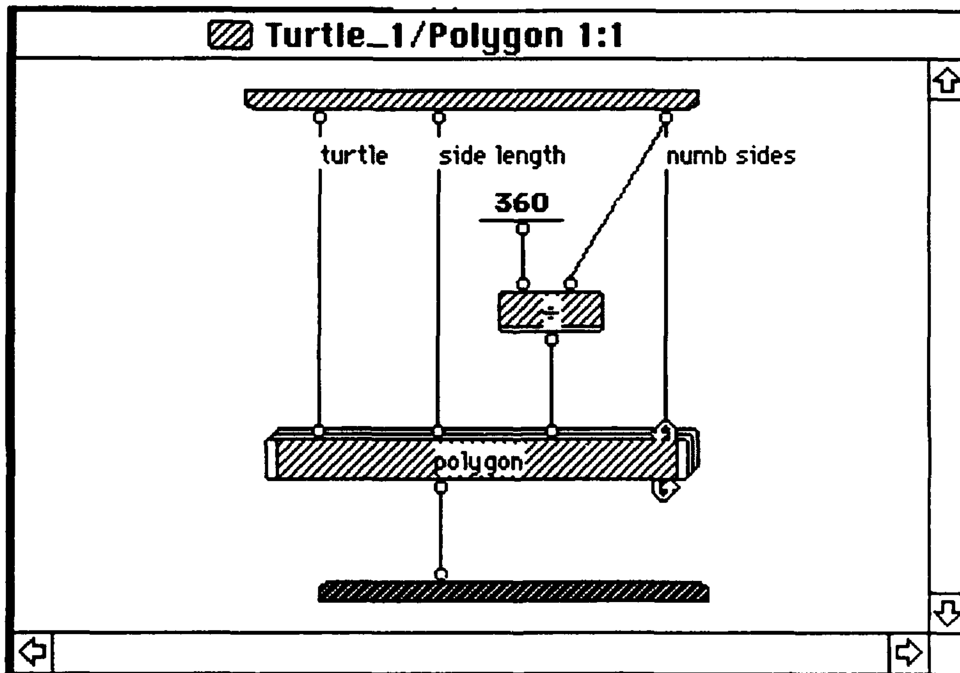


Figure 14: Turtle_1 / Polygon.

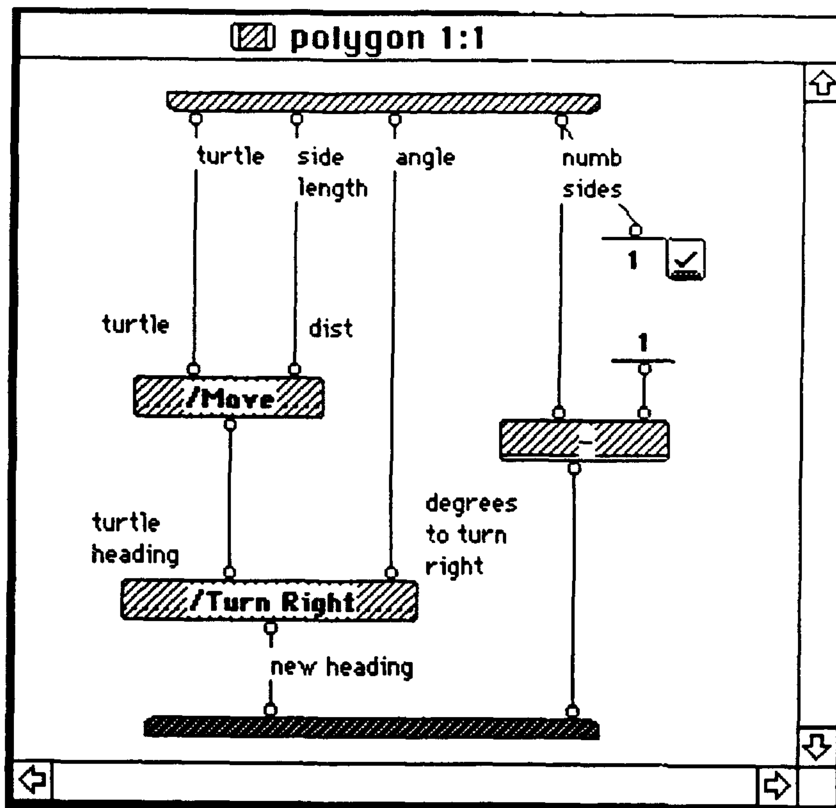


Figure 15: Polygon.

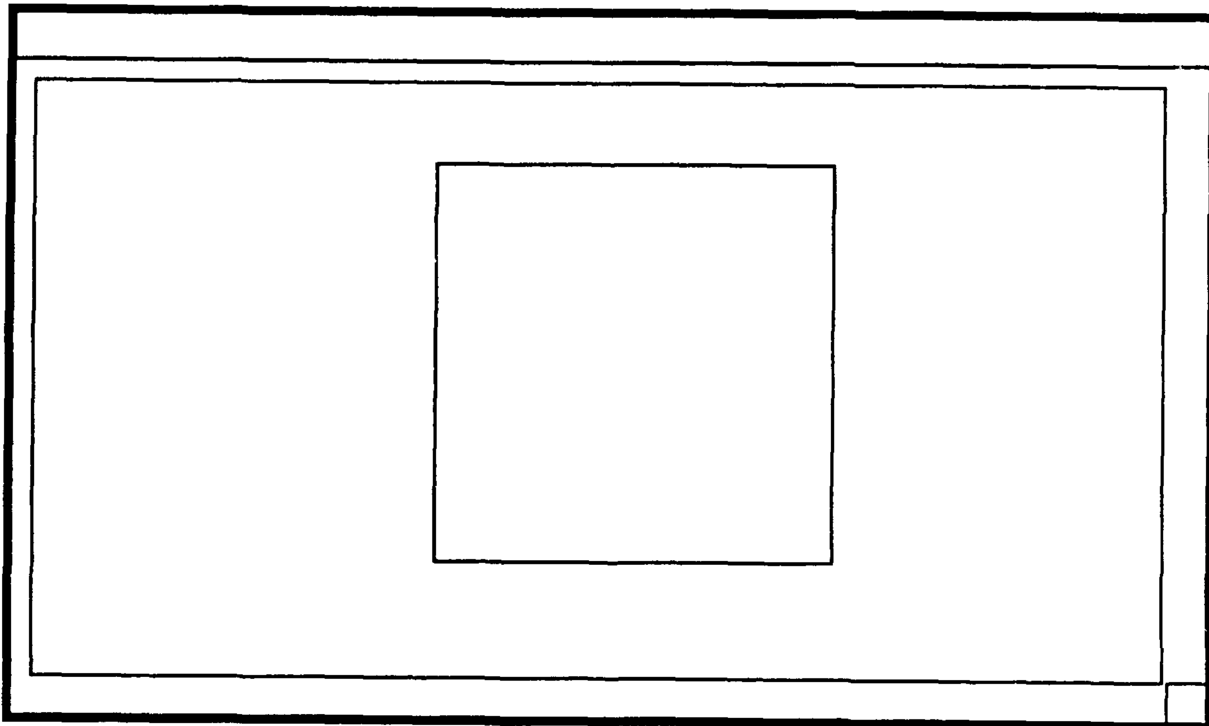


Figure 16: Square.

We used Main_Turtle's methods and attributes in Figure 9 & 10 to illustrate the notion of encapsulation. The user doesn't need to know how the different methods or attributes of Main_Turtle are implemented; he simply needs to know the names of the messages to be passed, and what will be returned by the object. For instance, with the visuality of Prograph, the user can see what values are flowing into an operation and what values are returned to the output terminal. As shown in Figure 17, the method Move, like all other methods of Main_Turtle, is receiving one input from the left terminal of type Main_Turtle object, a second input from the right terminal provided by the user, and finally produces an output of type Main_Turtle object. This kind of implementation of the methods is specific to our prototype, but not necessary bound to it. The design is determined by the developer.

Each method constitutes a specific and independent module with its own implementation. Each of the methods can be used as a seperate module over and over again by many objects without modifying its implementation. Draw Train in Figure 17 shows how the methods Init, Move, TurnTo and Turn Right work within the Main_Turtle. During execution of the universal method Draw Train, the user can see the different values flowing into or from any operator by double clicking the desired terminal.

Similarly, the user may or may not know the different names of Main_Turtle's various attributes, but can still send messages to a specific object. He doesn't need to know how each of them is implemented in order to use them.

The modularity concept of producing code guarantees the reliability of the module and allows the modification and improvement of the code without affecting the user's access to the object. We can assume that each individual module is reliable and can be easily extended and smoothly integrated to other applications without compromising the reliability of the final product.

Using encapsulation to maintain modularity provides the necessary working environment for many developers to concentrate on seperate modules that would be integrated later in the same project.

Even if there is no public and private specification in the Prograph environment, as in C++, it is possible to prevent the user from accessing the hidden sector to ensure greater module reliability. All modules would be in the "execute only" form, where the user is allow to access them, but can not modify their internal implementation.

The implementation of this prototype would provide a developer with the necessary built in modules to construct a more complex program. With the encapsulated classes, we can focus on teaching the notion of object oriented programming instead of confusing the user with all the low level implementation.

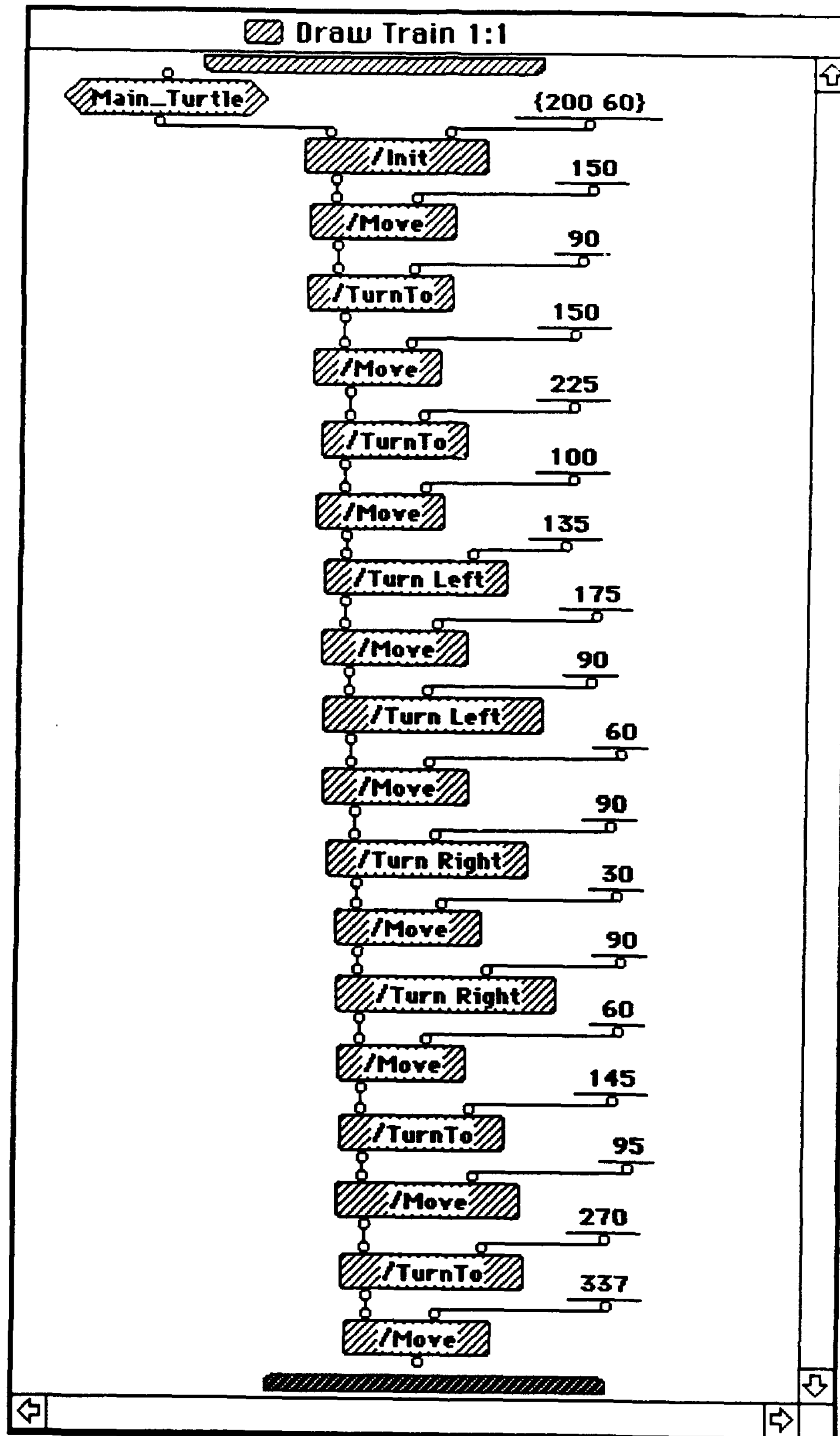


Figure 17: Draw Train.

D. POLYMORPHISM

Polymorphism is the phenomenon that occurs when messages of the same name are sent to different objects. Each objec responds with an appropriate method for its class. Polymorphism supports code sharing and extensibility, and has two aspects: the non-polymorphic operator that always refers to the universal method (designed to be used by any object class), and the polymorphic operator which refers to different methods.

A polymorphic operator is always prefixed with one forward slash (/) to refer to a method defined in the class of an object, or two forward slashes (//) to refer to a method defined in the class system. A non-polymorphic operator is invoked on its own without a slash, and always refers to the universal method. Polymorphism allows developers to add methods with the same name to classes that share some commonality and therefore use the same name to denote the specific function. Which method to execute depends to the object that flows into the operation.

The extensibility of a module is characterized by the fact that outdated or faulty modules can be replaced with a new module without requiring any changes to other modules. When the same module requires additional functionality, a new module could be created with the same name to replace the first module, by sharing the original code and adding more features. This characteristic of extensible modules will allow easier program maintenance and extension.

In almost all our examples provided, we illustrate the polymorphic operator with the single forward slash. The non-polymorphic operator is only used for the implementation of the prototype with the universal method Get Canvas Window.

E. ITERATION

Iteration plays an essential role in practical programming by reducing the amount of code needed to perform identical steps. The ability to repeat a certain segment of code not only reduces the apparent complexity of the program, but also decreases the chances of an error being reintroduced into previously debugged code. Error reintroduction could occur lines if code had to be manually retyped each time repetition was desired.

In VOOL, iteration provides a powerful tool to the programmer, enabling a beginner to create complex pictures with relative ease and speed. The model in Figure 18 depicts one such example. This figure is actually composed of one simple pentagon rotated around a common point, but the result is a multi-faceted display of interwoven patterns.

The method "Polygon" is defined for Turtle_1 and allows the user to provide input for the desired size and number of sides. Once one polygon is drawn, the user then needs to turn the turtle slightly, (using an inherited method from Main Turtle), and repeat the process. This is where the power of a looping ability comes into play. The Prograph programming environment provides a system level icon to perform this task.

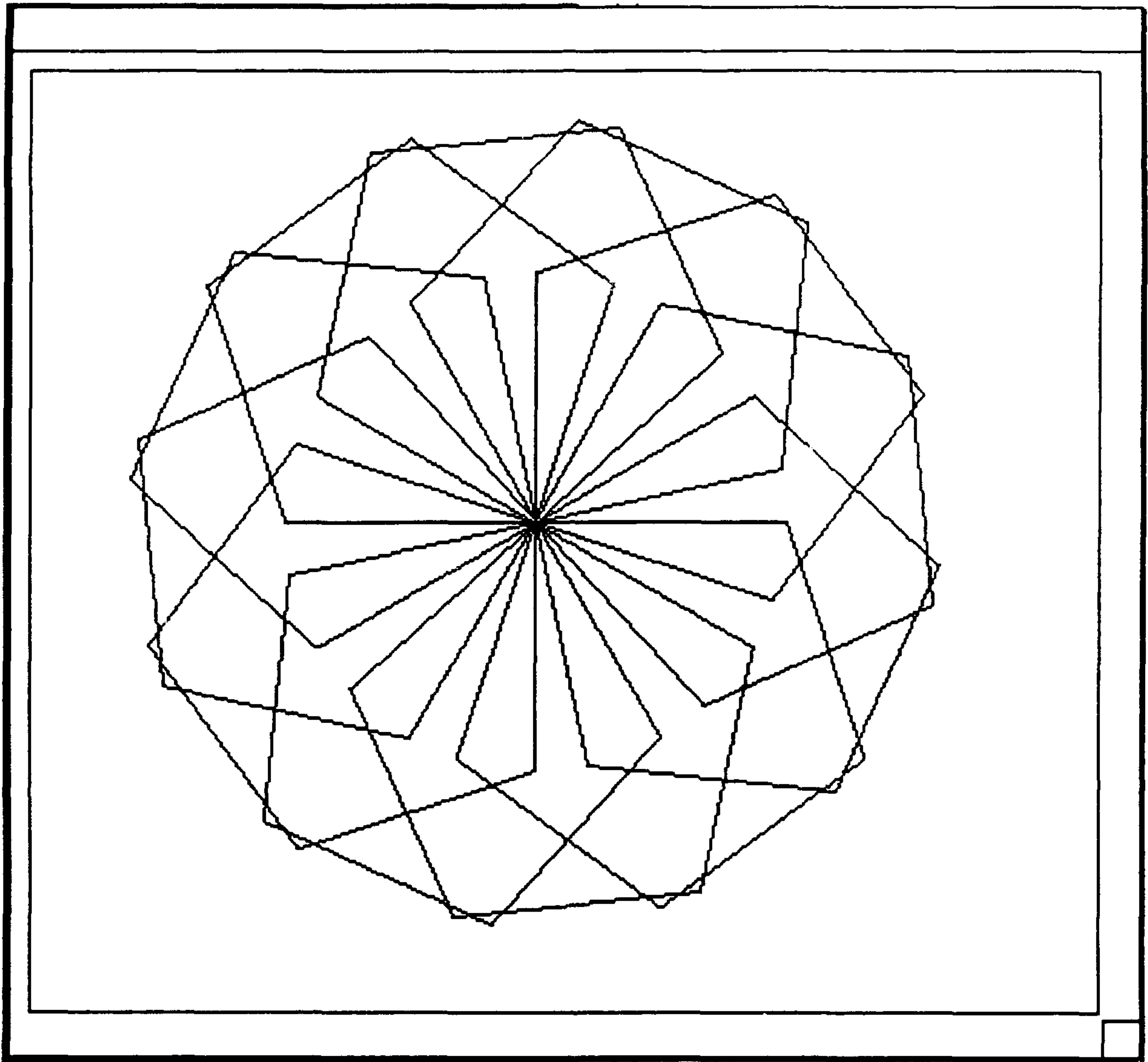


Figure 18: Iteration / Polygon.

The icon itself is multi-layered in appearance with circling arrows, giving an intuitive indication

that the local method will be repeated for the specified number of times. Figures 19 and 20 show the code used to produce the rotated pentagon.

Once the user grasps the concept of iteration, he can use it to write versatile, time-saving methods. For example, the method "rotator" can produce very different results depending upon the user's input. Simply by changing four parameters, the same method will draw a flower-like arrangement of circles, (see Figures 21 and 22).

F. SELECTION

The beginner must first concentrate on writing simple code with a linear progression. In the case of VOOL, this would mean a program that flows from the top of the screen to the bottom, following only one path. Once the beginner is ready to solve slightly more complex problems, however, he must have the ability to provide different paths of action depending on decision criteria. These paths of action would most likely involve multiple screens and require the user to exercise some logic in constructing the conditional executions.

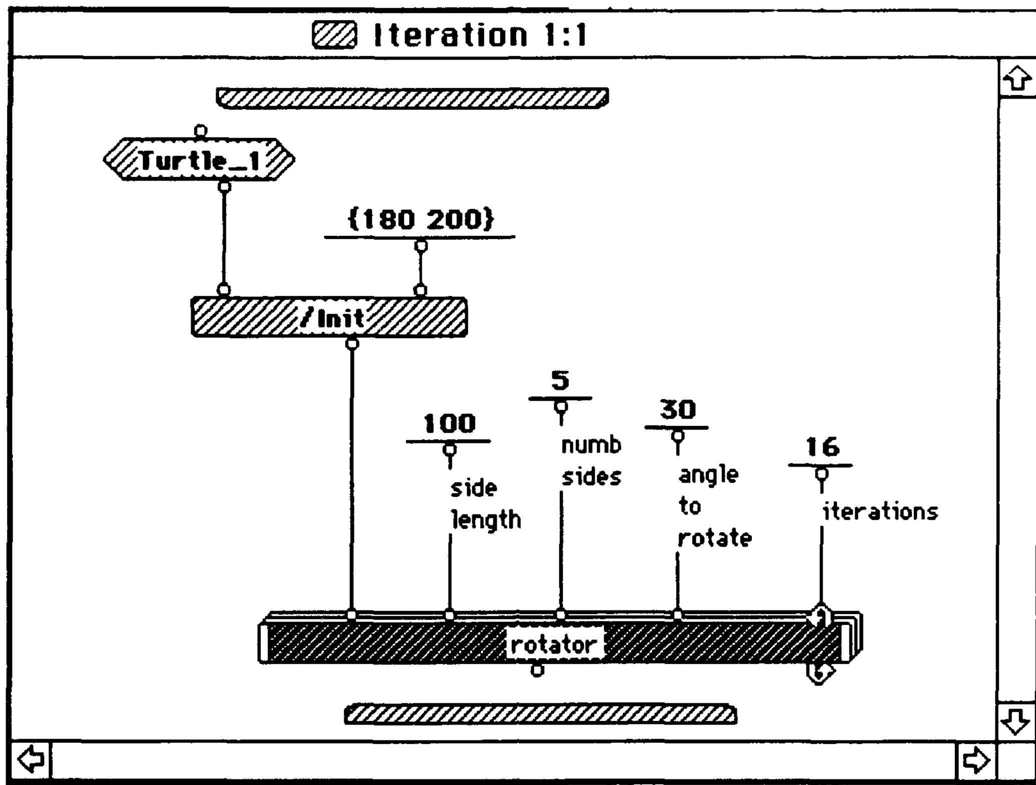


Figure 19: Iteration for Polygon.

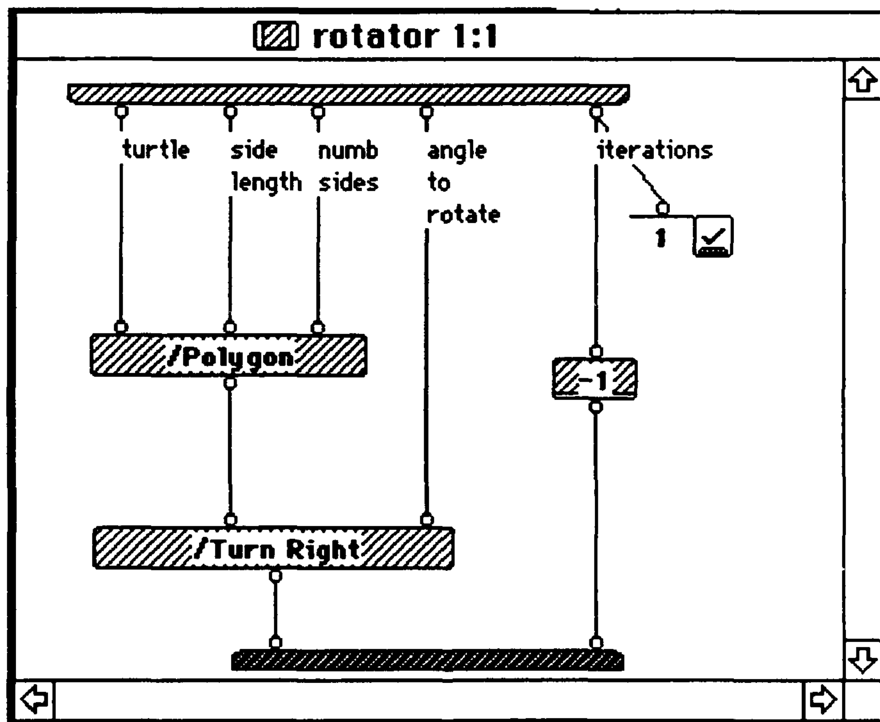


Figure 20: Rotator.

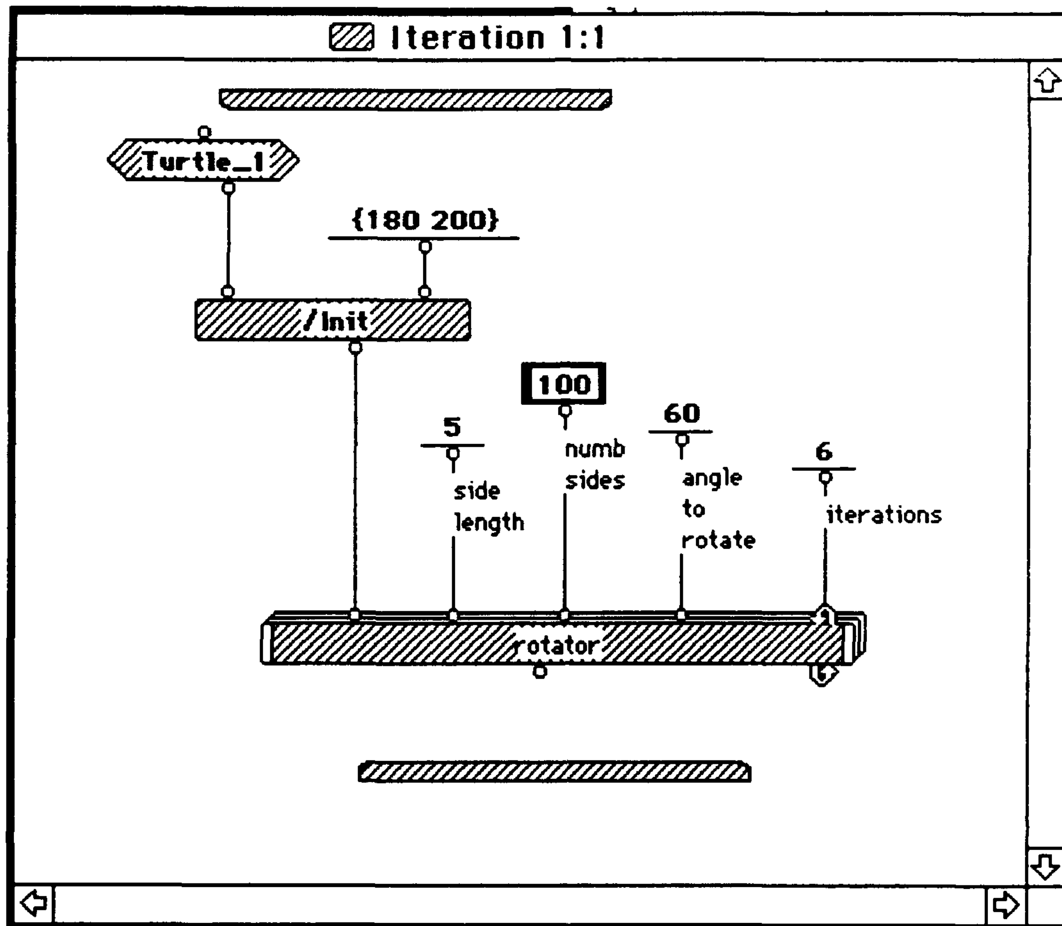


Figure 21: Ieration / Circle.

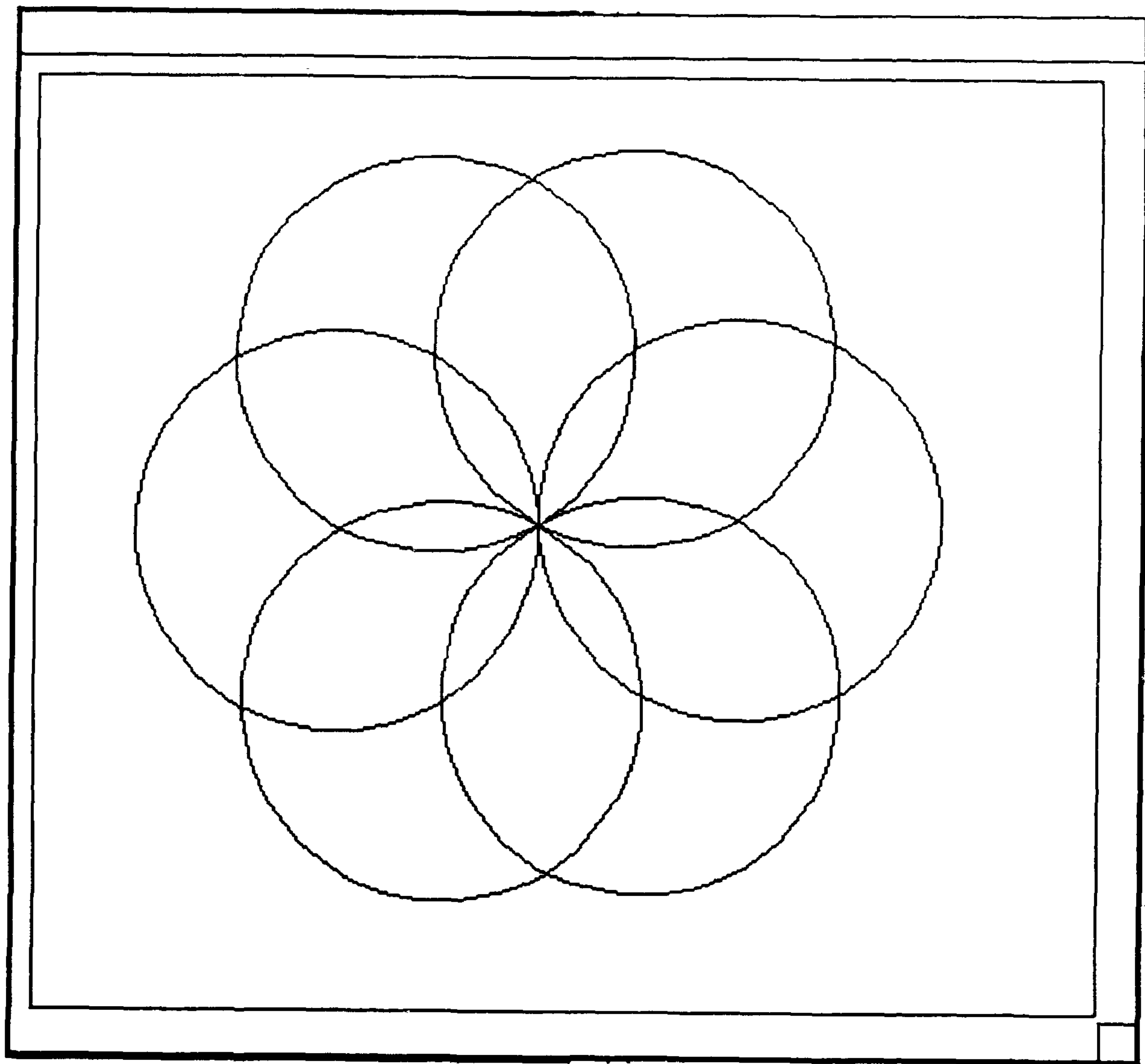


Figure 22: Circles.

The Prograph environment provides a complete and flexible range of functionality with its case structures, control annotations, and "Match" operations. The "Move" method defined for Main Turtle provides an example of path selection. When the user orders the turtle to move, the code to be executed depends upon the turtle's heading since there are four trigonometric solutions based on the four quadrants of a graph. Figure 23 shows the first of the four possible paths. Use of the "next case on failure" feature is demonstrated here. If the turtle's current heading is within the specified range, then the first case is executed; if the comparison is a failure, however, then control passes to the next case for further comparison.

Since "Move" is inherited from Main Turtle, VOOL allows the beginner to use it without needing to understand what is within the method's "black box". This enables the beginner to concentrate initially on fundamental programming skills, and to receive positive motivation by quickly accomplishing apparently advanced tasks. With a solid groundwork of linear programming experience, the user is then ready to progress to more advanced problem solving, including code requiring selection.

This chapter has discussed the core concepts of object oriented programming, specifically: classes, objects, inheritance, abstraction, encapsulation, and polymorphism. In addition, the key capabilities of iteration and selection were covered. In each case, the discussion addressed how best to present these concepts to a beginning programmer, and provided examples for clarification.

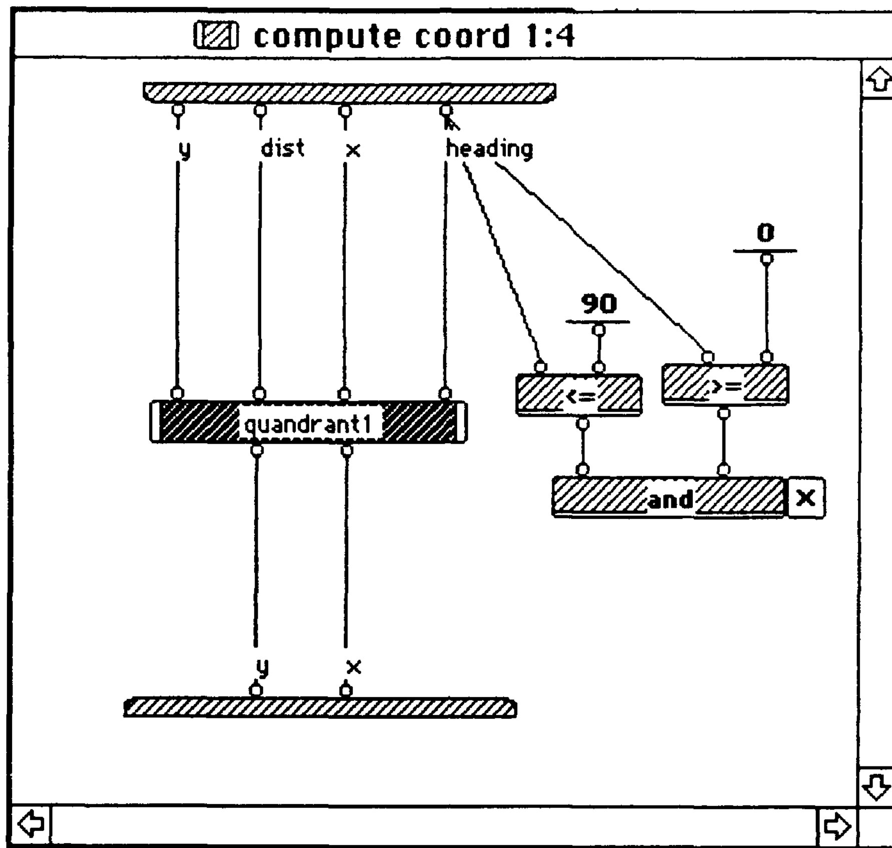


Figure 23: Compute Coord (Selection).

VII. CONCLUSION AND RECOMMENDATIONS

A. VISUAL APPROACH: PROS AND CONS

Using a visual approach to teach object c[illegible]amming has many advantages, but suffers from drawbacks as well.

On the positive side, the visual approach is an excellent means of introducing object oriented concepts. The concepts of classes, objects, encapsulation, and polymorphism can be abstract and difficult for a beginner to grasp. A visual programming language allows the beginner to actually see a class hierarchy, and watch how objects interact with each other. By manipulating objects which are visible on the screen, the beginner gains practical experience with object oriented features such as polymorphism and inheritance. Furthermore, it's easy to identify the attributes and methods that comprise an object, (especially guided by Prograph's triangular and square icons), and watch how certain of these components are passed along through inheritance. When learning a text-based object oriented programming language, the beginner must struggle with these abstract concepts without benefits of visual representation. This requires a certain level of mental sophistication from the beginner, and perhaps experience with other programming languages. While this may not be a problem for a college freshman, it is certainly a barrier to a young grade school child. Children must see and do and experiment. An esoteric, "chalk-talk" explanation would not only be above their level of comprehension, but would most likely dull their enthusiasm for attempting anything further with the language.

The visual approach not only conquers the level of understanding barrier, but circumvents two other barriers as well; those of language and reading ability. Beginning programmers of any age, from first grade through college and beyond, may be learning in a spoken language which is not their native tongue. A visual environment allows them to concentrate on the concepts without the added difficulties of translating text and worrying about syntax. Likewise, even youngsters who cannot yet read, can still draw simple shapes and start developing an intuitive feel for object oriented programming.

On the negative side, visual programming skills are not immediately transferable to text-based languages. Although the student would have a sound grasp of the object oriented concepts themselves, he would still need to study the specifics of the text-based language. This may not be a problem once visual object oriented languages become more prevalent, but presently the most popular OOPLs, such as C++ and Smalltalk, all use written code. Purely from a practical viewpoint, it would be beneficial for a programmer to master both paradigms.

B. BUILDING THE PROTOTYPE

The primary difficulty in building our prototype was in learning the Prograph environment. Prograph is very powerful, with many features designed to assist the programmer; once we gained experience with these features, our task was actually expedited. For example, Prograph provides a complete system library with base classes and primitive operations, as well as an extensive error tracing capability. In addition,

working in a windows environment with a mouse and icons was easy and familiar, facilitating the prototype development.

C. ASSESSMENT

How well does our prototype meet the original problem statement? To review, our research was intended to address how best to teach beginning programmers the necessary skills of object oriented programming. To accomplish this, we first discussed a current programming language, LOGO, used predominately in educational settings as a first language for children. We then addressed LOGO's strong points and shortcomings, paying particular attention to the use of a graphical turtle as a tool for conveying the potentially abstract concepts of the object oriented methodology. Finally, we explored the benefits of a visual approach to learning, and proposed a prototype called Visual Object Oriented LOGO.

We feel our prototype is very successful in presenting object oriented concepts in an intuitive manner that invites creative experimentation. The Main_Turtle is a real-world object that has the potential to promote interaction with even the youngest programmer. Turtle_1 guides the beginner to a slightly more advanced level, plus introduces the concept of inheritance. With this sound groundwork, the user is then equipped to create his own classes for further exploration.

Due to the power of Prograph, our prototype is capable of a wide range of applications, and is thus suitable for use in elementary schools as well as at the college

level. A first grader could be challenged with drawing a square, while those in an introductory college course could model more complex real-world problems. True beginners with no programming experience whatsoever can learn how to manipulate the turtle and have fun. Fun is a key element, in that it establishes a positive attitude and encourages a potentially life-long commitment to learning. Beginners who are new to object oriented programming, yet have experience with other languages, can concentrate on mastering the basic object oriented concepts and can progress quickly to more advanced programming.

The visual programming approach may not facilitate learning a specific text-based language at a later date, but the value of our prototype is that it provides a solid foundation in object oriented programming concepts. Mastery of these concepts is an invaluable advantage in learning other languages, allowing the student to concentrate simply on syntax and style. Overall, our prototype has the potential to make a significant contribution towards educating the next generation of object oriented programmers.

D. RECOMMENDATIONS

Collecting empirical data to support the viability of our prototype would be a lengthy process and is beyond the scope of this thesis. As a matter of future research, however, it would be enlightening to conduct a study of how successfully this prototype actually performs in teaching object oriented programming. Ideally, the study would involve incorporating VOOL into the curriculum of classr [illegible] om first grade to college,

and would take place over the minimum of a year. A pilot program might have two groups of students: those learning object oriented programming through traditional text-based methods, and those using the VOOL approach. At various points throughout the study, the two groups could be compared on the basis of how easily they assimilated and used new concepts, how quickly they progressed from one level of difficulty to the next, and how easily they were able to transfer skills from one object oriented programming language to another. Data could also be gathered on whether one approach was better than the other for a particular age group, or whether there was a universally preferred method.

It is our hypothesis that, were such a study to be conducted, it would showcase VOOL as the desirable approach, simultaneously applicable to all age groups, skill levels, and degrees of application complexity.

APPENDIX A - USER COMMAND AND METHOD DEFINITIONS

A. MAIN_TURTLE METHODS

1. Init

Description: Initializes the pen to start drawing from a particular position in the drawing area.

Input: Turtle; Two numbers (X-vertical displacement and Y-horizontal displacement)

Output: Turtle

2. Move

Description: Moves the turtle in a forward direction based on the nature of the present heading.

Input: Turtle; Number (distance to move)

Output: Turtle

3. GotoPos

Description: Moves the turtle to X and Y coordinates on the screen without drawing a line regardless of current position.

Input: Turtle; Two numbers (X-vertical displacement, Y-horizontal displacement)

Output: Turtle

4. TurnTo

Description: Turns the turtle's heading to the true compass direction indicated in degrees.

Input: Turtle; Number (angle to turn to in degree)

Output: Turtle.

5. Turn Left

Description: Turns the Turtle's heading to left from the current heading.

Input: Turtle; Number (angle to turn to in degree)

Output: Turtle

6. Turn Right

Description: Turns the Turtle's heading to rigth from the current heading.

Input: Turtle; Number (angle to turn to in degree)

Output: Turtle

7. PenUp

Description: Disables the pen from drawing.

Input: Turtle

Output: Turtle

8. PenDown

Description: Enables the pen to draw.

Input: Turtle

Output: Turtle

B. TURTLE_1

1. Polygon

Description: Creates any types of polygon (square, rectangle, circle etc) depending on the user's inputs.

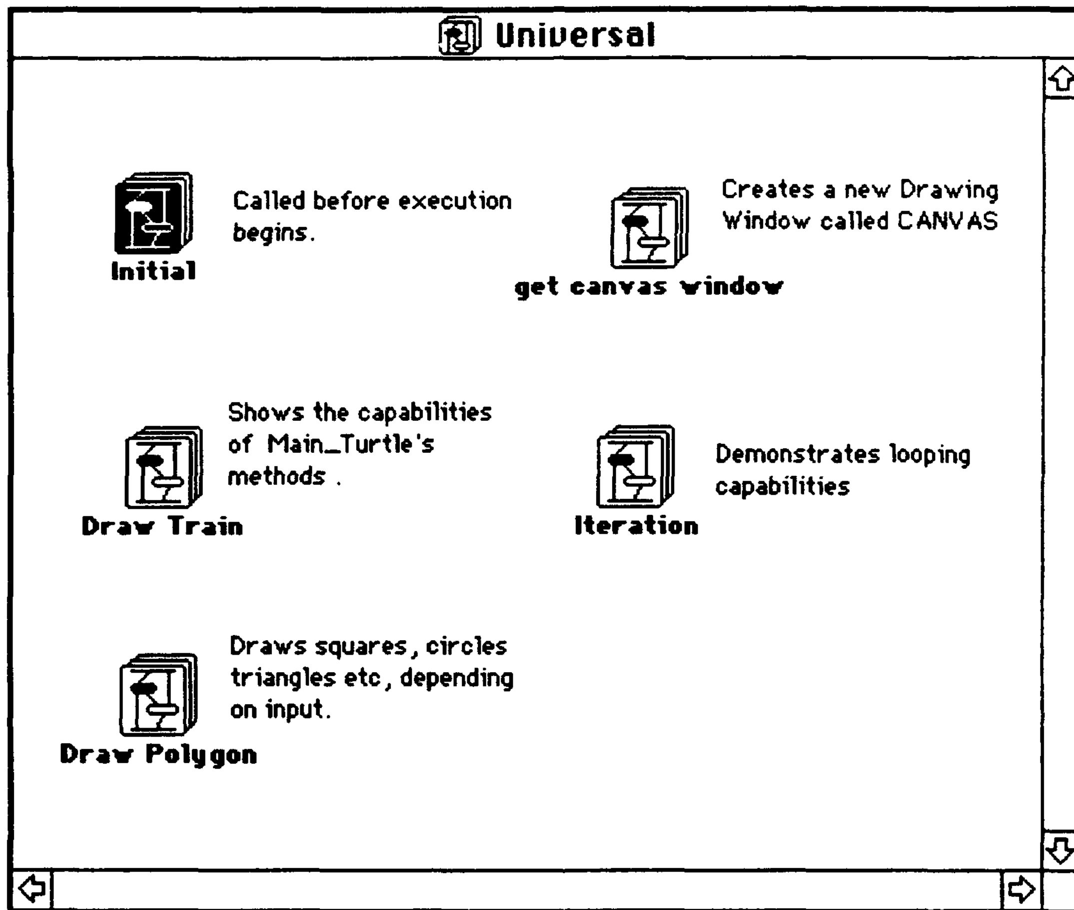
Input: Turtle; Side length; Number of sides

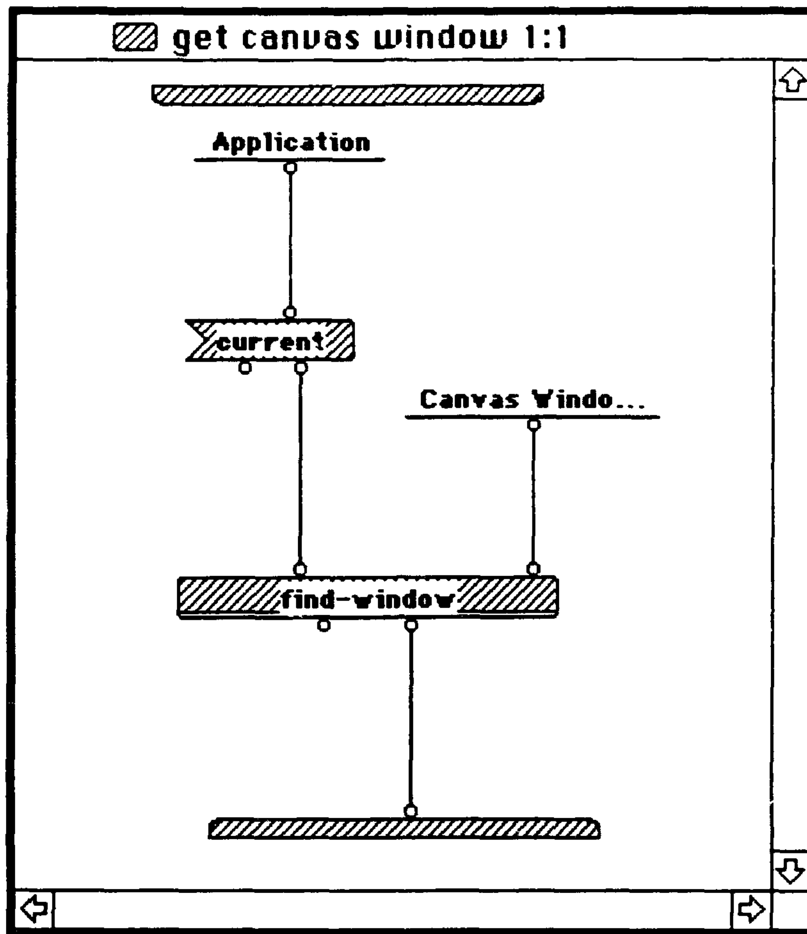
Output: None

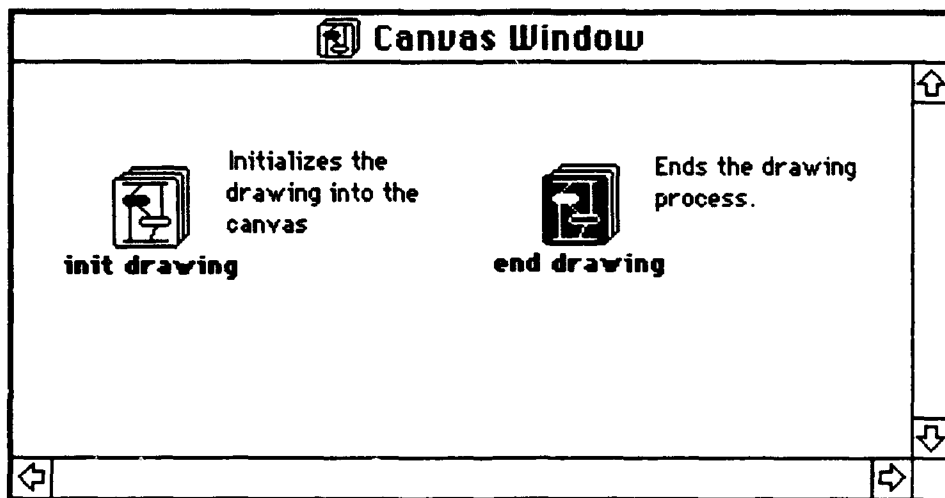
2. Pen Color and Pen Size

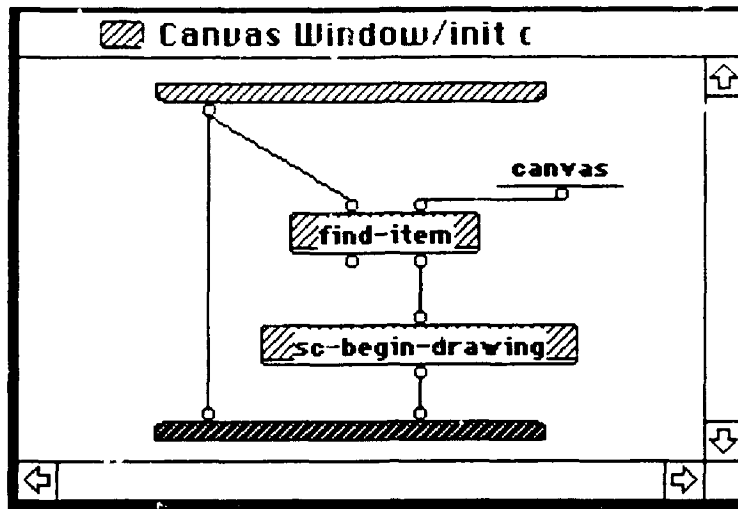
These methods have not been implemented yet.

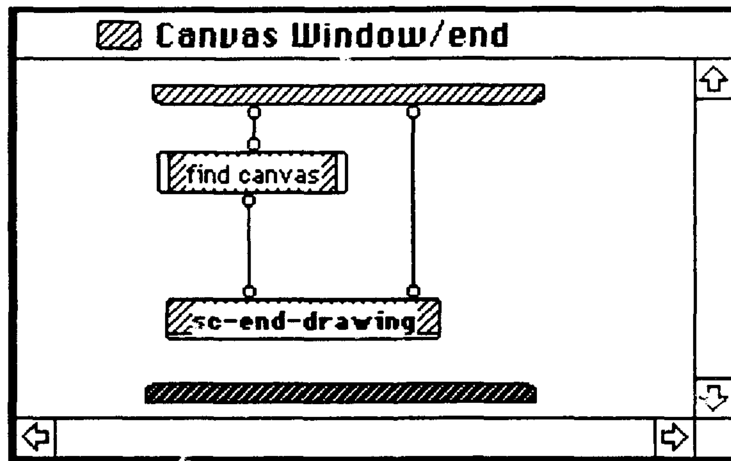
APPENDIX B - TURTLE GRAPHICS' SOURCE CODE

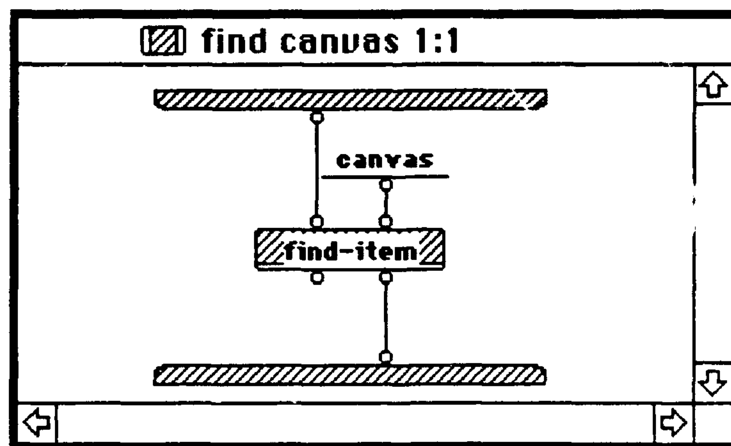


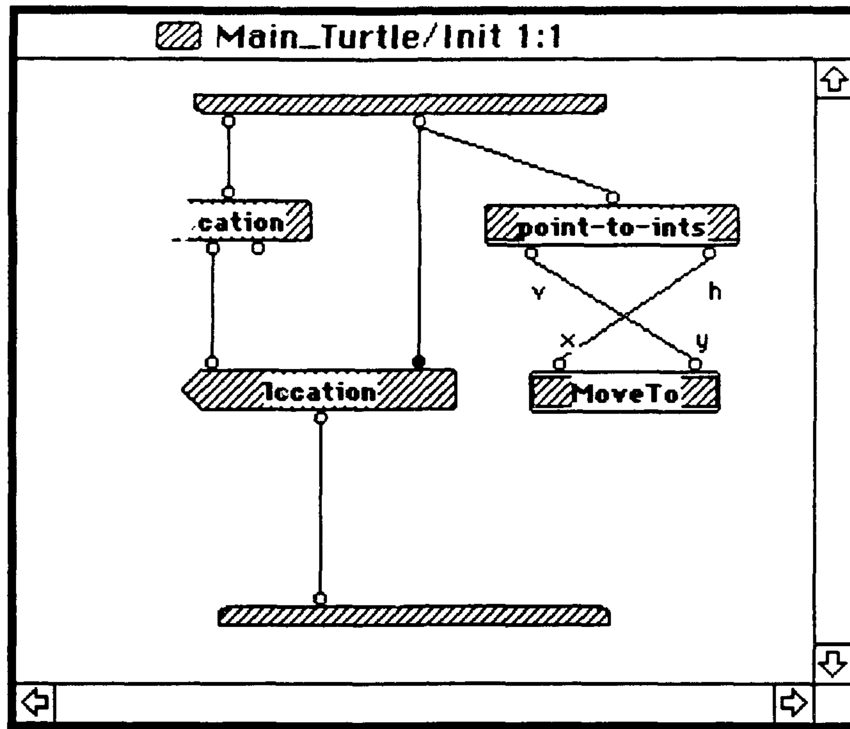

get canvas window 1:1
Application
current
Canvas Windo...
find-window

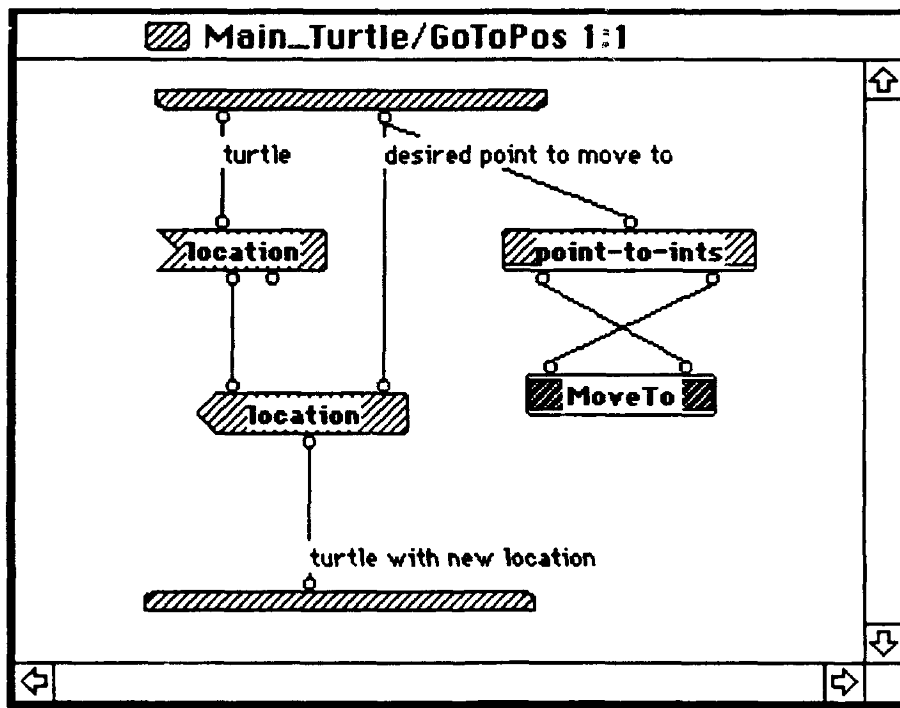

Canvas Window
Initializes the drawing into the canvas
init drawing
Ends the drawing process.
end drawing

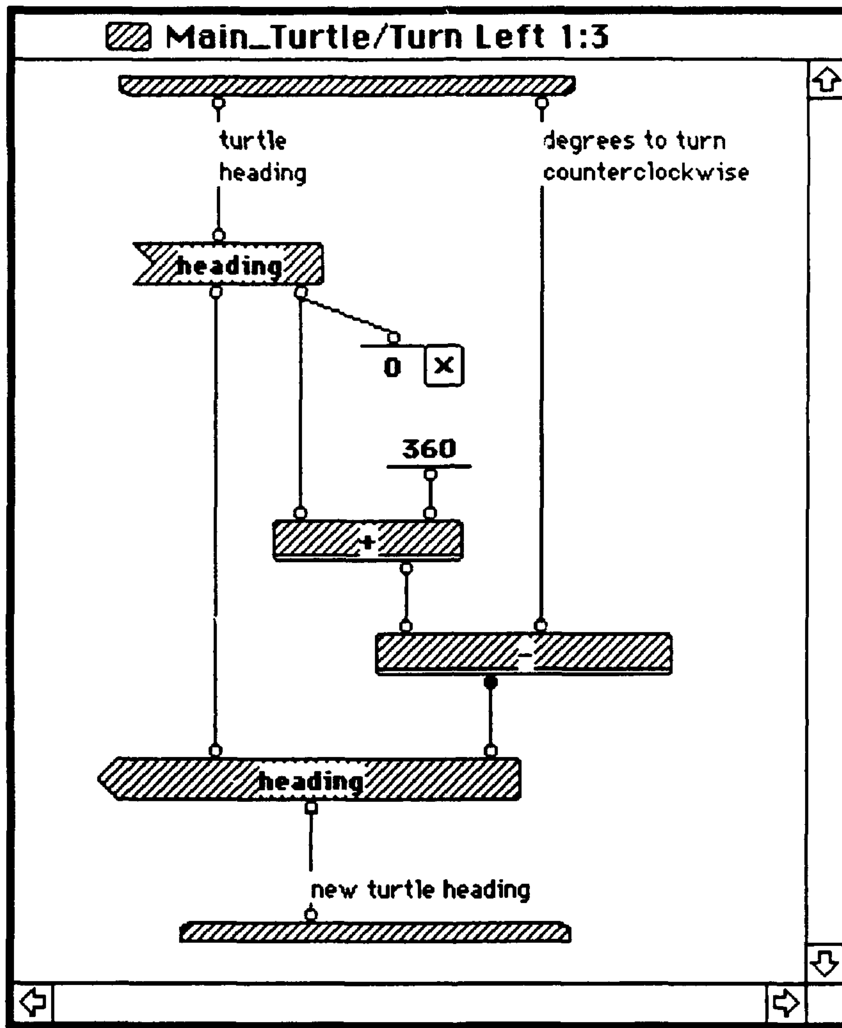

Canvas Window/init c
canvas
find-item
sc-begin-drawing

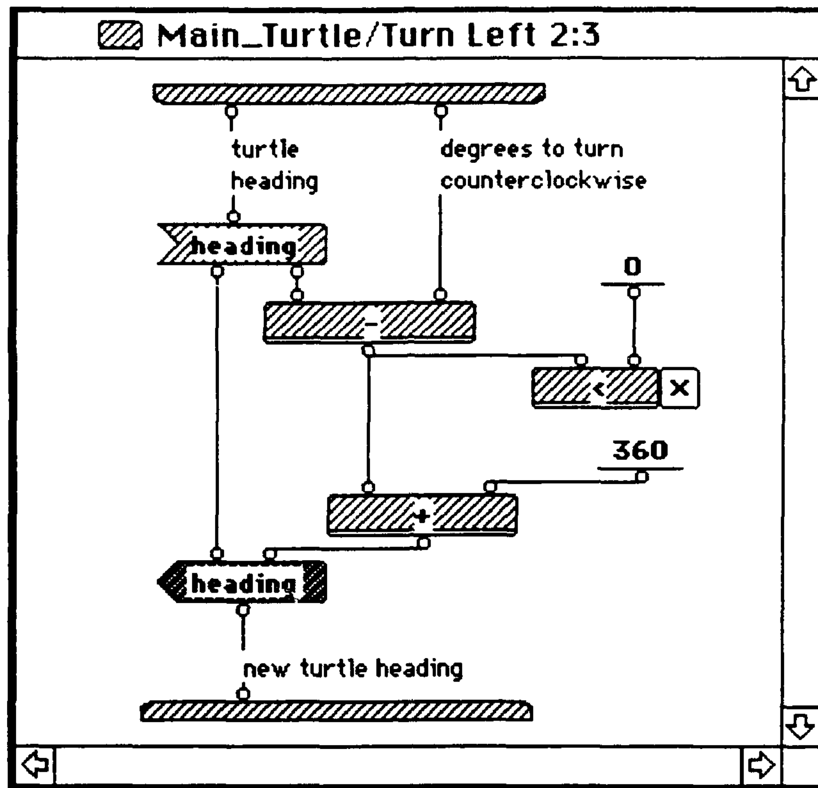

Canvas Window/end
find canvas
sc-end-drawing

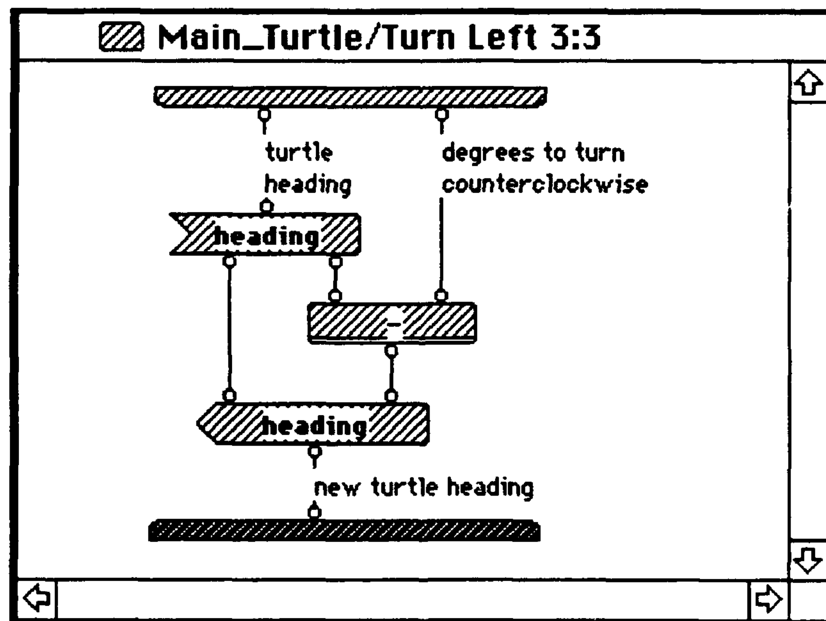

find canvas 1:1
canvas
find-item

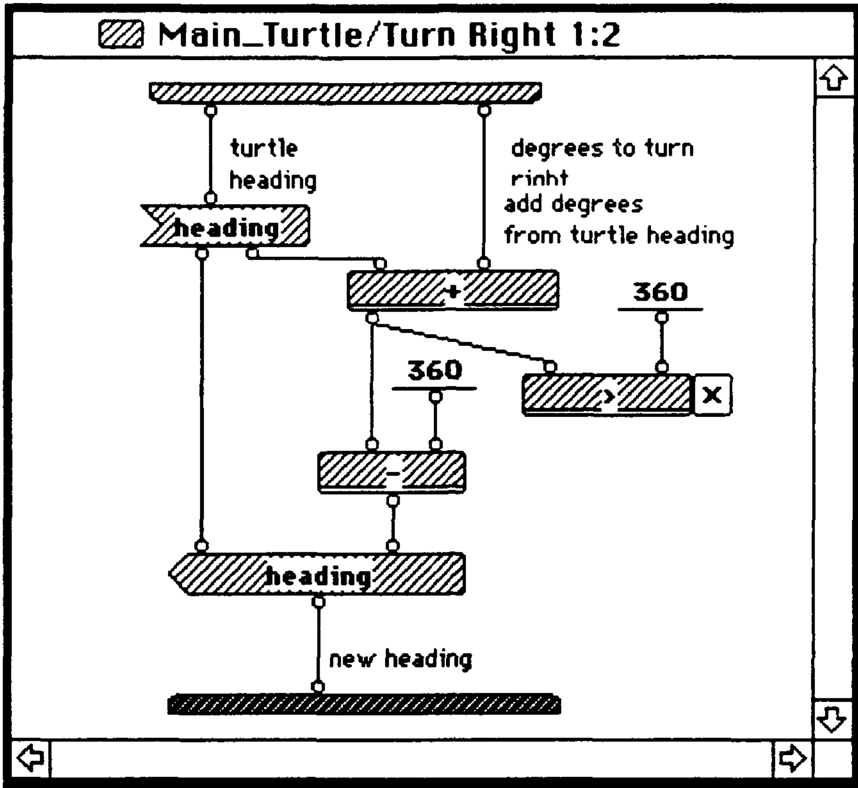

Main_Turtle/Init 1:1
cation
point-to-ints
v
h
x
y
location
MoveTo

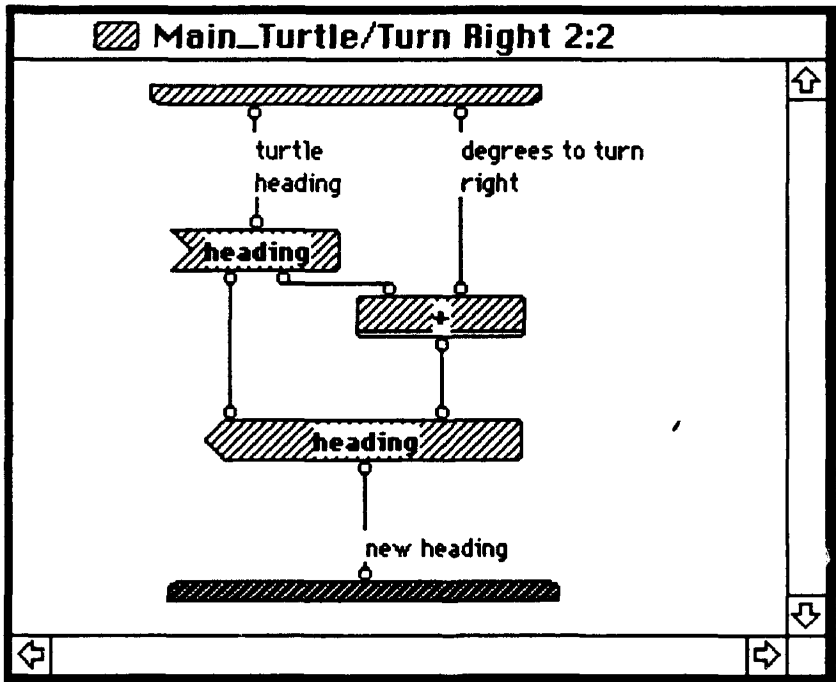

Main_Turtle/GoToPos 1:1
turtle
desired point to move to
location
point-to-ints
location
MoveTo
turtle with new location

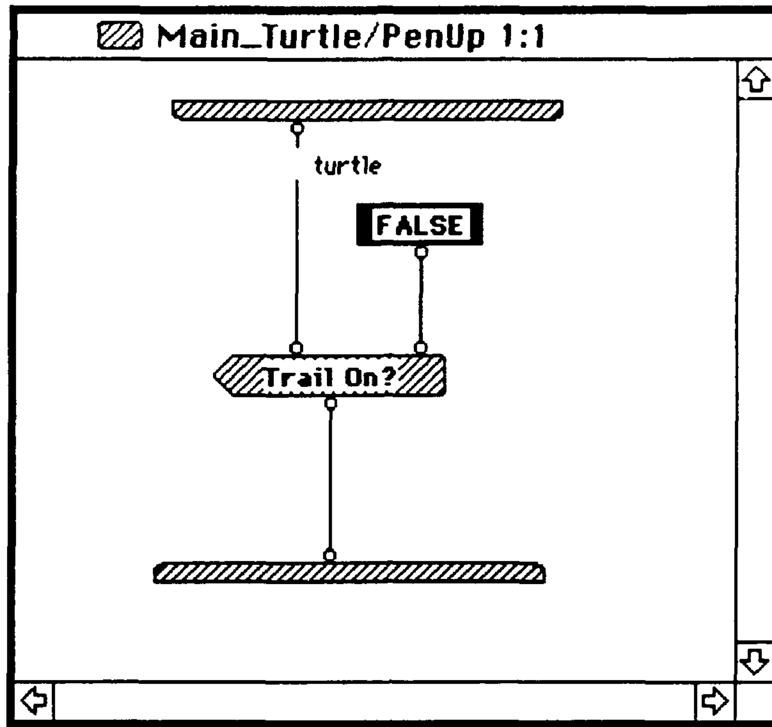

Main_Turtle/Turn Left 1:3
turtle
heading
degrees to turn
counterclockwise
heading
0
360
+
-
heading
new turtle heading

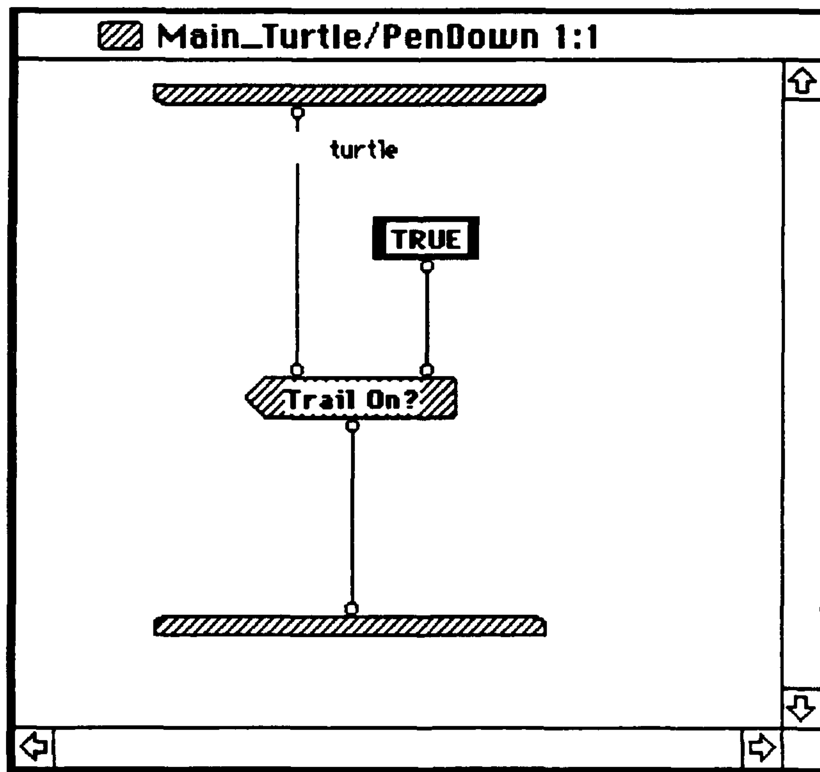

Main_Turtle/Turn Left 2:3
turtle
heading
degrees to turn
counterclockwise
heading
0
-
<
360
+
heading
new turtle heading

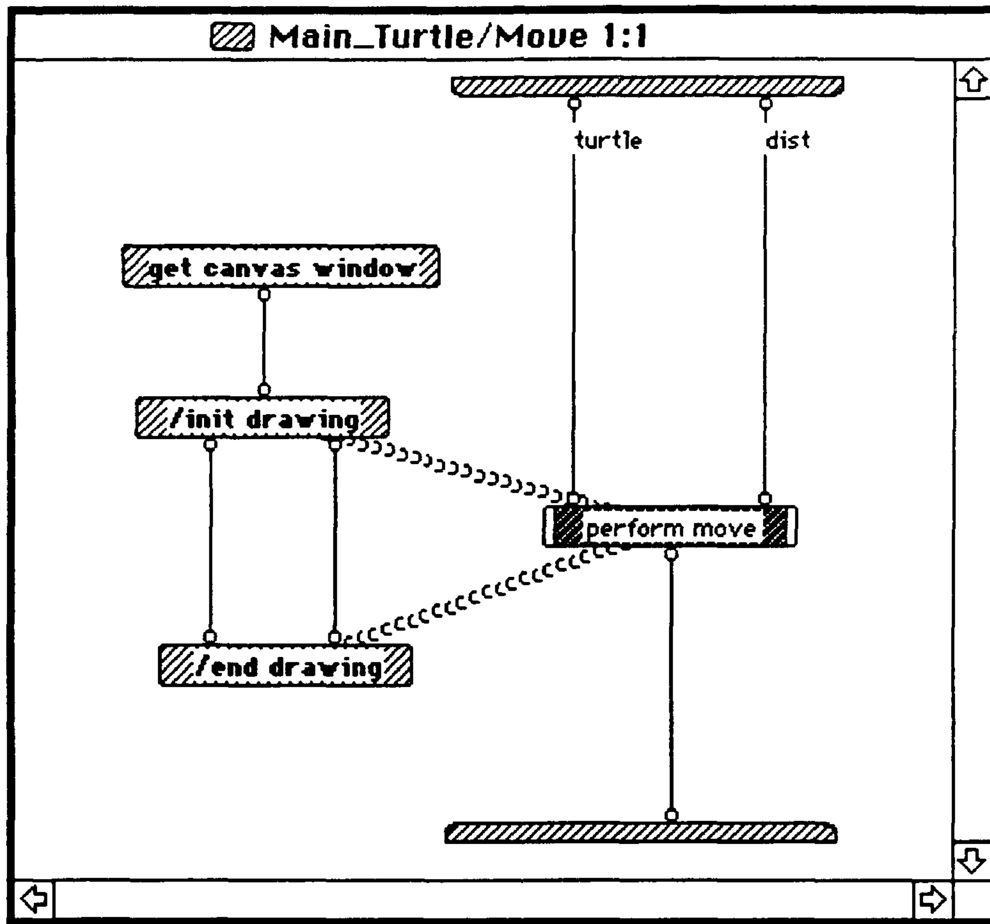

Main_Turtle/Turn Left 3:3
turtle
heading
degrees to turn
counterclockwise
heading
-
heading
new turtle heading

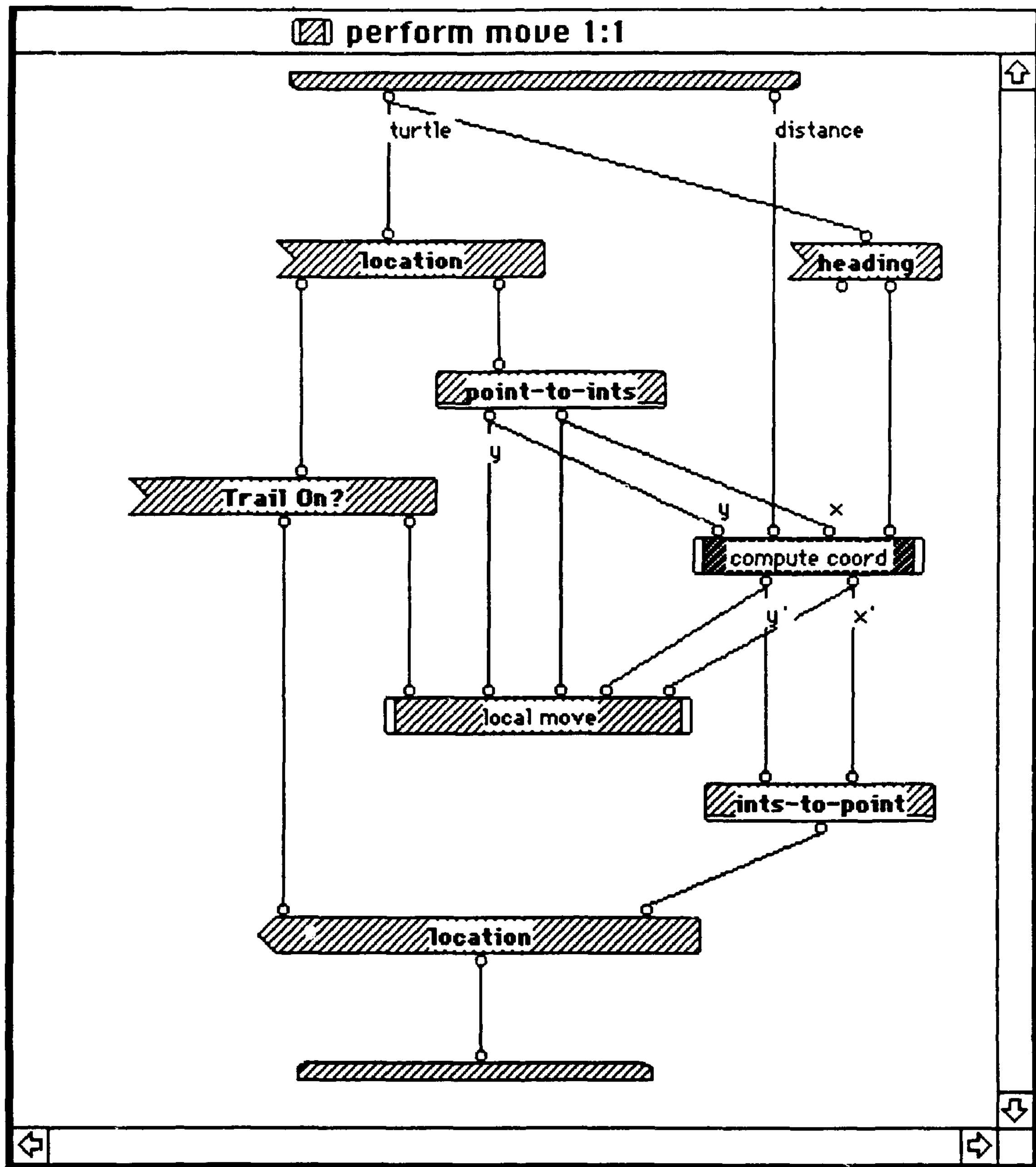

Main_Turtle/Turn Right 1:2
turtle
heading
degrees to turn
right
add degrees
from turtle heading
heading
360
360
new heading


Main_Turtle/Turn Right 2:2
turtle
heading
degrees to turn
right
heading
heading
new heading

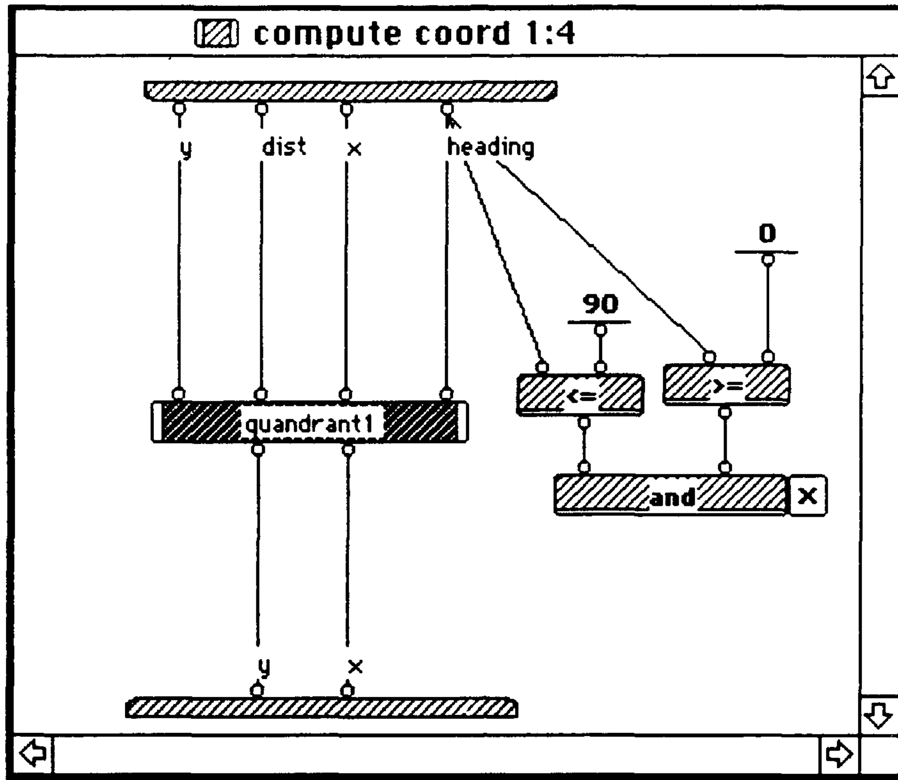

Main_Turtle/PenUp 1:1
turtle
FALSE
Trail On?

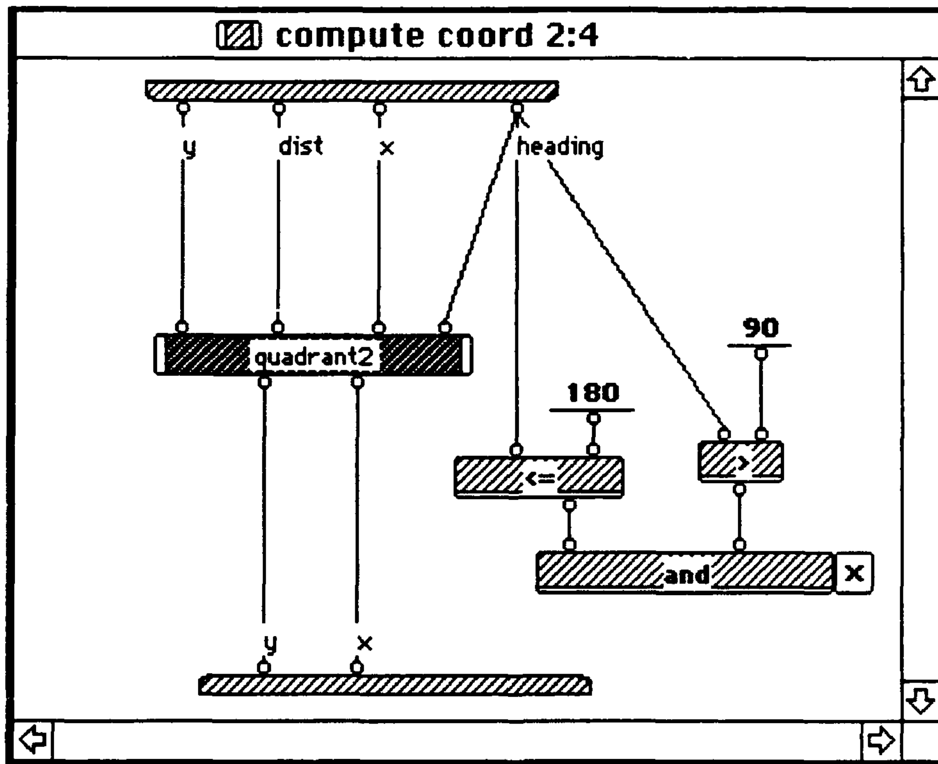

Main_Turtle/PenDown 1:1
turtle
TRUE
Trail On?

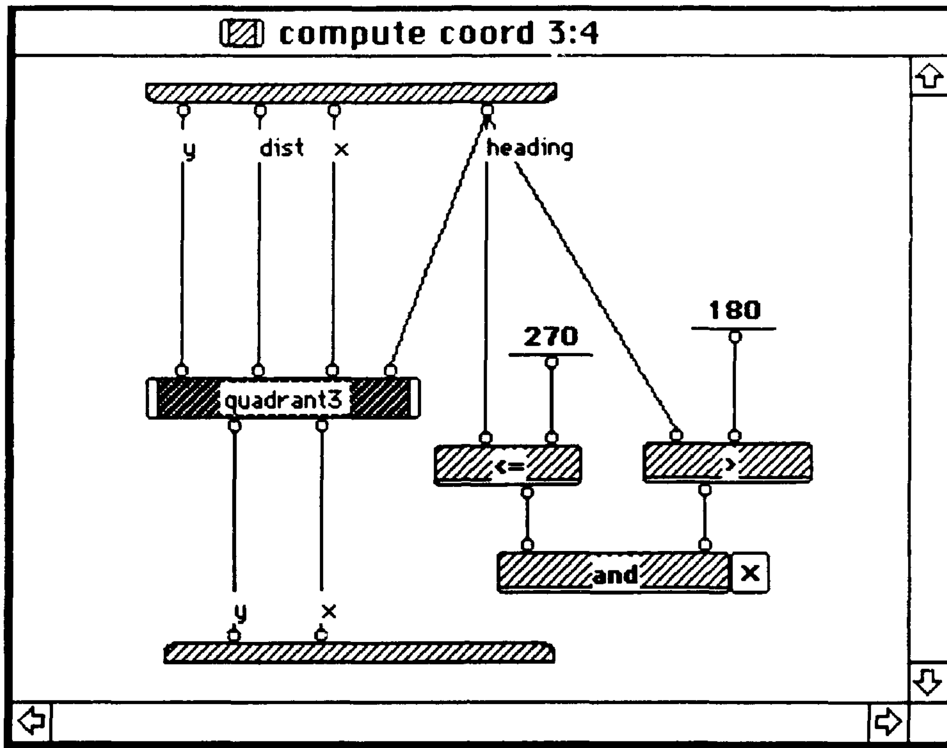

Main_Turtle/Move 1:1
turtle
dist
get canvas window
/init drawing
perform move
/end drawing

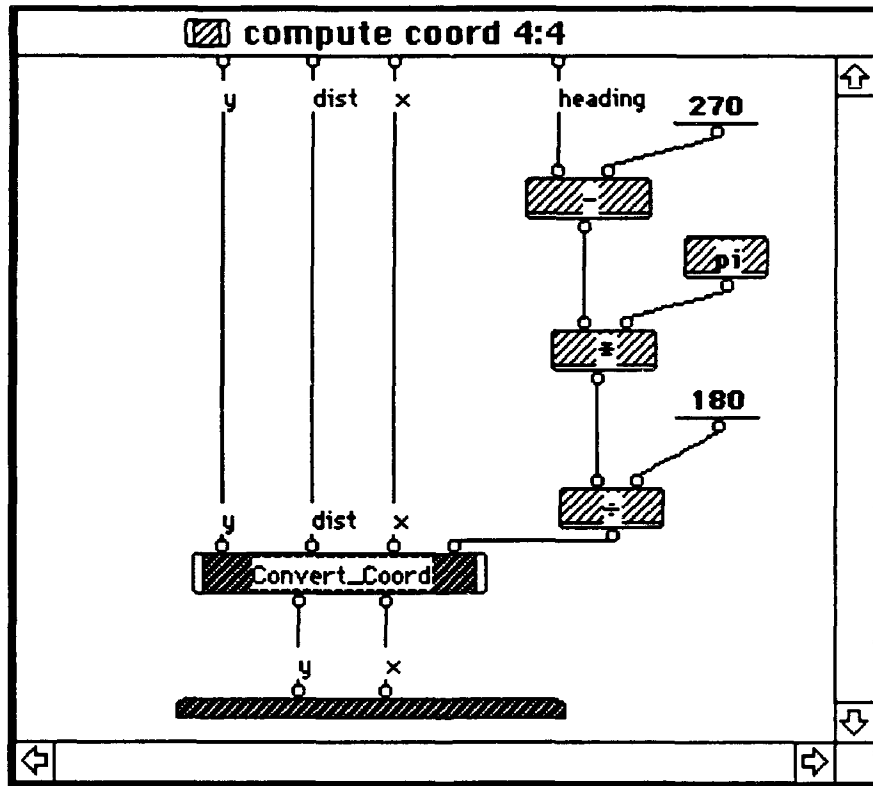


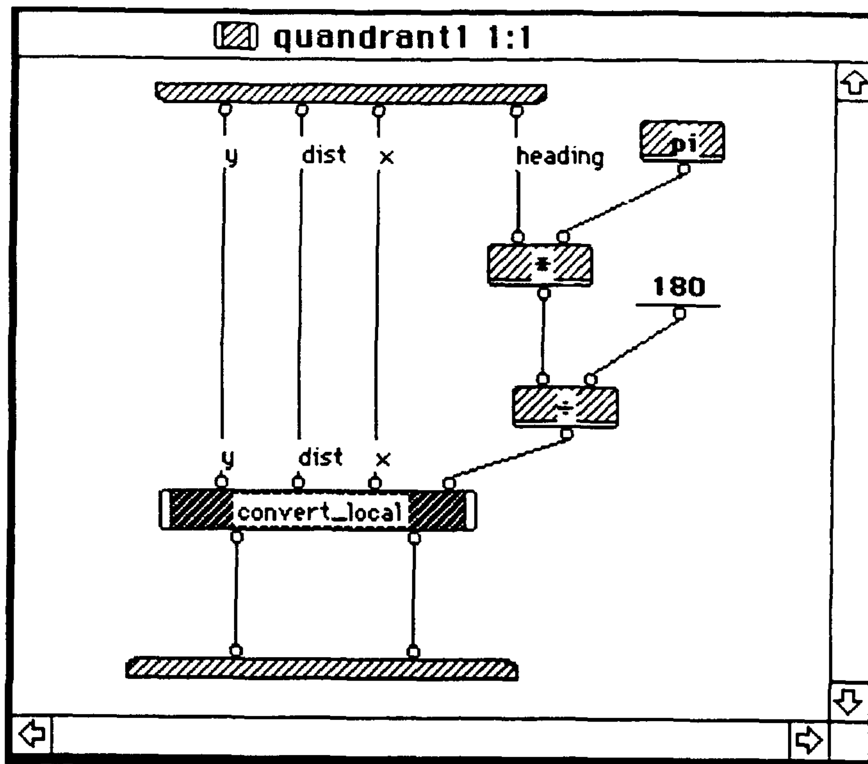
perform move 1:1
turtle
distance
location
heading
point-to-ints
y
Trail On?
y
x
compute coord
y'
x'
local move
ints-to-point
location

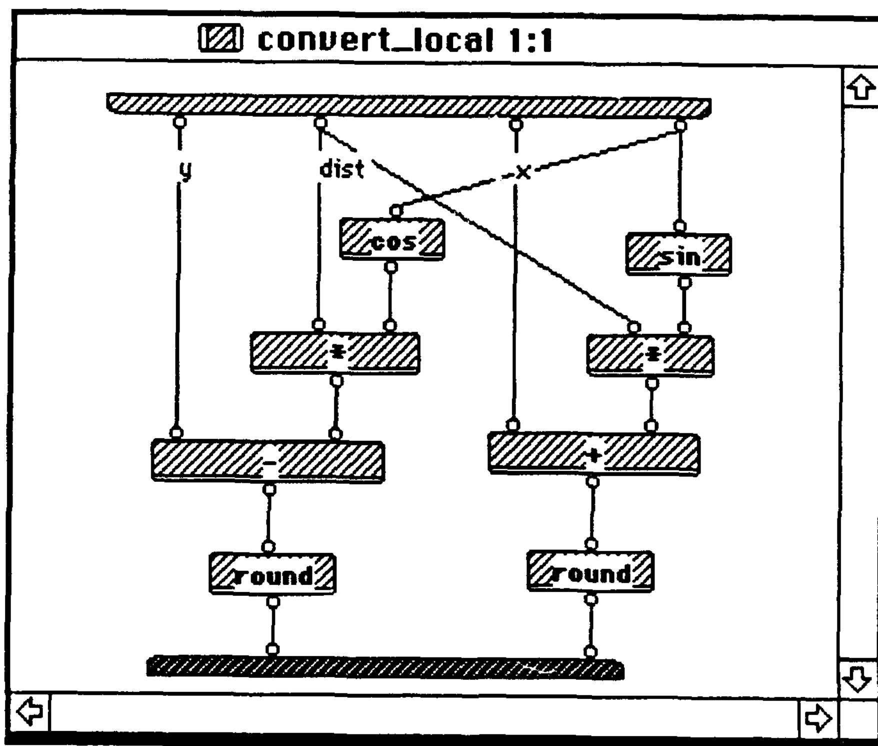

compute coord 1:4
y
dist
x
heading
0
90
quandrant1
<=
>=
and
y
x

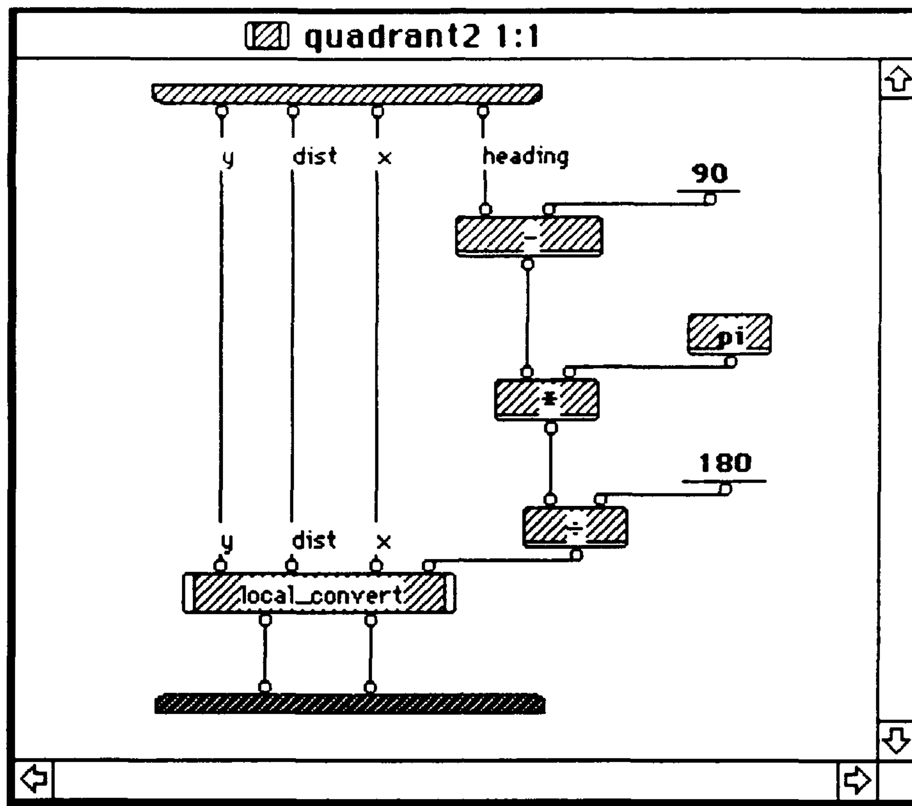

compute coord 2:4
y
dist
x
heading
90
quadrant2
180
<=
>
and
y
x

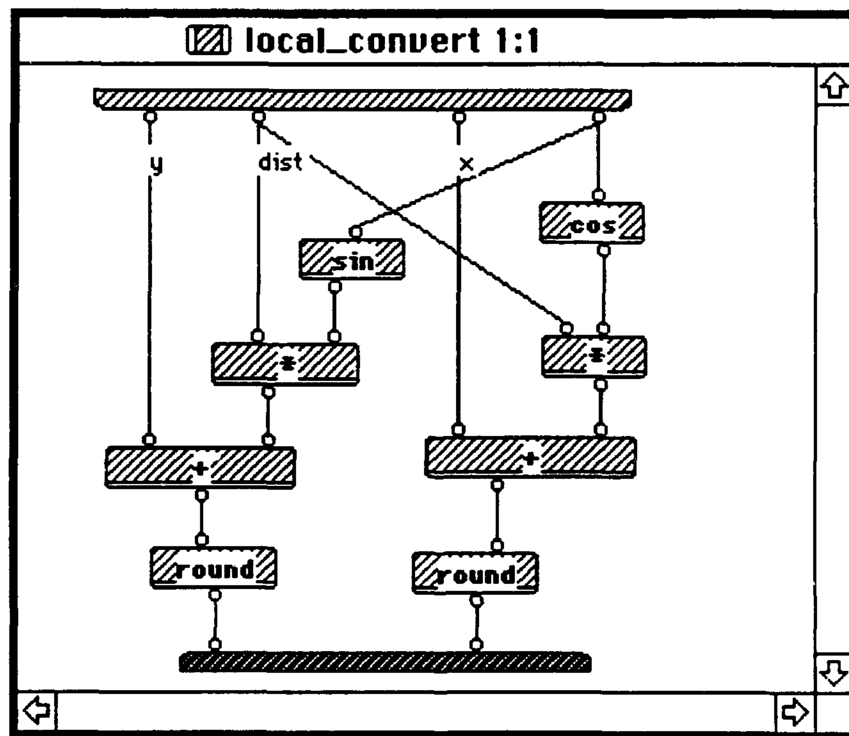

compute coord 3:4
y
dist
x
heading
270
180
quadrant3
<=
>
and
y
x

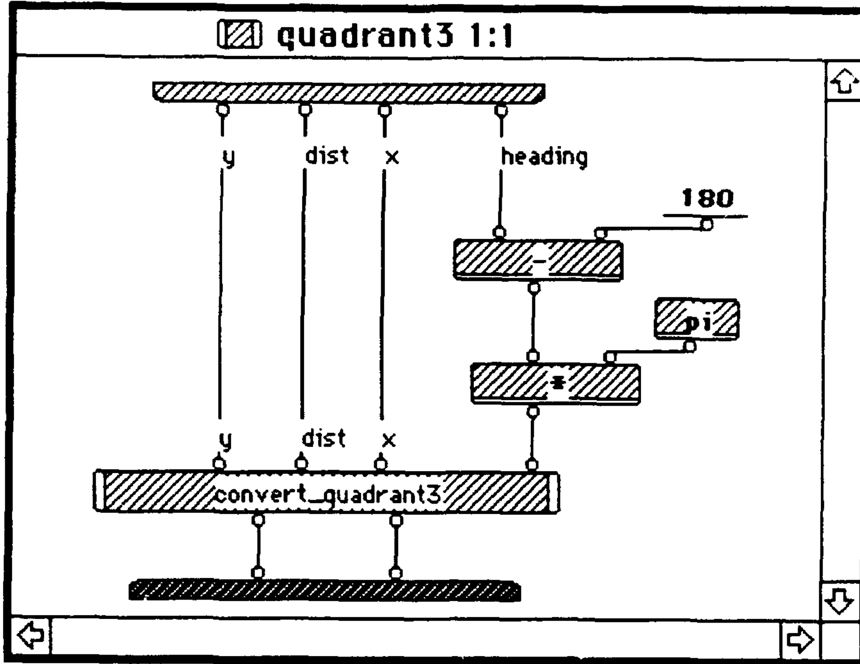

compute coord 4:4
y
dist
x
heading
270
-
pi
180
y
dist
x
Convert_Coord
y
x

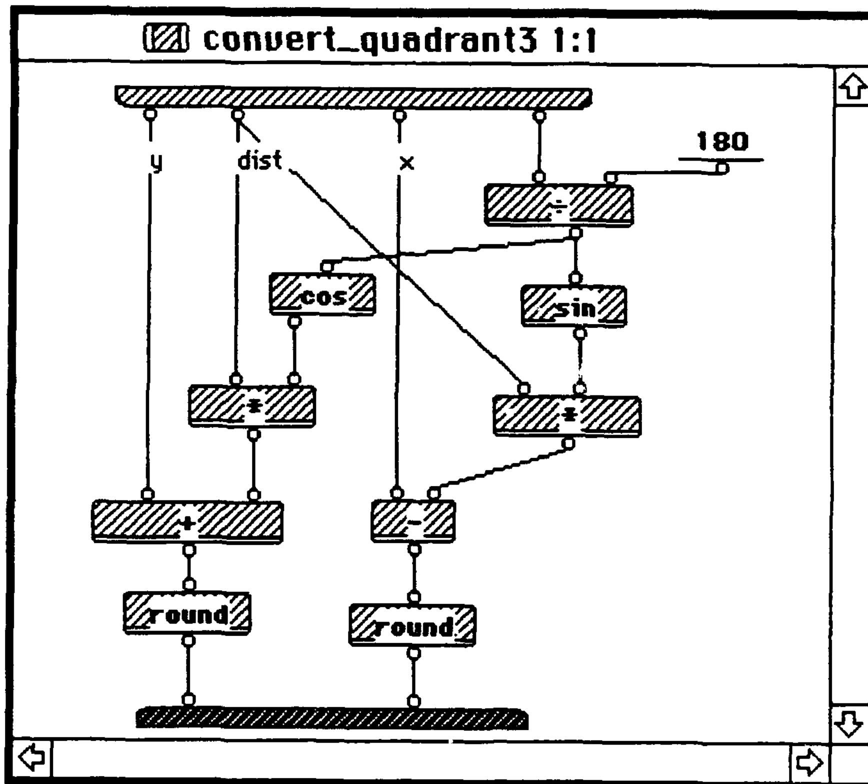

quandrant1 1:1
y
dist
x
heading
pi
*
180
÷
y
dist
x
convert_local

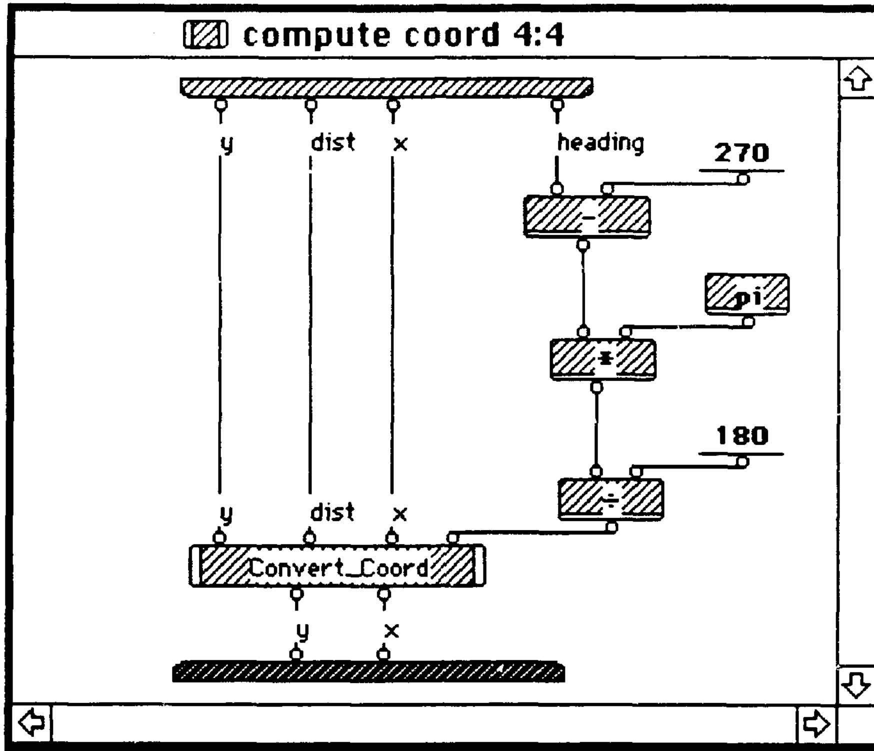

convert_local 1:1
y
dist
x
cos
sin
*
*
-
+
round
round

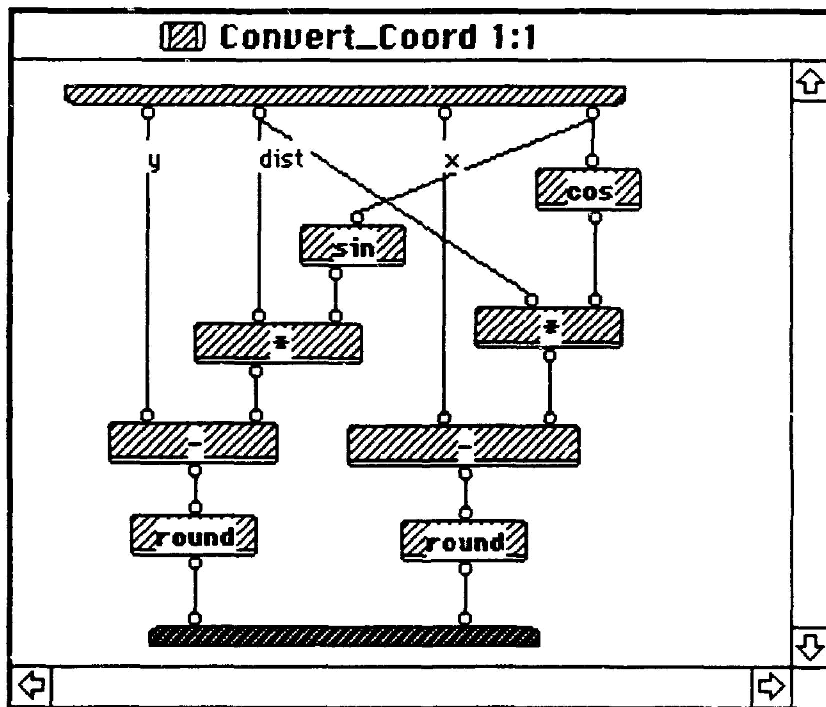

quadrant2 1:1
y
dist
x
heading
90
pi
180
y
dist
x
local_convert

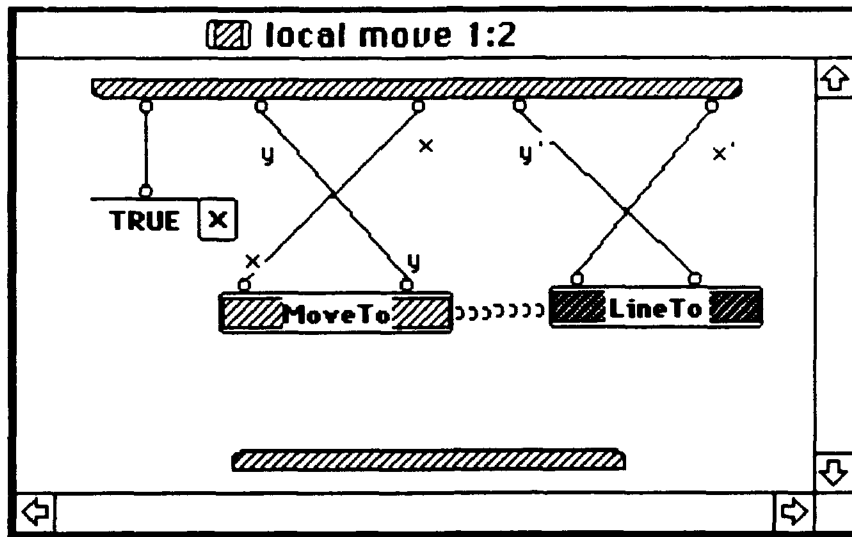

local_convert 1:1
y
dist
x
cos
sin
round
round

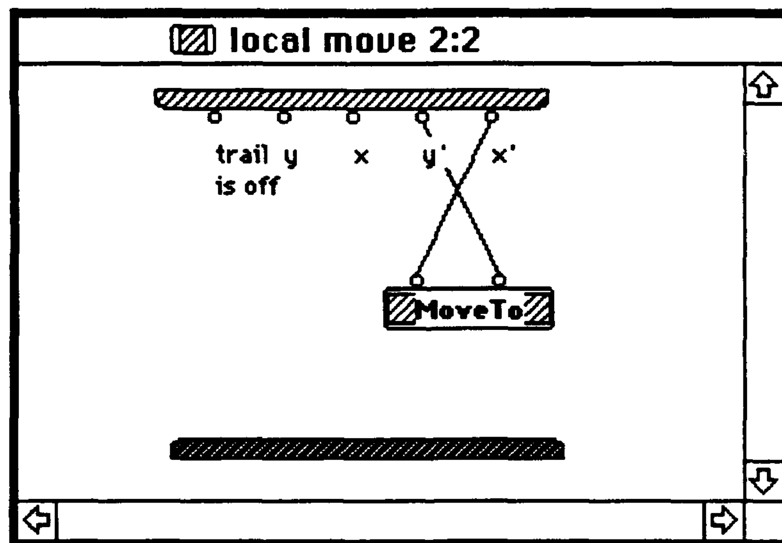

quadrant3 1:1
y
dist
x
heading
180
-
pi
*
y
dist
x
convert_quadrant3

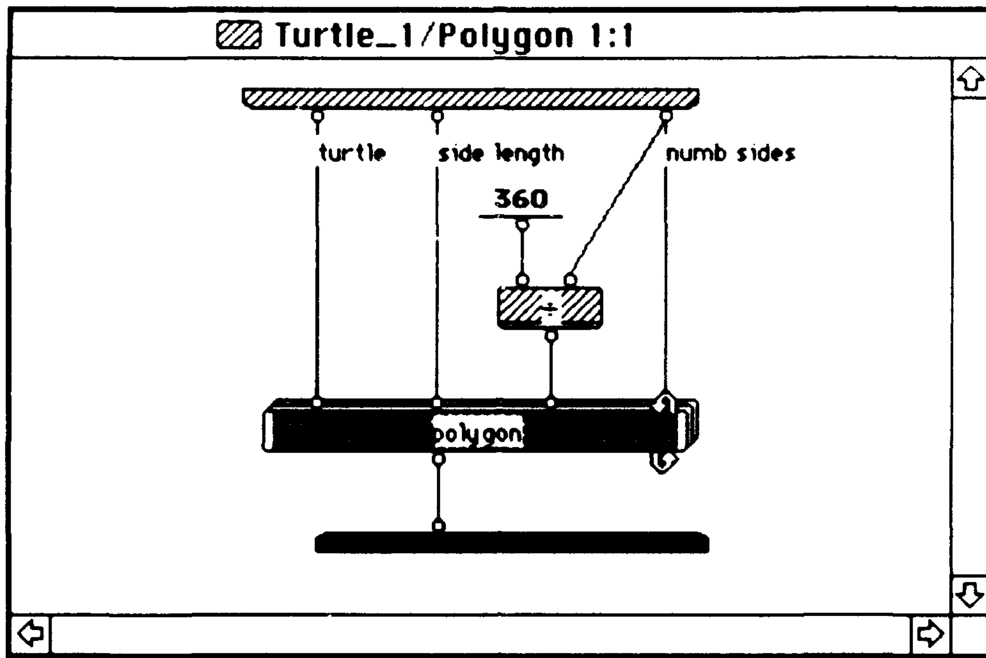

convert_quadrant3 1:1
y
dist
x
180
÷
cos
sin
*
*
+
-
round
round

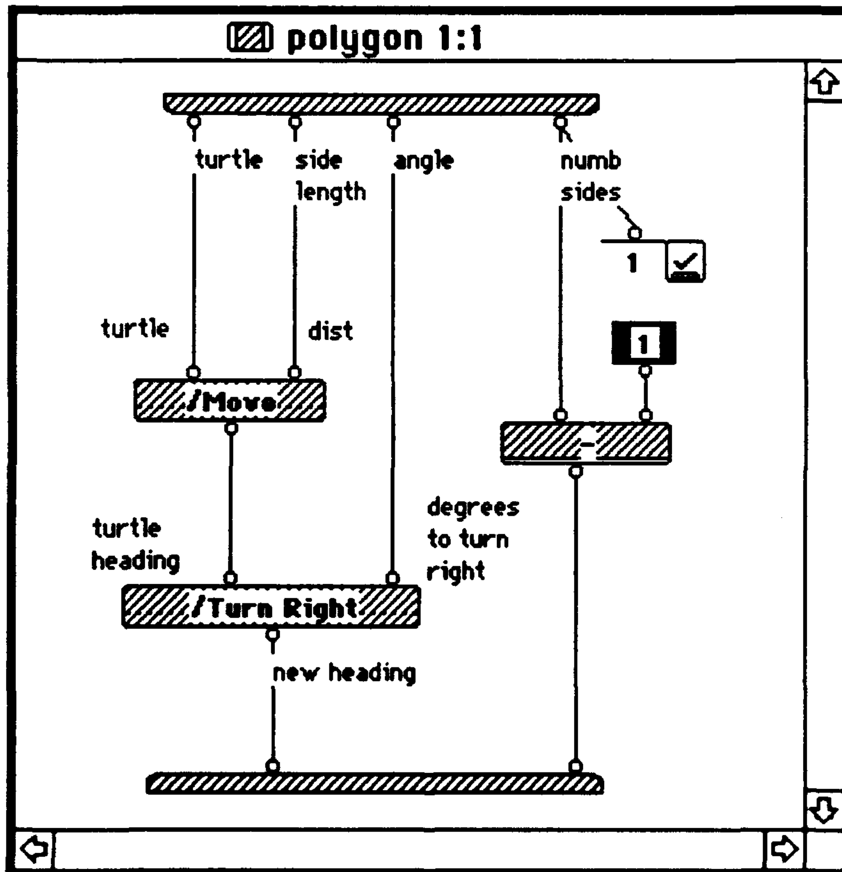

compute coord 4:4
y
dist
x
heading
270
pi
180
Convert_Coord

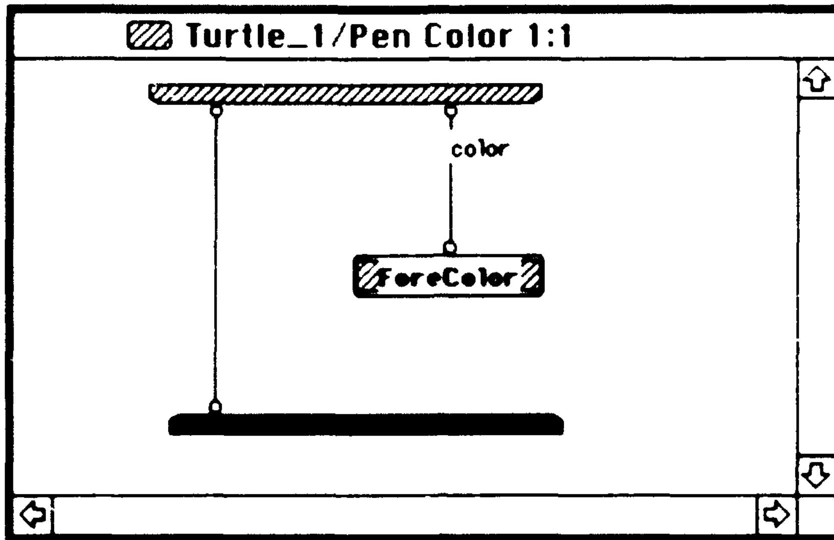

Convert_Coord 1:1
y
dist
x
cos
sin
round
round

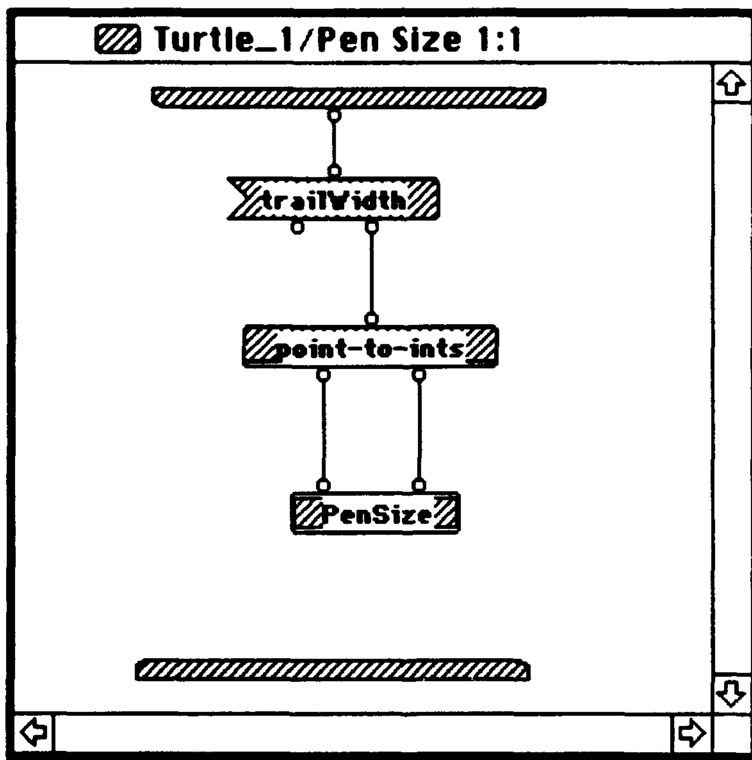

local move 1:2
y
x
y'
x'
TRUE
x
y
MoveTo
LineTo

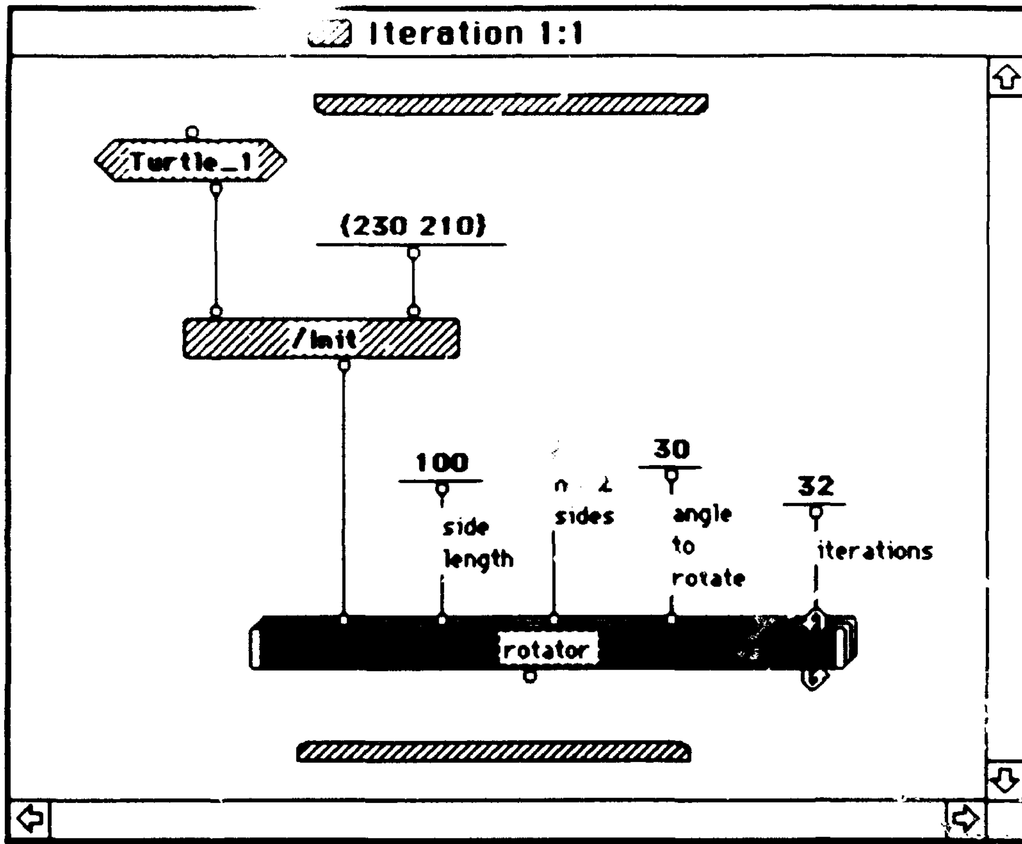

local move 2:2
trail
is off
y
x
y'
x'
MoveTo

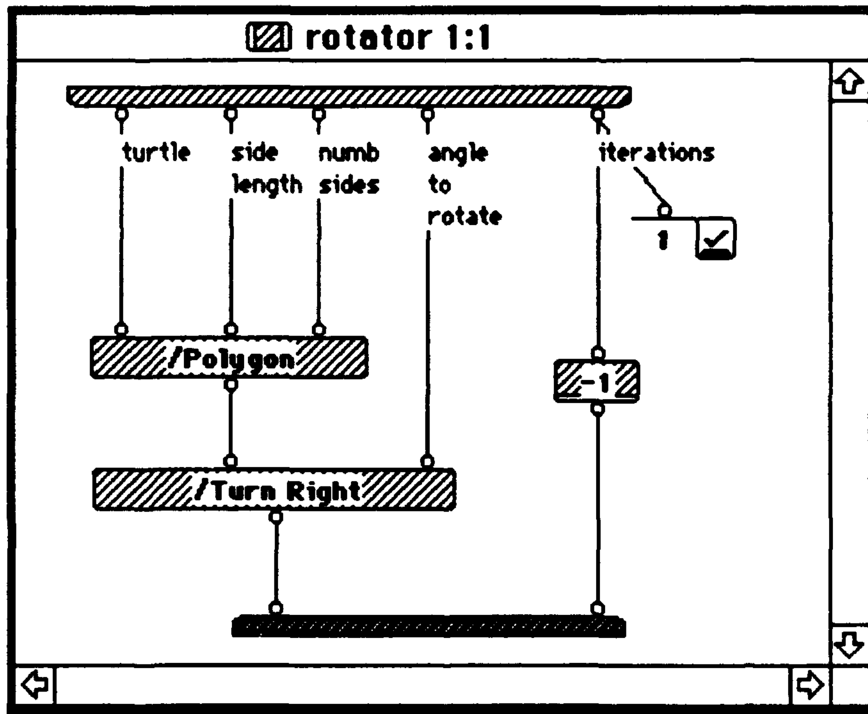

Turtle_1/Polygon 1:1
turtle
side length
numb sides
360
polygon

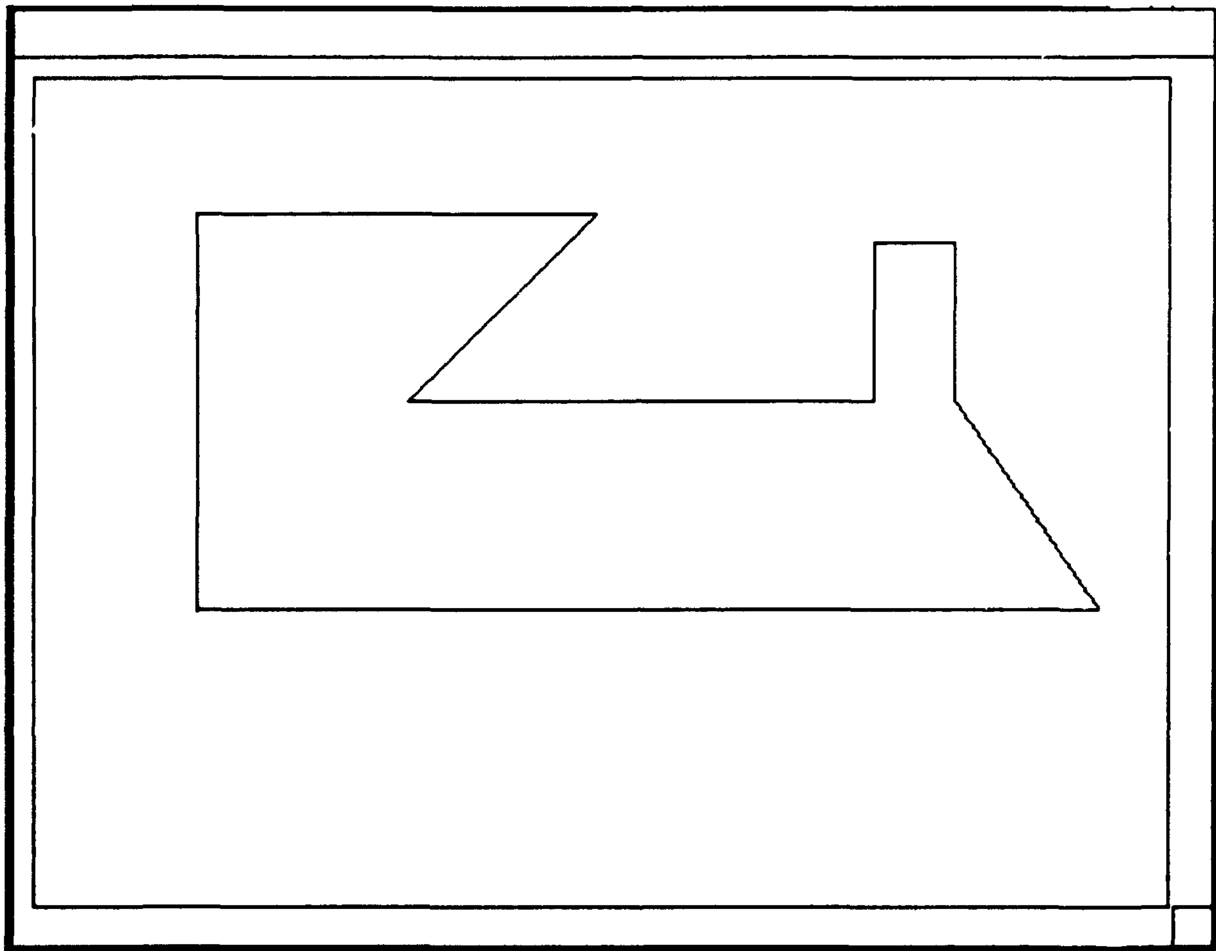

polygon 1:1
turtle
side
length
angle
numb
sides
1
turtle
dist
1
/Move
turtle
heading
degrees
to turn
right
/Turn Right
new heading

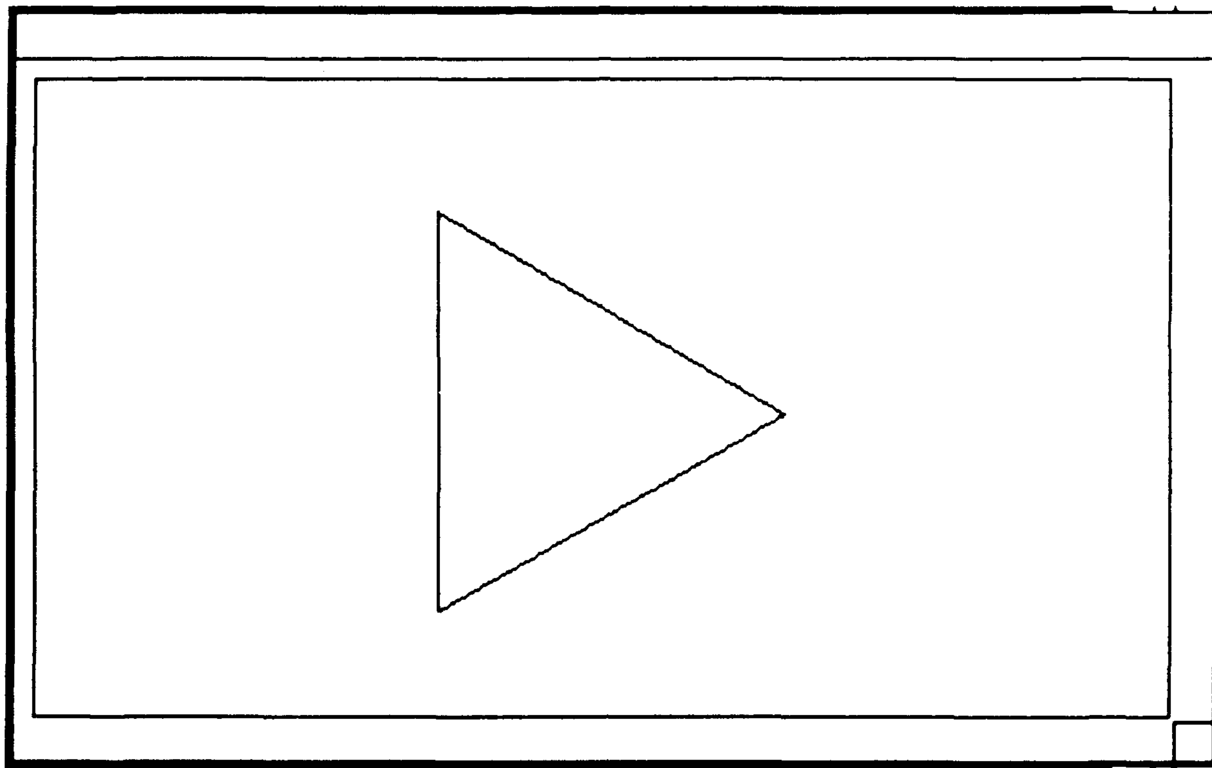

Turtle_1/Pen Color 1:1
color
ForeColor

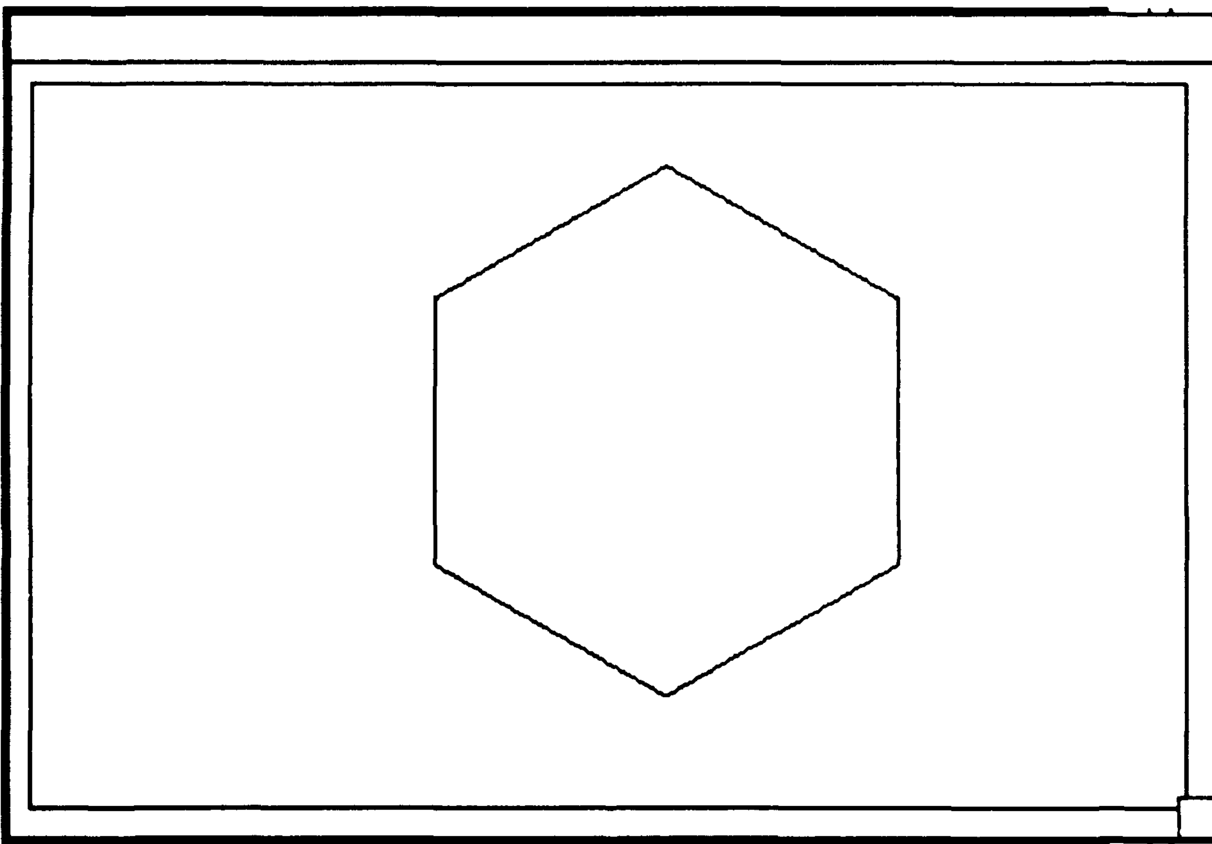

Turtle_1/Pen Size 1:1
trailWidth
point-to-ints
PenSize

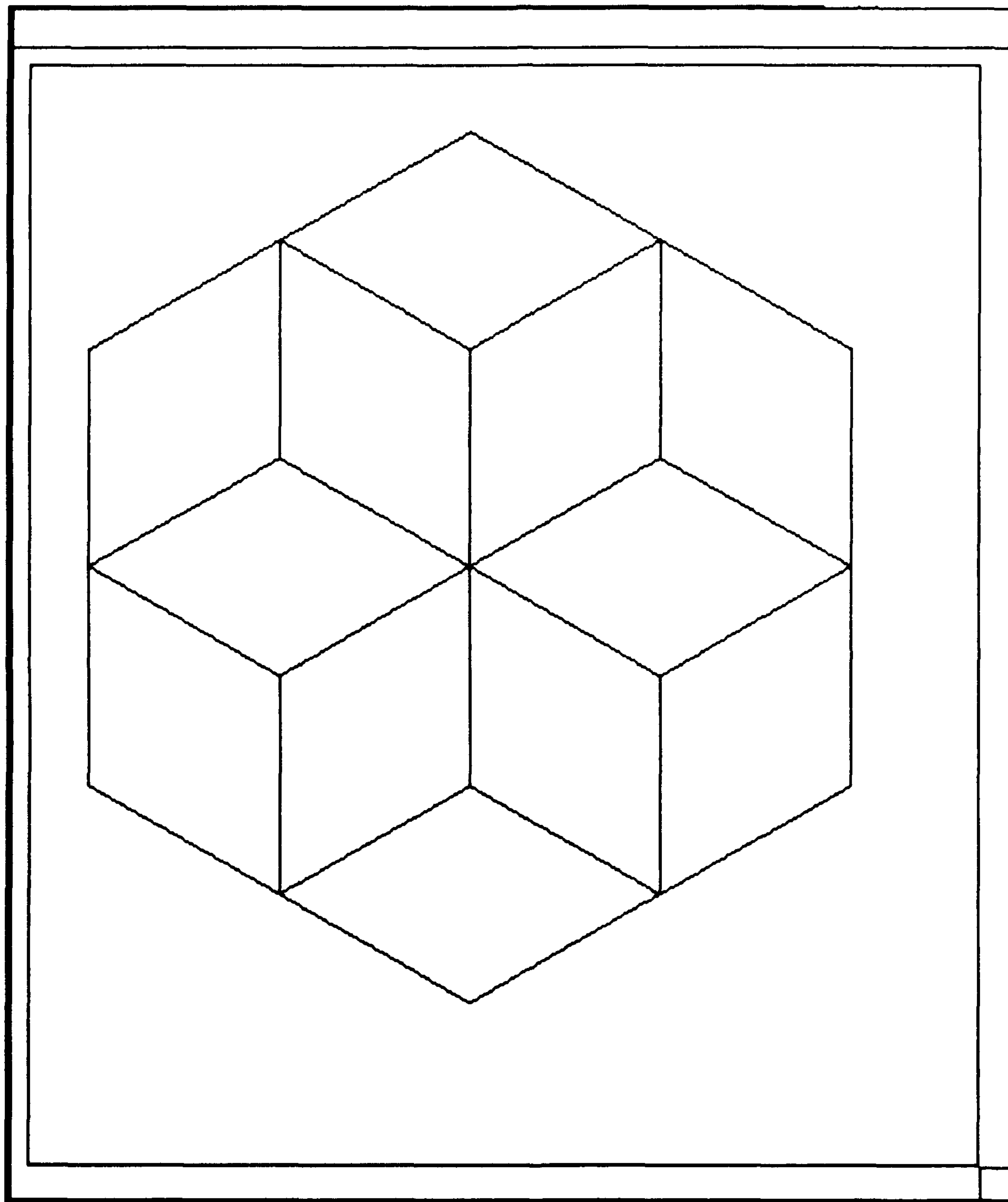

Iteration 1:1
Turtle_1
(230 210)
/Init
100
side
length
sides
30
angle
to
rotate
32
iterations
rotator

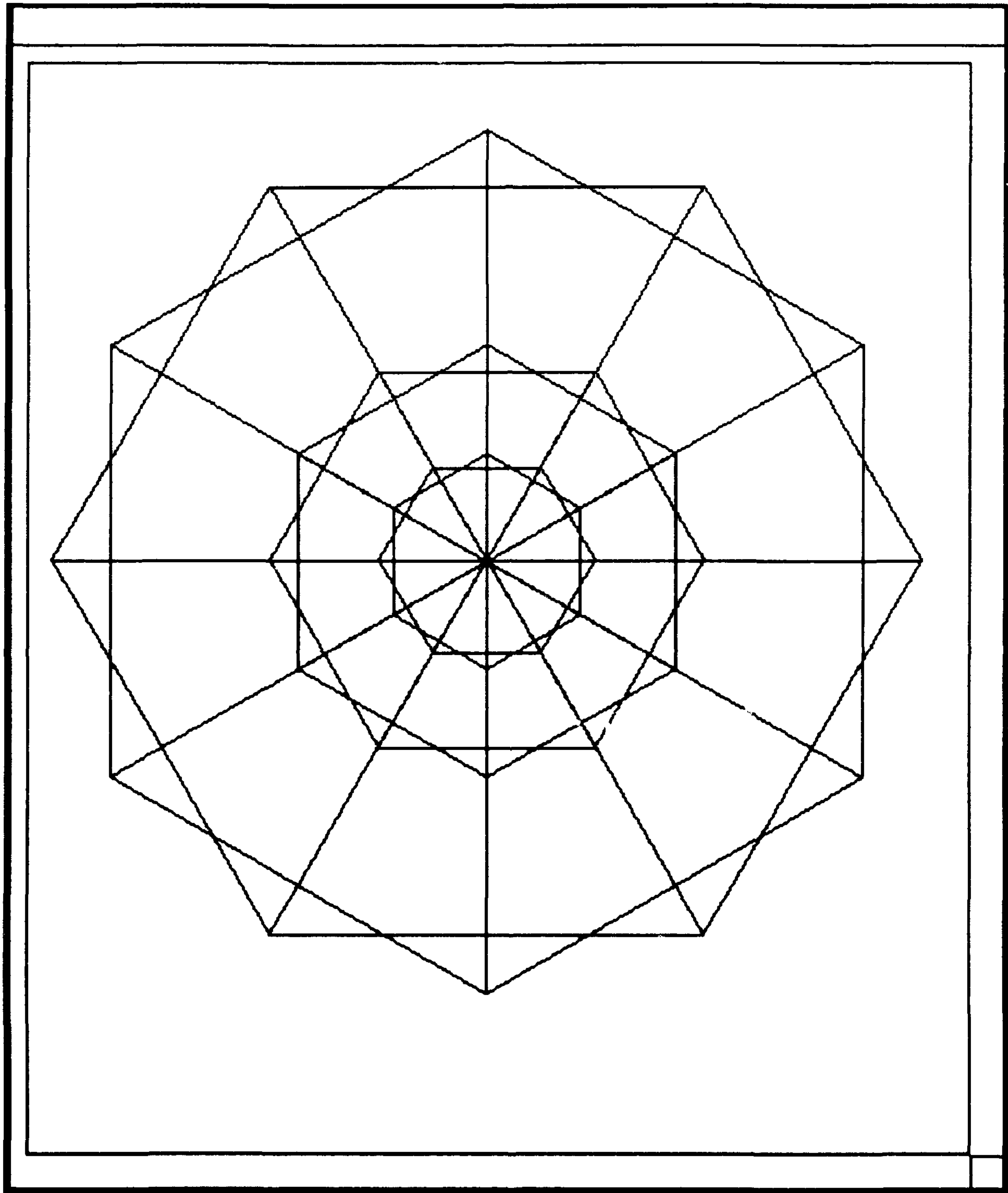

rotator 1:1
turtle
side
length
numb
sides
angle
to
rotate
iterations
1
/Polygon
-1
/Turn Right











LIST OF REFERENCES

[Abel 92] Abelson, H., and Abelson, A., *An Introduction Through Object Logo*. Paradigm Software Incorporated, 1992.

[Clay 88] Clayson, J., *Visual Modeling with LOGO*, The MIT Press Cambridge, MA 1988.

[diSe 85] diSessa, A. A., *Principles for the Design of an Integrated Computational Environment for Education*, Laboratory for Computer Science, MIT Cambridge, 1985.

[diSe 86] diSessa, A. A., *From Logo to Boxer, a New Computational Environment*, Australian Educational Computing, 1986.

[GF 87] Goldenberg, E. P., and Feurzeig, W., *Exploring Language with LOGO*, The MIT Press Cambridge, 1987.

[Pape 93] Papert, S., *Mindstorms; children, computers, and powerful ideas*, BasicBooks, A Division of HarperCollins Publishers, Inc, 1993.

[Shu 88] Shu, N. C., *Visual Programming*, Van Norstrand Reinhold Company Inc., 1988

[Terr 90] Terrapin Software, Inc. *Why use Logo? An overview of Logo in Education*, 1990.

[TGS 89] The Gunakara Sun Systems, Prograph Tutorial, 1989.

[TGS 89] The Gunakara Sun Systems, Prograph Reference, 1989.

[TGS 91] The Gunakara Sun Systems, Prograph 2.5 Updates, 1991.

BIBLIOGRAPHY

Abelson, H., and Abelson, A., *An Introduction Through Object Logo*, Paradigm Software Incorporated, 1992.

Chang, S., *Principles of Visual Programming Systems*, Prentice-Hall, Inc., 1990.

Chang, S., Ichikawa, T., and Ligomenides, P. A., *Visual Languages*, Plenum Press, 1986.

Clayson, J., *Visual Modeling with LOGO*, The MIT Press Cambridge, MA 1988.

Cox, P. T., Giles, F. R., and Pietrzykowski, T., *Prograph: A Step Towards Liberating Programming from Textual Conditioning*, Proceedings of the IEEE Workshop on Visual Languages, 1989.

diSessa, A. A., *Principles for the Design of an Integrated Computational Environment for Education*, Laboratory for Computer Science, MIT Cambridge, 1985.

diSessa, A. A., *From Logo to Boxer, a New Computational Environment*, Australian Educational Computing, 1986.

Goldenberg, E. P., and Feurzeig, W., *Exploring Language with LOGO*, The MIT Press Cambridge, 1987.

Papert, S., *Mindstorms; children, computers, and powerful ideas*, BasicBooks, A Division of HarperCollins Publishers, Inc, 1993.

Shu, N. C., *Visual Programming*, Van Norstrand Reinhold Company Inc., 1988.

Terrapin Software, Inc. *Why use Logo? An overview of Logo in Education*, 1990.

The Gunakara Sun Systems, Prograph Tutorial, 1989.

The Gunakara Sun Systems, Prograph Reference, 1989.

The Gunakara Sun Systems, Prograph 2.5 Updates, 1991.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, Virginia 22304-6145

2. Library, Code 52 2
   Naval Postgraduate School
   Monterey, California 93943-5002

3. Chairman, Code CS 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943-5002

4. C. Thomas Wu, Code CS/Wu 2
   Computer Science Department
   Naval Postgaduate School
   Monterey, California 93943-5002

5. Roger Stemp, Code CS/St 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943-5002

6. LCDR Emily Black, USN 1
   Communications School
   Cushing Rd.
   Naval Education and Training Center
   Newport, Rhode Island 02841

7. CAPT Thierno Fall, Senegal Army 2
   Ambassade du Senegal
   Mission Militaire
   1825 Connecticut Ave, N.W. 216
   Washington, D.C. 20009