Appendix B
Turtle Programs in Conventional Computer Languages

This appendix is an aid to adapting the programs in this book for work with computer systems and languages that are widely available. It should also be useful as a guide to selecting a computer system suitable for work with this book. Personal computers are changing rapidly for the better as regards advanced languages and high-resolution graphics, and readers should take this into account.

General Considerations

Display

In addition to a computer equipped with a suitable language, work with turtle geometry requires a graphic-output device capable of producing line drawings. One possibility is a computer-controlled plotter using paper and ink, although a display screen is superior for most purposes. Fortunately, inexpensive graphic displays based on television technology—some with built-in monitors and some designed for use with ordinary televisions—are rapidly becoming standard for personal computers.

Programing Language

Given enough memory, essentially any programing language can do the same things as any other. Thus, selecting a computer language is to some extent a question of style and personal preference. But a language that forces you to worry about details that have nothing to do with the process you are trying to implement, or does not mesh with good methods of problem solving and algorithm specification, can make using a computer tedious and burdensome. On the other hand, a computer language similar to the Turtle Procedure Notation used in this book can simplify the programing and allow you to concentrate on the mathematics, on inventing and exploring. It can even serve as a language to help you think about issues and solve problems.

We list here some issues of computer-language design and say what we think are desirable ways to deal with them from the point of view of turtle geometry.

Procedures, Inputs, and Outputs (Extensibility)

Dividing a problem into meaningful and relatively independent subparts is a vital step to solving any complex problem. A programing language

should recognize this by allowing you to define your own procedures at will and use them whenever you want with no fuss. For purposes of exploration it is especially useful to be able to create a complete environment of procedures, such as FORWARD, RIGHT, and LEFT, and to not only use them as steps in other procedures, but to also select and execute them at will in controlling an ongoing process (such as a picture coming into being).

Allowing procedures to have inputs vastly increases the flexibility of such an environment of procedures. The names of procedure inputs should be local to the procedure, so that the user need not worry about the internals of a procedure when using it (see the section on local variables in appendix A). The ability of procedures to output is also important, because it enables the programer to extend the language by adding new functions.

Repetition and Recursion (Flow of Control)

Repetition is such a useful computational structure that every language has constructs for carrying it out. The question is whether these constructs are natural and easy to use. For example, a "REPEAT...UNTIL..." construct can always be implemented in terms of GOTO's, but this often leads to careless errors. Recursion is nearly as useful, but unfortunately it is not implemented in primitive languages such as FORTRAN and BASIC since it requires the computer to keep track of different instances of the same procedure (in a recursive hierarchy).

Data Types

For the purposes of this book we focus on one important feature: the ability to create data types with several parts (*compound* data types) and to be able to deal flexibly with the whole *or* the parts. The case in point is the way we use lists to represent vectors. In Turtle Procedure Notation these can be passed as units into and out of a procedure (with inputs and outputs). But it is just as important to be able to easily assemble and dissassemble such units for internal computation. Unfortunately, mechanisms like Turtle Procedure Notation's structure-directed operations are not at all common, and various special tricks must usually be used, depending on the computer language. The only other data-type issue we will be concerned with is distinction between integers and real numbers. Real numbers are crucial for doing geometric calculations such as vector rotations, intersections, and so on. On the other hand, the display routines of many computer systems can handle only integer inputs, in which case the result of the geometric calculations must be converted to integers before being shown on the display.

Consideration of Specific Languages

BASIC

BASIC is by far the most widely available language on inexpensive microcomputers at the present time (1981). Unfortunately, it is deficient by several of the criteria listed above. BASIC also comes in many incompatible versions. We have selected one of these—APPLESOFT II BASIC (TM)—to use in examples, because it comes on a popular, inexpensive machine (the Apple II with high-resolution graphics) that has adequate graphics capability for turtle geometry.

Procedures Inputs and Outputs

BASIC lets you define only one program at a time. The RUN command, which executes the program currently in memory, is essentially the only way to make any user-definable thing happen. RUN may, however, take an optional input that specifies the number of the program line at which to start. In that case you can get the effect of selecting different programs in response to direct user commands by starting the program at one of a number of sections of code, each of which ends with END.

Most BASICs have another mechanism, which gives the effect of having separate procedures *within* a program: You can treat portions of your program as subprocedures, or "subroutines" as they are called, by using the GOSUB command (although you cannot name these procedures or insulate their effects with local variables, inputs, and outputs). GOSUB takes an input specifying the line number at which the subroutine starts. (The subroutine ends with a RETURN.) It is sometimes possible to use a variable to specify the line number so that programs will be a bit easier to read. If this is the case, then you can invoke a "FORWARD" subroutine by saying GOSUB FORWARD, where FORWARD is a variable that has been set to 10000 or whatever line number begins the "FORWARD" subroutine. In this appendix we will use this feature for its readability, although you may have to actually use literal numbers with GOSUB.

Passing "inputs" to subroutines must be handled as global variables, so you are always liable to find some subroutine destroying some other subroutine's inputs. You can minimize this problem by establishing an appropriate convention for choosing variable names, such as letting input variables consist of two characters: the first letter of the subroutine name and a number used to distinguish among possible multiple inputs. Thus you will find yourself regularly writing code such as

```
F1 = 10: GOSUB FORWARD
```

or

```
P1 = 90 : P2 = 100 : GOSUB POLY
```

Outputs must be handled in a similar way, using, say, RF1, RF2,... for the returned values of a FUNCTION subroutine.

Below is an APPLESOFT II BASIC implementation of turtle commands that works on Apple's high-resolution graphics system. It differs from the full vector version given in chapter 3 principally by keeping heading as an angle rather than as a vector. "POSITION VECTOR" is in PX, PY; "HEADING" is H; "PEN STATE" is in PS (pen is down when PS=1).

```
10 FORWARD = 10000 : REM the following index subroutines
11 LEFT = 10100
12 RIGHT = 10200
13 CLEARSCREEN = 10300
14 DRAWLINE = 10400

10000 REM *** FORWARD distance F1 subroutine ***
10010 HX = COS(H*3.14159/180) : REM heading vector components
10020 HY = SIN(H*3.14159/180)
10030 NX = PX + HX * F1 : REM add [HX HY]*F1 to position vector
10040 NY = PY + HY * F1 : REM save "new-position" vector [NX NY]
10050 IF PS <> 1 GOTO 10070 : REM if the pen is up, don't draw
10060 D1 = PX : D2 = PY : D3 = NX : D4 = NY : GOSUB DRAWLINE
10070 PX = NX : PY = NY : REM update position vector
10090 RETURN

10100 REM *** LEFT angle L1 (in degrees) subroutine ***
10110 H = H + L1
10120 RETURN

10200 REM *** RIGHT angle R1 (in degrees) subroutine ***
10210 H = H - R1
10220 RETURN

10300 REM *** CLEARSCREEN subroutine ***
10310 HGR : HCOLOR = 7 : REM clear the screen
10320 PX = 0 : PY = 0 : REM initialize position vector
10330 H = 0 : REM initialize heading
10340 PS = 1 : REM start with pen down
10350 RETURN
```

```
10400 REM *** DRAWLINE from [D1 D2] to [D3 D4] subroutine ***
10401 REM The inputs to HPLOT center [0 0] and make increasing
10402 REM y and x correspond to "up" and "to the right."
10410 HPLOT 140 + D1 , 80 - D2 TO 140 + D3 , 80 - D4
10420 RETURN
```

Here is a POLY program that uses these routines:

```
100 REM *** POLY program ***
110 INPUT "ANGLE?"; A
120 INPUT "DISTANCE?"; D
130 GOSUB CLEARSCREEN
140 F1 = D : GOSUB FORWARD
150 L1 = A : GOSUB LEFT
160 GOTO 140
```

Notice the practice of having a program such as this—which is meant to be RUN, that is, called from top level rather than called as a subprocedure—explicitly *ask* for its inputs by using the INPUT command.

Repetition and Recursion

Repetition can usually be handled reasonably in BASIC with GOTO statements and stop rules, as in the following sequence to compute the square root of a number A (compare the example in appendix A):

```
100 G = 1.0 : REM initialize guess
110 G = (G + A/G) / 2 : REM update guess
120 E = ABS(A - G*G) : REM error in guess squared
130 IF E > .001 GOTO 110 : REM stoprule is error no more than .001
```

Alternatively, one can use BASIC's do-loop mechanism as in the following sequence to accumulate in T the sum of the first 50 integers:

```
100 T = 0 : REM initialize accumulator
110 FOR N = 1 TO 50 DO
120 T = T + N
130 NEXT N : REM increment N and GOTO the line after FOR
```

Recursion in BASIC is so inconvenient in the general case as to be almost hopeless. (One would have to implement at the user level nearly the whole mechanism for keeping track of local variables that is built into more sophisticated languages.) Fortunately, many of the uses to which we put recursion can be handled by iterative processes. For example, any "tail recursion," where the recursive call is just the last line

of the procedure, is at most just a matter of using a GOTO after changing the value of a variable whose old value is never used again. POLYSPI and INSPI could well be written as repetitive loops with an inserted command to change SIDE or ANGLE variables each time through the loop.

Even in cases where true recursion seems to be required, as in the recursive designs of chapter 2, there are a few tricks that often work. If the number of levels is small you may be able to get away with using the subroutine mechanism. The chief problem is how to keep subroutines from destroying their own (recursively called) inputs. This is manageable in cases where the process of computing the inputs to the next level down is a reversible process. That way each subroutine can maintain a sort of "state transparency" (see subsection 2.3.2) by changing the value of its "inputs" when it starts and undoing the change just before it returns. For example, here is a BASIC implementation of the recursive binary tree of 2.3.2:

```
15 BRANCH = 2000 : REM so we can say GOSUB BRANCH

200 GOSUB CLEARSCREEN
210 INPUT "SIZE?"; SIZE
220 INPUT "LEVEL?"; LEV
230 GOSUB BRANCH
240 END

2000 IF LEV = 0 GOTO 2100 : REM stoprule
2010 SIZE = SIZE/2 : LEV = LEV - 1 : REM change parameters
2020 F1 = SIZE : GOSUB FORWARD
2030 L1 = 45 : GOSUB LEFT
2040 GOSUB BRANCH
2050 R1 = 90 : GOSUB RIGHT
2060 GOSUB BRANCH
2070 L1 = 45 : GOSUB LEFT
2080 F1 = -SIZE : GOSUB FORWARD
2090 LEV = LEV + 1 : SIZE = SIZE * 2 : REM undo change
2100 RETURN
```

Note how the main program merely sets up for the subroutine, and the subroutine's second and last steps are inverses. It may be valuable to have the main program set a more appropriate starting point on the screen.

A slight variation on this (check how parity is handled in line 3030) allows the HILBERT curve of section 2.4.3 to be written as follows:

```
16 HIL = 3000 : REM index subroutine

300 GOSUB CLEARSCREEN
310 INPUT "SIZE?"; F1
320 INPUT "LEVEL?"; L
330 P = 1
340 GOSUB HIL
350 END

3000 IF L = 0 GOTO 3140 : REM stoprule
3010 L = L - 1
3020 L1 = P * 90 : GOSUB LEFT
3030 P = (-P) : GOSUB HIL : P = (-P)
3040 GOSUB FORWARD
3050 R1 = P * 90 : GOSUB RIGHT
3060 GOSUB HIL
3070 GOSUB FORWARD
3080 GOSUB HIL
3090 R1 = P * 90 : GOSUB RIGHT
3100 GOSUB FORWARD
3110 P = (-P) : GOSUB HIL : P = (-P)
3120 L1 = P * 90 : GOSUB LEFT
3130 L = L + 1
3140 RETURN
```

Data Types

BASIC's compound-data-type capability is so weak that is is almost always best to deal with compound things in terms of the separate parts. Thus, vectors will be handled as separate variables for components, VX, VY, VZ, etc. Given this, it makes less sense to subroutinize vector addition and scalar multiplication, since setting up multiple inputs and outputs is so clumsy. You will find yourself writing more and thinking more in terms of components when you program in BASIC, but try to resist at least the thinking part. Here is a subroutine to rotate a two-dimensional vector (R1 and R2 are the input coordinates and R3 the angle of rotation; RR1 and RR2 are the rotated coordinates; we suppress line numbers). Compare this with the rotation procedure of subsection 3.2.2:

```
S = SIN(3.14159*R3/180)
C = COS(3.14159*R3/180)
RR1 = R1 * C - R2 * S
RR2 = R2 * C + R1 * S
RETURN
```

Notice we have not explicitly used PERP, but only its components.

A three-dimensional YAW subroutine (see subsection 3.4.3) operating directly on the components of heading and left vectors would look as follows:

```
S = SIN(3.14159*Y1/180)
C = COS(3.14159*Y1/180)
THX = C * HX + S * LX : REM "T" FOR TEMPORARY
THY = C * HY + S * LY
THZ = C * HZ + S * LZ
LX = C * LX - S * HX
LY = C * LY - S * HY
LZ = C * LZ - S * HZ
HX = THX : HY = THY : HZ = THZ
RETURN
```

If your display requires integer inputs, make sure to convert with an INTEGER function.

Logo

Logo is the name for a philosophy of education and for a continually evolving family of computer languages that aid its realization. The Logo computer language has been used over the past ten years at MIT and elsewhere as the basis for many innovative experiments in harnessing computer technology to enhance education, including the experiments that led to the writing of this book. Because Logo is such a sophisticated and powerful language, it was for many years not feasible to design Logo implementations for computers inexpensive enough to be used in homes and schools, and the use of Logo was restricted mainly to research centers. Recently, the tremendous decline in the cost of computers has changed this state of affairs. At least two Logo implementations for personal computers—the Apple II and the Texas Instruments 99/4—are commercially available. (Interested people should contact the Logo Project at MIT for further information.)

Except for minor variations of syntax, Logo is substantially the same as the Turtle Procedure Notation used in this book. In particular, Logo is highly interactive, organizes programs as collections of procedures, includes lists as "first-class" data objects, and has built-in turtle graphics. For comparison with Turtle Procedure Notation, here is a Hilbert curve program (see subsection 2.4.3) in Logo. (To save us from utter boredom, we have introduced a slight variation that draws the curve on a diagonal.)

```
TO HILBERT :SIDE :LEVEL :P
H (-:P)
RFR :P
H :P
RFR (-:P)
H :P
RFR :P
H (-:P)
END

TO RFR :P
RIGHT :P * 45
FORWARD :SIDE
RIGHT :P * 45
END

TO H :P
IF :LEVEL = 0 RIGHT :P * 90 STOP
HILBERT :SIDE :LEVEL - 1 :P
END
```

Notice that the program is basically a collection of recursive calls to itself (H) interspersed with a "corner turn" primitive motion (RFR). The major difference between the above code and the Turtle Procedure Notation used in this book is that the variable names are preceded by colons.

Pascal

Pascal has recently become widely available in a standardized version developed at the University of California at San Diego. UCSD Pascal (TM) has been implemented on a number of inexpensive machines in a form suitable for turtle geometry. We mention in particular the APPLE II (with Pascal package) and the Terak 8510A. (The programs below use the Apple version where there is a difference.) In addition to Pascal's advantages of being a rather well-developed language by the criteria we outlined, this particular version includes built-in turtle commands MOVE, TURN, PENCOLOR(WHITE), PENCOLOR(BLACK), MOVETO, and TURNTO, which correspond, respectively, to Turtle Procedure Notation's FORWARD, LEFT, PENDOWN, PENUP, SETXY, and SETHEADING. There are also TURTLEX, TURTLEY, and TURTLEANG, which correspond to XCOR, YCOR, and HEADING.

Pascal's principal disadvantages are three: Its highly structured form requires the programer to pay attention to certain details of form which we have deliberately omitted from Turtle Procedure Notation. For ex-

ample, one must always declare variables and their types (integer, real, array, etc.) before using them. Moreover, one cannot use the same variable for different types of data. These constraints not only lead to extra typing, but interfere with convenience and flexibility. Second, although Pascal does have facilities for handling compound data objects such as lists, the programer must deal with these structures in terms of special so-called "pointer variables" rather than as first-class data objects. The third disadvantage is that Pascal is a compiled language (in contrast to Logo and BASIC), which means there must be an extra compilation step between writing a program and running it. The way this is generally handled makes it difficult to generate environments of many independent, user-defined programs, such as one would do to create any of the various turtle geometries in this book (cube, three-dimensions, and so on). As a result, the image of creating and using a single program (as opposed to a continuously evolving cluster of procedures) dominates Pascal as it does BASIC.

Procedure Inputs and Outputs

Within a program, Pascal has extensive capabilities to have separate subparts (called procedures) which can take inputs and return values in a number of ways. Inputs specified by the user at run time, however, are usually handled, as in BASIC, by a separate `INPUT` command within the program. Below is a sketch of a program, `TURTLE`, which sets up a environment that allows a user to interactively choose commands according to what is seen developing on the screen. The main part of the program is a monitor loop (at the end of the program) that repeatedly allows the user to type in a command, which the monitor then executes as a procedure. We use a separate procedure `ASK` to ask the user for an input if that is necessary. In this way, procedures such as `FD` (`FORWARD` is reserved for a built-in command) can be used not only as direct user commands (in the monitor), but also in other procedures that do not ask the user for inputs.

```
PROGRAM TURTLE;
USES TURTLEGRAPHICS, APPLESTUFF;
VAR COMMAND: STRING; A, D: INTEGER;

   PROCEDURE ASK(QUESTION: STRING; VAR ANSWER: INTEGER);
   BEGIN
      WRITE(QUESTION);
      READLN(ANSWER)
   END;
```

```
   PROCEDURE GETCOMMAND;
   BEGIN
      REPEAT GRAFMODE UNTIL KEYPRESS;
      TEXTMODE;
      READLN(COMMAND)
      (*THIS ALLOWS GRAPHICS DISPLAY TO BE MAINTAINED UNTIL
        A COMMAND IS TYPED. THE METHOD OF HANDLING MIXED
        TEXT AND GRAPHICS WILL IN GENERAL DEPEND ON YOUR
        SYSTEM AND INGENUITY.*)
   END;

   PROCEDURE FD(D: INTEGER);
   BEGIN
      MOVE(D)
   END;

   PROCEDURE LT(A: INTEGER);
   BEGIN
      TURN(A)
   END;

BEGIN (*MONITOR LOOP*)
   INITTURTLE;
   PENCOLOR(WHITE);
   REPEAT
      GETCOMMAND;
      IF COMMAND='FD' THEN BEGIN
         ASK('HOW FAR? ', D); FD(D) END;
      IF COMMAND='LT' THEN
         BEGIN ASK('HOW MUCH? ', A); LT(A) END;
      IF COMMAND='CS' THEN
         BEGIN INITTURTLE; PENCOLOR(WHITE) END;
      IF COMMAND='PU' THEN PENCOLOR(NONE);
      IF COMMAND='PD' THEN PENCOLOR(WHITE);
   UNTIL COMMAND='DONE';
END.
```

Of course, this can serve as a model for setting up other environments for exploration. Unfortunately, this system-type organization does not allow the user to write even elementary experimental programs without recompiling.

Repetition and Recursion

Pascal has a wide variety of REPEAT UNTIL, WHILE, FOR constructs similar to those in Turtle Procedure Notation. Here are two samples, a POLY procedure (to be used as an extension of the TURTLE environment above) and a SQRT function (to be used as part of some main program). See also the use of FOR in TURTLE3D below.

```
PROCEDURE POLY(A, D:INTEGER);
VAR HEAD, INITHEAD: INTEGER;
BEGIN
   INITHEAD := TURTLEANG;
   REPEAT
      FD(D);
      LT(A);
      HEAD := TURTLEANG;
   UNTIL HEAD = INITHEAD;
END;
```

```
FUNCTION SQRT(A, B:REAL):REAL;
VAR GUESS: REAL;
BEGIN
   GUESS := 1.0;
   WHILE ABS(A - GUESS*GUESS) > 0.001 DO
      GUESS := (GUESS + A/GUESS)/2;
   SQRT := GUESS
END;
```

Recursion is also neat in Pascal. Here is HILBERT again:

```
PROGRAM HILBERT;
USES TURTLEGRAPHICS, APPLESTUFF;
VAR L, P, S: INTEGER;

   PROCEDURE HIL(L, P:INTEGER);
   BEGIN
      IF NOT (L = 0) THEN
         BEGIN
            TURN(P*90);
            HIL(L-1, -P);
            MOVE(S);
            TURN(-P*90);
```

```
            HIL(L-1, P);
            MOVE(S);
            HIL(L-1, P);
            TURN (-P*90);
            MOVE(S);
            HIL(L-1, -P);
            TURN(P*90)
         END
   END;

BEGIN
   WRITE('SIZE? '); READLN(S);
   WRITE('LEVEL? '); READLN(L);
   INITTURTLE;
   PENCOLOR(WHITE);
   HIL(L, 1);
   REPEAT UNTIL KEYPRESS; TEXTMODE
END.
```

Data Types

Pascal's compound-data types are adequate for turtle-geometry work. For vectors one will probably want to use arrays. Here are some service procedures in the context of a sketch of a three-dimensional turtle program (see subsection 3.4.3). Unfortunately, Pascal doesn't allow functions to output compound-data types, so subroutining vector arithmetic is not worth the effort. On the other hand, call-by-name parameters (see ROTATE below) make some types of vector operations transparent.

```
PROGRAM TURTLE3D;
USES TURTLEGRAPHICS, TRANSCEND, APPLESTUFF;
TYPE VECTOR = ARRAY[1..3] OF REAL;
   VAR I: INTEGER;
   A, D, LDIST: REAL;
   COMMAND: STRING;
   P, H, L, U, TEMP: VECTOR;

   PROCEDURE INIT;
   (*THIS PROCEDURE SHOULD SET INITIAL VALUES FOR LDIST, P, H, L
     U, AND PERFORM OTHER INITIALIZING ACTIVITIES*)

   PROCEDURE ASK; (*AS IN TURTLE PROGRAM ABOVE*)
```

```
   PROCEDURE ROTATE(VAR V, PERPV: VECTOR; A: INTEGER);
   VAR AR: REAL;
   BEGIN
      AR := A * 3.14159/180.0;
      FOR I := 1 TO 3 DO
         TEMP[I] := COS(AR)*V[I] + SIN(AR)*PERPV[I];
      FOR I := 1 TO 3 DO
         PERPV[I] := COS(AR)*PERPV[I] - SIN(AR)*V[I];
      V := TEMP
   END;

   PROCEDURE PROJECT(V: VECTOR; VAR X,Y: INTEGER);
   BEGIN
      X := ROUND(LDIST*P[1]/P[3]);
      Y := ROUND(LDIST*P[2]/P[3]);
      (*NOTE ROUNDING (CONVERTING TO INTEGER FOR MOVETO).
        APPLE DOES AUTOMATIC CLIPPING; OTHER SYSTEMS MAY
        REQUIRE SOME WORK TO MAKE SURE X AND Y ARE WITHIN
        RANGE REQUIRED BY MOVETO.*)
   END;

   PROCEDURE YAW(A: INTEGER);
   BEGIN
      ROTATE(H, L, A)
         (*PITCH OR ROLL ARE SIMILAR, USING H AND U OR
           U AND L AS INPUTS TO ROTATE*)
   END;

   PROCEDURE FD(D: INTEGER);
   VAR X, Y: INTEGER;
   BEGIN
      FOR I := 1 TO 3 DO
         P[I] := P[I] + D*H[I];
      PROJECT(P, X, Y);
      MOVETO(X+140, Y+95)
   END;

BEGIN
   (*MAIN LOOP HERE SIMILAR TO THAT OF THE PREVIOUS TURTLE
     PROGRAM. IT SHOULD INCLUDE INIT PROCEDURE.*)
END.
```

The program is intended to use the built-in penup and pendown mechanism (PENCOLOR) rather than keeping its own pen-state variable.

UCSD Pascal assumes that inputs to turtle (and other graphics) commands are integers, and you must make sure of this either by computing with integers or by converting to integers before calling them. You can use the built-in ROUND function as we did in PROJECT.

Other Languages

APL is quite suitable for turtle geometry. It is procedure-oriented, with inputs and outputs. It allows "environment building" in the way Logo does. Furthermore, it is recursive and handles compound data, such as vectors, in the cleanest fashion of any language mentioned here. APL has not yet been picked up by any of the major producers of inexpensive home computers. We do, however, know of at least one major installation that has turtle commands built into APL.

Lisp is another language that is structurally very compatible with our criteria. In fact, Logo is a direct descendant of Lisp. But so far Lisp has not been much associated with inexpensive, graphics-oriented hardware.

Smalltalk is a language that, like Logo, was developed explicitly for educational purposes. Not surprisingly, it fares well by our criteria, especially since it was designed to take advantage of the capabilities of high-resolution graphics and to be an extensible, interactive medium. Though it has not been publicly released by its developers at Xerox Corporation, there are plans to make the latest version, Smalltalk 80, available in the near future. Smalltalk programs are organized in a manner very different from the step-by-step procedure model that typifies the languages discussed above. Instead, the "knowledge" in a Smalltalk program is distributed among a number of "objects" which communicate among themselves by means of "messages." For example, a turtle might be a kind of object that knows how to respond to messages FORWARD, RIGHT, and so on. This so-called "object-oriented programing" has many advantages in dealing with graphics and simulations, and in particular with the kind of multiple turtle projects discussed in section 2.2. Although Smalltalk's fundamental constructs are very different from those of Turtle Procedure Notation, the translation of any of the above programs into Smalltalk is easy and elegant, with none of the difficulties of modularity and environment building encountered with Basic and Pascal. One can also expect Smalltalk implementations to include built-in turtle graphics.

Turtle Geometry

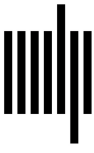
The Computer as a Medium for Exploring Mathematics

By: Harold Abelson, Andrea diSessa

Citation:

Turtle Geometry: The Computer as a Medium for Exploring Mathematics
By: Harold Abelson, Andrea diSessa
DOI: 10.7551/mitpress/6933.001.0001
ISBN (electronic): 9780262362740
Publisher: The MIT Press
Published: 1986

The open access edition of this book was made possible by generous funding and support from Arcadia – a charitable fund of Lisbet Rausing and Peter Baldwin



The MIT Press

First MIT Press paperback edition, 1986

This book was set using the TEX typesetting system by Michael Sannella and printed and bound by Halliday Lithograph in the United States of America.

Library of Congress Cataloging in Publication Data

Abelson, Harold.
Turtle geometry.
(The MIT Press series in artificial intelligence)
Includes index.
1. Geometry—Study and teaching. 2. Computer-assisted instruction. I. DiSessa, Andrea, joint author. II. Title. III. Series: MIT Press series in artificial intelligence.
QA462.A23 1981 516'.007'8 80-25620

ISBN 978-0-262-01063-4 (hardcover : alk. paper) —978-0-262-51037-0 (paperback)