1 Introduction

The direct manipulation interface introduced in the Xerox Star (Smith *et al.,* 1982; Johnson *et al.*, 1989) and popularized by the Macintosh (Williams, 1984) has encouraged people to use computers in their writing, drawing, and management activities. A serious shortcoming of current interactive point-and-click systems is their failure to supply a natural way for end users to create programs within the user interface. Without programming, only those operations designed into the system are automated—the rest are left for the user to perform manually. Enriching the system's repertoire with libraries of special commands, and enriching commands with optional arguments (as in Unix), tends to alienate the very people who are drawn to the simplicity of direct manipulation. An alternative approach is to base a system on relatively few primitive operations but make it easy for end-users to customize, creating their own procedures and importing those of others when desired.

End-user programming conflicts with the idea of direct manipulation because programs must include abstractions of objects and relations—and direct manipulation is about concrete, literal communication between user and machine. Programming a sequence of menu selections by demonstration is simple; the difficulty comes with the need to use abstractions (like variables and functions) and control structures (like iteration and conditional branching) that are normally implicit within a task. Although they can be specified by annotating the demonstration (e.g. Halbert, 1984, Pence & Wakefield, 1988), this distracts from real work, tends to deter users from setting up programs and, except in very simple tasks, is unsuited to those who have not been exposed to the art of programming. The alternative is to infer abstractions from the concrete traces that users provide. This will not be feasible unless search is restricted by focussing the system's attention on a small number of features at each step (Heise, 1989), and the system cannot reliably select these features itself without some help from the user and from domain knowledge.

Metamouse is a system for programming by example that helps users with annotation and focus of attention through a "coaching" metaphor. Users imagine they are training a graphical turtle named Basil. To work effectively they must understand the limits on the Basil's powers of perception and inference, and be aware of his focus of attention. The system communicates this information economically, by moving Basil to locations selected by the mouse, by highlighting objects he senses, and by asking questions through dialog boxes (MacDonald & Witten, 1987). Throughout this paper, "Basil" refers to the agent perceived by the user, and "Metamouse" to the underlying system.

The coaching metaphor comprises three elements of interaction. First, the Basil persona expresses the system's task model, including its focus of attention (nearby touch relations) and limits on its ability to make generalizations. Users understand that since Basil works by touch, measurements normally done "by eye" must be expressed by graphical construction. This extra information limits the search for generalizations but can be specified without abandoning the drawing program's direct-manipulation interface. Second, Basil demonstrates what he has learned at the earliest opportunity, so that users can benefit from it or correct it, as appropriate. Basil observes the user at work until he recognizes a pattern already learned, then predicts future actions, performing them for the user's approval. If he errs, or cannot find an action that fits the current situation, the user must resume demonstration. Third, Basil reacts immediately to the user's actions by providing feedback about their postconditions, from which program variables and conditional operators are abstracted. Feedback is graphical and limited to a simple classification of Basil's perceptions—objects and relations between them are highlighted one way if they are considered important, and another if merely observed. Should Basil need more information about the current postconditions, he requests it through a pop-up dialog that provides the possible replies.

In summary, Metamouse is an instructible system for graphical editing tasks. It learns complex, customized, iterative procedures based on a few editing primitives, and serves as an easy, effective technique for programming. It learns incrementally, so that a procedure invoked under circumstances which differ from those in which it was taught may be extended (or generalized, or debugged) quickly and easily. This greatly increases the reusability of end-user programs.

A prototype has been implemented which exhibits the basic structure and capabilities of an apprenticeship learning system. It observes the user's actions, performs a localized analysis of changes in spatial relations to isolate constraints, and matches action sequences to build a state graph that may contain conditional branches and loops. It induces variables for objects and distinguishes constants from run-time input parameters. It includes a constraint solver to perform the actions it has learned.

This paper describes the system's design, implementation and initial evaluation. We begin by illustrating how Metamouse can help the user with three particular graphical tasks. We review the history of tools to help automate graphical editing and contrast our approach with two main competitors, constraint-based and procedure-based systems, showing how it combines elements of both. We also briefly review similarity- and explanation-based generalization, techniques of machine learning that bear directly on demonstrational interfaces. Section 4 describes the components of the Metamouse system and how they work together. Section 5 evaluates its performance, first by showing how it copes with the three example tasks, and then by describing a small experiment on how human subjects come to understand it. Finally we appraise the limitations of the existing implementation and discuss how they might be overcome.

2 Applications

Aesthetically pleasing, visually coherent, meaningful pictures are characterized by the spatial relationships that join components, suggest relative importance, lead the eye through a visual narrative, and reveal subtle connections. These relationships are called "constraints." Often they compete with each other and must be considered together as a "constraint system." With or without the help of a computer, a graphic artist must manage complex constraint systems that may need compromise or careful ordering to be resolved. A drawing evolves as new objects and constraints are added and as some change while others remain. These elements often interact; for example, changing a type font may require enlarging and re-positioning boxes in a flowchart. Despite the features provided by interactive graphics editors to automate constraints, editing still involves repetitive manual work that demands precision, planning, and patience.

Metamouse induces graphical editing procedures from execution traces. We describe three tasks that exemplify important problems for users of interactive drafting packages: maintaining integrity of constraints throughout the editing process, coping with the tedium of repetition, and assimilating minor variations of a procedure. The need to achieve precision exceeding that of hand and eye, within a direct-manipulation system that shuns abstraction, underlies them all. The performance of the Metamouse system on these tasks will be evaluated below (Section 5).

2.1 Moving a stove

The first task illustrates a procedure to maintain constraints when a picture is edited, the use of an auxiliary object (a tie-line) to visualize a primary constraint, and sequential demonstration to express constraint dependencies. Figure 1 shows what happens when a kitchen design is altered by moving the stove (note that graphics are less detailed in our drawing program, but the actions are identical). Whenever the designer moves the stove, the computer should respond by relocating the hood above the burners and stretching or

shortening the stovepipe from the wall exit (compare Figures 1a and 1f). The designer expresses the desired relative position of stove and hood by drawing a tie-line between them (Figure 1b). She demonstrates the procedure's input and desired response by performing the complete edit as follows. First, she moves the stove (Figure 1c). The only change in touch (between tie-line and stove) is judged not to constrain the new position; by default this is an input. The user then drags the tie-line to touch the stove as before (Figure 1d). In the next steps, the user drags the hood to the tie-line (Figure 1e), deletes the tie-line and stretches the stovepipe to the hood (Figure 1f). The completed procedure is named, stored, and added to a menu.

When re-invoked, the routine draws the tie-line between stove and hood, then asks the user to move the stove. When she signals that she is done, the computer re-positions the hood and re-connects the stovepipe, using the touch constraints it had inferred. If the user dragged the stove (say, downwards) so that some constraint could not be solved (in this case, stretching the pipe to the hood), the system would ask her to demonstrate actions appropriate to that case.

2.2 Sorting a bar chart

In Figure 2, a set of rectangles is sorted by height and spaced at even intervals. This task illustrates iteration, precision, a selection rule, inputs, and constants. To teach it, the user must express implicit relationships such as distance and relative height using construction tools. Figure 2 shows the demonstration. Two display options have been set: show black tacks to indicate important touch relations, and show the turtle icon when predicting actions.

As a first step, the user draws a sweepline below the four boxes (Figure 2b). Finding no tactile constraints on its end-points, Basil suggests they are inputs; the user replies that they are constants. Next, the user draws a spacer at the left of the screen to control the distance between boxes (Figure 2b); it is an input. She picks the sweepline and drags it upwards till it touches the top of a box — this selects the shortest one above it (Figure 2c). She moves that box to the near end of the spacer (Figure 2d), then relocates the spacer to its opposite side (Figure 2e). Although both box and spacer are touching other objects as well, Basil places a tack where they meet, to let the user know he considers this to be the only important constraint on these actions. When the user picks the sweepline a second time, Basil predicts the loop by moving the line upwards until it touches the top of some box it has not touched in this way before (Figure 2f). Since the user accepts this loop, Basil performs all subsequent editing (Figure 2g,h) until no box remains in the sweepline's path. Failure to find a box is the loop's terminating condition. Basil then asks the user to demonstrate the rest of the program, which involves removing the construction tools (Figure 2i) and signaling that the lesson is over.

When this program is later invoked from the tasks menu, Basil creates a sweep line at the same window coordinates as before. When he creates a spacer he invites the user to edit it, since its position and length are inputs. Basil then performs the entire sort, regardless of the number of boxes.

2.3 Aligning boxes

Figure 3 shows a set of boxes moved horizontally to an arbitrary guideline given by the user. As in the sorting task, constraint amongst a set of objects is implemented by positioning each one using an auxiliary "tool," in this case the guideline. In Figure 3, the display options are to show Basil at all times and to show not only black tacks but also white ones, which indicate incidental touches. The figure also includes the dialog boxes through which Basil confirms hypotheses.

When the user draws the guideline (Figure 3a), its end-points are unconstrained; the default selection in the dialog box indicates that Basil assumes they are inputs. The user then draws a sweepline (to ensure that boxes move horizontally); the fact that it crosses the guideline does not constrain its end-points, so Basil assumes they are inputs also, but the user replies that the locations are constant (Figure 3b). The user picks the sweepline (Figure 3c) and drags it up to the first box; Basil infers that contact with the bottom of the next higher box terminates this action, and marks the constraints with black tacks (Figure 3d). The user grasps the same box and drags it to the point where the guideline and sweepline cross; Basil infers that this 3-way contact specifies where the box should go (Figure 3e). A third touch relation, between the box's lower left corner and the sweepline, is marked with a white tack, indicating that Basil does not consider it a constraint. When the user re-selects the sweepline (Figure 3f), Basil conjectures a loop and confirms it by performing the remaining iterations (Figure 3g-j). At each step, Basil asks for confirmation. Loop termination is detected as in the sorting task (Figure 3k). The user completes the task by deleting the tools (Figure 3l).

2.4 Adapting "align boxes"

One of the advantages of using a learning system is that a new task may be taught as a variant of something the system already knows how to do. This increases the re-usability of procedures: the user can invoke one that roughly fits the current problem, obtain immediate performance from those parts of it that are applicable, and manually perform (i.e. teach) the rest. An example of this is shown in Figure 4. This task differs from "align boxes" in several ways (see Figure 4a, b): the box at the far left is to remain where it is; tie-lines into boxes' left sides must be re-connected; and tie-lines from the upper and lower box into the middle one are to be re-connected to the latter at the middle rather than the left edge. The resulting program is shown in Figure 5; white circles are nodes created for "align boxes," black ones are for the variant.

The user begins by invoking the align boxes task. Basil draws the guideline at its default position and invites the user to edit it. When she is done, he draws the sweepline (Figure 4c), drags it up to the first box and performs the alignment (Figure 4d), all of which the user accepts. When Basil goes to grasp the sweepline again, the user stops him and re-attaches the tie-line (Figure 4f). This creates a branch in which the new action will be predicted at higher priority (see Figure 5, transition from node 8 to 12 vs. 5).

When the user picks the sweepline (Figure 4g), Basil recognizes this and conjectures a return to the main loop by dragging it up to the next box. Basil arbitrarily picks the one at the left (marked A in Figure 4h); the user rejects this and picks box C (Figure 4i). This introduces another branch from node 6 to 14 vs. 7 (Figure 5). When the user moves C to the guideline (Figure 4j), Basil fails to match this with the existing program (due to our implementation's simplistic conventions on merging variables). The user's return to the sweepline is recognized, whereupon Basil drags it to box D. He tries the new branch to node 14 but it fails for lack of a second box at the sweepline. Instead, he follows the old branch, aligning box D (Figure 4k) and, taking the branch to node 12, re-attaching its tie-line (Figure 4l).

Basil correctly exits from the loop and removes the tools (Figure 4m). When he predicts the end of the task, the user disagrees and edits the tie-lines from D to C (Figure 4n) and C to B (Figure 4o), thus introducing a final branch. The user can save the altered procedure as a new task or replace the old version. In either case, when it is re-invoked, the most recently taught branches are given priority, but their older alternatives are still accessible in case an entry condition fails.

This example illustrates the trading of control that can occur when debugging or adapting a procedure with Basil. These interruptions are worthwhile if the computer performs most of

the work, or the most difficult parts of it, or if the procedure is to be re-used often. Of 32 steps in this task, Basil performed 18. Improvements to Basil's generalization of actions would increase this to 20 (and eliminate the branch to nodes 14 and 15). Of 9 steps that involved precision positioning (aligning, re-connecting ties), Basil did 3. Feasible extensions of the system to generalize over symmetries would enable him to perform 5 or 6 of the 9.

3 Background

Historically, the automation of graphical editing tasks has progressed in two directions: interactive tools to help users with constraints, and graphics-oriented programming systems. The first seeks to improve either the naturalness or the power of declaratively specified constraints, while the second takes a procedural approach and lets the user construct programs that express his intention in geometric terms. Metamouse adopts a synthesis of the two. Constraints are not declared explicitly by users but are inferred from their actions, while to overcome the intractability of inference a procedural representation is used to decompose complex global constraints into structured sequences of local ones.

3.1 The declarative approach

The exploitation of constraints in interactive graphics began with SKETCHPAD (Sutherland, 1963), which used numerical relaxation to resolve several types of constraints: that lines be vertical, horizontal, parallel or perpendicular; that points lie on lines or circles; that symbols stand in vertical rows or be attached to points or lines. An interactive editing sequence typically involved new object definitions interleaved with constraint specifications. These early ideas are used in contemporary interactive drafting systems.

Two principal methods are used to facilitate high-precision interactive positioning: gravity and relaxation. "Gravity fields," which come in various forms in graphics systems (Foley & Van Dam, 1982), are limited in expressive power and offer only a small fraction of the desired types of precision. Relaxation-based methods are exemplified by White's (1988) "human interface to least squares" which lets users place constraints on distances and angles and then, on request through an *adjust* command, solves them using least-squares relaxation. Such systems force users to specify additional structures that are often difficult to understand and time-consuming to manipulate. To combine the convenience of grids with the power of constraints, Bier & Stone's (1986) SNAP-DRAGGING technique equips users with a variety of alignment objects—such as circles of specified sizes, horizontal and vertical lines—and "snaps" points of the drawing on to them. Once used, however, constraints are discarded, and subsequent manipulations do not respect the original positioning operations.

Most constraint-satisfaction drawing aids do not allow users to define new constraints. For example, to add new kinds of constraints to the original THINGLAB (Borning, 1981) one had to write code in SMALLTALK. However, the system has since been extended to allow graphical definition of constraints (Borning, 1986). The user draws an equational network, with icons that represent variables, constants, arithmetic operators, and function calls to other constraint routines. To define variables, one draws an example and labels points accordingly. Of course, the equational network requires users to have algebraic models of their problems.

Drawing is naturally procedural (Van Sommers, 1984), but constraint systems are declarative. THINGLAB insists on the user specifying programs declaratively. In White's (1988) scheme constraints are remembered, so that points, lines, and constraints can be added in any order and reapplied at any time, but it is not possible to store or manipulate a *sequence* of constraint-satisfaction problems.

3.2 The procedural approach

Computer drawing was originally a form of programming, images intended for production on a plotter being expressed as FORTRAN procedures. The theoretical basis for computer graphics was provided by Descartes' conceptual breakthrough of making geometry algebraic—although the supporting technology was a long time coming. From the point of view of the user, however, Descartes' invention was a *faux pas*. A more intuitive formulation of procedural graphics was created by the ancient Greeks—indeed geometry inspired the first investigations into the very notion of a formal procedure (Preparata, 1985). In the last two decades, constructive computer graphics have regressed a couple of millennia towards purely geometric specification of graphical procedures.

For example, LEGO specifies constraints using the traditional ruler and compass of geometric construction (Fuller & Prusinkiewicz, 1988). It provides primitives *point, line* and *circle,* and an operator that returns one or two points of intersection between objects. Constructions can be automated by procedural programming. Variables are identified by naming points—in this case those returned by the intersection function. Although a graphical interface is incorporated so that users can specify procedural constructs by menu selection, users are required to identify input and output variables and control structures explicitly. Noma *et al* (1988) also base a graphics language on Euclid's primitives: users create geometric constructions by writing small programs in this language. Concrete objects are named, but abstractions (like the length of a line) are not, because the concept of a "variable" was held to be too difficult for ordinary users. Instead, the language provides a limited stacking facility to allow each primitive in the program to communicate parameters to the next.

Procedural Euclidean geometry is a viable alternative to constraint systems for specifying figures to high precision. Kin *et al* (1989) argue that it is superior in two respects: constraint systems require considerable computation for large problems while constructive geometry is linear in the number of objects, and specifying consistent and sufficient constraints for a desired picture is a difficult task. However, the small and elegant set of Euclidean primitives, designed to provide a minimal basis for traditional "ruler-and-compass" methods of construction, does not relate well to real-life drawing—witness the fact that popular drafting programs find it expedient to offer a much richer set of pragmatically-motivated objects and operations. More important, the need to deal explicitly with procedural abstractions, expressed non-interactively as text or interactively through menu selection, negates the advantages of direct-manipulation environments.

3.3 The Metamouse approach

Instead of asking for an explicit specification of constraints or procedures, Metamouse observes the user at work and infers elementary relationships, constants, variables, loops and branches. This is programming by example: programs are constructed incrementally from execution traces. Some schemes that base programs on user-supplied example traces nevertheless force the user to work with programming abstractions. In TEMPO (Pence & Wakefield, 1988) users declare loops and conditional branches; SMALLSTAR (Halbert, 1984), which operates in a very general desktop domain, asks them to identify variables and their type and value range. PERIDOT (Myers, 1988) infers the range of a variable's legal values, certain spatial relations (such as "centered within box"), iteration over a list of objects, and the setting of active values conditional on selecting an object or mouse-button. It does not infer conditional branches that affect flow of control, and hence does not handle loops in general. Virtually no systems rely completely on automatic generalization; one that does, the robot teaching system NODDY (Andreae, 1985), performs an exponentially complex induction of functions and cannot cope with errors.

Inferring a program is not easy, but inducing complex transforms from examples of input and output is completely intractable (Angluin, 1983). In effect, a demonstration decomposes the transform into a sequence of simpler ones. Drawing is inherently procedural, often systematically ordered with each step governed by very few constraints (van Sommers, 1984). Nonetheless, it is hard to induce procedures even from simple steps. Typical users do not always construct the relevant measurements and relations, but work instead by visual inspection. Their drawings may lack important construction objects. For instance, in Figure 6a the square on the right is actually aligned with the diagonals of the squares on the left; these constraints are visualized in Figure 6b. But inferring constraints from a picture is unreliable because some relations may be incidental. In Figure 6b, the contacts between diagonal lines and the corners of the square seem to be constraints; a second example using a rectangle instead of a square (Figure 6c) demonstrates that these relations are incidental, that the constraint is between the diagonals and the the rectangle's center. Curve-matching methods such as those employed in graphical search and replace (Kurlander & Bier, 1988) are not powerful enough to induce patterns in drawings that contain "invisible objects" or incidental relations. Moreover, examining the whole screen for implicit spatial relations would often require an infeasible number of tests, and vastly expand the space of hypotheses for generalization. This is why our system restricts its attention to visible touch relations produced by explicit actions, marks touches with tacks so users can identify constraints by pointing at them, and asks users to specify complex relations by stepwise construction. To make this palatable we adopt the coaching metaphor, which combines demonstration, observation, correction and instruction. Our hypothesis is that by coaching, the user will gain insights needed to present explicit demonstrations and use constructions.

Our metaphorical apprentice employs both interaction and generalization to create a procedural model of the user's actions. It is the focus of attention of both user and system. Only local constraints involving it or an object it is grasping are examined. It incorporates an internal model of graphical constraints and asks for explanation when an action seems arbitrary—that is, insufficiently constrained. Rules of interaction between human teachers and pupils have been formulated as "felicity conditions" (van Lehn, 1983), and these apply when coaching Basil too: in particular *correctness* (examples shown are assumed to be correct), *show work* (demonstrate execution rather than just input and output), *no invisible objects* (express constraints by graphical construction), and *focus activity* (eliminate extraneous actions). To help untrained teachers obey these rules, Basil builds a model of the user's actions dynamically and predicts them as early as possible during a coaching session. The metaphor encourages the teacher to demonstrate constraints and adopt an intentional stance toward the system (Dennett, 1987) rather than guess the mechanisms behind its constraint and generalization models. Whether or not it succeeds is an experimental question, but initial results are encouraging (Section 5).

3.4 Machine learning and generalization

In order to create procedures from examples of their execution, Metamouse uses some generalization techniques that have been developed in the context of machine learning. A fundamental distinction in this area is between similarity-based learning and knowledge-intensive processes such as explanation-based learning (Witten & MacDonald, 1988). Given a set of objects that represent examples and counter-examples of a concept, a similarity-based learner attempts to induce a generalized description which encompasses all the positive examples and none of the counter-examples. Typically, background knowledge is not brought to bear on the problem except insofar as it is used to delimit the space of possible descriptions that are considered. In contrast, explanation-based generalization methods take a single example and deduce a general rule by relating it to an existing theory (Ellman, 1989). In effect they use examples to guide the operationalization of knowledge already implicitly known, so that it can henceforth be employed more efficiently.

The Metamouse system contains elements of both types of learning. Similarity-based learning is used when forming a sequential procedural model of a sequence of actions. User-demonstrated actions are assumed to be positive examples of connections in the model, as are those actions predicted by Basil and accepted by the user. Negative examples arise from predicting actions that the user rejects. The space searched is the set of automaton models consistent with the observed action sequence. A domain theory of programs—which might, for example, relate programs to their effects in some formal denotational semantics—is not used to guide generalization, because of its extreme complexity.

Explanation-based learning is used in two ways. First, the user is encouraged to create an explicit "explanation" of his action sequence by employing constructive techniques to reveal hidden relationships. This differs from conventional explanation-based learning in that the user is responsible for coming up with the explanation. If he fails to do so, learning will not just become bogged down while the system seeks its own explanation—it will fail completely. That is why we take great pains to encourage the user to demonstrate constructions explicitly. The alternative, to seek a theory that permits different constructions to be postulated and evaluated, seems too under-constrained to contemplate seriously.

The second use of explanation-based learning relates to identifying local constraints that govern actions. As explained more fully in Section 4.7, Metamouse incorporates a simple theory that distinguishes levels of significance of observed touches. The most significant are sifted out as constraints. This is a classic use of explanation-based learning to identify, via some domain theory, a subset of the currently-available information that serves to identify an equivalent situation rapidly in the future. The domain theory in this case is weak in the sense that its theorems are neither universal nor rigorously derived, but encapsulate important observed tendencies in the making of drawings. If an explanation turns out to be incorrect in a given situation, the wrong constraints will be stored and the system will either be inefficient or incorrect in identifying an analogous situation in future. If it is incorrect, the user will reject its prediction and enter the correct one, which will permit the information—but not the underlying theory—to be refined, perhaps corrected.

4 System components

Basil inhabits a simple interactive graphics environment. The user teaches editing procedures by demonstration, occasionally issuing simple instructions to focus attention and correct mistaken inferences. "Teaching mode" is identical to normal editing except that the turtle icon marks the most recent mouse click's location, and tacks mark intersecting objects. The learning module records each drawing operation at closure (a mouse click); until then, Basil waits at his last position, indicating that intermediate activity is ignored. The learning module associates objects with variables and distinguishes constraints from incidental touches. In constructing a program, it matches user actions with states, and confirms a loop or a joining of branches by predicting subsequent actions. Program states are generalized actions, bound to the current situation by a constraint solver. If the constraints have no solution (for instance, because a pool of objects for selection has been exhausted), the system forms a branch conditional on those constraints. The learning module's hypotheses, expressed in actions, icons and menus, can be corrected through direct manipulation: by performing the desired action, clicking on a tack, or picking an alternative menu item.

The following subsections give brief accounts of the drawing program and the Metamouse interface. Then comes a description of the learning module's components.

4.1 Drawing program

The drawing program A.Sq[1] resembles MacDraw (Cutter *et al.*, 1987) but includes only box and line primitives. The program has three modes (each indicated by a special cursor): *create-boxes, create-lines,* and *edit-objects*. The user edits objects by moving iconic handles, which appear whenever the cursor approaches them, as illustrated in Figure 7. Unlike MacDraw, A.Sq provides a multilevel undo/redo.

The choice of primitives and operators has a great impact on the user's expression of constraints. A.Sq's primitive object types, auxiliary data structures, and operators are summarized in Table 1. Points on the boundary of an object are represented in a parametric form. Any point on a line is designated by a number between 0 and 1, and on a rectangle by a number between 0 and 4. Thus, each vertex is a whole number and each edge is a line in parametric form. Coordinates and corresponding part names are shown in Figure 8.

The basic drawing operation is to select a point on the canvas, which becomes CurrentPoint. According to mode, this results in selecting an object or handle, creating a graphic, or re-locating a handle. The user and Basil view actions at a much higher level than A.Sq. For instance, drawing a new box involves a pair of actions for the user / Basil: in *create-boxes* mode, locate vertex 0; then locate vertex 2. A.Sq does the following: in *create-boxes* mode, *select-point* sets *CurrentPoint* and then activates *new-box* to allocate a rectangle having its vertex 0 there; *new-box* temporarily sets the mode to *edit-objects,* executes *select-handle* and *translate-handle-of-object-to-point* for the handle at vertex 2, whose interaction method in turn invokes *select-point*, which (since the mode is *edit-objects*) activates rubber-banding as the user re-locates vertex 2.

At present, the drawing program is relatively simple yet rich enough to study programming-by-example issues. No conceptual difficulties are envisaged in extending the system to work with touch constraints between polygons, ellipses, and splines. We also expect to be able to accommodate new operations such as rotation, grouping, and coloring.

4.2 Basil and the user interface

Prior to working with Basil, users skim the bio-sheet reproduced in Figure 9. Its concrete language and simple examples are intended to give an initial conceptual model of Basil's dependence upon explicit construction of spatial relations, so that users might meet the "show work" and "no invisible objects" felicity conditions (see Section 3.3). Further guidance is provided by several interaction devices: the Basil and Tasks menus, the turtle icon, touch indicators and dialog boxes.

The **Basil menu** contains two items that toggle their contents to set the teaching mode. The command pairs are: *begin / end lesson*; and *suspend / resume watching user actions*. The latter allows the user to work out a construction or fix up the drawing without introducing irrelevant steps into the program. A third useful mode would be *begin / end block of actions to be done only by the user*; thus a program could contain arbitrary manual steps.

The **Tasks menu** contains names of procedures the user has taught Basil. It is possible to select from this while teaching, so that tasks may be embedded, although the subroutine inherits no context from the current procedure.

The turtle icon has two purposes: to remind users that they are coaching, and to indicate the system's focus of attention. It moves to *CurrentPoint* (figuratively, Basil's snout) after each drawing action, thus maintaining the context of relative motions.

---

[1]Named after the protagonist of Flatland (Abbott, 1884).

Basil is described as near-sighted but touch-sensitive; moreover, the user needs to understand that only binary touch relations including Basil or *CurrentObject* (figuratively, the one *grasped* by the turtle) are checked. If the user intends that more remote touches play a role, she must move Basil to the relevant objects to check for them. To convey Basil's restricted focus of attention, the system marks touch relations it observes with tack icons, as shown in Figure 10. Black tacks mark touches that Basil considers to be important constraints; white tacks mark incidental touches. If the user disagrees, she may click on a tack to change it from black to white or *vice versa.*

Non-blocking (i.e. response optional) dialog boxes appear when the system makes an hypothesis about Basil's path (see Figure 3d) or an implicit constraint (Figure 3a,b). The path is stated in the prompt and shown in the icon's heading; should the user disagree, she is asked to turn Basil to the desired axis by tapping on him. The default implicit constraint (when no touch governs an action) is that the position is set by the user (Figure 3a). Should she disagree, she may select "always here," which means constant absolute position; "this far from last point," which means constant relative position; or "relative to an object," which means that the point should have been constructed. In the latter case Basil asks her to draw the construction and adjust the original object's position afterwards if necessary. (The construction steps are inserted into the procedure ahead of the original action.)

The system puts up three blocking "yes/no" dialog boxes. When the user's action matches some previous step, Basil requests permission to predict (Figure 3f). When performing a step, Basil displays a simplified description, stating the operator and grasped object type, and asks the coach to accept or reject it (Figure 3g-j). A third option would be useful here: always let the user do this step. At the end of a lesson, the user is asked whether she wants the task saved: if so, she types in a name that will appear in the Tasks menu.

If an action is to be done by the user, Basil puts up a dialog with the abbreviated description and response options "Done" and "No". The appropriate object is selected or created for the user to edit. In the case of a new object, it is drawn at the same position as originally taught. When the user has finished editing it, she clicks "Done".

When no prediction is performable (as in Figure 3k), Basil displays an advisory message asking the user to demonstrate the default action.

The system has several display options, intended mainly for use in our research. The turtle icon may be shown at all times during a lesson, or only when Basil is predicting an action, or not at all. Tacks of either color may be shown or hidden.

4.3 Overview of learning module

Figure 11 depicts the learning system. During a demonstration cycle, in the top half of the diagram, the user performs actions which the system analyzes and appends to the program. When a user action matches some existing program step, the system enters a prediction / performance cycle, shown in the diagram's lower half. Basil executes program actions until the user objects or no step is performable. When this happens, Basil asks the user to take over and the system returns to the demonstration cycle. The learning algorithm is summarized in the following two paragraphs.

Demonstration cycle For each action performed by the user, the system records its operator (e.g. *drag-handle*), its parameter (the active *Handle*), and touch relations involving *CurrentObject.* This "sensory-augmented" action is passed to the matcher, which searches the program, represented as a state transition graph, for a step whose constraints are met by the new action. If a matching step is found, the system attempts to predict its successors (see next paragraph). Otherwise, the user action is analyzed as follows: First, the variable inducer finds or allocates variables for objects in touch relations. The "variable-augmented" action is then passed to the constraint classifier, an explanation-based generalization module

that uses domain knowledge about the current operator in order to isolate those touches that would constrain its parameter values. Finally, this "constraint-augmented" action is passed to the graph manager (not shown in Figure 11), which appends it to the branch of the program currently being taught. If the user's action immediately follows a rejected or failed prediction (see next paragraph), the step is added as a new branch, a sibling of that prediction.

Prediction / performance cycle When the matcher finds a program step whose constraints are satisfied by the user's action, it signals the graph manager to do two things. First, the manager makes a link from the *previous* user action (the putative end of the new branch) to the matched step; this link is subject to confirmation (see below). Second, it updates the program state, marking the matched step as the one most recently performed, and re-binding variables to objects in the user's action. The system then enters the prediction / performance cycle, executing from the step's successor set, which is a preference-ordered list of alternative actions. The system predicts each alternative in order until both the constraint solver and the user accept one, or all are exhausted. In the former case the program state advances, variables are updated, and execution continues from the accepted step's successors. In the latter the system returns to the demonstration cycle, with new actions to be appended after the last accepted step. If at least one prediction was accepted during this cycle, the new link is confirmed; otherwise it is deleted, the match is cancelled, and new actions will be appended to the branch from which the link was tried.

The following subsections describe the main modules in Figure 11 (which has been annotated with section numbers for easy reference). A running example, the "align boxes" task (Section 2.3), illustrates their function.

4.4 Action recorder

Each of the user's editing actions is recorded, together with its context — a part of A.Sq's state isolated by Basil's focus of attention. (Recording only part of the state constitutes implicit generalization.) All actions of the current lesson are remembered in sequence in an *ActionTrace,* from which the matcher builds a program (Section 4.5). Each program step references the actions from which it was generalized.

An *ActionRecord* has three components: *Operator, Parameter,* and *Results.* For instance, when the user drags the sweepline upwards to a box in Figure 3d, the following is recorded:

Operator :	*drag-handle*
Parameter :	*Handle = Line041.midpt*
Results :	*CurrentPoint = (240,130)*
	grasp (Line041.[0.52])
	touch (Line041.[0.07] : Box057.[3.00])
	touch (Line041.[0.35] : Box057.[2.00])
	touch (Line041.[0.38] : Line040.[0.22])

The *Operator* is a composite of those defined by the drawing program, so that it corresponds with the granularity of actions as seen by the user — one action per mouse click. Thus, the five basic operators are: *locate,* which places a new graphic's first point, *select,* which picks a new *CurrentObject* or active *Handle*; *draw-line* and *draw-box,* which sweep out new objects; and *drag-handle,* which translates the active *Handle* to a new *CurrentPoint.*

Parameter identifies *CurrentObject* and the currently active *Handle* (if Basil is dragging or drawing something).

The *Results* comprise the new mouse location *(CurrentPoint)* and a list of *TouchRelations* in Basil's focus of attention. The constraint classifier (Section 4.7) examines *TouchRelations* in order to identify constraints. Screen coordinates of *CurrentPoint* may be used to define a position or direction constraint. Restricting the sensory focus of attention reduces the time spent checking for touches and simplifies the analysis of constraints.

To assemble *TouchRelations,* the recorder scans A.Sq's *DisplayList,* selecting graphics that touch either Basil or *CurrentObject.* A *TouchRelation* is defined as:

$$touch\ (Object_1.Part_1 : Object_2.Part_2),$$

where $Part_i$ is an edge coordinate in $Object_i$ (see Table 1) and $Object_i$ is the graphic's address in *DisplayList.* $Object_1$ is either Basil or *CurrentObject.* If Basil, then $Part_1$ is 0 (his snout). If $Object_2$ is *CurrentObject,* the touch relation is distinguished as $grasp(Object_2.Part_2)$ — the difference between touch and grasp proves important when inducing constraints.

Results can be thought of as an action's observed postconditions, in the sense of Fikes and Nilsson (1971). Metamouse generalizes them into a model of the action's goal; moreover, it uses the satisfiability of postconditions as the condition on which an action is selected from a set of options (branches). Although conditional branching may depend on preconditions as well, our current implementation does not support this.

If the user undoes an action, it is removed from the trace and reference to it is removed from the corresponding program step. If the step references no other actions, it is replaced with a dummy node, which has the effect of linking all of its predecessors to its successors.

4.5 Action matcher

The action matcher searches the program for a step that is equivalent to the one just performed by the user. A step is equivalent to an action if it can produce the same effects in the same situation: in other words, if the program could have predicted the user's action had it been told to do so.

A program learned by Basil is a directed graph of *ProgramSteps* with no restrictions on connectivity: it may contain arbitrary multiple-way branches and loops with jumps into or out of their bodies. An example is shown in Figure 5. The graph is equivalent to a production system whose context (left-hand side) length equals the number of steps Basil must match before a new link is confirmed.

Each *ProgramStep* has three components: *Predecessors, ActionGenerator,* and *Successors.* The first and third are lists of *ProgramSteps* that precede or follow this one in the graph. *Successors* are ordered according to the priority at which they may be predicted. *ActionGenerator,* a generalized *ActionRecord* (Section 4.4), has three parts: *Operator, Parameter,* and *Constraints.* For example, step 5 of the align boxes task, in which the user grasps the sweepline (see Figures 5 and 3c) is:

Predecessors :	*{Step4, Step8}*
Successors :	*{Step6, Step9} Note: do Step9 if Step6 fails*
ActionGenerator:	*Operator = select*
	Parameter = nil
	Constraints = grasp ([L2=Line041].[P9=0.5])

Operator is one of the five actions listed in Section 4.4 and *Parameter* specifies the object and handle in Basil's grasp prior to the action. *Constraints* is a set of touch relations or position specifiers that must hold after executing *Operator* with the given *Parameter.*

Constraints are generalized touch relations, where variables (e.g. *L2, P9*) stand for object and part identifiers. They are instantiated and checked by a constraint solver (Section 4.8).

A *ProgramStep* matches an *ActionRecord* if the following conditions hold (they are checked in order for quick rejection of mismatches). First, *Operators* must be the same. Second, the *ActionRecord* must contain at least as many *TouchRelations* as the *ProgramStep* has *Constraints*. Third, *Parameters* must agree: that is, the *ProgramStep's* object and part selectors (see Section 4.6) must generate the object address and part coordinates specified in the *ActionRecord* (for the example above, they are nil). Fourth, each *Constraint* must have a corresponding *TouchRelation* such that its selector functions evaluate to the latter's object address and part coordinate. (Part coordinates match if they lie within a defined tolerance.)

Some *Constraint* selector functions that search for objects must be evaluated with respect to the A.Sq environment as it was just before the user's action. This is accomplished by temporarily restoring Basil and *CurrentObject* to their previous coordinates, without updating the display screen.

In effect, action matching is solving for constraints where only one potential solution, the demonstrated action, can be checked. For instance, Figure 3f matches *Step5* of Figure 5: both *Operators* are *select*; both *Parameters* are nil; and 3f's *Results* include a *TouchRelation* corresponding to the only *Step5 Constraint, grasp ([L2 = Line041].[P9 = 0.5]).*

A user action matches a *ProgramStep* with a constant position constraint if its resultant *CurrentPoint* lies within several pixels of the constant. A step whose constraint is an input position will match an action with no touch relations, regardless of where its *CurrentPoint* lies.

The matcher can be parameterized to search forwards or backwards, breadth- or depth-first, starting from the graph's entry point or from the last accepted step, and to stop at the first match or find all matches. For the evaluation study (Section 5.1) it was configured to search backwards depth-first from the last accepted step until the first match was found.

4.6 Variable inducer

For Metamouse to apply the same task to different objects, as when iterating over a set, it must use variables in constraint expressions. Variables may be thought of as representing roles (Rich & Waters, 1988) such as "X: the object in Basil's grasp three steps before this one," or "Y: a box lying above Basil's current location and not previously used in this operation." Some aspects of a role are implicit in a variable's context (the action and touch relations in which it was defined) but other criteria, such as whether the object has been used before, are expressed by *Selector* functions. The variable inducer creates placeholders for objects and parts in touch relations. These will be used by the constraint solver to instantiate a program step.

A *Variable* is local to a *ProgramStep* and often appears in several *Constraint* records. Each has four components: *Name, Type, Selector,* and *Bindings*. For instance, here is the variable for the box found by the sweepline in Figure 3g (the second iteration of align boxes):

Name :	*B1*
Type :	*box*
Selector :	*find-novel-object ([Box057], box, Upwards)*
Bindings:	*[Box061, Box057]*

The *Name, B1,* is a globally scoped symbol. *Type* is one of *{box, line, handle, edge}*. *Selector* is the function to be called by the constraint solver (Section 4.8) when a new value

is required. Table 2 lists the four functions currently in use: the first two are common to both objects and parts; the third, *find-named-part,* is a table-lookup from part names to edge coordinates (see Figure 8); the fourth, *find-novel-object,* chooses a graphic of this variable's type but not bound to it before (this prevents re-editing an object that may have been moved into the range of search). *Find-novel-object* has an optional argument, the direction of search, in case the constraint classifier deems it relevant.

Bindings is a stack of the variable's values — object addresses or part coordinates. Previous bindings are remembered just in case *Selector* requires them, as in *find-novel-object* above.

Algorithm Given an *ActionRecord,* the variable inducer assigns each object and part value in each *TouchRelation* to some local *Variable.* First, ensure a 1:1 mapping of objects and *Variables:* If an object address is already assigned to some *Variable,* use that *Variable* again. If a part coordinate and its containing object are both already assigned, use the existing part *Variable.* Otherwise create a new *Variable,* initialize its *Name, Type* and *Bindings,* and then find an appropriate *Selector* according to the rules given below.

If an object was drawn by the current action, its *Selector* is *created (Type).* Otherwise, scan backwards through the action trace: if the value is the current binding of some variable *X* in a previous action, then the *Selector* is *use-value-of (X).* The default part *Selector* is *find-named-part,* where the name is that corresponding to the part coordinate in the *TouchRelation* (Figure 8). In the present implementation it is assumed that a line is directed, so the selector for an end-point indicates whether it is the start or end of the line. The default object selector is *find-novel-object.*

4.7 Constraint classifier

The key to generalizing an action is to distinguish those touch relations that are intended from those that are incidental. We call the former "constraints". Isolating constraints is, of necessity, a heuristic procedure. One approach is to gather multiple examples and choose those touches that occur in every one — this is similarity based generalization. In order to minimize the number of examples the user gives, we took another approach — explanation based generalization. The constraint classifier examines each *TouchRelation* of the current *ActionRecord* and assigns it to one of eight levels of "significance." The most important touches are selected as constraints and the rest are deemed incidental. If insufficient constraint is found, the module may mark direction of movement as an additional criterion, or initiate a dialog with the user regarding implicit constraint (see Section 4.2 and Figure 3a,b).

A *ConstraintRecord* has three parts: *TouchRelation, Level,* and *Classification.* For instance, when the the sweepline meets the first box (Figure 3d), the following *ConstraintRecords* are added to the action (note that variables have been inserted already):

grasp (L2.midpt)	*Level: Trivial*	*Class: Incidental*
touch (L2.? : L1.?)	*Level: Sustained*	*Class: Incidental*
touch (L2.? : B1.bottom.left)	*Level: Weak 2*	*Class:* ***Constraint***
touch (L2.? : B1.bottom.right)	*Level: Weak 2*	*Class:* ***Constraint***
Path (Upwards) is a ***Criterion***		

Since the grasp cannot be changed by *drag-handle,* it is of no significance. The fact that sweep- and guidelines cross, expressed above as *touch (L2.? : L1.?),* is sustained throughout the action, hence judged less significant than contacts between sweepline and box, which result from the action. Only the most significant touches are chosen as constraints. Since they have two options (the choice of a box for *B1* and the point of contact along *L2*), the upwards path is added as a criterion to resolve these decisions.

Method The selection of constraints from observed touch relations is a three-stage process. First, each touch is classified according to whether the action caused it, altered it, or had no effect on it. Second, the touch is assigned a level of significance according to the type of effect and the number of variables that take their value from a set of multiple options. Third, all touch relations are ranked by significance, and those at the highest level are selected as constraints.

1. Type of effect To determine how the action affected a given touch relation, the classifier consults a decision tree like that in Table 3. A *Sustained* relation holds true throughout the action: that is, object and part identifiers remain the same, although part coordinates may change (as when the sweepline slides along the guideline). An *Effected* relation occurs as a result of the action (e.g. the touches between sweepline and box). An *Inevitable* relation is "*Sustained* by definition," that is, under no circumstances could it cease to hold as a result of the action (e.g. grasping a handle as it is dragged). An *Unaffected* relation must have held prior to this action, even though not sensed beforehand (e.g. when Basil moves to grasp an object, any touch relations it has with others are *Unaffected*).

The decision rules check the *Operator,* locality of touch, whether both parts remain stationary, and whether the touch relation persists. Locality of touch distinguishes grasp (that is, between Basil and *CurrentObject*), direct touch (between Basil and some other object), and indirect touch (between *CurrentObject* and some other). Direct touches occur when Basil locates the start of a line or box at some point on another object, or moves to grasp at a point where several objects intersect.

2. Level of significance To decide a touch's level of significance, the classifier consults the rules given in Table 4. Factors to consider are the number of free variables in the relation, the type of effect (found above) and in one special case the *Operator*. A variable is free if its value is chosen from a set of alternatives. A *TouchRelation* has at most three such variables: $Object_2$, $Part_1$ and $Part_2$. An object variable is free if its *Selector* scans the *DisplayList*, as does *find-novel-object* for *B1* in the example above; because of the way TouchRelations are defined, this may be true of $Object_2$ but not of $Object_1$. A part variable is free if its *Selector* returns a range of parameter values, as in *find-named-part (?)* for *L1,* which returns the edge 0…1. In effect, contact with an edge affords the touch relation a degree of freedom, and one or both part variables may stand for edges.

In decreasing order, the levels of significance are: *Determining*; *Sufficient*; *Weak 1, 2, 3*; *Unaffected*; *Sustained*; and *Trivial.* This ordering reflects the ability of a touch to limit a set of positions derived by the constraint solver (Section 4.8).

A ***Determining*** touch is *Effected* by the action and involves no options — it chooses a specific object, part and point of contact for each item. Indeed, it specifies exactly the position Basil must occupy after the action. In Figure 3e, *grasp (B1.center)* involves a pre-determined object and a single point of contact, hence it is *Determining.*

A ***Sufficient*** touch, *Effected* by a *select* action, has one option — the point of contact along the grasped edge. This happens when an object already known to Basil is picked on one of its edges, which A.Sq treats as equivalent to grasping the center or midpoint of that edge.

Weak touches are *Effected* by operators other than *select* and have one, two or three options. In the example above, *touch (L2.? : B1.bottom.left)* results from drag-handle applied to *L2*, with options in the choice of box for *B1* and its point of contact along *L2*.

Unaffected and ***Sustained*** touches are assigned to levels of the same name regardless of options. They are considered of low significance because typically they do not limit

constraint solutions as much as the higher levels. ***Trivial*** touches are *Inevitable* and can have no effect upon constraint solutions.

3. Selection Having assigned each touch to a level, the classifier then selects all those at the highest level present as *Constraints* and marks the rest *Incidental.* If there is no *Determining* constraint, the Path is made a *Criterion* with the caveat that the solver may have to relax it. Should there be no touches at a level higher than *Unaffected,* the classifier signals inadequate constraint. In effect, the solver would require user intervention to produce a specific result, so a default *Criterion,* "ask the user", is adopted.

The classifier signals the interface to display appropriate interaction devices for *Constraints, Incidental* touches and *Criteria,* as described in Section 4.2.

Obviously, the classifier's judgements cannot be guaranteed correct. For instance, in the alignment task variant (Section 2.4), the grasping of a box is governed by a *Determining* constraint, yet the *Unaffected* contacts between a box and its tie-lines should be used to restrict the choice of box to one having a tie-line at the left edge. This points to the need for similarity-based generalization (which is not used) and user intervention (which is supported).

4.8 Constraint solver

Solving constraints is the process by which a predicted *ProgramStep* is realized as an action with specific values for object and part variables and for *CurrentPoint* (i.e. Basil's new location). Figure 12 shows what happens when a line A is to be moved so that any part of it touches both lines B and C. The solution zone is a convex 2-dimensional area within which *CurrentPoint* may lie, such that the stated touch relations hold. A set of constraints is solved by intersecting their zones: this involves trying different combinations of permitted values for variables (consistent across constraints), and intersecting each region with the next until the result is empty (failure) or no constraints remain (success). If no combination of variable values yields a non-empty solution, the action is not performable (see Section 4.3). Otherwise the solver chooses a point within the solution using additional criteria (see below).

Algorithm The solver recursively processes a list of *Constraints.* The initial zone is the entire drawing. For each *Constraint C,* it tries alternative values of variables *owned* by *C* until it finds a combination for which *C's* zone overlaps both the area already computed and the solution to the remaining *Constraints* given the current variable bindings. If no result is non-empty, the solver returns to the previous *Constraint* and tries alternative bindings there. It reports failure if none remain for variables of the first *Constraint;* otherwise it returns a non-empty solution zone.

A *Constraint* "owns" variables that occur in no earlier member of the list. Only these may be re-bound, otherwise the previous zone would be invalidated. Combinations are generated, one at a time, by re-binding one variable and re-initializing those for which no alternative values remain. Since the first object in a touch relation is either Basil or *CurrentObject,* a *Constraint* has at most two variables (object and part) to re-bind. A variable is bound by its *Selector* function, which chooses a value from the *DisplayList* (objects) or object description (parts). As noted in Section 4.6, *Selectors* impose their own constraints on variables; for instance *use-value-of (Var)* permits only one value. *Find-novel-object* uses a *Path* criterion to select only *DisplayList* items whose location is in a certain half-plane relative to Basil.

For instance, in Figure 3g, Basil must solve the following pair of constraints with the given Path criterion:

Path (Upwards)
C1: touch (L2.? : B1.bottom.left)
C2: touch (L2.? : B1.bottom.right)

When processing *C1,* the solver binds the variables it owns: *L2, L2.?, B1* and *B1.bottom.left.* Since *L2's Selector* is *use-value-of (L2),* its value is not changed. *B1's Selector* is *find-novel-object,* so it is assigned the nearest box along the vertical dimension (given by *Path*). The part variables are given parameter ranges (0..1 and 3 respectively) according to definitions in Table 1. When processing *C2,* the solver can bind only one variable, *B1.bottom.right.*

A *Constraint's* zone is a polygon whose vertices are extremal positions that Basil might occupy (e.g. see Figure 12). Consider the touch relation *touch* $(o_1.p_1 : o_2.p_2)$, where o_1 is Basil or *CurrentObject*. Each part p_n can be thought of as having one or two vertices (it is a handle or an edge). Basil is at offset dx_v, dy_v from each vertex v of p_1. For each vertex w of p_2, the solution zone has one or two vertices at $(x_w + dx_v, y_w + dy_v)$.

The solution zones for the example from Figure 3g are detailed in Figure 13. Basil is at *L2.midpt;* hence, if $L = length(L2)$, then $(dx_1, dy_1) = (-L/2, 0)$ and $(dx_2, dy_2) = (L/2, 0)$. For *C1*, p_2 has one vertex *B1.bottom.left* whose coordinates are (x_{bl}, y_{bl}). Thus, *C1's* zone is a line extending from $(x_{bl} + dx_1, y_{bl})$ to $(x_{bl} + dx_2, y_{bl})$. Similarly, *C2's* zone is a line from $(x_{br} + dx_1, y_{br})$ to $(x_{br} + dx_2, y_{br})$.

Since solution zones are (necessarily) convex polygons, a fast clipping algorithm developed by Sutherland and Hodgman (1974) is used to intersect them. Zones that are single points, line segments, or vertical or horizontal lines are treated as special cases to further speed intersection calculations. In Figure 13, where the zones of *C1* and *C2* are horizontal lines, their intersection is a line from $(max\ (x_{bl} + dx_1, x_{br} + dx_1), y_{bl})$ to $(min\ (x_{bl} + dx_2, x_{br} + dx_2), y_{bl})$.

Having processed all constraints, the solver chooses a single point within their intersection as the final solution. If *Path* is a criterion, it chooses the point nearest to Basil's present location. Otherwise, it chooses the zone's centroid. In Figure 13, *Path* is relevant and the nearest point is directly above Basil, $(CurrentPoint.x, y_{bl})$.

The constraint solver always terminates. The combinatorial component's complexity is proportional to the product of the number of variables and the number of feasible values. The numerical component, invoked for each iteration of the combinatorial one, is linear in the number of constraints. *Selector* functions are linear in the number of possible values but in the worst case could be invoked on each iteration of the combinatorial solver. This suggests that potential values should be ordered (for example, *DisplayList* objects could be sorted along the search path), and that feasible values should be memoized.

5 Evaluation

We now establish that Metamouse actually learns procedures from example execution traces, and summarize the results of a study of the extent to which inexperienced teachers understand its behavior. More detailed usability studies will be carried out when our new implementation has been thoroughly debugged.

5.1 Performance of tasks

Concepts learned by Metamouse cannot be judged correct or incorrect because users demonstrate only enough to solve the problem at hand. However, their coverage, robustness and complexity may be examined during coaching sessions. *Coverage* is measured as the ratio of actions correctly predicted to the total number performed by both

user and apprentice. The *rate of learning* is measured as the increase in this ratio from one trace to the next, or between iterations of a loop. *Robustness* is measured as the ratio of incorrect predictions (i.e. ones rejected by the user) to total predictions. *Complexity* is related to the number of edges (i.e. transitions between actions) in the program graph.

Another important performance measure is actual running speed — specifically, delays introduced by matching the user's latest action and by solving for constraints when predicting. Although we have not done detailed timings, we find that for relatively small tasks like those described here, the system (running on a Sun SPARCStation) responds in real time.

In order to establish that Metamouse can infer constraints and procedures from graphical constructions, it was taught the tasks described in Section 2 using the same procedures. This study does not purport to show that typical users of Metamouse would produce demonstrations with similar constructions — although we believe this to be the case.

Each task was demonstrated once, and then invoked several times on different data; the demonstrations were free of incorrect or extraneous actions. It was found that the system quickly achieved competence and constructed simple models. Table 5 summarizes performance data. It compares the total number of actions in each trace with the number correctly predicted by Basil, also shown as a percentage of the total. The count of predictions includes the number of user inputs, noted beside it. The number of predictions the user rejected is also shown. The size of the program graphs is given as the number of edges.

Stove The stove editing task is a simple sequence of actions. No predictions can be made during the first trace as it contains no repeated actions. All actions in the second trace are predicted correctly. One input from the user to move the stove is required.

Sorting For the first trace of the Sorting task, 4 boxes were used; for the second, 5 boxes. By inducing loops, the system was able to perform 56% of the actions in this task the first time the user performed it. Automation increased to 100% on the second trace, with two inputs to edit the spacer.

Different construction tactics for sorting and spacing can be used: their effectiveness depends on exploiting Basil's model of constraints (of which the user is well informed) while remaining within the system's inferencing limits (of which she is not). "Violations" of the latter condition can have unpredictable results. For instance, suppose the sweepline is eliminated when sorting. This makes no real difference since Basil orders the *find-novel-object* selections by distance along the axis of *Path*. It would help the user understand Basil better, however, if a sweepline were generated automatically when using this selector.

A more serious misunderstanding occurs when the first box is placed to the left of the spacing tool, with the remainder to the right. Basil predicts that the second box should go to the left like the first. When the user rejects this, a branch is formed that gives priority to putting the currently selected box at the right. On the next invocation, Basil puts the first box at the right: when the user rejects this, he tries his alternative prediction, which is accepted, causing the priorities to be reversed again. Thus on every invocation Basil handles the first two boxes incorrectly (unless the user accepts his putting the first box at the right!). This problem arises from using a fixed number of action matches (namely, one) to induce a loop. We have created a new learning algorithm that is capable of extending the match context *post hoc* so that such loops can be split; this will be incorporated into the next version of Metamouse.

Aligning boxes Five traces of the alignment task were produced. The first involved three boxes as in Section 2.3; the second was run on four boxes; the third and fourth introduced and repeated the variant described in Section 2.4; the final trace was a repetition

of the second. Basil was able to learn the variant and yet retain the ability to do the original task. In the first trace, the system predicted the second and third iterations of the loop, or 40% of the work. In the second trace, it processed all four boxes, with two inputs for the guideline's end-points. The third trace adapted the alignment procedure to the task illustrated in Figure 4; 56% of its actions were predicted, while steps demonstrated by the user introduced 15 new state transitions (see Figure 5). During training, Basil made three faulty predictions. On the first iteration, he went to the sweepline after editing the box; the user rejected this and edited the tie-line. On the second iteration, when the sweepline touched two boxes, he picked the one on the left, but the user rejected this and chose the one on the right. After the final iteration, he predicted the end of the task, but the user edited the vertical tie-lines.

Since new actions are given priority over old ones, Basil was able to repeat the variant in trace 4 without error. On the other hand, since actions are predicted only if their constraints can be solved, Basil was able to repeat the original task in trace 5 without making irrelevant predictions concerning non-existent tie-lines.

5.2 Evaluating interaction

A critical aspect of a learning system is that the teacher must understand its behavior (MacDonald & Witten, 1987). The suitability of the Basil metaphor is measured as the ease with which teachers learn to predict what it will do. This has been studied in two questionnaire-based experiments (reported in more detail by Maulsby *et al.*, 1989b). The first, a pilot study without controls, was intended to establish the viability of a questionnaire. The second introduced controls on the amount of prior knowledge subjects were given regarding the metaphor, and also measured correlations with previous computing experience. The results of both experiments show that even without live interaction Basil's behavior is largely self-explanatory or easily rationalized. They do not, however, directly address the issue of creating procedures by coaching Basil.

Pilot study

In the pilot study, subjects were given a brief description of Basil (an earlier version of Figure 9) and then asked to work through a self-study guide. Typical questions depict a situation and ask the subject to predict Basil's response. Correct answers were provided after each page of questions to simulate system feedback. A sample page is shown in Figure 14: the subject is asked to state the discriminations he or she believes Basil would make between touch relations. The experiment was run with eight volunteer subjects who worked at their own pace.

If the metaphor were difficult to understand, one would expect numerous errors in early questions, with at best a slow improvement. If completely obvious, one would expect near-perfect performance from the beginning with no degradation. The results show excellent performance initially, with occasional mistakes and difficult spots after which near-perfect performance is restored. It was concluded 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 obvious, but learnable.

Controlled study

A follow-up study investigated two hypotheses: first, that the amount of explanation of Basil's behavior given prior to examples of it does not significantly affect its predictability; and second, that prior experience with computer systems does not significantly affect comprehension of the metaphor. The first hypothesis was not disproven, indicating that the metaphor communicates essential aspects of the system's operation intuitively. The second was contradicted, but it was found that the most useful *types* of experience were of

graphical interfaces and drawing programs, as opposed to computer programming and spreadsheets.

The subjects of this study were students of architecture and industrial design (16 responses received) and first-year computer science (20 responses).

Several controls were introduced. First, the introductory material was varied. One version of the questionnaire contained the full description of Basil (Figure 9). A "less informative" version came with a two-page worked example of Basil learning a simple task. A minimal version provided a meagre one-paragraph explanation of terms used in the questionnaire. This variation had no significant impact on subjects' overall scores, nor on scores for the first page of the questionnaire. Second, the order of questions was varied, in order to simulate interaction rather than guided study; this was found to have no significant effect on performance. Third, some subjects were given no answer key (i.e. no feedback); this control group was eliminated due to lack of response.

6 Future work

A project that combines machine learning, constraint solving and graphical interaction affords many avenues for further research. Some of these relate to improvements in and evaluations of Metamouse, others to the wider problems of programming by demonstration. We consider the following projects most important:

- Perform usability studies on Metamouse to determine whether novice users can program tasks by demonstration using graphical constructions.
- Perform ergonomic studies, measuring improvements in task execution time achieved through programming by demonstration.
- Develop a richer set of object selector functions and provide an interface similar to that for constraints (marked by tacks), so that the user can see and alter Basil's hypothesis.
- Implement the formation of conditional branches based on the preconditions of actions.
- Augment the system with similarity-based learning of constraints and selector functions, to reduce spurious branching and increase predictions.
- Extend A.Sq to include circles, polygons, and rotation; this will necessitate changes to the constraint solver since solution zones will no longer be restricted to convex polygons.
- Introduce orientation-dependent naming of object parts (e.g. leftmost end of line); use similarity-based learning and allow direct user access to choose the appropriate selector.
- Induce certain common spatial relations (such as alignment along a major axis) so that construction is not always required.
- If Metamouse infers a spatial relation or an ordering of selections along a path, express it by generating a tool (like a sweepline) so that users will learn from Basil how to make appropriate constructions.
- Investigate the use of a different modality (e.g. voice) for the dialog with Basil; this would enhance the separation in the user's mind between the application and the apprentice.

- Develop a cleaner and more elegant theory of constraints in a drawing world, without sacrificing predictive power.
- Provide a clearer separation between the programming-by-example method and the application domain, and test the method's viability in other domains.
- Develop a more layered approach to implementation that reduces its complexity and general unwieldiness.

We are beginning to address a number of these issues within the framework of an "instructible system" — one that combines inference from examples with direct instructions from the user.

7 Conclusions

The nature of Metamouse raises several important questions. The system is designed to build a predictive model of human performance by conjecturing intentions behind isolated actions. It is illuminating to consider what kinds of procedure Metamouse can and cannot learn. In a trivial sense, the system is "Turing complete": one can teach it to emulate any finite-state automaton, including the control for a Turing machine, and give it access to a graphical "memory" of linear structure and arbitrary size. The issue then becomes one of teachability rather than learnability: what users find natural to teach rather than what can in principle be taught.

Some tasks cannot be represented without creating unnatural objects to support the computation—a good example is counted loops, where an external graphical counter can in principle be constructed but is tedious to create, update, and test. Other tasks may present teachers with an unreasonable mental burden, for example reference to higher-order indirect touches and demonstrating many different cases by iterating over each one in turn. Further problems arise from the difficulty of structuring programs and the fact that subprocedures are not supported. Perhaps it is reasonable to assume an upper limit on the complexity of any program that it is worth teaching a system that does not provide an externalized, written record.

Metamouse constructs non-deterministic procedures that use constraints and, as a last resort, recency to disambiguate alternative branches at run time. This has striking benefits when debugging, extending and reusing procedures. It also finesses the problem of prematurely formed loops, formed when a sequence includes matching subsequences that are long enough to satisfy loop confirmation.

The preliminary human factors experiments on Metamouse's usability may lead to modifications of the user interface. In fact, subjects of the pilot experiment had difficulty in predicting Basil's sensory discriminations and classification of constraints, and this led to a more telling graphical representation for Basil's sensory feedback (Maulsby *et al.,* 1989c).

Metamouse demonstrates that it is indeed possible for users to create graphical procedures by direct manipulation. Applications range from producing complex, repetitive drawings, through constructively specifying figures governed by graphical constraint, to generating simple animated algorithms for tasks such as sorting. Metamouse reveals its predictions as soon as it can. This has three advantages. First, users reap early benefits when performing repetitive operations. Second, they can correct errors as soon as they occur. Third, they develop confidence in their programs without ever viewing any kind of listing. The principal shortcomings of the current system are its limited repertoire of graphical objects and transformations, the lack of a formal underpinning for the constraint model, and our limited experience of how users react to the new experience of working with Metamouse.

Acknowledgements

This research is supported by the Natural Sciences and Engineering Research Council of Canada and by Apple Computer Inc. We gratefully acknowledge the key role Bruce MacDonald has played in helping us to develop our ideas, and the stimulating research environment provided by the Knowledge Science Lab at the University of Calgary. We have benefited from contributions to this work by Rosanna Heise, Fritz Huber, Greg James and Antonija Mitrovic. Many thanks are due to Allen Cypher, Ted Kaehler, Alan Kay, David Kosbie, John Matheny and Dave Smith for their insightful comments. Finally, we thank the editor and reviewers for their discriminating counsel.

References

Abbott, E.A. (1884). *Flatland—a romance of many dimensions.* Signet Classics edition, New York.

Andreae, P.M. (1985). *Justified generalization: acquiring procedures from examples.* Unpublished PhD Dissertation, Department of Electrical Engineering and Computer Science, MIT, Boston, Massachusetts.

Angluin, D. & Smith, C.H. (1983). Inductive inference: theory and methods. *Computing Surveys, 3*, 15, 237–269.

Bier, E.A. & Stone, M.C. (1986). Snap-dragging. *Proceedings of ACM SIGGRAPH,* 233–240. Dallas, Texas.

Borning, A. (1981). The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems, 3*, 4, 353–387.

Borning, A. (1986). Defining constraints graphically. *Proceedings of ACM SIGCHI,* 137–143. Boston, Massachusetts.

Cutter, M., Halpern, B., & Spiegel, J. (1985) *MacDraw.* Apple Computer.

Dennett, D.C. (1987). *The intentional stance.* MIT Press, Cambridge, Massachusetts.

Ellman, T. (1989) "Explanation-based learning: a survey of programs and perspectives," *Computing Surveys 21*(2): 163–221; June.

Fikes, R. E., Nilsson, N. J. "STRIPS — a new approach to the application of theorem proving to problem solving." *Artificial Intelligence,* vol. 2, pp. 189-288. 1971.

Foley, J.D. & van Dam, A. (1982). *Fundamentals of interactive computer graphics.* Addison–Wesley, Reading, Massachusetts.

Fuller, N. & Prusinkiewicz, P. (1988). Geometric modeling with Euclidean constructions. *Proceedings of Computer Graphics International,* 379–391. Geneva, Switzerland.

Halbert, D. (1984). *Programming by example* (Research Report OSD–T8402). Xerox PARC, Palo Alto, California.

Heise, R. (1989). *Demonstration instead of programming.* Unpublished MSc Thesis, Department of Computer Science, University of Calgary, Canada.

Johnson, J., Roberts, T.L., Verplank, W., Smith, D.C., Irby, C., Beard, M., & Mackey, K. (1989). The Xerox Star: a retrospective. *IEEE Computer, 22*, 9, 11–29.

Kin, N., Noma, T., & Kunii, T.L. (1989). PictureEditor: A 2D picture editing system based on geometric constructions and constraints. *Proceedings of Computer Graphics International,* 193–207. Leeds, England.

Kurlander, D. & Bier, E.A. (1988). Graphical search and replace. *Proceedings of ACM SIGGRAPH*, 113–120. Atlanta, Georgia.

MacDonald, B. A. & Witten, I. H. (1987). Programming computer controlled systems by non-experts. *Proceedings of the IEEE SMC Annual Conference,* 432–437. Alexandria, Virginia.

Maulsby, D.L., Kittlitz, K.A., & Witten, I.H. (1989a). Constraint-solving in interactive graphics: a user-friendly approach. *Proceedings of Computer Graphics International*, 305–318. Leeds, England.

Maulsby, D.L., James, G.A., & Witten, I.H. (1989b). Acquiring graphical know-how: an apprenticeship model. *Proceedings of European Knowledge Acquisition Workshop,* 406-419. Paris, France.

Maulsby, D.L., Kittlitz, K.A., & Witten, I.H. (1989c). Metamouse: specifying graphical procedures by example. *Proceedings of ACM SIGGRAPH*, 127–136. Boston, Massachusetts.

Mitchell, T.M. (1982). Generalization as search. *Artificial Intelligence, 18*, 2, 203–226.

Myers, B.A. (1987). Creating dynamic interaction techniques by demonstration. *Proceedings of ACM SIGCHI.* Toronto, Canada.

Myers, B.A. (1988). *Creating user interfaces by demonstration.* Academic Press, San Diego.

Noma, T., Kunii, T.L., Kin, N., Enomoto, H., Aso, E., & Yamamoto, T.Y. (1988). Drawing input through geometrical constructions: specification and applications. *Proceedings of Computer Graphics International*, 403–415. Geneva, Switzerland.

Pence, J. & Wakefield, C. (1988). *Tempo II.* Affinity MicroSystems, Boulder, Colorado.

Preparata, F.P. & Shamos, M.I. (1985). *Computational geometry.* Springer–Verlag, New York.

Rich, C. & Waters, R. (1988). The programmer's apprentice: a research overview. *IEEE Computer, 21*, 11, 11–25.

Smith, D.C., Irby, C., Kimball, R., Verplank, W., & Harslem, E. (1982). Designing the Star user interface. *Byte, 7*, 4, 242–282.

Sutherland, I.E. (1963). Sketchpad: a man-machine graphical communication system. *Proc. AFIPS Spring Joint Computer Conference, 23*, 329–246.

Sutherland, I.E. & Hodgman, G.W. (1974). Reentrant polygon clipping. *Communications of the ACM 17, 1.*

van Lehn, K. (1983) *Felicity conditions for human skill acquisition: validating an AI-based theory* (Research Report CIS–21). Xerox PARC, Palo Alto, California.

van Sommers, P. (1984). *Drawing and cognition.* Cambridge University Press, Cambridge, England.

White, R.M. (1988). Applying direct manipulation to geometric construction systems. *Proceedings of Computer Graphics International*, 446–455. Geneva, Switzerland.

Williams, G. (1984). The Apple Macintosh computer. *Byte, 9*, 2, 30–54.

Witten, I.H. and MacDonald, B.A. (1988) "Using concept learning for knowledge acquisition," *International J Man-Machine Studies 29*(2): 171–196; August.

List of Tables and Figures

Table 1 Elements of the A.Sq drawing program
Table 2 Selector functions
Table 3 Decision tree for identifying causality of touch relation in action
Table 4 Decision tree for classifying the level of touch constraint
Table 5 Performance of the learning system on three tasks

Figure 1 Maintaining constraints amongst objects
Figure 2 A group of boxes is sorted by height
Figure 3 Teaching Basil to align a set of boxes
Figure 4 Aligning boxes and editing tie-lines
Figure 5 Program induced for "aligning boxes" task and variant
Figure 6 Visualizing constraints
Figure 7 Highlighting distinguished points near cursor (arrowhead) while rubber-banding a line
Figure 8 Selector functions for parts of an object
Figure 9 Description of Metamouse given to users
Figure 10 Touch constraints are marked by tacks, which the user may push in or pull out
Figure 11 Components of learning system
Figure 12 Intersection of solution zones for competing constraints
Figure 13 Calculating solutions for constraints between a sweepline and a box
Figure 14 Sample page from questionnaire, with correct answers

Graphic object types	• box : specified by top.left=(x1,y1), bottom.right=(x2,y2); edge coordinates 0 ≤ top ≤ 1 ≤ right ≤ 2 ≤ bottom ≤ 3 ≤ left ≤ 4; handles at each vertex, mid-point of edge, and center (coordinate 5). • line : specified by start=(x1,y1), end=(x2,y2); edge coordinates 0…1 (eg. midpt=0.5); handles at start, end and midpt.
Auxiliary objects	• Mode : {create-boxes, create-lines, edit-objects} • CurrentPoint : (x,y) location most recently selected by user • PreviousPoint : previous value of CurrentPoint • CurrentObject : graphic object most recently selected by user • Handle : currently selected (activated) handle of CurrentObject • DisplayList : list of graphic objects in drawing • ActionStack : list of actions done or re-done (see below) • UndoneStack : list of actions undone (see below)
Drawing operators	Note: arguments marked « are accessed; » are set; * reference objects that are altered. • set-mode («{create-boxes, create-lines, edit-objects}, »Mode) • select-point («X, «Y, »PreviousPoint, «»CurrentPoint) • select-object («DisplayList, «CurrentPoint, »CurrentObject) • select-handle («CurrentPoint, «CurrentObject, »Handle) • new-line («CurrentPoint, »*CurrentObject, »DisplayList) • new-box («CurrentPoint, »*CurrentObject, »DisplayList) • translate-handle-of-object-to-point (»PreviousPoint, »CurrentPoint, «*Handle, «*CurrentObject) • delete-object («»CurrentObject)
Action operators	• undo («ActionStack, »UndoneStack) • redo («UndoneStack, »ActionStack) • define-action («Operator, «PreviousPoint, «CurrentPoint, «CurrentObject, »*Action, »ActionStack)

Table 1. Elements of the A.Sq drawing program

created (Type)	This step created the object.
use-value-of (Var)	The object appears as the binding of some variable *Var.*
find-named-part (PartName)	Returns the edge coordinates of the named part.
find-novel-object (PreviousBindings, Type, Path)	There is no other reference to the object in the current binding environment—it is encountered for the first time.

Table 2. Selector functions

Operator	Locality of touch	Both parts stationary	Relation is changed	Type of effect
locate or select	grasp or direct	—	no	Sustained
			yes	Effected
	indirect	—	—	Unaffected
create-line or create-box or drag-handle	grasp	—	—	Inevitable
	direct	—	no	Sustained
			yes	Effected
	indirect	yes	—	Inevitable
		no	no	Sustained
			yes	Effected

Table 3. Decision table classifying ways a touch relation results from an action

Free variables	Type of effect	Operator	Level
$n = 0$	Effected	—	Determining
	Unaffected	—	Unaffected
	Inevitable	—	Trivial
	Sustained	—	ERROR
$n > 0$	Effected	select	Sufficient
		<any other>	Weak *n*
	Unaffected	—	Unaffected
	Sustained	—	Sustained
	Inevitable	—	ERROR

Table 4. Decision table for the level of touch constraint

Task		Actions performed				Size of
	Trace	Total	Predict : Inputs	Ratio	Rejected	program
Stove	1	10	0	0%	0	11
	2	10	10 : 1	90%	0	11
Sorting	1	32	18	56%	1	21
	2	38	38 : 2	100%	0	22
Align boxes	1	20	8	40%	0	13
	2	24	24 : 2	100%	0	13
… variant	3	32	18 : 2	56%	3	28
	4	32	32 : 2	100%	0	28
… original	5	24	24 : 2	100%	0	28

Table 5. Performance of the learning system on three tasks

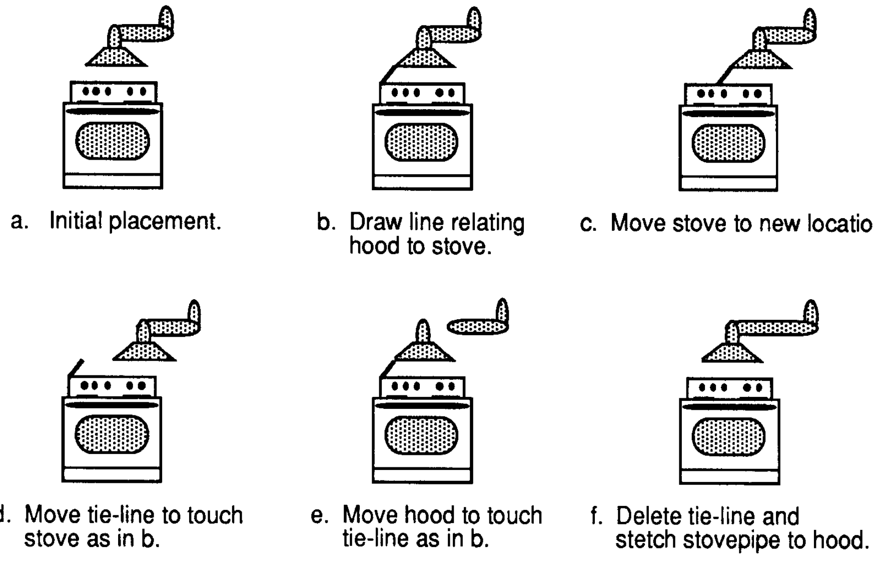


Figure 1. Maintaining constraints amongst objects

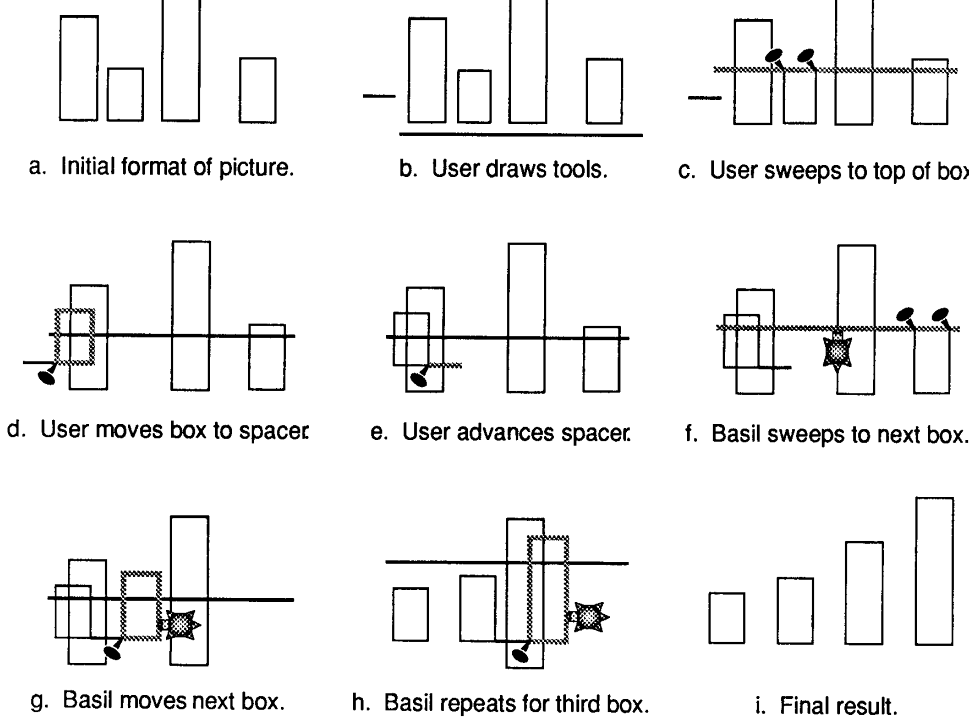


Figure 2. A group of boxes is sorted by height

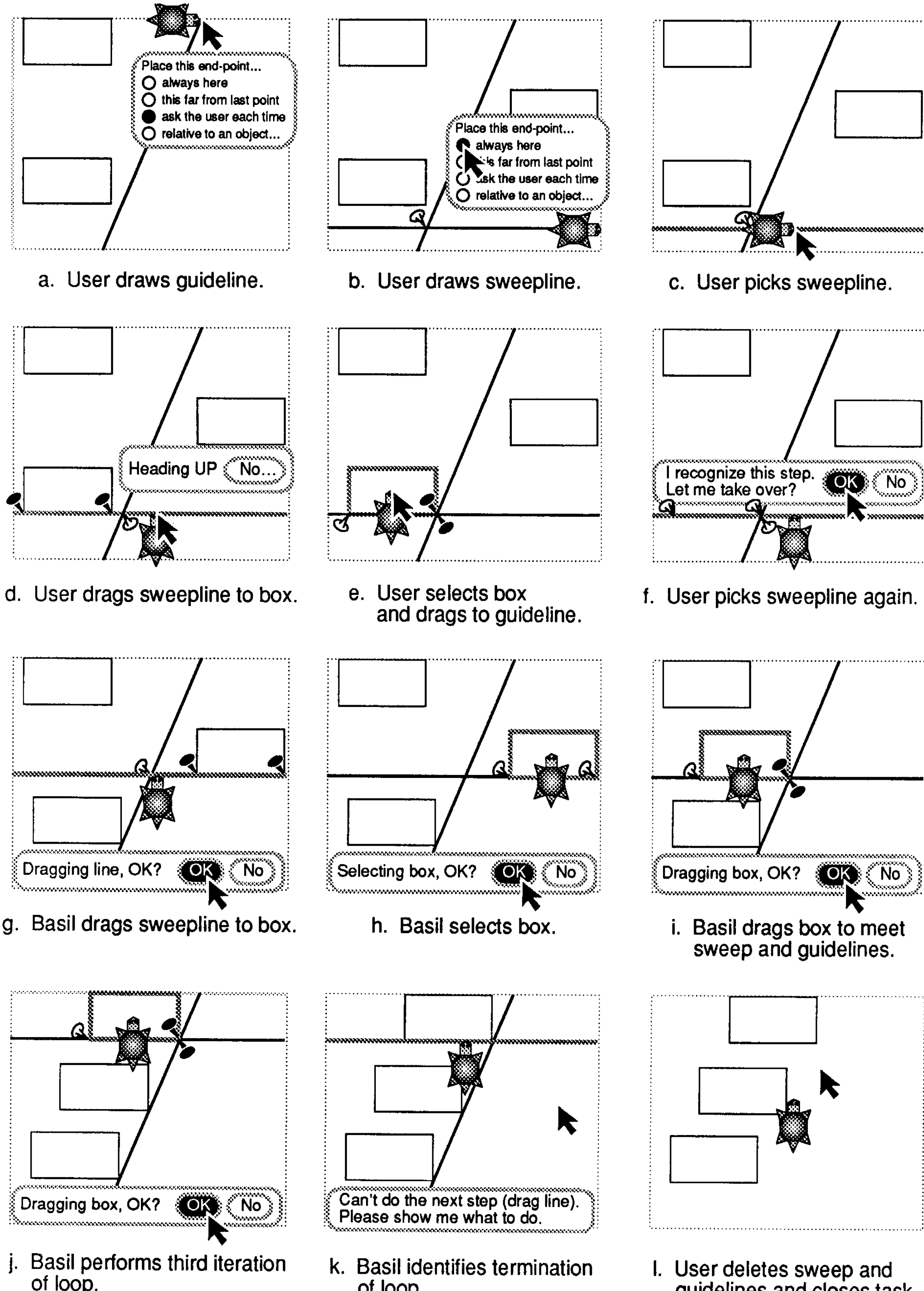


Figure 3. Teaching Basil to align a set of boxes

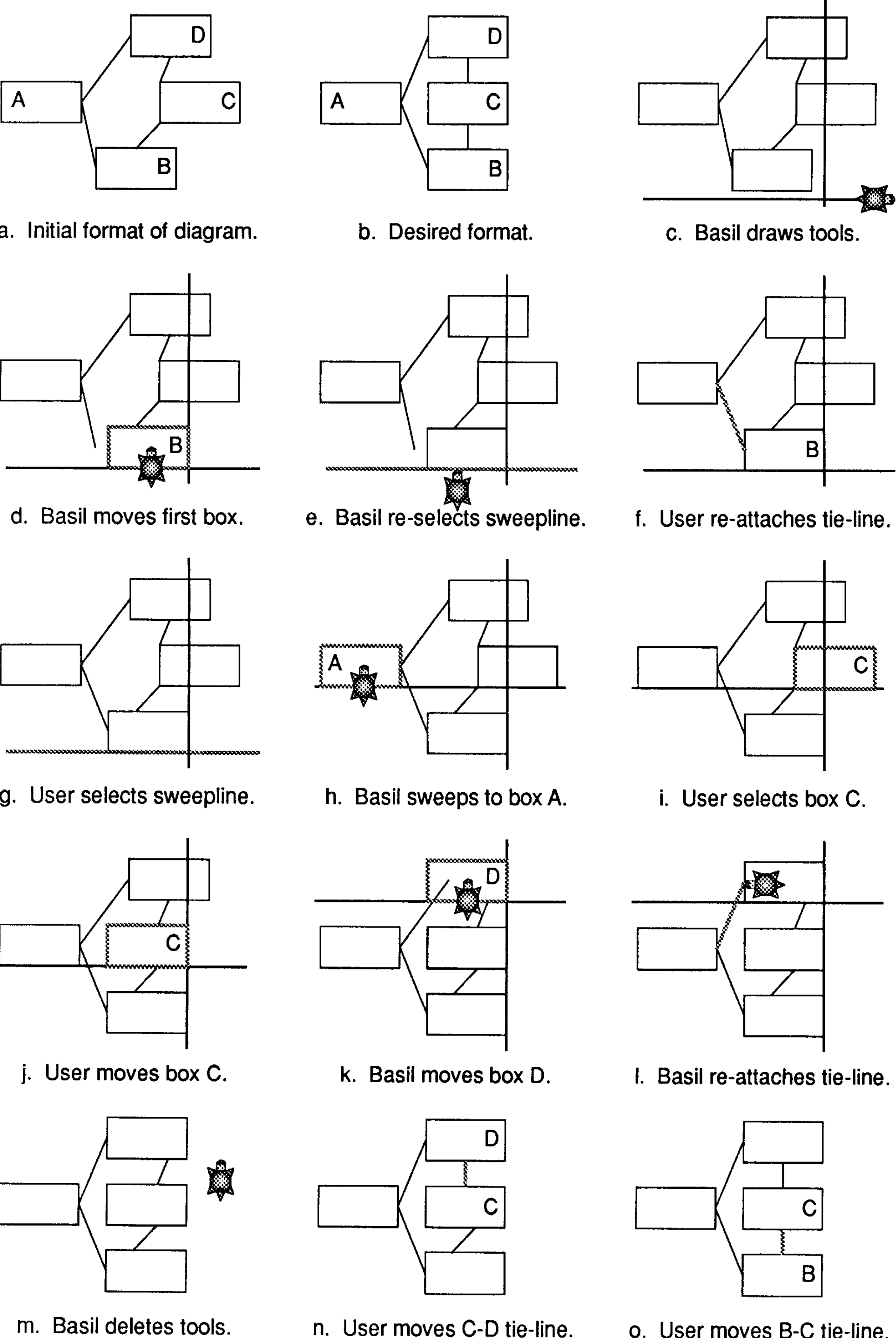


Figure 4. Aligning boxes and editing tie-lines

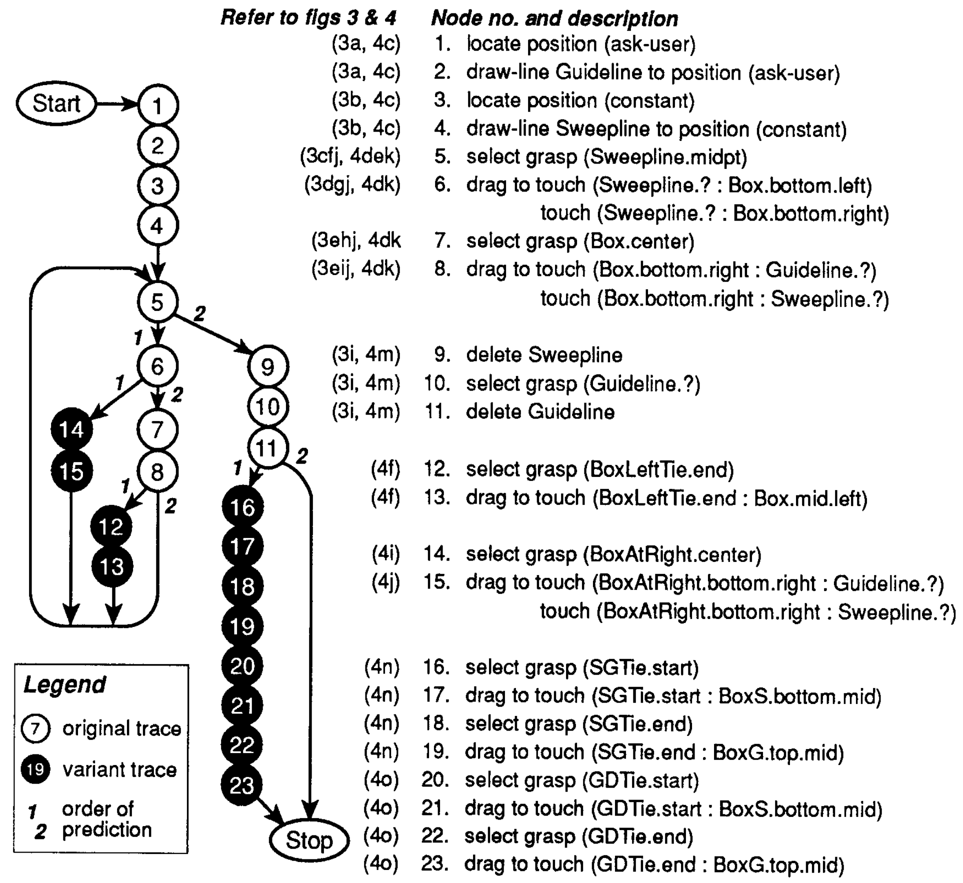
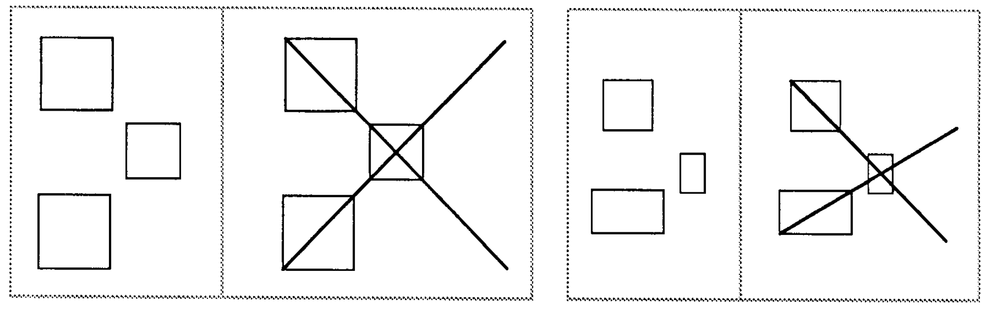


Figure 5. Program induced for “aligning boxes” task and variant

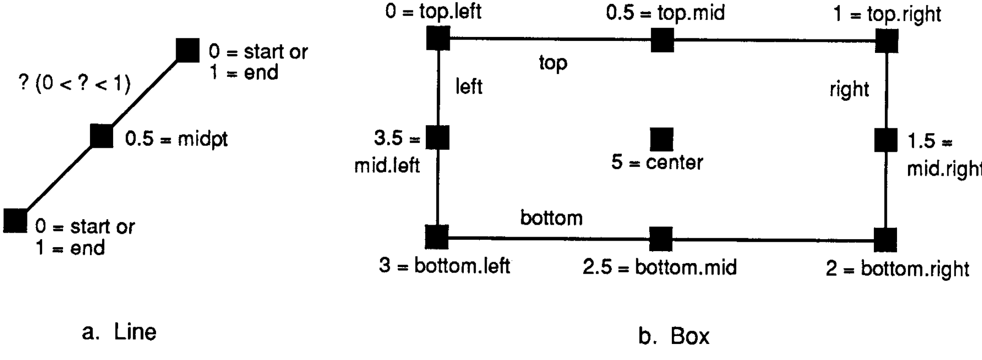


a. Arrangement. b. Constraints made visible. c. Second example isolates constraints.

Figure 6. Visualizing constraints

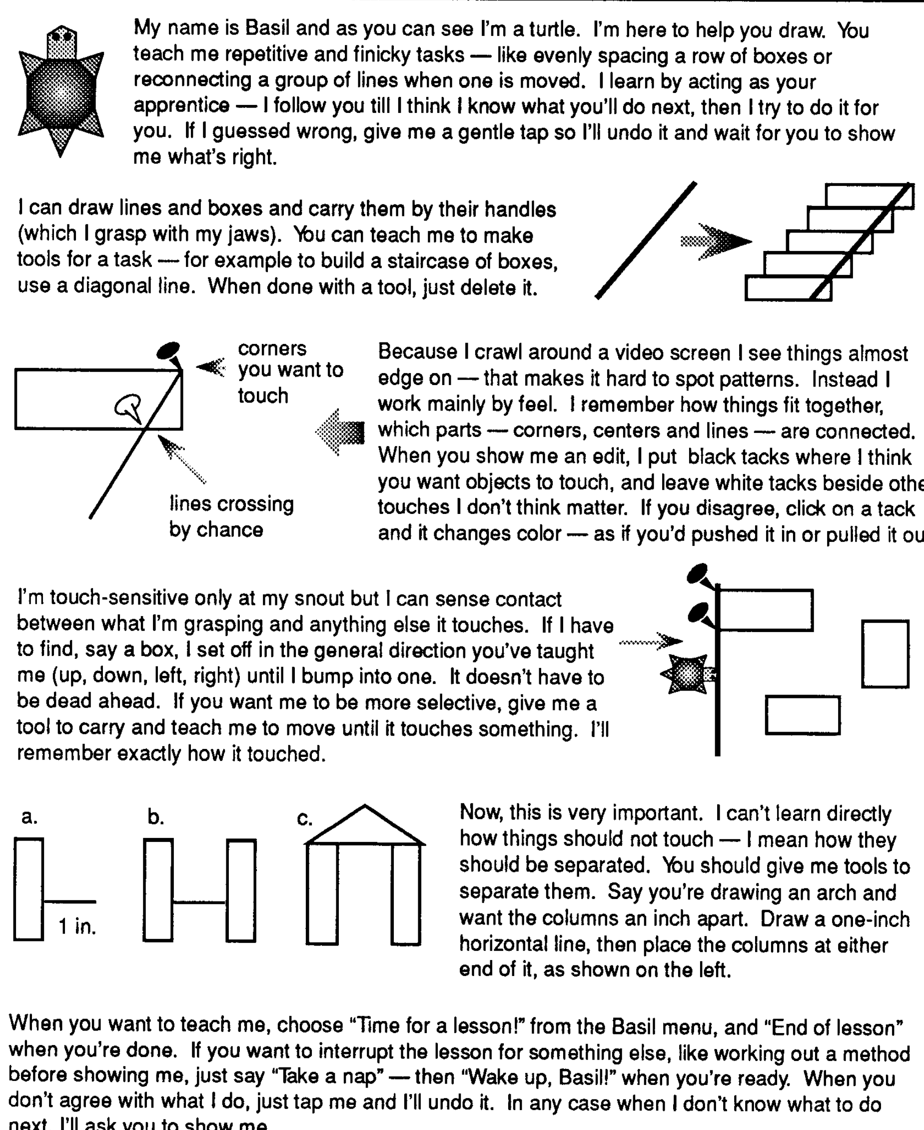
a. Near edge of box. b. Near vertex of two lines.

Figure 7. Highlighting distinguished points near cursor (arrowhead) while rubber-banding a line



a. Line b. Box

Figure 8. Selector functions for part of an object



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 a row of boxes or reconnecting a group of lines when one is moved. I learn by acting as your apprentice — I follow you till I think I know what you'll do next, then I try to do it for you. If I guessed wrong, give me a gentle tap so I'll undo it and wait for you to show me what's right.

I can draw lines and boxes and carry them by their handles (which I grasp with my jaws). You can teach me to make tools for a task — for example to build a staircase of boxes, use a diagonal line. When done with a tool, just delete it.

Because I crawl around a video screen I see things almost edge on — that makes it hard to spot patterns. Instead I work mainly by feel. I remember how things fit together, which parts — corners, centers and lines — are connected. When you show me an edit, I put black tacks where I think you want objects to touch, and leave white tacks beside other touches I don't think matter. If you disagree, click on a tack and it changes color — as if you'd pushed it in or pulled it out.

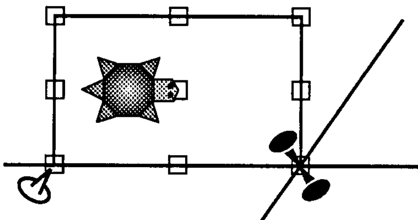
I'm touch-sensitive only at my snout but I can sense contact between what I'm grasping and anything else it touches. If I have to find, say a box, I set off in the general direction you've taught me (up, down, left, right) until I bump into one. It doesn't have to be dead ahead. If you want me to be more selective, give me a tool to carry and teach me to move until it touches something. I'll remember exactly how it touched.

Now, this is very important. I can't learn directly how things should not touch — I mean how they should be separated. 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, then place the columns at either end of it, as shown on the left.

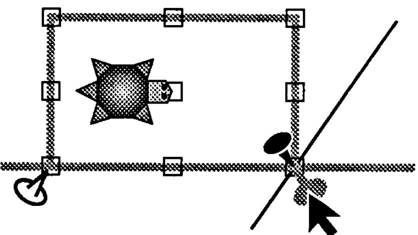
When you want to teach me, choose "Time for a lesson!" from the Basil menu, and "End of lesson" when you're done. If you want to interrupt the lesson for something else, like working out a method before showing me, just say "Take a nap" — then "Wake up, Basil!" when you're ready. When you don't agree with what I do, just tap me and I'll undo it. In any case when I don't know what to do next, I'll ask you to show me.

So in general you teach me by doing the task yourself, using some extra tools to help me see patterns by feel. As soon as I can predict what to do, I'll take over the task, but I'm always ready to learn something new.

Figure 9. Description of Metamouse given to users



a. Box is moved to intersection of lines. Basil marks constraints.

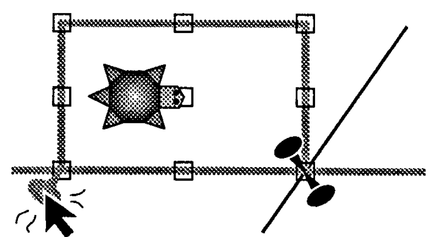


b. User moves cursor onto tack; objects attached to it are highlighted.

Legend: Icons for touch relations

Important constraint

Touch deemed irrelevant



c. User presses tack at disregarded touch; it becomes a constraint.

Figure 10. Touch constraints are marked by tacks, which the user may push in or pull out

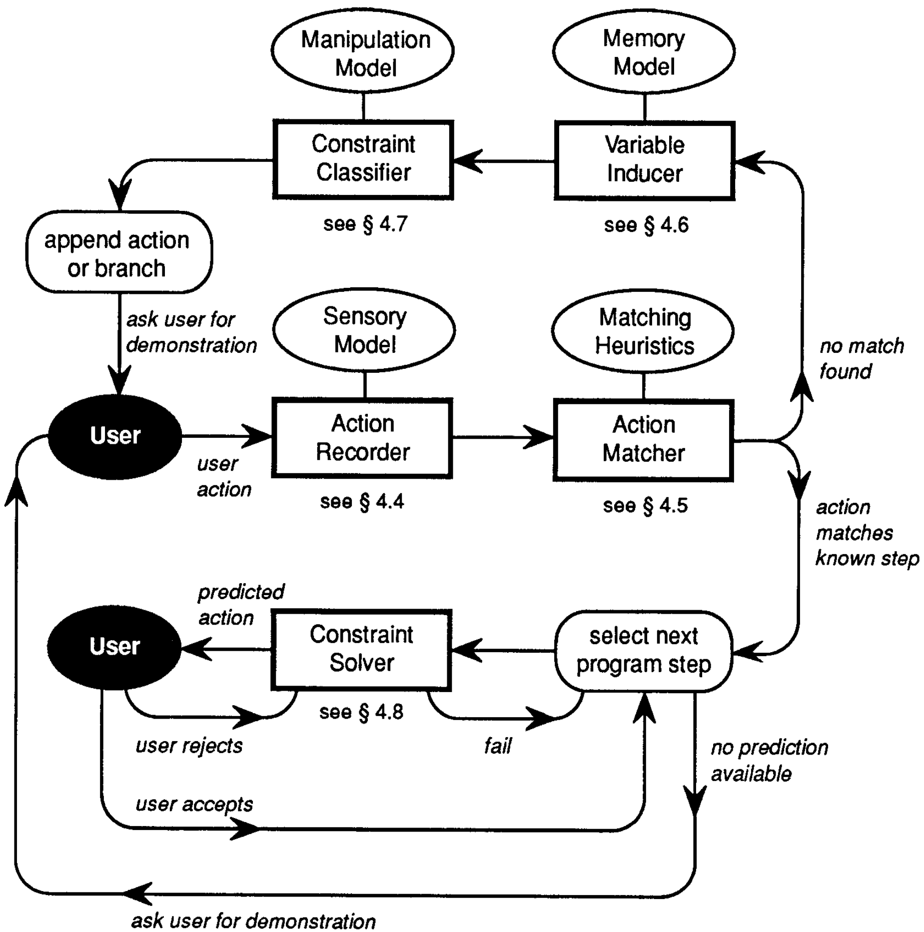


Figure 11. Components of learning system

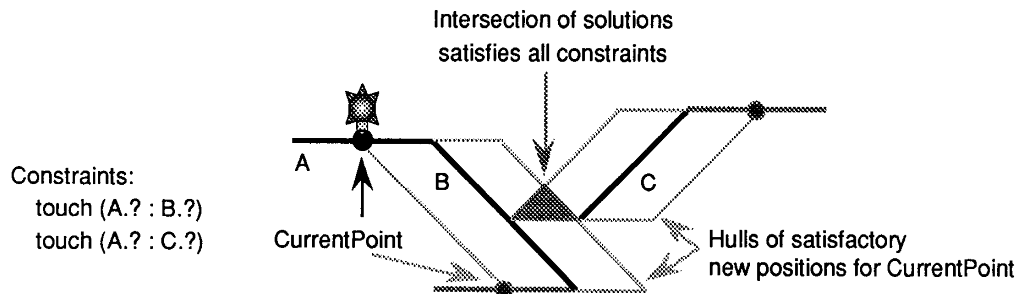


Figure 12. Intersection of solution zones for competing constraints

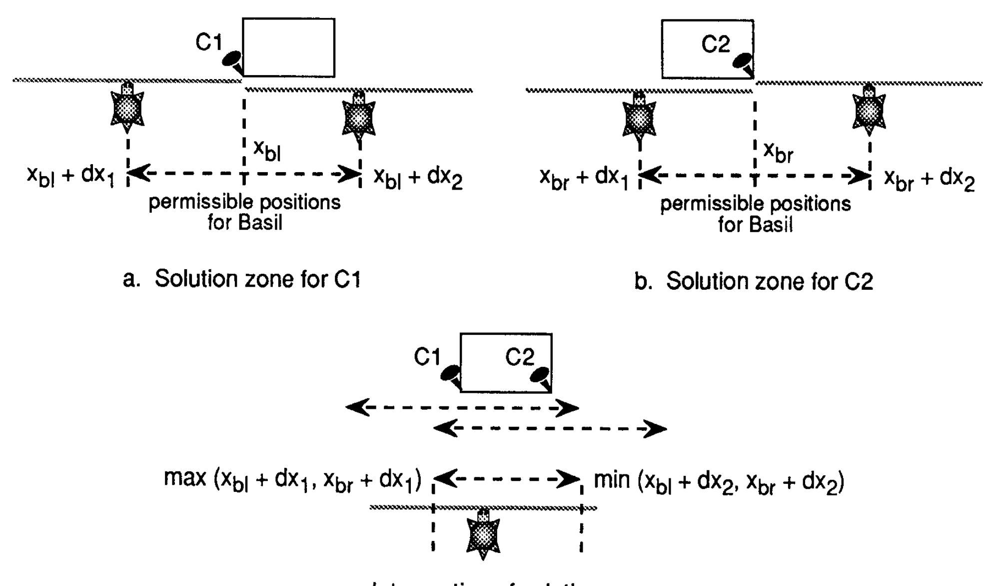


Figure 13. Calculating solutions for constraints between a sweepline and a box

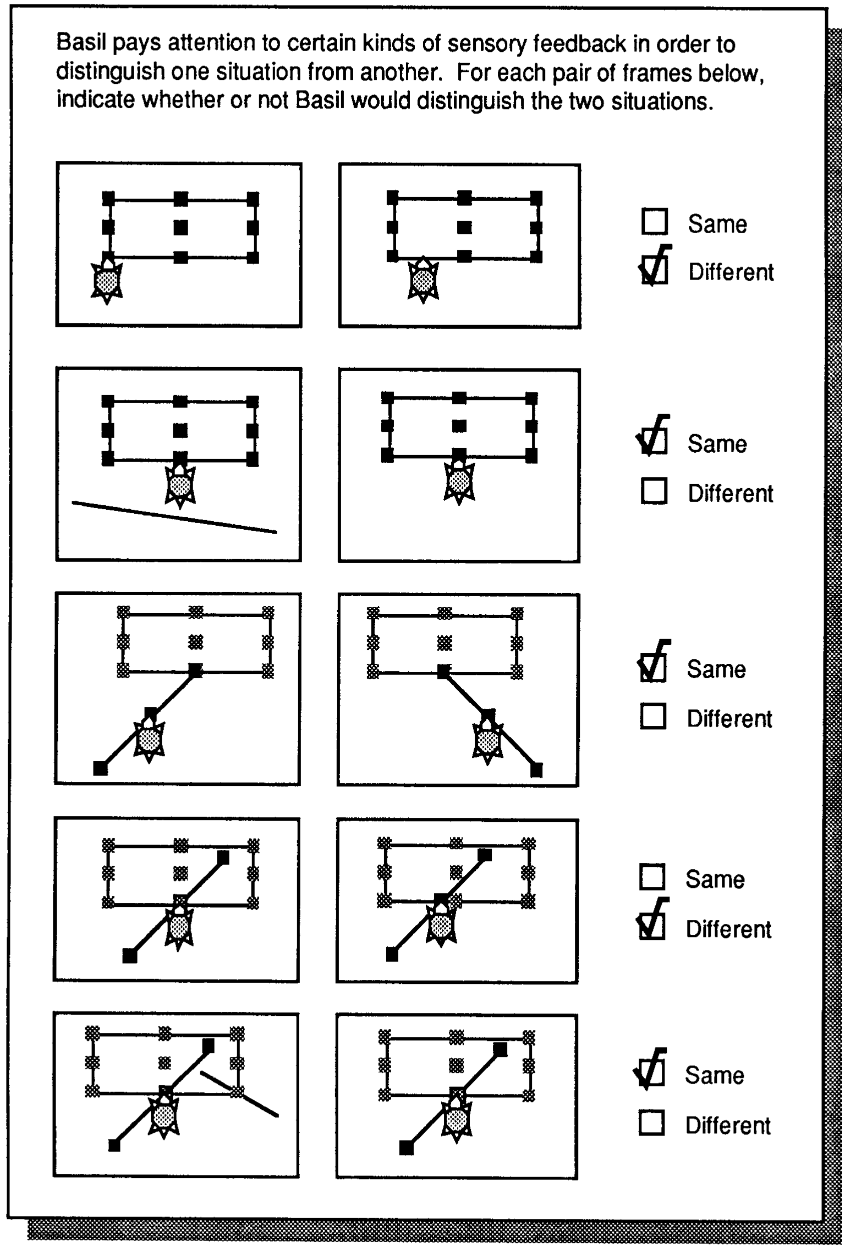


Figure 14. Sample page from questionnaire, with correct answers