What You See Is What You Code: A Radically Dynamic Algorithm Visualization Development Model for Novice Learners

Christopher D. Hundhausen and Jonathan Lee Brown
Visualization and End User Programming Laboratory
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164-2752 USA
{hundhaus, jbrown}@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, educators have explored 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 the algorithm is temporally distinct from viewing and interacting with the resulting visualization. Given that novice learners are known to lack robust mental models of how code executes, they would especially stand to benefit from a more dynamic development model that narrows the gap between coding an algorithm and viewing its visualization. We have implemented "What You See Is What You Code," a novel, "radically dynamic" development model to facilitate learner-constructed algorithm visualizations. In our development model, the line of algorithm code currently being edited is reevaluated on every edit, leading to the dynamic update of an accompanying visualization of the algorithm. Analysis of usability 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 understand their code's execution.

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 algorithm visualization 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, 8-13] 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*. Because novice learners are known to have difficulties with formulating syntactically-correct code, and with conceptualizing the execution effects of program code [14], it would appear that novice learners 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 paper explores the above possibility by presenting and empirically evaluating "What You See Is What You Code" (WYSIWYC), a "radically dynamic" AV development 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 visualization 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

- dynamic feedback on the line's syntactic correctness,

- dynamic suggestions for how to formulate syntactically correct code, and
- the dynamic update of an accompanying visualization of the algorithm in an adjacent window.

Thus, in this "radically dynamic" 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. In a usability study involving introductory computer science students, this development model showed great promise in helping novices to identify and correct program errors quickly, and ultimately to understand the execution of the code they were writing.

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

2. Related Work

In order to increase learner engagement with AV technology, several recent lines of AV research have explored approaches and technology to facilitate visualization construction (level 4 in the engagement framework of Naps et al. [2]). For example, SAMBA [6] and AnimalScript [8] are AV scripting languages with which intermediate and advanced learners can annotate their algorithm source code in order to generate custom visualizations of their algorithms. Stasko [6] reports on his generally positive experiences with using SAMBA as the foundation for "visualization construction assignments" within an advanced college algorithms course.

To address the problems of novice learners, a legacy of research has explored novice programming environments with edit-time syntactic feedback and automatically generated visual representations of program state. By actively engaging novices in the programming process, such environments can help novice programmers to form correct mental models of program execution. For example, the HANDS environment [9], which is firmly rooted in empirical studies of how novices naturally formulate program solutions, supports a declarative programming paradigm in which program state is continuously visible. The work presented here differs from HANDS in that our goal is to help novices learn the imperative programming paradigm commonly taught in undergraduate introductory computer science courses.

Sharing that goal, ALICE [10], JELIOT [11], BlueJ [12], and RAPTOR [13] 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. With the exception of JELIOT, these systems also provide some form of edit-time feedback on syntactic correctness. ALICE, in fact, goes further by actually *preventing* syntactic errors through a novel drag-and-drop coding interface. However, while ALICE is notable for its "WhyLine" feedback mechanism, which has been shown to greatly reduce debugging effort [17], all of these environments support a similar development model in which writing code and viewing the resulting visual representation are temporally distinct activities. The "radically dynamic" editing model presented here differs from these environments in that it provides novice programmers with immediate visual feedback on their code's semantic correctness *at edit time*.

3. Motivation: Ethnographic Studies

Why might it be beneficial for an AV system to dynamically generate a visualization of code execution at *edit-time*? Our interest in exploring a "radically dynamic" AV development model was motivated by an ethnographic field study of an introductory college computer science course. As in our prior ethnographic 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.

3.1 Field Techniques

In these studies, we collected data using a variety of ethnographic 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.

3.2 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. A full treatment of our study is beyond the scope of this paper. Here, we focus on some key observations that served to motivate our interest in a more dynamic 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.

3.3 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 discovery and problem-solving. As prior empirical studies of novice programming would have predicted, we found that novices 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 novice programming environments reviewed in Section 2, 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.

4. The WYSIWYC Model

Figure 1 presents an annotated snapshot of the ALVIS LIVE! environment, in which we have implemented the WYSIWYC AV development model. Using a pseudocode-like language called "SALSA," the user can directly type commands into the Script Window on the left. The graphical results of those commands are 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 right.

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 "radically dynamic" editing, the execution arrow, which can also be moved around with the Execution Controls, 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.

4.1 A Sample Programming Session

To make this AV development model more concrete, we now step through a sample session in which we use ALVIS LIVE! to code the simple "Find Max" algorithm. The "Find Max" algorithm scans an array of numbers from left to right in search of the largest number in the list. It does this with the help of a variable `maxSoFar`, which holds the largest number that has been encountered so far.

To code this algorithm, we begin by creating the array of numbers to be searched. 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 five columns (Figure 2a). When

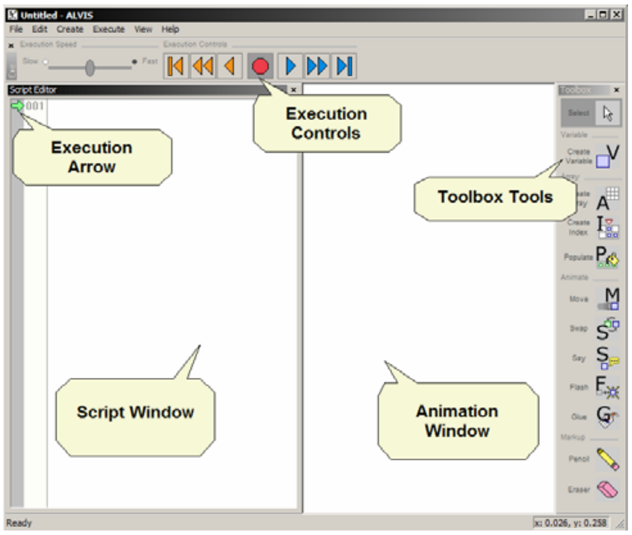


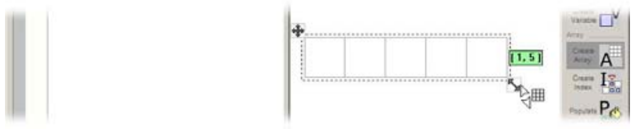
Figure 1. Annotated Snapshot of the ALVIS LIVE! Environment. "SALSA" Pseudocode can be directly typed into the Script Window. Alternatively, Toolbox tools can be used within the Animation Window to generate many commands by direct manipulation.

we release the mouse button, the SALSA code segment `create array a1 with 5 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.

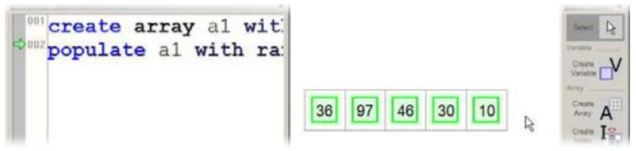
Next, we need to populate the array with numbers. To do this, we first click on the Populate tool to activate it. After clicking anywhere in the array, we see the array fill with values. Simultaneously, the SALSA code `populate a1 with random ints between 1 and 100` appears on the second line of the Script Window (Figure 2b). The execution arrow advances to this line to indicate that it has just been executed.

We now need to create the variable `maxSoFar`, which will keep track of the largest value encountered in the array as we iterate through it. Instead of generating this variable by direct manipulation, we decide to type the following code directly into the script window on the line immediately following the populate statement: `set maxSoFar to 0`. As we type in the code, it is highlighted in red to indicate that it is syntactically invalid; a dynamic tooltip containing an "up-to-the-keystroke" syntax error appears directly below the line on which we are typing (Figure 2c) However, at the instant we release the "0" key, the line becomes syntactically valid. Its red highlighting is removed, and we see a blue box that graphically represents `maxSoFar` appear in the upper left-hand area of the Animation Window. This immediate visual feedback indicates that the `maxSoFar` variable has been created and initialized to 0.

In a similar fashion, we construct an array index called current, which will allow us to iterate through the elements of array a1. We type in the following SALSA code:

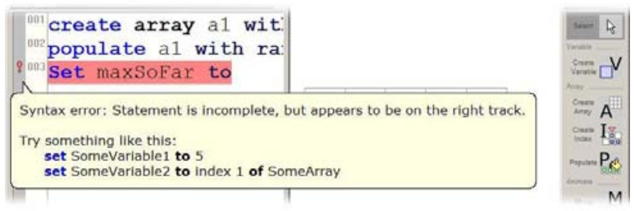


(a) Dragging out an array by direct manipulation



```
create array a1 wit
populate a1 with ra
```

(b) Populating the array with values by direct manipulation

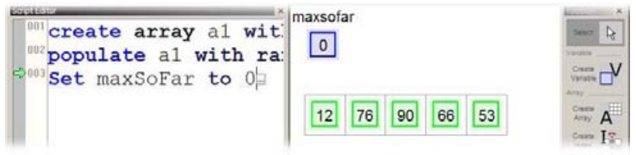


```
create array a1 wit
populate a1 with ra
Set maxSoFar to
```

Syntax error: Statement is incomplete, but appears to be on the right track.

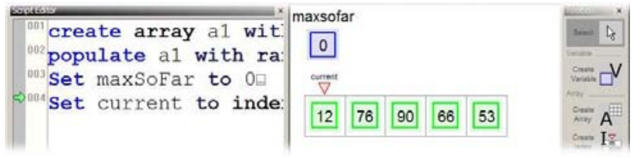
Try something like this:
set SomeVariable1 **to** 5
set SomeVariable2 **to** index 1 **of** SomeArray

(c) Typing in a (not yet valid) `set` command to create the `maxSoFar` variable; the code is evaluated on every keystroke



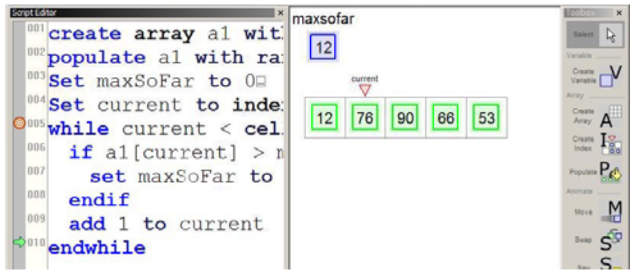
```
create array a1 wit
populate a1 with ra
Set maxSoFar to 0
```

(d) Once the "0"key is pressed, the `set` command becomes valid, and `maxSoFar` appears in the Animation Window



```
create array a1 wit
populate a1 with ra
Set maxSoFar to 0
Set current to inde
```

(e) Once the "1"key is pressed, the `current` array index appears in the Animation Window



```
create array a1 wit
populate a1 with ra
Set maxSoFar to 0
Set current to inde
while current < cel
  if a1[current] > m
    set maxSoFar to
  endif
  add 1 to current
endwhile
```

(f) As the while loop is typed in, it is dynamically executed; when it is fully typed in, one iteration of the loop is complete

Figure 2. Coding the FindMax Algorithm in ALVIS LIVE!

`set current to index 0 of a1`. The instant we release the "1" key, the statement becomes syntactically valid, and we see a pointer labeled "current" appear above cell 0 of `a1` (Figure2e).

The trickiest part of the code is the loop that iterates through the elements of array `a1`. Positioning the cursor on the next line, we type in the following SALSA code:

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

As we type in each line of the code block above, ALVIS eagerly evaluates it on every keystroke, coloring it in red and reporting the current syntax error until it becomes syntactically valid. Since the `while` and `if` statements are control statements, they do not produce any explicit visual feedback in the Animation Window. In contrast, when the set statement becomes valid, we see the value of `maxSoFar` change in the Animation Window. Likewise, when the add statement becomes valid, we see the current arrow slide one cell to the right in the Animation Window (Figure 2f).

To explore the execution of our code further, we use the Execution Controls to step through our code, one line at a time. As we do so, the execution arrow advances, and the Animation Window is dynamically updated to reflect the execution results. We can also execute to any point in the script simply by clicking on that line with the mouse.

5. Empirical Evaluation

Recall that development of the WYSIWYC model was 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? As an initial step toward addressing that question, and in order to evaluate and further 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 tweaked in response to the usability issues that arose. Nonetheless, the basic WYSIWYC model presented in the previous section remained intact.

5.1 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 (see Section 4), 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 that elicited their subjective ratings of ALVIS with respect to several usability categories. Sessions typically lasted between 90 and 105 minutes.

5.2 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. However, our analysis also revealed some problems with the WYSIWYC model. Below, we first present evidence of the potential value of WYSIWYC'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.

5.2.1 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.

None of the 14 participant groups noticed the bug when they first typed in the buggy line of code, most likely because they were focused on following task instructions, rather than on looking for bugs. However, once they were informed that a bug existed in the code, participants needed an average of just 89 seconds (*sd* = 61 sec) to identify the location of the bug. Moreover, from the time they identified the location of the bug, participants needed an average of just 72 seconds (*sd* = 71 sec) more to actually fix the bug.

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."

With respect to the WYSIWYC model's ability to help novices make coding progress without inordinate delays, we were encouraged by the fact that 13 of the 14 participant groups developed a correct algorithmic solution for the "Replace Zeroes" task in an average time of 25 minutes and 41 seconds (*sd* = 13 min, 13 sec). 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 "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 ALVIS LIVE!. 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 quick reference guide, 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.

5.2.2 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 being more serious.

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, 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 [10].

Hypothetical execution. In order to keep the Script Window and Animation Window continuously synchronized, the WYSIWYC model requires that every line of code be executed as it is edited. This property of the model introduces a knotty problem: How can a programmer type in the body of a conditional statement (e.g., *if*, *while*) whose condition initially evaluates to false? In order to allow the user to type in the body of an arbitrary conditional, the version of ALVIS LIVE! used in our usability study eagerly evaluated conditional bodies even if they should not have been reached. As one might imagine, this behavior led to program executions in which the Animation Window reflected invalid program states.

We regard *hypothetical execution* states, in which the user edits presently unreachable code, as the most challenging problem faced by the WYSIWYC model. However, we are confident that, with additional iterations of our design cycle, a satisfactory solution can be found. In a solution that we are presently testing, we simply suspend immediate execution when the user edits a block of unreachable code; immediate execution resumes on the line immediately below the unreachable block of code. We believe that temporarily losing immediate feedback may be a fair price to pay for maintaining a valid execution state.

6. Summary and Future Work

As we have shown, the development model supported by existing novice programming and visualization environments does not provide edit-time execution feedback. Because novices in particular are known to lack robust mental models of how code executes, we have argued that this lack of edit-time feedback can lead to gulfs of evaluation that inhibit programming progress. Motivated by an ethnographic study of novice algorithmic problem solving and visualization, we have presented a solution to this problem: an alternative AV development model that provides, on an edit-by-edit basis, immediate visual feedback on the syntactic and semantic correctness of code. An analysis of usability study data suggests that our model's dynamic visual feedback has the potential to aid novice programmers in understanding their code and quickly identifying and remedying programming errors.

In addition to exploring solutions to the problems with the WYSIWYC model discussed above, further research is needed in order to evaluate the WYSIWYC model more rigorously. To that end, over the past three months, we have conducted two larger scale empirical studies:

- an *experimental study* (*n* = 60) in which five treatment groups coded algorithmic solutions using either a simple text editor (the control group) or three competing versions of ALVIS LIVE! that provided differing levels of edit-time feedback; and
- a follow-up *ethnographic study* (*n* = 53) in which students, working in pairs, used ALVIS LIVE! to complete a "studio experience" lab identical to the lab students completed with a text editor and art supplies in the Fall of 2004 (see Section 3 above). In this study, students used many of the additional animation features of ALVIS LIVE! to develop custom visualizations of their algorithms for presentation in front of the class.

We have recently completed the data collection phase of these two studies; however, our analysis has only just begun, and will require several months to complete, owing to the more than 160 hours of video and thousands of lines of source code that need to be considered. We look forward to reporting the results of these studies in future publications.

7. Acknowledgments

Sean Farley and Sudhir Garg helped to program 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, and J. T. Stasko, "A meta-study of algorithm visualization effectiveness," *Journal of Visual Languages and Computing 13* (3), 2002, pp. 259-290.

[2] T. L. Naps, G. Roessling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. Mchally, S. Rodger, and J. A. Valazquez-Iturbide, "ITiCSE 2002 working group report: Exploring the role of visualization and engagement in computer science education," *SIGCSE Bulletin 35* (2), 2003, pp. 131-152.

[3] J. Stasko, A. Badre, and C. Lewis, "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, 1993, pp. 61-66.

[4] M. D. Byrne, R. Catrambone, and J. T. Stasko, "Evaluating animations as student aids in learning computer algorithms," *Computers & Education 33* (4), 1999, pp. 253-278.

[5] A. W. Lawrence, A. N. Badre, and J. T. Stasko, "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, 1994, pp. 48-54.

[6] J. T. Stasko, "Using student-built animations as learning aids," in *Proc. ACM SIGCSE 1997*. ACM Press, New York, 1997, pp. 25-29.

[7] C. D. Hundhausen, "Integrating Algorithm Visualization Technology into an Undergraduate Algorithms Course: Ethnographic Studies of a Social Constructivist Approach," *Computers & Education 39* (3), 2002, pp. 237-260.

[8] G. Rößling and B. Freisleben, "AnimalScript: An Extensible Scripting Language for Algorithm Animation.," in *Proc. ACM 2001 SIGCSE Symposium*. ACM Press, New York, 2001, pp. 70-74.

[9] J. F. Pane, B. A. Myers, and L. B. Miller, "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, 2002, pp. 198-206.

[10] W. Dann, S. Cooper, and R. Pausch, "Making the connection: Programming with animated small world," in *Proc. ITiCSE 2000*. ACM Press, New York, 2000, pp. 41-44.

[11] R. Ben-Bassat Levy, M. Ben-Ari, and P. Uronen, "The Jeliot 2000 program animatino system," *Computers & Education 40* (1), 2003, pp. 1-15.

[12] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The BlueJ system and its pedagogy," *Journal of Compuer Science Education 13* (4), 2003, pp. 249-268.

[13] M. Carlisle, T. Wilson, J. Humphrieis, and S. Hadfield, "RAPTOR: A visual programming environment for teaching algorithmic problem solving," in *Proc. ACM SIGCSE 2005 Symposium*. ACM Press, New York, 2005.

[14] J. Bonar and E. Soloway, "Preprogramming knowledge: A major source of misconceptions in novice programmers," in *Studying the Novice Programmer*, vol. 325-353, E. Soloway and J. Spohrer, Eds. Lawrence Erlbaum, Hillsdale, NJ, 1985, pp. 133-161.

[15] D. A. Norman, *The Design of Everyday Things*. Doubleday, New York, 1990.

[16] C. D. Hundhausen and S. A. Douglas, "Low fidelity algorithm visualization," *Journal of Visual Languages and Computing 13* (5), 2002, pp. 449-470.

[17] A. Ko and B. Myers, "Designing the Whyline: A debugging interface for asking questions about program failures," in *Proc. ACM SIGCHI 2004*. ACM Press, New York, 2004, pp. 151-158.