Low Fidelity Algorithm Visualization

(Submitted Dec. 12, 2000; revised June 22, 2001; accepted Oct. 25, 2001)

CHRISTOPHER D. HUNDHAUSEN
Laboratory for Interactive Learning Technologies
Information and Computer Sciences Department
University of Hawai`i
Honolulu, HI 96822

SARAH A. DOUGLAS
Human-Computer Interaction Lab
Computer and Information. Science Department
University of Oregon
Eugene, OR 97403–1202

Please address correspondence to

Christopher D. Hundhausen
Information and Computer Sciences Department
University of Hawai`i
1680 East West Road, POST 303D
Honolulu, HI 96822 USA
Phone: (808) 956-3887
Fax: (808) 956-3548
hundhaus@hawaii.edu

Abstract

Computer science educators have traditionally used algorithm visualization (AV) software to create graphical representations of algorithms for use as visual aids in lectures, or as the basis for interactive labs. Typically, such visualizations are *high fidelity* in the sense that (a) they depict the target algorithm for arbitrary input, and (b) they tend to have the polished look of textbook figures. In contrast, *low fidelity* visualizations illustrate the target algorithm for a few, carefully chosen input data sets, and tend to have a sketched, unpolished appearance. Drawing on ethnographic field studies of a junior-level undergraduate algorithms course, we motivate the use of low fidelity AV technology as the basis for an alternative learning paradigm in which students construct their own visualizations, and then present those visualizations to their instructor and peers for feedback and discussion. To explore the design space of low fidelity AV technology, we present SALSA (Spatial ALgorithmic Language for StoryboArding) and ALVIS (ALgorithm VIsualization Storyboarder), a prototype end-user language and system firmly rooted in empirical studies in which students constructed and presented visualizations made out of simple art supplies. Our prototype end-user language and system pioneer a novel technique for programming of visualizations based on spatial relations, and a novel presentation interface that supports human discussions about algorithms by enabling reverse execution and dynamic mark-up and modification. Moreover, the prototype provides an ideal foundation for what we see as the algorithms classroom of the future: the interactive "algorithms studio."

1. Introduction

Algorithm visualization (AV) software supports the construction and interactive exploration of visual representations of computer algorithms, e.g., [1-5]. Traditionally, computer science instructors have used the software to construct visualizations that are later used either as visual aids in lectures, e.g., [6], or as the basis for interactive labs, e.g., [3].

Despite the enthusiasm and high expectations of AV software developers, a review of 21 experimental evaluations ([7], §2 and 3; [8]; [9], ch. 4–9; [10]; [11], I – VIII; [12]; [13]; [14]) casts doubt on the software's pedagogical benefits. Indeed, only 13 of those experiments ([7], §2 and 3; [10]; [9] ch. 6, 7, 9; [11], I, II, IV, V, VII, VIII; [12]) showed that some aspect of AV technology or its pedagogical application significantly impacted learning outcomes.

Further analysis of these experiments suggests that they fall into two broad categories based on the factors they identify as critical to the experimental evaluation of AV. Lawrence ([9], ch. 4, 5, 7, 8), Stasko, Badre, and Lewis [8], Gurka [14], and Hansen et al. ([11], VI – VIII) varied *representational characteristics* of the learning materials, including (a) text versus animation, (b) text-first or animation-first, and (c) various graphical attributes of animations. By contrast, a second group of studies ([7], §2 and 3; [10]; [11], I – V; [13]; [9], ch. 6 and 9) varied level of *learner involvement*. In addition to having learners passively view an animation, these studies involved learners more actively by having them (a) construct their own data sets, (b) answer strategically-chosen questions about the algorithm; (c) make predictions regarding future algorithm behavior, or (d) program the algorithm.

If one considers the results of the experiments vis-à-vis these two categories (Figure 1), a notable trend emerges: experiments that manipulate learners' level of involvement have consistently yielded significant results, whereas experiments that have manipulated representation have not. This suggests that what learners do, not what they see, may have the greatest impact on learning—an observation that well accords with *constructivist* learning theory (see, e.g., [15]). (For an expanded meta-analysis of these experiments, including an assessment of the extent to which they support alternative learning theories, see [16].)

Given this observation, it makes sense to consider pedagogical approaches that get students maximally involved in the process of visualizing an algorithm. To that end, Stasko [5] advocates the use of "visualization assignments" in which students perform end-user programming by using AV software to construct their own visualizations of the algorithms under study. Inspired by social constructivist learning theory [17], which views AV software as pedagogically valuable insofar as it enables students to participate in a course in ways that increasingly resemble the ways in which teachers participate, we became interested in a teaching approach that takes visualization assignments one step further by having students not only construct their own visualizations, but also *present* their visualizations to their instructor and peers for feedback and discussion ([18], ch. 4). Notice that, in this approach, students assume two tasks that closely resemble those typically performed only by teachers: (1) they become end-user programmers by constructing their own visualizations for classroom presentation; and (2) they become end-user *discussants* by using their end-user programs (algorithm visualizations) as a basis for talking about algorithms with others.

To explore the practical costs and benefits of this approach, we conducted a pair of ethnographic field studies in consecutive offerings of a junior-level undergraduate algorithms course that included visualization construction and presentation assignments. Our findings have led us not only to endorse this teaching approach as an effective way of getting students involved in and excited about algorithms, but also to advocate a fundamental redesign of traditional AV software so that it better supports the construction and presentation of algorithm visualizations by student end-users.

Specifically, our findings point out a key distinction between *high fidelity* and *low fidelity* visualizations. In our first field study, we had students use conventional AV software to construct high fidelity visualizations, which (a) are capable of illustrating the target algorithm for arbitrary input, and (b) tend to have the polished look of textbook figures. By contrast, in our second study, we had students use simple art supplies (e.g., construction paper, pens, scissors) to construct low fidelity visualizations, which do not necessarily illustrate the target algorithm for arbitrary input, and tend to have a sketched, unpolished look. We found that, compared to the construction and presentation of high fidelity visualizations, the construction and presentation of low fidelity visualizations engages students in activities and discussions that are much more relevant to an undergraduate algorithms course.

In this article, we present a novel, *low fidelity* approach to integrating AV technology into an undergraduate algorithms course. We begin by describing a pair of ethnographic field studies that both motivate, and inform, our approach. To explore the design space of low fidelity AV technology, we next present a prototype end-user language and system firmly grounded both in our ethnographic studies, and in prior detailed studies of how students construct and execute low fidelity visualizations made out of simple art supplies [19, 20]. We then step back and consider the specific ways in which our prototype differs from extant AV systems. We conclude by discussing future directions for low fidelity AV technology, including our vision of the interactive "algorithms studio": a novel teaching approach that uses low fidelity AV technology to implement a studio-based model of algorithms instruction.

2. Ethnographic Studies

The redesign of AV software advocated in this article is motivated by a pair of ethnographic field studies we conducted in consecutive offerings of a junior-level undergraduate algorithms course at the University of Oregon ([18], ch. 4). These ten week courses revolved around three, 50 minute lectures per week, and one help session per week, led by the teaching assistant. The book used in these courses was the standard Cormen, Leiserson, and Rivest [21] text.

Thirty-eight computer science majors were enrolled in the first of these CIS 315 courses; forty-nine were enrolled in the second. Students ranged in age from 20 to over 40, with most of them closer to 20. Most students in the courses were male; five females were enrolled in each of the two courses. The course instructor was a tenured professor who had been teaching the algorithms course at the university for over 12 years. In addition to holding regular office hours, the instructor gave nearly all of the course lectures, did some of the grading, and led in some of the weekly discussion sections. A teaching assistant for the course held weekly office hours, did most of the grading, led most of the weekly discussion sections, and occasionally gave lectures when the instructor was out of town.

For an assignment worth roughly 20% of the course grade, 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. In the remainder of this section, we describe the field techniques we employed to study the use of algorithm visualization technology in these courses; we present our key results in each of our two studies; and we consider their implications for the design of algorithm visualization technology.

2.1 Field Techniques

In our study, the first author played the dual-role of *student observer* and (volunteer) *teaching assistant for algorithm animation*. As student observer, he sat in on lectures and took notes; interacted with students before and after lectures, and occasionally when he ran into them in the computer science department; and arranged to observe and work with certain groups of students as they worked on animation assignments in the undergraduate computer lab.

As the teaching assistant for algorithm animation, the first author collaborated with the instructor in the development of the algorithm animation curriculum; set up and maintained the software used for the algorithm animation assignments; gave introductory lectures on algorithm animation and the course animation assignments; made himself available via e-mail, and before and after class, for questions regarding algorithm animation; and interacted regularly with the instructor regarding a variety of issues surrounding the animation assignments.

In this dual-role, the first author made use of seven different *ethnographic field techniques* (see, e.g., [22]). First, he made extensive use of *participant observation* to participate in and observe the algorithm animation-related activities of both students and the instructor. Second, he used two different kinds of interviewing techniques to elicit informants' perceptions and experiences. In his day-to-day interaction with students and the instructor, he tended to ask a lot of questions on an informal basis (*informal interviewing*). On several occasions, he followed up on the important themes and issues that emerged from those informal inquiries by audiotaping (and subsequently transcribing and analyzing) *semi-structured interviews* with students and the instructor. Third, during Study I, he administered two brief on-line *questionnaires* to the members of a volunteer mailing list. These questionnaires elicited students' general impressions regarding the algorithm animation assignment, what activities they performed, and estimates of the amount of time spent on each activity.

Fourth, he took extensive *fieldnotes* during both terms of the fieldwork, both during lectures, and after his discussions with the instructor and student informants. Fifth, he *audio*- or *videotaped* (and subsequently transcribed and analyzed) all of the storyboard and final animation presentation sessions that were held during both terms of the fieldwork. Sixth, he *collected and analyzed artifacts*—both the Samba animations, and the low fidelity storyboards that students handed in. Finally, he *collected and analyzed diaries* that students were required to hand in as part of the assignment. Their diaries documented what they did for the animation assignment, what problems they encountered, and how much time they spent.

2.2 Ethnographic Study I: High Fidelity Visualizations

In the first of our ethnographic studies, students used the *Samba* algorithm animation package [5] to construct *high fidelity* 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 as a byproduct of algorithm execution.

To construct such 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" [1]. 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 the tape recorder-style interface illustrated in Figure 2. The interface 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 (not shown) allowed them to zoom and pan animation views.

In this first study, three key findings are 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 (using Cartesian coordinates) 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. Indeed, modifying a high fidelity visualization requires one to edit, recompile, and re-execute low-level source code, which is not feasible within the scope of an interactive presentation.

2.3 Ethnographic Study II: Low Fidelity Visualizations

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 *low fidelity* visualizations that (a) illustrated the target algorithm for just a few 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 [19, 20], we have called such low fidelity visualizations *storyboards*.

In this second study, three key findings stand out. First, students spent 6.2 hours on average (n = 20) constructing and refining a single visualization 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. In particular, discussions surrounding four key questions emerged:

- What aspects of the algorithm should be illustrated and elided?
- How should those aspects be visually represented?
- What are appropriate sample input data?
- How should multiple visualization views be coordinated?

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.

2.4 Discussion: From High to Low Fidelity Algorithm Visualization

Our study findings point out an important distinction between what we have labeled *low fidelity* and *high fidelity* visualizations. We believe that, rather than being binary (*high* or *low*), *visualization fidelity* can actually be seen as a continuum. On the lowest end of this spectrum, visualizations resemble those that one might quickly sketch on a napkin at a cocktail party. Such low fidelity visualizations are highly unpolished, and illustrate the target algorithm for exactly one set of input data. While uncommon in the domain of algorithm visualization, several end-user sketching systems have been developed for quickly creating low fidelity visualizations in other domains. For example, Landay and Myers' SILK [23] enables end-users to sketch low-fidelity user interface prototypes, and Lin *et al.*'s DENIM [24] supports the creation of low fidelity website prototypes by sketching.

By contrast, on the highest end of the spectrum, visualizations have the highly-polished look of textbook figures, and are capable of illustrating the target algorithm for any reasonable input. For example, Naps's GAIGS system [3] automatically produces high fidelity data structure drawings that closely resemble those that appear in textbooks. Likewise, Brown's BALSA animation system was used to generate high fidelity visualizations that appear as figures in some versions of Sedgewick's algorithms textbook [25].

In between the "low" and "high" fidelity extremes, one can imagine many possible variations. For example, an illustrator commonly creates highly-polished textbook figures for *specific* input data sets. Similarly, non-artistic students and instructors commonly create less-polished general-input algorithm animations using an algorithm animation package like Samba [5].

In addition to suggesting this *visualization fidelity* continuum, our studies furnish three reasons why constructing and presenting low fidelity visualizations constitutes a more productive learning experience in an undergraduate algorithms course than constructing and presenting high fidelity visualizations. First, low fidelity visualizations not only take far less time to construct, but also keep students more focused on topics relevant to an undergraduate algorithms course: algorithm concepts, rather than low-level implementation details. Second, in student presentations, low fidelity visualizations stimulate more relevant discussions that focus on algorithms, rather than on implementation details. Finally, low fidelity visualizations are superior in interactive presentations, since they can be backed up and replayed at any point, and even marked-up and modified on the spot, thus enabling presenters to respond more dynamically to their audience.

Finally, in identifying the educational advantages of low fidelity AV technology, our study findings motivate further research into the low fidelity AV approach. For AV technologists, one key research topic has to do with the design of a computer-based environment. In our ethnographic fieldwork, we had students construct low fidelity visualizations out of simple art supplies. We view this as the "paradigm case" for low fidelity visualization technology. Indeed, our observations suggest that people appear to be familiar and comfortable with art supplies, enabling them to express their visualizations of algorithms in an extremely natural way. At the same time, we believe that there is good reason to pursue a computer-based environment modeled after art supplies. For example, rather than having to discard a visualization storyboard that has been marked up during the course of a presentation, a presenter of a computer-based storyboard can re-use the storyboard by simply "erasing" any annotations. Likewise, rather than having to create a new visualization storyboard, by hand, for each input data set of interest, users of a computer-based environment can create one visualization storyboard, and simply tweak the data values of the objects.

In short, we believe that the flexibility and dynamism of the computer, if leveraged properly, can lead to a computer-based low fidelity visualization environment that, as compared to conventional art supplies, is easier to use and saves time. An important research question thus becomes, "If we take our study findings with simple art supplies as constraints on the design space of low fidelity AV technology, what kind of computer-based end-user AV software emerges?" We present research into that question in the following section.

3. Prototype Language and System

To explore the design space of low fidelity AV technology circumscribed by our findings, we have developed a prototype end-user language and system that supports the student construction and presentation of low fidelity visualizations. We begin this section by deriving the design requirements for the system from our empirical research. Next, we overview our prototype's key features and functionality. To provide a feel for our prototype in use, we then present a detailed example. We conclude by juxtaposing our prototype with related work. For a more comprehensive treatment of our prototype system, see ([18], ch. 7).

3.1 Deriving Design Requirements

A fundamental objective or our prototype implementation was to root its design firmly in empirical data. Pertinent observations from our ethnographic studies, as well as from our prior detailed studies of how students construct algorithm visualizations out of simple art supplies [20, 26], provide a solid empirical foundation for the design of our prototype. In our ethnographic fieldwork, we gathered 40 low fidelity storyboards that were constructed using conventional art supplies, including transparencies, pens, and paper. The following generalizations can be made regarding their content:

- The storyboards consisted of groups of movable, objects of arbitrary shape (but most often boxes, circles, and lines) containing sketched graphics; regions of the display, and locations in the display, were also significant.
- Objects in storyboards were frequently arranged according to one of three general layout disciplines: grids, trees, and graphs.
- The most common kind of animation was simple movement from one point to another; pointing (with fingers) and highlighting (circling or changing color) were also common. Occasionally, multiple objects were animated concurrently.

These observations motivate our first design requirement:

R1:	Users must be able to create, systematically lay out, and animate simple objects containing sketched graphics.

With respect to the process of visualization construction, we made the following observations in our previous studies of how humans construct low fidelity visualizations out of art supplies [20, 26]:

- Storyboard designers create objects by simply cutting them out and/or sketching them, and placing them on the page.
- Storyboard designers never size, place or move objects according to Cartesian coordinates. Rather, objects are invariably sized and placed relative to other objects that have already been placed in a storyboard.

These observations motivate our second and third design requirements:

R2:	Users must be able to construct storyboard objects by cutting and sketching; they must be able to position objects by direct placement.

R3:	Users must be able to create storyboards using spatial relations, not Cartesian coordinates.

Finally, with respect to the process of visualization execution and presentation, observations made both in our ethnographic studies, and in our prior empirical studies [20, 26], suggest the following:

- Rather than referring to program source code or pseudocode, storyboard presenters tend to simulate their storyboards by paying close attention to, and hence maintaining, important *spatial relations* among storyboard objects. For example, rather than maintaining a numeric looping variable, storyboard designers might stop looping when an arrow advances to the right of a row of boxes.

- Storyboard presenters provide verbal play-by-play narration as they run their storyboards. In the process, they frequently point to storyboard objects, and occasionally mark-up their storyboards with a pen. The audience often interrupts presentations with comments or questions. To clarify their comments and questions, they, too, point to and mark up the storyboard.
- In response to audience comments and questions, presenters pause their storyboards, or even fast-forward or rewind them to other points of interest.
- Audience suggestions often lead to on-the-spot modifications of the storyboard—for example, changing a color scheme, adding a label, or altering a set of input data.

These observations motivate the following two requirements, which round out our list:

R4: The system must enable one to program an animation by using spatial logic to model algorithmic logic.

R5: The system must enable users to present their storyboards interactively. This entails an ability to execute storyboards in both directions; to rewind and fast forward storyboards to points of interest; and to point to, mark-up, and modify storyboards as they are being presented.

3.2 Overview of Language and System

The requirements just outlined circumscribe the design space for a new breed of *low fidelity* AV technology. To explore that design space, we have developed a prototype language and system for creating and presenting low fidelity algorithm visualizations.

The foundation of our prototype is SALSA (Spatial Algorithmic Language for StoryboArding), a high-level, interpreted language for programming low fidelity storyboards. Whereas conventional *high fidelity* AV technology requires one to program a visualization by specifying explicit mappings between an underlying "driver" program and the visualization, SALSA enables one to specify *low fidelity* visualizations that drive themselves; the notion of a "driver" algorithm is jettisoned altogether. In order to support visualizations that drive themselves, 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.

The SALSA language is compact, containing just three data types and 12 commands. SALSA's three data types model the core elements of the art supply storyboards observed in the empirical studies discussed in the previous sections:

- *Cutout*. This is a computer version of a construction paper cutout. It can be thought of as a movable scrap of construction paper of arbitrary shape on which graphics are sketched.
- *Position*. This data type represents an x,y location within a storyboard. Note that, in accordance with R3 above, SALSA users never have to deal directly with Cartesian coordinates. They can create positions and lay out objects by direct manipulation, as we shall explain shortly.
- *S-struct.* A spatial structure (s-struct for short) is a closed spatial region in which a set of cutouts can be systematically arranged according to a particular spatial layout pattern. The prototype implementation of SALSA supports just one s-struct: grids.

SALSA's command set includes *create*, *place*, and *delete* commands for creating, placing and deleting storyboard elements; an *if-then-else* statement for conditional execution; *while* and *for-each* statements for iteration; and *move*, *flash*, *resize*, and *do-concurrent* statements for animating cutouts.

The second key component of the prototype is ALVIS (ALgorithm VIsualization Storyboarder), an interactive, direct manipulation front-end interface for programming in SALSA. Figure 3a presents a snapshot of the ALVIS environment, which consists of three main regions:

- *Script View (left).* This view displays the SALSA script presently being explored; the arrow on the left-hand side denotes the line at which the script is presently halted—the "insertion point" for editing.
- *Storyboard View (upper right).* This view displays the storyboard generated by the currently-displayed script. The Storyboard View is always synchronized with the Script View. In other words, it always reflects the execution of the SALSA script up to the current insertion point marked by the arrow.
- *Created Objects Palette (lower right).* This view contains an icon representing each cutout, position, and grid that has been created thus far.

The ALVIS environment strives to make constructing a SALSA storyboard as easy as constructing a homemade storyboard out of simple art supplies. To do so, its conceptual model is firmly rooted in the physical metaphor of "art supply" storyboard construction. An important component of this metaphor is the concept of *cutouts* (or *patches*; see [27]): scraps of virtual construction paper that may be cut out and drawn on, just like real construction paper. In ALVIS, end-users create storyboards by using a graphics editor (Figure 1b) to cut out and sketch cutouts, which they lay out in the *Storyboard View* by direct manipulation. They then specify, either by direct manipulation or by directly typing in SALSA commands, how the cutouts are to be animated over time.

Likewise, ALVIS strives to make presenting a SALSA storyboard to an audience as easy and flexible as presenting an “art supply” storyboard. To that end, ALVIS’s presentation interface supports four features that are taken for granted in “art supply” presentations, but that are notably absent in conventional AV technology. First, using ALVIS’s execution interface (Figure 1c), a presenter may reverse the direction of storyboard execution in response to audience questions and comments. Second, ALVIS provides a conspicuous “presentation pointer” (fourth tool from left in Figure 1d) with which the presenter and audience members may point to objects in the storyboard as it is executing. Third, ALVIS includes a “mark up pen” (third tool from left in Figure 3d) with which the presenter and audience members may dynamically annotate the storyboard as it is executing. Finally, presenters and audience members may dynamically modify a storyboard as it is executing by simply inserting SALSA commands at the current insertion point in the script.

3.3 Example Use of Language and System

Perhaps the best way to provide a feel for the features and functionality of SALSA and ALVIS is through an illustrative example. In this section, we use SALSA and ALVIS to create and present the “football” storyboard (see Figure 4) of the bubble sort algorithm we observed in prior empirical studies [20, 26].

The “football” storyboard models the bubble sort algorithm in terms of a game of American football. Elements to be sorted are represented as football players whose varying weights (written under each player in Figure 4) represent element magnitudes. At the beginning of the game, the players are lined up in a row. The referee then tosses the ball to the left-most player in the line, who becomes the ball carrier. The object of the game is to “score” by advancing the ball to the end of the line of unsorted players. If the ball carrier is heavier than the player next in line, then the ball carrier simply tackles the next player in line, thereby switching places with him. If, on the other hand, the ball carrier is lighter than the player next in line, then the ball carrier is stopped in his tracks, fumbling the ball to the next player in line. This process of ball advancement continues until the ball reaches the last player in the line of unsorted players. Having found his rightful place in the line of players, that last player with the ball tosses the ball back to the referee. A pass of the algorithm’s outer loop thus completes. If, at this point, there are still players out of order, the referee tosses the ball to the first player in line, and another pass of the algorithm’s outer loop begins.

3.3.1 Creating the Storyboard Elements

We begin by creating the cutouts that appear in Figure 4: four players, a football, a referee, and a goal post. With respect to the football players, our strategy is to create one "prototype" player, and then to clone that player to create the other three players. To create the prototype player, we select *Cutout…* from the *Create* menu, which brings up the *Create Cutout* dialog box (see Figure 5a). We name the cutout "player1," and use the *Cutout Graphics Editor* (Figure 3b) to create its graphics in the file "player.cut": a square scrap of construction paper with a football player stick figure sketched on it. In addition, to indicate that this player weighs 300 pounds, we store the value "300" in the cutout's *data* attribute. We decide not to set any other attributes explicitly, accepting the defaults.

When the *Create Cutout* dialog box is dismissed, ALVIS inserts the following SALSA create statement into the *SALSA Script View*:

```
create cutout player1 –graphic-rep "player.cut" –data "300"
```

In addition, the player1 cutout appears in the *Created Objects Palette*, and the execution arrow (in the *SALSA Script View*) is advanced to the next line, indicating that the create statement has been executed.

We now proceed to clone player1 three times. For each cloning, we first select the `player1` icon in the *Created Objects Palette*, and then choose *Clone…* from the *Create* menu. This causes the *Create Cutout* dialog box to appear, with all attribute settings matching those of `player1`. In each case, we change the name (to "player2", "player3", and "player4", respectively), and the data (to "250", "200", and "150", respectively), while leaving all other attributes alone. After dismissing the *Create Cutout* dialog box each time, a new create statement appears in the script—for example,

```
create cutout player2 as clone of player1 –data "250"
```

In addition, the new cutout appears in the *Created Objects Palette*, and the execution arrow advances to the next line. We proceed to create the football, referee and goal post in the same way.

3.3.2 Placing the Storyboard Elements

The next step is to place the cutouts in the storyboard. In order to lay the players out in a row, we use a *grid* s-struct, which we create by choosing *Grid…*from the *Create* menu and then filling in the *Create Grid* dialog box (see

Figure 5b). We name the grid `a`, and give it one row and four columns, with a cell height and width corresponding to the height and width of `player1`. We accept all other default attributes, and click on the "Create" button to create the grid.

We now place all of the objects we have created into the storyboard. First, we drag and drop the grid to an acceptable location in the middle of the storyboard; the corresponding `place` statement appears in the script, with the execution arrow advancing to the next line. With the grid in place, it is straightforward to position the players at the grid points:

```
place player1 at position 1 of a
place player2 at position 2 of a
place player3 at position 3 of a
place player4 at position 4 of a
```

Finally, we drag and drop the referee to a reasonable place below the row of football players; we drag and drop the football to the left center position of the referee; and we drag and drop the goal to a position to the right of the line of football players. Figure 3a shows the ALVIS environment after all `place` statements have been executed.

3.3.3 Programming the Spatial Logic

We are now set to do the SALSA programming necessary to animate the storyboard. ALVIS users may program much of this code through a combination of dialog box fill-in and direct manipulation. However, in the presentation that follows, we will focus on the SALSA code itself in order to demonstrate the expressiveness of the language.

As each player reaches his rightful place in the line, we want to turn his outline color to green. Since we have set the outline color of the goal post (the right-most cutout in the storyboard) to green, we know that when the outline color of the cutout immediately to the right of the ball carrier is green, the ball carrier has reached the end of the line, and we are done with a pass of bubble sort's inner loop. In SALSA, we may formulate this as a `while` loop:

```
while outlinecolor of cutout right-of cutout touching
   football is-not green
      --do body of inner loop of bubblesort (see below)
endwhile
```

We know that we are done with all passes of the outer loop when all but the first player in line has a green outline color. In SALSA, we may express this logic as another `while` loop:

```
while outlinecolor of cutout at position 2 of a is-not green
     --do body of outer loop of bubblesort (see below)
endwhile
```

Now we come to the trickiest part of the script: the logic of the inner loop. We want to successively compare the ball carrier to the player to his immediate right. If these two players are out of order, we want to swap them, with the ball carrier maintaining the ball; otherwise, we want the ball carrier to fumble the ball to the player to his immediate right.

The following SALSA code makes the comparison, either fumbling or swapping, depending on the outcome:

```
if data of cutout touching football > data of cutout right-of
                      cutout touching football --swap players
  assign p1 to position in a of cutout touching football
  assign p2 to position in a of cutout right-of cutout
    touching football
  doconcurrent
    move cutout touching football to p2
    move cutout right-of cutout touching football to p1
    move football right cellwidth of a
  enddoconcurrent
else --fumble ball to next player in line
  move football right cellwidth of a
endif
```

All that remains is to piece together the outer loop. At the beginning of the outer loop, we want to toss the ball to the first player in line. We then want to proceed with the inner loop, at the end of which we first set the outline of the player with the ball to green (signifying that he has reached his rightful place), and then toss the ball back to the referee. Thus, the outer loop appears as follows:

```
move football to left-center of cutout at position 1 of a
--inner loop (see above) goes here
set outlinecolor of cutout touching football to green
move football to top-center of referee
```

Figure 6 summarizes the example by presenting the complete script.

3.3.4 Presenting the Storyboard

Using ALVIS's presentation interface, we may now present our "football" storyboard to an audience for feedback and discussion. Since nothing interesting occurs in the script until after the *Storyboard View* is populated, we elect to execute the script to the first line past the last `place` command (line 19 in Figure 6) by positioning the cursor on that line and choosing “Run to Cursor” from the *Present* menu. The first time through the algorithm’s outer loop, we walk through the script slowly, using the “Presentation Pointer” tool to point at the storyboard as we explain the algorithm’s logic. Before the two players are swapped (line 26 in Figure 6), we use the “Markup Pen” tool to circle the two elements that are about to be swapped.

Now suppose an audience member wonders whether it might be useful to flash the players as they are being compared (line 23 in Figure 6). Thanks to ALVIS’s flexible animation interface and dynamic modification abilities, we are able to test out this idea on the spot. As we attempt to edit code, which lies within the script’s inner `while` loop, ALVIS recognizes that a change at this point will necessitate a re-parsing of that entire loop. As a result, ALVIS backs up the script to the point just before the loop begins (line 18 in Figure 6). At that point, we insert the following four lines of code:

```
doconcurrent --flash players to be compared
  flash cutout touching football for 1 sec
  flash cutout right-of cutout touching football for 1 sec
enddoconcurrent
```

We can now proceed with our presentation without having to recompile the script, and without even having to start the script over from the beginning.

4. Juxtaposition with Related Work

SALSA’s verbose, natural language syntax was inspired by Atkinson and Winkler’s HyperTalk language [28], which was designed to empower novice programmers to develop their own sophisticated applications within Apple’s HyperCard environment. ALVIS was inspired by, and has key features in common with, a family of systems geared toward rapidly prototyping graphical user interfaces. QUICK [29] supports a direct manipulation interface for programming graphical applications (which may include animation) in terms of an underlying interpreted language with explicit support for spatial relations. SILK [23] and PatchWork [27] are sketch-based systems that support the rapid construction of low-fidelity interface prototypes; the latter’s notion of “patches” is similar to SALSA’s concept of “cutouts.”

Beginning with Brown's BALSA system [1], a legacy of interactive AV systems have been developed to help teach and learn algorithms, e.g., [2-5, 30]. SALSA and ALVIS differ from these systems in four fundamental ways, which we discuss in the remainder of this section.

4.1 Empirically-Based Design

The first key difference between SALSA and ALVIS and past AV systems is their grounding in a theoretically- and empirically-driven design process. Indeed, whereas the design of past systems has been based mainly on designer intuitions (but see [31]), we have derived the low fidelity approach supported by SALSA and ALVIS from a comprehensive review of experimental studies [16], along with an established learning theory [17]. Moreover, we have made an earnest effort to root the design of SALSA and ALVIS firmly in empirical data. Specifically, as part of a user-centered design process, we have used observations from several empirical studies as a basis for formulating the functional and usability requirements of the language and system. For example, before we began implementation, we verified and refined the syntax and semantics of SALSA through a pilot study involving students enrolled in a junior-level undergraduate algorithms course. Likewise, our observations of how students construct storyboards out of simple art supplies clearly inspired ALVIS's conceptual model, which is rooted in "paper-and-pencil" storyboard construction.

4.2 Spatially-driven Visualization Construction

A second key difference between ALVIS and SALSA and extant AV technology is the visualization construction technique they pioneer. To place ALVIS and SALSA into perspective, Figure 7 presents a taxonomy of AV construction techniques. The taxonomy makes a top-level distinction between those AV systems that require visualizations to be defined in terms of an underlying virtual machine, and those systems that support hand-crafted, "one-shot" visualizations whose execution lacks an underlying virtual machine. With most extant AV systems, one defines a visualization in some sort of programming language that executes on an underlying virtual machine. However, a few construction techniques abandon a formal underlying execution model altogether. With these techniques, there exists no chance of constructing AVs that work for general input; AV execution is entirely under human control. Examples of construction techniques that lack an underlying virtual machine include *storyboarding*

[26], which inspired ALVIS and SALSA; *morphing* [32]; and *drawing and animation editors*—most notably, Brown and Vander Zanden's [33] specialized drawing editor with built-in semantics for data structure diagrams.

Visualization construction techniques that rely on an underlying virtual machine may be further classified according to the way in which they map a visualization to the underlying virtual machine. *Algorithmically-driven* construction involves the specification of mappings between an implemented algorithm and a visualization. By executing the algorithm, one generates the visualization as a byproduct. Variations on algorithmic AV construction include (a) *predefined* techniques, in which one chooses when and what to view, but has no choice with respect to the form of the visualization (see, e.g., [3, 34]); (b) *annotative* techniques, in which one annotates an algorithm with event markers that give rise to updates in the visualization (see, e.g., [1, 2, 5, 35]); (c) *declarative* techniques, in which one specifies a set of rules for mapping an executing program to a visual representation (see, e.g., [4, 30]); and (d) *manipulative* techniques, in which uses some form of direct manipulation and dialog box fill-in to map an algorithm to a visual representation (see, e.g., [31, 36, 37]).

In stark contrast to algorithmically-driven techniques, *spatially-driven* techniques completely abandon a virtual machine rooted in an underlying algorithm to be visualized. Instead, spatially-driven techniques define a spatial analogy to the algorithm to be visualized; that is, they are grounded in a virtual machine driven by the spatiality of the visualization itself. SALSA and ALVIS support a more formal version of storyboarding with an underlying spatially-driven virtual machine. In particular, the SALSA language pioneers a *procedural* approach to spatially-driven construction, while ALVIS supports a *manipulative* approach by enabling many components of SALSA programs to be programmed by direct-manipulation. In a similar vein, Michail's Opsis [38] supports a *state-based* approach in which binary tree algorithms may be constructed by manipulating abstract visual representations, while Brown and Vander Zanden [33] describe a *rule-based* approach in which users construct visualizations with graphical rewrite rules.

4.3 Socially-negotiated Correctness

A third key difference between SALSA and ALVIS and existing AV systems follows directly from the difference in visualization construction technique just noted. As just discussed, algorithmically-driven visualizations are directly generated as a byproduct of executing an implemented algorithm. To the extent that the mappings underlying an algorithmically-driven visualization are correctly defined, an algorithmically-driven visualization can be guaranteed to match the behavior of the underlying algorithm. Because of this guarantee, an algorithmically-driven visualization essentially serves as a "knowledge conveyor" by portraying the correct procedural behavior of an algorithm under study. It is a teacher who is always right.

In stark contrast, because a SALSA visualization is defined purely in terms of its own spatiality, there is no guarantee that it corresponds to the algorithm it is intended to depict. As a consequence, rather than serving as a "knowledge conveyor," low fidelity AV technology plays the dual role of "knowledge constructor" and "conversation mediator." In its role as "knowledge constructor," low fidelity AV technology enables students to build their own understandings of a target algorithm through experimenting with and constructing spatial analogies. In its complementary role as conversation mediator, AV technology facilitates instructor-learner conversations in which the meaning and significance of the target algorithm, including its correctness, are socially negotiated.

While AV technology's inability to guarantee correctness might concern educators committed to accurately teaching how algorithms work, we regard a SALSA visualization's lack of guaranteed algorithmic correspondence as a key pedagogical feature of the low fidelity approach. This is because it creates a situation in which students and instructors must *socially negotiate* an algorithm's significance (including its correctness), and we believe that the ability to participate competently in such discussions about algorithms is an important skill for students to learn in an algorithms course.

4.4 Explicit support for Visualization Presentation

A fourth key difference between SALSA and ALVIS and existing AV technology lies in their support for visualization presentation. Most existing AV technology has adopted Brown’s [1] animation playback interface. By enabling one to start and pause an animation, step through an animation, and adjust animation speed, this interface essentially treats animations as uni-directional videotapes. A notable exception is LEONARDO [30], which defines a custom C execution environment that, by supporting reverse execution, is especially well suited to debugging. As we have seen, in order to support interactive presentations, ALVIS’s interface goes beyond the standard animation playback interface by supporting reverse execution, and dynamic markup and modification. Notably, in most existing AV systems, modifying an animation entails changing and recompiling source code, which is seldom feasible within the scope of an interactive presentation.

5. Summary and Future Work

A review of past experiments designed to substantiate AV technology's pedagogical benefits suggests that the more actively learners are involved with AV technology, the better they learn the target algorithms. Motivated by this observation, we have used ethnographic field techniques to explore a novel approach to using AV technology as a learning aid: one in which students become end-user programmers and discussants by constructing and presenting their own visualizations, rather than viewing visualizations constructed by their instructor. As we have seen, our ethnographic study findings point out a key distinction between high and low fidelity algorithm visualizations, and motivate a shift from high fidelity to low fidelity AV technology as the basis for exercises in which students construct and present their own visualizations.

To explore the design space of low fidelity AV technology, we have presented a prototype end-user language (SALSA) and system (ALVIS) rooted in our studies of how students construct and present low fidelity visualizations made out of simple art supplies. The SALSA language pioneers a novel spatial approach to specifying visualizations. In this approach, one performs the visualization's layout, and specifies the visualization's inner logic, in terms of the *spatial relations* among the objects of the visualization; there exists no notion of an underlying "driver algorithm." Similarly, the ALVIS environment pioneers a novel interface for visualization presentation—one that supports reverse execution, as well as dynamic mark-up and modification.

Aside from illustrating the promise of our novel, low fidelity approach to integrating AV technology into an undergraduate algorithms course, the work presented in this article opens up many directions for future research. As we continue to explore this approach, three directions are of particular interest to us.

First, while the low fidelity approach appeared promising in our ethnographic studies, experimental studies must be carried out to validate the educational effectiveness of the approach. To that end, we have already conducted a preliminary experiment that tested the educational impact of the self-construction factor [39]. Although the trends were favorable, this experiment found no statistically significant differences between students who actively viewed (by designing their input data sets) a visualization constructed by an expert, and students who constructed their own visualizations out of art supplies. However, this experiment focused on just one aspect of the low fidelity approach. Clearly, further experiments are needed in order to better understand its overall educational impact. For example, a key pedagogical feature of the low fidelity approach is that it creates a situation in which experts and learners can communicate about algorithms. Thus, another major factor we posit to influence learning is *instructor communication*: whether or not students have the opportunity to discuss, with an instructor, the visualizations that they construct or interact with. In future work, we would like to carry out further experimental studies that systematically explore the instructor communication factor in concert with self-construction.

Second, ALVIS and SALSA are clearly works in progress. Much future work remains to be done in order both to refine the design features that they demonstrate, and to explore their benefits in the real world. In future research, we would like to focus our efforts in four key areas:

First, while SALSA does enable programmers to specify visualizations in the spatial domain, it is clearly a non-trivial programming language. Indeed, SALSA programmers are susceptible to the same kinds of programming errors that can be made in non-spatial languages. Prior to implementing SALSA, we conducted preliminary paper-based studies of SALSA's design with undergraduate algorithms students. In future research, we need to subject SALSA and ALVIS to iterative usability testing in order to verify that undergraduates can easily create the kinds of visualizations they want to create, and in order to improve the usability of their design. Moreover, our testing will need to explore whether students can use SALSA and ALVIS's create visualizations of larger, more complex algorithms. To date, we have used SALSA and ALVIS only to develop small-scale visualizations of simple algorithms.

Second, we originally designed ALVIS with the intention of using it with a large electronic whiteboard (e.g., the "Smart Board," http://www.smarttech.com), so that groups of students and instructors could engage in collaborative algorithm design. In such a setting, ALVIS would need to be used with some sort of stylus (pen) as its input device. In future research, we would like to explore the potential for ALVIS to be used in this way.

Third, SALSA and ALVIS are presently frail research prototypes; they were not implemented to be robust systems suitable for use in the real world. Thus, an important direction for future research is to improve both the completeness and the robustness SALSA and ALVIS, so that they may ultimately be used as the basis for assignments in an actual algorithms course.

Finally, once ALVIS and SALSA become stable enough to be used in a real algorithms course, we believe that they could serve as a foundation for our vision of the algorithms course of the future. Our vision draws from the instructional model presently used to teach architectural design, which constitutes a prime example of the learning-by-participation model advocated by constructivism [17]. Unlike algorithms students, who spend most of their time attending lectures and studying on their own, architectural students spend most of their time with their peers in an architectural studio, where they work on collaborative, directed design projects. Gathered around drafting tables, student groups work out their ideas using drafting paper, pencils, and cardboard models. During scheduled review sessions, students present their work-in-progress to their instructor and peers for feedback.

So, too, could it be in an algorithms course. While lectures would likely remain a necessary part of the curriculum, the length or the frequency of the lectures could be reduced in order to give students more time in the "algorithms studio." There, they would engage in collaborative algorithm design and analysis projects at their "drafting tables"—electronic whiteboards running a conceptual algorithm design tool like ALVIS. At these drafting tables, students could explore alternative ways of solving assigned algorithm design problems. In addition, students could use ALVIS to present their solutions to their peers and instructor—both informally during studio work sessions, and more formally during scheduled review sessions. Such review sessions would not only provide instructors with an important basis for assessing students' progress, but they would also, as we found in our ethnographic fieldwork, set up ideal conditions under which to discuss the conceptual foundations of algorithms.

6. Acknowledgments

This research was conducted as part of the first author's dissertation [18], supervised by the second author. The first author is grateful for the second author's inspiration and astute guidance on the project. SALSA and ALVIS were programmed in Microsoft® Visual C++™ using the Microsoft® Foundation Classes. We used the Cygwin port of Gnu Flex and Gnu Bison to generate the SALSA interpreter. Hank Bennett programmed ALVIS's back-end animation engine, and part of ALVIS's front-end user interface. Ted Kirkpatrick assisted greatly in debugging ALVIS. We are grateful to our reviewers for many constructive comments that helped us to improve the focus and clarity of this article.

7. References

1. M. H. Brown (1988) *Algorithm animation* The MIT Press, Cambridge, MA.
2. J. T. Stasko (1990) TANGO: A framework and system for algorithm animation. *IEEE Computer* **23**, 27-39.
3. T. Naps (1990) Algorithm visualization in computer science laboratories. In: *Proceedings of the 21st SIGCSE Technical Symposium on Computer Science Education* ACM Press, New York, pp. 105-110.

4. G. C. Roman, K. C. Cox, C. D. Wilcox, & J. Y. Plun (1992) Pavane: A system for declarative visualization of concurrent computations. *Journal of visual languages and computing* **3**, 161−193.

5. J. T. Stasko (1997) Using student-built animations as learning aids. In: *Proceedings of the ACM Technical Symposium on Computer Science Education* ACM Press, New York, pp. 25-29.

6. M. H. Brown & R. Sedgewick (1984) Progress report: Brown University Instructional Computing Laboratory. *ACM SIGCSE Bulletin* **16**, 91-101.

7. M. D. Byrne, R. Catrambone, & J. T. Stasko (1999) Evaluating animations as student aids in learning computer algorithms. *Computers & Education* **33**, 253-278.

8. J. Stasko, A. Badre, & C. Lewis (1993) Do Algorithm Animations Assist Learning? An Empirical Study and Analysis. In: *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems* ACM Press, New York, pp. 61-66.

9. A. W. Lawrence (1993) Empirical studies of the value of algorithm animation in algorithm understanding. Unpublished Ph.D. dissertation, Department of Computer Science, Georgia Institute of Technology.

10. C. Kann, R. W. Lindeman, & R. Heller (1997) Integrating algorithm animation into a learning environment. *Computers & Education* **28**, 223-228.

11. S. R. Hansen, N. H. Narayanan, & D. Schrimpsher (2000) Helping learners visualize and comprehend algorithms. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning* **1**.

12. M. E. Crosby & J. Stelovsky (1995) From multimedia instruction to multimedia evaluation. *Journal of Educational Multimedia and Hypermedia* **4**, 147-162.

13. D. J. Jarc, M. B. Feldman, & R. S. Heller (2000) Assessing the benefits of interactive prediction using web-based algorithm animation courseware. In: *Proceedings SIGCSE 2000* ACM Press, New York, pp. 377-381.

14. J. S. Gurka (1996) Pedagogic Aspects of Algorithm Animation. Unpublished Ph.D. Dissertation, Computer Science, University of Colorado.

15. S. Papert (1980) *Mindstorms: Children, Computers, and Powerful Ideas* Basic Books, New York.

16. C. D. Hundhausen, S. A. Douglas, & J. T. Stasko (In press) A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*.

17. J. Lave & E. Wenger (1991) *Situated Learning: Legitimate Peripheral Participation* Cambridge University Press, New York, 138 pp.

18. C. D. Hundhausen (1999) 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.

19. S. A. Douglas, C. D. Hundhausen, & D. McKeown (1995) Toward empirically-based software visualization languages. In: *Proceedings of the 11th IEEE Symposium on Visual Languages* IEEE Computer Society Press, Los Alamitos, CA, pp. 342-349.

20. S. A. Douglas, C. D. Hundhausen, & D. McKeown (1996) Exploring human visualization of computer algorithms. In: *Proceedings 1996 Graphics Interface Conference* Canadian Graphics Society, Toronto, CA, pp. 9-16.

21. T. H. Cormen, C. E. Leiserson, & R. L. Rivest (1990) *Introduction to Algorithms* The MIT Press, Cambridge, MA.

22. H. Wolcott (1999) *Ethnography: A Way of Seeing* AltaMira Press, Walnut Creek, CA.

23. J. A. Landay & B. A. Myers (1995) Interactive sketching for the early stages of user interface design. In: *Proceedings of the ACM CHI '95 Conference on Human Factors in Computing Systems* ACM Press, New York, pp. 43-50.

24. J. Lin, M. Newman, J. Hong, & J. Landay (2000) DENIM: Finding a tighter fit between tools and practice for web site design. In: *CHI 2000 Conference Proceedings* ACM Press, New York, pp. 510-517.

25. R. Sedgewick (1988) *Algorithms* Addison-Wesley, Reading, MA.

26. S. A. Douglas (1995) Conversation analysis and human-computer interaction design. In: *Social and interactional dimensions of human-computer interfaces* (P. Thomas, ed.) Cambridge University Press, Cambridge.

27. M. van de Kant, S. Wilson, M. Bekker, H. Johnson, & P. Johnson (1998) PatchWork: A software tool for early design. In: *Human Factors in Computing Systems: CHI 98 Summary* ACM Press, New York, pp. 221-222.

28. (1988) *HyperTalk Language Reference Manual* Addison-Wesley, Menlo Park, CA.

29. S. A. Douglas, E. Doerry, & D. Novick (1992) QUICK: A tool for graphical user interface construction. *The Visual Computer* **8**, 117-133.

30. P. Crescenzi, C. Demetrescu, I. Finocchi, & R. Petreschi (2000) Reversible execution and visualization of programs with LEONARDO. *Journal of Visual Languages and Computing* **11**, 125-150.

31. S. Mukherjea & J. T. Stasko (1994) Toward visual debugging: Integrating algorithm animation capabilities within a source-level debugger. *ACM Transactions on Computer-Human Interaction* **1**, 215-244.

32. W. Citrin & J. Gurka (1996) A low-overhead technique for dynamic blackboarding using morphing technology. *Computers & Education*, 189-196.

33. D. R. Brown & B. Vander Zanden (1998) The Whiteboard environment: An electronic sketchpad for data structure design and algorithm description. In: *Proceedings of the 14th IEEE Symposium on Visual Languages* IEEE Computer Society Press, Los Alamitos, CA, pp. 1-8.

34. B. A. Myers (1983) Incense: A system for displaying data structures. *Computer Graphics* **17**, 115-125.

35. R. A. Duisberg (1987) Animation Using Temporal Constraints: An Overview of the Animus System. *Human-Computer Interaction* **3**, 275-307.

36. J. T. Stasko (1991) Using Direct Manipulation to Build Algorithm Animations by Demonstration. In: *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems* ACM Press, New York, pp. 307-314.

37. R. Duisberg (1987) Visual programming of program visualizations. In: *Proceedings of the IEEE 1987 Visual Language Workshop* IEEE Computer Society Press, Los Alamitos, CA.

38. A. Michail (1996) Teaching binary tree algorithms through visual programming. In: *Proceedings of the 12thIEEE Symposium on Visual Languages* IEEE Computer Society Press, Los Alamitos, CA, pp. 38-45.

39. C. D. Hundhausen & S. A. Douglas (2000) Using visualizations to learn algorithms: Should students construct their own, or view an expert's? In: *Proceedings 2000 IEEE International Symposium on Visual Languages* IEEE Computer Society Press, Los Alamitos, pp. 21-28.

Figures

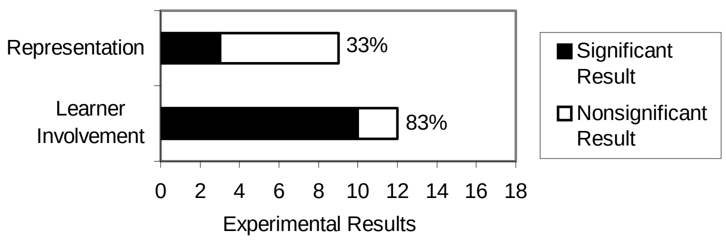


Figure 1. Results of AV effectiveness experiments broadly classified by their independent variables

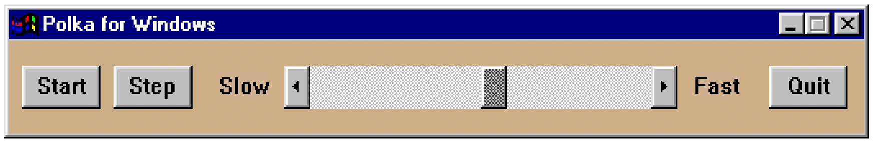
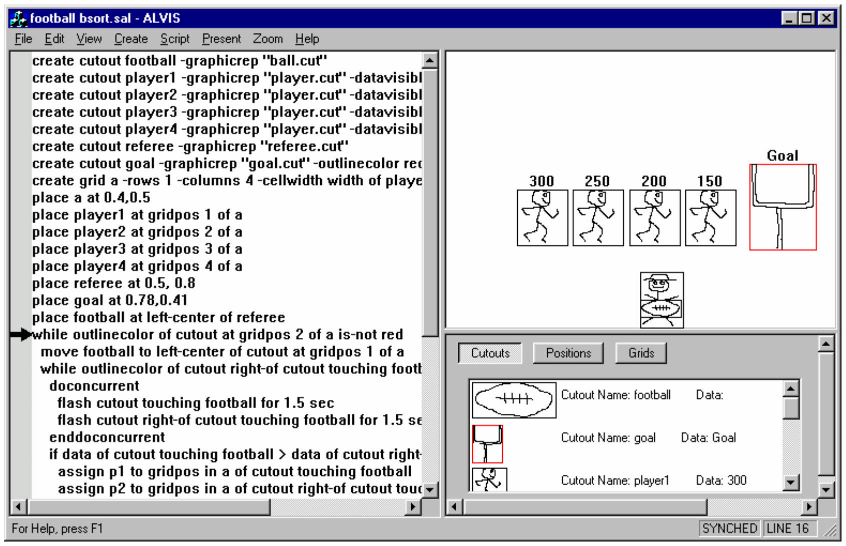
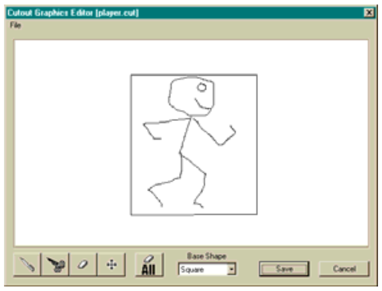


Figure 2. Samba’s “tape recorder” interface

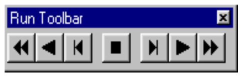


```
create cutout football -graphicrep "ball.cut"
create cutout player1 -graphicrep "player.cut" -datavisibl
create cutout player2 -graphicrep "player.cut" -datavisibl
create cutout player3 -graphicrep "player.cut" -datavisibl
create cutout player4 -graphicrep "player.cut" -datavisibl
create cutout referee -graphicrep "referee.cut"
create cutout goal -graphicrep "goal.cut" -outlinecolor rec
create grid a -rows 1 -columns 4 -cellwidth width of playe
place a at 0.4,0.5
place player1 at gridpos 1 of a
place player2 at gridpos 2 of a
place player3 at gridpos 3 of a
place player4 at gridpos 4 of a
place referee at 0.5, 0.8
place goal at 0.78,0.41
place football at left-center of referee
while outlinecolor of cutout at gridpos 2 of a is-not red
 move football to left-center of cutout at gridpos 1 of a
 while outlinecolor of cutout right-of cutout touching footb
  doconcurrent
   flash cutout touching football for 1.5 sec
   flash cutout right-of cutout touching football for 1.5 se
  enddoconcurrent
  if data of cutout touching football > data of cutout right-
   assign p1 to gridpos in a of cutout touching football
   assign p2 to gridpos in a of cutout right-of cutout touc
```

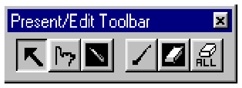
(a) Snapshot of a session with ALVIS



(b) Cutout graphics editor



(c) Execution control interface



(d) Presentation interface

Figure 3. The ALVIS interactive environment

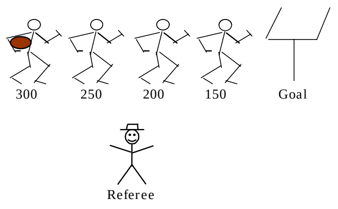
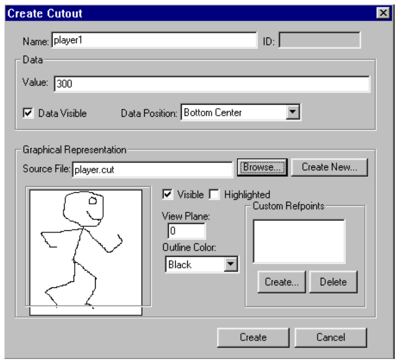
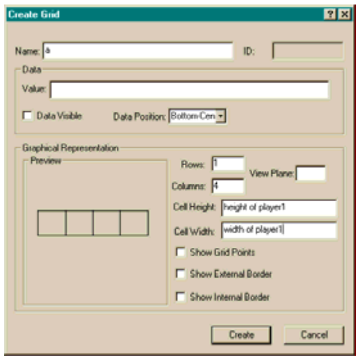


Figure 4. The “Football Bubble Sort” storyboard



Name: player1
ID:
Value: 300
[✓ Data Visible]
Data Position: Bottom Center
Source File: player.cut
[✓ Visible] Highlighted
View Plane: 0
Outline Color: Black

(a) *Create Cutout* dialog box



Name: a
ID:
Value:
Data Visible
Data Position: Bottom-Cen
Rows: 1
Columns: 4
View Plane:
Cell Height: height of player1
Cell Width: width of player1
Show Grid Points
Show External Border
Show Internal Border

(b) *Create Grid* dialog box

Figure 5. Dialog boxes for creating SALSA objects

```
--create the storyboard elements
create cutout player1 –graphicrep “cutout.cut” –data “300”
create cutout player2 as clone of player1 –data “250”
create cutout player3 as clone of player1 –data “200”
create cutout player4 as clone of player1 –data “150”
create cutout referee –graphicrep “referee.cut” –data “referee”
create cutout football –graphicrep “football.cut”
create cutout goal –graphicrep “goal.cut” –outlinecolor green
create grid a –rows 1 columns 4 –cellwidth width of player1
–cellheight height of player1
--place the storyboard elements
place a at 0.5,0.5
place player1 at position 1 of a
place player2 at position 2 of a
place player3 at position 3 of a
place player4 at position 4 of a
place referee at
place football at top-center of referee
--ready to perform bubble sort. . .
while outlinecolor of cutout at position 2 of a is-not green --do outer loop
  move football to left-center of cutout at position 1 of a
  while outlinecolor of cutout right-of cutout touching football is-not green
    --do inner loop; swap or fumble
    if data of cutout touching football >
        data of cutout right-of cutout touching football
      --swap players
      assign p1 to position in a of cutout touching football
      assign p2 to position in a of cutout right-of cutout touching football
      doconcurrent
        move cutout touching football to p2
        move cutout right-of cutout touching football to p1
      move football right cellwidth of a
      enddoconcurrent
    else --fumble ball to next player in line
      move football right cellwidth of a
    endif
  endwhile  --inner loop
  --ball carrier has reached rightful place:
  set outlinecolor of cutout touching football to green
  --toss ball back to referee, in preparation for next pass of outer loop
  move football to top-center of referee
endwhile --outer loop
set outlinecolor of cutout at position 1 of a to green
```

Figure 6. SALSA Script for “Football Bubble Sort” storyboard

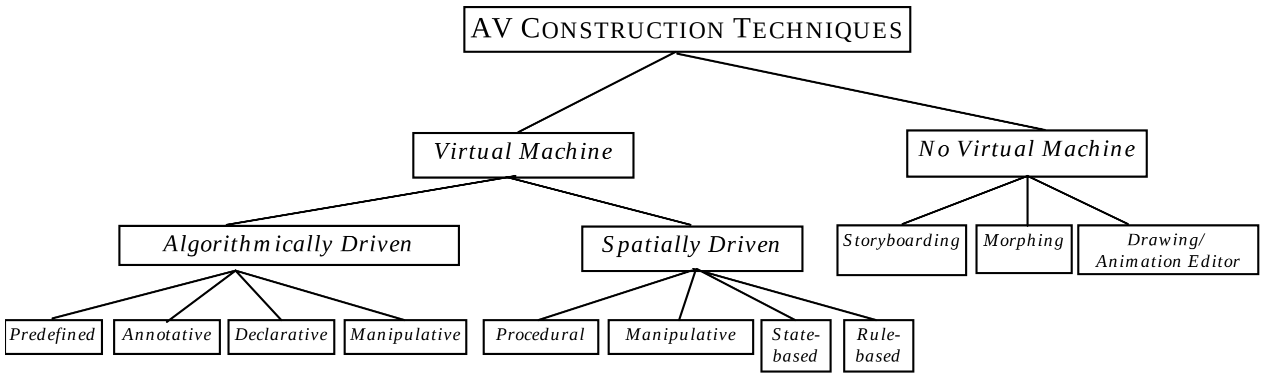


Figure 7. A taxonomy of AV construction techniques

List of Figure Captions

Figure 1. Results of AV effectiveness experiments broadly classified by their independent variables

Figure 2**.** Samba's "tape recorder" interface

Figure 3. The ALVIS interactive environment

(a) Snapshot of a session with ALVIS

(b) Cutout graphics editor

(c) Execution control interface

(d) Presentation interface

Figure 4. The "Football Bubble Sort" storyboard

Figure 5. Dialog boxes for creating SALSA objects

(a) *Create Cutout* dialog box

(b) *Create Grid* dialog box

Figure 6. SALSA Script for "Football Bubble Sort" storyboard

Figure 7. A taxonomy of AV construction techniques