What You See Is What You Code: A "Live" Algorithm Development and Visualization Environment for Novice Learners

(Submitted October 9, 2005; revised February 3, 2006)

CHRISTOPHER D. HUNDHAUSEN & JONATHAN L. BROWN
Visualization and End user Programming Laboratory
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164-2752

Please address correspondence to

Christopher D. Hundhausen
School of Electrical Engineering and Computer Science
Washington State University
PO Box 642752
Pullman, WA 99164-2752
Phone: (509) 335-4590
Fax: (509) 335-3818
hundhaus@eecs.wsu.edu

Abstract

Pedagogical algorithm visualization systems produce graphical representations that aim to assist learners in understanding the dynamic behavior of computer algorithms. In order to foster active learning, computer science educators have developed algorithm visualization systems that empower learners to construct their own visualizations of algorithms under study. Notably, these systems support a similar development model in which coding an algorithm is temporally distinct from viewing and interacting with the resulting visualization. Given that they are known to have problems both with formulating syntactically correct code, and with understanding how code executes, novice learners would appear likely to benefit from a more "live" development model that narrows the gap between coding an algorithm and viewing its visualization. In order to explore this possibility, we have implemented "What You See Is What You Code," an algorithm development and visualization model geared toward novices first learning to program under the imperative paradigm. In the model, the line of algorithm code currently being edited is reevaluated on every edit, leading to immediate syntactic feedback, along with immediate semantic feedback in the form of an algorithm visualization. Analysis of usability and field studies involving introductory computer science students suggests that the immediacy of the model's feedback can help novices to quickly identify and correct programming errors, and ultimately to develop semantically correct code.

1. Introduction

Pedagogical algorithm visualization (AV) systems produce graphical representations that aim to assist learners in understanding the dynamic behavior of computer algorithms. A recent meta-study of 24 experimental studies of AV effectiveness [1] identified an important trend in these studies: the more actively learners were involved in activities involving AV technology, the better they performed.

Given this trend, a key focus of AV research has been to explore approaches and technology that foster active learning. Naps *et al.* [2] present a framework of five progressively active levels of learner engagement that have been considered by AV research:

1. Viewing a visualization (see, e.g., [3])
2. Responding to questions concerning a visualization (see, e.g., [4])
3. Changing a visualization (see, e.g., [5])
4. Constructing a visualization (see, e.g., [6])
5. Presenting a visualization for feedback and discussion (see, e.g., [7])

Several lines of recent AV research have investigated approaches and technology to facilitate level 4: *visualization construction*. A review of the existing AV systems that support learner-constructed visualizations [6, 8-12] of imperative algorithms reveals a notable similarity in these systems: they all support a "delayed feedback" AV development model, in which writing algorithm code and viewing the resulting visualization are temporally distinct activities. A shortcoming of the "delayed feedback" development model is that it prevents programmers from leveraging concrete visual feedback on their coding progress at *edit time*. Yet, novice programmers are known to have difficulties not only with formulating syntactically-correct code, but also with conceptualizing the execution effects of code [13], thus requiring them to perform more frequent *progressive evaluation* of their code during the programming process [14]. It would therefore appear that novices in particular would stand to benefit from such edit-time feedback. Indeed, such feedback could prevent *gulfs of evaluation* [15] in which progress is impaired by an inability to see the execution effects of code immediately.

This article explores the above possibility by presenting and empirically evaluating "What You See Is What You Code" (WYSIWYC), a "live" algorithm development and visualization model for introductory computer science students who are first learning to program. We have implemented this model in a new version of the ALVIS

software [16] called ALVIS LIVE!. In the WYSIWYC model, a learner develops an algorithm through a combination of typing into a program editor and directly manipulating program objects. On every edit, the line of algorithm code currently being written is reevaluated. The edit-by-edit reevaluation of a line of code leads to

- immediate feedback on the line's syntactic correctness,
- immediate suggestions for how to formulate syntactically correct code, and
- the immediate update of an accompanying visualization of the algorithm in an adjacent window.

Thus, in this "live" algorithm development and visualization model, the distinction between writing an algorithm and visualizing an algorithm is blurred; writing the code produces an automatic visualization of the algorithm as it is written.

We have gathered preliminary empirical evidence that this increased level of "liveness" benefits novice programmers. In a usability study involving introductory computer science students, the WYSIWYC development model showed great promise in helping participants both to identify and correct program errors quickly, and ultimately to understand the execution of the code they were writing. In a follow-up field study of ALVIS LIVE! within an introductory computer science course, we found that ALVIS LIVE! enabled students to develop semantically-correct algorithms with minimal intervention from a teaching assistant.

The remainder of this article explores the WYSIWYC model in greater detail. In Section 2, we motivate the model by presenting key results from a field study of novice coding and visualization construction activities. Section 3 demonstrates the WYSIWYC model by way of an example session with ALVIS LIVE!. Section 4 presents a usability and field evaluation of the ALVIS LIVE!. We compare the WYSIWYC model to related work in Section 5. Finally, section 6 summarizes our contributions and outlines directions for future research.

2. Motivation: Preliminary Field Study

Why might it be beneficial for a novice algorithm development and visualization environment to dynamically generate a visualization of code execution at edit-time? Our interest in exploring an AV development model that supports level 3 or 4 "liveness" was motivated by a field study of an introductory college computer science course. As in our prior studies of an advanced college algorithms course [7], this study focused on "studio-based" learning activities in which learners constructed their own visualizations of algorithms under study, and then presented those visualizations to their instructor and peers for feedback and discussion.

During two separate weeks, we observed four different laboratory sections of the Fall, 2004 offering of the introductory computer science course at Washington State University. As part of an "algorithms-first" introduction to programming at the beginning of the course, the 80 students in these four laboratory sections participated in a series of two, three-hour "studio experiences" designed to get them to "think and talk algorithmically." In each studio experience, student pairs were given a set of algorithm design problems, e.g.,

> "Design two alternative algorithms that first prompt the user to input an integer value n. Your algorithms should then create a list containing n random integers between 1 and 100. Finally, your algorithms should build a list in which the values in the original list are in reverse order."

Using a text editor, pairs were tasked with developing pseudocode solutions to these problems. In addition, to make the algorithms more concrete, they were asked to use simple art supplies to create homemade visualizations of their algorithms operating on sample input data. They presented their visualizations to the class at the end of each studio experience for feedback and discussion.

2.1 Field Techniques

In these studies, we collected data using a variety of field techniques. First and foremost, we performed participant observation as voluntary teaching assistants in the course. In this capacity, we both observed student pairs, and assisted them on request. Second, we videotaped both the coding activities of selected pairs of students, and all of the visualization presentation sessions. Third, we collected artifacts: all of the pseudocode and homemade visualizations developed by students. Fourth, we followed up on themes that emerged from our observations by conducting brief interviews with selected students in each lab. Finally, we administered to all students a written questionnaire that asked them to reflect on their subjective experiences in the lab.

2.2 Key Observations

In our prior studies of this approach within an advanced algorithms course, we found the construction and presentation of algorithms under study to be valuable learning activities, primarily because these activities succeeded in getting students to participate more centrally in the course [7]. In this study of novice learners, we fully expected to make similar observations. To our surprise, however, we discovered both markedly different

student reactions to the coding activities, and markedly different dynamics in the presentations. Below, we focus on some key observations that served to motivate our interest in a more immediate AV development model.

We observed that many students expressed frustration as they coded algorithmic solutions in the pseudocode language they had been taught. In addition to being unsure about correct pseudocode language syntax, students commonly complained about the lack of execution feedback provided by the text editor they were using. This lack of feedback led to a low level of confidence that their code was correct. As one student put it in an interview, "[I was] not very confident [that my algorithm was correct as written], because when you write it, it's hard to conceptualize in your mind what it does. But if you actually run it, it would be easier; you can see if it's right or wrong instantly."

As was the case for many students, this student concluded that most challenging part of the coding exercise was "to see if [the code] was right."

It was not surprising, then, that a key role played by the teaching assistants who oversaw these labs was that of "code checker": Students would frequently ask teaching assistants to step through their code for them and tell them whether it was correct. The minority of students who were lucky enough to enlist a teaching assistant for this purpose ultimately converged on reasonably correct code. However, a post hoc analysis of code written by a random sample of 50% of the students who participated in these studio labs suggests that students' algorithmic solutions still contained an average of 1.5 semantic errors (n = 44 algorithmic solutions consisting, on average, of 19.23 lines of code).

Given the lack of execution feedback, we were not surprised by students' error rates. We were surprised, however, by how much time students spent being "stuck" in a holding pattern. Indeed, we documented many cases in which students consulted lecture notes and code examples for several minutes at a time, all the while making little or no visible progress on the programming task at hand.

In fact, 80% of the student pairs whose editing sessions we videotaped (n = 10 pairs) experienced one or more "stuck" periods of at least 2.5 minutes during which they made no visible progress in coding their solutions. Further analysis of this sample reveals that students spent only 30% of their time actually coding solutions. The remainder of that time was spent conversing with their partner (31%), referring to code samples and lecture slides (10%),

talking to lab assistants (20%), or doing absolutely nothing (7%). In their effort to generate correct solutions, several of these students simply gave up until they could summon a teaching assistant for help.

2.3 Discussion

In this study, we aimed to collect baseline data on novice programming and visualization activities undertaken independently of computer-based environments. In so doing, we hoped to gain insights into how to design a computer-based environment to support novice algorithm development and presentation. As prior empirical studies of novice programming would have predicted, we found that introductory computer science students required the ability to execute their code in order to understand it and gain confidence in its correctness. We did not expect, however, to observe such a prevalence of "stuck" periods during which students failed to make any coding progress at all.

We suspect that if students had used one of the existing novice programming environments reviewed in Section 5.3, they would have committed fewer mistakes, and they would have made faster progress than they did. Yet, our observations did raise an intriguing research question: Could an environment that presented an *immediate* visualization of an algorithm (at edit-time) help novice learners to develop a mental model of program execution, allowing them to make coding progress more quickly than they could in existing novice programming environments, which delay such feedback? An interest in exploring this question spurred the development of the alternative editing model presented in the next section.

3. The WYSIWYC Model

Figure 1 presents an annotated snapshot of the ALVIS LIVE! environment, in which we have implemented the WYSIWYC model. Using a compact pseudocode-like language called SALSA (see Table 1), the user can directly type commands into the Script Window on the left. As the user types in a command, the user receives up-to-the-keystroke syntactic feedback in a tooltip that appears immediately below the line being typed in. In addition, once a line of code is syntactically valid, the graphical results of the command is immediately visible in the Animation Window on the right. Alternatively, the user can use the Toolbox Tools on the right to directly lay out and animate program objects (variables and arrays) in the Animation Window. Through such direct manipulation, SALSA code is dynamically inserted into the Script Window on the left.

The focal point of the environment is the green Execution Arrow, which marks the line of code that was most recently executed, and that is currently being edited. On every keystroke, that line of code is re-executed, and the graphics in the Animation Window are dynamically updated to reflect that execution.

In order to facilitate "live" editing, the execution arrow, which can also be moved around with the Execution Controls (see Table 2), follows the editing caret around in the Script Window. Thus, whenever the programmer moves the caret to a new line, the script is automatically executed (either forwards or backwards) to that point, and the visual representation of the program displayed in the Animation Window is updated accordingly. This "execution-follows-the-caret" behavior serves two key purposes:

(1) It ensures a valid graphical context (in the Animation Window) in which to program by direct manipulation using Toolbox Tools, and

(2) it provides, on an edit-by-edit basis, a dynamic visualization that gives feedback on the programmer's coding efforts.

3.1 A Sample Programming Session

To make the WYSIWYC model more concrete, we now step through a sample session in which we use ALVIS LIVE! to code the "Partition" algorithm. Part of the "Quicksort" algorithm, the "Partition" algorithm arranges the values of an array around a partition value such that all values less than or equal to the partition value are placed to the left of the partition value, and all values greater than the partition value are placed to the right of the partition value. In addition to illustrating the way in which the ALVIS LIVE! environment supports "live" editing, this sample session demonstrates the use of the visualization customization features of the environment, which are designed to enable the user to personalize an algorithm by portraying it terms of a story. (For more on the pedagogical motivation behind the use of personal storytelling in novice algorithm construction and presentation activities, see [17].) In this example, a narrative is created along with the algorithm that tells story of two brothers cleaning their bedroom.

To code the "Partition" algorithm, we begin by creating the array of numbers to be partitioned. First, we activate the Create Array Tool in the Toolbox by clicking on it. Positioning the cursor in the top center of the Animation Window, we drag out an array with eight columns. When we release the mouse button, the SALSA statement `create array a1 with 8 cells` appears on the first line of the script editor; the execution arrow adjacent

to this line of code indicates that it has just been executed (see Figure 2). The ellipsis icon that appears at the end of this statement allows us to further tailor this statement at any time. By double clicking the ellipsis icon, we obtain an Array Properties dialog box, through which we can customize the properties (e.g., number of rows and columns) of the array that is created by the statement.

Next, we need to populate the array with values. To do this, we first click on the Populate tool to activate it. Double-clicking on the toolbox button activates the Populate Properties dialog box (see Figure 3a), in which we can customize aspects of the populate statement we are about to create. In this case, we set the array value for each array element to `random ints between 1 and 10`. We use the sketch pad editor accessible from this dialog box to draw a toy car, which we then set to be the custom image of each value to be populated. Note that the properties of any tool in the Toolbox can be customized via a similar properties dialog box.

After clicking anywhere in the array, we see the array instantly fill with random values (see Figure 3c). Simultaneously, the SALSA statement `populate room with random ints between 1 and 10` appears on the second line of the Script Window (see Figure 3b). The execution arrow advances to this line to indicate that it has just been executed.

The algorithm requires two array index variables to track the leftmost and rightmost indices inspected by the algorithm as it executes. To create these index variables, we again choose to use a Toolbox tool—this time, the Create Index tool. First, we use the Create Array Index Tool Properties dialog (see Figure 4) to customize the array index variables. We name the leftmost index variable `lil`, and give it the image of an orange stick figure selected from the Picture Gallery. Similarly, we name the rightmost index variable “big,” and give it a similar green figure for an image. We want the index variable `lil` to initially occupy the first array cell, and `big` to start at the last cell. As with the Populate Tool, simply clicking on the first and last cells generates two statements to create the index variables at the corresponding cells (see Figure 4c); semantic feedback on the execution of these statements is instantaneously provided in the Animation Window (see Figure 4c).

Next we add a narrative element to the story by employing the Say Tool from the Toolbox. The tool creates an animation statement that, when executed, displays a textual speech bubble emanating from a variable (see Figure 5). Here we specify a short conversation between the two “brother” index variables that motivates the pivot behavior by way of an analogy to cleaning a bedroom full of toys.

We need only define one more variable before we move to the implementation of the algorithm's loop: a variable `pivot` to hold the pivot value, which we initialize to the value in the first cell of the array. The variable `pivot` is created by first customizing the Create Variable Tool so that it creates a statement specifying the name `pivot`, a preliminary value `a[lil]`, and variable image of a toy car. Since this image was previously created, it can be selected from the Image Gallery. Now, with the Create Variable Tool selected, we simply click on the position at which we want the `pivot` variable to appear; a corresponding set statement is automatically generated (see Figure 6a) and executed, resulting in the pivot variable appearing at the desired location (see Figure 6b).

We are now ready to specify the main functionality of the algorithm: a `while` loop that walks the two array index variables towards each other, swapping values as necessary. To implement the loop, we directly type in the following SALSA code:

```
while lil < big
      while room[lil] <= pivot
            move lil right
      endwhile
      while room[big] > pivot
            move big left
      endwhile
      if lil < big
            swap room[lil] with room[big]
      endif
endwhile
```

As we type in each line of the code block above, ALVIS eagerly evaluates it on every keystroke, highlighting invalid or incomplete statements with a red background and reporting the current syntax error via a pop-up tooltip until the erroneous statement becomes syntactically valid (see Figure 7a). Most statements produce visual semantic feedback in the Animation Window; however, since `while` and `if` statements are control statements, they do not produce any explicit visual feedback.

Notice that an interesting situation—one that highlights a key problem that must be addressed by any "live" imperative programming environment—arises when we attempt to enter the second inner while loop in the above code. In the current execution context, `room[big]` is actually less than the `pivot` (since 3 < 10). This implies that execution cannot reach the `set` statement within that while loop. How can the WYSIWYC model enable the user to edit the body of an `if` or `while` statement whose condition evaluates to false, given its commitment to a "live" editing model in which code is continuously executed as it is typed in? In order to allow editing to proceed,

ALVIS LIVE! handles this problem by temporarily suspending "live" execution in such situations. Syntactic feedback continues as usual, but the Animation Window turns gray to indicate that it is not providing semantic feedback on what is currently being edited. Once the user types in the `endwhile` and proceeds to the next line, the Animation Window changes back to its normal appearance, and "live" execution proceeds as before.

Once the while loop is fully specified, all that remains is to move the original pivot value into its final position at `room[big]`. To accomplish this, we type in one final swap statement: `swap room[0] with room[big],` which immediately generates a swap animation, as shown in Figure 8. Our complete SALSA "Partition" algorithm is presented in Figure 9.

3.2 The WYSIWYC Model: A Closer Look

Having presented a sample programming session with ALVIS LIVE!, we now turn to a discussion of the subtleties of the WYSYWYC model's behavior, with an eye toward how that behavior was implemented. Recall that the design of the WYSIWYC model was guided by two key principles: (a) that a program need not be complete or correct in order for the programmer to explore its execution; and (b) that semantic feedback, in the form of a visual representation of execution state, should always be visible and accurately reflect the execution of the algorithm up to the current execution point. We have realized these principles through our implementation of an *execution controller* module that manages program execution state by maintaining an *execution stack.* The execution stack holds information about each statement that has been executed—most importantly, the next target line to be executed, along with a *reverse* statement, which undoes the effects of the executed statement. For example, if the statement `create array room with 8 cells` is executed, a reverse statement of the form `delete room` is pushed onto the stack.

Supporting user navigation in the WYSIWYC model proves tricky because of the fact that execution must follow the editing caret. This means that if the user moves the caret to a line that lies below the line at which execution is presently halted, the execution controller must automatically execute to the statement on that line. Conversely, a mouse click on a line that lies above the line currently being executed will cause a reverse execution back to that line. However, because such mouse clicks are seen as *navigation*, as opposed to an explicit request for *execution*, the execution controller makes the assumption that the user wants to elide the step-by-step semantic feedback that would normally be generated in the Animation Window. Thus, statements are executed "behind the

scenes" in order to get from the current execution line to the line to which the user navigated. This results in the user instantly seeing the current state of the program in the Animation Window, without any intermediate animations.

For example, suppose the user is navigating around in the "Partition" algorithm implemented in the previous section, (see Figure 9). Let's say that the user has just edited line 1, and subsequently clicks on line 11 (the start of the outer `while` loop). This click initiates a behind-the-scenes execution as follows. First, since line 1 is already executed and the next target line has already been calculated, the current line is set to that target line. The execution controller parses the text of that target line (line 1) into a `create array` statement object, and pushes a `delete array` statement object onto the reverse execution stack. The execution controller then proceeds to execute the target line, which enables it to locate the next target line. Since none of the statements between line 1 and 11 is a control structure, execution proceeds in a sequential manner; the target is always simply the next statement in the script.

Once execution reaches line 11, notice that there will be one statement object on the stack for each statement executed—11 statements in total. Since the user chose to navigate to line 11 instead of to forward-execute to that line, the execution controller suppresses updates of the Animation Window until line 11, at which point the Animation Window abruptly transitions from how it appeared after line 1 (see Figure 2b) to how it appears after line 11 (see Figure 6b).

Suppose the user instead presses the Step Backward button. In this case, the top element from the execution stack is popped, and its reverse statement is executed, with the resultant program state being visualized in the Animation Window. This is reverse *execution*, as opposed to reverse navigation. Likewise, if the user presses the Play Backward button, stack elements continue to be popped, and corresponding reverse statements continue to be executed and visualized, until the Stop button is pressed or until the stack is emptied. It is important to note that reverse execution proceeds without reference to the contents of the Script Window; it is based solely on the contents of the reverse statements stored on the execution stack.

Up to this point, we have discussed our implementation of "live" editing only as it pertains to sequential statements. Let us now consider the "live" editing of control and iterative statements (`if` and `while`), which potentially alter the flow of execution. Suppose, for instance, that the user would like to navigate back to a line in

the body of a `while` loop that has already been executed. Since the loop has already been executed, the last execution of its conditional statement must have ultimately evaluated to false. This implies that the body of the loop is no longer reachable. Thus, if execution were simply reversed until the desired loop body statement is reached, the user would be placed in an invalid "live" editing context. To ensure a valid editing context in such situations, the execution controller automatically *unwinds* a fully executed loop before allowing editing operations within that loop. Here, unwinding amounts to reverse executing to the line that immediately precedes the control statement, and then forward executing to the desired line.

Notice that this situation becomes more complex when we consider that, even when a loop is unwound, the target statement may be in the body of a control statement that does not evaluate to true. Our execution controller handles this case in the same way it was handled in the example above: by (a) pausing immediate execution at the start of the control statement; then (b) changing the animation window to gray to indicate it is no longer in sync; and finally (c) permitting the user to edit the statement in "non-live" mode, i.e., with syntactic feedback, but no semantic feedback. Once the user moves out of the unreachable block of code, the Animation Window changes back to normal, and "live" execution resumes.

4. Empirical Evaluation

Recall that our development of the WYSIWYC model was originally motivated by a key research question: "Can more immediate edit-time feedback enable novice learners to develop a mental model of code execution, ultimately allowing them to make coding progress more quickly than they would with a delayed feedback model?" Given the model's focus on novices who are first learning the imperative programming paradigm, one might ask an additional question: "Can a 'live' novice programming and visualization environment improve student experiences and success in an introductory computer science course?" In this section, we present a usability study we conducted as an initial step toward answering the first question, as well as a follow-up field study we conducted as an initial step toward answering the second question.

4.1 Usability study

To test and refine the WYSIWYC model, we conducted a usability study involving 21 novice programmers (18 male, 3 female) recruited out of the Fall, 2004 introductory computer science course at Washington State

University. We ran a total of 14 usability sessions over a one month period. In half of these sessions, students worked in pairs; the remaining seven sessions involved single participants. Students received course lab credit for participating.

Because our usability study was part of an iterative user-centered design process, not all participants interacted with the exact same version of the software. Indeed, over the course of the study, ALVIS LIVE! was refined in response to the usability issues that arose. Nonetheless, the basic WYSIWYC model presented in the previous section remained intact.

4.1.1 Procedure and Tasks

After filling out an informed consent form, participants first worked through a 15 minute tutorial in which they were guided through the construction of a (deliberately) buggy version of the "Find Max" algorithm, which iterates through an array of values in order to find the largest value. For their first task, participants were asked to explore the execution of that code in order to identify and fix the bug in the code. In a second task, participants coded, from scratch, the "Replace Zeroes" algorithm, which iterates through an array of values and replaces all zero values with a value specified by the user. Two additional tasks had participants embellish their animations by changing object properties, creating a custom background, and adding new animation code. At the end of each session, participants completed an exit questionnaire, adapted from the QUIS standard [18], that elicited their subjective ratings of ALVIS with respect to several usability categories. Sessions typically lasted between 90 and 105 minutes.

4.1.2 Results

A post-hoc analysis of the 20 hours of video collected in our study suggests that the WYSIWYC model's edit-time feedback mechanism was generally successful in enabling participants to understand, at edit time, the effects of their code, and ultimately to develop correct algorithmic solutions. In the debugging task, all 14 of the participant groups successfully identified the bug in the "Find Max" algorithm in an average time of just 89 seconds (sd = 61 seconds); these 14 participant groups needed an average of just 72 seconds more (sd = 71 seconds) to actually correct the bug. Participants had nearly equal success in the programming task: 13 of the 14 participant groups coded a correct solution to the "Replace Zeroes" problem in an average of 25 minutes and 41 seconds (sd = 13 minutes and 13 seconds).

With respect to participants' subjective reactions to the WYSIWYC model, we found reason to be optimistic. Averaging the results of 12 different QUIS exit questionnaire questions that addressed the learnability of the system, we find that ALVIS LIVE! scored 7.3 out of 10. While this overall learnability rating may appear somewhat low on the surface, we believe it is reasonable, given that our participants were all novice programmers who were performing potentially challenging tasks with an environment that they had never seen before.

Below, we first present evidence of the value of the WYSIWYC model's edit-time feedback. We then present evidence of two potential problems with the model, and consider how they might be overcome in the future.

4.1.2.1 Evidence of the Potential Value of Edit-Time Semantic Feedback

Our analysis of the video record revealed many instances in which participants were able both to write and understand code by exploiting the edit-time feedback provided by the WYSIWYC model. With respect to the ability of the WYSIWYC model to help one understand how code executes, we observed encouraging results in the debugging task, in which participants were required to find a bug that had been deliberately injected into the loop of the "Find Max" algorithm:

```
while current < cells of a1
  if a1[current] > maxSoFar
    set a1[current] to maxSoFar
  endif
  add 1 to current
endwhile
```

Notice that the `set` statement is buggy: instead of properly updating the maximum value encountered so far, the statement has the effect of setting all of the elements of the array to 0—the original value of the `maxSoFar` variable.

As stated above, all 14 participants groups were able to find and fix the bug in short order. To illustrate how ALVIS LIVE!'s dynamic visualization mechanism helped participants to identify and fix the bug so quickly, let us examine the interaction sequence of pair P6, which is representative of how participants commonly performed this task.

When pair P6 read that a bug existed in the code, they began by forward-executing the script from the `endwhile` statement—the point at which they had last made an edit. As they watched the animation unfold in the Animation Window, one member of the P6 pair (P6a) identified an obvious problem: "It definitely just wrote zeroes over all our stuff." To explore the problem further, pair P6 decided to reverse-execute the script to the beginning, and then to single-step through the code. When they reached the buggy line `set a1[current] to maxSoFar`, they clearly saw in the Animation Window that the first array cell's value erroneously changed to 0. As P6b explained, "OK, it set it (moves mouse cursor to array cell representing `a1[0]`) to 0. It's supposed to copy this (moves mouse cursor from the array cell representing `a1[0]` to the variable icon representing `maxSoFar`) to `maxSoFar`." In response to P6b's observation, P6a eagerly shared his new insight: "Oh, here's what it's doing. [It's] setting `a1[current]` to `maxSoFar`. So we just need to change those (points to `set a1[current] to maxSoFar` in Script Window) up." From the time P6a shared this insight, it took P6 just x seconds to fix the bug.

As mentioned above, all but one of the participant groups developed a correct algorithmic solution for the "Replace Zeroes" task. With respect to the WYSIWYC model's ability to help novices make coding progress without inordinate delays, we found strong evidence of the potential value of the WYSIWYC model's edit-time feedback in one particular step of the "Replace Zeroes" task. In that step, participants had to formulate a command to replace each zero found in an array of values with a user-inputted replacement value. Due to problems with mapping the concept of "replace" to the SALSA `set` command, several participants had difficulties with this task. The following sequence of interactions that led participant P1 to a solution was typical.

Initially, P1 mapped the "replace" wording in the task description directly to a "replace" command, which does not exist in SALSA. Accordingly, he proceeded to type "replace" into the Script Window. After he released the "e" key, he observed red highlighting and an error tooltip, which succeeded in letting him know that "replace" is not a valid command. Consulting the one page SALSA quick reference guide with which participants were furnished, P1 identified the `swap` command as a viable candidate to accomplish the task. He then typed in a valid `swap` command; however, the visual feedback in the Animation Window showed him immediately that the effect of the command was not what he wanted. As P1 explained,

> "I saw in the visualization that swap was gonna set the 0 to 5, which I wanted, but it also set my replace variable to 0, and I might need that later… It shows you right after you type it in. So I'm going to have to get used to watching what it's doing while I type, and that'll be handy."

After this slight misstep, P1 settled on the `set` command, and quickly formulated a correct solution.

We speculate that, in the above scenario, a novice programmer operating under a conventional "delayed feedback" development model would have written the entire algorithm before discovering that "replace" and "swap" were inappropriate commands. In contrast, as the above interaction sequence illustrates, the WYSIWYC model provides a concrete graphical context that gives programmers an immediate basis for thinking about the execution of their code and evaluating its effects. As a result, they are able to make corrections on the spot, without having to guess what a given line of code will actually do. As we can see, an ability to recognize syntactic and semantic errors at edit time can provide a powerful context for their resolution.

4.1.2.2 Evidence of Potential Problems with the WYSIWYC Model

As with most preliminary studies of new technology, we discovered several problems with the WYSIWYC development model. While most of these turned out to be minor bugs and design flaws that could be easily fixed, two problems stand out as worthy of further discussion.

Edit-time syntactic feedback not noticed. We observed that many participants did not even look at the Script Window as they typed in code, so they failed to notice the tooltips that are updated on every keystroke with edit-time syntactic feedback. Moreover, after entering a syntactically-incorrect line of code, many participants seemed perfectly content to move to the next line, despite the fact that an exclamation point icon (indicating a syntax error) appeared to the left of the line they had just entered. Despite our attempts to make this edit-time error feedback more prominent—for example, by highlighting syntactically invalid lines of code in red—many of our participants still did not fix syntactically incorrect code. Since ALVIS LIVE! simply ignores syntactically incorrect code, this led to situations in which participants obtained unexpected execution results because they thought a line of code was correct and had executed, when, in fact, it had not.

While we have gathered evidence that the edit-time execution feedback that appears in the Animation Window is beneficial to novice programmers, we are not convinced that allowing syntactically incorrect code to execute with

no visible effect is appropriate. In ongoing research, we are considering ways to illustrate the execution of syntactically incorrect code in the Animation Window, or, alternatively, to disallow the formulation of syntactically incorrect code completely, as is the case with ALICE's drag-and-drop editor [9].

Input dialog boxes disruptive to work flow. In order to facilitate user input, the SALSA language provides an `input` command (see Table 1). In ALVIS, that command manifests itself in the form of a modal dialog box that appears whenever the command is executed. The dialog box presents a text string prompt (customizable through the properties of the `input` statement) requesting the user to input a value into a text box; the user must explicitly dismiss the dialog box in order for the script's execution to proceed beyond the `input` statement.

Most participants in our usability study understood the need to enter a value into the input dialog box when they first specified an input command; after all, they wrote input commands for the purpose of getting data from the user. However, in order to test and debug their code, participants often re-executed it from the beginning, thus requiring them to repeatedly re-execute their `input` commands. Each subsequent execution of an input command generated a modal dialog box. That they were required to enter a value into, and explicitly dismiss, each such dialog box turned out to be an annoyance for most participants. This was because such dialog boxes often appeared when participants were consumed by a debugging or testing task; the need to explicitly tend to these dialog boxes clearly disrupted their workflow.

One way to address this problem would be simply to remove the `input` command from the SALSA language. Indeed, the "liveness" of the WYSIWYC model turns the `set` command into a de facto `input` command. However, obtaining user input is a fundamental part of algorithmic problem solving, as evidenced by the fact that the algorithm design problems found in introductory texts often contain explicit requests to obtain user input. As a result, we believe that the SALSA language should retain the `input` command. The design question then becomes, "How can we make the `input` command less disruptive to the workflow of the ALVIS LIVE! user?"

In ongoing work, we are the exploring the possibility of having an input command generate a modal dialog box only the *first* time it is executed. In each subsequent execution of the command, the value originally entered by the user is assumed. If the user does, in fact, want to re-enter that input value each time the input command is executed, the user is able to override this behavior by right-clicking the command and toggling a check mark from "Use input

value from original execution of input command" to "Explicitly request input on each execution of the input command."

4.2 Follow-up Field Study

In order to evaluate the WYSIWYC model within a real-world setting, we conducted a follow-up field study of the use of ALVIS LIVE! in an introductory college computer science course—in particular, the Spring, 2005 offering of CptS 121 at Washington State University. The design of our follow-up field study was nearly identical to that of the field study that motivated our interest in a more "live" AV development model (see Section 2). As in that field study, students completed a 3 hour "studio experience" in which they constructed and presented alternative solutions to algorithm design problems. Likewise, we used the same field techniques in our follow-up study as we used in our original study (*viz*., participant observation, videotaping, artifact collection, interviewing, questionnaires; see Section 2.1). The only difference between the two studies was that, in our follow-up field study, students used the ALVIS LIVE! software, instead of a text editor and art supplies, to complete the "studio experience." Below, we present and discuss key observations relevant to the ALVIS LIVE! software. For a comprehensive treatment of this study and the one presented in Section 2, see [17].

4.2.1 Key Observations

As in the study presented in Section 2, we performed a post hoc video analysis of the coding activities of a random sample of 9 pairs of students who participated in the ALVIS LIVE! studio experience. Recall that, when we presented our analysis of student coding activities in Section 2, a key concern we raised was that students appeared to experience an inordinate number of "stuck" periods during which they made no visible coding progress. As we pointed out in Section 2, 80% of the student pairs we observed experienced at least one stuck period of 2.5 minutes or longer. In contrast, we observed no such stuck periods in our field study of ALVIS LIVE!. In fact, the longest "stuck" period we could find in our ALVIS LIVE! field study video samples was only 30 seconds. Indeed, we observed that, even when ALVIS LIVE! pairs became "stuck," they were still able to engage in constructive activities, such as reviewing the execution of their code so far.

Another relevant observation concerns the amount of outside help students in our two field studies required in order to code their solutions. Recall that, in the study presented in Section 2, students spent 20 percent of their algorithm coding time consulting with teaching assistants. In stark contrast, students in our follow-up study spent

only 7 percent of their time talking with teaching assistants. It thus appears that, in addition to spending less time being "stuck," ALVIS LIVE! pairs were able to write their solutions more independently than the pairs of students in our original study..

Given that the students who used ALVIS LIVE! experienced fewer "stuck" periods, and did not consult with teaching assistants as frequently, one might ask whether they actually performed coding tasks more quickly and accurately. While we unfortunately did not gather time-on-task data in these two studies, we did analyze the semantic correctness of the algorithmic solutions produced by the students who participated in both field studies (see Table 3). According to a non-parametric Kruskall-Wallis[1] test, students who used the ALVIS LIVE! environment made significantly fewer semantic errors in their solutions than students who used a text editor and art supplies ($df = 1$, $H = 4.53$, $p = 0.03$). Thus, as one might have expected, the immediate edit-time visual feedback provided by the ALVIS LIVE! environment resulted not only in students' experiencing shorter "stuck" periods and needing fewer interventions by a teaching assistant, but also in students' developing code that was significantly more correct.

4.2.2 Discussion

In this follow-up field study, we wanted to investigate whether students might benefit from using a "live" algorithm development and coding environment like ALVIS LIVE! within the kind of real-world "studio" setting that ALVIS LIVE! has been designed to support. Although our findings must clearly be seen as preliminary because of the relatively small sample size and limited scope of the study, we believe that they give one reason to be optimistic about the potential of ALVIS LIVE! in such a setting. Indeed, as an algorithm development tool, ALVIS LIVE! enabled novice learners who were in the early weeks of their programming careers to develop correct algorithmic solutions with minimal intervention from teaching assistants. Moreover, while beyond the scope of this article, a post hoc video analysis of the algorithm presentation sessions that took place in this follow-up field study provides additional evidence of the value of ALVIS LIVE! in a studio setting. That analysis, which we report in detail elsewhere [17], suggests that ALVIS LIVE! promotes educationally-valuable *discussions* with a sharp focus on the specific details of algorithm behavior, leading to the collaborative identification and repair of semantic errors.

[1]Because our data points are ratios with varying denominators, they violate the assumptions of parametric statistics; we therefore use a non-parametric Kruskal-Wallis test in this analysis.

5. Related Work

The key contribution of the WYSIWYC model presented here is the immediacy of the feedback it provides to the user. In order to help us compare the WYSIWYC model to prior research, we find it helpful to enlist Tanimoto's "liveness" taxonomy [19], which formally distinguishes four levels of programming environment "liveness":

Level 1: a programming environment that allows one to interactively create, but not execute, a program.

Level 2: a programming environment that allows one to interactively create a program. The program must then be explicitly submitted to the environment for execution.

Level 3: a programming environment in which "any edit operation triggers computation by the system. It is thus not necessary for the user to explicitly submit [a program] for execution" ([19], p. 129).

Level 4: a programming environment that allows a program to be edited *while it is running*; the program does not have to be stopped or restarted in order for changes to be reflected in the program's execution.

With respect to Tanimoto's taxonomy, the WYSIWYC model presented here falls somewhere between level 3 and 4 "liveness": it automatically reevaluates the code on every edit, leading to the dynamic update of a visualization of the program. (level 3 "liveness"). In addition, it enables a user to edit code, albeit in a restricted way, while a program is executing (level 4 "liveness"). In particular, if the user attempts to edit code while a program is running, the execution arrow automatically jumps to the point at which the user attempts to edit the code, where execution automatically resumes. As discussed in Section 3, the WYSIWYC model behaves in this way in order to ensure that the visualization is always consistent with the execution of the code that appears in the Script Window.

In the review that follows, we use Tanimoto's taxonomy to help us juxtapose the WYSIWYC model with prior research that has developed environments and frameworks to support novice algorithm development and visualization. We organize our review of this related work into three areas:

- scripting languages that support learner-constructed algorithm visualizations,
- non-imperative visual programming environments geared toward novices and non-programmers, and
- imperative programming and visualization environments that target novice learners, and especially introductory computer science students.

Note that a thorough treatment of the prodigious body of research into novice programming environments is beyond the scope of this article; for a recent comprehensive review, see [20].

5.1 Algorithm Visualization Scripting Languages

Traditional algorithm visualization systems (see, e.g., Brown's BALSA [21] and Stasko's TANGO [22]) support a user model in which an expert creates visualizations, which are then explored by learners. In order to increase learner engagement with AV technology, a more recent line of AV research has explored approaches and technology to facilitate learner-constructed algorithm visualizations (level 4 in the engagement framework of Naps *et al*. [2]). For example, SAMBA [6] and AnimalScript [23] are AV scripting languages with which computer science students can annotate their algorithm source code in order to generate visualizations of their algorithms. Such scripting languages empower learners to build visualizations that are far more customizable than the ones supported by our ALVIS LIVE! environment; however, because such scripting languages are used in tandem with conventional compiled programming languages, they support only level 2 "liveness," and they are beyond the reach of novice computer scientists first learning to program.

5.2 Visual Programming Environments for Novice Learners

Numerous visual programming environments have been developed to empower non-programmers to program. In order to make programming more accessible to children who are first learning to program, one line of this research has explored programming environments that enable learners to program by building and manipulating animated visual representations of program execution. Perhaps the earliest example of this line of work is Papert's Logo system [24], which enables children to explore the world of programming by commanding a "turtle" to draw computer graphics. The Logo system, and the constructionist learning theory upon which it is based, inspired a family of more powerful and sophisticated children's programming environments—most notably, StarLogo [25] and NetLogo [26], which enable students to develop massively parallel animated simulations of scientific phenomena. Because Logo, StarLogo, and NetLogo require the user to explicitly request code to be executed, they all support level 2 "liveness." However, the most recent addition to the Logo family, Kedama [27], uses the Squeak programming framework [28] to support level 4 "liveness." The user can edit the code for a massively parallel simulation while it is executing, and the changes will be automatically reflected in the visual display.

In a similar vein, several declarative (rule-based) programming environments aim to empower children to program. For example, in ToonTalk [29], children program by specifying rules to robots that act out animated cartoons. In Stagecast Creator [30], children specify simulations by demonstrating graphical rewrite rules in a grid-based world. Drawing extensively on empirical studies in which children specified programs independently of programming environments, HANDS [8] enables children to program by using a visual playing card metaphor to specify to a computing agent the rules for how the elements of a graphical simulation should behave; the computing agent carries out the rules to execute the simulation. Both ToonTalk and Stagecast Creator require the user to explicitly execute a program, which must be halted in order to be edited; thus, they support level 2 "liveness." In contrast, HANDS allows the user to change computational rules while a simulation is running, and hence supports level 4 "liveness."

As we have seen, because they are declarative, the programming environments reviewed in this section sometimes achieve a higher level of "liveness" than our WYSIWYC editing model; however, these systems differ fundamentally with respect to their underlying goal: Whereas these systems aim to empower non-programmers (especially children) to program, our goal is to help novices to learn the *imperative* programming paradigm commonly taught in undergraduate introductory computer science courses.

5.3 Imperative Programming and Visualization Environments for Novice Learners

Sharing that goal, novice programming environments such as ALICE [9], JELIOT [10], BlueJ [11], and RAPTOR [12] have all been enlisted to teach imperative programming in undergraduate introductory computer science courses. Like the environment presented here, all of these environments generate concrete visual representations of program state as a program executes. Most of these systems also provide some form of edit-time feedback on syntactic correctness. (In fact, ALICE disallows syntactically incorrect code through a drag-and-drop code editor.) However, while ALICE is notable for its "WhyLine" feedback mechanism, which has been shown to greatly reduce debugging effort [31], all of these environments support a similar development model in which writing code and viewing the resulting visual representation are temporally distinct activities (level 2 "liveness"). The WYSIWYC editing model presented here differs from these environments in that it provides novice programmers with immediate visual feedback on their code's syntactic and semantic correctness *at edit time*, and

even allows novices to edit their code while it is running, subject to the limitation that the execution arrow automatically jumps to the line of code being edited (level 3½ "liveness").

6. Summary and future research

As we have shown, existing imperative programming and visualization environments for novices support an editing model that incurs a temporal lag between the time code is edited, and the time execution feedback on that code is provided to the user. With respect to Tanimoto's "liveness" taxonomy [19], this type of editing model supports only "level 2 liveness." Because novices in particular are known to lack robust mental models of how code executes, we have argued that a lack of edit-time execution feedback can lead to gulfs of evaluation [15] that inhibit programming progress. Motivated by a field study of novice algorithmic problem solving and visualization, we have presented a solution to this problem: WYSIWYC, an alternative editing model that provides, on an edit-by-edit basis, immediate visual feedback on the syntactic and semantic correctness of code. As we have shown, our WYSIWYC model falls somewhere between level 3 and 4 liveness on Tanimoto's scale, thus exceeding the level of liveness of previous imperative novice programming and visualization environments. Moreover, an analysis of usability and field study data suggests that our model's dynamic visual feedback has the potential to aid novice programmers in quickly identifying and remedying programming errors, leading to the development of semantically correct code.

In addition to refining solutions to the problems with the WYSIWYC model discussed in Section 4, we would like to explore the following two issues in future research.

Expand the range of programs that can be developed within the WYSIWYC model. In developing the WYSIWYC model, we have specifically targeted introductory computer science students who are in the first five weeks of their programming careers. As a consequence, we have deliberately restricted the range of programs that can be written in the SALSA language, in line with the philosophy of pedagogical "mini-languages" [32]. Indeed, as implemented in ALVIS LIVE!, our WYSIWYC model only supports single-procedure, array-iterative algorithms. While we have found this programming domain to be sufficiently rich for the first five weeks of an introductory programming course, it quickly proves too restrictive beyond that time period, thus requiring students to transition to a more powerful programming language and environment.

Clearly, one of the reasons the WYSIWYC model works is because it targets the extremely restricted programming domain of single-procedure algorithms. However, we believe that this kind of "live" editing model has the potential to aid novice programming beyond this restricted domain. In future work, we would like to investigate whether novice learners might benefit from using the WYSIWYC model for a greater portion of an introductory programming course, and perhaps for the entire course. To that end, we intend to explore an expanded programming domain consisting of both (recursive) procedures and linked structures, both of which are central to a standard introductory programming course. While we believe that the WYSIWYC model will easily be able to support the programming of linked structures, we anticipate that finding a highly usable means of supporting the "live" development of procedures will prove tricky. Indeed, because procedures must be written *before* they are called, we will have to come up with a scheme for constructing a useful programming context for procedures—one that enables programmers to leverage the benefits of "live" editing. To that end, we might consider artificially creating a "live" procedural context by automatically assigning reasonable default values to procedure parameters; if desired, programmers could override those defaults with values that they felt were more realistic.

Evaluate the effectiveness of the WYSIWYC model more rigorously. While the empirical evaluation of ALVIS LIVE! presented here constitutes a reasonable start, we believe that the potential benefits of "live" editing need to be evaluated more rigorously. In one previous experimental study of programming enviornment "liveness," Wilcox et al. [33] found that "live" feedback had advantages over "delayed" feedback in a debugging task. However, we are not aware of any prior experimental studies that have specifically examined the impact of immediate feedback within a novice programming environment. To that end, we recently conducted an experimental study (n = 60) in which novice programmers coded algorithmic solutions using three alternative environments with varying levels of "liveness":

- a simple text editor (the *control* group);
- a "non-live" version of ALVIS LIVE! in which programmers must explicitly request syntactic and semantic feedback (the *delayed* group); and
- a full version of ALVIS LIVE! (the *immediate* group).

In ongoing work, we are performing a statistical analysis of the syntactic and semantic correctness of the programs developed in each alterative environment, and we are undertaking a detailed post-hoc video analysis of

the temporal evolution of participants' code in each alternative environment, with the goal of understanding the ways in which feedback immediacy specifically impacts novice coding practices. We look forward to reporting the results of these analyses in future publications.

7. Acknowledgments

This article is an expanded version of a paper we originally presented at the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing [34]. We are grateful to the reviewers of that paper for their helpful comments and suggestions, which allowed us to improve the literature review and focus of this expanded article. Sean Farley and Sudhir Garg helped implement ALVIS LIVE!. This research is funded by the National Science Foundation under grant no. 0406485. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF.

8. References

1. C. D. Hundhausen, S. A. Douglas, & J. T. Stasko (2002) A meta-study of software visualization effectiveness. *Journal of Visual Languages and Computing* **13,** 259-290.
2. T. L. Naps, G. Roessling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. Mchally, S. Rodger, & J. A. Valazquez-Iturbide (2003) ITiCSE 2002 working group report: Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin* **35,** 131-152.
3. 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.
4. M. D. Byrne, R. Catrambone, & J. T. Stasko (1999) Evaluating animations as student aids in learning computer algorithms. *Computers & Education* **33,** 253-278.
5. A. W. Lawrence, A. N. Badre, & J. T. Stasko (1994) Empirically evaluating the use of animations to teach algorithms. In: *Proceedings of the 1994 IEEE Symposium on Visual Languages* IEEE Computer Society Press, Los Alamitos, CA, pp. 48-54.
6. 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.
7. C. D. Hundhausen (2002) Integrating algorithm visualization technology into an undergraduate algorithms course: Ethnographic studies of a social constructivist approach. *Computers & Education* **39,** 237-260.
8. J. F. Pane, B. A. Myers, & L. B. Miller (2002) Using HCI techniques to design a more usable programming system. In: *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments* IEEE Computer Society, Los Alamitos, pp. 198-206.
9. W. Dann, S. Cooper, & R. Pausch (2000) Making the connection: Programming with animated small world. In: *Proc. ITiCSE 2000* ACM Press, New York, pp. 41-44.
10. R. Ben-Bassat Levy, M. Ben-Ari, & P. Uronen (2003) The Jeliot 2000 program animatino system. *Computers & Education* **40,** 1-15.
11. M. Kölling, B. Quig, A. Patterson, & J. Rosenberg (2003) The BlueJ system and its pedagogy. *Journal of Compuer Science Education* **13,** 249-268.
12. M. Carlisle, T. Wilson, J. Humphrieis, & S. Hadfield (2005) RAPTOR: A visual programming environment for teaching algorithmic problem solving. In: *Proc. ACM SIGCSE 2005 Symposium* ACM Press, New York.

13. J. Bonar & E. Soloway (1985) Preprogramming knowledge: A major source of misconceptions in novice programmers. In: *Studying the Novice Programmer* (E. Soloway and J. Spohrer, Eds.) Lawrence Erlbaum, Hillsdale, NJ, pp. 133-161.
14. T. R. G. Green & M. Petre (1996) Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Langauges and Computing* **7,** 131-174.
15. D. A. Norman (1990) *The Design of Everyday Things* Doubleday, New York, 257.
16. C. D. Hundhausen & S. A. Douglas (2002) Low fidelity algorithm visualization. *Journal of Visual Languages and Computing* **13,** 449-470.
17. C. D. Hundhausen & J. L. Brown (2005) Personalizing and discussing algorithms within CS 1 Studio Experiences: An observational study. In: *Proc. 2005 International Computing Education Research Workshop* ACM Press, New York, pp. 45-56.
18. J. P. Chin, V. A. Diehl, & K. Norman (1987) Development of an instrument measuring user satisfaction of the human-computer interface. In: *Proc. 1988 ACM SIGCHI Conference on Human Factors in Computing Systems* ACM Press, New York, pp. 213-218.
19. S. L. Tanimoto (1990) VIVA: A visual language for image processing. *Journal of Visual Languages and Computing* **1,** 127-139.
20. C. Kelleher & R. Pausch (2005) Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys* **37,** 83-137.
21. M. H. Brown (1988) *Algorithm animation* The MIT Press, Cambridge, MA.
22. J. T. Stasko (1990) TANGO: A framework and system for algorithm animation. *IEEE Computer* **23,** 27-39.
23. G. Rößling & B. Freisleben (2001) AnimalScript: An Extensible Scripting Language for Algorithm Animation. In: *Proceedings ACM 2001 SIGCSE Symposium* ACM Press, New York, pp. 70-74.
24. S. Papert (1980) *Mindstorms: Children, Computers, and Powerful Ideas* Basic Books, New York.
25. M. Resnick (1994) *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds* MIT Press, Cambridge, MA.
26. P. Bilkstein, D. Abrahamson, & U. Wilensky (2005) NetLogo: Where we are, where we're going. In: *Proc. 2005 Interaction Design and Children* ACM Press, New York.
27. Y. Ohshima (2005) Kedama: A GUI-based interactive massively parallel particle programming system. In: *Proceedings 2005 IEEE Symposium on Visual Languages and Human-Centric Computing* IEEE Computer Society, Los Alamitos, pp. 91-98.
28. J. Steinmetz (2001) Computers and Squeak as environments for learning. In: *Speak: Open Personal Computing and Multimedia* (M. Guzdial and K. Rose, Eds.) Prentice Hall, New York, pp. 453-482.
29. K. Kahn (1996) ToonTalk: An animated programming environment for children. *Journal of Visual Languages and Computing* **7,** 197-217.
30. D. C. Smith, A. Cypher, & J. Spohrer (1994) KidSim: Programming agents without a programming language. *Communications of the ACM* **37,** 54-67.
31. A. Ko & B. Myers (2004) Designing the Whyline: A debugging interface for asking questions about program failures. In: *Proc. ACM SIGCHI 2004* ACM Press, New York, pp. 151-158.
32. P. Brusilovsky, E. Calabrese, J. Hvorecky, A. Kouchnirenko, & P. Miller (1997) Mini-languages: A way to learn programming principles. *Education and Information Technologies* **2,** 65-83.
33. E. Wilcox, J. Atwood, M. Burnett, J. Cadiz, & C. Cook (1997) Does continuous visual feedback aid debugging in direct-manipulation programming systems? In: *Proceedings CHI '97* ACM Press, New York, pp. 258-265.
34. C. D. Hundhausen & J. L. Brown (2005) What You See Is What You Code: A radically dynamic algorithm visualization development model for novice learners. In: *Proceedings 2005 IEEE Symposium on Visual Languages and Human-Centric Computing* IEEE Computer Society, Los Alamitos, pp. 140-147.

9. Figures

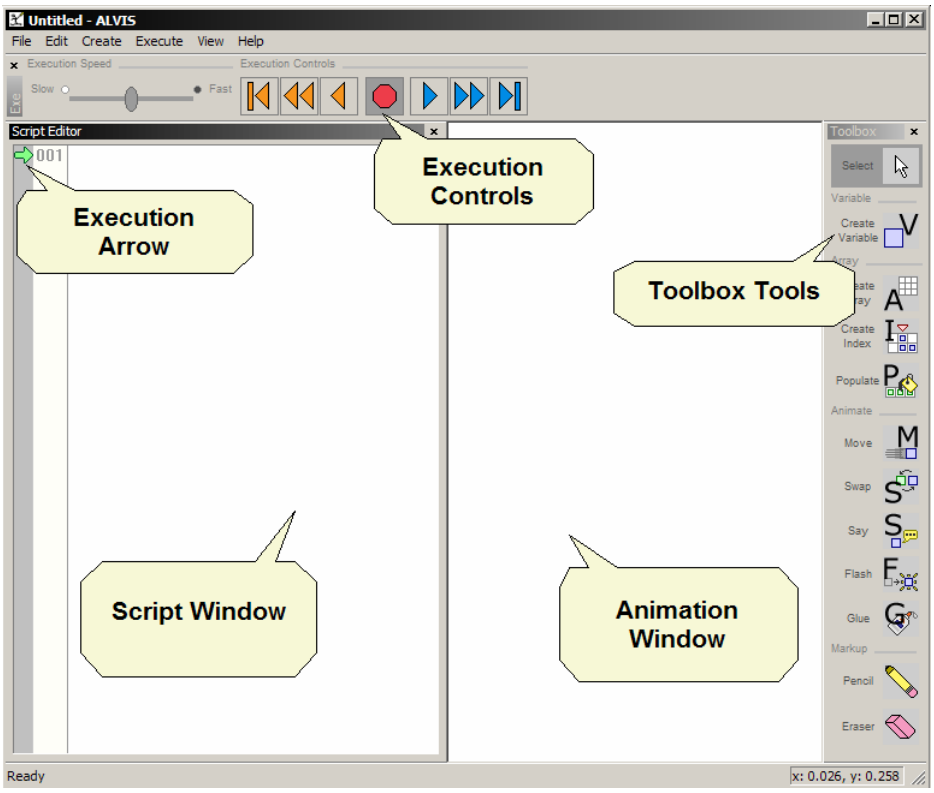
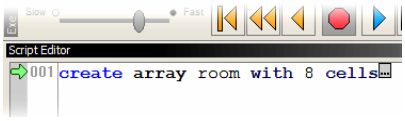
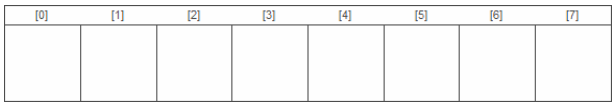


Figure 1. Annotated Snapshot of the ALVIS LIVE!

Environment. SALSA Pseudocode can be directly typed into the Script Window. Alternatively, Toolbox tools can be used within the Animation Window to generate many commands by direct manipulation

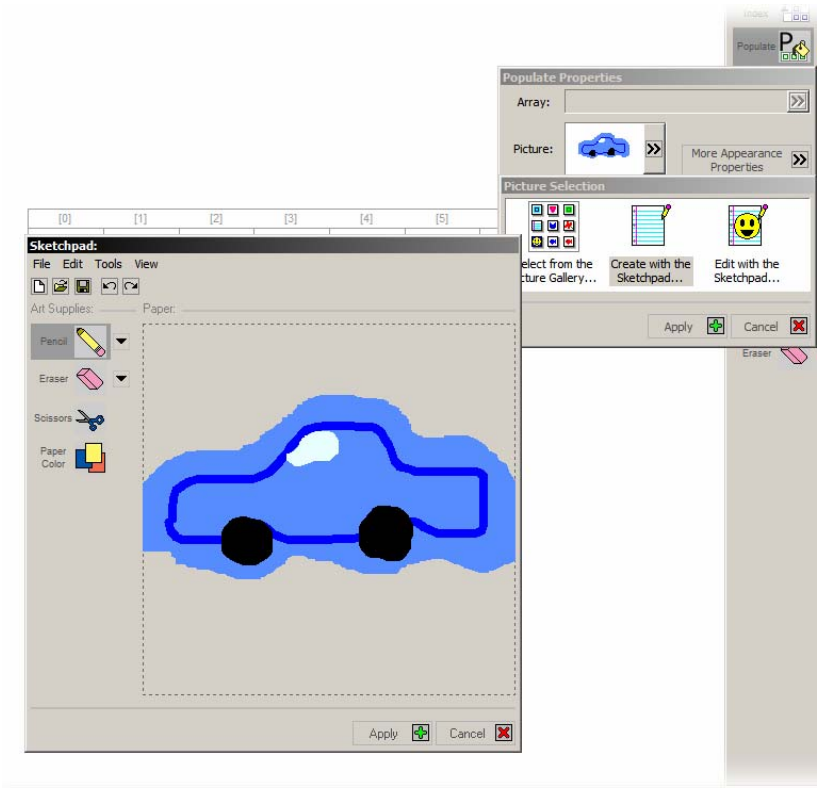


(a) SALSA statement generated in the Script Window by application of the Create Array Tool

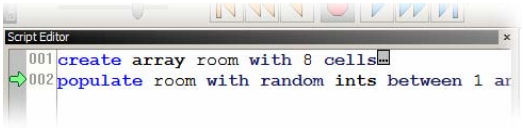


(b) Visual representation of the array 'room' presented in the Animation Window

Figure 2. Creating an array in ALVIS LIVE!



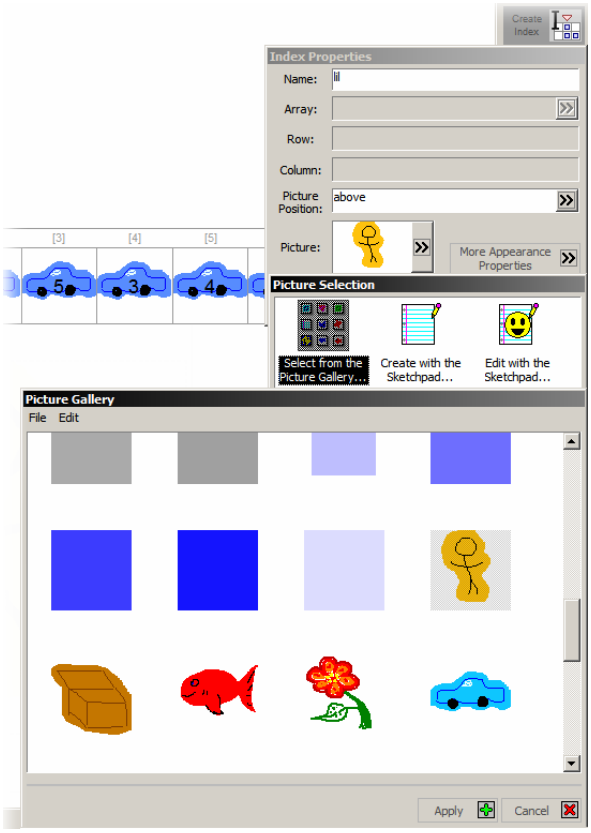
(a) Customizing the Populate Array Tool with the Populate Properties Dialog



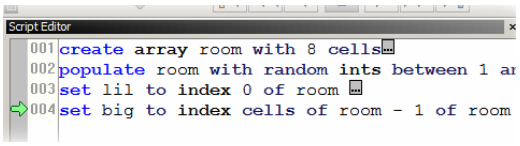
(b) SALSA statement generated in the Script Window by the application of the Populate Array Tool

(c) Visual representation of the populated array 'room' presented in the Animation Window

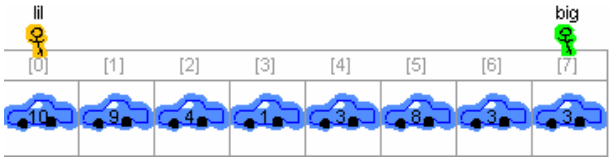
Figure 3. Populating an array in ALVIS LIVE!



(a) Customizing the Create Iterator Tool

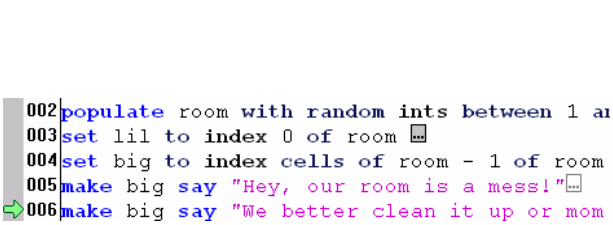


(b) SALSA statement generated in the Script Window by two applications of the Create Iterator Tool

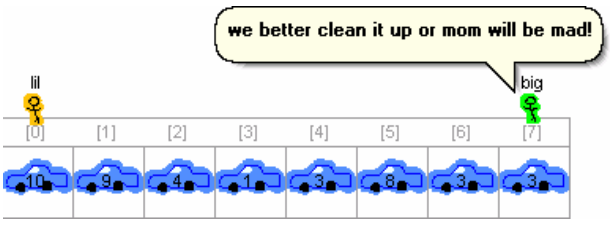


(c) Visual representation of the iterators 'lil' and 'big' presented in the Animation Window

Figure 4. Adding array indexes in ALVIS LIVE!

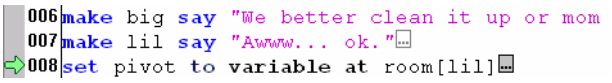


(a) SALSA statement generated in the Script Window by two applications of the Say Tool

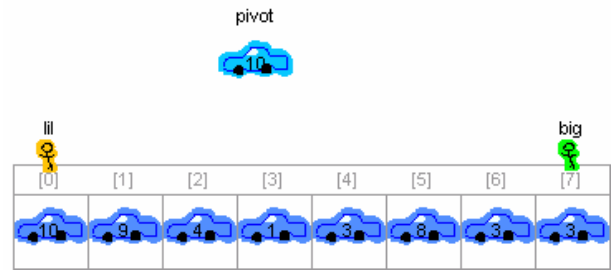


(b) Animation of a say statement presented in the Animation Window

Figure 5. Adding narrative animations to a SALSA script in ALVIS LIVE!

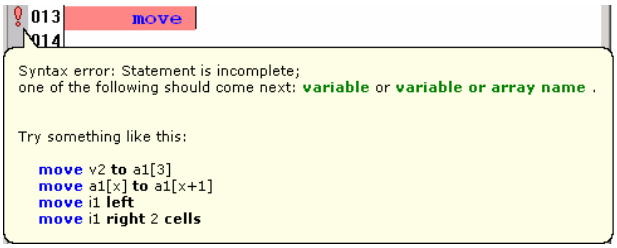


(a) SALSA statement generated in the Script Window by application of the Create Variable Tools

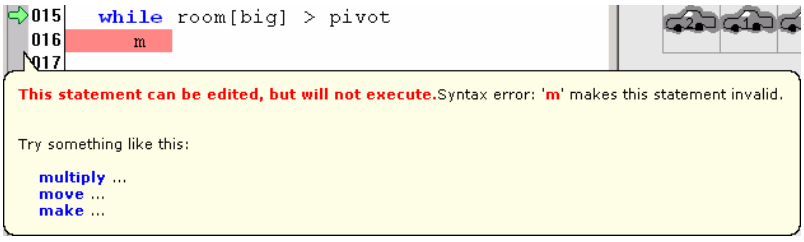


(b) Visual representation of the variable 'pivot' and iterator 'placement' presented in the Animation Window

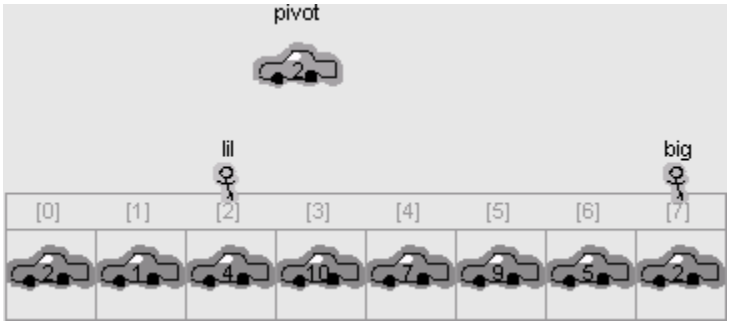
Figure 6. Creating a variable in ALVIS LIVE!



(a) ALVIS LIVE! provides up-to-the-keystroke feedback on a statement as it is entered



(b) ALVIS LIVE! allows a statement that is currently unreachable (because it is in the body of an if or while statement that evaluates to false) to be edited; in such cases, syntactic, but not semantic, feedback is provided



(c) When an unreachable statement is being edited, the Animation Window is disabled in order to maintain an accurate synchronization between the Script and Animation Windows.

Figure 7. Editing statements in ALVIS LIVE!

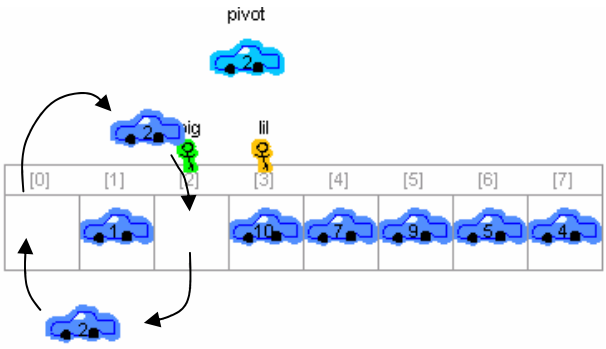


Figure 8. Adding a swap animation to a SALSA script in ALVIS LIVE!. Effect lines were added to this figure to convey the motion of the animation; they are not actually part of the animation as seen by the user.

```
001 create array room with 8 cells
002 populate room with random ints between 1 and 10
003 set lil to index 0 of room
004 create variable big as index columns of room - 1 of room
005 make big say "Hey, our room is a mess!"
006 make big say "We better clean it up or mom will be mad!"
007 make lil say "Awww... ok."
008 set pivot to variable at room[lil]
009 make big say "Anything bigger than (or equal to)" & pivot & " is mine!"
010 make lil say "OK, then anything smaller than " & pivot & " is mine!"
011 while lil <= big
012    while room[lil] <= pivot
013       move lil right
014    endwhile
015    while room[big] > pivot
016       move big left
017    endwhile
018    if lil < big
019       swap room[lil] with room[big]
020    endif
021 endwhile
022 swap room[0] with room[big] --position pivot
```

Figure 9. The complete Partition Algorithm implemented in Section 3.1

10. Tables

Table 1 Summary of SALSA Pseudocode Language

COMMAND	MEANING
`create`	Creates new variables, arrays, and array indexes (special variables that reference array cells)
`set`	Creates and sets the value of new variables and array indexes on the fly, or changes the values of existing variables, arrays, and array indexes
`input`	Prompts the user for a variable's value; the variable is created if it does not already exist
`populate`	Fills the empty cells of an array with new values
`print`	Outputs a value to the user
`if…` `elseif…` `else…` `endif`	Specifies blocks of code that execute conditionally based on the results of Boolean tests
`while…` `endwhile`	Specifies a loop that executes as long as the Boolean test evaluates to true
`move`	Moves a variable, array, or array index to a new location
`swap`	Causes two variables or array indexes to change positions
`flash`	Causes a variable, array or array index to flash for a period of time.
`make…say`	Causes a variable or array index to "speak" a text string through an animated speech bubble
`glue`	Sticks two variables or indexes together so they move or swap in unison.

Table 2 Explanation of the ALVIS Live Execution Controls

CONTROL	FUNCTION
 Execution Speed	Controls the rate at which execution occurs.
 Jump to Start of Script	Jumps immediately to the beginning of the script, effectively resetting all execution and clearing the Animation Window.
 Play Backward	Automatically reverse executes each previously executed statement one by one, updating the Animation Window to reflect the effects of each.
 Step Backward	Reverse executes the most recently executed statement and updates the Animation Window to reflect the results of the computation.
 Stop Playing	Stops execution if automatic reverse or forward execution is active.
 Step Forward	Forward executes the most next statement and updates the Animation Window to reflect the results of the computation.
 Play Forward	Automatically forward executes each statement in the script one by one, updating the Animation Window to reflect the effects of each.
 Jump to End of Script	Jump immediately to the end of the script effectively executing every statement in the script until no other statement can be found to execute.

Table 3. Comparison of the Semantic Accuracy of Solutions in the Original and Follow-up Field Studies

	Lines of code per solution		Semantic errors per solution		Semantic errors per line of code	
Field Study	*M*	*SD*	*M*	*SD*	*M*	*SD*
Original (Text editor + Art Supplies)	19.07	7.77	0.77	0.95	0.04	0.01
Follow-up (ALVIS LIVE!)	21.55	6.56	0.36	0.78	0.01	0.01