A Language and System for Constructing and Presenting Low Fidelity Algorithm Visualizations

Christopher Hundhausen[1] and Sarah Douglas[2]

[1]Laboratory for Interactive Learning Technologies, Department of Information and Computer Sciences, University of Hawai'I, Honolulu, HI 96822
hundhaus@hawaii.edu
[2]Human-Computer Interaction Laboratory, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403–1202
douglas@cs.uoregon.edu

Abstract. Computer science educators have traditionally used algorithm visualization (AV) software to create graphical representations of algorithms that are later used 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 the findings of ethnographic studies we conducted in a junior-level algorithms course, we motivate the use of low fidelity AV technology as the basis for an alternative learning paradigm in which students construct and present their own visualizations. To explore the design space of low fidelity AV technology, we present a prototype language and system derived from empirical studies in which students constructed and presented visualizations made out of simple art supplies. Our prototype language and system pioneer a novel technique for programming visualizations based on spatial relations, and a novel presentation interface that supports reverse execution and dynamic mark-up and modification

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]. More recently, computer science educators have advocated using AV software as the basis for "visualization assignments," in which students construct their own visualizations of the algorithms under study [5].

Inspired by social constructivist learning theory [7], we have explored a teaching approach that takes "visualization assignments" one step further by having students *present* their own visualizations to their instructor and peers for feedback and discussion [8]. To better understand this approach, we conducted a series of

ethnographic studies in a junior-level algorithms course that included such 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 this teaching approach.

In this paper, we use the findings of our ethnographic studies to motivate the need for a new breed of AV technology that supports the construction and presentation of *low fidelity* visualizations, which are capable of illustrating a target algorithm for a few, carefully chosen input data sets, and which have an unpolished, sketched appearance. Drawing on the findings of our ethnographic studies and prior empirical studies, we then derive a set of specific requirements for low fidelity AV technology.

To demonstrate what low fidelity AV technology might look like, we next present SALSA, an interpreted, high-level language for constructing low fidelity AVs, along with ALVIS, an interactive, direct manipulation environment for programming in SALSA. In manifesting the design implications of our empirical research, SALSA and ALVIS pioneer a novel *spatial* approach to algorithm visualization construction, as well as a novel visualization presentation interface that supports reverse execution, and dynamic mark-up and modification. We conclude by presenting a taxonomy that places SALSA and ALVIS into the context of past work, and by describing some directions for future research.

2 Motivation

The redesign of AV software advocated in this paper was motivated by a pair of ethnographic field studies we conducted in consecutive offerings of a junior-level algorithms course at the University of Oregon ([8], ch. 4). 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 the use of AV technology in these courses, we employed a variety of ethnographic field techniques, including participant observation, semi-structured interviews, videotape analysis, and artifact analysis.

2.1 Ethnographic Study I: High Fidelity

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 annotated the algorithms with procedure calls that, when invoked, generated and updated their visualizations using Samba routines. Finally, they engaged in an

iterative process of refining and debugging their visualizations. This process involved compiling and executing their algorithms, noting any problems in the resulting visualization, and modifying their C++ code to fix the problems.

In this first study, three key findings are noteworthy. First, students spent over 33 hours on average constructing and refining a single visualization. They spent most of that time steeped in *low-level graphics programming*—for example, laying out graphics objects in terms of Cartesian coordinates, and writing general-purpose graphics routines. 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: Low Fidelity

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 [9,10], we have called such low fidelity visualizations *storyboards*.)

In this second study, three key findings stand out. First, students spent only about six hours on average 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. 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.

2.3 Discussion: From High to Low Fidelity

Our study findings furnish three compelling 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. 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.

3 Deriving Design Requirements

The findings reported in the previous section motivate the development of a new breed of AV technology that supports the construction and presentation of low fidelity visualizations. 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 [9, 10], provide a solid empirical foundation for design.

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 storyboard, and locations in the storyboard, 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 [9, 10]:

- 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 [9, 10], 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 support an execution model based on spatial, rather than 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.*

4. Prototype 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:

1. *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.
2. *Position*. This data type represents an x,y location within a storyboard.
3. *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 1 presents a snapshot of the ALVIS environment, which consists of three main regions:

1. *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.
2. *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.
3. *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 [11]): scraps of virtual construction paper that may be cut out and drawn on, just like real construction paper. In ALVIS, users create storyboards by using a graphics editor (Figure 2a) 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

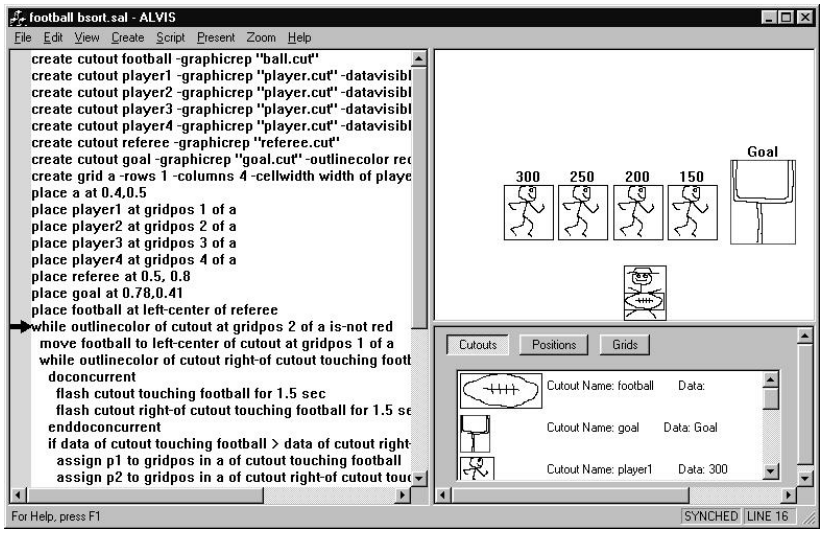
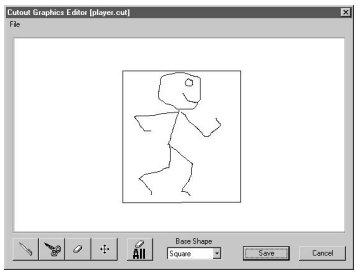


Fig. 1. A Snapshot of a session with ALVIS



(a) Cutout graphics editor

(b) Execution control interface

(c) Presentation interface

Fig. 2. Elements of the ALVIS interactive environment

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 2b), 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 2c) 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 2c) 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.

5 Example Use of the 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 Fig. 3) of the bubble sort algorithm we observed in prior empirical studies [9,10].

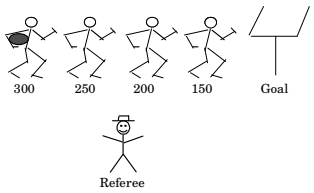


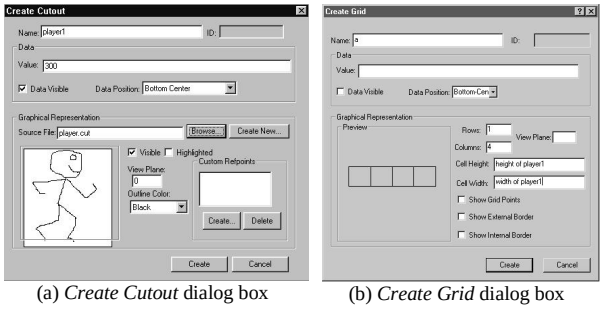
Fig. 3. The Football Bubblesort storyboard

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 below each player) 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. 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 end of the unsorted line of 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.

5.1 Creating the Storyboard Elements

We begin by creating the cutouts that appear in Figure 2: 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 4a). We name the cutout "player1," and use the *Cutout Graphics Editor* (Figure 2a) 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, we set the data attribute to "300," and decide to accept all other default attributes. 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"
```



(a) *Create Cutout* dialog box (b) *Create Grid* dialog box

Fig. 4. Dialog boxes for creating SALSA objects

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.

5.2 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 4b). 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 1a shows the ALVIS environment after all `place` statements have been executed.

5.3 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 while loop (see above) goes here
set outlinecolor of cutout touching football to green
move football to top-center of referee
```

5.4 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 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, 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. 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. 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.

7 Related work

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]. SALSA and ALVIS differ from these systems in two fundamental ways.

First, the visualization construction technique pioneered by ALVIS and SALSA differs from those supported by existing systems. To place ALVIS and SALSA into perspective, Figure 5 presents a taxonomy of AV construction techniques. These may be divided into three main categories:

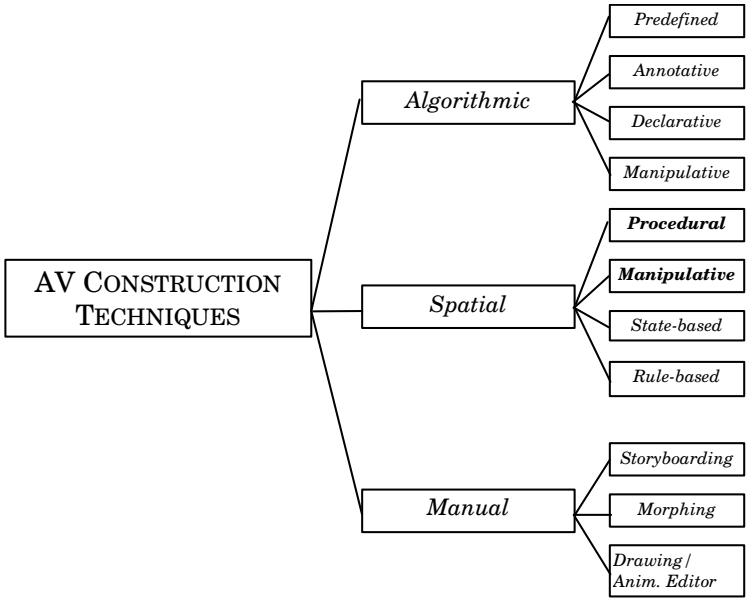


Fig. 5. A taxonomy of AV construction techniques

1. *Algorithmic*. Algorithmic construction involves the specification of mappings between an underlying (implemented) algorithm, and a visualization. Most existing AV systems support one of the four algorithmic techniques (see [12] for a review): *predefined, annotative declarative* or *manipulative*.
2. *Spatial*. In stark contrast to algorithmic techniques, spatial techniques completely abandon the use of an underlying "driver" algorithm that generates a visualization as a byproduct of its execution. Instead, spatial techniques specify a visualization in terms of the *spatiality* of the visualization itself. The SALSA language pioneers a *procedural* approach to spatial 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 [13] supports a *state-based* visual programming approach in which binary tree algorithms are constructed by manipulating of abstract visual representations, while Brown and Vander Zanden [14] describe a *rule-based* approach in which users construct visualizations with graphical rewrite rules.
3. *Manual*. In *manual* AV construction, one abandons a formal underlying execution model altogether. Thus, there exists no chance of constructing AVs the work for general input; AV execution is entirely under human control. Manual construction techniques include *storyboarding* [9] on which ALVIS and SALSA are based; *morphing* [15], and *drawing and animation editors*—most notably, Brown and Vander Zanden's [14] specialized drawing editor with built-in semantics for data structure diagrams.

The second key difference between SALSA and ALVIS and existing AV technology lies in their support for visualization presentation. Almost without exception, existing AV technology has adopted Brown's [1] animation playback interface, which allows one to start and pause an animation, step through the animation, and adjust animation speed. As we have seen, in order to support interactive presentations, ALVIS's animation control interface goes well beyond this 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.

8 Summary and future work

In this paper, we have used the findings of ethnographic studies of an undergraduate algorithms course to develop a key distinction between high and low fidelity algorithm visualizations, and to motivate a shift from high fidelity to low fidelity AV technology as the basis for assignments in which students construct and present their own visualizations. Based on empirical studies of how students construct and present low fidelity visualizations made from simple art supplies, we have derived a set of requirements for low fidelity AV technology. To explore the design space circumscribed by these requirements, we have presented SALSA and ALVIS, a prototype language and system for constructing and presenting low fidelity visualizations. SALSA and ALVIS introduce a novel spatial technique for visualization creation, as well as a novel interface for presenting visualizations to an audience.

Although they demonstrate the key features of low fidelity AV technology, SALSA and ALVIS are clearly works in progress. In future research, we would like to focus our efforts in three key areas. First, prior to implementing SALSA, we conducted paper-based studies of SALSA's design with undergraduate algorithms students. In future research, we need to subject ALVIS to iterative usability testing in order to verify and improve the usability of its design. Second, we originally designed ALVIS with the intention of using it with a large electronic whiteboard (e.g., the "SmartBoard," 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. Finally, 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. See http://lilt.ics.hawaii.edu/lilt/software/alvis/index.html for up-to-date information on our progress.

9 Acknowledgments

This research was conducted as part of the first author's dissertation [8], 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. Thanks to Hank Bennett for programming a significant portion of ALVIS, and to Ted Kirkpatrick for providing invaluable help with debugging ALVIS.

References

1. M. H. Brown, *Algorithm animation*. Cambridge, MA: The MIT Press, 1988.
2. J. Stasko, "Tango: A framework and system for algorithm animation." Ph.D. Dissertation (Tech. Rep. No. CS-89-30), Department of Computer Science, Brown University, Providence, RI, 1989.
3. T. Naps, "Algorithm visualization in computer science laboratories," in *Proceedings of the 21st SIGCSE Technical Symposium on Computer Science Education*. New York: ACM Press, 1990, pp. 105-110.
4. G. C. Roman, K. C. Cox, C. D. Wilcox, and J. Y. Plun, "Pavane: A system for declarative visualization of concurrent computations," *Journal of visual languages and computing*, vol. 3, pp. 161−193, 1992.
5. 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.
6. M. H. Brown and R. Sedgewick, "Progress report: Brown University Instructional Computing Laboratory," in *ACM SIGCSE Bulletin*, vol. 16, pp. 91-101, 1984.
7. J. Lave and E. Wenger, *Situated Learning: Legitimate Peripheral Participation*. New York: Cambridge U. Press, 1991.
8. C. D. Hundhausen, "Toward Effective Algorithm Visualization Artifacts: Designing for Participation and Communication in an Undergraduate Algorithms Course." Ph.D. Dissertation (Tech. Rep. No. CIS-99-07), Department of Computer and Info. Science, University of Oregon, Eugene, 1999. Available at http://lilt.ics.hawaii.edu/~hundhaus/dis/.
9. 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.
10. S. A. Douglas, C. D. Hundhausen, and D. McKeown, "Exploring human visualization of computer algorithms," in *Proceedings 1996 Graphics Interface Conference*. Toronto, CA: Canadian Graphics Society, 1996, pp. 9-16.
11. M. van de Kant, S. Wilson, M. Bekker, H. Johnson, and P. Johnson, "PatchWork: A software tool for early design," in *Human Factors in Computing Systems: CHI 98 Summary*. New York: ACM Press, 1998, pp. 221-222.
12. G. C. Roman and K. C. Cox, "A taxonomy of program visualization systems," *IEEE Computer*, vol. 26, pp. 11-24, 1993.
13. A. Michail, "Teaching binary tree algorithms through visual programming," in *Proceedings of the 12th IEEE Symposium on Visual Languages*. Los Alamitos, CA: IEEE Computer Society Press, 1996, pp. 38-45.
14. D. R. Brown and B. Vander Zanden, "The Whiteboard environment: An electronic sketchpad for data structure design and algorithm description," in *Proceedings of the 1998*

IEEE Symposium on Visual Languages. Los Alamitos, CA: IEEE Computer Society Press, 1998, pp. 288-295.

15. W. Citrin and J. Gurka, "A low-overhead technique for dynamic blackboarding using morphing technology," *Computers and Education*, pp. 189-196, 1996.