

THE UNIVERSITY OF CALGARY

Inducing Procedures Interactively

Adventures with Metamouse

by

David Maulsby

A thesis submitted to the Faculty of Graduate Studies
in partial fulfillment of the requirements for the degree
of

Master of Science

Department of Computer Science

Calgary, Alberta

December, 1988

© David Maulsby 1988

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.


L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

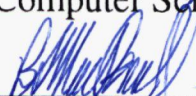
ISBN 0-315-50339-4

The University of Calgary
Faculty of Graduate Studies

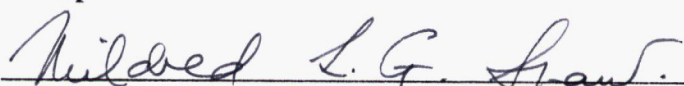
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Inducing Procedures Interactively," submitted by David Maulsby in partial fulfillment of the requirements for the degree of Master of Science.



Supervisor Ian H. Witten
Computer Science



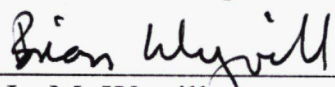
Bruce A. MacDonald
Computer Science



Mildred L. G. Shaw
Computer Science



Ron Wardell
Environmental Design



Brian L. M. Wyvill
Computer Science

Date December 8, 1988

Abstract

Direct-manipulation interfaces have greatly extended the class of casual computer users and encouraged them to conceptualize the system through metaphors. They have not, however, successfully incorporated facilities for end-user programming without breaking out of the direct-manipulation paradigm.

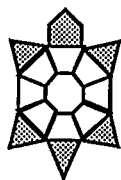
This thesis supports the contention that “teaching” provides an appropriate metaphor for programming in such an environment. It presents a system for inducing procedures that enables users of a graphics editor to teach it routine tasks by working through example traces. A central problem in the design is to meet the requirements for instructibility without imposing excessive demands on the teacher.

A key component of the system is its teaching metaphor, a graphical apprentice called Metamouse. Metamouse is the target of the teacher’s demonstrations. It is an eager learner designed to encourage constructive methods, clarify ambiguous situations, reduce errors and extraneous activity, and discourage free variation in teaching. Its behavior is expected to be understood by users at a metaphorical, intentional level rather than from a precise specification.

Metamouse has been fully designed but not yet fully implemented. However, a pilot system has induced procedures with variables, generalized actions, conditional branches and loops. Its ability to reduce errors and extraneous activity by prediction, and to identify underspecification, has been demonstrated. Tests showed that the metaphor is easily understood. Consequently the thesis argues that it is feasible for a system to induce procedures interactively from casual users. This significantly broadens the scope of application of machine learning techniques and opens new areas of research in knowledge acquisition. It facilitates the investigation of intelligent user interfaces and, last but not least, benefits the many users of interactive graphics systems.

Parturient montes, nascetur ridicula testudo.

— after Horace



Acknowledgements

This research has been supported in part by a Natural Sciences and Engineering Research Council of Canada Post-Graduate Scholarship, and by Graduate Teaching and Research Assistantships from the University of Calgary.

I would like to thank my supervisor, Ian Witten, for his editorial thoroughness, and Tamara Lee for excellent work on the figures, some of which I know were taxing indeed. I would also like to thank the Department of Computer Science support staff — Bev, Bonnie, Camille, Dolores, Joan, Lesia, Lorraine, Marion and Parin — for all their patient help.

Research may appear to be a solitary endeavor, but I have found that most ideas result from collective cogitation. I have been fortunate to work in a lively research environment at Calgary. Seminars and meetings have done their part. Much more, however, has come from casual conversation, looking over someone's shoulder, misunderstanding, bluffing, and wild speculation. Bibliographic citations ignore these vital resources. My fellow students — Saul Greenberg, David Pauli, Jeff Allan, Debbie Leishman, Bruce Conrad, Rosanna Heise, Thong Phan, Mike Hermann, Phong Truong, Dan Mo, Dan Freedman and Brent Krawchuk — have all contributed to this thesis, often unknowingly. I wish them all the best with their own research. Ken Kittlitz has taken up the cause of Metamouse — I hope he enjoys it as much as I have.

I enjoyed not only excellent supervision from Ian Witten, but also the help of Bruce MacDonald; our discussions in the early stages of this research gave the project its shape and direction. I very much appreciated Bruce's insistence on defining a mission, and Ian's willingness to let me dream a little.

Contents

Approval	ii
Abstract	iii
Acknowledgements.....	iv
Contents.....	v
List of Figures.....	viii
List of Tables.....	viii
 1 Teaching a Mouse How to Draw	 1
1.1 Drawing Programs	1
1.2 Drawing Procedures	3
1.3 Programming by Example.....	10
1.4 Outline of Thesis	15
1.5 Introducing Metamouse.....	17
1.6 Box-to-Line—a Worked Example.....	17
 2 Meeting the Felicity Conditions	 20
2.1 User Study	21
2.1.1 Tasks.....	21
2.1.2 Observations	23
2.2 Towards a Theory of Drawing.....	27
2.2.1 Empirical Studies.....	28
2.2.2 Phenomenology and Construction	29
2.3 Felicity Conditions	34
2.3.1 Correctness.....	35
2.3.2 Show Work.....	36
2.3.3 No Invisible Objects	37
2.3.4 Minimal Activity.....	38
2.4 Design Principles.....	39
 3 Metamouse	 41
3.1 A.Sq—the Drawing World.....	42
3.2 Basil—the Metamouse.....	45
3.2.1 Basil’s Body	46
3.2.2 Basil’s Sensory System	48
3.2.3 Basil’s Memory.....	52

3.3	Generalization—Actions.....	56
3.3.1	Variables.....	58
3.3.2	Constraints.....	59
3.4	Generalization—Procedures	61
3.4.1	Definitions.....	62
3.4.2	Algorithm.....	63
3.5	Putting the System Together	65
4	An Implementation.....	67
4.1	Phase 0 Implementation	67
4.2	A.Sq	69
4.3	Basil.....	74
4.4	Daedalos.....	77
4.5	Generalization.....	79
4.5.1	Variables.....	79
4.5.2	Constraints.....	81
4.6	A Worked Example	85
5	Three Empirical Studies.....	95
5.1	The Metamouse Metaphor.....	95
5.2	Inducing Procedures.....	98
5.2.1	Box-to-Line.....	99
5.2.2	Picket Fence.....	100
5.2.3	Connectivity.....	101
5.3	Learning without Prediction.....	102
6	What Have We Learned from Metamouse?.....	104
6.1	Project Status	104
6.2	Empirical Studies.....	105
6.2.1	The Metamouse Metaphor.....	105
6.2.2	Inductive Generalization.....	106
6.2.3	Benefits of Interaction	107
6.3	Analytic Evaluation of the System.....	108
6.3.1	Representing Problems	108
6.3.2	Distinguishing Programs	109
6.3.3	Sensitivity to Teaching Sequence.....	110

6.4 Support for the Thesis	110
6.5 Further Work Proposed	112
6.5.1 Integrated System	112
6.5.2 Human Factors Studies.....	112
6.5.3 Graphical Domain Analysis	113
6.5.4 Generalization and Learning	114
6.6 Summary and Conclusion.....	114
References	117
Appendix A.....	120

List of Figures

1.1	The “box-to-line” task.....	4
1.2	Using a sweep-line.....	4
1.3	Rules for positioning labels at arrowheads.....	5
1.4	Positioning a label at an arrowhead.....	6
1.5	Constructing the cyclic order of points.....	7
1.6	Distributing boxes along a line.....	8
1.7	Description of Metamouse given to teachers	16
1.8	Teaching Metamouse a trace of “box-to-line”	18
2.1	Left-to-right ordering of points	30
2.2	Use of sweeping methods to distinguish order	32
2.3	Two concepts of “thin box”	33
2.4	Spatial relation defined by an “invisible object”.....	37
3.1	Prototype system for programming graphics by example	42
3.2	Parts of A.Sq objects.....	43
3.4	Touch relations in sensory feedback	50
3.5	Program graph for “box-to-line”.....	54
3.6	Definition and instantiation of a variable.....	59
3.7	Main modules of graphical programming system.....	66
4.1	A.Sq user interface.....	69
4.2	Class hierarchy of A.Sq graphical objects.....	72
4.3	A.Sq system data structure.....	73
5.1	Performance of three typical subjects on Basil questionnaire	97
5.2	Connectivity task	101
6.1	A graphical counting device	109

List of Tables

5.1	Learning system performance.....	99
5.2.	Performance data for passive vs interactive modes.....	103

Chapter 1

Teaching a Mouse How to Draw

At one time not so long ago, the task of preparing charts and diagrams presented the vast majority of professionals and students with a dilemma. To draw them by hand required a great deal of time and effort with no guarantee of satisfactory results. To hire someone else to draw them implied considerable expense. Either approach was bound to be time-consuming and onerous. Within the past five years however, drawing with the help of a computer has become widely available and popular. Anyone with access to a personal computer has the opportunity to draw with powerful and efficient software tools. The ability to edit pictures without using an eraser is perhaps the greatest convenience of all.

The result is plainly visible in the workplace: the quality of drawings in unpublished documents has improved tremendously. On the other hand, it appears that the amount of time people spend drawing has increased as they produce more pictures to higher personal (and communal) standards of draftsmanship. Of course, computers breed perfectionism — out of nowhere springs a new concern for the semantic implications of alignment and centering. Nonetheless, much of the effort people put into drawing with computers is surely worthwhile. The problem is that popular drawing programs do not help their users as much as they could with delicate and repetitive tasks. This thesis proposes the use of programming-by-example to address this problem, so that computer users can meet their drafting standards and concentrate more upon the design and meaning of their creations.

1.1 Drawing Programs

The average user of a drawing editor is quite unaware that she¹ is really specifying a program. The static picture she sends off to the laser printer is translated for her into a sequence of device-driver commands. When Sutherland first experimented with SKETCHPAD, computer drawing was very much like programming [Sutherland 63]. Images intended for production on a graphics plotter were typically FORTRAN programs. The user of SKETCHPAD could program interactively and incrementally, toggling groups of

¹ Please note that the use of singular pronouns is a matter of convenience; any distinction between male and female is deemed irrelevant to the subject matter of this thesis.

switches to select shapes and twisting dials to set their parameters, with the results immediately visible on a cathode-ray tube. Sutherland went even further, introducing an interactive tool called the *rubber-band line*. With this the user could literally rough out a picture for SKETCHPAD'S constraint-satisfaction system to beautify.

In the following two decades interactive computer graphics developed steadily. By the mid-1970's, the digitizing tablet and color raster display made painting programs feasible. Using such a program was very much like painting with watercolors, or with oils, or like drawing with an air-brush — or like nothing that could have been done so simply by hand. These programs illustrated the practicality of interactive graphics, but also the potential for entirely novel methods of drawing made possible by *computation on an internal representation of the picture*.

In the early 1980's programs like MacPaint and MacDraw brought the basic capabilities of SKETCHPAD and paint systems into the popular domain [MacDraw 87]. These programs stress the benefits of utterly concealing the internal representation and the computational model; they attempt to sustain the illusion of drawing on paper (this illusion gets shattered now and then, as when MacDraw exposes the peculiar logic of its "alignment" commands). They offer the user a kit of graphical tools that have great intuitive appeal — greater perhaps than the physical ruler and compass.

Despite their obvious virtues, contemporary drawing programs have limitations that rob users of their time and patience. Surprisingly, the most prevalent and annoying of these can be overcome by reintroducing the very activity that has been banned — programming. Examples of functional limitations prevalent in popular editors include:

1. The lack of alignment facilities. MacDraw, for example, can align objects with reference to their bounding boxes — at their centers, or a common edge or corner. It cannot align the left edge of one box to the right edge of another. Nor will it move objects to a guideline of arbitrary orientation given by the user.

2. The difficulty of positioning objects exactly as desired. Most commercial drawing programs provide a reduced resolution grid or object-gravity for exact positioning. Anyone who has used these knows of their virtues, but also their deficiencies. A gravity grid relentlessly frustrates attempts at fine adjustments of size or position until it is turned off. Object gravity does not typically support such useful operations as bisecting a line.

3. The lack of facilities for creating specialized shapes. MacDraw and MacDraft, though widely used, do not provide such useful shapes as hexagons, parallelograms, and isosceles triangles, let alone *n*-gons. Perhaps this is because these are not supplied with the Macintosh firmware.

4. The lack of user-specifiable constraints. If a drawing program does not support the control of spatial relations then the user must “debug” the rest of her picture whenever she edits some member of a constrained relation. Drawing programs support the constraints inherent in their graphical tools — for example, that the edges of a polygon remain connected — and also permit fixed relations by grouping elements. But suppose the user wants an edge to remain vertical even when one of its end-points is moved. This local, dynamically satisfied constraint is inexpressible.

The items in this brief catalogue have key features in common. They all require that a constraint be specified. They have multiple parameters that must be given at run-time; they are just a little too complex for the “friendly” direct-manipulation interfaces of popular drawing programs. They are too complex for typical macro-defining facilities such as [Tempo 86], yet even casual users know how to perform them manually. Commissioning an application programmer would be impractical. Yet how can the typical *user* of MacDraw be expected to *re-program* MacDraw?

1.2 Drawing Procedures

An answer to this question arises from the way users produce and refine their “program specifications” for the hard-copy device using MacDraw. The following tasks are good candidates for programming; all are useful, and some would be quite difficult for a non-expert to program in a formal language.

First, an alignment operation not supported by MacDraw. The task is to move one or more boxes onto a guideline of arbitrary orientation, so that all boxes lie entirely to one side of the line. The procedure, called “box-to-line,” is illustrated in Figure 1.1. The user/executor of this procedure decides *ad hoc* where to place the guideline and on which side the boxes shall lie when aligned. The first of these input parameters is drawn by the user, the second is inferred from the way the user re-positions the first box. An algorithm for “box-to-line” is given below.

box-to-line:

ask user to draw the guideline, G
 ask user to move the first box, B_1 , to G
 note which corner, C , of B_1 is on G (*ie.* $B_1.C$ is on G)
 for each box, B_i , of those remaining:
 move B_i until $B_i.C$ is on G
 remove G

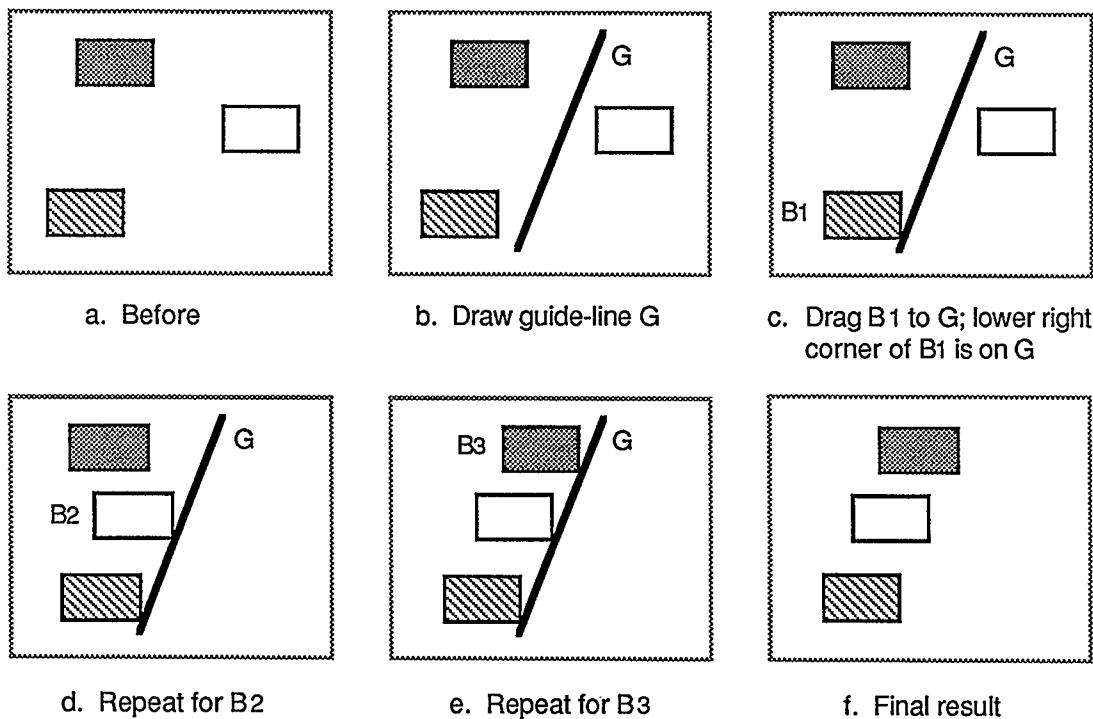


Figure 1.1 The “box-to-line” task

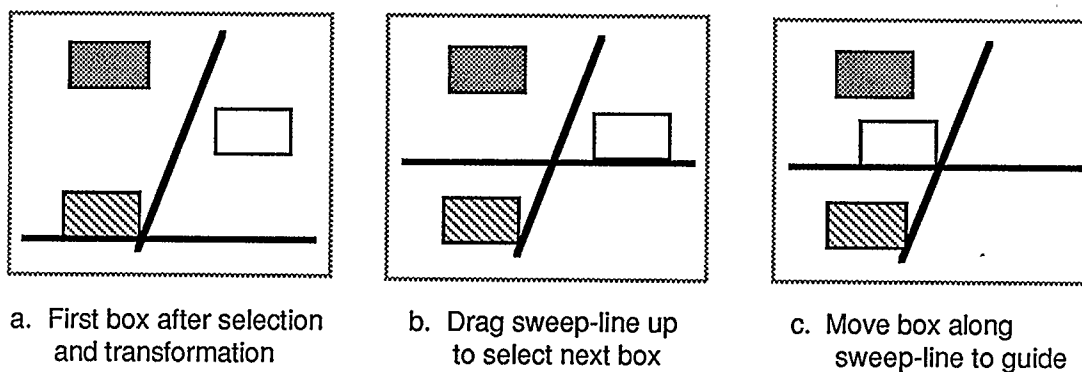


Figure 1.2 Using a sweep-line

The path a box takes from its origin to its target position may be more or less constrained — along the horizontal, or perpendicular to the guideline, etc. This illustrates a problem in specifying a graphics editing procedure: isolating a clearly defined task from the user's goal set. After all, she might move the boxes so as to refine other aspects of their arrangement at the same time. Figure 1.2 illustrates the use of a horizontal “sweep-line” to select boxes and specify their path.

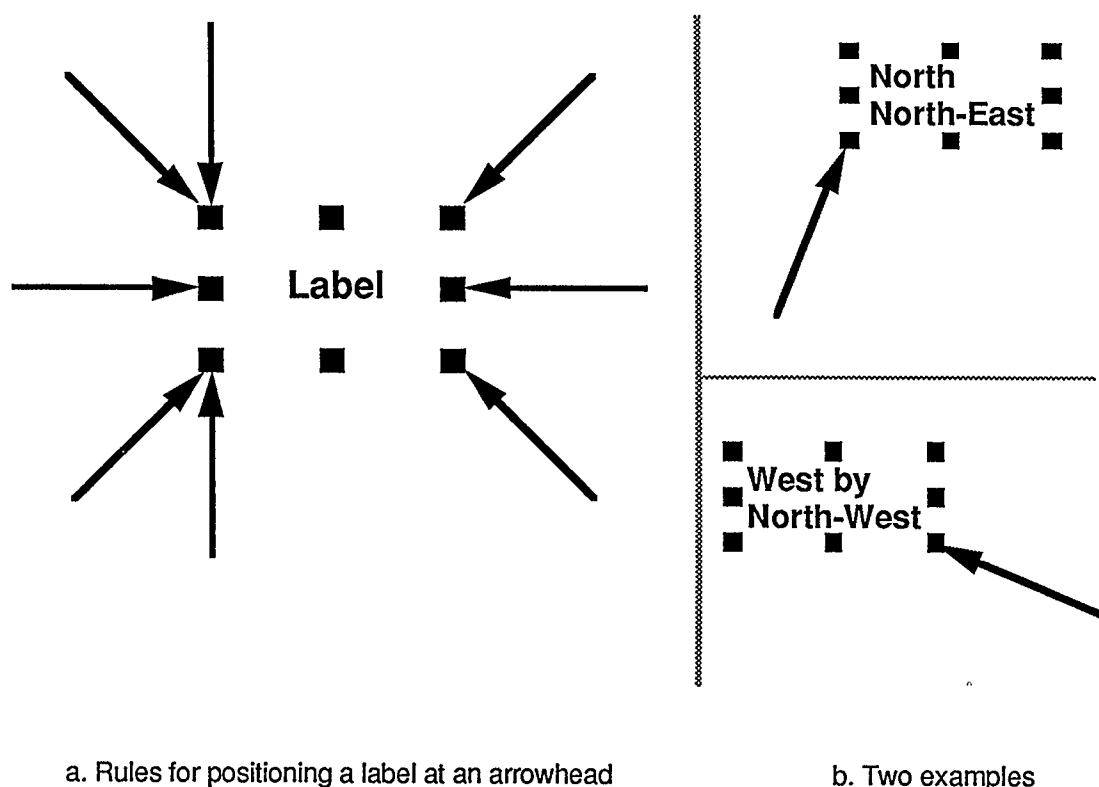


Figure 1.3 Rules for positioning labels at arrowheads

The positioning of textual labels near arrowheads is an extension of the alignment problem; some selected part of the label is made collinear with the arrow, but at a certain distance from its end point. Figure 1.3 shows a graphical declaration of the rules for positioning labels, where each case illustrated represents the center of the range over which its rules applies.

Obviously, this declarative approach involves implicit information — conventions that interpret the cases. A procedural, constructive specification can be more self-contained, by demonstrating the range over which a rule applies. A rotating sweep line, as on a radar scope, measures the angle to the particular arrow from a standard initial position in Figure 1.4. The sweep line pauses at each of the stations where one rule gives way to the next. If it crosses the arrowhead on its journey between two stations it will stop sweeping. The label is then moved into position at the arrowhead in accordance with the currently active rule.

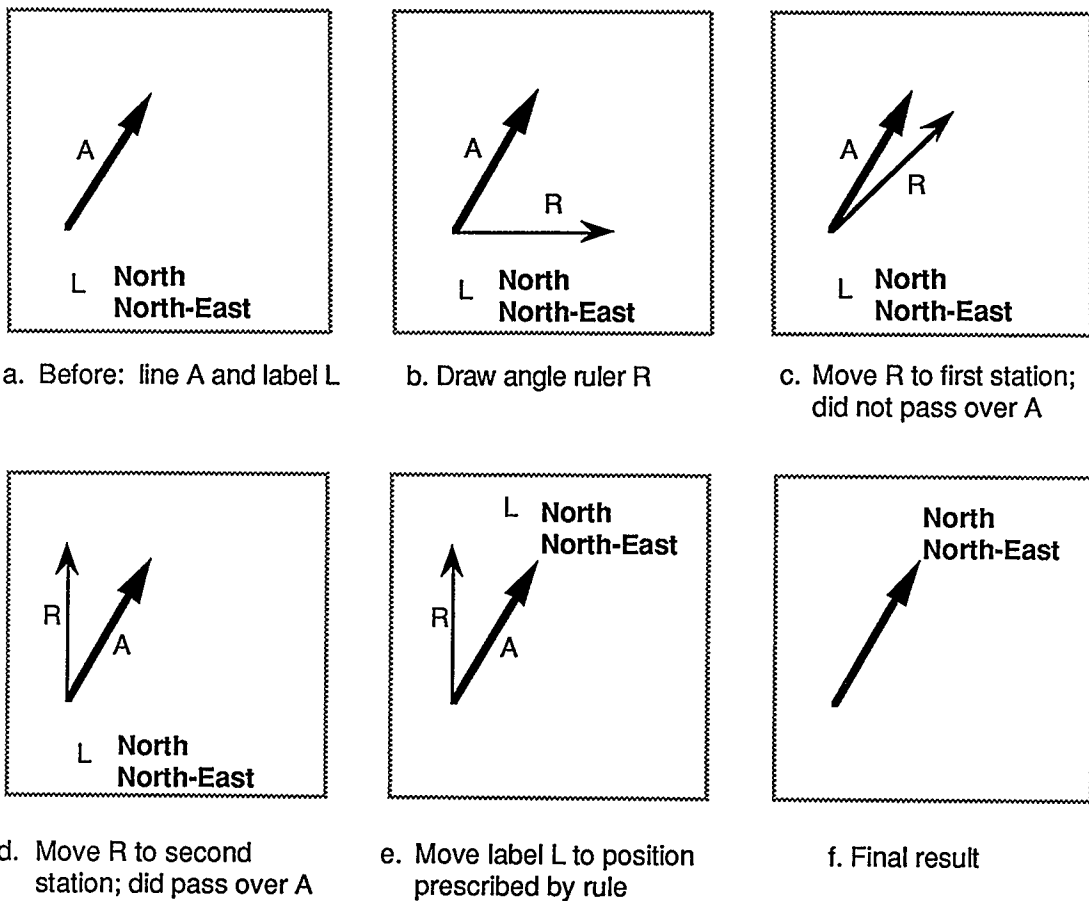


Figure 1.4 Positioning a label at an arrowhead

Next, let us examine a task from computational geometry — finding the convex hull of a set of points. This procedure may not be used much but it is nicely illustrative. Suppose we have some key points in our drawing, and want to make a polygon around them. We choose our polygon tool, anchor it at the starting vertex, and then proceed around the

vertices in a rotational order, say, counter-clockwise. We have just performed Jarvis' March, a classic algorithm [Preparata 85].

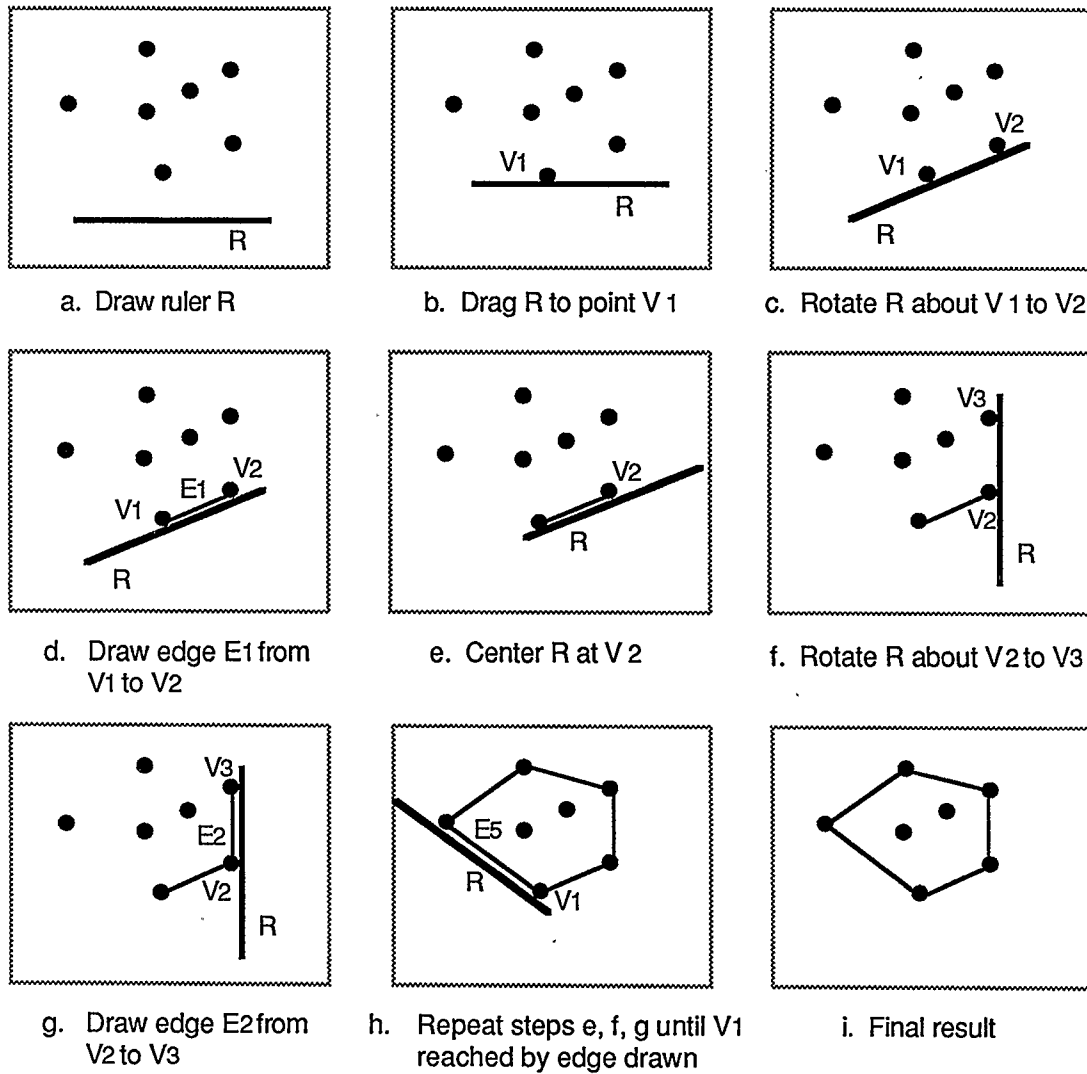


Figure 1.5 Constructing the cyclic order of points

We took advantage of an ordering of the data, without having to sort them or know how to express their ordering in mathematical terms. The graphical procedure was easy,

but specifying it in words requires special knowledge. Adopting more expressive graphical methods, shown in Figure 1.5, yields a constructive form of Jarvis' March:

Jarvis' march:

```

draw the ruler line, R, near the bottom of the display
move R upwards until it touches some vertex,  $V_1$  (a point on convex hull)
for each vertex,  $V_n$ , encountered until done:
    rotate R counter-clockwise about  $V_n$  until it touches another vertex,  $V_{n+1}$ 
    draw a line segment from  $V_n$  to  $V_{n+1}$ 
    slide R along  $(V_n, V_{n+1})$  until it is centered on  $V_{n+1}$ 
    if  $V_{n+1}$  is  $V_1$ , signal done
remove ruler line R
  
```

Finally, consider the fairly simple, useful task of arranging some boxes in an equally spaced row; that is, such that a gap of constant width separates each box from its neighbor to the left. Figure 1.6 illustrates the procedure.

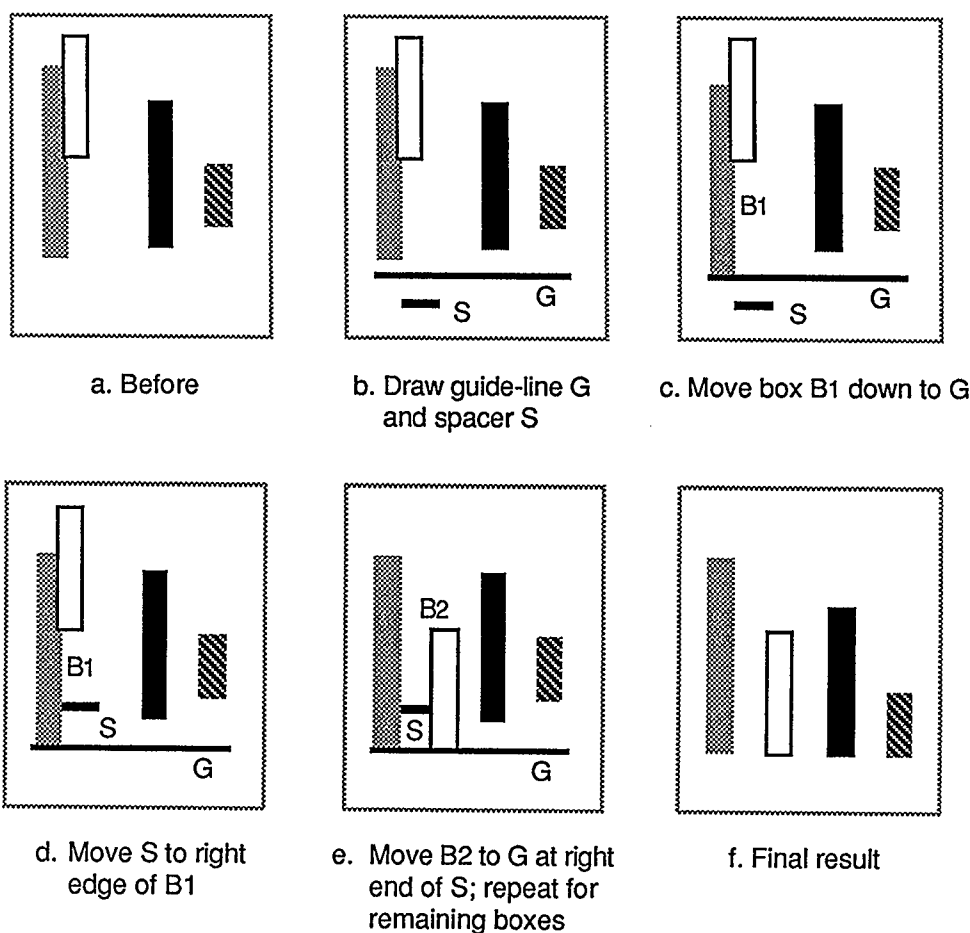


Figure 1.6 Distributing boxes along a line

The input to be filtered through this “picket-fence” procedure is the set of boxes. Two other inputs parameterize it: a horizontal guideline that represents the ground underneath the boxes, and a horizontal spacer line that specifies the size of the gap. If these parameter objects are invisible, then the user is performing measurement by *visual inspection*; if visible, she is *constructing*, in the traditions of geometry and drafting. The procedure involves selecting and translating each box to the guideline, with the additional constraint that the spacer separate it from its neighbor to the left. If the user is systematic about this, she is following this algorithm:

Picket-fence:

```

create horizontal guideline G
create horizontal spacer S
translate S until it lies above but near left end of G
for each box  $B_i$  in the input set:
    translate  $B_i$  until both
        1. bottom edge of  $B_i$  lies on G, and
        2. right end of S lies on left edge of  $B_i$ 
    translate S until left end of S lies on right side of  $B_i$ 
remove S and G

```

Notice that some of the terms in this algorithm, as in the others above, are somewhat vague; for example, the phrase, “lies above but near”. Moreover, the algorithm does not determine the order in which boxes are selected. These under-constrained decisions are not really important to the task, and are easily settled if need be; they are the sort of trivial decisions the user makes *ad hoc*. Thus the algorithms examined above may well describe what a human being would do in carrying out such routine tasks, but lack the determinism expected of computer programs. To ask the user to write a deterministic program in a suitably unambiguous, abstract language would be asking quite a lot: her “natural algorithm” is expressed in a visually and kinesthetically reactive intercourse with a picture; she does not have to account for what *might* happen, nor define trivial parameters of her actions. The average user, even one who has a talent for programming, would be disinclined to write such programs. Short-term economy would easily convince her that the task is “not worth programming.”

This thesis proposes that an amateur draftsman can indeed create programs to accomplish constraint satisfaction and other tasks such as sorting, without having to abandon her natural mode of work. She already, albeit indirectly and unwittingly, programs the graphics plotter through the mediation of the drawing system. Another

intermediary, another level of indirection, enables her to program new functions for the drawing system itself. A graphical device called the *Metamouse* expresses what the physical input device, the mouse, *would* be doing under user control during execution of a task. The Metamouse is connected to a learning system that records and generalizes actions and induces the program's control structure. The system learns a program incrementally so that the programmer need only execute those parts of it that accomplish her current task. In short, it is a system for programming by graphical demonstration.

1.3 Programming by Example

Graphical programming almost inevitably involves the production of an example, since particular representative objects must take the place of symbols. The tradition of graphical programming is entwined with that of programming by example, beginning with SKETCHPAD. A recent taxonomy of all programming systems [Myers 86Chi] classified them according to three characteristics of the program translator: whether it is batch or interactive; whether it processes a textual or visual representation, where "visual" refers to the significance of two (or more) dimensions of input; and whether it analyzes examples (of input and output, or of execution). Systems distinguished in the following discussion as *graphical* are called "interactive, visual programming by example" in Myers' taxonomy. Graphical approaches differ markedly, however, in their use of symbols, examples, and inductive inference. A system for the programming tasks described in the previous section ought to require no symbolic convention apart from those already present in the user interface. Instead it should infer the attributes of program objects such as constants, variables and control structures from representative example objects, based on an inductive hypothesis that multiple example instances have a common reference. Visual programming systems already in the literature do not meet this requirement.

The computational complexity of inductive inference under various conditions has been thoroughly characterized [Angluin 83]. A graphical system that infers a program from its input-output pairings alone is impractical owing to the enormous number of possible programs. Fortunately this is not required for interactive graphics, where execution traces are available. Nonetheless a trace of manual execution may well contain inputs and outputs of complex functions computed but not expressed by the user. Existing systems can deal with this problem by requiring that she augment her examples with symbolic specifications,

or tackle it directly by trying to induce the function. We will see later that non-symbolic augmentation is another option. A further difficulty in analyzing traces is to identify the conditions that govern branching and looping. Some systems require the user to annotate her trace with symbolic markers at these decision points. This is clearly undesirable in graphics, although it seems impracticable to escape from marking at least one decision — task closure.

The remainder of this section describes graphical programming systems already in the literature, in light of the issues raised above.

The user of SKETCHPAD [Sutherland 63] could not program a graphical transformation, because the system recorded only the data structures produced by an interaction sequence. Thus, SKETCHPAD demonstrated the simplest type of graphical programming — using graphical input to set the values of system-defined object attributes.

The LOGO system [Papert 80] records procedures enacted with a graphical, robotic “Turtle.” LOGO programs create and transform hierarchically structured objects. The underlying language is general-purpose, Turing-complete, and permits the recombination of objects and actions by invoking them as subroutines. The system introduces the idea of a programming metaphor — the Turtle whose physical functions correspond with the programming language’s operators. Even young children have no trouble learning how to control a Turtle. LOGO seems very close to what we want. Unfortunately, the LOGO Turtle is too literal-minded. Despite the coordinate-frame independence achieved by using body-centered coordinates in Turtle geometry [Abelson 80], the system can record only numeric constraints on an object’s relative Cartesian coordinates or on distance moved; it cannot capture constraints between named parts of objects. Moreover, interactive sequences are recorded as given, without the inference of variables or control structures. If an action is meant to be iterative, the programmer must edit a textual version of the procedure to make it so.

THINGLAB [Borning 86] has excellent facilities for describing and solving constraints. The programmer illustrates a constraint, and the method of solving it, by a combination of pictures and text. This is not so awkward as it sounds; a pictorial example is simply annotated with textual labels that name (graphically selected) points whose coordinates are variables of the program. The important limitation of THINGLAB with respect to graphics applications is the declarative method used to specify the program. The user must draw an

equational network (in which nodes are quantities or operators) — a nice representation, but nonetheless requiring the user to have a mathematical model of her problem.

To specify a constraint without having to build a symbolic model the user must be able to demonstrate its effect on example objects. The SNAP-DRAGGING technique [Bier 86] shows how this can be done for binary constraints (such as extending the end-point of one line to the mid-point of another) by letting the user point to the parts of objects that snap onto each other. SNAP-DRAGGING does not program a constraint solution; the purpose of a demonstration is to produce a new tool for shaping objects interactively. Thus the programming element of SNAP-DRAGGING is nothing more than the setting of parameters, as in SKETCHPAD. Nonetheless, the system demonstrates interesting advances in graphical interaction. Heuristics about drawing (for example, that the translation of one vertex of a polygon is often intended to align it with some other vertex) are combined with a strong model of geometric relationships (and the construction tools expressing them) so that the system can automatically generate appropriate tools for the operation the user appears to be engaged in.

Another approach to specifying constraints by constructive techniques is the programming language L.E.G.O. [Fuller 86]. With four primitives (*point*, *line*, *circle*, and *intersection*, an operator that returns one or two points of intersection between objects) the programmer constructs relationships by traditional ruler and compass techniques. As in THINGLAB, variables are identified by naming points — in this case those returned by *intersection*. A single demonstration generates a LISP function. The programmer must explicitly identify input and output variables and control structures by (textually) editing her LISP program.

The five systems described above demonstrate advanced facilities for programming in the domain of geometry, but have minimal facilities for programming by example. The following five systems exhibit the reverse. The first, PYGMALION [Smith 75], is a general-purpose visual programming system. Its basic graphical construct is the box, semantically equivalent to “()” in LISP. A program’s details are textual but arranged graphically; the nesting of boxes visualizes logical and scoping relations. Since a box may contain a value, it can replace a named variable. Thus the programmer may give specific data which the system generalizes to variables. The programmer defines each program step in order of execution, annotating the trace by symbolic though pictorial (*ie.* “iconic”) markers for

branches and loops. PYGMALION cannot discover branching and iteration inductively, but does permit incremental programming, since each branch is developed only when it first needs to be executed.

The SMALLSTAR system [Halbert 84] also employs the demonstrative method with symbolic annotation. Here, the trace is itself graphical — the system observes the user / programmer (the distinction becomes increasingly blurred) carry out a task in a desk-top interface with most input coming through the locator device (mouse). SMALLSTAR does no inference however; the programmer must identify constants and variables and insert control structures by editing her program. Halbert provides a convenient form-filling dialogue for this purpose. It works well since only the inherent attributes of an object (*eg.* the name of a file) can be selected as constants and variables. Clearly, this would not be useful when programming graphics, since the relevant values may well be spatial relations.

Programming by demonstration has been used to create user interfaces, as in the recent PERIDOT system [Myers 87]. The programmer defines the screen-layout graphically, using the techniques of MENULAY [Myers 86CG], then demonstrates relationships that hold between program data and their graphical presentation. For example, to establish the height of a scroll-bar as a function of the current position of a buffer window on a file, the programmer manually adjusts the former and sets extremal values for the latter (0% and 100%); the system infers a linear relation. The interpretation of actions with an input device, such as the mouse, is also induced from a demonstration. A moveable mouse icon represents the actual locator in its spatial context. To show that the mouse can grab the scroll bar and move it (thereby adjusting the buffer window), the programmer moves the mouse icon to the scroll-bar, selects the mouse icon's button to show that it is to be pressed during this operation, and then with the real mouse grabs and moves the scroll-bar. PERIDOT deduces that the scroll-bar and the buffer position are to be adjusted under mouse control. Although demonstrations can be action sequences, PERIDOT does not learn procedures but only relations between actions at the user interface and elements of the application program. On the other hand, the simulated, or "meta-" mouse, like the caret in SNAP-DRAGGING, affords a means of describing the behavior of the mouse using the mouse itself.

None of the systems considered above can induce the control structures inherent in a task. NODDY [Andreae 85], a system for incrementally programming a robot, performs far

more sophisticated inference. The initial model of a program is simply the first trace of its execution. Each subsequent trace is matched to the previous model, which is generalized to cover it. A clever algorithm for matching program structures identifies loops and conditional branches. NODDY identifies constants and variables by inducing functional relationships between the parameters of actions; thus, even implicit constants can be generated. NODDY'S function induction algorithm is powerful — but not powerful enough to avoid searching the vast combinatorial space of functions that includes interesting graphical relationships. Another factor that makes NODDY less appropriate to graphics programming is the care that the programmer / teacher must exercise in ordering lessons.

LOGO and NODDY have a well-defined programming metaphor — a robotic pupil led through a procedure. A recent pilot system, the Office Clerk [MacDonald 87], uses the teaching metaphor to make program annotation easier and more natural. Just as a teacher would tell a pupil to pay attention to some attribute of an object, the programmer can direct the Office Clerk, represented as a face that moves under control of the mouse, to a particular data field in an application's form-filling dialogue. The Office Clerk demonstrates the next stage in "Metamouse" programming.

Several important points emerge from the work described above. First, systems that employ inference have achieved robustness by requiring the programmer to annotate her examples or at least present them in a carefully chosen order suited to the learning algorithm. Symbolic annotation seems effective and convenient, when the annotation consists of labeling points, but is awkward or arcane when more complex features must be described. Second, the use of an attention device has emerged as a technique for non-symbolic annotation; it isolates features and localizes the context of inferences. A point locator is not sufficient for drafting problems however, since features of interest may be spatially distributed. Third, the teaching metaphor, by presenting an intuitive model of the learning system, permits increased use of inference and is readily embodied in an attention device such as Metamouse. Finally, the literature shows a lack of reported empirical studies of the potential or actual users of these innovative systems. SMALLSTAR for example was tested by a few people at Xerox PARC who were reportedly quite impressed with it [Halbert 84]; its usefulness to office workers has not been established. User characteristics ought to receive more detailed consideration in the design of end-user programming systems, and implementations should be followed by extensive testing. Of course, it is not quite fair to demand this of pioneering investigations.

To date, no system for programming graphics has combined a rich model of geometric constraints, a suitable teaching metaphor, and inductive inference. Nonetheless, results from systems in the literature suggest, tantalizingly, that such a system is not far out of reach.

1.4 Outline of Thesis

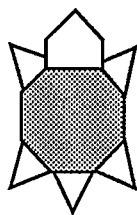
This thesis addresses the problem of end-user programming in a direct-manipulation environment — specifically the problem of inducing procedures from execution traces of graphical tasks. It proposes a system whose design takes into consideration the human factors requirements of the user, the difficulties inherent in a rich graphical task domain, and the technical limitations on a practical system for inducing programs. The crux of the thesis is stated below, followed by an outline:

End-user programming for computer graphics should be graphical and demonstrative. A practical programming system must limit the complexity of functional components to be induced, by analyzing traces and by requiring that the user employ graphically constructive techniques to satisfy simple felicity conditions. These requirements can be met by intensive interaction between user and learning system through a device, the Metamouse, that embodies the teaching metaphor and thereby enforces and helps the user to satisfy the felicity conditions.

Chapter 1 has defined the project's goal, emphasizing pragmatic and human factors considerations. Previous work has provided insights and techniques, but no method by which casual users can teach drawing procedures without symbolic annotation. This chapter concludes with an introduction to the Metamouse and a worked example.

Chapter 2 presents empirical and theoretical studies of the problem of teaching graphical tasks. It proposes four aspects of a teaching protocol, called "felicity conditions," that the user must satisfy if the system is to learn from her. In response to the difficulty this causes the user, interaction techniques are proposed that serve as underlying principles for the design of a programming system.

Chapter 3 describes the system's design in detail. The primary components are a model of the graphics world in which it operates, the Metamouse interaction device, and a method of inducing procedures.



My name is **Basil**, and as you can see I'm a turtle. I'm here to help you draw. You teach me repetitive and finicky tasks like evenly spacing out a row of boxes. I learn by acting as your apprentice: I follow you around till I think I know what you'll do next, then I pitch in and do it for you. If I guessed wrong, you give me a gentle tap so I'll undo it and wait for you to show me what's right. I'm eager, but don't worry — I only predict after I see you do something you've already taught me.

I can draw lines and boxes and drag them by their handles by grasping with my jaws. You can teach me to make tools for a task; for example, build a staircase of boxes using a diagonal line. When done with a tool, delete it so I learn to clean up after a job.

Although I have a good memory, I don't see too well. Because I crawl around a video screen I see things edge on, which makes it hard to spot patterns. Instead, I work mainly by feel. I remember how things fit together, which parts — such as handles and line segments — are connected. Building the staircase mentioned above, I can learn to copy a step by tracing over it: all the handles of the new step then mate with the old, and the new step is moved until it sits on top of the old one, offset horizontally.

I'm touch-sensitive at my snout and I sense contact between the object I'm grasping and anything else it touches. If I have to find, say a box, I set off in the general direction you've taught me (up, down, left or right) until I bump into one. The box doesn't have to be dead ahead — I'm not stone blind. But if you want me to be more selective, give me a tool to carry and teach me to move until it touches something.

Now, this is very important. I can't learn directly how things should not touch — I mean how they should be separated. Instead you should give me tools to separate them. Say you're drawing an arch and want the columns an inch apart. Draw a one-inch horizontal line and put the columns at either end of it.

If you move me to some point and I don't sense something touching, I'll ask whether I should always move there, or always let you show me where to move. This is helpful when making tools that measure, since some are constant and some need to be varied. If you answer no to both options, I'll ask you to make a tool that reaches to this point.

When you want to teach me, choose "Time for a lesson!" from the Basil menu, and "End of lesson" when you're done. To interrupt the lesson for something else, like working out a method before showing me, just choose "Take a nap" and then "Wake up, Basil!" when you're ready.

As soon as I can predict what to do, I'll take over the task, but I'm always ready to learn something new. And as I've said, just tap me if you don't like what I've done.

Well that's all I have to say for myself. Hope you enjoy teaching me!

Figure 1.7 Description of Metamouse given to teachers

Chapter 4 describes the system actually implemented. Certain elements of the “ideal” system were left for future work. The implementation nonetheless permits investigation of the thesis.

Chapter 5 presents three empirical evaluations conducted on the system. The intuitive appeal of Metamouse and its model of graphics were tested using a questionnaire given to potential users. The performance of the learning system was tested on several graphical tasks.

Chapter 6 evaluates the thesis according to all the analytic and empirical evidence gathered. The empirical studies are criticized. Modifications to the system and further research projects are suggested.

1.5 Introducing Metamouse

The Metamouse, which embodies the teaching metaphor, is a graphical turtle like that used in the LOGO system. Its behavior depends on the algorithm for inducing procedures and on its model of graphical constraints. The Metamouse introduces itself to potential teachers through a short autobiography, given in Figure 1.7.

1.6 Box-to-Line—a Worked Example

Consider the “box-to-line” procedure as taught by example (Figure 1.8). The teacher leads Basil through a trace of the task. Basil — to be precise, the learning system — starts predicting actions as soon as it observes the teacher repeat an action already learned (provided the predicted actions are appropriate and can be accomplished). The system expects the parameters of actions to be constant, input by the user at run-time, or constrained by tactile events (such as a corner coming into contact with the guide-line). It generalizes constraints and induces program structure.

When the teacher places the guide-line’s two end-points (Figure 1.8b), Basil observes the absence of a contact, classifies the event as underconstrained and interrupts to ask (through a dialogue box, see Figure 1.8c) whether the location is constant or a run-time input. The teacher indicates that both points are to be specified at run-time.

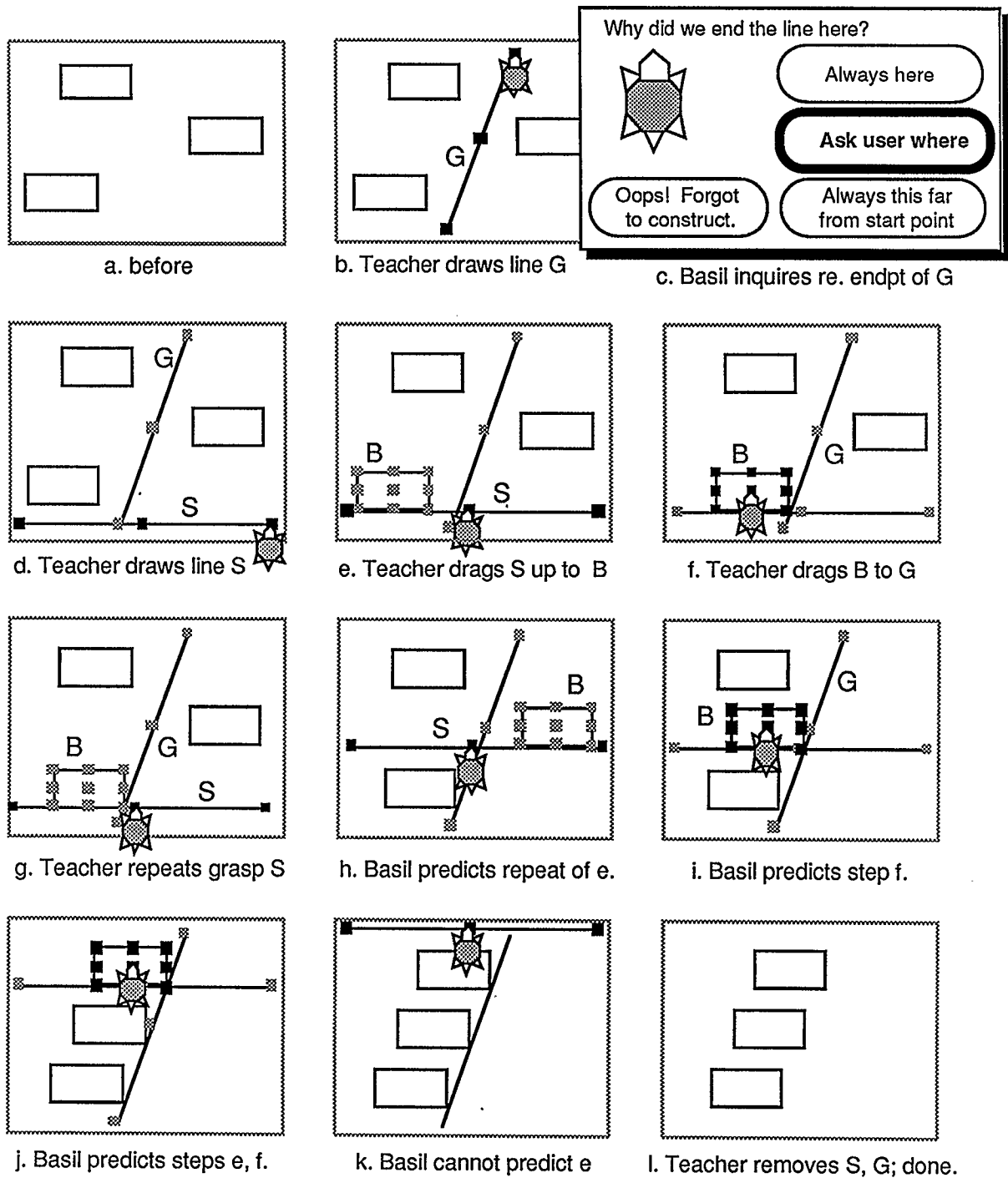


Figure 1.8 Teaching Metamouse a trace of “box-to-line”

In general the selection of objects and the iteration of action sequences depend on any number of properties of objects or situations. Thus they should be ordered and conditioned on events that Basil can sense by touch. A horizontal sweep-line serves this purpose, and also constrains the boxes' path of translation. The teacher draws it near the bottom of the screen (Figure 1.8d) and indicates (through a dialogue box) that its initial placement is constant. She then grabs the sweep-line at its mid-point handle and moves it upwards until it touches the bottom edge of some box (Figure 1.8e). This contact is the condition on which a box is selected for translation. When the line is swept past the last remaining box, Basil notes that the sweep-until-contact action will fail. She interprets this as the reason for terminating the loop.

Observe that this program achieves alignment through constraint-governed translation, where the constraint is a visible contact. A suitable description, say "lower left corner of box in grasp is coincident with some point on guide-line," is invariant over iteration on the input set of boxes. The learning system distinguishes this contact event and induces its invariance.

When the sweep-line touches the first box, the teacher grasps and moves it rightward until its lower right corner touches the guide-line, its bottom edge still on the sweep-line (Figure 1.8f). The teacher then re-grasps the sweep-line (Figure 1.8g). Basil has seen this action before; consequently he conjectures a loop and predicts the sweep upwards to the next box that is to follow (Figure 1.8h). Up to this point Basil has been following the teacher like a studious apprentice; now he takes the lead and performs his predicted action. The teacher does not object, so Basil continues executing the loop. The second box, however, must be moved to the left. Basil is biased towards easily generalizing directions of movement, so this does not faze him; he moves the box in order to achieve the same type of contact observed on the first iteration (Figure 1.8i). The teacher accepts this action as well; Basil has now learned the body of the loop and operates on the next box by himself (Figure 1.8j).

After processing the third and final box, Basil recognizes that he cannot complete the action of moving the sweep-line as he has learned it (Figure 1.8k). Hence he terminates the loop and calls upon the teacher to show him what to do. At this point, she removes the sweep- and guide-lines (Figure 1.8l) and then announces that the lesson is over.

Chapter 2

Meeting the Felicity Conditions

What human factors need to be considered in designing a system for inducing graphical procedures? How can the learning system utilize knowledge of human visual perception and of drawing practice? This chapter begins to answer these questions.

It begins with a description of a preliminary user study conducted to identify important features of human performance in drawing. Although drawing is inherently procedural, the results argue against inducing procedures by passive observation: the typical execution trace leaves much of the computation unexpressed and is rife with noise. A strong model of drawing practice does not emerge, but conditions are identified under which a learning system should direct its own instruction by asking for explanations and generating its own examples.

Section 2.2 examines more rigorous empirical studies of drawing on paper. These reveal two basic elements of drawing practice: first, people decompose their procedures hierarchically, that is, they use readily identifiable subroutines; and second, they carefully order the production of sub-pictures, first drawing the part that constrains and then attaching other parts to it. Comparison with such studies suggests that people are rather more systematic when not using a computer. Nonetheless the basic principles are seen at work in computer drawing. It follows that non-systematic actions in traces can be isolated, and that the learning system should observe contact events in order to model constraints.

A consideration of some aspects of visual perception leads to a better understanding of computation not expressed in traces. Much of it is particularly costly to perform, let alone induce. Fortunately, key operations have strong analogies to the tools of geometric construction. It is concluded that the learning system can reasonably expect and require the user to express constraints by constructing them.

Finally, Section 2.3 summarizes the rules of protocol that govern the teaching process. These “felicity conditions” are difficult to satisfy, but a primitive model of drawing practice generates strategies the learning system can use to help the teacher. The system uses a metaphor to explain its computational limitations, expects actions to arise from and result in

contact events, is able to ignore the teacher, predicts actions as early as possible, and learns incrementally.

2.1 User Study

The work of other researchers ([Bier 86], [Borning 86], [Fuller 86]) suggests a widespread belief that the users of drawing programs must annotate their procedures in order for a practical learning system to generate code. Having found no empirical studies on the practice of drawing with a computer, the author decided to perform a small study of his own. Rather than test an hypothesis, the study was designed to illustrate the use of graphical construction and to identify sources of noise in execution traces. It showed that people use constructions in their graphical procedures, but not predictably; some construct where others use visual inspection. The designers of a system that learned by passively observing the user at work could make only weak assumptions about her use of construction; they could not predict the complexity of functions the system must induce. Participants in the study also exhibited a significant number of errors, experiments and other extraneous actions during execution traces. These elements of noise make it impossible for a passive learning system to induce the structure of a program from execution traces alone.

A group of 10 subjects performed as many as 7 graphical tasks over a one-hour period, using MacDraw; all subjects were able to complete the first 5. Execution traces were recorded with a commercial programming-by-example system [Tempo 86]. The subjects ranged widely in experience: 3 had never used MacDraw; 3 had used it fewer than 5 times; 2 were occasional users; and 2 were regular users. Inexperienced users were introduced to the program in a preliminary training session. An observer / coach sat with each subject through the task session.

The remainder of this section presents the tasks, some of the observations, and the implications for graphical programming.

2.1.1 Tasks

The tasks were designed to meet several criteria. First, they should be “realistic,” *ie.* similar to the problems actual users of MacDraw must commonly solve. No case-book

was available, so problems were taken from the researcher's own experience with MacDraw. Second, they should gradually increase in difficulty because most subjects would be unfamiliar with the program. Third, some of the tasks should pose problems that strongly motivate the use of explicit, planned constructions to achieve constraints. The basic problems were altered and combined in order to meet all three criteria.

The instructions given to each subject are included as Appendix A.

For task 1, the subject was asked to reproduce a row of four objects illustrated on the instruction sheet, then interpose a cross so that its arms lay above the other objects. This task introduced the various primitive shapes and provided an opportunity to use MacDraw's automated transformation facilities (rotation and alignment) to satisfy two constraints between the stem and arms of the cross — that they be perpendicular, and that the arms be as wide as the stem is tall.

Task 2 required the user to practise centering one object on another, in three different situations: a circle inside a square; a pair of rectangles to make a Greek cross; and a circle touching all four sides of a square. The first two could be done using MacDraw's alignment command; the third required considerable ingenuity or patience to accomplish exactly.

For task 3, the subject was asked to transform a square into a close approximation of a rhombus; this transformation of one set of constraints into another could not be done using MacDraw's built-in operations but required the subject to invent a procedure.

Task 4 illustrated a sequence of increasing constraints; the subject drew a scalene triangle, then an isosceles, then a right isosceles, and finally an equilateral inscribed in a circle. This last sub-task proved particularly difficult since MacDraw does not facilitate arbitrary degrees of rotation.

Task 5 was presented in such a way as to encourage the use of constructive techniques. The subject was given two objects, A and B, and asked to translate B horizontally until its leftmost point lay at a distance of twice A's height from A's leftmost point.

Task 6, performed by about half the subjects, demonstrated a simple case of two degrees of constraint and inheritance across groups; the subject had to position squares at

the vertices, mid-points and center of a polygon's extents box. The use of MacDraw's alignment and grouping commands could make this procedure quite elegant.

Task 7, also performed by about half the subjects, presented a set of rules for positioning labels next to arrowheads depending on orientation; the subject then applied these rules to an actual case. The task illustrated the difficulty of positioning exactly with respect to distance — the use of MacDraw's grid introduced quantization errors — and also the difficulty of conceptualizing such *a priori* rules.

2.1.2 Observations

The user study produced a number of observations concerning the use of graphical construction and visual inspection, the variety and internal consistency of methods, and the occurrence of extraneous activity. No quantitative correlations were attempted; the study was not arranged to isolate variables, and in any case the sample of subjects was too small relative to the number of identifiable variables. As expected, subjects having prior experience with MacDraw used more of its "advanced" facilities (duplicating objects, rotation, alignment) in constructions. After tutoring most subjects used at least some of these, but only the more experienced (and one first-timer) used all three together.

Construction and Visual Inspection. Graphical construction, as opposed to visual inspection, is the use of tools to automate the processes of measurement and selection. MacDraw supplies many such tools as primitives: commands for alignment and rotation; a special key for constraining cursor movement to one axis; automated duplication of objects; and positioning grids. Although by no means adequate as a set of operators for a graphical language, they do illustrate the advantages of construction as a basis for programming.

Task 5, for example, requires that the subject take the height of A and use it as a horizontal measurement. This involves rotating a vertical measure, impossible without storing that information either in symbols (say, if the user measured in terms of grid coordinates) or a graphical form. One approach is to draw a vertical line the height of A and then rotate it to the horizontal; but if A is a complex polygon, exact measurement involves considerable work: the user must draw the line to approximate length, based at A's lowest point, then translate it horizontally to cut A's highest point, then move its top end down to that point. A far simpler and more accurate procedure is to duplicate A and then rotate it 90°; the width of the duplicate is used as the desired measure. Note that this

can be trivially learned — that is, without knowing about such things as top and bottom extremal points, nor having to observe contacts.

Only two of the subjects employed this particularly clever approach. Three others used a similar technique, drawing and then rotating an extents box around A. This entailed initial approximation, the fixing of one extreme and adjustment to find the other, as when using a vertical ruler, but avoided the danger of accidentally undoing the first alignment while moving into position to find the second extreme, since the box required no horizontal translation. One subject used MacDraw's "show sizes" mode to get A's top and bottom coordinates. The remaining three used the vertical ruler much as described above.

Subjects' performance of the measuring sub-task described above suggests that the use of construction requires considerable understanding of graphical tools. On the other hand, many useful constructive procedures were readily adopted by the majority of users, as in task 1, where all subjects but one constructed the arms of the cross by duplicating and rotating the stem. Performance over all tasks indicates that subjects tended to adopt straightforward, albeit sub-optimal constructions. It is not necessarily the case that a learning system would have more difficulty inducing a program that contains longer sequences of operations. The real problem with construction is that many subjects did not use it at all, but chose instead to measure by eye. In performing task 3 (constructing an approximate rhombus from a square) seven subjects positioned the offset vertices by visual inspection alone, locating the second vertex with respect to the first by adjusting it until opposite sides appeared parallel; one subject used a vertical ruler to assist with visualizing the horizontal offset; two subjects only used a fully constructive method (duplicate and horizontally offset the square, then connect the lower corners of the first square to the corresponding upper corners of the second).

The examples above also indicate the variety of methods employed. This suggests that only elementary models of human behavior are likely to be of use in designing a graphical programming system. Typical observations were: subjects used point-to-line contacts and line intersection to determine position (tasks 3, 4, 5); they used visual inspection for vertical and horizontal alignment (task 5); more experienced subjects preferred single operations of higher-degree constraint over a sequence of less constrained operations (tasks 3, 5, 6).

The variety of methods that might be employed to perform a given task and the apparent paucity of behaviors having a reliable standard interpretation imply that the learning system will have to be quite general. The use of visual inspection implies that it must be able to induce functions from input-output comparisons (that is, the positions of objects, or, more generally, the state of the display, before and after a sequence of operations). The search could begin with some obvious candidate relations such as alignment. The system may be able to model the individual user sufficiently well to predict the maximal complexity of such latent functions.

The observations made above can be operationalized as an hypothesis for the learning system: if the transformation of an object does not result in contacts with other objects, the teacher is using visual inspection.

Extraneous Activity. Subjects exhibited a great deal of inconsistent and extraneous activity over the course of executing a given task. The experimental situation likely exaggerated this problem, since they were unfamiliar with the tasks and several indicated that they felt some pressure to perform well. The “noisy” activity observed has been classified according to the subjects’ own verbal explanations (collected during and afterwards) of what they were doing at the time.

Missteps are those actions quickly retracted, such as vertically reflecting the duplicate stem (task 1) rather than rotating it. These occurred often, especially with less experienced subjects.

Experiments are extensive action sequences later retracted. When not performed on temporary data, a successful experiment is of course indistinguishable from a previously planned execution. Most subjects experimented with unfamiliar commands before trying them out (*eg.* alignment in task 1, reshaping a polygon in task 3); several also performed experiments to develop algorithms (there were many failed attempts to construct an equilateral triangle in task 4). Experiments may also be performed after completion of a task, to verify its success. One subject at the end of task 4 checked that his triangle was indeed equilateral by rotating it several times.

Non-systematic experimentation, called *fishing*, is seemingly useless activity that may lead to serendipity. For example, one subject toyed with circles while thinking about

equilateral triangles. Most of the novice and casual users would scan across the pull-down menus when stuck for an approach to their task.

Another variant on experimentation is *method drift*, the abandonment of an action sequence without retracting it or returning the task environment to its initial state. If the actions were incorrect, the subject might have to fix up the display before continuing. Method drift is hard to distinguish from an inelegant method (*ie.* one that unnecessarily treats situations as special cases) without asking the subject to explain her actions. Method drift was observed in task 3 (rhombus), where one subject who had measured the offset at the left using a rectangle suddenly duplicated the square and offset it using the rectangle. The subject explained that using the rectangle gave him the idea of using the square instead. In performing task 5 (2 x height) another subject initially measured off the distance with a pair of horizontal lines placed end-to-end; after several failed attempts to align the left extremes of the two polygons with their corresponding end-points, he switched to a pair of boxes, without removing the lines or returning the polygons to their original positions.

Bustle is apparently purposeless activity performed while the mind is otherwise engaged. Several subjects would occasionally move an object back and forth rhythmically before committing to its placement. Most bustle, such as toying with the mouse, does not affect the display at all.

Having identified some sources of noise, how can we apply this knowledge to the design of a learning system? Much of this “useless” activity appeared to help subjects accomplish their tasks; the learning system should not punish the user by failing to perform in the presence of noise. To cope with extraneous actions, it must be able to identify suspicious action sequences and remove them with surgical economy. Unfortunately, the user study uncovered no reliable symptoms of incipient noise (though the mouse did appear to cover more territory when a subject was not working systematically). This leaves only more expensive methods of analyzing traces *post hoc*: measuring and comparing the effect of action sequences.

For example, suppose sequence A is undone by sequence U — that is, U returns the display to a close approximation of its state prior to A — then clearly AU can be identified as a misstep, experiment or perhaps bustle. Since AU has no effect, it may be removed from a model of the procedure; sequences before and after will form a seamless join.

If an iterative sequence R^* (1 or more iterations) is followed by a different sequence D^* , and one iteration D can be shown to have the same effect as one R , then the system has strong evidence to suspect method drift. In this case it could remove D^* .

In general one cannot expect noisy sequences to be entirely undone or readily identified as equivalent to others. Thus the *post hoc* removal of noise, apart from obvious cases as above, seems intractable for the present. But a passive learner is inevitably exposed to noise. Clearly, the system should avoid it by selectively ignoring the teacher, or even prevent it by restricting or reducing the teacher's activity.

Summary of Results. In summary, two important lessons have been learned from the user study. First, people do indeed use graphical constructions, but often rely upon visual inspection. A passive learning system would therefore have to cope with under-specification of procedures by performing function induction. An active system could ask for an explanation when explicit construction is not used. Second, graphical procedures also contain a good deal of extraneous activity. A passive system would have to classify it. An active system could prevent it by guiding or pre-empting the teacher.

Function induction is difficult even in restricted and noise-free situations. The author has found no evidence of research on inducing functions from traces by casual users. Distinguishing extraneous actions also seems intractable. Therefore, an active learning system is the preferred choice. A further investigation of the drawing process sheds light on the methods it should employ to elicit reliable information.

2.2 Towards a Theory of Drawing

Cognitive scientists have a growing interest in drawing and have proposed some theoretical principles [van Sommers 84]. It is clear that human behavior, even in fairly simple tasks, is too variable, and thus the connection between action and intention too tenuous, to be modelled by an inductive system. Nonetheless, theory and experiments do suggest that human beings can communicate a graphical procedure to a rather naive system. Drawing is essentially procedural, and incrementally constructive: when drawing, people seek an orderly execution, and employ constructive techniques to assist themselves. Moreover, thinking graphically seems to involve construction. It follows that graphical programming

need not require symbolic annotation as previous systems have — or at any rate that the annotation can be purely graphical.

2.2.1 Empirical Studies

It is obvious that drawing is a procedural activity. The user study did not show whether it is systematic and therefore amenable to representation by programs. That drawing with pencil on paper is quite systematic is in fact well supported by experiments [van Sommers 84]. People who draw, even young children, optimize their procedures to reduce the number and complexity of mechanical and cognitive operations. They tend to make pencil strokes in the direction of least resistance, but if they can reduce cognitive load at the expense of mechanical effort, they will do so. Thus, if one line is to branch off from another into empty space, it is almost always drawn from the stronger constraint (the point of contact with the parent line) to the weaker, even against the preferred direction of stroke.

Contact constraints strongly influence the order of execution: drawing appears to be primarily a process of accretion. A pyramid of rectangles is normally drawn bottom up, so that the width of successive levels is constrained by those below. In experiments on reconstructing shapes from memory, van Sommers found that visual memory seems to record only scattered, local constraints (a result that would not have surprised Escher). Nearly all subjects who attempted to reconstruct a triskele, for example, could draw some of the joints, but almost none could reproduce the whole to a good approximation. Often a subject would start out well, then suddenly become utterly confused.

Hierarchical decomposition, that is, the use of subprocedures, is another tactic employed to reduce cognitive load. In drawing a hierarchical form — a tree, for example — people typically follow one of two plans: order execution on the basis of similarity by drawing all objects at each level of the hierarchy in turn (for example, the trunk, then the boughs, smaller branches, twigs); or execute according to relations among sub-pictures, such as connectivity and adjacency, by working down the hierarchy (draw one bough, one branch, one twig, then repeat). Van Sommers found that such subroutines are quite systematic: they are commonly repeated without any intervening action; they preserve their internal order within a single drawing and often when a similar form is produced on another occasion.

In summary, three rules of optimization govern drawing: choose stroke direction to minimize mechanical effort; work from stronger constraints to weaker; decompose tasks hierarchically as appropriate to cognitive / mechanical trade-off. Note, however, that van Sommers studied drawing with a pencil on paper; the ergonomics of drawing with a computer must be quite different. Although the differences have yet to be studied empirically, it is possible to speculate that stroke direction is less important, and that the ease of editing makes contact constraints less significant. Supposing, however, that the three rules still apply, the question remains as to how they can be used by an inductive learning system. Clearly, these rules do not determine behavior, and therefore cannot interpret or predict it. On the other hand, behavior should be sufficiently well regulated that the system can expect to observe hierarchies of action sequences, in which most actions are conditional upon contact constraints.

2.2.2 Phenomenology and Construction

In the graphical domain a learning system with a numerical representation of objects is at a peculiar disadvantage relative to its human teacher, who has a specialized image processing system connected to a large knowledge base and facilities for planning. Human beings have in addition a certain computational enrichment owing to their mobility in 3-space. Thus the computer labors to find complex numerical relationships that the teacher perceives instantaneously. This “computation gap” can be characterized in terms of differences in the apparent complexity of geometric problems. Its most serious impact is on function induction, which is vastly more difficult for the computer. Moreover, the teacher may be unable to 1) recognize situations in which the system would have difficulty; 2) identify implicit computations that need expressing; or 3) find a means of expressing them.

One approach to the problem is to give the computer visual processing facilities comparable to a human's. This is interesting — pioneering work has been done on systems analogous to the retina, for example [Marr 79], [Kienker 86] — but the research is just beginning. A practical alternative is to require the teacher to give more information and provide her with a powerful language: the constructive methods that form the intuitive basis of computational geometry. The intuitive appeal and expressive power of these methods are evident. They are easily represented in both graphical and numerical terms, as in [Fuller 86].

[Freudenthal 67] describes a phenomenology of geometry whose three basic concepts — order, measure and classification — yield powerful problem-solving methods.

Order. Order is a means of abstracting a structure from a set of objects. It may encompass one dimension (a total order) or several (partial order). Humans often perceive order in space as temporal order on the focus of attention. For example, relative distance can be discovered by mapping perceptual events onto a time-line. A linear order is found by sweeping one's gaze across a scene; a rotational (cyclic) order, by sweeping around some imaginary center of gravity; a partial order, by two orthogonal sweeps. Nature has cunningly equipped human beings with superb mechanisms to discriminate order — consider our prodigious visual acuity for detecting misalignment. Other kinds of order, such as relative size, seem in general to be processed less efficiently, although familiar techniques of imaginary visualization permit the use of sweep-selection for some problems: for example, to compare the heights of two objects one can imagine them side by side; to compare volumes, imagine trying to fit one inside the other.

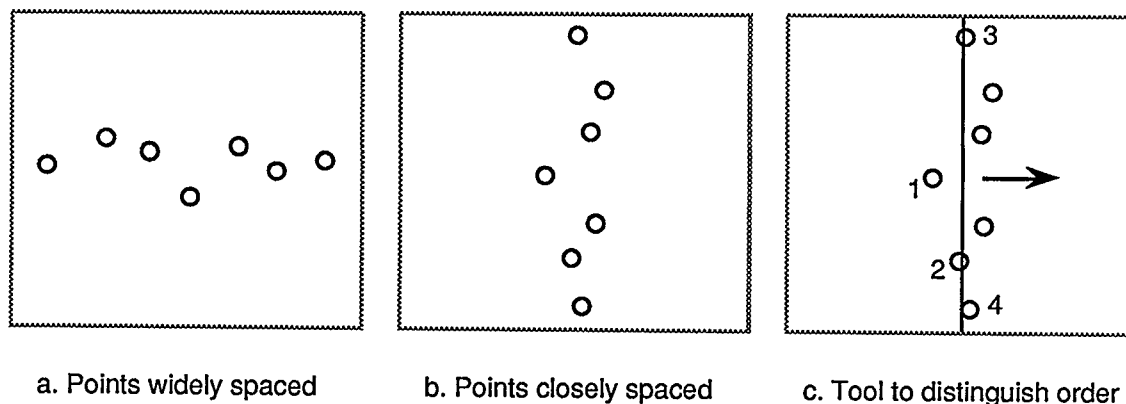


Figure 2.1 Left-to-right ordering of points

Human vision exploits a high degree of computational parallelism. In addition, the very mechanics of vision give human beings a significant advantage over von Neumann computers. For the latter, naive sorting algorithms require $O(n^2)$ binary comparisons; clever ones, $O(n \log n)$. Consider the task of labelling a set of points in order from left to right (see Figure 2.1). If they are spaced fairly far apart along the horizontal axis but quite closely in the vertical (Figure 2.1a), the sorting task is trivial for a human being, who merely sweeps across the set and performs no explicit comparisons. On the other hand, suppose the points are fairly close horizontally but quite scattered vertically (Figure 2.1b);

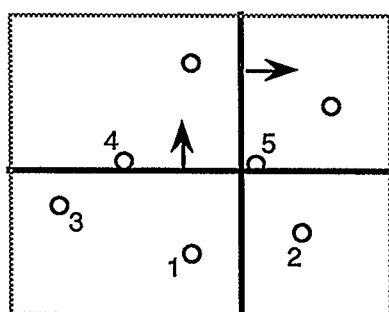
in this case, the human being must do $O(n^2)$ comparisons. But if she uses a windowing tool to focus her visual attention, she can reduce or even trivialize the sorting problem (Figure 2.1c). Now, suppose that this task is to be taught to a computer drafting program that will begin predicting actions as soon as it has induced a candidate function; the difference between human and computer becomes important. The sorting algorithm involves a predictable amount of work for the computer, a highly variable amount for the human teacher. If she does not understand this difference, she may be disappointed that her pupil does not induce the sort during presentation of the easy case more quickly; on the other hand, she may be quite surprised to discover that the computer has no more difficulty with the complicated case, and hence ascribe to it far greater intelligence than it possesses.

This barrier of mutual misunderstanding is revealed to be even greater when we consider the ease with which a person can revamp her representation of visual space to accommodate different references of order. Suppose she wanted to sort objects with respected to an axis tilted at 60° ; she need only cock her head and proceed as usual. Her pupil, the computer, could take the same approach, by rotating the coordinate system (applied as an inverse rotation of the displayed objects); but first it must guess that this pre-processing will help it induce some function, and then find a suitable angle. A learning system capable of such conjectures would have to be highly specialized or else search an enormous space of models. The teacher, for whom the difference between this ordering and the horizontal may seem marginal, would not appreciate the difficulty.

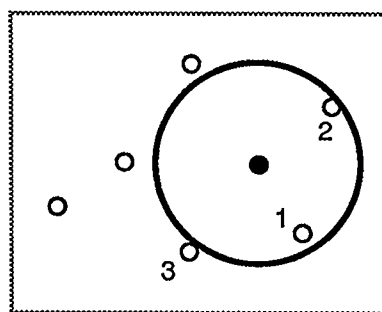
The problem of inferring relations of order from a large space of candidate relations is not solved by observing traces of the user selecting objects in order; as shown above, the space of candidate models for a sequence of invariant binary relations is still too large. The user can, however, express many relations of order through an intuitively appealing device — a construction of one or more *sweep-lines*, as in Figure 2.2. These are used in many algorithms studied in computational geometry [Preparata 85]. A single sweep-line expresses a total order in one dimension (see Figure 2.1); an ordering along an arbitrary axis is easily expressed by initially rotating the sweep-line appropriately. A rotational sweep can discover a rotational order. A pair of sweep-lines can find a partial order in two dimensions.

Any ordering distinguishes certain points or thresholds, such as the mid- and end-points of a line segment, the vertices of a polygon, or the boundary between half-planes.

Interactive drawing programs typically designate such points as “handles” by which the user can manipulate the spatial relations between objects and parts of objects. A sweep-line’s initial contact with an object will be at one of its extremal points (a point on its convex hull), which corresponds to one of its handles. Hence a great many useful orderings can be observed by a system that distinguishes contacts between some part of a line segment and a handle.



a. Two-dimensional sweep



b. “Sweep-circle” to order distance

Figure 2.2 Use of sweeping methods to distinguish order

Measure. An object’s position within an unrealized ordering is given by its measure; for example, its distance from the origin. Measures used commonly in describing space are distance and angle. Apart from certain relative measures, such as bisection and trisection, humans are not particularly well equipped to measure without the use of tools; after all, absolute measures are cultural institutions. It is not surprising, therefore, that constructive techniques for measuring are in wide use and have been thoroughly studied [Breidenbach 67]. One would expect the user of a drawing program to construct exact measures by creating and manipulating graphical rulers; the parameters of their transformation would express the measure. In practice, however, no absolute measure can be expressed unambiguously without resort to symbolic annotation. The user can employ devices external and hence unknown to the learning system, such as a tape measure held up to the display. If the user draws a ruler line on the screen, it is not clear which parameters of the line are relevant — its length, its angle from the horizontal, or the exact positions of its end-points. If a drawing program, such as MacDraw, provides “passive” rulers to which the user refers visually, the learning system cannot tell when they are in use unless it can observe the movement of the user’s eyes. Clearly, the most feasible approach for getting

the parameters of measurement is to give the user a tool for selecting them: for example, a potentiometer in the form of a ruler or protractor.

Most measures employed in drawings are either relative (as in the “2 x height” task) or estimated (as in the “spacing” task). Constructive techniques are quite adequate for these, since the parameters need not be isolated: for example, it does not matter whether the gap line used in the spacing task is perfectly horizontal. If the learning system attends to contacts made between handles and line segments, and generalizes position within the latter, then it can correctly model the use of ruler lines. Should the contact between the ruler’s handle and some line segment need to be constrained to some particular point within the latter, that point can be constructed, perhaps with a second ruler.

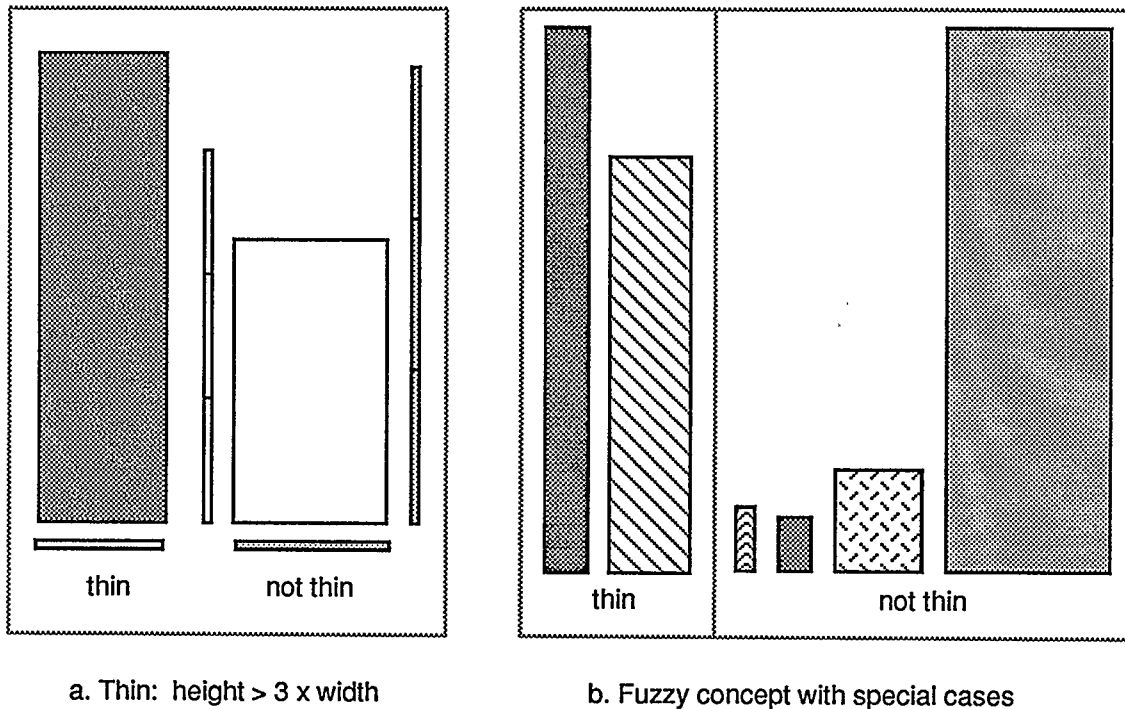


Figure 2.3 Two concepts of “thin box”

Classification. Many problems involve distinguishing certain objects as equivalent in some respect, so that they receive similar treatment. Equivalence is often determined by a group of operations that transform one object into another [Freudenthal 67]. Common experience suggests that human beings tend to think in terms of transformational constructions. For example, to decide whether two polygons are the same shape, one

might imagine superimposing one upon the other by rotating, translating and scaling it. Classification problems subsume those of order (*eg.* identify the vertices of a convex hull) and measure (*eg.* identify points at equal distance from some reference). Thus the difficulties and techniques germane to these problems apply here as well.

Classification is made even more difficult to communicate because often class definitions are quite subtle. Consider two concepts of “thin box”, illustrated in Figure 2.3. The first case is straightforward; the height is at least 3 times the width. The second, however, would be much more difficult to induce; boxes more than 2 cm wide are not thin, regardless of their height, nor are boxes less than 1 cm tall, regardless of their width. It is unlikely that a human being would be so precise in her definition as this. The constructions used in ordering and measuring can be brought to bear, if the user is able to see how. For example, to select squares one can employ a sweep-line that moves horizontally but is angled at 45°; the line will contact a square at two corners. Many other problems, such as “thin boxes,” are apt to be unprogrammable because the learning system would never find the concept by search, and the constructive technique is elusive.

2.3 Felicity Conditions

An alternative way of modeling the interaction between user and programming system is to specify the conditions under which procedural knowledge is most efficiently transferred. Given that the user is not a trained programmer, the transfer of knowledge is best understood as teaching [MacDonald 87]. The onus is on the teacher to present knowledge in the most readily assimilated form. The teacher has an approximate, evolving model of the system’s capabilities — a far weaker model than does a programmer. Since she does not know exactly how the system will interpret each component of a lesson, the teacher follows a communication protocol, a set of “felicity conditions” whose primary purpose is to limit the range of possible interpretations her pupil need consider [van Lehn 83]. The transfer of procedural knowledge has four basic felicity conditions; van Lehn identified these by modeling the teaching of arithmetic to schoolchildren. Similar conditions apply when the pupil is a computer. The teacher in this case, however, is an amateur who cannot be expected to satisfy the felicity conditions, especially when her pupil is so different from herself (see §2.2). The remainder of this section describes the felicity conditions and techniques by which the learning system actually helps the teacher satisfy them.

2.3.1 Correctness

The fundamental felicity condition is that lesson materials be correct. From the teacher's point of view, correct data are consistent with the model (procedure, concept) being taught. From the learning system's point of view, they are consistent with some model it could learn — a much weaker criterion. The system can suspect data that disagree with its current model, but must be prepared to alter that model to accommodate them. On the assumption of correctness, the learning system need only consider those models that cover all the data and may ignore the vastly larger number that cover only some. Moreover, the system itself can be simpler: it need not analyze data for likelihood of error, it need not use forgetting as a means of pruning out bad data.

The user study suggested that errors are bound to occur in the teaching of graphical procedures. Since the space of models (graphical procedures) is practically infinite, the learning system cannot in general identify these errors. The computer's perfect memory is coupled with perfect forgetting. If the teacher recognizes her mistake, she can delete it from the lesson by retracting her action trace; the learning system should provide an undo facility for this purpose.

If the teacher does not recognize her mistake, or does not properly retract it, then the program will contain a bug. To permit the removal of bugs without requiring that the teacher examine a symbolic representation of the program, the system should combine editing and execution; that is, at any point during execution, the user should be able to intervene and begin teaching (this also facilitates incremental development). An erroneous action could be excised upon the teacher's instruction; but the teacher may be mistaken that the action is incorrect — perhaps the conditions under which it is performed need to be specialized. It is therefore advisable not to eliminate a so-called bug altogether but to learn alternative actions and limit activation of the bug by specializing its preconditions or by reducing its priority amongst the alternatives.

Perhaps the best defense against the teacher's mistakes is to prevent them by minimizing her activity during the lesson. To achieve this, the learning system could start predicting actions as soon as possible. As soon as it observes the teacher do some recollected action α and the current state of the drawing permits some action β that has been seen to follow α , the system could perform β on its own initiative. It must be able to retract β should the teacher reject the prediction. Thus the learning system alternates

between observation and performance. Prediction structures the interaction: success initiates performance, failure initiates observation.

2.3.2 Show Work

The show work felicity condition requires that the teacher demonstrate an execution of the procedure to be taught. This eliminates the need to induce procedures from presentations of their inputs and outputs. Such induction is practically impossible for a number of reasons: the search space is infinite and not usefully ordered; the identification of input and output objects, often a useful clue to function inducers, may be lost due to transformations; data almost never sufficiently constrain the number of equivalence classes of procedures that account for them. An execution trace limits candidate models to those that could have generated it. A rote-learning system, such as the emacs macro facility [Stallman 81], could model the procedure well enough to execute on sufficiently similar data. If the learning system models the steps of the trace in terms of cause and effect, such as the (precondition, action, postcondition) tuples used in the STRIPS planning system [Fikes 71], then it can derive the conditions that govern loops and branches, and thereby produce a more general model of the procedure.

Consider for example the task of finding a convex hull. If the system does not observe its construction, but rather sees as input only a set of points and as output the same set with certain points connected, then some very useful information is unavailable to it. The relation of each vertex to the rest of the set — that it lies on the boundary of a half-plane containing all the other points — must be induced from a large range of possibilities, and is made even more difficult by the computer's perceptual disabilities, as noted above. The cyclic order of the hull's vertices, which could be used as the basis of a simple derivation — a sequence of pairwise relations — must be induced from the transitive closure of their connectivity. A procedure for finding the convex hull is easily derived from a constructive demonstration (using the algorithm given in Chapter 1) if the learning system's attention is focussed upon the events that bound the successive transformations of the rotational sweep-line.

The show work felicity condition is well suited to the users of a drawing system. In fact, it minimizes the effort of teaching because it joins teaching to the accomplishment of the task at hand — an efficient practice employed since time immemorial in workshops,

where a master would show her apprentices how to solve problems. On the other hand, show work implies “no invisible objects” and “minimal activity” — two felicity conditions that are much more difficult to satisfy.

2.3.3 No Invisible Objects

Invisible objects are parameters of actions or conditions that the teacher does not expressly describe. The no invisible objects condition requires that the teacher make them explicit. The pupil must see them to know that they are in use and what values they may assume. The range of relevant graphical attributes and spatial relations is combinatorially explosive over the number of objects displayed; a visible construction of the relevant relation can eliminate search entirely. Constructing “invisible” objects is just a matter of showing work at a finer granularity of decision-making and reduces the space of induction in the same way.

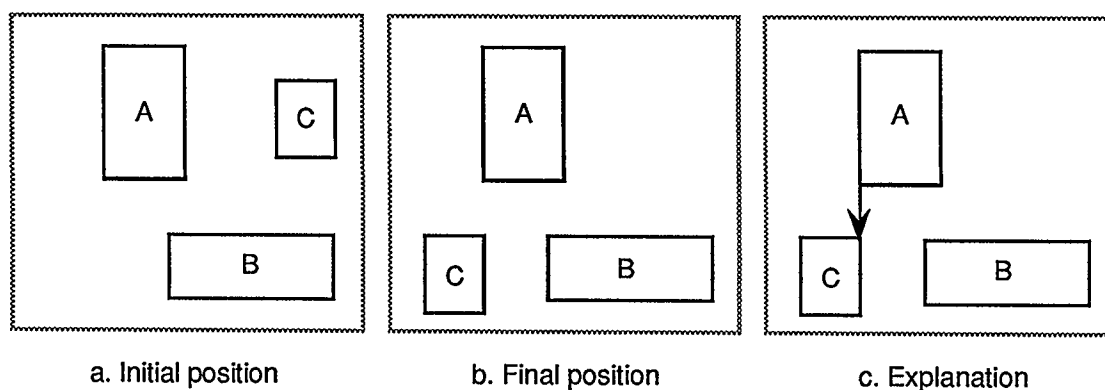


Figure 2.4 Spatial relation defined by an “invisible object”

For example, suppose the teacher moved box C as shown in Figure 2.4. It is not immediately apparent why C is moved to that position — whether the distance moved or the destination point is a constant, or whether some spatial relation governs the translation. If the latter, then the system must find a sufficiently “obvious” relation that would be safe to assume: clearly, two obvious relations obtain here — are both meant to, or just one of them? Of course, the best way to answer these questions is to ask the teacher. In that case, she would do what she should have done to begin with — point out the relevant parameters, in this case, alignment of C with the left of A and a constant vertical distance,

which only coincidentally aligns it with B. A simple graphical technique to express this condition is to draw a spacer line from A's lower-left corner.

Observations from the user study support the contention that invisible measurement objects are used so efficiently that knowing when and how to express them can be difficult. Most commonly used in drawing are alignments and distances, both well supported by visual inspection. It follows that the onus of enforcing this felicity condition falls upon the learning system: it should not attempt to induce the parameters of a transformation, but rather deduce them from visible, constructed relationships. This means that the system will look for tactile relations between the object transformed and other objects: thus it observes the contact between the upper-right corner of C and the lower end-point of the spacer line. If no contact is observed, as when setting the end-point of the spacer line, the system must ask the user to provide a construction or verify that the selected position is a constant or *ad hoc* input.

2.3.4 Minimal Activity

The minimal activity condition requires that the teacher show the pupil only those actions that would be generated by the procedure. In its strictest form it bans all irrelevant activity, such as experiments and doodling, and implies uniformity of method over iterations within a single trace or among multiple traces. This limits the candidate models to those that would generate the actions demonstrated in execution traces, eliminating models that generate other actions or sequences having the same effect. Uniformity allows the learning system to classify actions as novel without having to consider equivalence or commutativity. The banning of irrelevant activity prevents the break-up of sequences; they can be recognized more reliably when inducing loops and branches. The learning system need not know how to determine that a sequence of actions, such as an experiment, has no effect and may be ignored. Irrelevant activity may also corrupt the display and thus violate the correctness condition.

The user study indicates that minimal activity is very difficult to satisfy. Methods evolve as they are attempted; users have to conduct experiments; often they rearrange the display to help them visualize relationships better, to make situations more general, or simply to increase their comfort by making the layout more aesthetically appealing; and sometimes, while thinking, they repeatedly perform some useless action. But if the

learning system is not able to recognize irrelevant and non-uniform activity, how can it enforce this felicity condition?

One approach to the problem is to accept this form of noise — to learn procedures that are less general, due to non-uniformity; that contain bugs, due to irrelevant activity. Generalized loops and branches may well evolve and effectively supplant special cases as the teacher provides more examples. Useless actions should be quite harmless (provided that they do not corrupt the display). How well the system achieves desired behavior with a model corrupted in these ways depends in part upon its representation of actions. Using the cause-effect model (proposed in response to “show work”) and preferring actions more recently learned or more frequently observed, it should strongly favour uniform and useful code.

The cause-effect model provides a front-line defense against misleading activity. If the teacher launches into an experiment, her first actions will likely not follow from constructed conditions; the system will ask for these and thereby remind the teacher that she should not be teaching right now. To prevent the system’s observing such actions, the teacher could simply put it to sleep — luckily, computer systems don’t forget what they’ve been doing. Even more useful than this, however, is the ability to predict actions from their causes. Using prediction, the learning system can supply uniformity where the teacher might not: by matching a new action to one it has seen before, the system can conjecture a loop and predict the action sequence within it; similarly, the actions of a new trace may be predicted from the procedure as already learned, so that the teacher need only perform novel actions. Prediction is perhaps the best way to help the teacher satisfy minimal activity; and has the added benefit of demonstrating the pupil’s understanding of the procedure. Moreover, it provides that the computer will take over performance of the task as soon as possible — a welcome reward for the user’s effort of teaching.

2.4 Design Principles

Specifying the felicity conditions has provided a sufficiently well defined model of the interaction between user and graphical programming system to discover some useful principles of design.

First, the learning system must be active. The user study showed that a passive system will not do. The felicity conditions are too difficult for average users to satisfy. The system must question the user and help her maintain the conditions of instructibility. The other design principles prescribe the techniques used to accomplish this.

The learning system should represent itself and its computational model in metaphorical terms through an attention device. This device should convey the system's perceptual limitations by demonstrating its current awareness.

The learning system should be strongly biased towards geometric construction. It should model graphical actions in terms of cause and effect, that is, as (precondition, action, postcondition) tuples. Conditions should be contacts between the parts of geometric (*ie.* graphical) objects. Since some actions cannot be constructed, it must also permit numerical parameters. If an action is not constructed, the system should question the user as to the reason.

The learning system should attempt to predict the user's actions whenever possible. As soon as a user action matches one it has already seen, it should conjecture iteration and predict the subsequent action. Since predictions may be incorrect, the system should respond to the user's disagreement, predict only actions it could retract, and be able to make all alternative predictions.

The learning system should be able to ignore the user, but only at the user's request. In the event that the user does not warn the learning system of her extraneous activity, the system should be relatively tolerant of the noise. This is achieved by allowing any number of alternative actions to be available at each point of the execution; that is, the run-time user may always reject an action and supply one of her own, from which the system may predict subsequent actions.

The next chapter applies these principles to the design of a "Metamouse" programming system.

Chapter 3

Metamouse

The design principles developed in Chapter 2 are applied in the design of a prototype system for graphical programming. The design focuses upon an interaction device that embodies the teaching metaphor. The drawing program, knowledge base and learning system are described.

Chapter 2 established that the desired graphical programming system is really a learning system with some stringent felicity conditions. To help the user/teacher satisfy them, the system requires a metaphorical pupil (*cf.* [MacDonald 87]), which is conceptualized more precisely as an apprentice. The apprentice assists the teaching process by focussing the teacher's attention upon itself, by refusing to accept input (actions) it cannot justify in terms of its own model of the drawing world, and by predicting actions in new contexts. It follows that the apprentice must have a body and behaviors the teacher can evoke, sensors and an internal representation of the world in order to investigate justification conditions, a memory with generalization capabilities, and a model of program structure. The apprentice's "psychology" should be simple enough for the teacher to predict or at least understand its behavior after a brief period of familiarization.

The prototype system for this project is illustrated in Figure 3.1. It consists of a simple drawing program closely coupled to a learning system that induces variables, isolates important constraints, constructs a procedure in the form of a directed graph of actions, and predicts actions whose parameters are set by a constraint solver. The learning system's user interface is a metaphorical apprentice. This appears as a special icon that selectively tracks the movement of the mouse and is hence called the *Metamouse*.

This chapter first describes the drawing program, which edits pictures consisting of boxes and lines. Section 3.2 explains the design of the Metamouse in terms of its body, sensory system and memory. Section 3.3 describes the generalization of actions, which involves inducing variables and isolating constraints. Section 3.4 discusses the construction of procedures. The chapter concludes with a brief review of the system's organization.

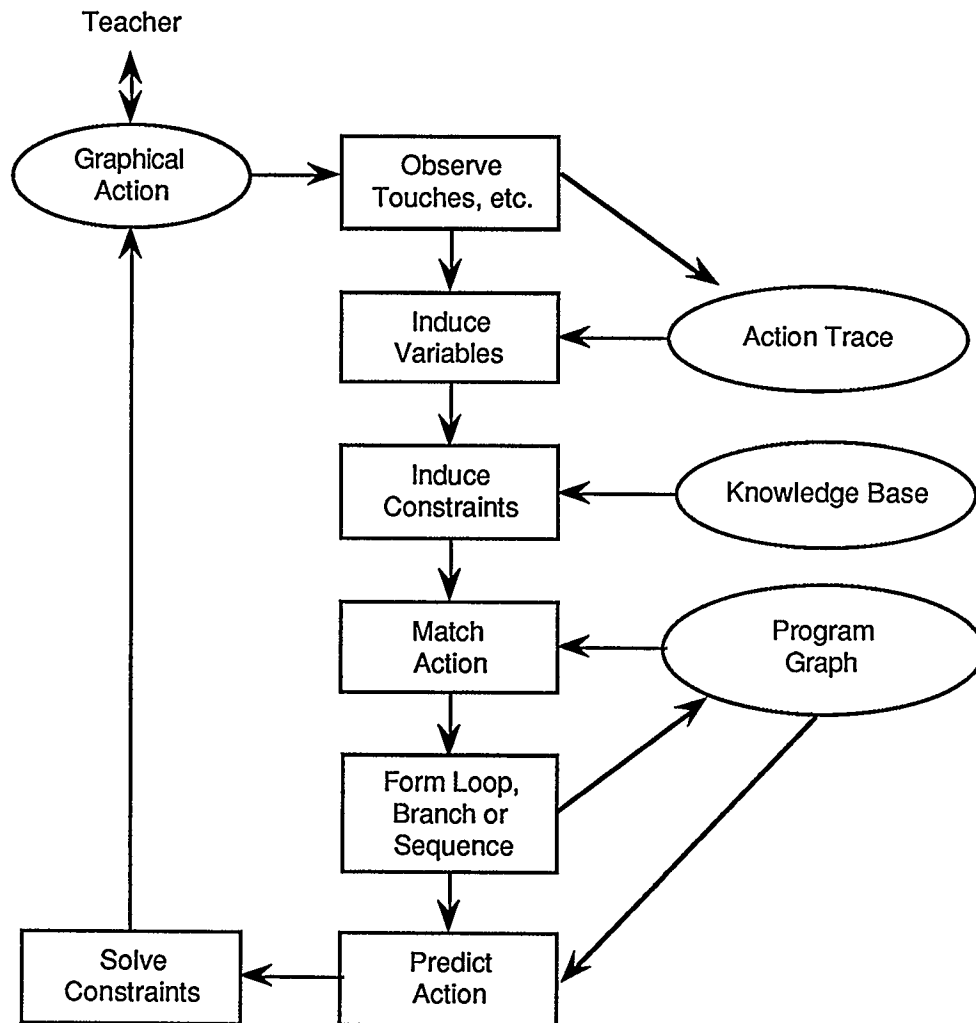


Figure 3.1 Prototype system for programming graphics by example

3.1 A.Sq—the Drawing World

The goal of this research is to help users extend the functionality of practical drawing programs. These programs typically provide a rich (more than complete) set of operators at various levels of abstraction. A graphical programming system, especially one intended for research, might well be based on a small set of graphical primitives and operators, formally represented as a system $G : \{P, O\}$. The minimal system sufficient to create and edit pictures on a raster display is $G_0 : \{[\text{pixel}(x, y)], [\text{place}(p), \text{remove}(p)]\}$, but this is of course impractical, since graphic displays typically contain more than 100,000 pixels, and since most drawings are composed of constrained groupings of pixels, such as continuous

and straight lines and circles. Noma *et al* have proposed [Noma 88] what we will call G_1 , where:

$P_1 =$ [line-through-points (l, p_1, p_2),
circle (c, ctr, rad)]
 $O_1 =$ [draw-line (l), erase-line (l)
draw-circle (c), erase-circle (c),
define-intercept-point-of-lines (pt, l_1, l_2),
define-point-at-distance-from-point-along-line (pt, d, p, l),
define-intercept-points-of-circle-and-line (p_1, p_2, c, l),
define-intercept-points-of-circles (p_1, p_2, c_1, c_2),
measure-distance-between-points (d, p_1, p_2)]

This system, similar to [Fuller 88], provides for the abstraction of graphical constraints from examples by naming special points derived using classical ruler-and-compass methods. A graphical interface to this system is conceivable: the user would draw lines and circles and mark intersection points; an interpreter would associate these actions with system operations. Although capable in theory of producing any picture, it falls far short of the facilities users expect.

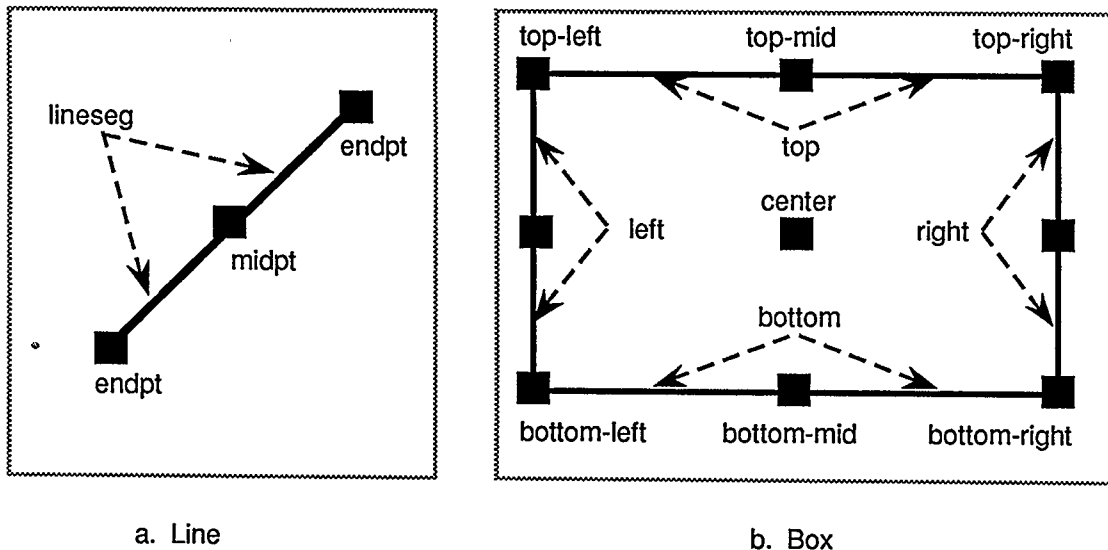


Figure 3.2 Parts of A.Sq objects

Simplicity and realism are both required for testing the thesis. A good compromise is to provide a few of the popular drawing program primitives and, perhaps even more important, the normal methods of manipulating them. The drawing program, called A.Sq (after the protagonist of Flatland [Abbott 1884]), emulates MacDraw and has a similar user

interface. It provides rubber-banding tools to make straight lines and boxes (rectangles with vertical and horizontal sides).

The user manipulates objects by moving their “handle” icons as in MacDraw (see Figure 3.2). A box has nine handles: the *center* effects translation of the entire box; handles on its boundary scale it in either or both dimensions. A line has three handles: its *mid-point* for translation and two *end-points* that effect a combined rotation and scaling (the user’s intention is probably to relocate that particular point). An object’s handles are normally concealed; in order to activate them the user must “pick” the object by moving the cursor near one of its edges and pressing a button on the mouse. Only one object may be active at a given time; it is designated A.Sq’s *active object*.

An object is erased using a deletion operator. A.Sq does not provide a rotation operator — but the Metamouse does (see below). To summarize the facilities formally:

A.Sq :

P = [line-between-points (l, p₁, p₂),
box-between-corner-points (b, p₁, p₂)]

O = [draw-line (l),
draw-box (b),
delete-object (obj),
translate-handle-of-object-to-point (hndl, obj, pt)]

The single operator for object manipulation, *translate-handle...*, is quite powerful and could of course be conceived instead as a number of routines for moving each type of handle. The formal model above is intended to express the uniformity of the system as the *user* sees it. A.Sq interprets *translate-handle...* according to the type of handle:

translate-handle-of-object-to-point (hndl, obj, pt):
type of hndl is

center or *mid-point*
translate obj by (pt – center-of (hndl)).

box-corner
scale obj in x and y by x and y components of
(pt – center-of (obj)) / (center-of (hndl) – center-of (obj))
relative to origin at
center-of (handle-at-opposite-corner-of (hndl))

box-top-or-bottom-edge-mid-point

construct target point p' as the intercept of
 horizontal-line-thru (pt) and vertical-line-thru (center-of (obj))
 scale obj in y by y component of
 $(p' - \text{center-of}(\text{obj})) / (\text{center-of}(\text{hdl}) - \text{center-of}(\text{obj}))$
 with respect to origin at
 center-of (handle-opposite (hdl))

box-left-or-right-edge-mid-point

construct target point p' as the intercept of
 vertical-line-thru (pt) and horizontal-line-thru (center-of (obj))
 scale obj in x by x component of
 $(p' - \text{center-of}(\text{obj})) / (\text{center-of}(\text{hdl}) - \text{center-of}(\text{obj}))$
 with respect to origin at
 center-of (handle-opposite (hdl))

end-point

translate line-end-point-at (center-of (hdl) to pt

If nothing else, the above shows how much simpler a pictorial interpretation is! Although A.Sq must resolve this high-level function into its several cases, it should be noted that the learning system uses the same simple model of primitives and operators that its teacher does.

3.2 Basil—the Metamouse

One way to operationalize the “show work” felicity condition is to *teach by demonstration*. The instructor performs a task so that the apprentice observes each step in its execution. Another approach is to *teach by leading*, a method employed in programming robots (see [MacDonald 84]). The instructor carries out a manual task but substitutes the robot’s grasper for his own hand, which he uses instead to manipulate the robot’s arm. The simplest learning system based on either method merely records the movements made, but with the addition of sensory feedback and some model of procedural decomposition, it could learn complex tasks.

The proposed system supports both approaches to teaching, with a view to investigating their relative merits and ability to supplement each other. The metaphorical apprentice is a graphical robot that tracks the movement of the regular graphics cursor under control of the mouse and is hence called a *Metamouse* (*cf.* [Myers 87]). In order to demonstrate an action the teacher performs it as usual. Each time she fixes the cursor’s position (as at the start and end points of a line), Metamouse moves there to indicate that the

action has been observed. In order to lead Metamouse, the teacher gives it a command by touching the appropriate part of its body. Demonstration and leading are not segregated “modes” of operation — the teacher can switch from one to the other without issuing any special command.

To help the teacher identify with Metamouse as an intelligence of some order, it is given a personal name, “Basil.” This section describes Basil’s interaction with the teacher, his sensory system and memory, and his knowledge of graphical semantics. Basil is designed to enforce the felicity conditions and to make reasonable conjectures regarding the constraints on parameters of graphical actions, yet nonetheless be simple enough that the teacher can readily understand his behavior. This implies that we should be able to describe Basil in relatively few words; a description given to teachers before they meet Basil in person is shown in Chapter 1, Figure 1.7.

3.2.1 Basil’s Body

The Metamouse augments the drawing facilities available in A.Sq (*eg.* by providing rotation) and gives names to some of the constrained movements performed by the mouse under user control (*eg.* move right without deviating vertically). These capabilities are literally embodied in the Metamouse, a moveable menu in the form of a turtle — an icon made familiar by the LOGO system [Papert 80]. Figure 3.3 illustrates.

The segments of Basil’s shell and body are buttons, arranged so that leading him resembles training a live animal. Touching his snout causes him to grasp or let go of an object; prodding a segment of his carapace causes him to move away along a straight line; touching one of his feet causes him to rotate about the tip of his snout in the direction of mouse movement. Some body parts have less obvious functions. Tapping Basil’s head causes him to go into his shell, that is, to ignore the teacher’s actions (he is reawakened by tapping any part of his shell). A tap on the center of his carapace causes him to move directly to the next position selected by the mouse. A tug on the tail causes him to undo his last action (this is how the teacher signals disagreement with one of Basil’s predictions).

Translation and rotation are continuous in space (to the limits of screen resolution) hence the teacher must have some means of indicating the interval or degree. For translation this is trivial: she selects the destination point with the mouse. For rotation a

number of approaches are attractive, such as measuring the mouse's angular movement. Control must be precise, so that the teacher can demonstrate rotation until contact (as used in the convex hull procedure). The method chosen is to continue moving in small but increasing steps as long as the mouse button is held down, so that the teacher relies on feedback from Basil to determine the limit of rotation.

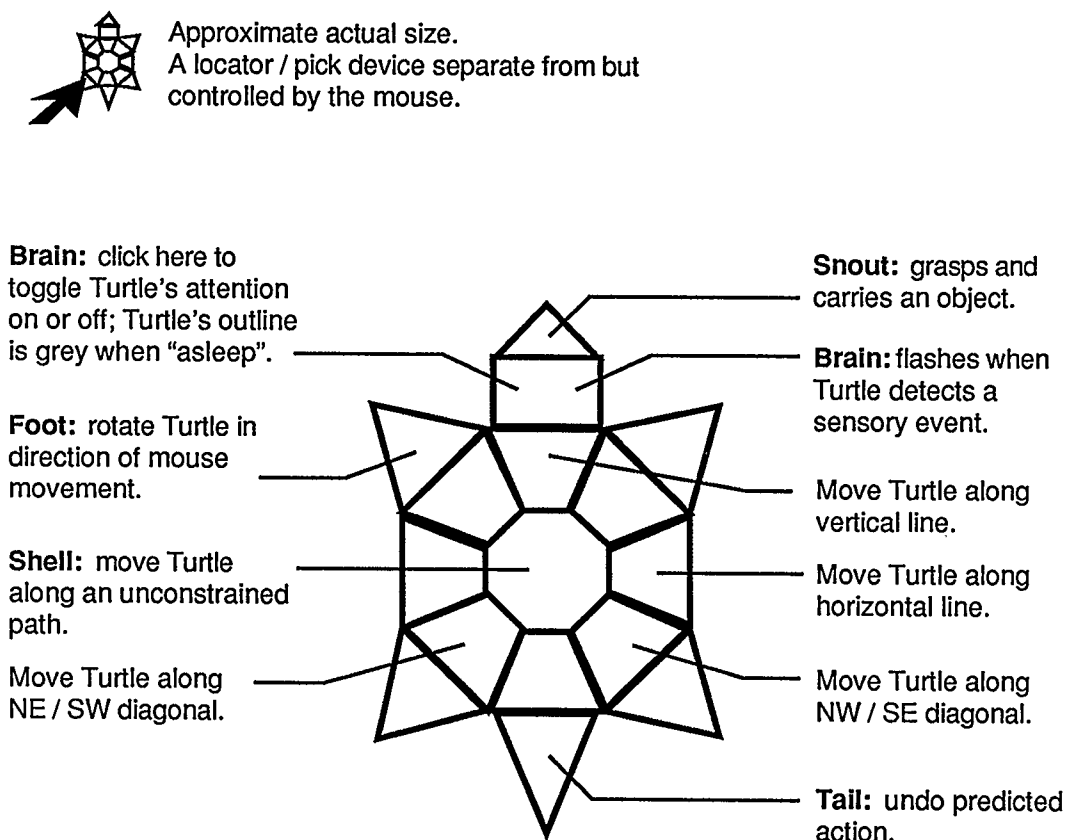


Figure 3.3 Metamouse icon with function buttons

Some implementations may not support a moveable menu and, in any case, users may prefer not to interrupt their work to prod the Metamouse. In teaching by demonstration the user shows Basil where to go by picking the destination point. Since the mouse travels an unconstrained path, some useful information is lost, but Basil conjectures a path constraint (see below). If the pick selects an object, Basil moves to it and grasps it. If the teacher then drags the object Basil re-grasps it at its new position. Letting go matters only if Basil must leave an object behind or grasp something else, hence this mode of activity offers no

explicit ungrasp command. Nor is rotation by demonstration supported, since A.Sq has no such command.

Path constraints are important when generalizing actions. They enable Basil to ignore the exact path or destination of the mouse, which may be noisy or irrelevant. On the other hand, they eliminate the need to induce the reason for going in one general direction rather than another. Moreover, when combined with other weak constraints (*eg.* one that stipulates contact somewhere along a line segment) they can determine an action's parameters. Eight path constraints considered the most useful in drawing are attached to buttons on Basil's carapace: two vertical (up, down), two horizontal (left, right) and four diagonal. Often Basil will be directed to go to a particular part of an object (*eg.* its bottom-left corner) along such a path. If the destination is slightly off the path but nearer to no other (*ie.* within 22.5°) Basil will deviate from the exact constraint to achieve the goal. The path constraint is recorded nonetheless. Similarly, if Basil is moved without constraint to the goal, he will find the nearest constrained path and record this as the direction in which to search for such a goal when executing this action in future. Thus goals have priority over paths but paths are used to narrow and order the search for a goal.

Finally, it should be noted that Basil records actions that modify other actions, such as selecting a drawing tool. In fact, Basil can learn any A.Sq command, though learning about the parameters of file manipulation is not attempted.

Formally, an action step executed by the teacher or by Basil is a tuple *action-step* (*precond, action, path, postcond*), where *precond* is sensory feedback prior to the *action*, which is a move (possibly modified by a drawing tool) or turn, whose direction is constrained by *path*, and whose parameters are further constrained by sensory feedback listed in *postcond*.

3.2.2 Basil's Sensory System

Geometric constructions — even the informal ones employed in van Sommers' model of drawing — conditionally distinguish points in space. These distinguished points, individually or in groups, result from operations whose semantics are order, measurement and classification. To satisfy the “no invisible objects” felicity condition, the teacher must draw objects whose intersections produce such points; Basil senses and remembers them as

the *tactile* postconditions of a step in the program. The learning system is conservatively biased towards highly specific tactile constraints, and limits the scope of its attention to touch relations between other objects and Basil's snout or the object in his grasp. If an action is not governed by tactile conditions, the system considers other constraints: absolute position, heading, or distance moved. Basil's model of "haptic perception" classifies and orders constraints so that only the most effective items observed are used to form postconditions. Moreover, it is biased towards novelty — information that has not changed since the precondition is ignored unless needed to distinguish the postcondition from an alternative.

Formally, the sensory state is *sensing*(G, DT, IT, D, P, H), where G is a *touch* relation (defined below) between Basil and the object in his grasp, DT is a list of touch relations between objects and Basil's snout, IT is similarly a list of relations between other objects and the object in grasp, D is distance moved in the current step, P is current position, and H is current heading. Details regarding each sense are given below. When analyzing sensory data, the system derives a *constraint* ($Data, Class, Used$) descriptor from each touch relation or other percept ($Data$), assigning it to a $Class$ and deciding whether it should be *Used* or ignored when explaining or generating actions. A precondition is *precond* (α) and a postcondition is *postcond* (β), where α and β are sets of constraint descriptors.

As mentioned above, touch is the most important sensory feedback. To clearly convey Basil's visual processing limitations, the teacher is told that Basil (as a Flatland inhabitant) sees objects almost edge-on. Nonetheless he can tell the shape and size of something by nudging it. Since Basil is an essentially tactile creature the appropriate analog for spatial relations is tactile, hence the construction of distinguished points (which in geometric construction are points of intersection) by establishing touch relations.

A pair of objects may touch at several points. Each touch relation is expressed as a correspondence of object parts, *touch* ($object_1.part_1 : object_2.part_2$), where $part_i$ indicates some part of $object_i$ (rather than the i th in some ordered set of parts). Objects are the A.Sq primitives box and line defined above. Parts are the handles (*viz.* specific points) and line segments illustrated in Figure 3.2. Thus the constraints expressed by touch relations are of three types, from strongest to weakest: 1) coincidence of two specific points, 2) intersection of a point and a line, and 3) intersection of two lines. The second is weaker

than the first because any point on the line may be chosen. To meet the third type, any point on either line will do. Figure 3.4 illustrates types of touch.

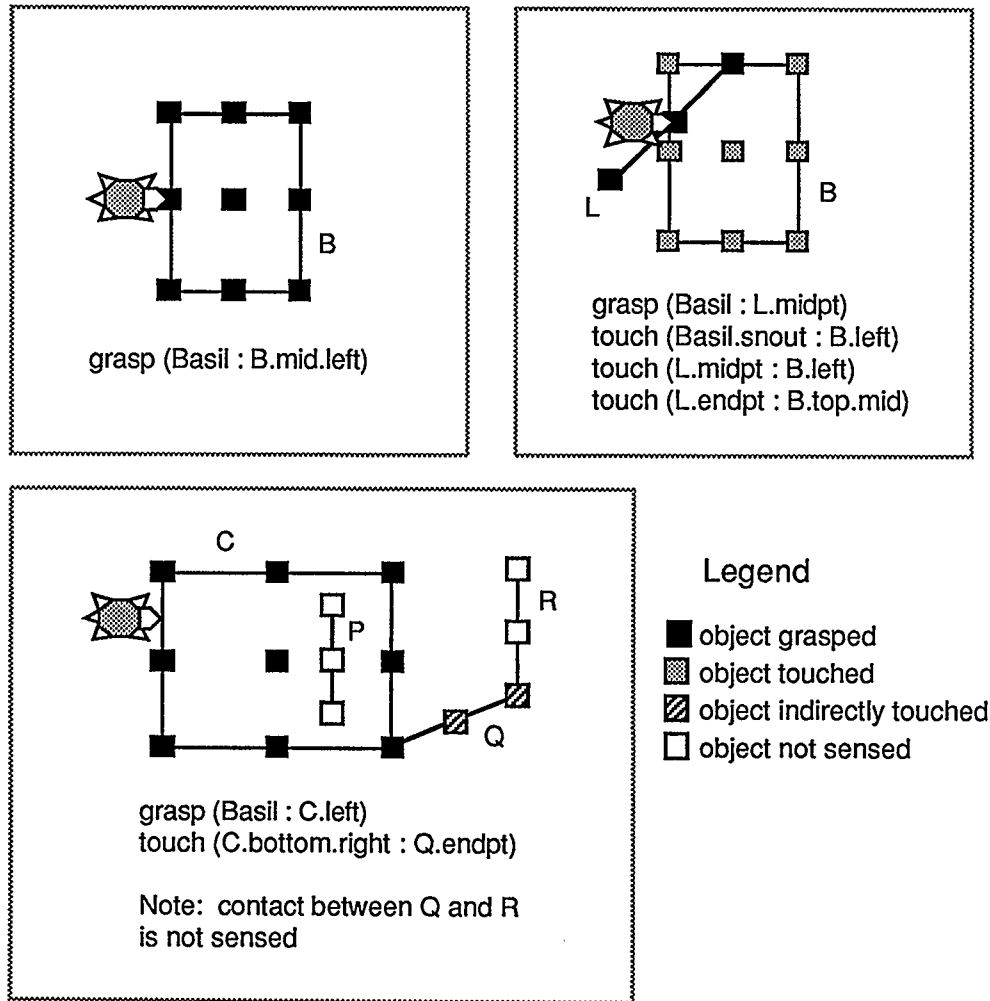


Figure 3.4 Touch relations in sensory feedback

Direct touch is a point-to-point or point-to-line constraint between Basil's reference point (his snout) and some part of an object. It isolates the current focus of attention and the origin point of the next operation. Basil may directly touch several objects, but only one part of each (the most specific part is selected). The touch relation with the object in Basil's grasp is not duplicated. Direct touch is expressed as a set of relations:

$$\{ \text{touch (Basil.snout : Obj}_1\text{.Part}_1), \dots, \text{touch (Basil.snout : Obj}_n\text{.Part}_n) \\
\mid \forall i,j [\text{Obj}_i \neq \text{Basil} \wedge \neg \text{grasp (Basil : Obj}_i) \wedge (i \neq j) \Rightarrow (\text{Obj}_i \neq \text{Obj}_j)] \}.$$

Grasping is a direct touch of the A.Sq *active object* (see 3.1). Grasping covers the same types of spatial constraints as direct touch but expresses Basil's ability to transform an object. It is defined:

$$\text{grasp (Basil : Obj.Part)} \equiv \\ \text{touch (Basil.snout : Obj.Part)} \wedge \text{Obj} = \text{active object.}$$

Indirect touch occurs between the grasped object and other objects on the display. It covers all three types of spatial relations and expresses more complex constraints than either direct touch or grasping. It is expressed as a set of touch relations:

$$\{ \text{touch (G-obj.G-part}_1 : \text{Obj}_1.\text{Part}_1), \dots \text{touch (G-obj.G-part}_n : \text{Obj}_n.\text{Part}_n) \\ \mid \text{grasp (Basil : G-obj)} \wedge \\ \forall i,j [\text{Obj}_i \neq \text{G-obj} \wedge \text{Obj}_j \neq \text{Basil} \wedge \\ \neg((i \neq j) \Rightarrow (\text{Obj}_i \neq \text{Obj}_j)) \wedge \neg((i \neq j) \Rightarrow (\text{Part}_i \neq \text{Part}_j))] \}.$$

In Box-to-Line for example, when Basil contacts the first box with the sweep-line, his sensory state (the postcondition of moving the sweep-line upwards) includes the following touch relations (see §4.6, step 6):

grasping:
 grasp (Basil : S.midpt)

touching:
 nil (*ie.* nothing he is not grasping)

indirectly touching:
 touch (S.lineseg : G.lineseg) touch (S.lineseg : B.bottom-left)
 touch (S.lineseg : B.bottom-mid) touch (S.lineseg : B.bottom-right)

definitions of variables:
 S: the sweep-line created by Basil
 G: the guide-line created by Basil
 B: the object (a box) just encountered by the sweep-line

Observe that Basil notes the three contacts between S.lineseg and the handles on B.bottom but not the line-to-line contact, touch (S.lineseg : B.bottom). This is because any one of the point-to-line contacts implies a line-to-line constraint. Basil records only the most constraining relations observed. In this case collinearity is captured by the occurrence of at least two point-to-line touches.

In addition to touch, Basil senses current position and heading in both absolute and relative terms. Absolute position, formally represented as *position*(x, y), can provide a pair

of absolute screen coordinates as a program constant, useful when creating tools such as the sweep-line. Similarly, *heading(h)*, where *h* is an anti-clockwise angle in degrees from the horizontal, provides a directional constant. Relative — in effect the change in — position or heading is derived from the magnitude of a move or rotate operation and is formally *displacement(m, path)*, where *m* is a distance in screen coordinate space (move) or an angle in degrees (rotate), and *path* is a heading descriptor. Displacement facilitates programming in terms of body-centered coordinates, as in turtle geometry [Abelson 80].

3.2.3 Basil's Memory

The use of variables and constants mentioned above entails some sort of memory. The organization of a memory may be considered in terms of data types or persistence over time. Basil remembers *procedures* “forever,” *objects* for their lifetime, and sensory feedback — or *state* — briefly. Basil does not “actively forget,” even though this could be used to remove bugs from procedures. There are six memory partitions (one of which properly belongs to A.Sq) that fall into three data types and four degrees of longevity.

Current-Step (immediate)

the *action-step* Basil has selected to execute or teacher is demonstrating

Recent-Steps (short-term)

action-steps recently performed, which Basil may undo or to which he may refer when inducing variables

Program (long-term)

the executable procedure learned by Basil;
may be stored in a higher-level long-term memory (an archive)

Created (medium-term)

the list of variables bound to A.Sq objects created by Basil during this task

Transformed (medium-term)

the list of variables bound to A.Sq objects transformed but not created by Basil during this task

Display-List (medium-term)

the A.Sq objects currently visible on the graphics display

The three data types are: *action-steps*, defined in 3.2.1; *procedure*, a directed graph of *action-steps*; and *variable*, a name dynamically bound to an A.Sq primitive. Degrees of longevity are *immediate*, persisting through duration of the current *action-step*; *short-term*,

over a small number of *action-steps*; *medium-term*, throughout a single teaching session; and *long-term*, “forever.”

The system remembers actions in two forms: specific actions performed by the teacher or Basil during a given trace; and generalized actions recorded in the Program memory. *Current-Step* is the *action-step* tuple describing the action most recently performed by Basil or the teacher.

Recent-Steps is a stack of previous contents of *Current-Step* but in which direct references to A.Sq objects have been associated with variables. *Recent-Steps* is used when Basil must undo an action the teacher has rejected; this permits the undoing of several actions. The *undo ()* function performs the inverse of the top element of *Recent-Steps* and pops it off the stack. *Recent-Steps* is also used to find previous occurrences of an A.Sq object in touch relations, so that a variable can be induced. The *find-most-recent-occurrence-of (Object)* function searches back through *Recent-Steps* for a variable bound to *Object*. The use and induction of variables is described in Section 3.3.

The size of the *Recent-Steps* memory is one of the parameters on which the learning system’s power can be conditioned. If it is too small, some variables may not be induced. Moreover, it limits the number of actions that can be undone, hence the amount of prediction that can be required to confirm the program’s structure. On the other hand, if it is too large, the system will spend more time looking for variables and is more likely to induce them speciously. A reasonably small size, say 7 ± 2 , is recommended for a prototype system; the performance of larger memories is certainly worth further investigation.

The long-term *Program* memory contains the procedure that Basil is learning or executing. It is a directed graph of *action-step* tuples with generalized pre- and postconditions. Each node may have many successors and predecessors; this is sufficient to represent flow of control through sequences, loops and branches but does not impose a block structure. Figure 3.5 illustrates an example program graph. This representation is at least as general as that used by NODDY [Andreae 85].

The function *successors (A, P)* returns the set of action steps that immediately follow step A in program graph P and therefore may be predicted after A. Function *predecessors (A, P)* returns the actions that immediately precede A. The function *find-match (D, A, P,*

R) invokes *predecessors()* and *successors()* to search the graph *P* starting at step *A* for a possible match with *D*, the action just demonstrated by the teacher, ignoring matches *R* already found but rejected for other reasons. Although potentially exhaustive, the search is biased so that loops and local jumps are considered first, and suspends when the first match is found. The biasing heuristics are described in Section 3.4.

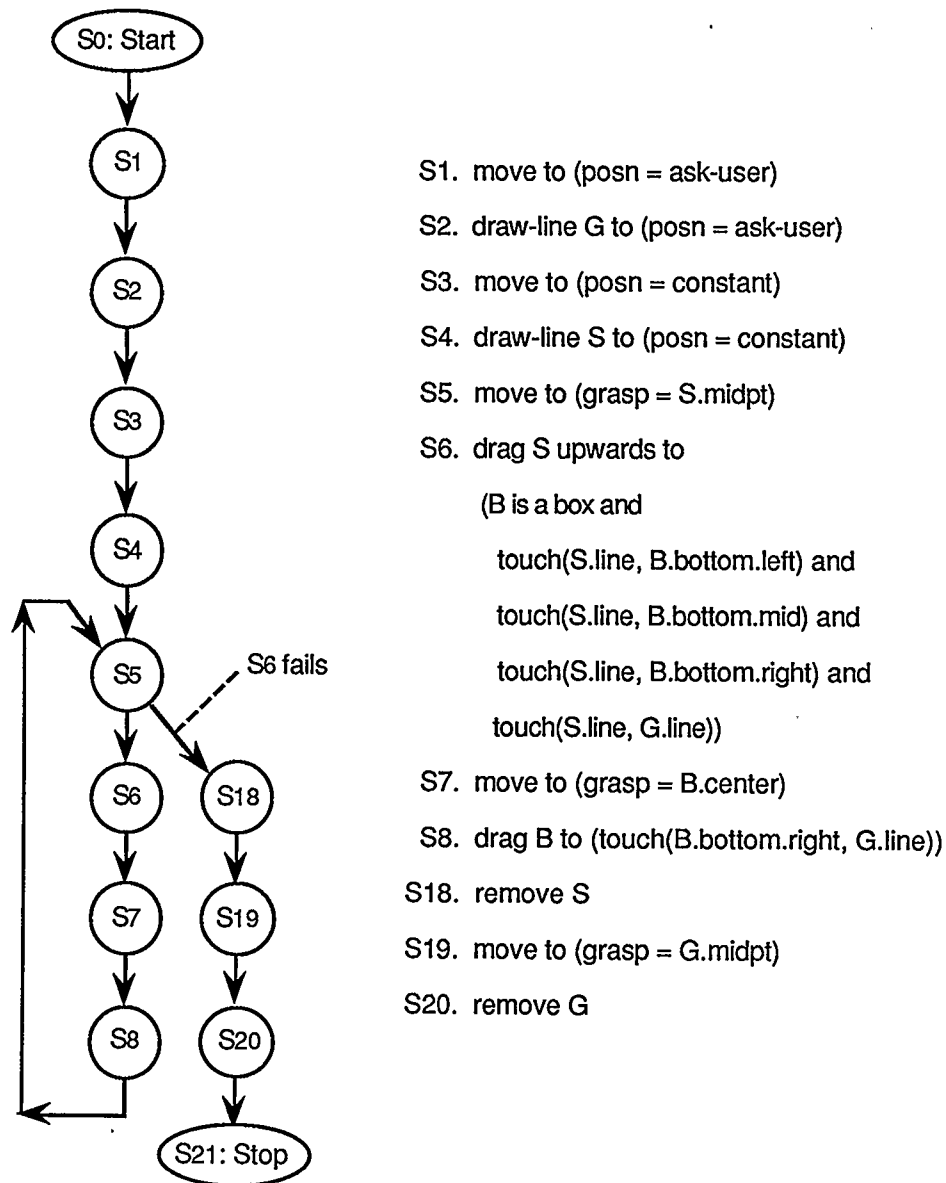


Figure 3.5 Program graph for “box-to-line”

Since Basil must be able to undo *action-steps*, he must remember the bindings of variables in past steps. Bindings hold until a step is re-executed, but this is insufficient since it may be re-executed before its previous execution is undone. Hence Basil always appends a copy of *Current-Step* to a short-term memory, *Recent-Steps* — in effect an undo list. The length of *Recent-Steps* is fixed; once the limit is reached, the oldest step is forgotten as the newest is recorded.

Graphical construction typically involves “auxiliary” objects to measure distances or otherwise capture constraints. To satisfy the “no invisible objects” felicity condition, the teacher must draw such objects. Since they are apt to be re-used throughout the procedure they should be remembered beyond the limits of *Recent-Steps*. Thus the system creates a variable for every object drawn in the execution of a task and records it, with its binding, in the *Created* memory. Variable names are themselves program constants; this causes one serious problem. If a step that draws an object is executed more than once it cannot give unique names to each one. The current design does not eliminate this problem; instead, it adopts the convention that a variable name may appear more than once in *Created*, and that variable instances are selected non-deterministically; objects having the same name can be differentiated by other criteria. For example, suppose that *Created* contains several boxes called X. The following subprogram would delete all of them:

```
while (move along any path to achieve grasping (Basil : X.center)) succeeds
  delete-object (X)
```

In fact, “while (delete-object (X)) succeeds” would work, although Basil would inevitably learn to move to X by observing the teacher pick X. If objects having the same name must be ordered, then the teacher must express this through a construction such as a sweep-line.

Graphics programs process objects by transforming them. It may matter that objects be processed only once and in a particular order. Transforming an object however may position it such that it will be selected again by mistake (consider for example the Spacing task in Chapter 1). To prevent this, Basil remembers references to objects already transformed in the *Transformed* memory. If the teacher rejects Basil’s prediction that such an object T is selected again, the caveat “T not already in *Transformed* memory” is added to the action’s postcondition.

When a program step is learned (or executed) its variables are bound to actual values provided by Basil’s “sensors.” A.Sq records the state of all objects currently displayed in a

list of object data structures, *Display-List*. The function *select-by-constraint* (*Display-List*, *Type*, *Touch-Specifier*, *Rejected*) finds an object of given *Type* (eg. box) for which *Touch-Specifier*, a touch predicate, holds, and which has not been found and rejected for other reasons already.

3.3 Generalization—Actions

In order to learn procedures inductively the system performs generalization. References to individual objects are replaced by variables that may be bound dynamically as the procedure is executed. Sensory feedback is generalized to pre- and postconditions by ignoring some aspects of it and by relaxing constraints on numbers (as in distance) or on touch relations. Sequential action traces are reorganized as a program graph with branches and loops. This section discusses the generalization of actions by introducing variables and relaxing conditions. The method for constructing a program is described in Section 3.4.

Action generalization is performed at three stages of program development.

The first stage is when trying to match a new action with a program step. Many program steps can potentially match an action if generalization is permitted. Preference should be given to a match that requires a combination of the least generalization and the least amount of search through the graph. The prototype system requires exact matching. Generality is introduced by disjoining, that is by creating branches in the graph.

When matching a touch relation in a program step with actual sensory feedback, variables in the relation must be bound to objects occurring in the same role in the feedback. For example, touch (L.midpt : B.center) matches touch (<Obj #99>.midpt : <Obj #31>.center) provided L = <Obj #99> and B = <Obj #31>.

The second stage in which generalization occurs is when adding a new step to the program. Sensory data are classified in order to identify determining constraints; the rest can be considered irrelevant for generating actions, though they may later prove useful in distinguishing situations.

At this stage, variables replace object references. If the object occurs in *Recent-Steps* or *Created*, the variable is inherited; otherwise a new variable is created.

Note that this second level of generalization could be performed *first*. It is analogous to perceptual processing in that significant sensory data are abstracted from the input. This “perceptual” generalization is quite powerful, since it captures essential constraints which are likely to be intended by the teacher. Performing it on actions before attempting to match them with program steps would of course reduce the amount of generalization required at that stage but this does not matter. More important is the consistency of Basil’s behavior. Since perceptual generalization can be explained to teachers in terms of a simple and consistent model, it should be given priority.

A third level of generalization is performed when executing a program. At any point a number of alternative steps (branches) may be available. If a step cannot be predicted because its precondition does not match the current feedback or its postcondition is unattainable, it may be generalized by relaxing the troublesome condition. This type of generalization seems especially risky; it is recommended to let the teacher demonstrate the correct action and then generalize the program step to match it provided excessive generalization is not required. The prototype system skirts this issue by disjoining the teacher’s action with the current alternatives.

Sometimes the teacher rejects a prediction but performs an action that matches the offending program step nonetheless. This would occur if the step had been overly generalized or had been selected for prediction instead of a more appropriate alternative. To prevent the latter case alternatives should be ordered by generality if this can be determined. In the former case the step should be specialized by reclassifying some of its conditions as relevant. To do this requires that all conditions be remembered permanently. Moreover, to prevent over-specialization, the system should also remember the most specialized version of the step that covered all situations actually encountered. Since the teacher may reject incorrectly, or the constraint may not be strictly learnable in terms of the system’s representation, even this minimal version may conflict with the required specialization. The prototype system avoids this problem by not specializing steps at all; instead, it adds the teacher’s action to the program as a preferred alternative.

The remainder of this section concerns the second level “perceptual” generalization: specifically the induction of variables and analysis of constraints.

3.3.1 Variables

An object or value may occur several times in the execution of a procedure. Some of these occurrences may be related. For example, the teacher may wish to instruct Basil to grasp “the center of the box contacted by the sweep-line at the end of the previous step.” We can think of an object playing one or more roles in the course of a task. Further, a role may be played by several objects on different occasions. For example, the boxes in “box-to-line” play the same role in each iteration of the loop. This notion of role is captured by variables and valuation functions that get or change their binding. In the prototype system, variables refer only to A.Sq objects. Numerical values such as position and distance are typically specified by objects, but it is acknowledged that variables would be useful to capture notions like “the position occupied 3 program steps ago.”

Formally, a variable is a relation *variable-definition* (*Name*, *Binding*), where *Name* is unique (scoping is global since the program graph has no block structure) and *Binding* is an A.Sq object. Since the distinction of objects by type is especially important, *Binding* is decomposed into (*Type*, *Object*), where *Type* is one of {box, line} and *Object* refers to a specific instance.

A given program step may set or re-set a variable’s value, whereas another may merely inherit it. With reference to Figure 3.5, step S6 sets the value of B by solving a constraint, whereas S7 and S8 inherit B. At the next iteration of S6, B is re-set. To ensure that values are changed as required, individual occurrences of variables in program steps are given as the relation *variable-reference* (*Valuation*, *Definition*), where *Valuation* is a function to set or inherit the value and *Definition* refers to the variable-definition record. Figure 3.6 illustrates the referencing scheme.

There are three valuation functions (examples refer to Figure 3.5): *Create* binds a variable to the new object drawn in *Current-Step*, eg. line G in step S2. *Find* binds a variable to an object found by solving constraints, eg. B in S6, “the first box whose bottom-edge handles contact the sweep-line as it moves upwards.” *Same* inherits the existing binding, eg. B in S7, “the box contacted by the sweep-line at the end of the previous step.” The choice of valuation function depends on occurrences of the object and can be decided when the variable is induced. *Create* and *Find* are associated with a new variable; *Create* if the object was just drawn, *Find* if it occurs in a touch relation. *Same* is selected when a previous occurrence is found.

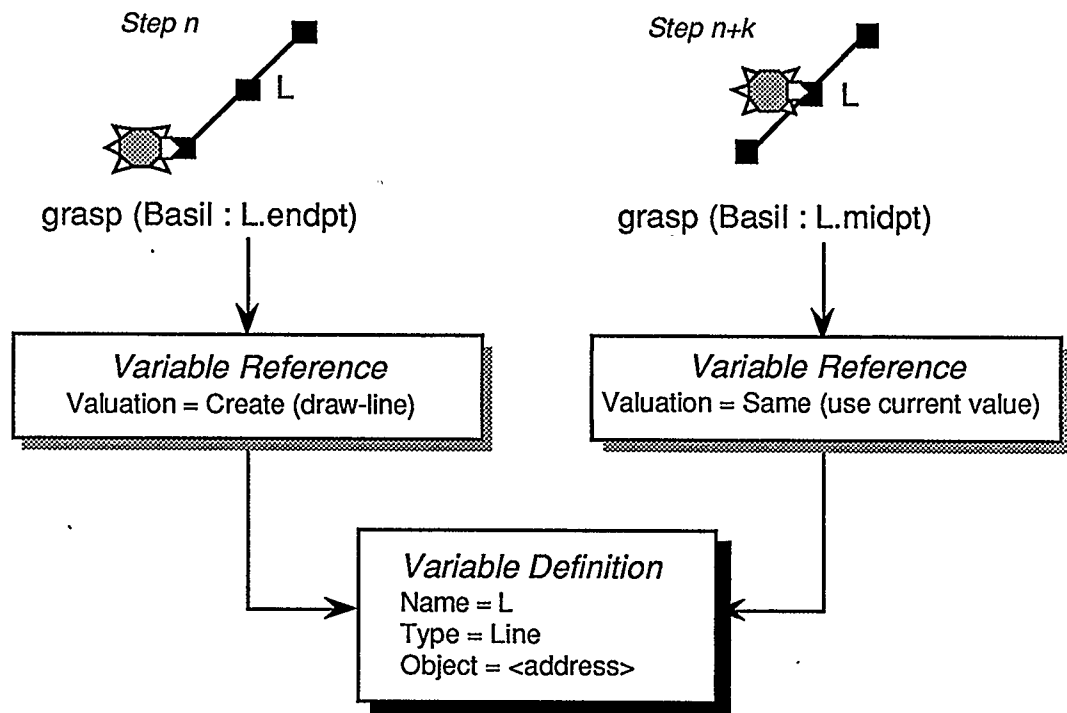


Figure 3.6 Definition and instantiation of a variable

Induction of variables rests on the assumption that references to the same object on different occasions are not merely coincidental. A variable is induced from an object reference in *Current-Step* and previous occurrences in *Recent-Steps* or in the *Created* objects memory. It is convenient to create a new variable for any object appearing for the first time, and to keep variable-references in *Recent-Steps* and *Created*. In order to distinguish different roles an object may play during a task, the degree of look-back or the size of *Recent-Steps* should be limited. The most suitable limit has not been determined. Since variables in *Created* are never re-bound, greater generality is achieved by inducing variables from *Recent-Steps* and so this should be searched first. An algorithm for inducing variables is presented in Chapter 4.

3.3.2 Constraints

Constraints implicit in touch relations and other sensory feedback must be isolated from coincidental occurrences in order to capture the teacher's intentions. Here again the best one can do is to make the inductive assumption that the important constraints are those occurring consistently. Since actual feedback is a maximally specific description of

constraint, the isolation of features is a matter of generalization. [Michalski 83] describes a number of methods of generalizing conjunctions of predicates. The prototype system generalizes a feedback pattern by ignoring predicates that match generalization criteria (heuristics). They should be remembered, however, in case they are needed for specialization, to distinguish the pattern from others.

The criterion for dropping a feedback predicate could be related to its type or structure. For example, Basil ignores third-degree indirect touch — an implicit generalization of feedback. More interesting, however, is an event's effectiveness as a constraint. For example, *touch (Basil.snout : S.midpt)*, where *S* is a particular line, is more constraining than *touch (Basil.snout : B.center)*, where *B* is some box found by moving generally upwards. Thus, if both occur together, the latter is more likely to be relaxed by generalization. The feedback generalization function, specified in Chapter 4, classifies touch relations and other feedback according to the constraint they place upon the operation performed in *Current-Step*. The generalization heuristic declares that predicates of certain classes should be ignored. The constraint classes are described below.

Effective constraints help determine an operation's parameters. A *determining* constraint determines them exactly. Contact with a handle of an object whose variable valuation is *Same* or *Created* is determining: for example, grasping the mid-point of created line *L* determines the (x, y) components of a move. A *strong* constraint can determine exact position when combined with a weak constraint: contact with a handle of an object whose valuation is *Find* is strong. A *weak* constraint does not determine Basil's exact position: contact with an edge and path of motion are weak. A *crossing* constraint is an indirect touch between edges and has a very weak effect on position. A crossing constraint on drawing a line through another line keeps Basil somewhere within the frustum bounded by the line segment crossed and the projector rays from Basil's starting position. In the presence of a determining constraint, other conditions should be reclassified as *overdetermined*.

Unchanged conditions persist through an operation and thus appear in both pre- and postconditions. The *trivial* ones, such as maintaining the grasp of a handle while dragging it, are given by the definitions of operators. Discarding them makes no difference to a postcondition's generality. *Sustained* are typically weak constraints that do not change.

For example, if Basil moves along an edge the contact is sustained though not at the same position.

If touch predicates fail to sufficiently constrain (explain) an action's parameters, then either position or distance may be a determining constraint. Thus Basil asks the teacher whether the action is determined by a constant or a run-time input position or distance. If none of these, then the trace contains an invisible object (see Chapter 2).

Having classified all the feedback, the system generalizes it by marking some features as ignored. Overdetermined and trivial constraints are safely ignored — though it is not impossible that overdetermined constraints have distinguishing power. Whether or not to discard a sustained feature is a question of considerable importance to the types of problems that can be programmed. In the real world, things that do not change are usually not noticed. Since the generalization discussed here is based on a sense-perception analogy, it is suggested that Basil will be more comprehensible to the teacher if he normally ignores sustained constraints. Nonetheless, the system should prepare for those occasions when the teacher intends that a feature be preserved, by remembering the constraint.

3.4 Generalization—Procedures

Given the internal representation of a graphical task as a sequence of actions governed by tactile conditions, which is conveyed to the teacher through the Metamouse's predictions and questions, consider the matter of generalizing a sequence of *action-steps* into a program with branches and loops. The *Daedalus* algorithm constructs a network of *action-steps* that embodies such control structures. Since the algorithm both learns and executes procedures, it meets the design requirements of prediction and continuous, incremental learning. The algorithm depends upon a (precondition, action, postcondition) model of *action-steps* but is otherwise domain independent, although its effectiveness depends upon the suitability of the generalization terms used in pre- and postconditions.

The learning algorithm constructs a directed graph (the structure specified for Basil's Program memory). Each node is an *action-step* with links to preceding and ensuing steps that may include itself. Thus the graph can represent looping and k-way branching with return from a branch (*cf.* block or subroutine) to its parent branch. A loop's entry point is itself a k-way branch. The alternatives at any branch point should be ordered on the

generality of their preconditions, then on postconditions, and finally on path constraints. Figure 3.5 showed an example graph. The learning algorithm can add nodes and links but never alter or remove them, although a *tentative* link may be removed if evidence supporting it (*ie.* the success of predictions) proves insufficient.

Given that Daedalos constructs an arbitrarily connected graph, it follows that the programs it learns are not block structured. This may be unappealing to disciplined computer programmers but may have quite the opposite effect upon teachers, who are free to structure a task as they wish. The algorithm is nonetheless biased towards localizing jumps (see below).

3.4.1 Definitions

The algorithm is presented here in its most general form; functions specific to the Basil implementation are described in Chapter 4. Several important variables are here defined in terms of how they are used:

PredictionSet

When executing a program, Daedalos selects the next step to perform from PredictionSet, the options available at this point in the program; if none both applies and is accepted by the user, Daedalos learns a new step and adds it to the PredictionSet.

ProposedStep

If a member of PredictionSet fits the current situation (*ie.* its preconditions match the current state), and if its parameters *P* can be instantiated to *P'* satisfying its postconditions, then it becomes ProposedStep and Daedalos tentatively performs it.

LastAcceptedStep

If the user accepts the ProposedStep, it becomes LastAcceptedStep and Daedalos proceeds to its successors (the next PredictionSet); otherwise Daedalos looks for an alternative from the current PredictionSet.

CurrentStep

If Daedalos is unable to predict the next step (nothing in PredictionSet can become ProposedStep, or user rejects all ProposedSteps), the user must teach it. If the new step, called CurrentStep, matches one already learned in another part of the program, Daedalos proposes a link from LastAcceptedStep to the match; otherwise CurrentStep, deemed novel, is appended to LastAcceptedStep and then itself becomes LastAcceptedStep.

Several integer parameters control the algorithm:

OptionsLimit

When choosing the next step to execute, Daedalos may find several whose pre- and postconditions match current and projected situations; Daedalos predicts each in turn until either the user accepts one of them or the number of predictions exceeds OptionsLimit.

ConfirmsLink

When Daedalos matches the teacher's action to a program step, a link to that step is conjectured, but must be confirmed by successful prediction of at least ConfirmsLink steps beyond that point.

LinkAttemptsLimit

If a conjectured link is rejected, Daedalos may try as many as (LinkAttemptsLimit - 1) other linkages before deeming the teacher's action a novel program sequence and appending it to the current branch.

3.4.2 Algorithm

The Daedalos algorithm both learns and executes a procedure, commuting between these two modes, signalled respectively by closure and failure. A failure occurs when no member of the prediction set is performable or the teacher rejects all alternative predictions. The end of the program is also treated as a special sort of failure so that the user may add a continuation. A closure occurs when the teacher indicates the lesson is over or Daedalos verifies that the teacher's actions match some part of the existing program. Thus in simplest terms the learning algorithm is:

Daedalos-Lesson (Program, Trace)

Initialize Program

while **Check-for-End-of-Lesson** signals not-end-of-lesson

Execute Program starting with PredictionSet of LastAcceptedStep
 {until failure or end-of-program}

Learn from Trace a new sequence in Program following LastAcceptedStep
 {until join-achieved or end-of-lesson}

Store Program

Initialize. The algorithm is given a procedure Program that may have been selected by the user from an archive. If Program is new, it is initialized to an empty program containing dummy Start and Stop events which are used to control the execution of Daedalos.

Check-for-End-of-Lesson. The program will be executed from the PredictionSet following LastAcceptedStep. If the only successor is Stop, Daedalos may need to learn new actions from the trace.

Execute. If the lesson is not over, there may be more program to execute before new actions are learned. Check-for-End-of-Lesson could signal end-of-program if the next step is Stop, but Execute must check for this anyway. Execute, given the PredictionSet (successors) of the LastAcceptedStep, looks for an option whose relevant preconditions hold in the current state of the world and whose relevant postconditions can be achieved by applying its operator with variable parameters set by a constraint-solver. If the teacher accepts this prediction, Execute recursively calls itself with the next PredictionSet. Otherwise, the action is undone. The system will look for alternatives from the PredictionSet until OptionsLimit is reached. In the event that no prediction succeeds, Execute returns and the Learn routine is entered.

In the graphical world of Basil, preconditions hold if they and their variables match corresponding items in Basil's immediate memory (the CurrentStateOfWorld).

To determine whether a Step's postconditions are attainable requires a constraint solver to find values for the operator's variable parameters such that postconditions hold. Checking the postconditions is easy — the method is the same as for preconditions. Designing and implementing a constraint-solver is more difficult.

Learn. When Daedalos is unable to continue executing the program, it acquires new program steps from the action trace. The Learn routine examines the next step in the trace and looks for a matching step already in the program. If a matching step is found, the LastAcceptedStep is connected to it through a Join node; otherwise, the trace step is copied to a new program step, which is appended to LastAcceptedStep, and Learn continues from the new step.

```
Learn (Trace, Program, LastAcceptedStep)
  if CurrentStep ← Get-next-trace-step from Trace is null
    signal end-of-program
  else if LinkTo ← Find-and-confirm-join in Program
    from LastAcceptedStep to a step (LinkTo)
    that Matches CurrentStep
    Make-join from LastAcceptedStep to LinkTo
    signal join-achieved
```

otherwise

Append CurrentStep to LastAcceptedStep

LastAcceptedStep \leftarrow CurrentStep

Learn next step in Trace, trying to join it to updated Program
or else appending it to LastAcceptedStep

The routine to get a new action step from the trace creates a Step node (which will be thrown away if the step is matched) and fills in its *action-step* data slots.

Find-and-confirm-join. When the new step has been read in, Daedalos searches the program for a step that matches it. If a matching step is found, Daedalos must confirm the link to it by successfully predicting at least ConfirmsLink steps beyond that point (or by reaching the end of the program), before making a join from LastAcceptedStep.

To find a matching step Daedalos can search the entire program graph. The search is depth-first, but biased towards nodes near the LastAcceptedStep, where search commences. It is also biased towards finding loops, by searching chains of LastAcceptedStep's predecessors first. These biases are suitable to the two types of hierarchical task decomposition found by van Sommers (see §2.2.1) Duplicate searches and endless loops are prevented by marking nodes as encountered.

Matches. The action matching predicate compares *action-steps* and succeeds if the program step is (or can be made) a generalization of the demonstrated action. Given that steps can be generalized in order to match, the number of candidate matches could increase uncontrollably. Moreover, the best matches may be missed if the first match found is accepted. Hence the system should limit the amount of generalization, produce all possible matches and order them by closeness and other preference criteria. To avoid these problems, the prototype system should generalize only path, position, distance and heading — very easily justified generalizations — and not bother ordering candidates.

3.5 Putting the System Together

From the user's point of view, the graphical programming system has two modules — the drawing program A.Sq and the apprentice Basil. For the purposes of research, however, Basil, the interaction device with senses and memories, is distinguished from Daedalos, the system that constructs program graphs. Basil is an intermediary between Daedalos and A.Sq and performs a first level of generalization.

The three modules, their components and interfaces are shown in Figure 3.7. The teacher transmits mouse events to A.Sq, which interprets them as graphical actions and returns visual feedback. If Basil is active, he observes these A.Sq actions and the sensory feedback before and after each step. Basil induces variables and constraints, then transmits these observations to Daedalos, which further generalizes them and incorporates them into the program. When predicting the next action step, Daedalos sends the generalized action specification to a constraint solver to determine the action parameters; if the solver succeeds, it sends a graphics command to A.Sq; otherwise it informs Daedalos of the failure.

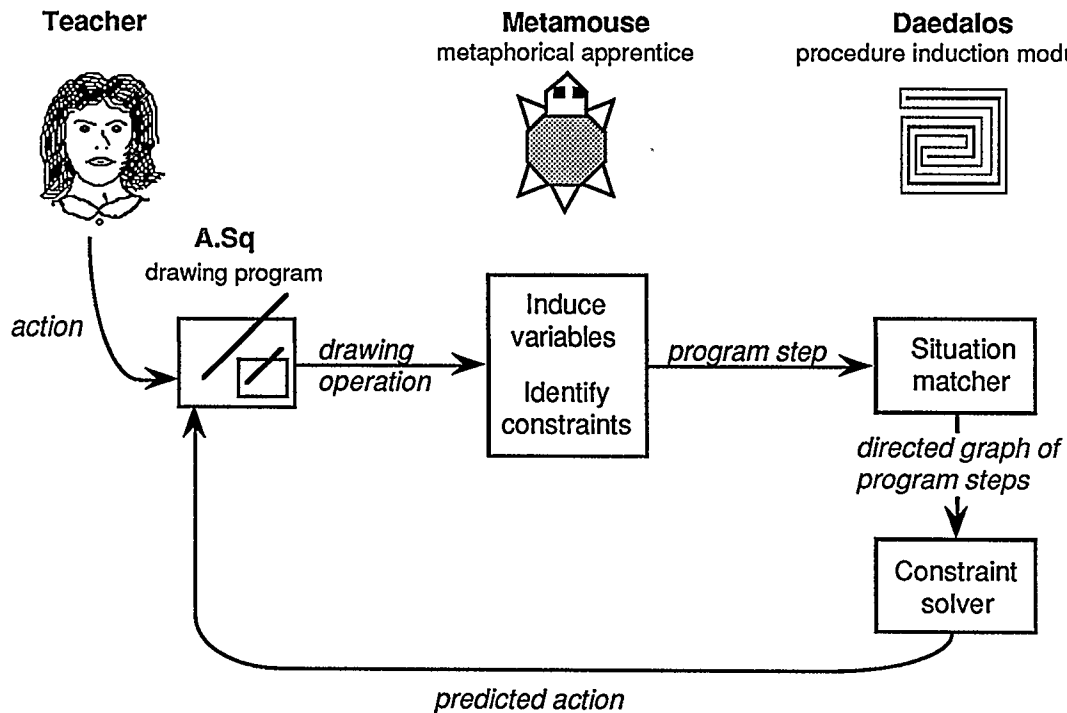


Figure 3.7 Main modules of graphical programming system

Chapter 4

An Implementation

A phased implementation of the system described in Chapter 3 allows feasibility studies and investigation of generalization heuristics and the requirements for constraint satisfaction, prior to committing the resources needed for a working prototype. The initial (Phase 0) system described here has no constraint solver. Although able to induce procedures it cannot generate actions in the display environment, hence full graphical interaction with the teacher is not supported. The Phase 0 system includes the A.Sq drawing program and a Metamouse that follows the teacher and informs her of its sensations. It records execution traces but induces variables and constraints off-line (*ie.* after the trace is completed). Daedalos operates interactively but in a separate session and uses textual representations.

Implementing the system presents some interesting challenges. Its major modules (A.Sq, Basil, and Daedalos) are inherently complex: they receive, process and generate large amounts of information. The interfaces between them have a high bandwidth and interactivity. Efficiency is a major concern. The fully working system must respond to events in real time. It must perform complex generalization inferences and solve geometric constraints within the teacher's short-term attention span.

4.1 Phase 0 Implementation

The system's modularity enables successive versions to be built on the same basic structure. The Phase 0 implementation includes most subcomponents of the major modules, A.Sq, Basil, and Daedalos. A.Sq is as described in Chapter 3. Basil has the sensory capabilities given in the design. He can follow the teacher's actions but cannot be told directly to move or rotate. Basil therefore can be taught by demonstration but not by leading. Daedalos follows the algorithm given in Chapter 3.

The major omission is the interface between Daedalos and Basil. In a complete implementation Basil would report each action to Daedalos immediately after processing it. Daedalos in turn would transmit predictions to Basil. In Phase 0, Basil processes and

stores the entire action trace. Daedalos then re-processes it, interacting with the teacher through a textual dialogue. Since there is no constraint solver, Daedalos predicts actions but not the particular objects involved. Phase 0 facilitates a feasibility study to answer such questions as:

1. Can the teacher predict Basil's sensory responses?
2. Does the condition-action model apply to procedures actually performed by users, *ie.* is Daedalos able to construct a procedure from a typical trace?
3. What sorts of generalization heuristics would help Daedalos correctly match situations in actual traces of the sample problems from Chapter 1?
4. Is inescapable "noise," such as coincidental, irrelevant contacts between objects, too widespread to be ignored?
5. Should Basil continuously monitor his sensory feedback, rather than just before and after an action, *ie.* does this sampling introduce irrelevant postconditions?

If the answers to these questions suggest that the system is indeed feasible, further implementations will proceed. Phase 1 has interaction between Daedalos and Basil, but without a built-in constraint solver. Daedalos will print predictions in a text window attached to the A.Sq display. The teacher acts as constraint solver by carrying out actions she accepts. Phase 2, including a solver, will serve as a prototype system suitable for testing alternative generalization methods.

The hardware and software support required for each phase is the same. The usability of A.Sq and Basil depends upon a high-speed graphics processor and a mouse or tablet input device. Basil and Daedalos benefit from built-in memory management to support complex yet ephemeral intermediate representations of data. The system overall has been designed as a group of entities cooperating in response to events. The Macintosh computer running object-oriented ExperCommonLisp provides a suitable operating environment. In Phase 0, A.Sq and Basil run as a single process in the Lisp environment; Daedalos is run separately. A.Sq and Basil are defined as Lisp objects that send each other messages. Other important components, such as user interface devices and graphical objects, have their own class definitions. This object-oriented coding style could take advantage of a multi-processor architecture.

The user interface was designed using the InterfaceBuilder, a commercial user interface management system based on the same object-oriented Lisp. It provides a graphical editor for windows and pull-down menus and a run-time interface to the Macintosh ToolBox. It

traps user-generated events and translates them into messages it sends to the appropriate modules of the application.

4.2 A.Sq

The user of Phase 0 A.Sq will find it a Spartan imitation of MacDraw. Its primitives and operators are as described in Chapter 3. This implementation of A.Sq is adequate for representing the problems described in Chapter 1.

The user interface provides tools to draw and transform boxes and lines by rubberbanding. The application has no interface to files and the display window cannot be panned or zoomed. The A.Sq “panel,” shown in Figure 4.1, consists of a row of pull-down menus, a menu of drawing tool icons, and a drawing pad. The cursor is sensitive to this division of the display region: when moving over menus it appears as an arrow; on the pad it resembles the currently selected drawing tool. Menus, objects and locations are selected by depressing the mouse button. Drawing lines and boxes requires two selections (of end-points or corners); the button is held down after the first and released after the second.

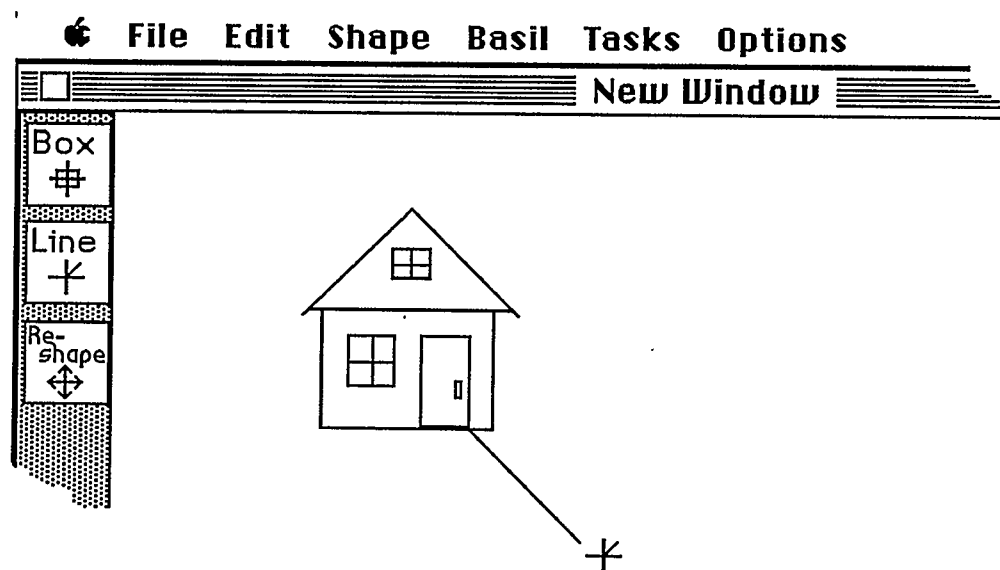


Figure 4.1 A.Sq user interface

Pull-Down Menus. The bar at the top of the screen contains six pull-down menus. A mouse event in this region invokes one of the A.Sq methods attached to individual menu items. There are six pull-down menus ordered as follows from left to right on the menu-bar:

File

file operations (currently none) and Quit, the command to exit A.Sq.

Edit

the Cut operation to remove an object from the display list; and Undo, to reverse the effects of the previous operation

Shape

same as the iconic menu of drawing tools (see below).

Basil

commands to delimit the teaching session and control Metamouse (see below).

Tasks

names of procedures learned by Basil; not supported in Phase 0.

Options

control parameters for A.Sq; currently, only Show Handles is provided (see Feedback below).

Basil. Since the Phase 0 instructional paradigm is teaching-by-demonstration and rotation is not supported, the Basil menu currently contains only two items. Begin Lesson (which toggles to End of Lesson when selected) delimits a single teaching session. Basil appears on screen below the drawing icons and will observe the teacher's actions until put to sleep or told the lesson is over. Take a nap (which toggles to Wake up, Basil!) temporarily suspends Basil's attention.

Drawing Tools. Selecting one of the icons at the left of the panel (or under the Shape menu) sets the current mode of operation on the drawing pad, which persists until changed by another selection. There are three options: Box and Line modes create new objects by rubber-banding; Reshape permits objects to be picked and transformed. The cursor takes on the current tool's shape.

Drawing Pad. Picking a point inside this region commences a drawing operation. A.Sq must also bring itself and Basil (if awake) up to date:

Handle Mouse-Button Down in Drawing Pad (Location)

```

CurrentPoint ← Location
Inform Basil of Operation commencing at CurrentPoint
Invoke Operation bound to Pad:
  Box / Line:
    CurrentObject ← create and install object in DisplayList
    CurrentPoint ← location at which mouse-button is released
  Reshape:
    CurrentObject ← Pick from DisplayList at Location
Inform Basil of operation completed and updates to CurrentObject, CurrentPoint

```

To draw a new box for example, the teacher 1) places the cursor on the pad where she wants one corner to lie, 2) picks this point by pressing the mouse button, 3) sizes the box by moving the mouse while keeping the button pressed and 4) picks the opposite corner by releasing the button. User action 2 invokes the *Handle Mouse-Button Down in Drawing Pad* routine. It immediately informs Basil that a location has been picked; if Basil is awake and not already at CurrentPoint, he will move there. This move is itself a step in the program. User action 3 is monitored by the routine to edit a box by rubber-banding. User action 4 returns control to the pad handler, which installs the box in the DisplayList and activates its handles so that the teacher may immediately re-edit the box if she wishes. The handler informs Basil of the cursor's new position and the box's address. Basil moves to the opposite corner of the box, grasps it, and records the drawing operation.

Pick and Transformation. A.Sq employs a gravity pick with cyclic disambiguation. If the current operation is Reshape, *Handle Mouse-Button Down in Drawing Pad* tries to update CurrentObject to some object that lies near CurrentPoint. "Near" means that one of its edges or handles is no more than 3 pixels away. Several objects may lie near enough to be selected, but only the first found in DisplayList is picked. It is moved to the end of DisplayList to give other objects higher priority for the next pick. To disambiguate a pick the user need only repeat the mouse-button press.

Only CurrentObject can be edited. Each of its handles is bound to a method for transforming the object when that handle is moved (see §3.1). The handles of CurrentObject become active regions of the display, with priority over the drawing pad. Thus when the mouse-button is pressed within the extents of the handle, the event manager invokes the handle's method rather than the pad handler.

Cut and Undo. Only CurrentObject can be deleted from the display; the user may select Cut from the Edit menu or hit the backspace (delete) key. The deleted object is moved from the display list to a deletions list and CurrentObject is set to nil. Basil is informed of the cut and updates his sensory record accordingly.

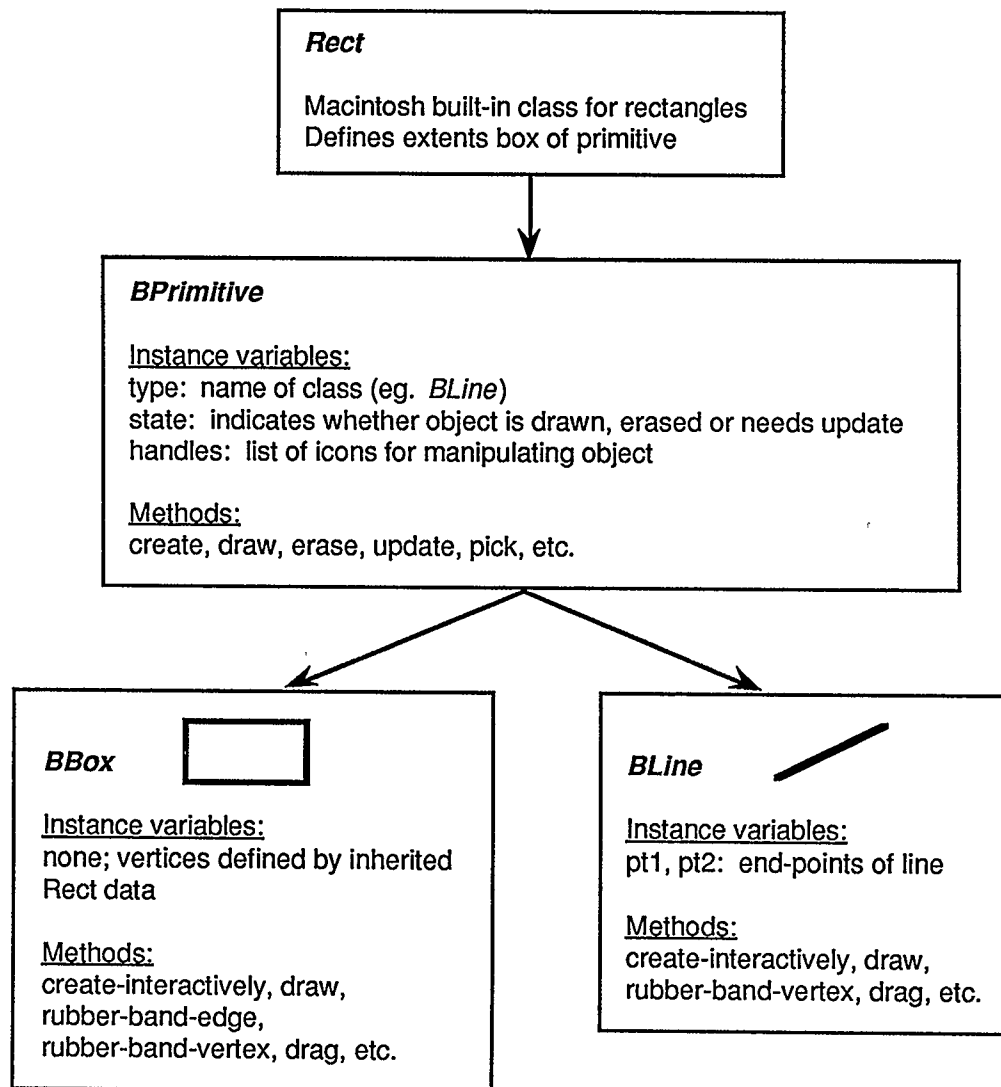


Figure 4.2 Class hierarchy of A.Sq graphical objects

The user can reverse the most recent drawing, transformation or cut by selecting Undo. A.Sq's undo is self-reversing rather than regressive. Basil, upon hearing of an undo, marks or unmarks as undone the most recent step of the action trace; Daedalus ignores an undone step.

Feedback. Apart from the graphical display of objects, A.Sq provides several forms of graphical feedback. As mentioned above, the cursor icon matches that of the current drawing tool. Since the user is assumed to be focussed upon the cursor, CurrentPoint is not explicitly displayed. When it does matter, as the anchor point during rubber-banding, or as the current location of Basil, its display would be superfluous.

CurrentObject is indicated by displaying its handles filled in black (handles of other objects are hidden). If the ShowHandles option is selected, handles are shown in outline whenever the cursor moves near an object. This helps the user locate important points on objects, such as centers, that are distinguished only by handles. The feedback routine is run when the processor is idle and at regular intervals during rubber-banding. It invokes the same proximity search as Pick, but with a slightly larger tolerance and permission to highlight more than one object; moreover, boxes are selected by extents overlap.

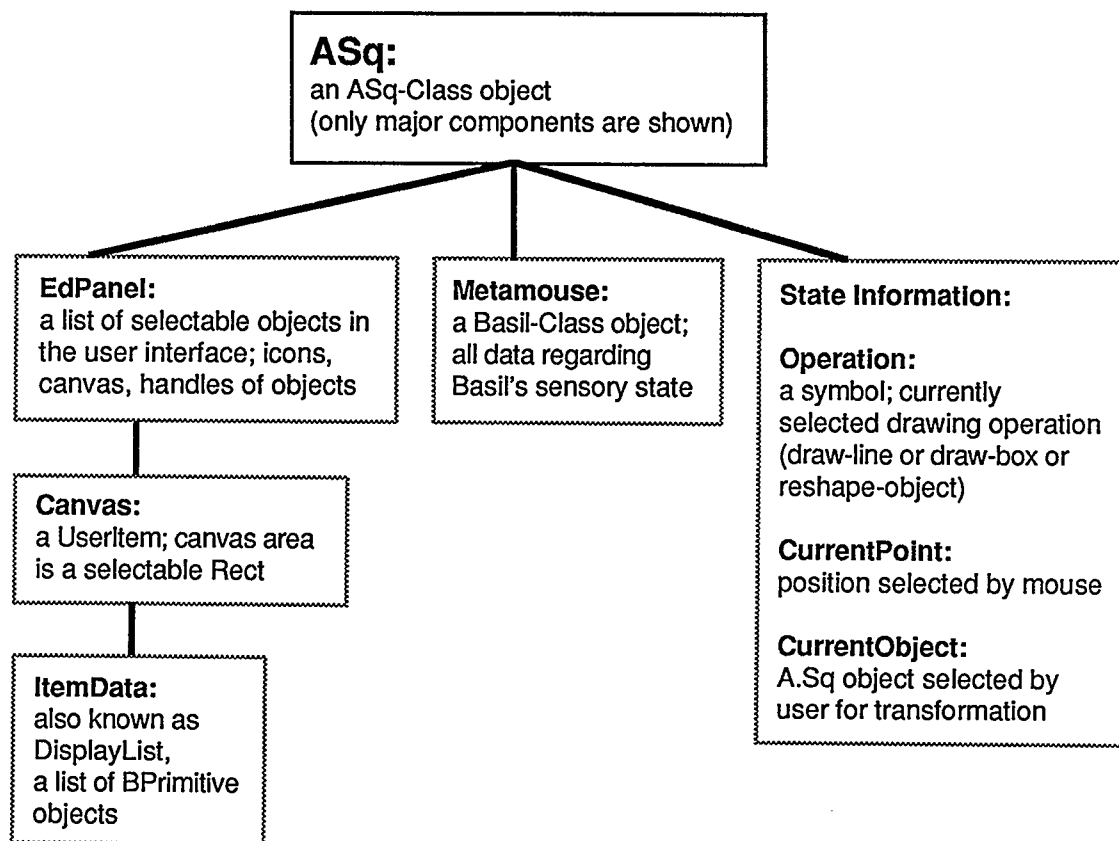


Figure 4.3 A.Sq system data structure

Data Structures. Phase 0 A.Sq has line and box primitives only. Figure 4.2 illustrates the hierarchy of graphical object classes: each subclass has its own methods for rubber-banding, transforming, displaying and picking. Objects are collected into DisplayList from which the event manager updates the graphics window, and the Pick and ShowHandles routines select objects. An overview of the A.Sq data structure is shown in Figure 4.3; profiling information for such features as the size of object gravity fields is not shown.

4.3 Basil

Basil 0, the Metamouse implemented for Phase 0, meets the requirements for teaching by demonstration. Basil 0's sensory feedback and memories follow the Chapter 3 specification, except for the omission of heading.

Data Structure. Basil is an ExperCommonLisp object that contains the following slots:

Activity-state

flag indicating whether Basil is currently awake or asleep; determines his response to messages from A.Sq (when asleep, responds only to "begin lesson", "end of lesson" and "wake up")

Icon

pointer to a Basil-Icon object

Application

pointer to A.Sq, so that Basil can access the DisplayList, etc.

Current-position

Basil's reference location at which touch and grasp are sensed; the screen-coordinates of the snout on Icon

Previous-position

value of Current-Position prior to last movement of Icon, used to compute distance moved; could use position recorded in postconditions of Previous-action

Touching

list of touch-predicates, naming points of intersection between Basil's snout and other objects

Grasping

single-element list of touch-predicates describing Basil's relation to CurrentObject; implemented as a list for uniformity with Touching & Indirectly-Touching

Indirectly-Touching

list of touch-predicates giving relations between object in Grasping and other objects

Action-trace

list of action-steps recorded by Basil

Current-action

action-step node being instantiated by current A.Sq operation

Previous-action

previous value of Current-action, the last node in Action-trace, from which Basil searches for recurrence of objects

Objects-Created

medium-term memory; list of pointers to A.Sq primitives drawn during teaching trace

Objects-Transformed

medium-term memory; list of pointers to A.Sq primitives whose handles were dragged during teaching trace

Body. Basil 0 learns programs that consist of moving, dragging, and drawing. The restriction to teaching by demonstration makes direct manipulation of the Basil 0 icon unnecessary. The Basil menu, described above, contains no commands for moving Basil. Nonetheless, it is desirable to show Basil that his position (the location of his snout) should not change between certain operations — for example, when drawing a sequence of connected lines. Moreover, when Basil awakens from a nap, he may need to be told to grasp what he is touching. Thus, rather than have the teacher try to place the cursor exactly at Basil's snout, the Basil 0 icon is treated as a moveable button — a simpler version of the original Metamouse device. In keeping with the demonstration paradigm, the method attached to the icon invokes the drawing pad handler, but uses Basil's snout rather than the cursor's position as the location.

Basil 0 delimits actions by observing *mouse-down* (teacher presses mouse-button) and *mouse-up* (teacher releases button) events filtered through A.Sq's drawing pad and handle server routines. Basil interprets a *mouse-click* event (button pressed and released immediately) as a *move* to CurrentPoint. A *mouse-drag* event (mouse moved while button held down) is a *draw-line*, *draw-box* or *drag* (transform object) as determined by the current drawing mode. When informed by the server that an action has commenced, Basil sets up a new action-step — and if necessary interposes a move step to reach the starting point:

Basil Informed of Action Commencing (Operation, Location)

```

Previous-action ← Current-action
Current-action ← Create new action-step data structure
Generalize and record Current-action's preconditions
    {based on Previous-action's postconditions}
If Operation is not Reshape and Location ≠ Current-position
    {Basil must move to starting point of operation}
    Current-position ← Move to Location
    Record Action (move) Path and Distance
    Generalize and record postconditions of move to Location
    Append Current-action to Action-trace
    Current-action ← Create new action-step data structure
    Generalize and record Current-action's preconditions
        {based on move's postconditions}
else do nothing {Basil is ready for operation}
return to A.Sq and await completion of Operation

```

Once the teacher's action (mouse event) is completed the server informs Basil, who updates his position, touch and grasp feedback and records the operation and its postconditions:

Basil Informed of Action Completed (Operation, Location, Current-Object)

```

Current-position ← Move to Location
If Current-Object is not nil, grasp it
Record Operation, Path and Distance to Location
Generalize and record postconditions of Current-action
Append Current-action to Action-trace

```

In Phase 0 the action-step node is appended to an Action-trace which is analyzed and run through Daedalos after the teaching session; in Phase 1, it will be passed on to Daedalos immediately.

Sensory Model. The Phase 1 routine to justify and record postconditions mentioned above would 1) note Basil's current sensory feedback; 2) perform object generalization; 3) classify feedback to determine whether sufficient constraint has been given to explain the action's parameters; and, if need be, 4) ask for justification. The Phase 0 routine performs step 1 only; filter programs run after the interactive process perform steps 2 ... 4. The sensory capabilities implemented in Basil 0 are exactly those described in Chapter 3, except that heading is omitted since Basil does not rotate.

Touch relations are represented as a pair of *Object-Info* records. An *Object-Info* record associates object and part information with a *variable-reference*:

Address

pointer to the object, an A.Sq primitive in DisplayList

Part-Name

name of part of object in the relation

Part

pointer to the part's data structure, if any (nil if Part-Name is "line-segment")

Variable

pointer to the variable-definition associated with this object; derived from a search through Basil's memories

Valuation

function that instantiates Variable (*ie.* sets Address) or leaves it unchanged

Basil 0 observes touch relations between itself and objects in DisplayList, and records actual values for each Address, Part-Name and Part. A filter program to induce variables, described in §4.5.1, sets Variable and Valuation according to the rules of generalization. Since Phase 0 lacks a constraint solver, Valuation merely names an operation that the researcher/user must perform manually in order to find a value for Object (see §4.4).

Recall the three types of touch relations defined in §3.2.2 — point-on-point, point-on-line, and line-crosses-line — and the rule that only the most specific touches are recorded. The *Find-Touches* routine looks for touches of each type in order. Although an object's gravity field is implemented by enlarging its bounding box, there is no need to extend line segments, since end-points lie within handles. Moreover, collinearity need not be checked, since overlapping line segments will have handle-handle or handle-segment touches.

4.4 Daedalos

The Daedalos learning algorithm implemented for Phase 0 processes an action trace translated into textual form by the researcher. It matches steps in the action trace to steps in the program under construction, without a representation of particular graphical objects. Generalization matching is omitted from Phase 0; steps are matched by syntactic identity. Despite these limitations, a session with Daedalos 0 can be used to evaluate Basil's generalization of objects by creating variables and of conditions by dropping terms. Of course, a session with Daedalos also tests the learning algorithm itself.

Daedalos elicits the action trace from the teacher step by step. An action-step data structure is a list of three items (precondition, operation, postcondition). The internal

syntax of each item does not matter provided it is consistent with items against which it may be matched. Daedalos prints predictions and asks for confirmation.

The learning algorithm differs from the original design only in that `OptionsLimit` and `LinkAttemptsLimit` are fixed at 1; `ConfirmsLink` is still variable. It is coded in `ExperCommonLisp`. An object of class *PgmNode* represents a program step or node in the directed graph. The node comprises:

Action-info

pointer to the action-step data structure

Predecessors

list of *PgmNodes* whose `Successors` lists include this node

Successors

list of *PgmNodes* whose `Predecessors` lists include this node

Mark

flag indicating whether this node has been visited during a given search of the graph; reset before each invocation of Find-and-confirm-join (see §3.4.2)

The program graph is initialized to a connected pair of dummy “Start” and “Stop” *PgmNodes*. A sequence is created by inserting new nodes after Start and before Stop. A branch is opened by appending a new node to the `Successors` of `LastAcceptedStep`. A branch or loop is closed by appending the matched step (called the “join step”) whose successors were successfully predicted to the `Successors` of `LastAcceptedStep`.

Recall from §3.4.2 that the search for a join step gives priority to nodes preceding and near `LastAcceptedStep` (Daedalos is biased towards hierarchical task decomposition) but potentially selects from the entire graph. Ideally the Find-and-confirm-join function would initiate a bounded search along chains of `LastAcceptedStep`’s predecessors, followed by a similar search along successors, followed by a total graph search. The Phase 0 version crudely approximates this by recursively searching predecessors, followed by successors. It thus gives priority to “non-local” connections over nodes near to but succeeding `LastAcceptedStep`. This should not matter since the graphs of programs for tasks given in Chapter 1 are small and offer few matching candidates (typically no more than 1) — especially since Phase 0 matching involves no generalization.

4.5 Generalization

In order to abstract traces into program graphs, the Phase 0 system performs two kinds of generalization on individual actions: 1) variables replace individual objects; 2) constraints are dropped from pre- and postconditions. These are described in §3.3. Since Phase 0 does not support matching by generalization in Find-and-confirm-join, actions are generalized by filter programs that process the entire action trace before it is handed on to Daedalus. The first filter program induces variables; the second classifies constraints and generalizes pre- and postconditions. The filters must run in this order since the occurrence of variables affects the classification of constraints.

4.5.1 Variables

The variable filter finds multiple occurrences of a given object in the action trace and associates them with common variables according to the rules outlined in §3.3.1. It searches recent steps of the action trace and the list of objects created by Basil. It sets the fields of Object-info records in touch relations (see §4.3 : Sensory Model) that correspond to variable-references. Variable-definitions are maintained in a global symbol table. The implementation introduces one new valuation function, *Transformed*, to facilitate investigation of the usefulness of remembering transformed objects (see §3.2.3).

Induce-variables. The filter algorithm is given below. Note that all the information needed to construct the memories of objects created and transformed is contained in the trace, hence the system need not produce these beforehand.

```
Induce-variables (Trace)
  set up empty Created and Transformed lists
  for each step S in Trace
    for each touch relation T in postcondition of S
      for each object X in T
        Get-variable-&-valuation-fn for X,
          searching Trace, Created and Transformed
```

Get-variable-&-valuation-fn. This routine searches (and constructs) memories to identify recurrences of objects and variables. The *Action-Trace* contains variable-references, *Created* contains pointers to variable-definitions, and *Transformed* contains object addresses. In keeping with the rules for inducing variables, *Action-Trace* is searched first, followed by *Created* and *Transformed*. If the current action creates an

object, no search is performed and a new variable is defined. If the search fails, a new variable is defined and a new item added to *Transformed* if appropriate. *Object* is an Object-info record.

. *Get-variable-&-valuation-fn* (*Object*, *Action-Trace*, *Created*, *Transformed*)

```

    if Current-Step.Operator is Draw-box or Draw-line
        Object.Variable ← New-variable in Created list, bound to Object
        Object.Valuation ← "Create"
    else if V ← Variable-found in Action-Trace bound to Object.Address is not null
        Object.Variable ← V
        Object.Valuation ← "Same"
    else if Object.Address occurs as binding of some variable V in Created
        Object.Variable ← V
        Object.Valuation ← "Same"
    else if Object.Address occurs in Transformed list
        Object.Variable ← "Trans" {don't individually name}
        Object.Selector ← "Transformed"
    otherwise {no previous occurrence of Object found}
        Object.Variable ← New-variable bound to Object
        Object.Valuation ← "Find"
    {remember any object that has been transformed}
    if Current-Step.Operator is Drag
        append Object.Address to Transformed

```

Variable-found. The routine to search back through the trace for a previous occurrence is governed by a parameter *LookbackLimit*, the maximum number of steps to be examined (conceptually, the range of Basil's short-term memory). It checks each Object-Info record in a given Trace step's postcondition. Note that the search begins with *Current-Step*; this ensures that all occurrences of an object in a given step are associated with the same variable.

Variable-occurrence (*Trace*, *Object*)

```

    V ← null
    for lookback ← 0 through LookbackLimit, until found is signalled
        SearchStep ← Current-Step – lookback
        if Object.Address occurs as binding of some variable V in SearchStep
            signal found
    return V

```

New-variable. When no previous occurrence exists, a new variable is created and bound to the current object. The initial binding in no way restricts future bindings. The

rules of touch-relation matching guarantee that a variable is always bound to objects of the same type (Box or Line).

4.5.2 Constraints

Recall that a condition is simply a conjunction of constraint predicates generalized from Basil's sensory feedback. Given a trace in which conditions have already been generalized by introducing variables, a second filter generalizes them further by dropping predicates. The decision regarding which predicates to drop is made by classifying them according to their ability to distinguish a situation (precondition) or constrain the parameters of the operation they govern (postcondition). The classes were described in §3.3.2. The mappings of these classes onto the decision to keep or discard constitute generalization heuristics. The filter permits different mappings for experimentation with heuristics.

The constraint generalization filter considers each trace step in turn. Precondition predicates and their classification are inherited from the previous step's postcondition (the first step has no precondition), but the discard heuristic is different.

Postconditions are simplified by discarding trivially uninformative predicates, and generalized by ignoring others that may or may not be informative. The postcondition generalizer rationalizes the selection of predicates to achieve sufficient constraint before attempting to generalize.

The hierarchy of classes to which constraint predicates are assigned is based on the model in §3.3.2 and is given below:

1. Effected
 - a. Determining
 - b. Strong
 - c. Weak
 - d. Crossing
2. Unchanged
3. Overdetermined
 - a. Strong
 - b. Weak
 - c. Crossing
 - d. Sustained
 - e. Trivial

4. Teacher-given
 - a. Input
 - b. Constant

The postcondition filter classifies and reclassifies predicates, checks for sufficient justification and finally discards some predicates as required by the generalization heuristics currently in force. If touch predicates do not provide sufficient justification, the filter asks the teacher to reclassify position and distance predicates. If a determining constraint is present, other constraints are reclassified as overdetermined. The algorithm follows:

Generalize-Postcondition (PostCond, PreCond, Operator, Path, Heuristics)

{for class definitions see below}

Set-Class of PostCond *position*, *distance* and *path* as Trivial

Classify PostCond *touch* predicates into subclasses of Effected or Unchanged using knowledge about Operator and PreCond - to - PostCond transitions if any item is in class Determining then

Reclassify items in Effected or Unchanged to Overdetermined

if **Sufficient-justification** of Operator,

based on classification of PostCond touch predicates, then

if no item is Determining, **Reclassify** *path* as Weak

Discard some Overdetermined predicates according to Heuristics

else if **Reclassify** *position or distance* to class Teacher-given succeeds

Discard all other PostCond predicates

otherwise

signal failure to justify action

Classify. Initially, all touch predicates are assigned to subclasses of Effected and Sustained; all other predicates are assigned to Trivial on the assumption that touch constraints will prove sufficient. The rules for selecting a subclass of Effected or Sustained are given below. The main features of interest are the type of relation (point-to-point, point-on-line, line-crosses-line), its relation to Basil (grasp, direct or indirect touch), and the role the object plays, as expressed in the valuation function of its variable. T is the touch relation to be classified; U is some other touch relation; P and Q are Object-Info records.

Trivial

T = grasp (Basil : P)

and Operation is one of {Draw-line, Draw-box}

ie. Basil is grasping the object just drawn

Determining

T is one of {grasp (Basil : P), touch (Basil : P)}
 and P.Part is a handle
 and P.Valuation is one of {Created, Same}
 and T fails tests for Trivial and Unchanged
ie. Basil is moving to an object already identified to him

or

T = touch (P : Q)
 and grasp (Basil : P)
 and P.Part is a handle and Q.Part is a handle
 and Q.Valuation is one of {Created, Same}
ie. Basil moves to achieve point-to-point touch between the object in grasp and another already identified to him

Strong

T is one of {grasp (Basil : P), touch (Basil : P)}
 and P.Part is a handle
 and P.Valuation = "Find"
ie. Basil is moving to a handle of an object found by solving constraints

or

T = touch (P : Q)
 and grasp (Basil : P)
 and P.Part is a handle and Q.Part is a line-segment
 and Q.Valuation is one of {Created, Same}
ie. Basil moves to achieve point-to-line touch between the object in grasp and another already identified to him

Weak

T is one of {grasp (Basil : P), touch (Basil : P)}
 and P.Part is a line-segment
ie. Basil is moving to a line

or

T = touch (P : Q)
 and grasp (Basil : P)
 and P.Part is a handle and Q.Part is a line-segment
 and Q.Valuation = "Find"
ie. Basil moves to achieve point-to-line touch between the object in grasp and some other object found by solving constraints

Crossing

T = touch (P : Q)
 and P.Part is a line-segment
 and Q.Part is a line-segment
ie. Basil senses an indirect touch between lines or edges of boxes

Unchanged

T matches some U in precondition
ie. relation has not significantly changed as a result of the action

Overdetermined

class of T is not Determining
 and $\exists U$ in current precondition whose class is Determining
 {select subclass of Overdetermined that matches T's previous classification}
*ie. a determining constraint has been found; all others are reclassified as
 overdetermined*

Sufficient-justification. After classifying all touch constraints, the generalization filter checks for sufficient constraint to enable determination of action parameters. If any predicates of classes Determining, Strong or Weak are present, the action is justified. Otherwise the program asks the teacher (as Basil would do) for a reclassification of position or distance as Input or Constant; the touch predicates are all reclassified as Overdetermined.

If there is sufficient justification in touch predicates but none is Determining, path is reclassified as Weak so that the constraint solver could use this to further constrain the search for contacts.

Discard. The default generalization heuristic for Phase 0, represented as a list of classes to be ignored, discards all Overdetermined items. Better modeling may be achieved by retaining some Overdetermined constraints, such as Sustained; in any case all Trivial items should be discarded.

Preconditions. Precondition predicates inherit the classification of the previous corresponding postconditions. Phase 0 provides a separate generalization heuristic for preconditions, but currently it covers the same classes as the postcondition heuristic. When predicting actions, Daedalos ignores preconditions altogether if postconditions are attainable. Should the teacher reject the prediction, the preconditions are marked "necessary" and cannot be generalized again.

Path. A path constraint augments non-determining touch constraints. It is useful however to generalize path somewhat when matching and predicting actions. Currently Basil distinguishes 8 paths along octant boundaries in 2-D space. The boundaries themselves constitute the four axes {vertical, horizontal, SWNE, SENW}. A pair of opposite but (roughly) collinear paths may be generalized to the nearest axis.

4.6 A Worked Example

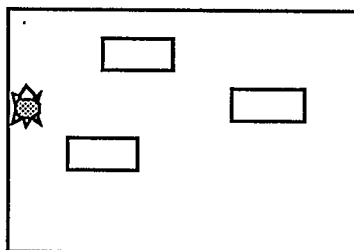
A teaching session of the box-to-line task was presented in Chapter 1. The series of figures below illustrate the generalization of each step in the action trace by the filter programs. The trace is one of those gathered for use in the studies described in Chapter 5. Daedalos induced the program graph illustrated in Figure 3.4.

The number at the upper left of each frame corresponds to a step number in Figure 3.4. At the upper right is a description of the teacher's and Basil's actions at that step. Below each frame is a description of the action-step record. Sensory feedback is shown as recorded after generalization. Feedback items are of the form (sense : data *class*), where sense is the type of feedback (eg. touch), data is the generalization of Basil's observation, and *class*, printed in italics, is the constraint classification. Feedback items ignored due to generalization are in plain type; items considered relevant are in bold. Variable-references for each step are listed below the postconditions. Each entry is of the form (variable name : valuation function).

Note that the frames are not scaled-down snapshots of the Macintosh display: the real drawing pad is somewhat larger in relation to Basil. The printed representation of touch relations of the form "grasp (Basil : Obj.Part)" has been abbreviated to "grasp (Obj.part)".

Step 0. The teacher initiates the lesson and Basil appears at his standard position at the left of the display. Basil records no action or feedback at this point.

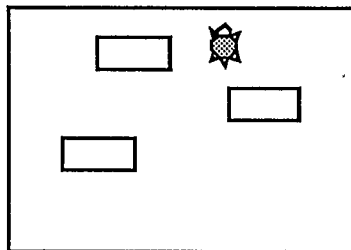
0.



Starting Position;
Teacher has selected "Time for a Lesson"
from the Basil menu

Steps 1, 2. The teacher draws the guide-line. Both end-points are classified by the teacher as inputs. In each case the generalization filter detects the lack of tactile constraint and asks the teacher to explain Basil's current position. Note that step 2's preconditions are inherited from step 1's postconditions: this of course does not mean that Basil would expect another input position, but that he should be at the position established by step 1 — a trivial precondition but potentially useful to prevent incorrect formation of a loop.

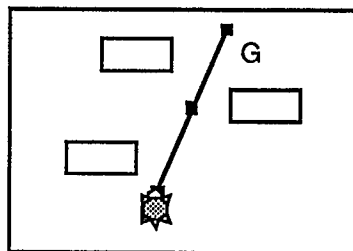
1.



Teacher begins line stroke; Basil moves to position, asks why; Teacher answers that point is a parameter

Preconditions : none
 Operation : Move
 Path : *overdetermined*
 Postconditions : **position input**

2.

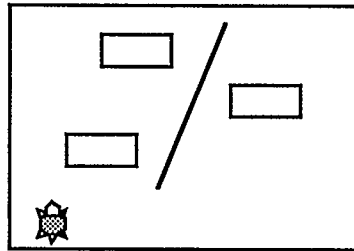


Teacher draws guide line; Basil requests rationale for endpt; Teacher calls it input parameter

Preconditions : = Step 1 Postconditions
 Operation : Draw-line
 Path : *overdetermined*
 Postconditions : **position input**
 grasp(G.endpt) *trivial*
 Variables : G : create

Steps 3, 4. The teacher draws the sweep-line. The only difference between this and the guide-line, as far as Basil is concerned, is that its end-points are constant.

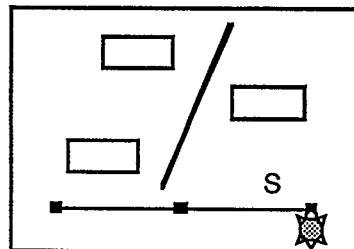
3.



Teacher places anchor pt for sweep line;
Basil asks rationale; Teacher replies point is
constant

Preconditions : = Step 2 Postconditions
Operation : Move
Path : *overdetermined*
Postconditions : **position constant**

4.

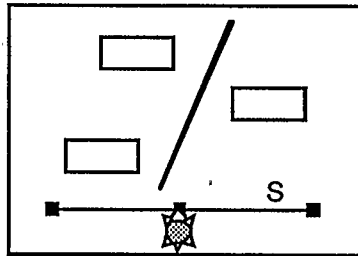


Teacher draws sweep line; Basil requests
rationale; endpt is constant

Preconditions : = Step 3 Postconditions
Operation : Draw-line
Path : *overdetermined*
Postconditions : **position constant**
 grasp(S.endpt) *trivial*
Variable : S : create

Steps 5, 6. The teacher grasps the sweep-line at its mid-point and drags it upwards until contact with a box. Since the sweep-line was created by Basil in the previous step, it is a known object. Moreover the move to grasp its mid-point establishes a point-to-point touch relation. Therefore *grasp(S.midpt)* is determining and all other postconditions are overridden. In step 6 the drag to establish contact between the sweep-line and a box has only weak point-to-line constraints. Hence the upward path is reclassified as a weak constraint.

5.



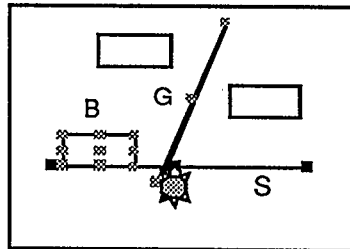
Teacher begins dragging S; Basil follows

```

Preconditions : = Step 4 Postconditions
Operation    : Move
Path         : overdetermined
Postconditions : position overdetermined
               grasp(S.mid) determining
Variables    : S : from step -1 (i.e., previous step)

```

6.



Teacher drags S to contact a box

```

Preconditions : = Step 5 Postconditions
Operation     : Drag
Path          : upwards weak
Postconditions : position overdetermined
               grasp(S.mid) trivial
               touch(S.line: B.bottom.left)    weak
               touch(S.line: B.bottom.mid)     weak
               touch(S.line: B.bottom.right)   weak
               touch(S.line: G.line)           crossing
Variables     : B : found by solver
               S : from step -1
               G : from step -4

```

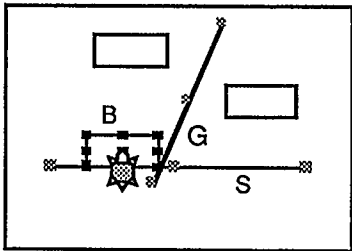
Steps 7, 8. The teacher grasps the box by its center and drags it horizontally to the guide-line. Since the box was encountered in the previous step, grasping its center determines the move in step 7. The contact between box and guideline established by step 8 is only point-to-line, so the drag operation has only weak constraints; hence the rightward path is promoted to weak.

7.  Teacher picks box, Basil follows

```

Preconditions : = Step 6 Postconditions
Operation    : Move
Path         : overdetermined
Postconditions : position overdetermined
               : grasp(B.mid) determining
               : touch(S.line: B.bottom.left) overdetermined
               : touch(S.line: B.bottom.mid) overdetermined
               : touch(S.line: B.bottom.right) overdetermined
Variables    : B : from step -1
               : S : from step -1

```

8. 

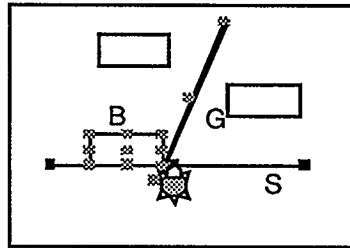
```

Preconditions : = Step 7 Postconditions
Operation     : Drag
Path          : rightwards weak
Postconditions : position overdetermined
               : grasp(B.mid) trivial
               : touch(S.line : B.bot) unchanged
               : touch(B.br : G.line) weak
Variables     : B : from step -1
               : G : from step -2
               : S : from step -1

```

Steps 9, 10. The teacher returns to the sweep-line and drags it up to the next box. Step 9 patently repeats step 5. The preconditions at step 10 differ from those at step 6; Daedalos ignores them and correctly predicts step 10 as a repeat of step 6.

9.

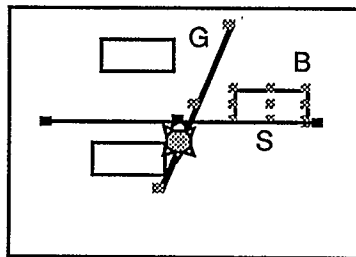


DAEDALOS: MATCH 5

Teacher picks S; Basil follows

Preconditions : from Step 5, *generalized*
 Operation : Move
 Path : *overdetermined*
 Postconditions : position *overdetermined*
 grasp(S.mid) *determining*
 touch(S.line: B.bottom.left) *unchanged*
 touch(S.line: B.bottom.mid) *unchanged*
 touch(S.line: B.bottom.right) *unchanged*
 touch(B.bottom.right: G.line) *unchanged*
 Variables : S : from step -1
 B : from step -1
 G : from step -1

10.



DAEDALOS: GENERALIZE 6, PREDICT

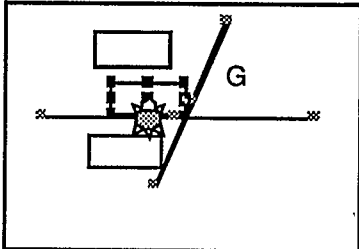
Basil predicts drag S up to a box;
 Teacher accepts

Preconditions : from Step 6, *generalized*
 touch(S.line: B.bottom.left) *optional*
 touch(S.line: B.bottom.mid) *optional*
 touch(S.line: B.bottom.right) *optional*
 touch(S.line: G.line) *optional*
 Operation : as in Step 6
 Path : "
 Postconditions : "
 Variables : "

Steps 11, 12. Daedalos correctly predicts that the box encountered at step 10 is grasped and dragged to the guide-line. In order to predict step 12 the path must be generalized to "horizontal." Note that step 12 establishes a point-to-point contact between the box's bottom-right corner and the guide-line's mid-point. When predicting actions Daedalos is concerned only with establishing the constraints given in the program step; additional constraints that happen to be established are ignored. In the figures here they are marked "incidental."

11.  **DAEDALOS: PREDICT 7**
Basil predicts pick box

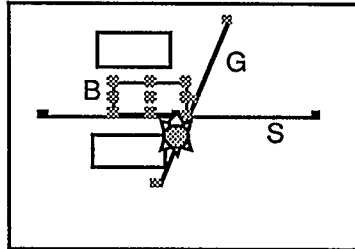
Preconditions : as in 7
Operation : "
Path : "
Postconditions : "
Variables : "

12.  **DAEDALOS: GENERALIZE 8, PREDICT**
Basil predicts drag box to G, contact at lower right

Preconditions : as in 8
Operations : "
Path : *horizontal generalized*
Postconditions : as in 8
touch(B.mid.right: G.midpt) *incidental*
Variables : as in 8

Steps 13...16. Daedalus correctly predicts the next iteration through step 16.

13.

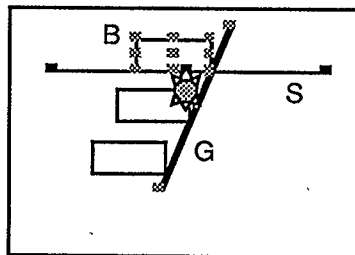


DAEDALOS: PREDICT 5
Basil predicts pick S midpt

Preconditions : as in 5
Operations : "
Path : "
Postconditions : as in 5, and
touch(B.mid.right: G.midpt) *incidental*
Variables : as in 5

Step 17. The teacher accepts Daedalus' prediction that Basil re-grasps the sweep-line as in step 5. But when Daedalus tries to predict another repeat of step 6, the constraint solver (*viz.* the teacher) cannot find a box. The prediction fails and Daedalus asks the teacher for the next action.

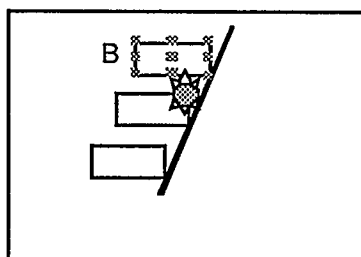
17.



DAEDALOS: PREDICT 5
Basil predicts pick S at midpt

Steps 18...20. The teacher deletes the sweep- and guide-lines. The cut operation need not have any postconditions but this exception is not modelled in the current version. At step 19 Basil encountered a box remembered only as "transformed." The window on Recent-Steps, set to 5, prevented the variable filter from finding a previous occurrence of this box in the trace.

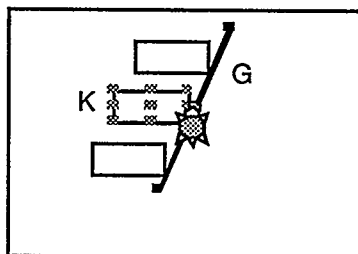
18.



Basil unable to perform "drag up to some box" as in step 7; Teacher resumes control; deletes S

Preconditions : as in 7, and
 unable to satisfy Postconditions of 7
 Operation : Cut
 Path : *irrelevant*
 Postconditions : position *overdetermined*
 touch (Basil.snout: B.bottom) *weak*
 Variable : B : from step -1

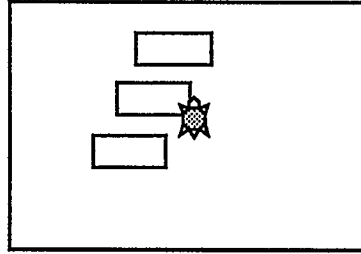
19.



Teacher picks G at midpt

Preconditions : = Step 18 Postconditions
 Operation : Move
 Path : *overdetermined*
 Postconditions : position *overdetermined*
 grasp(G.midpt) *determining*
 touch(K.mid.right: G.midpt) *strong*
 touch(K.bottom.right: G.line) *weak*
 Variables : K : previously transformed
 G : from step -2

20.

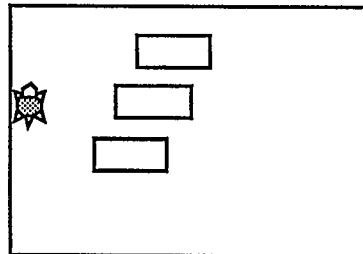


Teacher deletes G

Preconditions : = Step 19 Postconditions
 Operations : Cut
 Path : *irrelevant*
 Postconditions : position *trivial*
 Variables : G : from step -1

Step 21. The teacher selects “End of Lesson” from the Basil menu. Basil returns to his standard position. Daedalos prints out the program corresponding to Figure 3.4.

21.



End of Lesson.

Chapter 5

Three Empirical Studies

A system for programming by example is of course an open system — it has a teacher. Moreover it is in a real sense non-deterministic. The teacher provides three kinds of inputs: actions, input data when requested, and responses to predictions. Because the system may not be able to model every decision the teacher makes, it may find that the teacher disagrees with an apparently determined state transition. Therefore the system's performance cannot be predicted solely through analysis of its structure — it must be studied and assessed empirically.

Given an incomplete pilot implementation, how can we test its performance in conjunction with human teachers? This chapter describes three empirical studies on segregated components and abilities of the system. From these the performance of an actual system can be projected — not with certainty but at least credibly. These experiments constitute a feasibility study of the integrated system. Chapter 6 expands further on the conclusions that can be drawn.

The first study measures how quickly potential users learn to predict Basil's sensory responses and behavior, providing some indication of the teaching metaphor's comprehensibility. The second establishes the system's ability to induce procedures from graphical traces (rather than contrived symbolic input). The third compares learning performance with and without the teacher's criticism of predictions; this isolates an important element of interaction.

5.1 The Metamouse Metaphor

A critical aspect of a learning system is that the teacher understand its behavior [MacDonald 88]. Users of the system are given a metaphor (Basil the Metamouse) to help their understanding. Its suitability, measured as the ability of teachers to quickly learn to predict its behavior, can be studied apart from a working system. The pilot experiment described here, though not sufficiently controlled or naturalistic to be conclusive, provides evidence that the metaphor is easy to understand.

A number of potential users were given the brief description of Basil shown in Figure 1.7 and then asked to work through a self-study guide, available from the author. Typical questions depict a situation and ask the subject to predict Basil's response. Scores on each set of questions were recorded separately so that progress could be measured. The study guide provides correct answers after each set, to simulate system feedback. Subjects were asked not to refer back to previous questions, however.

The study guide contains approximately 55 questions, of several types:

- two graphical situations depicted (Basil in contact with objects); subject asked whether Basil matches these situations to each other (approx. 30 questions)
- two actions depicted, showing situations before and after; subject asked whether Basil matches these actions (approx. 20 questions)
- a sequence of actions; subject asked to differentiate those performed by teacher from those performed by Basil (1 question)
- a graphical task is specified; subject asked to create an algorithm that Basil could learn (3 questions)

The questions are arranged in the order given above, so that difficulty tends to increase as subjects become more experienced. The solution to each question is presented immediately after the subject has completed it.

The pilot experiment was run with five volunteer subjects, all computer scientists. The first subject, a Metamouse "expert," was given a preliminary version without an answer key. The data for this subject were discarded, but editing suggestions were incorporated into the next version of the questionnaire, given to the remaining subjects, who did not have prior knowledge of the workings of Metamouse. All were allowed to work at their own pace at a time of their choosing.

The first three groups of questions were graded for use in this study. Due to bad photoduplication, some questions had to be discounted; hence the total number of questions varies amongst subjects. Subjects' understanding of Metamouse at any point is measured as the ratio of the number of questions correctly answered to the total questions counted. This ratio is plotted for three subjects in Figure 5.1; perfect performance lies along the line of slope 1. An increase in slope represents improvement in performance.

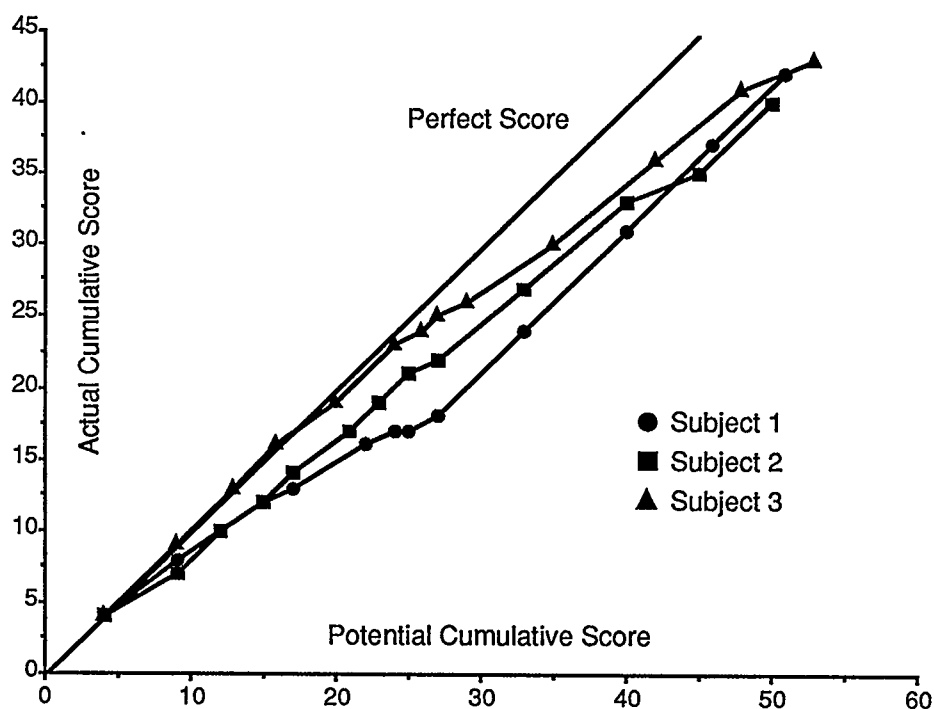


Figure 5.1 Performance of three typical subjects on Basil questionnaire

Recall that questions were presented in order of increasing difficulty. If the metaphor were unlearnable, one would expect to see a steady decline in performance (ever-decreasing slope in the graph). If difficult to understand, one would expect numerous errors in the early sets of questions, with at best a slow improvement. But if completely obvious, one would expect near-perfect performance from the beginning with no degradation. The actual plots maintain a fairly steady slope with some inflections. They show near-perfect performance initially, with occasional mistakes and difficult spots after which near-perfect performance is restored. This *suggests* that the “superficial” aspects of the metaphor — namely the rules that distinguish parts of objects and types of direct touch — are easily understood, while deeper aspects — the rules that govern action-matching and prediction — are less well understood but learnable.

A number of methodological deficiencies prohibit stronger conclusions from this experiment. Most of these problems are discussed in Chapter 6, but one particularly thorny issue is this. All of the subjects commented that a number of the questions were difficult. Some difficulties were due to bad photoduplication and have been eliminated from the data. Some clearly show in the data. Others however have been masked if the subjects thought

through the problems or luckily guessed. Since the number of questions in each sample set is small, such false positive data could seriously mask the degree of difficulty one is trying to measure. This lessens the significance of the results and the utility of a (necessarily small) questionnaire as opposed to real interaction.

5.2 Inducing Procedures

Consider the problem of testing the ability of a system that has no real performance component (in this case a constraint solver) to express what it has learned. The system can however express itself textually. Because its learning components (which select variables, generalize constraint descriptions and induce control structures) process symbolic data derived from the teacher's graphical input, a suitable way of assessing the system's ability to learn is to collect graphical traces and process their symbolic descriptions. Predictions given textually can be compared with what the teacher actually did, or with expectations based on a knowledge of the task. Performance of graphical actions through a constraint solver is not strictly necessary.

The pilot study tested the system's ability to generate a procedure that could produce correct sequences of actions covering the different situations given in the traces, but not necessarily matching the traces exactly, since they contained coincidental events, missteps and irrelevant variances in order of execution.

The system's goal is to learn procedures that are general (not complete), accurate (not perfectly correct) and minimally complex (not optimal). Generality achieved after each lesson was measured in terms of the ratio of actions correctly predicted to the total performed collectively by the system and the teacher in each trace. This ratio varied with the complexity of situations encountered in each lesson. Hence no normalization is useful; instead we get a rough measure of the rate at which Basil learns. Accuracy was measured as the ratio of accepted to rejected predictions. Complexity was measured as the number of edges in the program graph, which could be compared to the number in an "ideal" graph. The figures obtained for each lesson are given in Table 5.1.

Task	Trace #	Steps Performed in Task				Edges in Program Graph	
		Total	by Basil	Ratio	Rejected	Total	Growth
Box-to-Line	1	20	8	0.4	0	13	13
	2	24	24	1	0	13	0
	3	20	20	1	0	13	0
Picket Fence	1	35	12	0.34	5	22	22
	2	27	27	1	0	22	0
Connectivity	1	6	0	0	0	7	7
	2	6	6	1	0	7	0
	3	6	6	1	0	7	0
	4	6	6	1	0	7	0
	5*	4	1	0.25	2	11	4
	6	4	4	1	0	11	0
	7	6	6	1	0	11	0
	8	6	6	1	0	11	0

* variant of task: move one end-point rather than entire line

Table 5.1 Learning system performance

The researcher performed several different traces of three tasks in A.Sq with Basil activated. These were run through the variable and constraint generalization filters, giving augmented traces like that shown at the end of Chapter 4. The researcher fed the traces of each task into the Daedalus program in the order they were produced, to simulate incremental learning from multiple lessons. The researcher classified Daedalus' textual predictions as correct if the action and its conditions matched what his ideal model of the algorithm would have generated.

5.2.1 Box-to-Line

The first task was "box-to-line" as described in Chapters 1 and 4. The three traces presented different orientations of the guide-line and different arrangements and numbers of boxes. These variations were covered by generalization inherent in Basil's sensory model, by explicit generalization of constraints, and by the induction of a loop. The need to generalize a rightward path to a horizontal one could not be detected without a constraint solver, so this one generalization was done manually. As a result, Daedalus predicted the second and all subsequent iterations of the loop. Termination of the loop by failure could not be detected without a constraint solver, so the teacher simply rejected the last move of the sweep-line; this was not counted as a faulty prediction.

The first trace contained a coincidental contact when processing the third box. The sweep-line's mid-point touched the guide-line; hence Basil observed that the box was moved until its lower-right corner touched the mid-point. This overlapping of handles led to a "misstep" in which the teacher failed to pick the mid-point on the first try and had to repeat the pick before advancing the sweep-line. This "noise" in the trace did not trouble Daedalos, however. Since it had already learned the correct sequence to select a box, move it and advance the sweep-line, it predicted the same actions in this case, which the teacher accepted. The misstep was eliminated; hence the execution trace produced by the teacher and Daedalos working together was shorter than that produced by the teacher alone (see Table 5.1).

After the first lesson Daedalos was able to predict all actions in subsequent performances of the task. The learning system became competent at the "box-to-line" task as quickly as it could have. The program graph was judged to be minimal in complexity for this algorithm.

5.2.2 Picket Fence

Recall the "Picket Fence" task from Chapter 1. Boxes randomly scattered about the screen are moved onto a horizontal line such that the gap between them is constant. The algorithm uses a vertical sweep-line that moves left to right to select boxes. The first box is moved straight down to the guide-line; the gap-line is then attached to its right edge. Subsequent boxes are moved so that their bottom lies on the guide-line and their left edge touches the gap-line. The first point of the guide-line and the length of the gap-line are inputs.

Judging by the program size given in Table 5.1, this task was considerably more complex than box-to-line. It afforded more opportunity to vary the order of actions yet still accomplish the task. Boxes were translated to their final position along arbitrary directions, establishing the usefulness of generalizing path. Moreover, numerous coincidental contacts occurred. In particular, the sweep-line or the current box could be moved into contact with a box already transformed; this facilitated testing the potential usefulness of the "Transformed" attribute (see §3.2.3).

Despite the increase in complexity, Daedalos was able to construct a small program (though larger than the ideal) and was able to predict almost 1/3 of the actions in the first lesson, and all of the second.

Five of Daedalos' predictions during the first lesson were rejected: 1) that the gap-line would be of constant length like the guide-line (predicted because both have first points as inputs); 2) that the sweep-line's first point would be input (predicted because the input of first points of gap- and guide-lines formed a loop); 3) that the second box would be moved straight down to the guide-line; 4) that the third box would be grasped before the gap-line was advanced to the right of the second box (a bit of "bad teaching" caused this confusion!); and 5) in a related error that the sweep-line would be grasped again even though it was already in place. Once the correct actions were taught, Daedalos was able to run through the second trace without error. It is interesting to note that if Daedalos were required to match 2 steps before attempting to predict (instead of 1, the current setting of *ConfirmsLink*), none of these erroneous predictions would have been made.

5.2.3 Connectivity

The third procedure maintained a connectivity constraint: given that the user has moved one segment of a polyline, the program re-connects its vertex-mates as shown in Figure 5.2. The basic program contains six steps, two of which are performed by the user (*ie.* are inputs) to move the target segment.

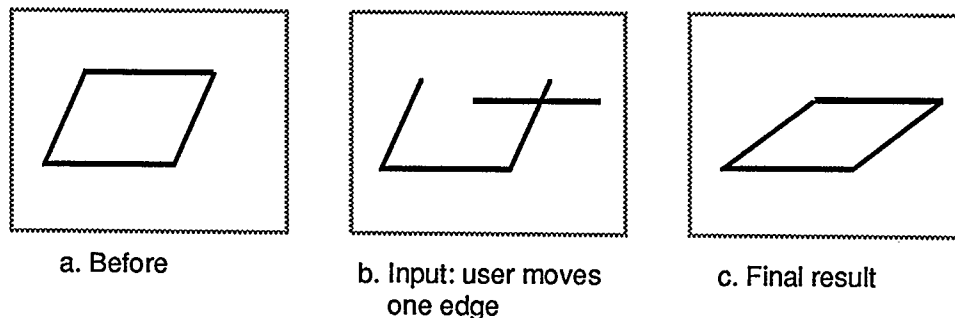


Figure 5.2 Connectivity task

Daedalos was presented with 8 traces of this procedure. Traces 1-4 and 6-8 presented different positionings of the target segment that resulted in coincidental contacts, and varied the order in which the teacher re-connected vertex-mates. The use of variables and constraint generalization easily eliminated these variances, so that Daedalos was predicting the entire task from the second trace on.

Trace 5 introduced a variant in which the user moved only one end-point of the target segment. Hence the user rejected Daedalos' prediction to grasp its mid-point and instead grasped its right end-point. The user also rejected Daedalos' prediction to transform its left vertex-mate (a harmless but also useless action). After learning this variant, Daedalos was able to predict either sequence in subsequent traces, the user's first input action (grasping either the mid-point or the end-point) being the trigger for the rest of the sequence.

It is most important to note that Basil did not classify the first grasping action as an input, but as a scan rightwards to some mid-point. Basil's discrete recording of actions precludes his observing that other lines were skipped.

Results of the tests above indicate that the system is capable of learning procedures. It is clear that for the most part the generalization model captures the essential constraints and structure of the tasks performed. The effectiveness of generalization was not quantified through controlled experiments but some work towards this is presented in the next section.

5.3 Learning without Prediction

Recall that eager prediction — to be precise the interleaving of performance with learning from the earliest possible moment — was adopted as a means of helping the teacher meet the correctness and minimal activity felicity conditions. Suppose this were eliminated; that is, suppose that performance were delayed until some number of complete traces had been integrated into a program. If eager prediction is useful, one would expect this to cause the learning system to produce a more complicated and possibly incorrect program.

Such an experiment was performed. Daedalos was presented with the same lesson traces used in the tests described in Section 5.2. This time, however, the researcher assented to a prediction only if it exactly matched the next step of the trace. In effect Daedalos was operating as a passive learner, matching its knowledge to observations rather than generating examples for the teacher to classify. Table 5.2 compares the two modes of learning (called "passive" and "interactive") with respect to efficiency of learning and quality of results. Efficiency is the rate at which the learning system becomes competent in a task; the measurement is based on the number of mismatches (between program and trace

steps) or rejected predictions. For the present, quality is synonymous with simplicity, measured by the number of edges in the program graph.

Task	Trace #	Steps in Trace		Edges in State Graph	
		Passive	Interactive	Passive	Interactive
Box-to-Line	1	21	20	18	13
	2	24	24	28	13
	3	20	20	33	13
Picket Fence	1	35	35	25	22
	2	29	27	32	22
Connectivity	1	6	6	7	7
	2	6	6	10	7
	3	6	6	10	7
	4	6	6	10	7
	5*	4	4	12	11
	6	4	4	12	11
	7	6	6	14	11
	8	6	6	16	11

* variant of task: move one end-point rather than entire line

Table 5.2. Performance data for passive vs interactive modes.

Clearly, the interactive mode learns faster and produces simpler programs. If we examine the actual mismatches, we find, not surprisingly, that coincidences, missteps and variant sequences are mostly to blame. The teacher has failed to meet the correctness and minimal activity felicity conditions when generating traces, and the system's built-in generalization capabilities are inadequate to filter out the noise. If the system's generalization capabilities (for matching program steps with observed actions) were enhanced, we might expect the passive mode to do nearly as well as the interactive, at (perhaps greatly) increased cost in computation. Eager prediction, on the other hand, makes correctness and invariance default conditions and changes the teacher's role to that of an informant, so that variance can be introduced only deliberately.

This preliminary experiment therefore establishes that eager prediction does significantly reduce the amount of effort the system must expend to become competent at a task, by helping the teacher meet the felicity conditions.

Chapter 6

What Have We Learned from Metamouse?

Recall the thesis stated in Chapter 1, Section 4:

End-user programming for computer graphics should be graphical and demonstrative. A practical programming system must limit the complexity of functional components to be induced, by analyzing traces and by requiring that the user employ graphically constructive techniques to satisfy simple felicity conditions. These requirements can be met by intensive interaction between user and learning system through a device, the Metamouse, that embodies the teaching metaphor and thereby enforces and helps the user to satisfy the felicity conditions.

Does the actual system (*ie.* the design and its pilot implementation) speak for or against this thesis? More importantly, does it clarify the meaning of the phrases “practical programming system,” “complexity of functional components,” “analyzing traces,” “intensive interaction,” and “teaching metaphor”? In this chapter it is argued that the system supports the thesis but does not place it entirely beyond doubt. The vague terms have taken on specific meaning through the implementation.

The chapter begins by establishing the limits of the project as it stands. Sections 6.2 and 6.3 assess key aspects of the system: the Metamouse metaphor; the generalization of traces and induction of programs; and the use of interaction. Sections 6.4 and 6.5 then reconsider the system as a whole, first in relation to the thesis and second in relation to more general problems. Shortcomings and ideas for further work are summarized. Finally, Section 6.6 summarizes the work, isolating its most significant aspects and indicating the nature of its contribution to knowledge.

6.1 Project Status

The project work related to this thesis comprises four parts: the development of design principles for a graphical programming system (Chapter 2); a design based on these principles (Chapter 3); a preliminary implementation (Chapter 4); and a series of experiments to evaluate the design (Chapter 5). The current implementation is as described in Chapter 4: a simple drawing program; a Metamouse icon that follows the teacher’s

actions and provides sensory feedback; the recording of traces of actions and Basil's sensory responses; a program to identify variables; a program to classify sensory events according to a hierarchy of constraints; and a program to induce program structure. This segregated system was used in experiments to show whether the system could work: action traces were collected and filtered through the generalization programs, resulting in symbolic program graphs (Chapter 5).

A phased implementation of the constraint solver is underway. When the first phase is complete, the system will be integrated so that studies with user populations can be conducted. The segregated system will continue to be of use to examine learning and generalization in detail.

6.2 Empirical Studies

Chapter 5 describes three assessment studies on the segregated system. The results are recapitulated here with particular regard to the thesis.

6.2.1 The Metamouse Metaphor

The rate at which potential users learn to understand the behavior of Basil the Metamouse, given a brief introductory description, was measured as the improvement in their ability to predict Basil, that is, to answer questions about what he would do and what distinctions he would make. The learning curves (Figure 5.1) indicated that the subjects had a good understanding from the start, and their performance neither improved nor deteriorated significantly as subsequent questions became harder. Furthermore, although sets of questions isolated aspects of Basil's behavior, performance did not greatly vary from set to set.

Several factors lessen the credibility of this study. First, it was not conducted in a controlled fashion; all subjects received the same introduction and the same questions in the same order. Second, the subject population was too small and too homogeneous (5 computer scientists). Third, the experimental situation was quite unlike the real one — a guided study booklet as opposed to an interactive graphical programming system. Given these shortcomings in the testing procedure, numerical results cannot be taken seriously.

Nonetheless, the gross results stated above have some value — it is easy to imagine very different results that did not in fact occur.

The conclusion of this preliminary study is that the metaphorical apprentice with a limited sensory system and a notion of conditional action is comprehensible to potential users.

6.2.2 Inductive Generalization

The ability of the system to induce programs from user demonstrations was examined for three example tasks (aligning boxes, equal spacing of boxes, and maintaining connectivity of lines). In each case the system did generate a program that could reproduce the useful actions originally performed by the teacher. The induced programs were general enough to operate in new situations as well. In the event that a program was unable to operate, new steps could be learned. Thus, capturing structure, achieving generality of performance, and incremental learning — three hallmarks of a useable system for programming by example — were demonstrated.

The major weakness of this study is the lack of proof that the results extend to an entire class of tasks that potential users would program; thus, although the learning system is shown to work, it is not shown to be useful. The study is defended on three grounds. First, although classifying tasks is beyond the scope of this thesis, it is suggested that the examples were at least representative: users actually perform these and very similar tasks; they included commonly occurring problems of measurement, relative position, and maintenance of constraint. Second, they expressed the fundamental elements of geometric phenomena described in Chapter 2: order (*eg.* the sequential selection of boxes during the alignment and spacing tasks); measure (*eg.* the spacer line used in the spacing task); and classification, (*eg.* the selection of points to transform based on their prior attachment to other points in the connectivity task). Thus the learning system is shown to be capable of expressing such phenomena in the programs it generates. Third, the tasks incorporated some of the basic problems in programming by example: detecting iteration (*eg.* sets of boxes transformed) and conditional branches (*eg.* variant on the connectivity task); identifying variables (*eg.* transformed points), constants (*eg.* initial placement of sweep-lines), and inputs (*eg.* guide-line and spacer).

A second weakness is that individual elements of the system were not isolated or varied to show their utility in the overall process of learning. The degree of lookback for variables was held constant. The test runs used only one set of rules for classifying constraints and only one set of generalization heuristics. The parameters that control the making and acceptance of conjectures in Daedalos were fixed.

Granting these weaknesses, the study constitutes an existence proof that inductive learning can be applied to graphical traces to produce generalized programs. The favorable results shown in Table 5.1 demonstrate that such programs can be compact and reliable. Moreover, the high ratio of actions performed by the system to those by the teacher indicated that interactive teaching by demonstration is pedagogically efficient.

6.2.3 Benefits of Interaction

The third study presented in Chapter 5 isolated a key element of interaction in the teaching process — the use of prediction to reduce variability in the action traces taught and thus help the teacher satisfy the “minimal activity” felicity condition. Programs induced with the aid of the teacher’s responses to predictions were simpler and thus — conforming with Ockham’s Razor, recently applied to machine learning [Quinlan 86] — better captured the structure of the task. To achieve identical programs without prediction would have required more generalization capability to conjecture partial matches and, more significantly, analysis to determine that different sequences have the same effect.

The main deficiency of this evaluation is that the experimental situation differs in two important respects from the real one. First, the predictions were made not by performing graphical actions but rather by printing a textual description of the program step without instantiating the variables. Second, the predictions were adjudicated by the researcher rather than actual users. Thus the results include no measurement of erroneous adjudication or its effects upon inductive learning. Moreover there is no anecdotal evidence regarding the comprehensibility of graphical predictions. Nonetheless the experiment was useful in establishing best-case results.

In summary, matching and predicting actions (in effect, learning from an informant [Michalski 83]) helped the system produce better programs than it would have by matching only. Moreover, the amount of task work done by the teacher was reduced — assuming that the cognitive load of accepting and rejecting predictions was not too high. Of course,

any useful learning system must eventually start predicting (*ie.* performing); the point is that doing so as early as possible is preferable.

6.3 Analytic Evaluation of the System

Apart from conducting empirical studies, one might also investigate the system's capabilities by analysis. In particular, the choice of internal representations and algorithms determine its abilities: to represent graphical problems; to produce distinct programs for different tasks; to generate equivalent programs from different lesson sequences. What follows is an initial attempt to reveal implications of specific design decisions — a proper analysis would require setting out a theory of graphics (a subsystem of geometry, no doubt) and a theory of human interaction with computers in drawing tasks.

6.3.1 Representing Problems

Clearly, as suggested in §6.2, the current system is capable of programming tasks that involve a sequence of point-to-point and point-to-line constraints. The limits of representability have not been worked out, but it is known that many problems can not be described because the action, sensory, constraint or generalization models fail to capture the constraints. Some examples are listed below, with suggestions for improvements to the system. The first two illustrate the need for more graphics operators — rotation as originally proposed, and grouping. The third and fourth reveal limitations in the learning algorithm and the system's model of constraints.

First, the pilot implementation cannot learn the Jarvis' march algorithm to construct a convex hull. The rotation operator was not implemented, and rotation of a line by moving one end-point without significantly altering its length is not modelled.

Second, given the current capabilities of A.Sq, the system cannot learn to trisect a line. In A.Sq one can bisect a line by drawing to or through its mid-point handle but no trisection constraint can be constructed. An operator for grouping objects so that they can be transformed by the same relative amounts makes such problems solvable. The procedure in this case is to make three copies of the original line, lay them end to end, group them and then scale the group to the length of the original line; each line in the group measures out one third of the original.

Third, the current design for Daedalos makes it impossible to learn to draw three boxes in a row — or any problem governed by a constant number. Daedalos would form a loop whose body draws one box, but the termination condition (that the third box has been drawn) is not representable. Of course, the teacher could input a graphical representation of the number, for example a box containing three line strokes with a line to sweep across and “count” them, as illustrated in Figure 6.1. Obviously this is inefficient from the user’s standpoint and may be arcane as well. The learning system can induce number, as suggested in [Maulsby 88a], if it records the actual count of iterations performed or accepted by the teacher so that constants can be observed.

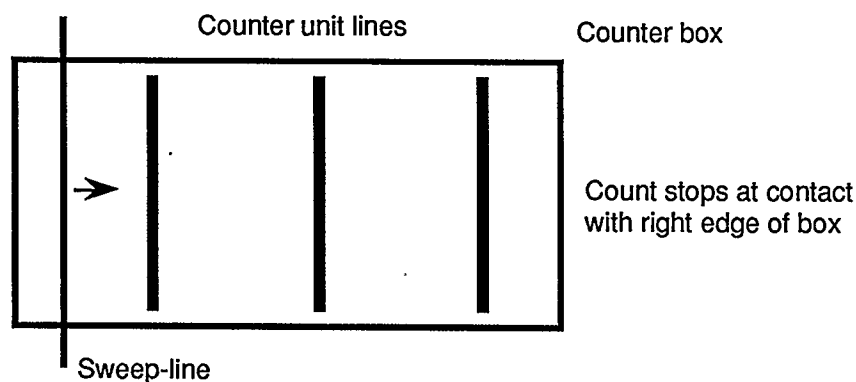


Figure 6.1 A graphical counting device

Finally, the system designed in Chapter 3 cannot learn an important input to the connectivity task. Selecting the line segment to shift out of place is really an *ad hoc* action to be performed by the user (*ie.* an input), but since Basil sensed a specific contact (with the mid-point handle) and a generally rightward path, the action is deemed sufficiently constrained to be performed automatically. In practice the constraint solver would likely select the leftmost line, regardless of which one was taught. The initial constraint analysis could be improved by attempting to replicate the action just demonstrated; if the constraint solver produces a different result, then what the teacher showed was underconstrained.

6.3.2 Distinguishing Programs

The ability to generate distinct programs for different tasks is a consequence of being able to represent tasks. For example, if the system can induce number then it can learn a program to generate four boxes but not five boxes, and another program to do the opposite.

Another example is being able to learn routines to translate boxes along specific paths (say 10° and 30°); because Basil's notion of direction is crude, programs induced from traces of either task would be equivalent (and incorrect) — both would translate boxes at 0° .

6.3.3 Sensitivity to Teaching Sequence

Teaching traces of a given task may vary in several ways not related to the different subtasks or conditional branches that must be taught. Traces may be composed of different actions — perhaps so different that a human observer would regard them as different methods. Traces may comprise the same actions but presented in different order. A more subtle variability, over which the teacher has little control, is introduced by coincidental sensory feedback that might falsely distinguish situations; for example, in box-to-line traces, sometimes a box is translated to the mid-point of the guide-line.

Eager prediction helps reduce the first two kinds of variability by reducing the influence of their source, the teacher, from a generator to a critic of actions. Its effect upon the third is quite unreliable: if the more general case (*eg.* moving the box to some point on the guide-line) is taught first, then more special cases are merely subsumed; but if the special case is seen first, the system will make incorrect predictions and must be taught the general case. This problem does not seriously impair learning or performance unless many special cases occur before the general case.

6.4 Support for the Thesis

The evidence for and against the thesis has been marshalled in the previous five chapters and recapitulated above. The terms have been defined through design and implementation. It is now time to decide whether the premises are reasonable, the propositions tenable.

The first premise, that end-user programming for computer graphics should be graphical and demonstrative, was argued in Chapter 1. The second premise, that a practical programming system must limit the complexity of functional components to be induced by requiring the user to satisfy four felicity conditions, was argued in Chapter 2. Given these premises, the thesis proposes 1) that an interactive device, the Metamouse, can be designed and implemented to help the user meet the felicity conditions, and 2) that a useful and easy-to-use system can be implemented for programming graphics by example. The thesis is

refuted if such a system is proven impossible; upheld if it exists; supported if obvious progress has been made towards it and it still appears to be feasible.

The system is *useful* if it can generate programs for commonly occurring tasks. Although a systematic inventory of tasks performed by users of drawing programs was not produced — partly because delimiting such tasks is practically impossible — there is at least anecdotal evidence that tasks involving alignment, spacing and connectivity constraints are common. It is suggested also that loops and conditional branches are signal attributes of useful programs. Hence the programs generated in the course of the performance trials are empirical evidence that the system is potentially useful.

The system is *easy to use* if a representative users of drawing programs find it preferable to the currently available alternatives. To determine this we must measure ease of use — admittedly a subjective metric, but appropriately so. Three approaches are: 1) ask users for ratings; 2) measure users' effort to execute vs teach given tasks in terms of time spent, number of actions performed, number of failed attempts (user abandons trace and recommences), and perhaps other quantifiable symptoms; 3) measure users' reliance on the system in a natural situation (*ie.* in performing their regular work) in terms of frequency of use. Collecting and interpreting any of these data is difficult. The first approach is clearly unreliable. The second presents methodological difficulties, since the objective evidence is indirect and must be carefully conditioned relative to individual subjects' skills. The third, while naturalistic, requires a fully working system and a long period of data collection.

Lacking a fully working system, the author has not performed any of the above measures. Instead he assessed the comprehensibility of the metaphor, on the assumption that this becomes the major factor in ease of use, given that a conventional direct-manipulation interface is provided. The results of that study suggest that the system is easy to use. This supports proposition 2 above.

In conclusion, the thesis has been neither upheld nor refuted, but supported through analytic and empirical means. The research has shown the feasibility of graphical programming by example.

6.5 Further Work Proposed

A final affirmation or refutation of the thesis requires considerable further development and investigation of the Metamouse programming system. If it is not refuted, the thesis should spawn a good deal of related research. The research agenda has four main subject areas: 1) development of the integrated programming system; 2) human factors studies; 3) theory of graphical tasks; 4) generalization and learning algorithms. Some activities under each category are proposed below.

6.5.1 Integrated System

An important step towards further research is to complete the prototype system so that studies of performance and human factors are more realistic. To achieve this we must build a constraint solver and interfaces between Basil, Daedalos and the solver. Working with the actual system may well reveal that it has been over-engineered and requires a rational reconstruction with more efficient code. Beyond this, the system can be usefully extended by:

- implementing the “leading” operators described in Chapter 3 to constrain paths
- adding rotation as an A.Sq or Metamouse primitive
- including more drawing primitives, such as ellipses and polygons
- providing an operation to group objects so that they are transformed identically
- being able to use previously learned procedures as subroutines
- supporting a voluntary explanation interrupt, so that the teacher can declare any action to be governed by a constant or input.

6.5.2 Human Factors Studies

As stated above, the system should be tested through further empirical studies of its interactive elements. A prototype system will allow “naturalistic” studies of actual use; but experiments on parts of the system will continue to be worthwhile because they isolate interesting features and hence permit greater control and easier analysis. The main lines of inquiry proposed are: 1) comprehensibility of the programming metaphor; 2) usefulness of interactive techniques to help users meet felicity conditions; 3) usefulness and usability of the integrated system; 4) users’ conceptions of the graphics domain, procedures and

constraints. These studies may reveal alternative metaphors, task models and interaction techniques. Examples are:

- submitting the “Getting to Know Basil” questionnaire to a larger and more diverse population
- adding controls to the questionnaire: vary the order of questions, exclude the answers from some papers
- conducting a study similar to the questionnaire but using the actual system and compare the results; such a study requires some means of monitoring subjects’ expectations or surprise
- quantifying users’ performance in teaching a set of standard tasks with prescribed algorithms, in terms of number of traces required, number of steps in each trace, and complexity, generality and correctness of programs produced
- repeating the teaching study but have subjects invent their own algorithms
- repeating the teaching study but variably eliminate elements of interaction: highlighting of indirect touches, of direct touches, of grasp; movement and presence of Metamouse; queries for explanation of unconstrained actions; prediction during lesson (*ie.* once performance has failed).

6.5.3 Graphical Domain Analysis

Other researchers have begun to investigate graphical tasks not only in terms of geometric operations [Henderson 80, Geller 87, Noma 88] but also in the perception and effect of constraints [van Sommers 84, Chow 88]. The graphical domain invites significant further theoretical and empirical investigation, for example:

- varying the degree of Basil’s sensitivity to indirect touch, from none at all to several and unlimited degrees of indirection; study the effects on generalization
- continuing the empirical study of drawing behavior, following [Chow 88]
- comparing Basil’s model of graphical constraints with actual human performance
- considering alternatives to the constraint classification model used in the system
- developing a similar constraint model for another domain, *eg.* text editing
- using conceptual graphs [Sowa 86] to represent graphical situations and constraints
- formalizing an algebra of drawing in terms of geometric constraints.

6.5.4 Generalization and Learning

Clearly, the choice of generalization methods and learning algorithm have a profound effect upon the usefulness of this system, yet these matters have received little critical attention so far. The pilot system, using the Daedalos algorithm and conservative generalization by disjunction with no explicit specialization¹ and no ordering of alternative predictions, generates programs useful in the sense that they can make correct predictions. In actual use, the system should make the best prediction first, where “best” means most specific or productive; the integrated system should order alternatives. The learning algorithm itself may be responsible for many inappropriate predictions because it has joined sequences too readily; the parameters that govern matching may need to be adjusted — perhaps even on the fly. Suggested projects include:

- repeating performance studies and teaching studies, varying Daedalos control parameters
- repeating the above studies varying generalization heuristics
- investigating adaptive selection of heuristics
- inducing constants and variables (*viz.* teaching study with queries for explanation eliminated)
- substituting the NODDY learning algorithm [Andreae 85] for Daedalos (sacrifice prediction during the first lesson)
- trying the Daedalos algorithm in other domains (*eg.* robot programming, text editing)
- characterizing the learning-power and instructibility of Daedalos, following guidelines given by [MacDonald 88].

6.6 Summary and Conclusion

The work described in this thesis makes real progress towards a system for end-user programming by example in graphics. Such a system has been shown to be feasible, and most of its components have been implemented.

Chapter 1 defined the project’s goal in general terms, emphasizing the importance of pragmatic considerations, chiefly that it be of service to a broad and diverse population of “ordinary” computer users. Previous work was surveyed and found to provide many

¹ A more specialized case is merely “disjoined” with the more general so that it can be selected as an alternative prediction.

insights and useful techniques, but no system was considered adequate for use by the general population. It was concluded that the system should be based on teaching by example. The statement of thesis indicated that a teaching metaphor and specific techniques of interaction would be required.

Chapter 2 examined the problems inherent in teaching and learning graphical tasks, as revealed by both empirical studies and theoretical approaches. The variability and noise inherent in graphical demonstrations was found to be a significant threat to the possibility of teaching graphics by example. A phenomenological approach to geometry revealed that human beings have an enormous advantage over computers in recognizing patterns and reducing the search space for procedural models. (On the other hand, the theory provides powerful tools to constrain and interpret actions.) In response to these problems, four felicity conditions — *correctness*, *show-work*, *no invisible objects*, and *minimal activity* — for teaching graphics were proposed. To help the teacher meet these felicity conditions five principles of design were proposed for the programming system — it must be an active learner, use a teaching metaphor embodied in an attention device, be based on geometric construction, predict actions whenever possible, and be able to suspend learning.

Chapter 3 presented the design of a programming system following the principles given above. The system is active in that it interacts with the teacher throughout the lesson; when an action is unexplained, it queries the teacher. Learning is incremental and interleaved with performance. The attention device is a metaphorical apprentice that embodies the system's pattern-matching limitations in terms of a sensory model and provides an action model appropriate to geometric construction, using transformation operators available in familiar commercial drawing programs. The learning algorithm enables prediction even during the first teaching trace (as soon as iteration is detected). The teacher can suspend learning at any time by putting Metamouse to sleep.

Chapter 4 detailed differences between the design and the actual current pilot implementation and defined the implementation path to a fully functional prototype. The pilot system permits testing and refinement of basic components, such as the modeling of graphical constraints, generalization heuristics, and so on.

Chapter 5 discussed three studies performed on the pilot system. The first presented potential users with a questionnaire regarding Basil's behavior. It was found that minimally instructed users became proficient at predicting Basil's responses and actions.

The second study examined the system's performance in learning several tasks from demonstrations. The system did generate programs for each of the tasks. These programs were quite compact and general even after a single teaching trace. The third study isolated a key element of interaction, prediction, in order to measure its usefulness. It was found that teaching by informant is extremely useful as a component of the system.

Chapter 6 summarized and criticized the work to date and rendered a verdict on the thesis. The main criticism is the lack of empirical or analytical results that could clearly affirm or refute the thesis. The studies conducted so far, despite their weaknesses, do nonetheless corroborate it. A rich agenda of further research was proposed.

The work presented in this thesis makes two significant contributions to research in end-user programming. First, it combines techniques of interaction and machine learning in a novel way. Second, it proposes a specific system for a rich and difficult task domain. The progress made to date is sufficient to warrant further research. And this research will be well worth the while if it results in systems that users can program by example with minimal effort. Once the barrier between *using* and *programming* is demolished, casual-user computing will burgeon.

References

- [Abbott 1884]
Edwin A. Abbott. *Flatland — A Romance of Many Dimensions*. Signet Classics edition. New York. 1984.
- [Abelson 80]
H. Abelson, A. di Sessa. *Turtle Geometry*. MIT Press. Cambridge MA. 1980.
- [Andreae 85]
P. M. Andreae. "Justified generalization: acquiring procedures from examples," PhD dissertation. Department of Electrical Engineering and Computer Science, MIT. January 1985.
- [Angluin 83]
D. Angluin, C. H. Smith. "Inductive inference: theory and methods," *Computing Surveys* 3 (15), pp. 237-269. September 1983.
- [Behnke 74]
H. Behnke, F. Bachmann, K. Fladt, H. Kunle, eds., S. H. Gould, transl. *Fundamentals of Mathematics, Vol. II: Geometry*. MIT Press. Cambridge MA. 1974.
- [Bier 86]
E. A. Bier. "Snap-dragging," *Computer Graphics: Proc. ACM SIGGRAPH '86*. Dallas. August 1986.
- [Borning 86]
A. Borning. "Defining constraints graphically," *Human Factors in Computing Systems: Proc. ACM SIGCHI '86*. Boston. April 1986.
- [Breidenbach 67]
W. Breidenbach, W. Suss. "Geometric Constructions," in [Behnke 74].
- [Chow 88]
U. Y. Chow, D. L. Mulsby, I. H. Witten. "Of mice and pens: human performance in drawing," Research report no. 88/319/31. Dept. of Computer Science, University of Calgary. September 1988.
- [Dybvig 87]
R. K. Dybvig. *The Scheme Programming Language*. Prentice-Hall. Englewood Cliffs, NJ. 1987.
- [Fikes 71]
R. E. Fikes, N. J. Nilsson. "STRIPS — a new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, pp. 189-288. 1971.
- [Freudenthal 67]
H. Freudenthal, A. Bauer. "Geometry — a phenomenological discussion," in [Behnke 74].
- [Fuller 86]
N. Fuller, P. Prusinkiewicz. "L.E.G.O.—an interactive graphics system for teaching geometry and computer graphics," *Proc. CIPS Edmonton*. 1986.

-
- [Fuller 88]
N. Fuller, P. Prusinkiewicz. "Geometric modeling with Euclidean constructions," in [Thalmann 88], pp. 379-391.
- [Geller 87]
J. Geller, S. C. Shapiro. "Graphical deep knowledge for intelligent machine drafting," *Proc. IJCAI 87*. Milan. August 1987.
- [Halbert 84]
D. C. Halbert. "Programming by example," Research report OSD-T8402. Xerox PARC. Palo Alto CA. December 1984.
- [Henderson 80]
P. Henderson. *Functional Programming Application and Implementation*. Prentice-Hall. Englewood Cliffs NJ. 1980.
- [Kienker 86]
P. K. Kienker, T. J. Sejnowski, G. E. Hinton, L. E. Schumacher. "Separating figure from ground with a parallel network," *Perception*, vol. 15, pp. 197-216. 1986.
- [MacDonald 84]
B. A. MacDonald. "Designing teachable robots," PhD dissertation. Department of Electrical and Electronic Engineering, University of Canterbury. Christchurch, NZ. 1984.
- [MacDonald 87]
B. A. MacDonald, I. H. Witten. "Programming computer controlled systems by non-experts," *Proc. IEEE SMC Annual Conference*. Alexandria VA. October 1987.
- [MacDonald 88]
B. A. MacDonald, I. H. Witten. "Autonomy, intelligence, and instructibility," Research report No. 88/335/37. Dept. of Computer Science, University of Calgary. October 1988.
- [MacDraw 87]
M. Cutter, B. Halpern, J. Spiegel. MacDraw. Apple Computer Inc. 1985, 1987.
- [Marr 79]
D. Marr, T. Poggio. "A computational theory of human stereo vision," *Proc. Royal Society of London, Series B*, 204, pp. 301-328. 1979.
- [Maulsby 88a]
D. L. Maulsby, I. H. Witten. "Teaching a mouse how to draw," Research report no. 88/294/06. Dept. of Computer Science, University of Calgary. January 1988.
- [Michalski 83]
R. S. Michalski. "A theory and methodology of inductive learning," in *Machine Learning*, ed. R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, pp 83-134. Tioga. Palo Alto CA. 1983.
- [Myers 86Chi]
B. A. Myers. "Visual programming, programming by example, and program visualization: a taxonomy," *Proc. CHI '86*. Boston. April 1986.

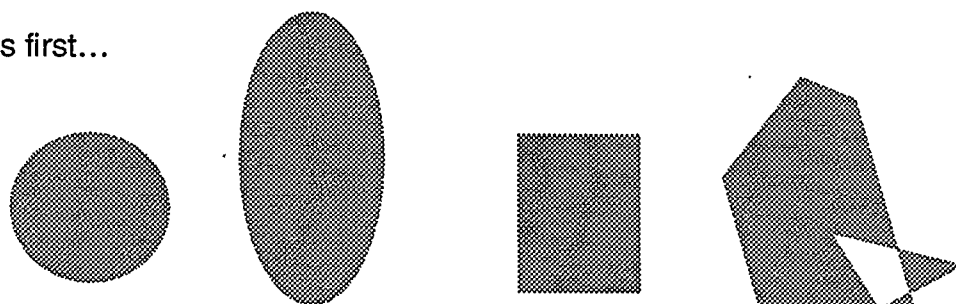
-
- [Myers 86CG]
B. A. Myers, W. Buxton. "Creating highly-interactive and graphical user interfaces by demonstration," *Computer Graphics: Proc. ACM SIGGRAPH '86*. Dallas. August 1986.
- [Myers 87]
B. A. Myers. "Creating dynamic interaction techniques by demonstration," *Proc. CHI + GI 1987*. Toronto. May 1987.
- [Noma 88]
T. Noma, T. L. Kunii, N. Kin, H. Enomoto, E. Aso, T. Y. Yamamoto. "Drawing input through geometrical constructions: specification and applications," in [Thalmann 88], pp. 403-415.
- [Papert 80]
S. Papert. *Mindstorms*. Basic Books. New York. 1980.
- [Preparata 85]
F. P. Preparata, M. I. Shamos. *Computational Geometry*. Springer-Verlag. New York. 1985.
- [Quinlan 86]
J. R. Quinlan. "Induction of decision trees," *Machine Learning 1*, pp. 81-106. Kluwer Academic Publishers. Boston. 1986.
- [Smith 75]
D. C. Smith, "Pygmalion: a creative programming environment," Report no. STAN-CS-75-499. Stanford U. 1975.
- [Sowa 86]
J. F. Sowa, E. C. Way. "Implementing a semantic interpreter using conceptual graphs," *IBM J. Res. Devel.*, vol. 30 no. 1. January, 1986.
- [Stallman 81]
R. M. Stallman. "EMACS — the extensible, customizable, self-documenting display editor," *SIOGA Newsletter*, vol. 2 no 1/2, pp. 147-156. Spring/Summer 1981.
- [Sutherland 63]
I. E. Sutherland. "Sketchpad: a man-machine graphical communication system," *Proc. AFIPS Spring Joint Computer Conference*, vol. 23, pp. 329-246. 1963.
- [Tempo 86]
Tempo. Affinity MicroSystems Ltd. Boulder CO. 1985, 1986.
- [Thalmann 88]
N. Magnenat-Thalmann, D. Thalmann, eds. *New Trends in Computer Graphics: Proc. CG International '88*. June 1988.
- [van Lehn 83]
K. van Lehn. "Felicity conditions for human skill acquisition: validating an AI-based theory," Research Report CIS-21. Xerox PARC. Palo Alto CA. November 1983.
- [van Sommers 84]
P. van Sommers. *Drawing and Cognition*. Cambridge Univ. Press. Cambridge UK. 1984.

Appendix A

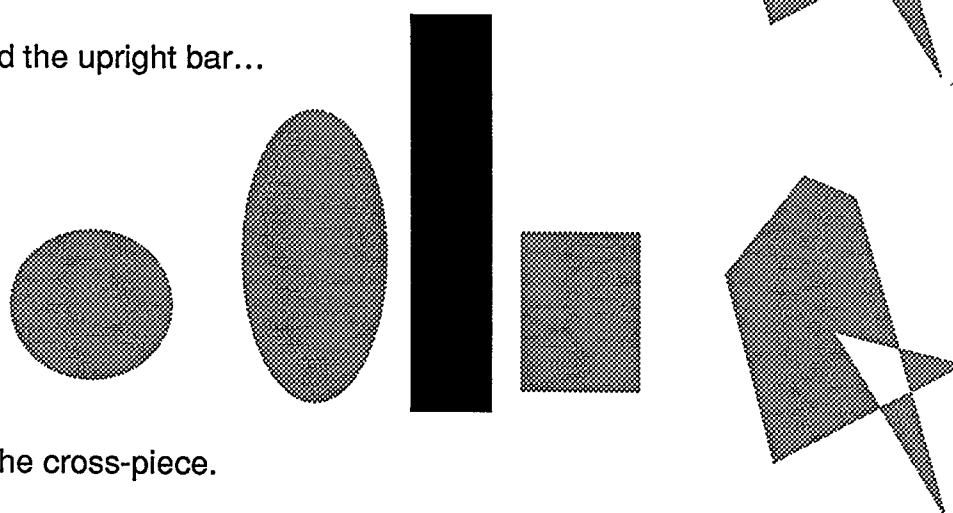
This appendix contains a copy of the instructions given to subjects of the user study described in §2.1. The pages have been reduced slightly. Note that subjects also received some verbal instructions, and that a researcher was available to answer their questions.

Task 1

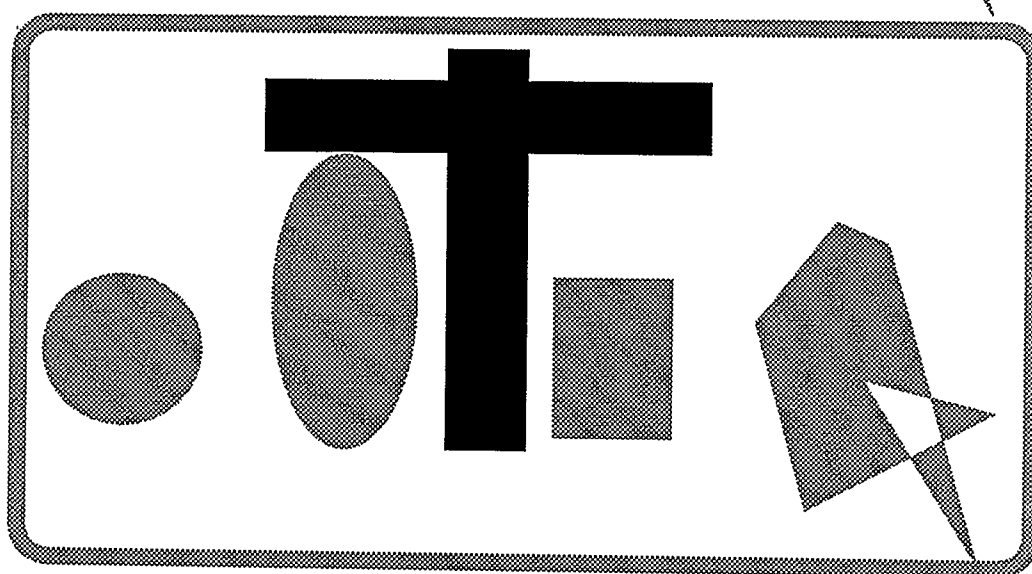
Draw this first...



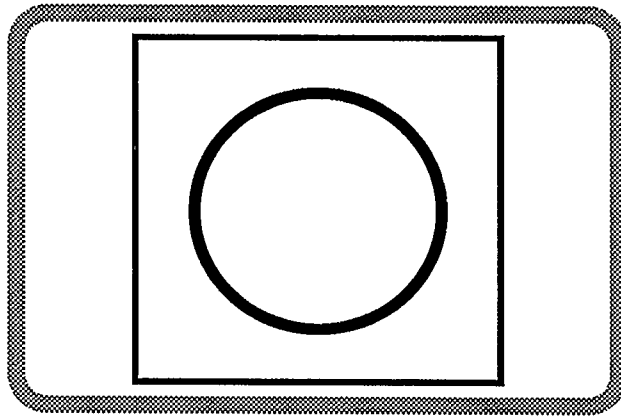
Then add the upright bar...



Finally, the cross-piece.



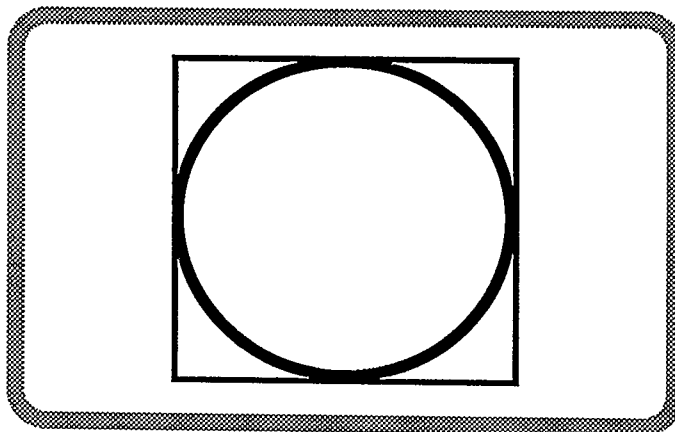
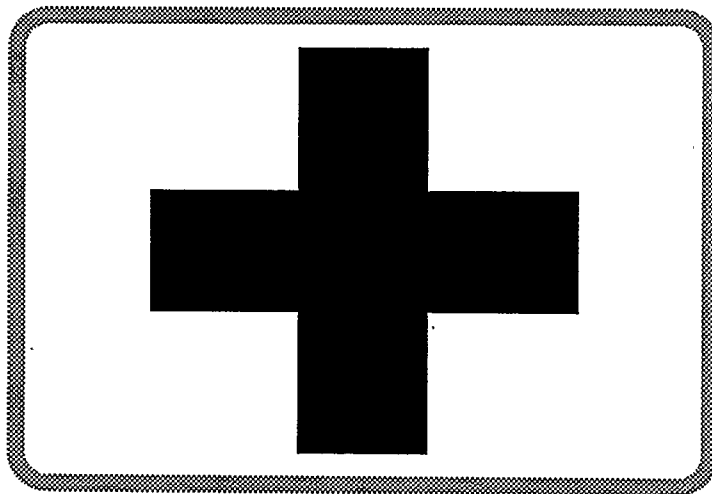
Task 2



Center a circle
in a square.

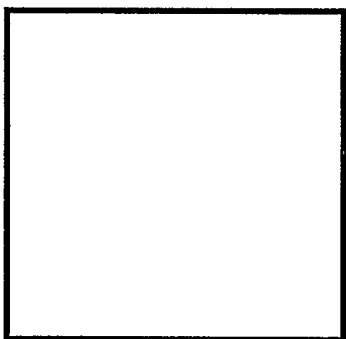
Try to do this
and the next two tasks
without using the grid.

Make a Greek cross.



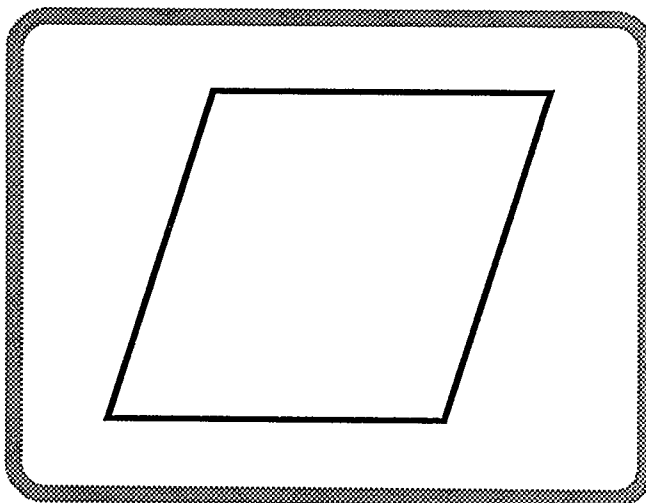
Square a circle.

Task 3

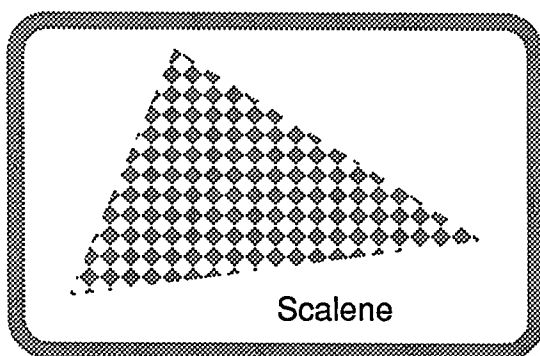


Turn a square...

...into a rhombus.



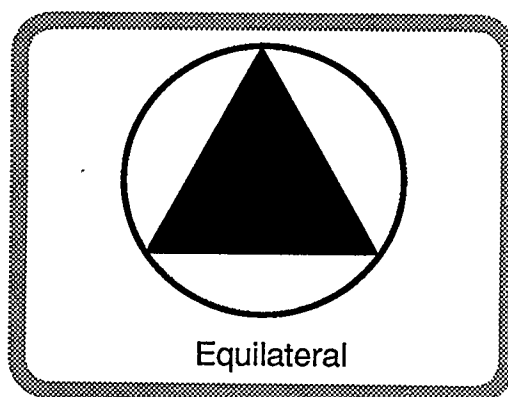
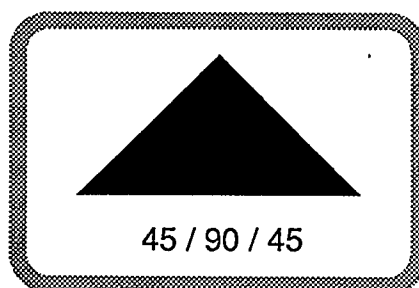
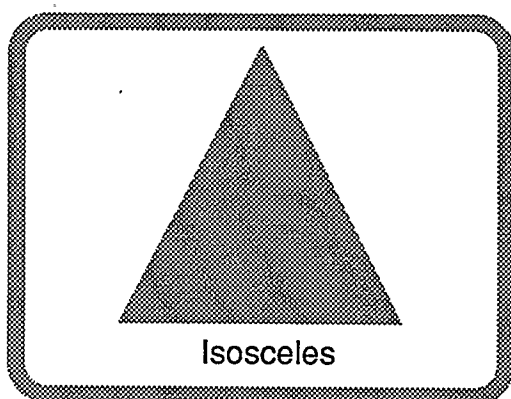
Task 4



Draw triangles
of the following types.

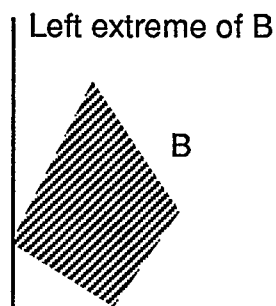
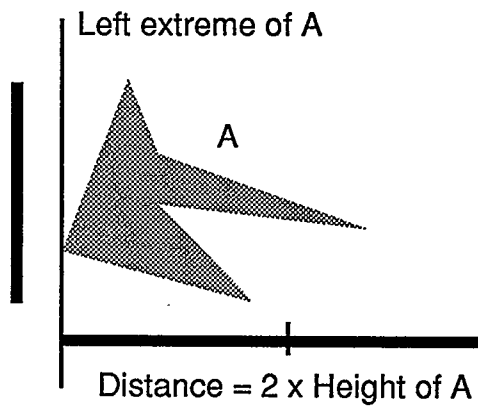
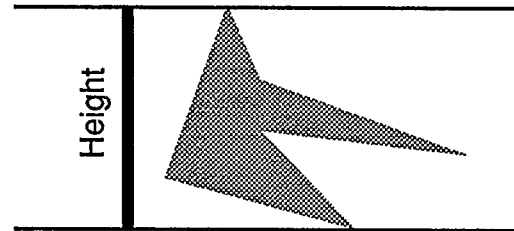
You may work on these tasks
in any order you think appropriate.

You may use the grid.



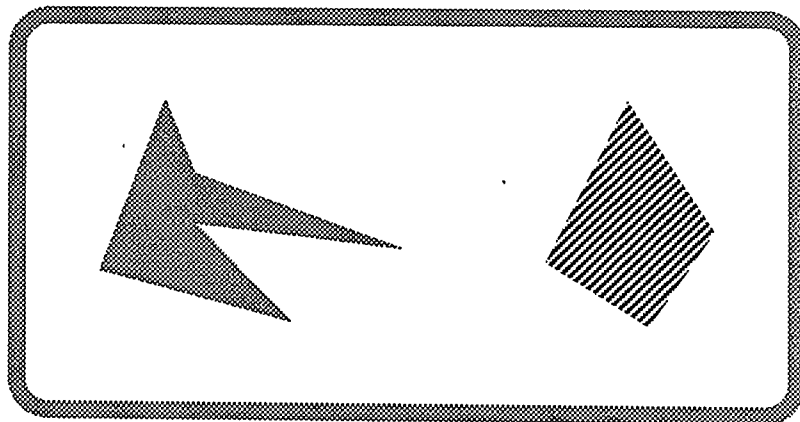
Task 5

Consider the height
of a polygon.

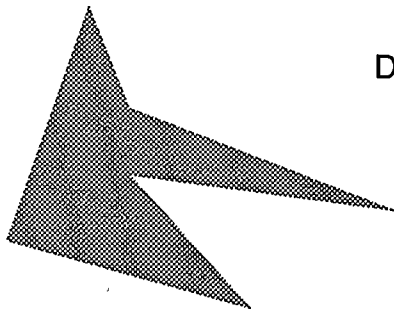


Here we use it as
the parameter of a
constraint on the distance
between two polygons.

You may use the grid.

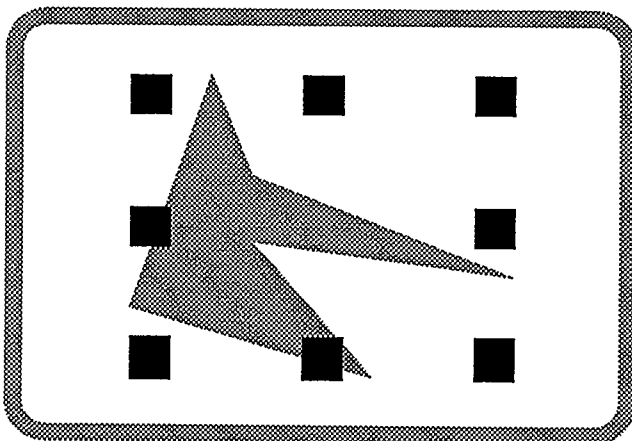
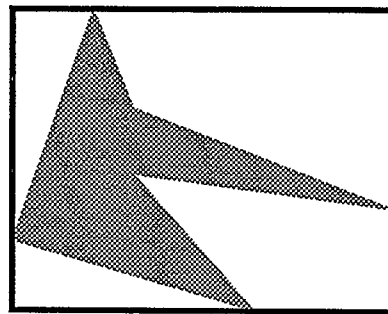


Task 6



Draw some sort of polygon.

Consider its “extents box”,
the rectangle that just
completely encloses it.

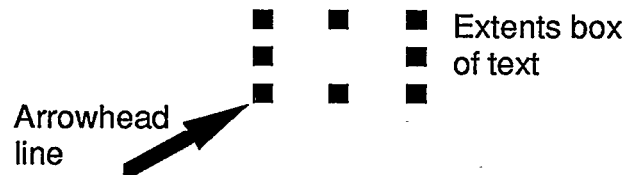


Represent the “extents box”
with eight squares,
the way MacDraw does.

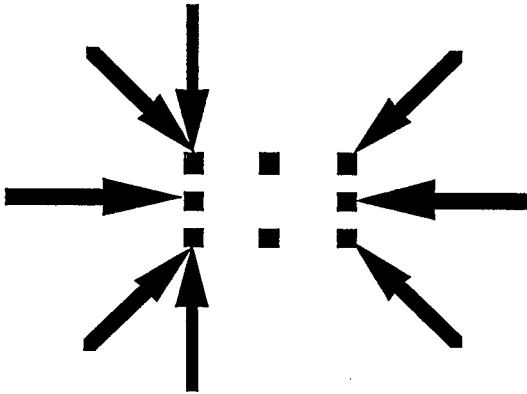
You may use the grid.

Task 7

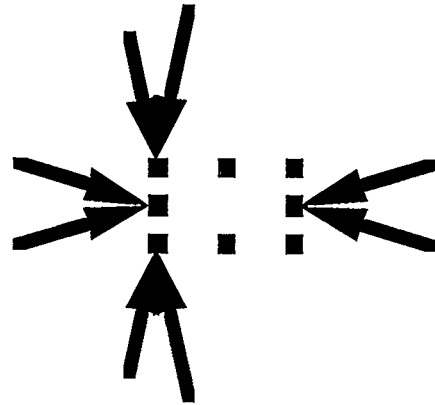
Here are the rules
for placing text
at the end
of arrowhead lines:



The limiting cases...



The in-between cases...



The subject is given a picture of lines as shown below, but without labels. After the first attempt the subject is shown the solution.

