Communicative Dimensions of End-User Environments

Christopher D. Hundhausen
Laboratory for Interactive Learning Technologies
Department of Information and Computer Sciences
University of Hawai'i
Honolulu, HI 96822 USA
hundhaus@hawaii.edu

Sarah A. Douglas
Human-Computer Interaction Laboratory
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403–1202 USA
douglas@cs.uoregon.edu

Abstract

In exploring how to make programming easier for non-programmers, research into end-user environments has traditionally been concerned with designing better human-computer interaction. That traditional focus has left open the question of how end-user environments might support human-human interaction. Especially in situations in which end-user environments are enlisted to facilitate learning, we hypothesize that a key benefit may be their ability to mediate conversations about a domain of interest. In what ways might end-user environments support human communication, and what design features make them well-suited to do so? Drawing on ethnographic studies of an undergraduate algorithms course in which students constructed and presented algorithm visualizations, we develop a provisional framework of six communicative dimensions of end-user environments: programming salience, typeset fidelity, story content, modifiability, controllability, *and* referencability. *To illustrate the design implications of these dimensions, we juxtapose conventional algorithm visualization technology with a prototype end-user environment specifically designed to facilitate communication about algorithms. By characterizing those aspects of end-user environments that impact social interaction, our framework provides an important extension to Green and Petre's* [1] *cognitive dimensions.*

1 Introduction

End-user environments couple an end-user language with a graphical user interface for programming in that language. The traditional aim of such environments has been to empower non-programmers to write their own programs. For example, computer spreadsheets have enabled non-programmers to write a variety of sophisticated numerical and accounting applications (see, e.g., [2]). Likewise, AgentSheets [3] empowers teachers and students to write their own scientific simulations.

In line with its goal to make programming easier for non-programmers, past research into end-user environments has focused squarely on the problem of *human-computer interaction*. This focus is well reflected by Green and Petre's [1] *cognitive dimensions* framework, which characterizes the effectiveness of end-user environments largely in terms of their influence on individual performance—for example, error rates, program comprehension, and programming time and effort.

Notice that this traditional focus neglects the possibility that end-user environments might facilitate *human-human* communication. Yet, especially in the literature on educational technology, researchers have noted the ability of end-user environments to mediate conversations about a particular domain of interest. For example, through his development of an end-user environment for exploring Newtonian physics, Roschelle [4] came to see the utility of such an environment not in terms of its ability to transfer physics knowledge to learners, but instead in terms of its ability to act as "a resource for managing the uncertainty of meaning in conversations, particularly with respect to the construction of shared knowledge" (p. 1). Likewise, in developing Belvedere, a system that enables learners to represent data, hypotheses, and evidential relations as they explore science problems, Suthers [5] has gradually shifted the focus of Belvedere away from supporting expert forms of scientific reasoning, and towards supporting meaningful conversations about the science problems under study.

As this line of research has illustrated, end-user environments can play an important role beyond their role in facilitating programming: namely, they can mediate meaningful human discussions about a domain of interest. Building on this work, we have been exploring the communicative role of algorithm visualization technology within an undergraduate algorithms course [6]. Inspired by social learning theory [7], we are particularly interested in its use as part of a novel teaching approach in which students construct their own visualizations of the algorithms under study, and then present those visualizations to their instructor and peers for feedback and discussion. Thus, students are not only end-user programmers, but also end-user *discussants*, since they use their end-user programs (*viz.*, algorithm visualizations) as a basis for discussing algorithms with others. Within the context of such student-constructed visualization exercises, we have been interested in addressing three research questions:

1. In what ways might end-user environments impact human-human communication?
2. What specific design features might make end-user environments well-suited to facilitating human-human communication?
3. What might an end-user environment designed specifically for communication look like, and how might it differ from conventional end-user technology?

In this paper, we use our research into algorithm visualization as a basis for exploring the communicative role of end-user environments. We begin by presenting our key observations in a pair of ethnographic field studies of an undergraduate algorithms course in which students constructed and presented their own visualizations of the algorithms under study. Drawing on those observations, we next present a provisional framework of six dimensions that characterize key ways in which end-user environments support human-human communication: *programming salience*, *typeset fidelity*, *story content*, *modifiability*, *controllability*, and *referencability*. By characterizing those aspects of end-user environments that influence social interaction, our framework provides an important extension to Green and Petre's [1] cognitive dimensions framework.

Making communication-supporting decisions along our communicative dimensions leads to the design of end-user environments with features that differ markedly from those of traditional environments. To illustrate the nature of those differences, we juxtapose conventional algorithm visualization technology with a prototype end-user environment designed specifically for communication about algorithms. Finally, we summarize our framework and its limitations, and we suggest directions for further research.

2 Ethnographic studies

The communicative dimensions presented here were motivated by a pair of ethnographic field studies we conducted in consecutive offerings of a junior-level algorithms course at the University of Oregon. For an assignment in these courses, students were required to construct their own visualizations of one of the divide-and-conquer, greedy, dynamic programming, or graph algorithms they had studied, and then to present their visualizations to their classmates and instructor during specially-scheduled presentation sessions.

To study students' use of algorithm visualization technology in these courses, we employed a variety of *ethnographic field techniques*, including participant observation, semi-structured interviews, videotape analysis, diary collection, and artifact analysis. Below, we briefly summarize the two studies' key findings, providing enough detail to set the stage for our discussion of communicative dimensions in Section 3. For a comprehensive treatment of these studies, see ([6], ch. 4 & app. A–B).

2.1 Ethnographic Study I

In the first of our ethnographic studies, students used the *SAMBA* algorithm animation package [8] to construct visualizations that (a) were capable of illustrating the algorithm for arbitrary input, and (b) tended to have the polished appearance and precision of textbook figures, owing to the fact that they were generated automatically as a by-product of algorithm execution.

To construct visualizations with SAMBA, students began by implementing their target algorithms in C++. Next, they wrote general "animator" classes whose methods were capable of drawing and updating the visualization display (using SAMBA routines) for any input data set. Third, they annotated algorithm source code with these methods at points of "interesting events" [9]. For example, in a sorting algorithm, points at which data items are compared and exchanged might be considered interesting. Finally, they engaged in an iterative process of refining and debugging their visualizations. This process involved (a) compiling and executing their algorithms; (b) noting any problems in the resulting visualization, and (c) modifying their C++ code to fix the problems.

To present visualizations in SAMBA, students used a tape recorder-style interface that allowed them to start, pause, and step through the animation (one frame at a time), and to adjust the execution speed. An additional set of controls in each animation window allowed them to zoom and pan animation views.

In this first study, three key findings were noteworthy. First, students spent 33.2 hours on average (n = 20) constructing and refining a single visualization. They spent most of that time steeped in *low-level graphics programming*—for example, writing general-purpose graphics routines capable of laying out and updating their visualizations for any reasonable set of input data. Second, in students' subsequent presentations, their visualizations tended to stimulate discussions about *implementation details*—for example, how a particular aspect of a visualization was implemented. Third, in response to questions and feedback from the audience, students often wanted to back up and re-present parts of their visualizations, or to dynamically mark-up and modify them. However, conventional AV software like SAMBA is not designed to support interactive presentations in this way.

2.2 Ethnographic Study II

These observations led us to change the visualization assignments significantly for the subsequent offering of the course. In particular, students were required to use simple art supplies (e.g., pens, paper, scissors, transparencies) to construct and present “homemade” visualizations that (a) illustrated the target algorithm for a few, carefully-selected input data sets, and (b) tended to have an unpolished, sketched appearance, owing to the fact that they were generated by hand. In prior work [10], we labeled such visualizations *storyboards*.

In this second study, three key findings stood out. First, students spent 6.2 hours on average (n = 20) constructing and refining a single storyboard. For most of that time, students focused on understanding the target algorithm’s procedural behavior, and how they might best communicate it through a visualization. Second, rather than stimulating discussions about implementation details, their storyboards tended to mediate discussions about the *underlying algorithm*, and about how the visualizations might bring out its behavior more clearly. Third, students could readily go back and re-present sections of their visualizations, as well as mark-up and dynamically modify them, in response to audience questions and feedback. As a result, presentations tended to engage the audience more actively in interactive discussions.

3 Communicative dimensions

As the key observations presented above indicate, the particular technology students used to construct algorithm visualizations, along with the requirements of the visualization programming task they were assigned, greatly influenced the quality and focus of subsequent discussions about those visualizations. What specific features of the end-user technology and task impacted discussions? How, exactly, were those discussions impacted? Drawing extensively from examples we observed in our ethnographic studies, this section develops a provisional framework of six key dimensions of end-user environments that influence their communicative value. The first three dimensions outline qualities of end-user program content that encourage conversations, while the final three dimensions articulate aspects of end-user environments that facilitate the dynamic nature of conversations.

3.1 Programming salience

The *programming salience* dimension asserts that whatever an end-user focuses on during the programming act tends to become the focus of subsequent discussions mediated by the program. On one end of the dimension, end-user environments with high programming salience focus the end-user on relevant domain concepts during the act of programming. It follows that end-users tend to focus on such domain concepts in subsequent discussions mediated by their programs. On the other end of this dimension, end-user environments with low programming salience steep end-users in irrelevant programming details—for example, Green and Petre’s [1] “programming games” and “hard mental operations.” As a result, end-users are distracted from salient domain concepts during programming, and their programs tend to mediate discussions about such irrelevant details.

In Study I, we observed at least three features of SAMBA that caused students' focus to stray from the main focus (algorithms) of the course in which they were enrolled:

- *Quantitative graphics*—the need to size and position the graphics in terms of Cartesian coordinates.
- *Low-level graphics programming*—the need to program visualization graphics in a low-level programming language like C++.
- *Polished graphics*—SAMBA’s support for textbook-like graphics encouraged a process in which students successively tweaked their visualizations until they appeared clean and polished.

Because of these programming distractions, student discussions in Study I tended to focus on implementation details. In particular, students tended to share “war stories” that related their programming toil. For example, in one Study I presentation, students described how they needed 600 lines to implement a single line of pseudocode from the book. They later described a 3,000 line general-purpose graphics library they had implemented to support their animation.

However, when students switched to simple art supplies in Study II, they were no longer encumbered by such implementation details. As a result, subsequent discussions tended to focus on the algorithms being visualized. For example, discussions often considered such questions as What aspects of the algorithm should be illustrated?; How should those aspects be visually represented?; and What are appropriate sample input data?

3.2 Typeset fidelity

The typeset fidelity of a program is the extent to which the program resembles a typeset textbook figure. On the one extreme, programs with *high* typeset fidelity have the highly polished, finished look of textbook figures. We hypothesize that the finished look of a high typeset fidelity program tends to discourage feedback on, and discussion about, the program, since the program tends to be perceived as a finished product that is not open to discussion. On the other extreme, programs with *low* typeset fidelity have a scruffy, unpolished, sketched appearance. Because they appear unfinished, programs with low typeset fidelity tend to encourage discussion and feedback.

Echoing the results of Schumann et al.’s [11] study of architectural drawings, observations made in our ethnographic studies illustrate the role of typeset fidelity in facilitating communication about algorithms. In Study I, students used SAMBA to construct high typeset fidelity visualizations. As we discovered, such visualizations tended to stimulate programming war stories, rather than discussions about algorithm concepts. In contrast, as we found in Study II, students' low typeset fidelity art supply visualizations tended to stimulate more relevant discussions about the algorithms being visualized. It was as though the unpolished appearance of students' art supply visualizations *invited* criticism and feedback.

3.3 Story Content

An end-user program with *story content* portrays domain concepts in terms of an underlying story or metaphor. We hypothesize that end-user programs with story content tend to stimulate livelier discussions than end-user programs that contain purely geometrical elements.

In our ethnographic studies, a minority of students (13%) constructed visualizations based on stories or scenarios in which real or fictitious human beings were engaged in some problem-solving venture. For example, one student in Study II constructed a visualization of the longest common subsequence problem* based on the story of "Knuth's Ark:"

> The world is flooded once again, and Knuth's Ark, which contains a pen of wild animals, lands on an island with a pen of wild animals of its own. Knuth's Ark has limited space, and Knuth must select only those animals on the island that have mates on the Ark. Which ones should he select so as to save the most pairs of animals?

As it turned out, this particular visualization generated a lively, lengthy discussion with two main threads. The first thread concerned the appropriateness of the story as an analogy for the longest common subsequence problem. As the course instructor immediately recognized, in order for the story to work, the animals must be kept in stalls so that their order is maintained. "Otherwise," as he pointed out, "you're solving the biggest common subset, which is another problem." However, this altered storyline proved unsatisfying; it seemed unlikely that Knuth would find the animals on the island confined to such stalls. Accordingly, in the second thread of the discussion, the student and instructor set out to find an appropriate algorithm that matched the Knuth's Ark scenario.

As this example illustrates, while end-user programs with story content may stimulate livelier discussions, such discussions run the risk of straying from underlying domain concepts, and may instead focus on features of the stories themselves.

*Given two sequences of objects, what is the longest (not necessarily contiguous) subsequence that the two sequences have in common? The problem can be solved efficiently using dynamic programming.

3.4 Modifiability

Closely related to Green and Petre's [1] notions of *progressive evaluation* and *viscosity*, a program's *modifiability* reflects the degree to which the program can be dynamically altered in response to the dynamics of a discussion—for example, requests for "what-if" analyses, and suggestions for improvement. We hypothesize that, the more dynamically modifiable the program, the better able the program is to mediate discussions about the underlying domain concepts being represented by the program. In essence, a highly modifiable program serves as a powerful *communicative resource* [12] that enables audience members to participate more fully in a discussion by couching their questions and comments in the program's terms.

In the visualization storyboard presentations of Study II, students and the instructor frequently exploited the modifiability of art supplies in their discussions of algorithms. For example, in her presentation of a storyboard of Kruskal's and Prim's minimum spanning tree algorithms, one student presenter realized partway through that it might be instructive for her storyboard to illustrate why a particular graph edge is *not* selected:

> I was thinking of highlighting the edge [being considered]. If it is not picked because it. . .creates a cycle. Yeah, then I would explain [why] in a window underneath."

Noticeably excited because he could see that something had just clicked for the student, the instructor responded by modifying her storyboard:

> Oh, that would be nice. It would be nice if you can highlight the cycle, so you can actually show, here's a cycle. You can use yet another color for that. Temporarily you show this whole cycle flashing. Sure, that would show this thing working.

Because the student's visualization was constructed out of art supplies, the instructor's suggestions could be tested out on the spot. In contrast, owing to SAMBA's low modifiability, Study I presentations seldom elicited design suggestions like this one.

3.5 Controllability

A program's *controllability* reflects the flexibility with which a presenter can control an end-user program's execution in response to twists and turns in a discussion. This entails the ability to step through a program both forwards and backwards and at varying speeds, as well as the ability to jump to an arbitrary point in the program.

Like modifiability, controllability facilitates communication by enabling a presenter to dynamically respond to an audience's questions and comments. For example, we observed that student presenters frequently fast-forwarded a presentation past the boring parts in order to highlight the interesting parts, with the meaning of "boring" and "interesting" being collaboratively defined during the presentation. In addition, audience members often requested that a presenter step through an interesting section slowly, or go back and revisit it again. In the art supply presentations of Study II, such requests could be easily accommodated. However, because SAMBA does not enable a presenter to execute a visualization in reverse, or to quickly jump to a point of interest, students in Study I could not accommodate such requests.

3.6 Referencability

Deictic gesture is essential to human conversation (see, e.g., [12], pp. 58-62). Indeed, without the ability to point to what one is talking about, statements like "*this* goes to *here*" and "No, I mean *that* one" would remain ambiguous. An end-user program's *referencability* reflects the ease with which conversational participants can refer to elements of the program. High referencability facilitates conversation by making it easy for conversational participants to use an end-user program as a resource for disambiguating contextual references like the ones above. Low referencability, by contrast, inhibits conversation by making it difficult for conversational participants to refer to the program elements they are talking about.

In our ethnographic studies, students' end-user programs served as essential anchors for discussion. Frequently in Study II, students annotated their storyboards as they presented them. For example, they circled two values that the algorithm was currently comparing, or they drew directional arrows to indicate process flow. In addition, they frequently enlisted some sort of a pointer—their fingers, a stick, or a laser—to focus their audience's attention on the referents of an explanation. In the Study I SAMBA presentations, students performed the same kind of deictic pointing, often with the computer mouse cursor; however, SAMBA did not allow them to mark up their visualizations during presentation.

4 Design implications

The communicative dimensions just presented have important implications for the design of end-user environments. In particular, decisions made along the dimensions can lead to end-user environments that more or less support human communication. Table 1 identifies the extremes of the design space circumscribed by the dimensions. For each dimension, the table lists the corresponding design implication for both communication-inhibiting and communication-supporting end-user environments.

The design implications listed in Table 1 are admittedly high-level and imprecise. Indeed, the next question to ask is, To what specific design features might communication-supporting and communication-inhibiting decisions along the dimensions lead? To answer this question, Table 2 grounds the design implications of Table 1 in concrete design features of two markedly different end-user environments for algorithm visualization: (a) SAMBA [8], the end-user environment used by students in our first ethnographic study (see Section 2.1); and (b) ALVIS ([6], ch. 7) a prototype end-user environment we have developed specifically to support human conversations about algorithms. In the remainder of this section, we elaborate on the features of these two environments that influence their communicative value.

4.1 Conventional: SAMBA

On one end of the communicative continuum defined by our dimensions is SAMBA [8], which defines a high-level scripting language geared specifically toward the construction of algorithm visualizations (see Figure 1b). Two features of SAMBA, however, give it low programming salience. First, it relies on Cartesian coordinates to size and position visualization objects. As we found in our ethnographic studies, having to lay out a program in terms of Cartesian coordinates is distracting. Second, as described in Section 2.1, SAMBA is intended for use as an annotation language: one programs a visualization by annotating a conventional program (e.g., C++) with calls to SAMBA routines that create and update the visualization (see Figure 1a). As we discovered in our ethnographic studies, programming with SAMBA in this way tends to steep one in low-level graphics, diverting one's focus from the algorithm being visualized.

With respect to typeset fidelity, SAMBA requires visualization programs to be laid out precisely in terms of Cartesian coordinates. As our ethnographic observations indicate, that requirement encourages "tweaking," resulting in visualizations on the high end of the typeset fidelity continuum.

While SAMBA is neutral with respect to the story content dimension, it tends toward the communication-inhibiting end of the modifiability, controllability, and referencability dimensions. As mentioned above, SAMBA is intended for use as an annotation language embedded within a conventional compiled programming language such as C++. As a result, modifying a SAMBA program requires a time-consuming edit-compile-execute cycle, which is clearly impractical within the scope of an interactive discussion. Likewise, SAMBA's reliance on a compiled language places it on the low end of the controllability dimension; SAMBA's execution interface supports only forward stepping and execution (see Figure 1c). Finally, SAMBA lies on the low end of the referencability dimension, since it provides no explicit support for on-the-spot mark-up or pointing.

DIMENSION	DESIGN IMPLICATIONS: Inhibit Communication		DESIGN IMPLICATIONS: Support Communication
Programming salience	Support low-level, general-purpose programming	⟶	Support high-level, domain-specific programming
Typeset fidelity	Support creation of polished, textbook-style graphics	⟶	Support creation of sketched, unpolished graphics
Story Content	Prohibit construction of programs with storyline	⟶	Support construction of programs with storyline
Modifiability	Support lengthy modification cycle that recompilation and reexecution	⟶	Support dynamic modification of a program while it is executing
Controllability	Support forwards stepping and execution only	⟶	Support reverse execution and ability to quickly jump to arbitrary execution point
Referencability	Do not support dynamic program mark-up or a conspicuous pointer	⟶	Support dynamic program mark-up and conspicuous pointer

Table 1. Implications of each communicative dimension for the design of communication-inhibiting and communication-supporting end-user environments

DIMENSION	CORRESPONDING DESIGN FEATURES: SAMBA (Conventional)	CORRESPONDING DESIGN FEATURES: ALVIS (Designed to support communication)
Programming salience	Requires programming algorithm and visualization in low-level programming language (e.g., C++)	Supports visualization programming at a conceptual level in terms of spatial relations
Typeset fidelity	Supports polished graphics with precise layout via Cartesian coordinates	Supports sketching and imprecise layout via direct manipulation
Story Content	No design features directly support story content	No design features directly support story content
Modifiability	Modification requires editing low-level source code, re-compiling, and re-executing	SALSA interpreted language can be easily modified during execution
Controllability	Supports forwards stepping and execution only	Supports forwards and backwards stepping and execution, as well as ability to jump to arbitrary execution point
Referencability	No support for dynamic annotation	Support for dynamic annotation and conspicuous pointer

Table 2. Juxtaposition of conventional (SAMBA) and communication-supporting (ALVIS) algorithm visualization technology vis-à-vis the six communicative dimensions

4.2 Communication-supporting: ALVIS

On the other end of the communicative continuum defined by our dimensions is ALVIS ([6], ch. 7) a prototype end-user environment we have developed specifically to support human communication about algorithms. Two specific features of ALVIS are designed to give it high programming salience:

- *Support for spatial analogies*. Underlying ALVIS is SALSA (Spatial Algorithmic Language for StoryboArding), a high-level, interpreted language specifically designed to focus an end-user on the target algorithm being visualized (see contents of left-hand window in Figure 2a for a sample SALSA script). Unlike SAMBA, SALSA enables the layout and logic of a visualization to be specified in terms of its *spatiality*—that is, in terms of the spatial relations (e.g., *above, right-of, in*) among objects in the visualization. Thus, programming an algorithm visualization in ALVIS amounts to constructing a *spatial analogy* for the target algorithm.
- *Support for direct-manipulation graphics*. ALVIS provides a sketch-based graphics editor (see Figure 2b) designed to make visualization object creation as quick and easy as cutting out and sketching on scraps of construction paper. After one creates a *cutout* with this editor, the cutout appears in the created object palette (lower-right window of Figure 2a). To place a cutout, one can simply drag-and-drop it into the visualization window (upper right window of Figure 2a).

ALVIS's sketch-based interface, which spares end-users from Cartesian coordinates, supports the creation of low typeset fidelity visualizations that encourage communication about algorithm concepts.

Like SAMBA, ALVIS is neutral with respect to the story content dimension; it contains no specific features that support the creation of visualizations with an underlying storyline. In fact, we are hard pressed to identify any specific design features that would support story content. As we see it, a storyline comes from a creative programmer's imagination, not an end-user environment.

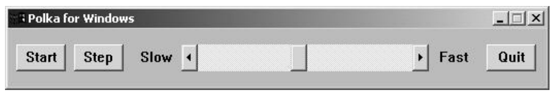
```
extern MyAnimator football_bsort;

  main(int argc, char *argv[])
  {
    int n,i,j,temp,a[50],count;
    char str[100];
     . . .
    football_bsort.Ready(n);
      for (j=n-2; j>=0; --j) {
        football_bsort.StartOuter();
        for (i=0; i<=j; ++i) {
          football_bsort.Compare(i,i+1);
           . . .
        }
        football_bsort.InPlace(j+1);
      }
    . . .
    }
```

(a) Annotating a C++ program with SAMBA routines

```
football_bsort::Compare(int i, int j) {
  //write SAMBA commands to script file
  outfile << "{\n"
          << "moveto "
          <<  player[i]
          << " "
          << step_up_loc[i]
          << "\n";
          << "moveto "
          << ball
          << " "
          << step_up_ball_loc[i]
          << "\n"
          << "}\n"
          << "delay 7\n";
 }
```

(b) A SAMBA script



(c) Execution control interface

Figure 1. The SAMBA end-user environment

Especially with respect to the remaining three dimensions, which characterize the ability of an end-user environment to mediate communication *in situ*, ALVIS has been designed on the communication-enhancing end of the continuum. First, because SALSA is an interpreted language, and because the visualization view (upper right-hand window of Figure 2a) is always synchronized with the SALSA script view (left-hand window of Figure 2a), ALVIS has high modifiability. An end-user always has the option of directly editing the SALSA script while it is running. Any changes made dynamically will be immediately reflected in the execution of the script. Second, ALVIS's execution interface (Figure 2c) supports high controllability. At any point, an end-user may reverse the direction of storyboard execution in response to audience questions and comments, or jump to the execution point defined by the cursor. Finally, ALVIS provides a "mark up pen" (third tool from left in Figure 2d) and a conspicuous "presentation pointer" (fourth tool from left in Figure 2d) that support high referencability. These tools enable the end-user to mark-up and point to a storyboard at any point during a discussion.

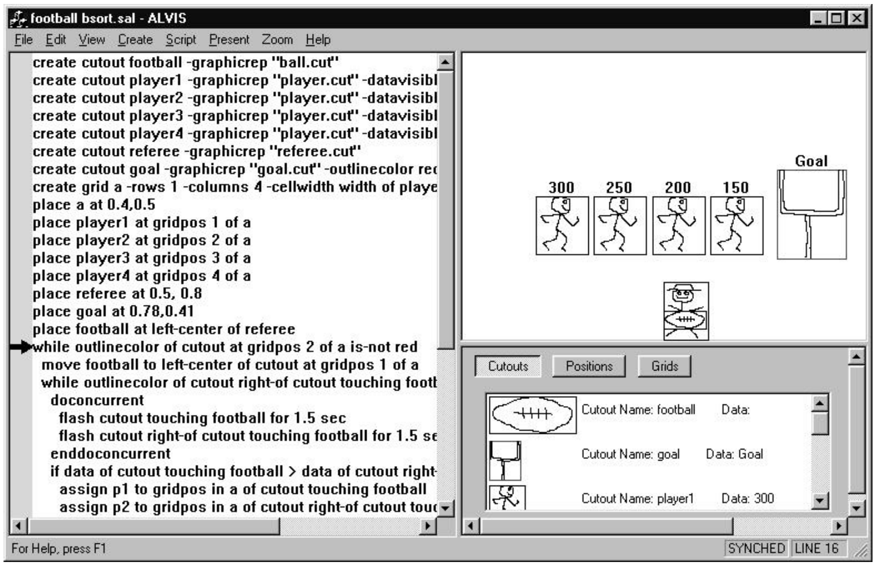
5 Summary and future research

In this paper, we have argued that, in addition to their traditional role in easing programming, end-user environments can serve as powerful resources for facilitating human communication about a domain of interest. Drawing on ethnographic studies of algorithm visualization technology, we have presented a preliminary characterization of that communicative role in the form of a framework of communicative dimensions. Just as Green and Petre's framework of cognitive dimensions [1] articulates "cognitively-relevant" (p. 131) aspects of end-user environments, so too does our framework aim to characterize those aspects of end-user environments that promote and support human communication.

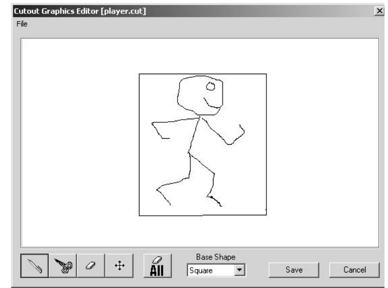
Like Green and Petre's cognitive dimensions framework, our framework is not intended to be a set of design guidelines. Rather, it aims to serve as a preliminary "discussion tool" ([1], p. 132) for designers and evaluators interested in *communication-supporting* end-user environments. This brings to light an important limitation of our communicative dimensions: They are by no means relevant to *all* end-user environments. Indeed, the dimensions apply mainly to *collaborative* end-user environments that, for example, aim to foster understanding (e.g., [4, 5]) or build design consensus (e.g., [13, 14]); they do not necessarily apply to single-user, task-oriented end-user environments such as spreadsheets.

The communicative dimensions presented here raise several key questions for future research. For example, might it be possible, through empirical studies, to establish more precise cause-effect relationships between communicative design features and communicative activity? Such empirical studies will require research methodologies capable of analyzing and quantifying human communication—for instance, conversation analysis [12] and content analysis.

A second key question has to do with theory: To what extent can the framework be grounded in an underlying theory of communication? To that end, we have found Suchman's [12] *situated action theory* to be a useful start-



(a) Snapshot of a session with ALVIS



(b) Cutout graphics editor

(c) Execution control interface

(d) Presentation interface

Figure 2. The ALVIS end-user environment

ing point. Future work will need to develop a sound rationale for the dimensions based on this and other theories.

A third key question concerns extending the framework: What other aspects of end-user environments impact human communication? For example, Suthers's [5] research into science learning discourse explores a "representational bias" hypothesis that asserts additional cause-effect relationships between design features and communication. In future research, we aim to collect these and other communicative dimensions into a unified framework for the design and evaluation of communication-supporting end-user environments.

6 References

[1] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Langauges and Computing*, vol. 7, pp. 131-174, 1996.

[2] B. A. Nardi, *A small matter of programming: Perspectives on end-user computing*. Cambridge, MA: The MIT Press, 1993.

[3] A. Repenning and W. Citrin, "Agentsheets: applying grid-based spatial reasoning to human-computer interaction," in *Proceedings of the 9th IEEE Workshop on Visual Languages*. Los Alamitos, CA: IEEE Computer Society Press, 1993, pp. 77-82.

[4] J. Roschelle, "Designing for conversations," presented at AAAI Symposium on Knowledge-Based Environments for Learning and Teaching, Stanford, CA, 1990.

[5] D. Suthers, "Representational support for collaborative inquiry.," in *Proceedings of the 32nd Hawai`i International Conference on the System Sciences*. Los Alamitos, CA: IEEE Computer Society Press, 1999.

[6] C. D. Hundhausen, "Toward effective algorithm visualization artifacts: Designing for participation and communication in an undergraduate algorithms course," Unpublished Ph.D. Dissertation, Department of Computer and Information Science, University of Oregon, 1999.

[7] J. Lave and E. Wenger, *Situated Learning: Legitimate Peripheral Participation*. New York: Cambridge University Press, 1991.

[8] J. T. Stasko, "Using student-built animations as learning aids," in *Proceedings of the ACM Technical Symposium on Computer Science Education*. New York: ACM Press, 1997, pp. 25-29.

[9] M. H. Brown, *Algorithm animation*. Cambridge, MA: The MIT Press, 1988.

[10] S. A. Douglas, C. D. Hundhausen, and D. McKeown, "Toward empirically-based software visualization languages," in *Proceedings of the 11th IEEE Symposium on Visual Languages*. Los Alamitos, CA: IEEE Computer Society Press, 1995, pp. 342-349.

[11] J. Schumann, T. Strothotte, A. Raab, and S. Laser, "Assessing the effect of non-photorealistic rendered images in CAD," in *Proceedings of the ACM CHI 96 Conference*. New York: ACM Press, 1996, pp. 35-41.

[12] L. A. Suchman, *Plans and Situated Actions: The Problem of Human-Machine Communication*. New York: Cambridge University Press, 1987.

[13] C. H. Damm, K. M. Hansen, and M. Thomsen, "Tool support for cooperative object-oriented design: Gesture based modelling on an electronic whiteboard," in *Proceedings of ACM CHI 2000 Conference*. New York: ACM Press, 2000, pp. 518-525.

[14] J. A. Landay and B. A. Myers, "Interactive sketching for the early stages of user interface design," in *Proceedings of the ACM CHI '95 Conference*. New York: ACM Press, 1995, pp. 43-50.