Exploring the Potential for Conversational Analysis in the Evaluation of Interactive Algorithm Visualization Systems

Christopher Hundhausen
Department of Computer & Information Science
University of Oregon
Eugene, OR 97403
June, 1993

Exploring the Potential for Conversational Analysis in the Evaluation of Interactive Algorithm Visualization Systems

1 Introduction

Predicated on the intuitive idea that one can gain insight into a computer algorithm through a mapping from its fundamental abstractions to computer graphics, algorithm visualization (AV) can be defined as the process of viewing an algorithm through a series of pictures, or through a dynamic movie, constructed both to illustrate the algorithm's dynamic behavior, and to illuminate the logic that underlies that behavior. Pioneered by Brown University's Electronic Classroom project (Van Dam 1984), and formalized by Marc Brown's seminal dissertation *Algorithm Animation* (1988), AV has gained an enthusiastic following among undergraduate computer science educators, who have come to see it as an effective and innovative method for teaching algorithms. Although computer science educators have made strong claims about AV's effectiveness (cf. Naps & Hundhausen 1991), their enthusiasm for the technique has no empirical basis. In a culture in which the proverb "A picture is worth 10,000 words" is held so dearly, it should be no surprise that a learning technique designed to exploit the visual sense has been embraced so widely; indeed, thus far, its powerful intuitive basis has been sufficient to justify its use.

In response to the lack of an empirical basis for AV, Stasko, Badre, and Lewis (1993) recently documented a study—the first of its kind—that purported to substantiate empirically the widely held intuition about AV's effectiveness. Unfortunately, the results of that study, which used post-test scores as a basis for comparing the understanding of two groups of students who studied an algorithm through different media (text-only versus text-and-animation), revealed no measurable advantage to AV. The reason for the disappointing results, as we argue elsewhere (Hundhausen 1993), lay in a fundamental incongruity between the comparative research methodology employed by the study, and its stated goals.

Proceeding from a recognition that, in implicitly defining a visual language, an interactive AV system requires its users to engage in three distinct activities: (1) conceptualizing a visualization for the algorithm; (2) mapping that conceptualization to the AV language; and (3) manipulating the AV system's interface to translate the conceptualization into the AV language, this paper reports our foray into addressing the inadequacies of the research methodology used by the Stasko, Badre, and Lewis study. In particular, we describe our initial experiences with employing an alternative research methodology, derived from Conversational Analysis (CA), to evaluate the Lens interactive AV system (Mukherjea & Stasko 1993). Section 2 documents the methodology, focus, and format of our study in greater detail. Using a sample transcription from our study as a foundation, Section 3 identifies some of the most significant observations on the Lens interface that we were able to glean from our analysis. Drawing from those observations, Section 4 suggests some ways of improving Lens's user interface, and considers the broader question of designing and evaluating more usable interactive AV systems.

2 The study

In this section, we consider some of the important details of our Lens study. We begin, in Section 2.1, by providing an overview of the research methodology on which our study is based. Section 2.2 introduces the Lens software system, and its intended user domain. In Section 2.3, we discuss the format of our study, and examine the particular exercises that we had subjects perform. We conclude, in Section 2.4, by presenting our goals for the study.

2.1 Methodology

As advocated by several ethnomethodologically-minded researchers (cf. Suchman 1987; Roschelle 1990; Douglas 1992), we employed a qualitative research technique called *constructive interaction* (Miyake 1986), in which two person, same-sex teams (Douglas 1992) are videotaped as they interact with software to complete a given exercise—often designed to expose the team to the full functionality of the software. As teams collaborate to construct shared meaning both of the procedures by which one interacts with the software, and of the underlying concepts that are conveyed through that interaction, they often make their understanding or misunderstanding of a particular task or concept inferable, if not explicit. Detailed analyses of the tapes of these learning episodes, which often give rise to transcriptions like the one we shall scrutinize in Section 3, can give software designers insight into (1) the ability of their software to act as a resource in mediating and facilitating shared understanding (cf. Roschelle 1990, pp. 21–24); and (2) the points at which breakdown of the interface occurs, and possible design alternatives that can remedy those breakdowns (cf. Douglas 1992, p. 3).

2.2 Introduction to the Lens AV system

An outgrowth of the system analyzed by the Stasko, Badre, and Lewis study cited above, the Lens system (Mukherjea & Stasko 1993) that we analyzed takes traditional AV systems (cf. Brown 1988; Stasko 1990; Naps & Hundhausen 1991) a step further by providing an environment in which users can define a visualization for an algorithm *interactively*, and *at runtime*. Written for XWindow platforms like the Sun SparcStation, Lens allows users to load the source code of their C programs (which must have been compiled in debug mode) into an interactive environment, in which they occupy a scrollable source code window. A snapshot of the Lens environment is shown in Figure 1.

Using the *interesting events paradigm* established by Brown (1988), users can interactively annotate their algorithms by (a) identifying points in their programs at which interesting events occur; (b) determining what graphical sequence would appropriately characterize those events; and (c) mapping the interesting events to graphics by manipulating the user interface. We see, then, that Lens requires users to engage in the same three central AV activities that we identified in our introduction.

In targeting the activities of "high-level debugging, program testing, and refinement, not low-level debugging" (Mukherjea & Stasko 1993, p. 3), the Lens system purports to provide direct-manipulation support for activity (c). In particular, it offers a menu of animation primitives, shown in Figure 2, for mapping interesting events to the underlying Tango AV language (Stasko 1990) on which Lens is based. The primitives on that menu form the foundation for defining an algorithm event—a process that consists of the following four steps:

1) Choose an animation primitive from the *Animation* menu.

2) Associate that primitive with the line of source code on which it should be generated (by using the mouse to click on the line of source code).

3) Use the graphical editor on the right-hand side of Figure 1 to associate a graphical representation with the primitive, if necessary. Note that certain animation primitives—for example, *flash* and *exchange*—work on animation primitives whose graphical representations have already been defined; hence, these animation primitives do not need to be defined within the graphical editor.

4) Enter information that helps Lens relate the animation primitive to the algorithm (e.g., is the height of the image dependent upon some variable in the algorithm?) into a series of modal dialog boxes. Samples of these dialog boxes, which illustrate the kind of information that Lens requests, can be found in Figure 3.

Users can choose to specify as many animation events as they wish using this process, and may even choose to specify multiple animation events on the same line. At any point in the process of annotating an algorithm, users may view the animation currently defined on the algorithm by choosing *Run* from the *Debug* menu; an animation window like the one in Figure 4 appears, and users may start the animation by clicking on the *Run Animation* button in that window. Thus, we see that Lens supports an iterative process of animation specification, in which users may interactively annotate the algorithm with animation primitives, and view, at any point, the animation to which those primitives give rise.

2.3 Study format

We videotaped three same-sex pairs of subjects (two consisting of men, one consisting of women) during 45 to 70 minute sessions with the Lens system. All subjects were graduate students in computer science, and all had prior experience with the C programming language and the X Window environment. One team of subjects had extensive experience with AV techniques, one team had no prior experience with AV, and one team's experience was limited to a knowledge of the sorting animations pioneered by the film *Sorting Out Sorting* (Baecker & Sherman 1981).

The common thread throughout the study was the exercise presented in Appendix A. With no prior introduction to Lens, all teams were asked to complete that exercise, which was designed to walk them through the process of defining a rudimentary animation on the popular bubble sort algorithm (Figure 5) in Lens. Note that prior to the videotaping sessions, all subjects reported that they had previously seen the bubble sort algorithm, and that they understood it.

After our first videotaping session, we decided that it might be interesting to obtain a feel for subjects' conceptualizations of a bubble sort animation prior to their defining it in Lens. Our idea was to compare subjects' natural conceptualizations of bubble sort—conceptualizations made *independently* of any specific computer-based animation framework—with the animations of bubble sort that they would define in Lens. To that end, we provided our next two pairs of subjects with several art supplies—a full spectrum of colored construction paper, a scissors, and different colored pens—and charged them with the following task: "If you were to explain the bubble sort algorithm to someone who had never seen it, how would you do it? Feel free to use any of these resources (construction paper, scissors, etc.) to assist you in your explanation." Both of these "construction paper" sessions were videotaped and analyzed.

In addition, in two of the videotaping sessions, we charged our subjects with the additional task of defining an animation on an algorithm without the benefit of a set of instructions. In both cases, they were asked to complete this task immediately after they completed the bubble sort exercise; in these situations, the bubble sort exercise served not only as a basis for our analysis of Lens, but also as an introduction that our subjects could use as a basis for their own independent experimentation with Lens. In one case, we provided the team with a working version of the popular insertion sort algorithm (Figure 6), which both said they understood. In the other case, we gave the team complete freedom in choosing an algorithm; they were then responsible for the entire process of writing, compiling, debugging, and animating that algorithm.

2.4 Study goals

We began the study with the underlying goal of becoming familiar with the use of CA as a technique for analyzing and evaluating interactive software. As Goodwin (1981) and Strauss (1987) point out, the technique is best learned through an apprenticeship, and this study was intended to provide the basis for such an apprenticeship. Second, as advocated by Douglas (1993), we wanted to use CA as a tool for identifying and remedying problems with the Lens user interface, especially with respect to the two serious forms of *Suchmanian* (Suchman 1987) breakdown: the *false alarm*, and the *garden path*; in doing so, we hoped to provide the Lens designers at Georgia Tech (Sougata Mukherjea and John Stasko) with a constructive review of their system, which could then be considered in the design of a new, improved version. Third, after conducting our first videotaping session, we decided that our study might shed light on two important questions: (1) How do people conceptualize algorithm animations in the first place?; and (2) To what extent do animations defined in Lens accord with such conceptualizations? Finally, we hoped that a detailed analysis both of the techniques people naturally use to explain an algorithm (independently of any AV framework), and of the detailed interaction that takes place between users and an AV system in specifying an animation, might give us insight into two higher-level problems: (1) developing more usable AV systems; and (2) devising more effective methods for analyzing and evaluating such systems.

3 Observations

We came away from our many hours in the viewing laboratory with an appreciation of the tremendous time and effort involved in thoroughly analyzing and diagnosing interface breakdown using CA-based techniques; more specifically, we found that every hour of videotape required roughly two hours of analysis. In this section, we report the fruits of that labor, in the form of several interesting observations we were able to glean from a detailed analysis of the videotapes. Using as a basis a transcription of an episode that one team had while trying to complete the bubble sort exercise, Section 3.1 identifies and analyzes several characteristics of the Lens user interface that proved problematic for our subjects. Section 3.2 comments on our subjects' construction paper sessions, in which they concretized their own conceptualizations of a bubble sort animation with construction paper, pen, gestures, and dialog. Finally, in Section 3.3, we briefly discuss our subjects' free-form sessions, in which they used Lens to animate an algorithm without the benefit of instructions, and, in one case, without the benefit of a previously written and compiled algorithm.

3.1 The bubblesort exercise

Although all teams of subjects were eventually able to define and view the bubble sort animation as specified in the exercise, they encountered several frustrating problems along the way. Appendix B presents a transcript of an episode that well illustrates several of those problems. In that episode, the subjects—referred to as "J" and "B" in the transcript—are trying to define and associate an *image* (see Figure 2) with the array a to be sorted (see Figure 5).[1] Unfortunately, as should be clear from the transcript, our subjects are not having an easy time of it. Using the transcript as a guide, the following three subsections probe three of the most serious problems our users had with the Lens interface. We conclude the section by briefly discussing our observations of our subjects as they viewed the bubble sort animation that resulted from their manipulation of the Lens interface.

[1] In Lens, the *image* primitive is used to define the graphical appearance of some object—usually a scalar variable or an array—in the program.

3.1.1 Image creation gives rise to *Suchmanian garden path* in two sessions

In identifying one of the most insidious forms of communication breakdown, Suchman (1987) defines a *garden path* as a situation in which "a misconception on the user's part produces an error in her action with respect to the prescribed procedure, the presence of which is masked" (p. 163). Although, in such situations, the user's actions belie her intent, they are seen as nonproblematic by the system because they, in fact, constitute valid actions prescribed by the system. Thus, the *Suchmanian garden path* begins at precisely the point where the user's true intent, and the intent that the system ascribes to the user's actions, diverges. On the surface, communication appears non-problematic; however, when we compare what the user believes she is doing to what the system believes is going on, we find a potentially sharp discrepancy.

In analyzing our videotapes, we found that one particular task within the bubble sort exercise —defining an *image*—gave rise to garden paths in two of our three videotaping sessions. To illustrate, let us consider the sequence that begins at line 1. By line 20, J and B are well on their way to defining the image to associate with the variable *a*; however, J jumps the gun, as it were, and drags out the image before reading all of the instructions in (4d) (Appendix A). The image he drags out is a black square, which, from the system's perspective, is non-problematic; indeed, on line 28, we see (in the "Design Rationale" column) that the system interprets this action as indicative of the user's intent to define an image for a *single element*. In contrast, J and B are trying to define an *array*. Nonetheless, as tenacious users, they arrive at an explanation for the black bounding box to which both of them agree (line 31): "Yeah, cuz that's the fill color we chose." We see, then, that from the users' perspective, the system understands their intent to define an array, but, from the system's perspective, the users are trying to define a single element. The Suchmanian garden path thus begins, and it is not until B's exclamation on lines 57 to 59 ("but I don't see that, so we must have done somethin' wrong") that the users realize that the system has not been interpreting their actions correctly.

3.1.2 Lens unable to anticipate local repair within the *create image* task sequence

Although the task of creating an image did not give rise to a garden path in all of our sessions, that task did prove problematic for all teams. In completing the five steps required to create an image, our users would invariably make some kind of mistake—the result of either jumping the gun, as J did above, or of misunderstanding the rigid sequential ordering that Lens imposes on those steps. As we shall illustrate below, both Lens's inflexibility with regard to the five steps entailed in creating an image, and Len's inability to anticipate and provide for the local repair of any of those five steps, led to confusion and frustration on our subjects' part.

The first problem we shall consider manifests itself in the sequence beginning at line 66. Here, having realized that they "must have done somethin' wrong" (lines 58–60), and having diagnosed the problem as a failure to click on the "Array" button before dragging out the bounding box (lines 70–71), J and B initiate a repair sequence. In particular, they (a) click on the "cancel" button, (b) click on the "Array" button to fix their mistake, and (c) begin a second attempt at the *create image* task by choosing "create image" from the "Animation" menu. Unfortunately, as B correctly identifies on lines 108–109, their repair "isn't doing the right thing"; indeed, they have fallen into precisely the same trap—namely, the unexpected "Image Information" dialog box—as they previously encountered.

The failure of J and B's repair clearly stems from Lens's insistence that the "Array" radio button be clicked *after* the users have completed the second step of the *create image* process (i.e., clicking on the line of source code). Lens's inflexibility surfaces on lines 77–79; having received a request to create an image, Lens meticulously toggles the "Array" button, which the users had just set to "Array" as part of their repair sequence, back to the default value "Single". Apparently, Lens resets this value in an at-

tempt to assist the user, who, from Lens's perspective, will most likely want to define a single element. In doing so, however, Lens not only imposes an overly rigid ordering of the steps required to create an image, but also undermines B and J's attempt to repair their previous mistake.

To illustrate the second major problem our subjects had with the *create image* task, let us consider the sequence beginning on line 108. Realizing that their attempt at repairing their *create image* sequence (lines 68–100) has failed (cf. "this isn't doing the right thing" on lines 108–110), J & B tenaciously initiate a second repair sequence by clicking on the "Cancel" button. In response to their clicking on "Cancel," the system removes the "A" that it had placed next to the line of source code on which J and B had previously clicked (line 91). Furthermore, as indicated on line 110 in the "Design Rationale" column, the system makes the spurious assumption that the users' cancellation applies to the *entire* create image task, and not to the *local* task of dragging out a bounding box for the array. However, as J's actions on lines 122–125 indicate, J and B see their cancellation as *local* to the task of dragging out a bounding box; accordingly, J tries to remedy the mistake by (a) resetting the type of bounding box to "Array" (line 114), and (b) re-dragging out the bounding box (lines 122–125)—this time, for an *array*, as opposed to a *single element*. Unfortunately, having returned to a state in which an image is no longer being defined, Lens is no longer in a position to interpret the local repair appropriately. Lens's lack of a response to J's attempted repair gives rise to yet another, more serious attempt to initiate local repair (lines 154–208). B's sarcastic comment that it "makes a nice theory" (line 209), together with his concession "I give up" (line 212), clearly indicate that our subjects are thoroughly confused about frustrated with the *create imag*e task..

3.1.3 User's confused by the system instruction "Click on the line before which the animation occurs"

In all three videotaping sessions, our subjects were noticeably vexed by the message "Click on the line of source code before which the animation occurs," which Lens presents in the lower right-hand corner of the screen immediately after users dismiss the "Create an image" dialog box. Upon first reading that message, our subjects intently pondered the meaning of "before which," which ostensibly contradicts the instruction in (4c) that they were following. Happily, in all three cases, they believed the instructions instead of the Lens message, and eventually clicked on the correct line of source code. Indeed, if they had heeded the Lens message, the animations they defined would not have worked.

3.1.4 Viewing the bubble sort animation

We are happy to report that, in all three cases, our subjects seemed to comprehend the bubble sort animation they defined; however, in two cases, the subjects did not immediately recognize that the animation was mapping array element magnitude to bar height (see Figure 4). Interestingly, the two situations in which subjects expressed surprise at the Lens mapping were the same two situations in which we had the subjects explain the algorithm with construction paper and pen prior to their defining the animation in Lens. In these cases, subjects initially uttered remarks like "Huh?" and "What?," until it became clear that the bars were gradually falling into order, at which point they said something to the effect of "Oh, I see; height indicates value."

3.2 The "construction paper" sessions

As it turned out, in explaining bubble sort to a hypothetical novice, all subjects made extensive and intriguing use of the art supplies with which we supplied them. In fact, both pairs of subjects used the materials to create a homemade animation of bubble sort operating on a sample data set; they aug-

mented their construction paper animations extensively with their own gestures and comments. Our most significant observation was that in neither case did the homemade animations constructed by our subjects resemble, in any strong sense, the animation they defined in Lens. One group used physical numbers (e.g. 5, 4, 3) to indicate the magnitude of each array element, and a color scheme to indicate whether an element was (a) in its rightful place; (b) the target of a comparison in which it *was* swapped; and (c) the target of a comparison in which it *was not* swapped. Each pass of the array was then depicted as a (separate) horizontal row of (number-labeled) elements whose colors indicated what had happened during that pass. At the end of their demonstration, they had accumulated a history "array" of the sort, each of whose (horizontal) rows showed one pass of the bubble sort.

In contrast, the other group used color to indicate magnitude; in particular, they constructed a legend that defined a canonical (spectral) order for a five-element array, and demonstrated, by illustrating the array after each successive pass of the loop, how the bubble sort gradually placed an unsorted array into that canonical order . Like the first group, this group had built, by the end of the sort, a history "array" in which each bubble sort pass occupied a row. Since they gave each array element its own color, the movement of array elements throughout the sorting process could be clearly seen.

3.3 The freelance animation sessions

Although a discussion of our detailed observations of the freelance sessions is beyond the scope of this paper,[2] we can point out two general observations we made. First, neither team fully succeeded in defining the kind of animation they wanted. For example, the team that worked on an animation for insertion sort had great difficulties in mapping the "insertion" event, which occurs after each pass of insertion sort's outer loop (`a[j+1] = x`; see Figure 6), to Lens animation primitives. In the end, they conceded that they needed another animation primitive—namely, *copy and move*—that Lens simply did not provide. And second, the group that engaged in the task of writing, compiling, and animating their own algorithm (inadvertently) introduced several run-time bugs into the code on which they were to define an animation. Interestingly, despite its authors claim that Lens is most suitable for high-level debugging, our subjects chose not to use Lens as a tool for debugging their program; indeed, it was only after they had debugged their thirty line program using trace statements that they loaded it into Lens and defined an animation.

4 Conclusions

Using our analysiss of the sessions, and, in particular the observations presented in the previous section, as a basis, this section draws some conclusions from our study. Section 4.1 reports our general impressions of the Lens systems, and diagnoses the detailed problems we presented in Section 3.1, while Section 4.2 draws from the material in Section 3.2 and 3.3 to consider the higher-level problem of developing and evaluating interactive AV systems.

4.1 Lens interface: shortcomings and improvements

Our discussion in Section 3.1 shed light on what we view as the most serious low-level problem with the Lens interface: namely, its inability to anticipate and provide for the local repair of the *create image* task. As we saw in Sections 3.1.1 and 3.1.2, Lens's assumption that the cancellation of a local step in that task can lead, at best, to much confusion and frustration on the user's part, and, at worst, to the most insidi-

[2] Indeed, it constitutes another paper in its own right!

ous form of *Suchmanian* breakdown: a garden path. In this section, we first make some general comments on the Lens interface. We conclude by presenting what we believe to be the three interface shortcomings that underlie its most serious problem , along with possible strategies for remedying each of those shortcomings.

4.1.1 General comments

We have three general comments regarding the usability of Lens. First, as we clearly noticed in our two freelance sessions, Lens, in its present form, is amenable to only a narrow range of algorithms—a range of algorithms which seems to have the common thread of arrays.[3] Unfortunately, as was evident in both freelance sessions, when an algorithm contains any kind of interaction between arrays and scalar elements (as was the case in both freelance sessions), users seem to have an extremely difficult time specifying that interaction with Lens animation primitives. For example, after three attempts, the team that tried to animate insertion sort failed to define an animation event that adequately inserted *x* (see Figure 5) into the array *a*.

Second, we take issue with the claim that Lens, in its present form, constitutes direct-manipulation software. Instead, we see it as more of a batch-processing environment, in which users must specify a relatively large amount of information—much of it by typing into dialog boxes—before obtaining any idea of what their animations might look like. As indicated by both the confusion our subjects expressed as they specified detailed information into dialog boxes, and by the surprised looks and comments of our users as they first watched the bubble sort animation they had defined, Lens does not seem to support a specification technique that gives users very direct access to the animation they are defining.

And third, our analysis of the session in which users went through the entire process of conceptualizing, implementing, debugging, and animating an algorithm suggests that the animation specification technique defined by Lens is sufficiently complex to deter users from using Lens, in lieu of text-based techniques, for the purpose of debugging. In fact, at one point in that session, a subject asked the question explicitly: "So, do we use Lens to actually debug the program here?" The conclusion at which the subjects immediately arrived was that using Lens for that purpose would be too difficult, because it would introduce an added variable into the debugging process: the abstraction-to-graphics mapping. In particular, they believed that, in viewing an animation of the algorithm, they would have a difficult time determining whether their code, or the mapping they defined on that code, was to blame for any bugs they detected.

4.1.2 Three perspectives on the *create image* problem

First, our analysis suggests that, in general, the tasks required to define an animation primitive— especially *create image* and *create location*—involve too many steps; moreover, both the ordering that Lens imposes on these steps, and its failure to provide for the local repair of any one of them, need to be reconsidered. If the Lens designers decide to stick with the present sequential task mechanism for specifying animation primitives, they ought to incorporate into each step a method for redoing the step without canceling the entire task. In particular, buttons labeled "Previous Step" and "Next Step"— perhaps augmented with forward (→) and backward (←) arrows—could do wonders in elucidating the ordering on task steps imposed by Lens. Using such navigational buttons, users would be given the opportunity to make the kinds of local repairs that, as our study clearly indicates, users will invariably need to make.

---

[3] Indeed, during the course of one of the freelance sessions, one subject was compelled to say something to the effect of, "Oh, I see, it has 'magic code' for arrays!"

Second, our study indicates that the kind of information requested by several Lens dialog boxes— especially the first two presented in Figure 3—is too complicated for users who are unfamiliar with the underlying Tango animation language on which Lens in based. In filling in such dialog boxes, our subjects often looked puzzled, if not clueless; in all cases, they certainly did not seem to understand what they were filling in, and had to rely intimately on our detailed instructions.[4] Although we do not doubt that the type of information requested by the compound dialog boxes is essential to mapping an algorithm event to graphics, we wonder whether the method by which Lens has chosen to request the information might be revised, so that users with a clear conceptualization of an algorithm animation would not have to concern themselves with such details.

As we see it, an interactive AV system ought to strive for an animation-to-graphics mapping specification method that requests the mapping in the same (or as similar as possible) terms as the terms in which a user conceptualizes it. While direct-manipulation techniques such as clicking on lines of source code, and dragging out bounding boxes may be a start, we believe that a more usable system will take direct-manipulation further. For example, might there be a way for users to define an exchange by using the mouse to drag elements to their new locations?

Third, our observations suggest that, in general, dynamic feedback from Lens needs to be more frequent, sensitive, and understandable. For example, consider the extended episodes in Appendix A (cf. lines 112–212), during which our subjects persistently tried to repair the *create image* task. Clearly, their misunderstanding of Lens's state during such episodes was due, in large part, to the fact that they did not notice that their cancellation (cf. line 112) caused the "A" to the left of a source code line to disappear; quite simply, that small bit of feedback was too subtle and ambiguous to be of use. Further, the general paucity of feedback during these episodes—clearly noticeable, for example, if one scans the "Available to the User" column between lines 112 and 212—gave our subjects absolutely no support in their attempt to figure out why their actions were not producing the expected results. Similarly, the messages that were provided were often too confusing (e.g. "Click on the line of source code before which the animation occurs."), or too terse ("Draw an image."), to be of much help to our subjects.

In light of our observations pertaining to feedback, we recommend that (a) more detectable feedback be employed in the case of user cancellation; (b) more sensitive feedback—perhaps in the form of informational messages—be employed to assist users who are trying to initiate the repair of a task ; and (c) alternative forms for some of the present feedback messages be considered (e.g., consider "Click on the line of source code *on* which the animation occurs," which is more accurate than "Click on the line of source code *before* which the animation occurs").

4.2 Towards more usable interactive AV systems

We believe that the study we have described above has succeeded, for the most part, in meeting the first two goals we set in Section 3.1. Indeed, in accordance with our first goal, we have obtained an excellent feel both for the logistics of carrying out a usability study using CA, and for the art of critically interpreting and analyzing videotaped sessions of human/computer interaction. Furthermore, as stipulated by our second goal, our study has assisted us not only in successfully identifying several problematic aspects of the Lens user interface, but also in sketching out possible solutions for alleviating those problems in the future. Because the second two goals of our study address interactive AV in a broader picture, it is much more difficult to determine the extent to which we have succeeded in meeting those goals. We conclude this paper by considering two higher-level issues that our final two goals purported to address.

---

[4] To date, we ourselves do not precisely understand what information the 3x3 grid of anchor points found in the "Exchange Information" dialog box (Figure 3c) supplies!

First, (a) have we learned anything about how people conceptualize algorithms?, and (b) do Lens animations accord with such conceptualizations? Clearly, both of these questions rely intimately on the extent to which our subjects' homemade animations were indicative of how they actually conceptualized the algorithm. However, even if our study "loaded the dice" by providing users with a *specific* kind of material (construction paper) that lent itself well to a *specific* kind of conceptualization (graphical)—and hence inhibited them from expressing the algorithm as they might have naturally—we maintain that the users had far fewer problems in expressing that conceptualization with construction paper than they did with Lens. Moreover, in the process of using the less problematic resource (i.e., construction paper, scissors, and pens), the users came up with vastly different animations in both cases; indeed, neither group's homemade animation performed the fundamental array-to-graphics mapping as Lens did (using bar height). We are forced to conclude, then, that although our study found no significant commonalties that might shed light on (a), it did shed light on (b) by indicating that Lens did not, and most likely could not have, supported the kinds of animations that our subjects created in their construction paper sessions.

Finally, have we gained any insight into how one might go about developing more usable interactive AV systems, and evaluating those systems? We believe that our study's results have clearly indicated the potential for CA as an evaluation method within this domain. Furthermore, our perception of the interactive AV problem as encompassing three fundamental ingredients—(a) conceptualization, (b) conceptualization-to-AV language mapping, and (c) mapping specification with an interface—has been reinforced by our study, which has helped us to obtain a better appreciation of the complexity of each of those ingredients. Thus, we believe that future research on interactive AV should proceed from a recognition of these ingredients and their interrelations. More specifically, research ought to remind mindful not only of the ease with which (c) facilitates the mapping in (b), but also the extent to which the AV language in (b) can support (a). As we see it, the development of usable AV system must be rooted in the kinds of studies that can further refine these interrelations. As we have illustrated here, the CA technique we have used holds promise in that endeavor.

References

Baecker, R.M, & Sherman, R.M. (1981). *Sorting out sorting.* 16mm color sound film shown at SIGGRAPH '81 (Dallas, TX).

Douglas, S.A. (1993). Using Conversational Analysis to Discover Breakdown in the Design of User Interfaces. *Proc. CSCW Workshop on Ethnographic Methods and Design.*

Goodwin, C. (1981). *Conversational organization: Interaction between speakers and hearers.* New York: Academic Press.

Hundhausen, C.D. (1993). Subverting the comparative research paradigm: The potential for ethnomethodology in evaluating the effects of algorithm visualization on learning. Unpublished seminar paper, Department of Computer and Information Science, University of Oregon.

Miyake, N. (1986). Constructive interaction and the iterative process of understanding. *Cognitive Science 10,* 151–177.

Mukherjea, S., & Stasko, J.T. (1993). Applying algorithm animation techniques for program tracing, debugging, and understanding. *Proc. 15th IEEE International Conference on Software Engineering* (Baltimore, MD), 456–465.

Naps, T.L, & Hundhausen, C.D. (1991). The evolution of an algorithm visualization system. *Proc. 24th Annual Small college Computing Symposium* (Morris, MN), 259-263.

Roschelle, J. (1990). Designing for conversations. In *Proceedings of the AAAI Symposium on Knowledge-Based Environments for Learning and Teaching* (Stanford, CA).

Stasko, J., Badre, A., & Lewis, C. (1993). Do algorithm animations assist learning? An empirical study and analysis. *Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems* (Amsterdam, The Netherlands), 61–66.

Stasko, J.T. (1990). Tango: a framework and system for algorithm animation. *IEEE Computer* (September), 27-39.

Strauss, A. (1987). *Qualitative analysis for social scientists.* Cambridge: Cambridge University Press.

Suchman, L. (1987). *Plans and situated actions: The problem of human–machine communication.* Cambridge, MA: Cambridge University Press.

Van Dam, A. (1984). The electronic classroom: Workstations for teaching. *International Journal of Man-Machine Studies 21* (4), 353–363.

Figures

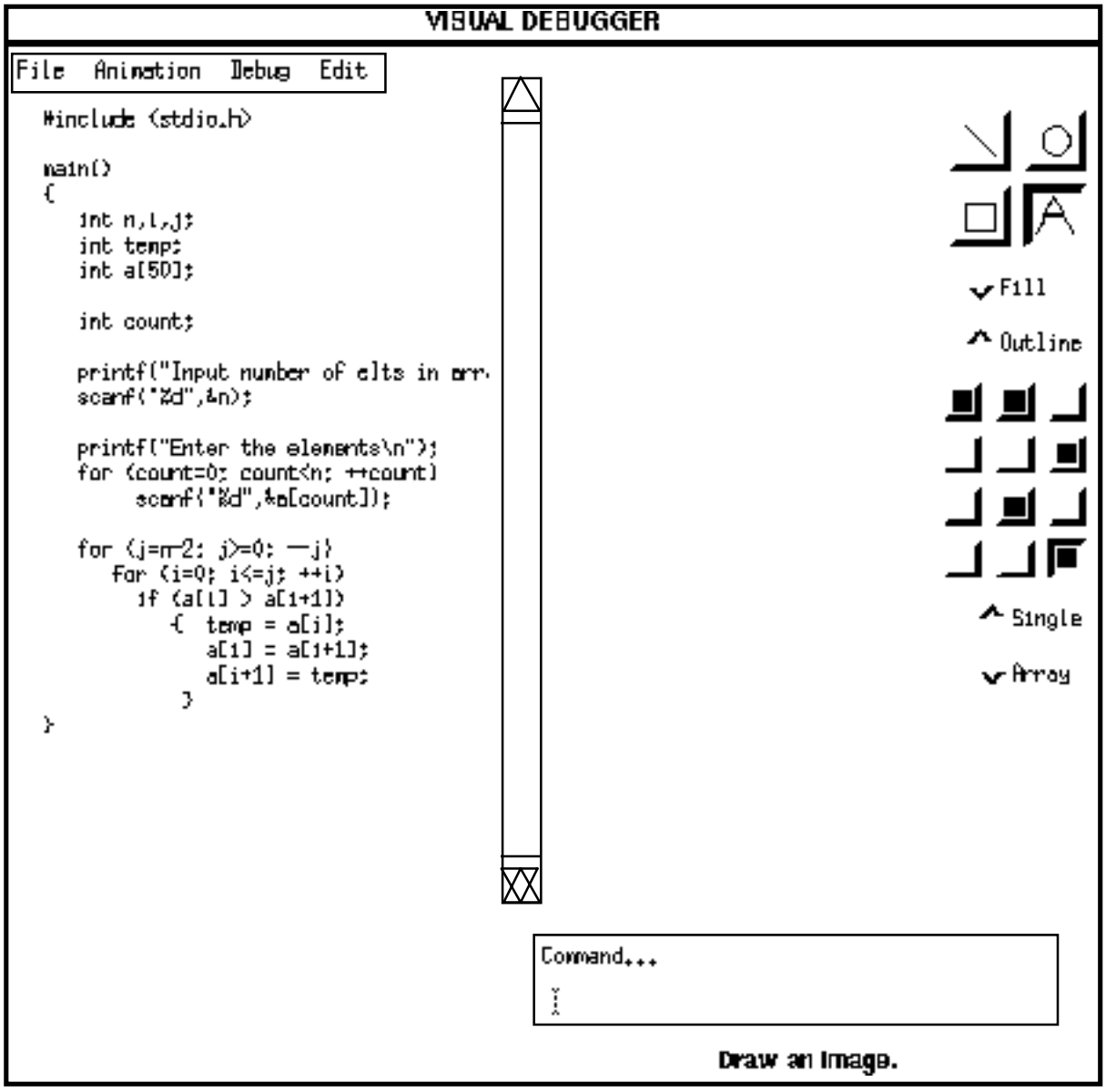

Figure 1. A snapshot of the Lens user environment

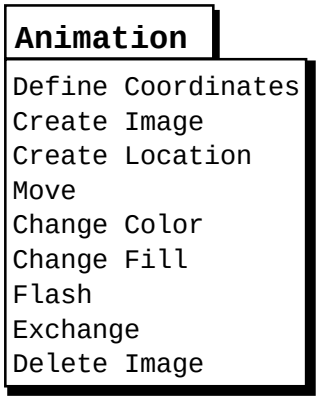

Figure 2. Lens's menu of animation primitives

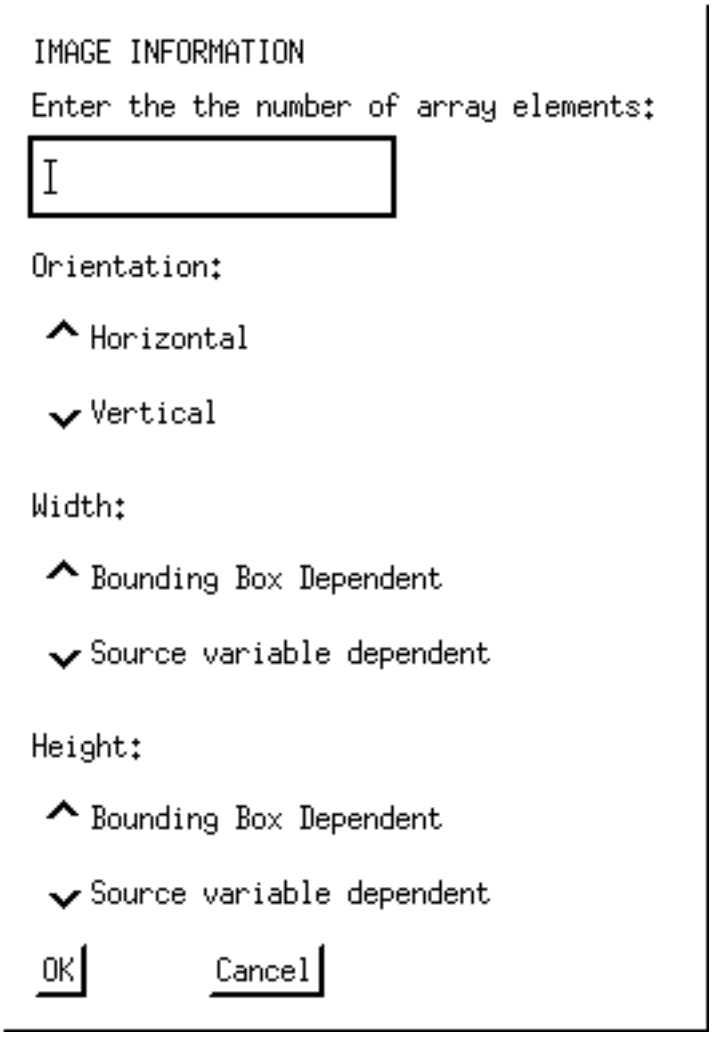

Figure 3a. First array image information dialog box

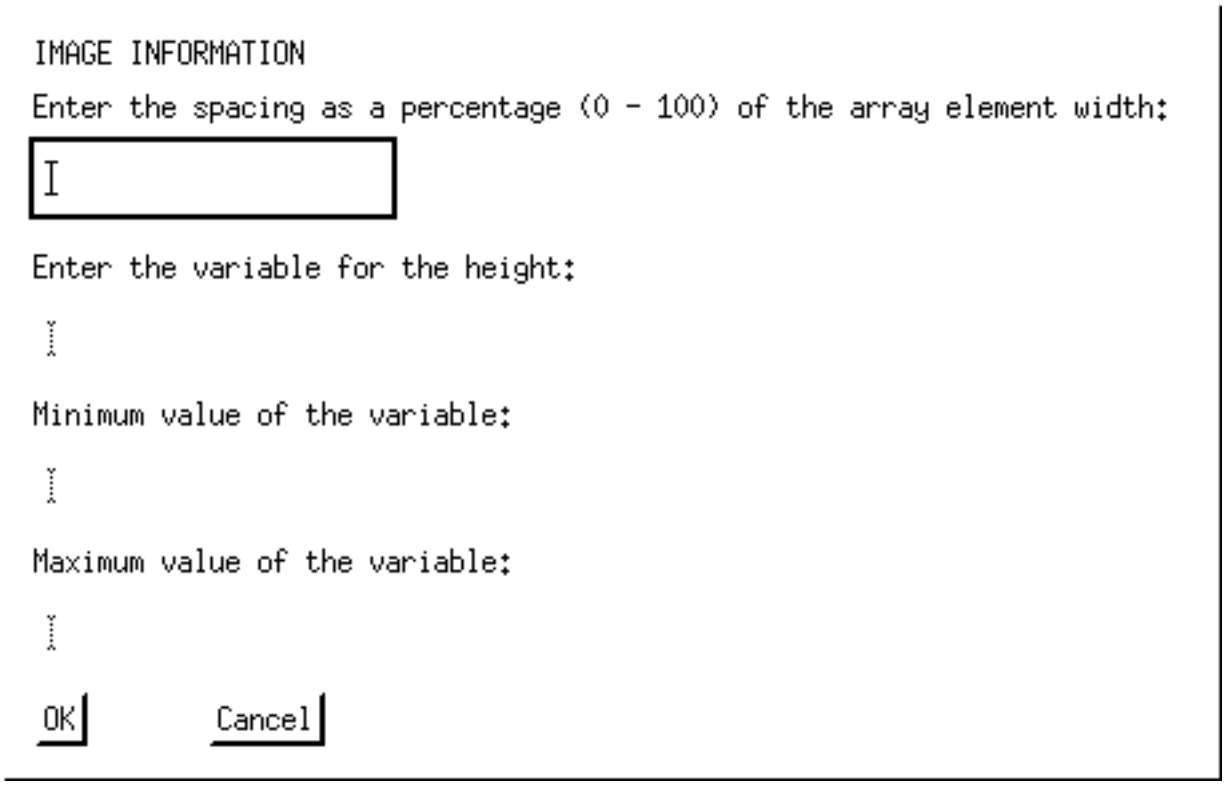

Figure 3b. Second array image information dialog box

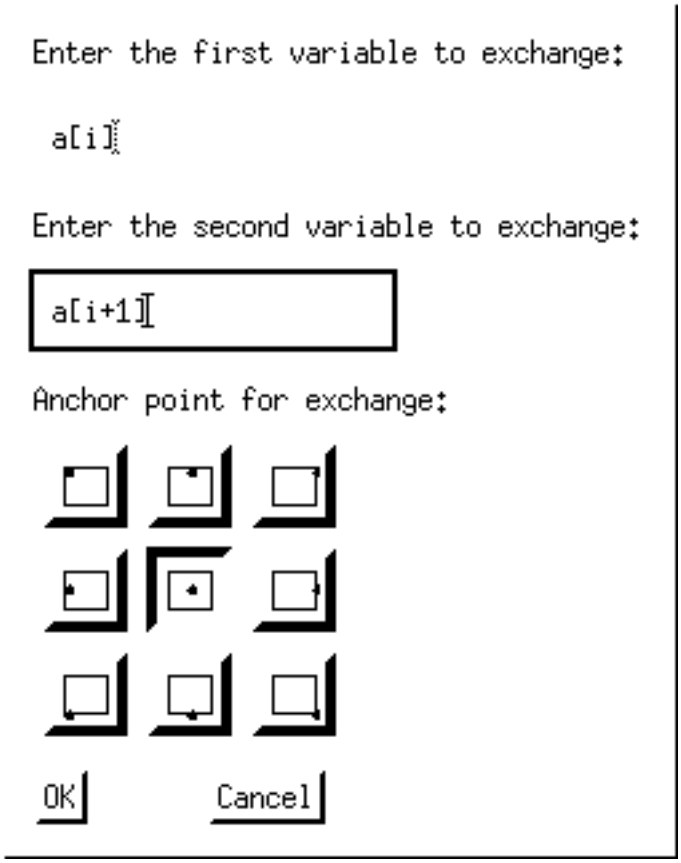

Figure 3c.. Exchange information dialog box

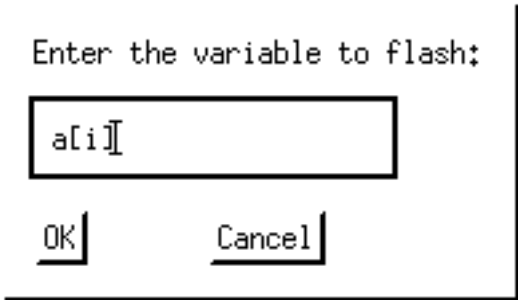

Figure3d. Flash information dialog box

Figure 3. Sample dialog boxes by which Lens requests graphical information on animation primitives

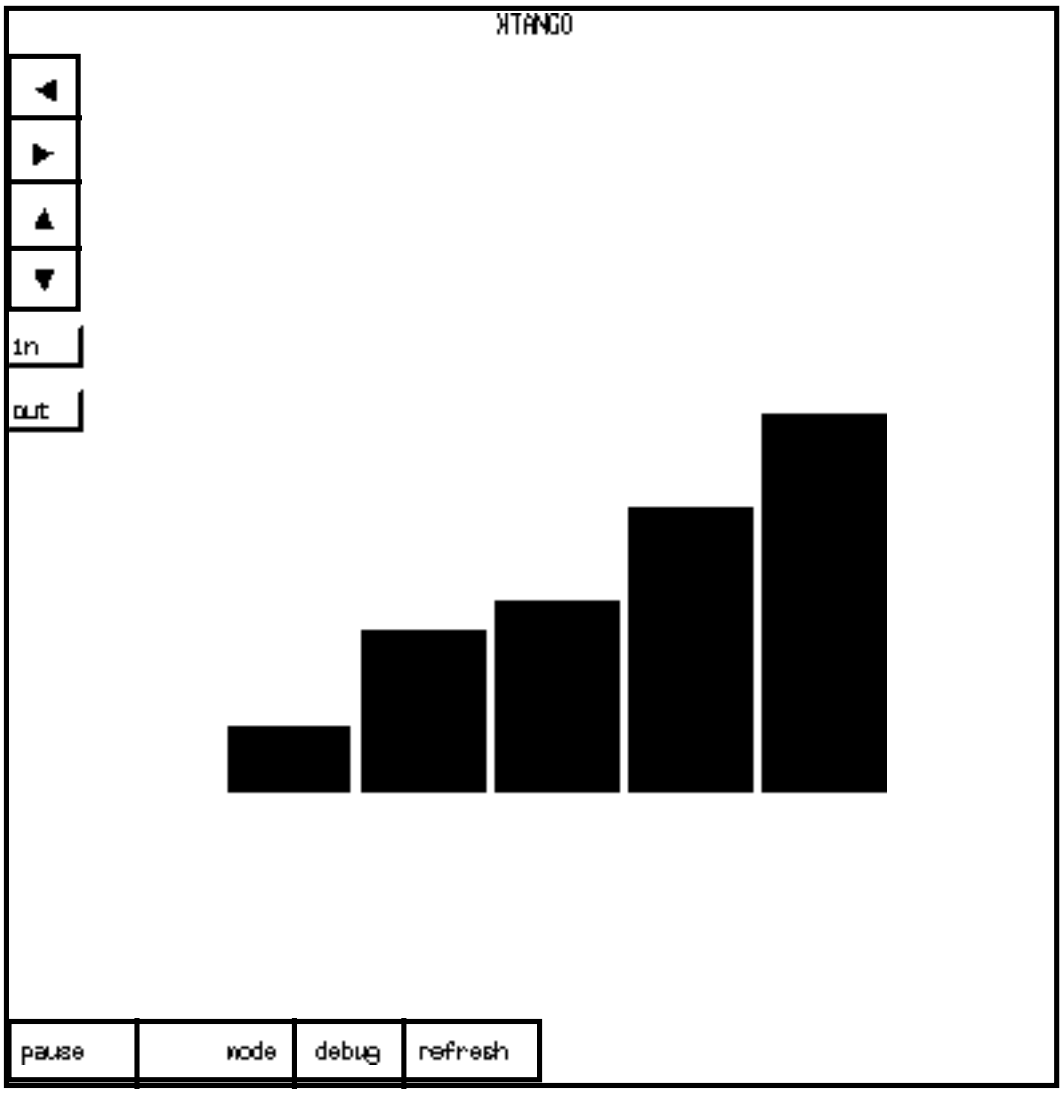

Figure 4. The Lens animation window. This particular window shows the final state of a dynamic bubble sort animation like the one subjects defined in the study

```
#include <stdio.h>

main()
{
   int n,i,j;
   int temp;
   int a[50];

   int count;

   printf("Input number of elts in array\n");
   scanf("%d",&n);

   printf("Enter the elements\n");
   for (count=0; count<n; ++count)
        scanf("%d",&a[count]);

   for (j=n-2; j>=0; --j)
      for (i=0; i<=j; ++i)
        if (a[i] > a[i+1])
           {  temp = a[i];
              a[i] = a[i+1];
              a[i+1] = temp;
            }
}
```

Figure 5. The C *bubble sort* algorithm that subjects animated

```
#include <stdio.h>

main()
{
   int x, xindex, j, n;
   int a[50];

   int count;

   printf("Input number of elts in array\n");
   scanf("%d",&n);

   printf("Enter the elements\n");
   for (count=0; count<n; ++count)
        scanf("%d",&a[count]);

   for (xindex = 1; xindex < n; ++xindex) {
     x = a[xindex];
     j = xindex - 1;
     while ( (j > -1) && (a[j] > x) ) {
       a[j+1] = a[j];
       --j;
     }
     a[j+1] = x;
   }

 }
```

Figure 6. The C *insertion sort* algorithm on which one pair of subjects attempted to define an animation

Appendix A: The Lens bubble sort exercise

```
*************************************************************************
Welcome to the Lens Usability Test.  Lens is a visual debugging system that
allows one to define mappings between an algorithm's source code and two
dimensional graphics.  In order to give us a better idea of how well the user
interface has been designed, and of what role the algorithm-to-graphics
mappings that can be defined in Lens might play in one's understanding of an
algorithm, you will be asked to complete the following short laboratory
exercise.
*************************************************************************
```

1) From the "[/home/grads/chundhau/av/lens]%" prompt in the open window, type "lens bsort".

2) A wire-frame outline of the Lens window will appear; drag the mouse to position the window where you'd like it, and click the first mouse button to make the Lens window appear.

   OK, now you're ready to begin the exercise, which will entail defining and observing a simple animation for a popular sorting algorithm called bubblesort.

3) From the "File" menu, choose "Open Source". When the File chooser dialog box appears, click on the file "bsort.c" and then click on "OK".

   The C source code for the bubblesort algorithm should now appear in the left hand portion of the Lens window. You're first order of business is to design an animation on the bubblesort; viewing this animation should assist in your understanding of the underlying logic of the bubblesort algorithm. In brief, the steps to defining an animation are: (1) identify the data in the bubblesort with which you'd like to associate graphical objects, associate images with those data, and define the graphical appearance and characteristics of the images; and (2) identify points in the source code at which events crucial to the algorithm's execution occur, and map those "interesting events" to graphical events in the animation. Let's get to it!

4) First, identify the data of interest in the algorithm. A brief scan of the bubblesort code reveals that the bubblesort is sorting the integer elements of the array 'a'; hence, we'll want to associate an image with that object:

   a) Choose "Create Image" from the "Animation" menu
   b) Type 'a' into the "" dialog box that appears
   c) Type 'a' into the "" dialog box that appears
   d) We want this image to be created after the array 'a' has been initialized in the "for" loop; use the pencil cursor to click on the blank line immediately following

      ```
      for (count=0; count<n; ++count)
        scanf("%d",&a[count]);
      ```

      Notice that an 'A' appears on the left-hand margin of that line.

   e) Now you're prepared to create the image to associate with the array 'a'. On the right-hand side of the Lens Window, you'll notice a palette of graphical primitives; you'll use these primitives to define the graphical appearance of the array. First, at the top of the palette, click on the button labeled by a black square; this sets the current drawing tool to a rubber-band bounding box. Next, click on the button labeled by "Fill"; this says that the array elements will be filled as opposed to hollow. Third, choose an appropriate fill color; click on the button whose color pleases you. Fourth, click on the button labeled "Array"; this tells Lens that the graphical object you're defining is being associated with an array. Finally, you are ready to draw the object. In the white space on the right-hand side of the Lens window, use the first mouse button to drag out a bounding box for the array. Don't worry about its exact size or location, but try to make

it reasonably big because the entire array 'a' will need to fit in it.

f) When the "Image Bulletin Board_popup" appears, enter 'n' as the number of array elements, and choose "Source Variable Dependent" under " Height:". You can leave the rest of the settings alone. Click "OK" to approve the settings.

g) Another "Image Bulletin Board_popup" will appear. Enter "10" for the spacing factor, "a" for the height variable, "0" for the minimum value of the variable, and "20" for the maximum value of the variable. Click "OK" to approve your settings. You have now successfully defined the image associated with the array 'a'.

5) We now move on to the second stage of defining the animation: namely, the identification of "interesting events" in the bubblesort, and the mapping of those events to graphical animation events.

a) The first event of interest in the bubblesort occurs when two array elements are exchanged; notice that this occurs in the following segment of code:

```
{  temp = a[i];
   a[i] = a[i+1];
   a[i] = a[i+1];
   a[i+1] = temp;
 }
```

To associate an exchange event with that code, choose "Exchange" from the "Animation" menu. A "Variable Bulletin Board 2_popup" will appear; enter 'a[i]' as the first variable to exchange, and 'a[i+1]' as the second variable to exchange. Click on the button whose point resides in the lower left-hand corner of the square as the anchor point for the exchange. Finally, click on 'OK' to approve your settings.

Once again, you'll need to click on a line of source code to associate with this event; using the pencil cursor, click on any line in the segment of code identified above. As before, a red 'A' will appear on the left-hand margin of the line you clicked on. The Exchange event has now been defined.

b) The second event of interest--and the last one that you'll define for this exercise--occurs when adjacent elements in the array 'a' are compared. Clearly, this occurs in the following segment of code:

```
if (a[i] > a[i+1])
```

To emphasize that the array elements a[i] and a[i+1] are being compared, we want to define a "Flash" event on the comparison. To do that, choose "Flash" from the "Animation" menu. A "Variable Bulletin Board 1" dialog box will appear; enter 'a[i]' as the variable to flash, and, with the pencil cursor, click on the if statement in the source code that is identified above.

Since, in a comparison, you'll also want the graphical object associated with a[i+1] to flash, you need to follow the same procedure just outlined for the variable 'a[i+1]'. Choose "Flash" from the "Animation" menu, 'a[i+1]' as the variable to flash, and click on the same if statement as before with the pencil cursor to identify where in the source code the flash occurs.

6) Alas, you are now done defining the animation for this exercise. The final step of the exercise is to reap the visual benefits of the animation you just defined by actually observing it. To do that, follow these steps:

a) Choose "Run" from the "Debug" menu. You'll notice that the wire frame for a new window appears. Using the mouse, position that wire frame where you'd like it, and click on the first mouse button to make the window appear. This window, labeled "XTango", is where the animation will actually take place.

b) To begin the animation, click on the "run animation" button in the XTango window. At this point, you'll need to find the original Pragmatix window from which you started lens with the command "lens bsort" in step 1. Notice that the prompt "Input number of elts in array" appears in this window. Follow the command by entering a number like '5'; this is the number of elements that the bubblesort will sort. Now actually enter the integer elements in the array; choose any values you like between 0 and 20, remembering that the sort will not be very interesting if you enter numbers that are already in order. After entering each number, hit return.

c) Once you have entered all array elements, the animation should commence. Notice that the "Flash" and "Exchange" events you defined earlier manifest themselves in physical objects' actually being flashed and exchanged as the animation progresses. When the animation completes, you are done with the exercise.

Thank you for your participation!

Appendix B: *Transcription of a session in which subjects are trying to define an image, and to associate the image with array "a". Note that the format and notation that we have used first appeared in (Suchman 1987).*

Sorry, Appendix B is not available in this PostScript file. Look for supplemental PostScript file.