THE EFFECT OF CORRESPONDENCE HIGHLIGHTING ON NOVICE PROGRAMMER INSTRUCTION

By

COLE NEVINS

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2009

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of COLE NEVINS find it satisfactory and recommend that it be accepted.

Christopher Hundausen, Ph.D., Chair

Robert Patterson, Ph.D.

Roger Alexander, Ph.D.

ACKNOWLEDGEMENTS

The author would like to gratefully acknowledge the encouragement, advice and assistance of Jon Brown of the Department of Electrical Engineering and Computer Science, who also designed and carried out the statistical analyses presented in Chapter 3 and 4.

THE EFFECT OF CORRESPONDENCE HIGHLIGHTING ON NOVICE PROGRAMMER INSTRUCTION

Abstract

By Cole Nevins, M.S.
Washington State University
May 2009

Chair: Christopher Hundhausen

Many novice programming environments utilize a dual-representation interface to display code textually and graphically. None has explored the design implications of Dual-Coding and Cognitive Load Theories, which offer significant insight into the benefits and pitfalls of presenting novice programmers with two separate views of program code. To address this gap, I have surveyed the existing Dual-Coding and Cognitive Load literature and identified a promising design modification to dual-representation novice programming interfaces: *correspondence highlighting*. I implemented correspondence highlighting in the ALVIS LIVE! novice programming environment. When an element in the code window of the ALVIS LIVE! environment is selected, ALVIS LIVE! highlights that element and its corresponding element in the animation window is highlighted in a color unique to that element. When an element in the animation window is selected, that element and every reference to that element in the code window highlighted. In an experimental study, this highlighting mechanism failed to provide a significant performance advantage over a version of ALVIS LIVE! without correspondence highlighting. I examine why this might be the case, and propose directions for further research into the effect of cognitive load on novice programmer performance, as well as opportunities for improving novice programming environments at every stage of learner experience.

TABLE OF CONTENTS

Chapter

LIST OF TABLES

LIST OF FIGURES

Dedication

This thesis is dedicated to my grandparents, Raimund and Agnes Rainer.

Yes Opa, the book is finally done.

CHAPTER 1

INTRODUCTION

For many wishing to pursue a computer science degree in the United States, their first courses are their last. Once enrolled in a computer science program, students face a trend of attrition in compute science programs (Howles, 2007; Beaubouef & Mason, 2005; Hundhausen, Farley, & Brown, 2006; Cohoon & Chen, 2003), but this may only be a symptom of a deeper problem. The ostensible difficulty new computer science students typically face may have a chilling effect on enrollment numbers for computer science programs; a yearly survey of approximately 185 Computer Science and Computer Engineering Departments in the United States and Canada shows a nearly 40% decline in total bachelor's degrees awarded since 2004. Current enrollment in undergraduate Computer Science programs is roughly 50% lower than in 2002 (Zweben, 2008), while the need for Software engineers is projected to increase by roughly 38% in the next ten years (Bureau of Labor Statistics, 2009). A continued decline in computer science bachelor’s degree production could present a myriad of United States industries – technical, medical, and engineering, to name a few – with a significant shortfall of engineers in the near future.

Possible Explanations

Several proposed factors that may contribute to CS student attrition include variability in the incoming student population with respect to prior programming experience, self-efficacy, and gender expectations (Ramalingam, LaBelle, & Wiedenbeck, 2004; Wiedenbeck, 2005; Byrne & Lyons, 2001). In addition to these internal factors, others have proposed pedagogical issues like lack of cohort community, inappropriate curricular advancement, and unsupportive novice programming environments as major confounding elements in undergraduate CS education (Beaubouef & Mason, 2005; Wilson & Shrock, 2001; Kelleher & Pausch, 2005; Hundhausen et al., 2006).

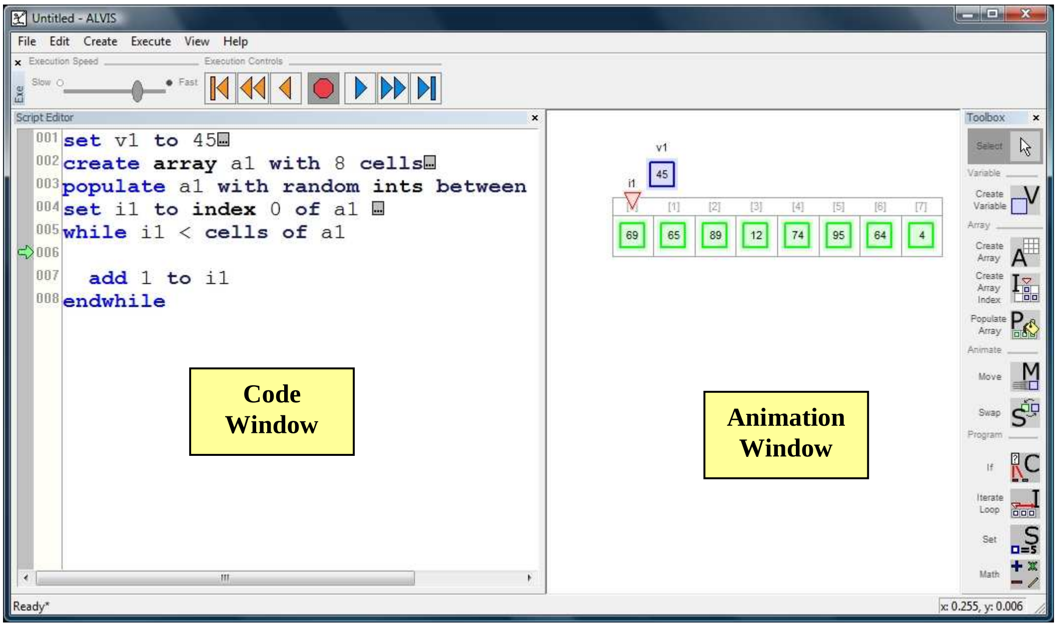
While there are several lines of research dealing with these pedagogical concerns, I view the development of an effective novice programming environment as particularly important. A novice programming environment is a key meeting ground for learning computer programmers—students collaborate and discuss using the environment's text and graphics as a shared representation. Thus the programming environment can be a mechanism for positively influencing other pedagogical concerns like community and can be used in a variety of curricular and educational situations. With this interest, the focus of this thesis is the improvement of novice programming environments.

Novice Environments

Novice programming environments attempt to help new programmers progress in generating and interpreting programming constructs and more complex programs by exploiting the principal of educational *scaffolding* (Soloway, Jackson, Klein, Quintana, Reed, & Spitulnik, 1996). In the context of software-based learning environments, *scaffolding* is a means of supporting the exploration and completion of learning tasks by allowing the environment to handle complex, non-essential, or lower-level tasks for the learner, allowing the learner to concentrate on higher-level construction and interpretation exercises (Soloway et al., 1996).

ALVIS Live! (Hundhausen et a., 2006) is one of many examples of novice programming environments that support scaffolding. ALVIS LIVE!, which supports a "live" execution model and code creation through direct manipulation, has proved to be a useful test-bed for evaluating novice interface elements (Hundhausen et al., 2006). In ALVIS LIVE!, learners have the option of programming algorithms textually and seeing the data structures and values represented in an accompanying animation window, or dragging and dropping (“directly manipulating”) elements in the animation window to generate code in the code window (see Figure 1). Past studies of ALVIS LIVE! showed that participants who were trained to program by generating code using only the Animation window outperformed those who generated code by using only the Code window, even though both groups could view their program in either representation (Hundhausen et al., 2006).

Given the preponderance of split-paned novice programming environments, I theorized that these environments might be improved by the ability to visually emphasize the correspondence between elements in different panes—what one might call *correspondence highlighting*. A brief examination of the theoretical justification for this interface modification follows; a more comprehensive review of existing work in this area is presented in Chapter 2.



```
set v1 to 45
create array a1 with 8 cells
populate a1 with random ints between
set i1 to index 0 of a1
while i1 < cells of a1

  add 1 to i1
endwhile
```

Figure 1. Example program in the ALIVS Live! Novice Programming environment.

Cognitive Theories Relevant to Dual-Representation Novice Programming Environments

The development of scaffolding environments is often influenced by one or more theoretical cognitive frameworks. Two key theories taken into account in this paper are *dual-coding* and *cognitive load theory*, which will be outlined briefly below, and further examined in Chapter 2.

Introduced by Paivio (1986) and supported by Baddeley's (2001) working memory model, dual-coding theory asserts that human cognitive functions are split between symbolic (diagrams) and language (textual) subsystems. Presenting the same information in both forms improves recall by simultaneously activating two different interconnected representations in working memory during the encoding process, improving long-term retention of details and concepts. Various empirical explorations of this theory in the context of multimedia learning (Mayer, 2001; Mayer, 1981; Mayer, 2003) largely support this theory, and ALVIS LIVE!'s interface, which shows textual and graphical representations of the user's program, also takes advantage of this effect.

In contrast, a dual representation may increase the learner's *cognitive load.* Sweller, Merrienboer, & Paas (1998) pointed out that the cognitive processing load imposed by material can be divided into three types of load: 1) *intrinsic load*, which is essential to learning the material and in educational contexts a focal element of the activity; 2) *germane load*, which is not intrinsic to the material but which facilitates learning and 3) *extrinsic load*, extraneous load imposed by the way in which the material is presented and which should be reduced or removed.

ALVIS LIVE!, like several other novice programming environments reviewed in Chapter 2, uses a two-paned interface containing text (the code window) and diagrams (the graphical interface) to allow the novice to interact with the same information (the program) in multiple ways (Hundhausen et al., 2006). While this interface can be more effective than a single representation according to dual-coding theory (Paivio, 1986), it may also suffer from a split-attention effect (Ginns, 2006), in which more working memory is required on the part of the learner to integrate two physically separated, heterogeneous representations into an understandable whole.

Research Questions

While dual-coding and cognitive load theories provide evidence that learners face unique challenges while integrating information presented in a dual-representation interface, many novice programming environments that utilize separate representations fail to provide explicit connections between these two types of representations. I theorized that* if the ALVIS LIVE! environment (Figure 1) were modified to dynamically highlight the corresponding visual and textual representation of the currently selected or edited program element, it would lower the extraneous cognitive load by mitigating the split-attention effect described by Ginns (2006). This might more tightly couple the textual and graphical representations, and draw the user's attention to the corresponding variables and program state represented in both the Text and Direct Manipulation windows of the ALVIS LIVE! interface. As students engage in basic programming tasks with this new environment, one might expect that the lower extraneous cognitive load would allow students to devote more cognitive resources to understanding and constructing algorithmic solutions. This led to the central research question of this thesis:

RQ1: Can correspondence highlighting, when added to the ALVIS LIVE! interface, more tightly integrate the Text and Animation windows and positively impact novice programmer performance?

To address this question, I augmented the existing implementation of ALVIS LIVE! to highlight elements being edited by the programmer in both representations. I then conducted an experimental study on the new interface with novice programmers from Washington State University's Computer Science 121 class as participants. While this study failed to detect a significant advantage provided by correspondence highlighting as implemented in the ALVIS LIVE! environment, this thesis provides a theoretically-grounded exploration of correspondence highlighting within the design space of dual-representation novice programming environments, as well as a framework for future empirical approaches to examining ways of improving dual-representation environments.

Thesis Outline

Chapter 2 establishes the context in which ALVIS LIVE! development took place by reviewing key theoretical perspectives as well as technological predecessors of the ALVIS LIVE! novice programming environment. Next, Chapter 3 details the design and development of the correspondence highlighting feature in the ALVIS LIVE! Environment. Chapter 4 outlines the experimental procedure employed to test the focal research question, and Chapter 5 relates the results of the experiment. Finally, section 6 summarizes implications of these results and identifies areas of further research.

CHAPTER 2

RELATED WORK

Over the past 20 years, many lines of research have focused on building more effective programming environments for novices. The results of these efforts can broadly be categorized as 1) environments that contain a somewhat concrete virtual world—a *microworld* in which the execution of the code can be visualized and displayed to the user (often via a character or story-based animation), or 2) environments that offer more abstract code visualization, possibly supporting direct interaction with the visualization. Some of these latter code-visualization environments use a single, novel interface for defining program structure and behavior, while others use a *dual-representation programming interface*—two or more simultaneous representations and interfaces for interacting with one program. Most fall somewhere between these two extremes.

Novice Environments[1]

With the introduction of affordable Lego Mindstorm robot kits aimed at K-12 students, computer science educators were presented with a powerful platform for teaching computer science subjects like artificial intelligence (Klassner, 2002), Java programming (Barnes, 2002), concurrency (Jacobsen & Jadud, 2005) and networks (Klassner & Anderson, 2003). Not only did these kits offer students a concrete, persistent

[1] For those interested in a broader picture of novice programming environments and their attendant concerns, (Kelleher & Pausch, 2005) presents an excellent survey of existing novice environments.

Figure 2. INVENTOR interface for Lego Mindstorms

representation of the program state (the robot and its activities), but the Lego NXT ™ development software possessed an interface that displayed diagrammatic representations of program commands and flow integrated with explanatory labels and tool-tips (Figure 2). Students were also able to use a powerful direct-manipulation interface to interact with this representation to graphically create program commands as well as manage memory and computational resources (Sharad, 2007)

Raptor (Carlisle, 2009) allows students to develop algorithmic solutions by constructing and manipulating flowchart symbols to create and execute procedural programs. The student may use flowchart symbols to specify assignment, selection, loop, call, input, and output commands. Unlike other popular interfaces, Raptor does not present both diagrammatic and textual representations of the program; students interact primarily with the flowchart display while constructing, running, and debugging programs (Carlisle, Wilson, Humphries, Hadfield, 2005). One preliminary multi-year examination of Raptor's effectiveness compared student performance with Raptor to those that were taught using MATLAB or Ada in an introductory computer science course. They found that students taught procedural programming with Raptor performed

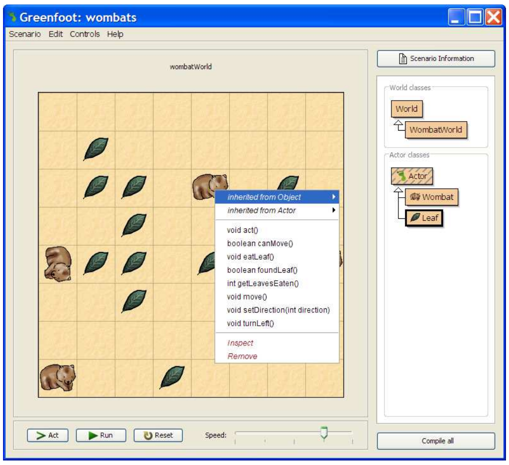


Figure 3. The Greenfoot interface. Greenfoot allows students to interact with objects directly – in this case, “leaf” and “wombat” objects instantiated in the world.

significantly better than those that didn’t use Raptor on two out of three programming questions on the final exam (Carlisle et al., 2005). However, another study comparing Raptor to commercial flowcharting software did not reveal significant differences in student performance with Raptor (Giordano & Carlisle, 2006).

Other novice programming environments with varying levels of separate representations abound. Greenfoot (Figure 3) is a novice programming environment that attempts to combine the programming supports of environments like BlueJ with the concrete representation of objects in a “microworld” (Henriksen & Kölling, 2004). To do this, it supports the instantiation and modification of objects via direct manipulation of elements in the world as well as the ability to test object methods and behavior via direct manipulation (Kolling, 2009). Teachers and students may create and modify objects and main program code via a Java editor.

While the environments mentioned above utilize an intermediate target language integrated with the diagrammatic representations, BlueJ attempts to help novice programmers simultaneously deal with the syntactical and conceptual challenges of object oriented programming in the Java programming language. To do this, it provides separate graphical representations in addition to a more conventional Java code window. As students construct Java classes in the text window, an integrated UML display (Figure 4) shows the inheritance relationships between classes (Bluej.org, 2009). BlueJ also allows students to instantiate objects on a separate window depicting the "object bench", visually examining an object's state and values by directly interacting with the object diagrams (Kouznetsova, 2007). This allows for immediate testing of objects without writing specific test drivers, and is a potential boon for instructors wishing to introduce novice programmers to proper testing methodologies (Patterson, Kölling, & Rosenberg, 2003). While a quantitative assessment of BlueJ's pedagogical effectiveness has yet to be published, two qualitative studies show several limitations of the interface in an Objects-First teaching context. Ragonis and Ben-Ari (2007) note that BlueJ's support for direct interaction with objects via the separate object bench window may have prevented

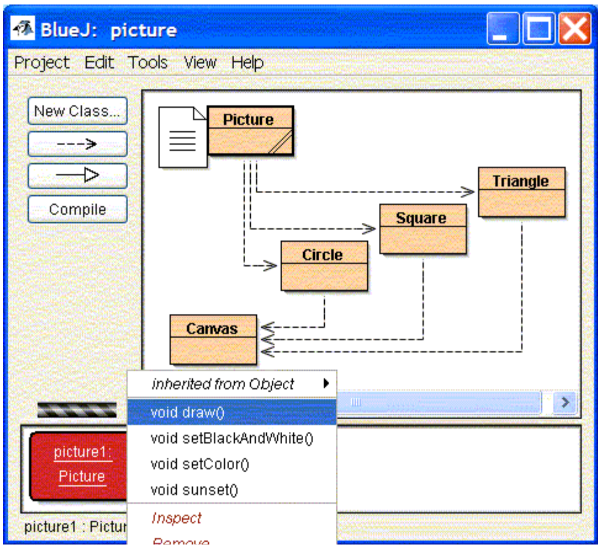


Figure 4. Example of BlueJ's UML view

students from seeing objects in the greater context of program execution, leading to problems conceptualizing object state, method invocation, parameters, return values, and constructors. Xinogalos et al. (2007) compare the performance of students taught with BlueJ across two years, noting the improved performance of students when instructors introduced the main method earlier in the course while grounding the object-oriented features of BlueJ in a larger coding context separate from the environment. Both of these reports conclude that while BlueJ is a powerful pedagogical tool, it must be thoughtfully integrated into a curriculum that balances procedural and object-oriented concepts.

Alice 2.0 (Alice.org, 2009) offers another dual-representation approach to object-oriented programming education. Like the Lego Mindstorm systems, Alice was initially targeted toward K-12 use but has been adopted by university level educators in recent years. As described in (Cooper, Dann, & Pausch, 2000; Mullins, Whitfield, & Conlon, 2009; Mullins & Conlon, 2008), Alice allows students to create three-dimensional animations and interactive environments in a separate “world” window through the use of a constrained drag-and-drop code interface as well as an interactive 3D Scene Window. This window’s character and scene paradigm lends itself to teaching object-oriented concepts, with support for object-level methods and variables (Lorenzen & Sattar, 2008; Mullins & Conlon, 2008) in the corresponding code window. Users may instantiate “objects”, (represented as characters, items, and buildings in the scene) via direct manipulation in the scene window, and may also specify fairly sophisticated behavior for these objects via the drag-and-drop code window interface that supports branching, iterative, and recursive structures as well as event handling and typed variables (Mullins et al., 2009; Mullins & Conlon, 2008). This code window is physically separated from the Scene Window, and thereby potentially vulnerable to split-attention effects. It is also a significantly constrained interface – it prevents students from dropping code snippets in places that could cause the program to not work, and thereby eliminates problems of syntax and related compile-time errors (Brown, 2008; Mullins & Conlon, 2008).

Like many novice environments, the effectiveness of Alice as a teaching tool has not been extensively studied. One notable exception is Moskal, Lurie, and Cooper's (2004) empirical examination of how at-risk students, those with no prior programming experience and poor math-readiness, perform with Alice as a teaching tool. They found that the use of Alice in introductory computer science courses improved grades, retention rates, and attitudes of at-risk computer science students. Additionally, a preliminary multi-year examination of classroom performance data indicated that the use of Alice (compared to Java) as part of an introductory CS class improved retention as well as increased the percentage of passing students (Mullins et al., 2009).

While few empirical evaluations of Alice exist, several case studies provide anecdotal evidence of the benefits of Alice. For example, Mullins et al. (2009) and Brown (2008) observed that Alice's concrete representation of objects facilitated the rapid teaching of a traditionally difficult concept like recursion. Furthermore, Alice-trained students reported experiencing less frustration with syntactical and logic errors, and were more motivated to expand on programming solutions (Brown, 2008). Lorenzen and Sattar (2008) describe the successful integration of Alice into the early weeks of a Java course to introduce object-oriented programming concepts. From a motivational perspective, Kelleher, Pausch, & Kiesler (2007) found that middle school girls who used a version of Alice that facilitated concrete storytelling spent more working on their programming assignments and were more likely to spend extra time on their programming assignments than girls who used a "generic" version of Alice. They posit that Alice's potential to facilitate storytelling may increase engagement and motivation in early computer science education.

Unfortunately, the some of the very benefits of Alice as a pedagogical tool in early computer science education may cause problems as students transition to a more traditional production language: Mullins et al. (2009) and Brown (2008) note anecdotally that students trained in Alice's syntax-free code interface had difficulty transferring that training to a subsequent Java-based computer course. Furthermore, while it appears ideal for introducing object-oriented concepts, Alice's lack of support for true encapsulation and inheritance can cause problems for students progressing onto more complete and abstract C++ or Java based object-oriented programming material (Mullins et al., 2009; Powers, Ecott, & Hirshfield, 2007).

The ALgorithm VIsualization Storyboarder (ALVIS LIVE!) is a "live" dual-representation novice programming environment which doubles as an effective test bed for investigating how low-fidelity algorithm visualization, guided visual direct manipulation editing, and dynamic interpretation of code impact learning outcomes (Hundhausen & Brown, 2005; Hundhausen et al., 2006). One component of this research effort implemented these features and subjected them to extensive usability testing (Hundhausen & Douglas, 2000; Hundhausen & Brown, 2005; Farley, 2006); another examined the effects of visual direct manipulation editing and dynamic interpretation of code in several empirical tests (Farley, 2006; Hundhausen et al., 2006).

These last studies have produced some evidence that programming tool features such as those used in ALVIS LIVE! may support students who lack previous programming experience. Specifically, these studies showed that students using ALVIS LIVE! were able to develop algorithmic solutions with significantly greater speed and accuracy than their counterparts who used a basic text editor to solve the same problems (Farley, 2006; Hundhausen et al., 2006).

Finally, Hundhausen et al. (2006) showed that learning that occurred during training with direct-manipulation in the ALVIS LIVE! interface transferred positively to text-only programming.

In an unpublished follow-up study, the ALVIS LIVE! team replaced the pseudo-code language SALSA with a C-like programming language called C-Flat (C*b*) and repeated the protocol reported in the previous study (Hundhausen et al., 2006). Tests of the altered environment yielded results that conflict with previous data, indicating that the positive transfer effects did not occur when the C*b* language was substituted for the SALSA language (See Chapter 3 for further discussion). An examination of video recorded during these sessions revealed several sets of similar programming errors, which I describe and categorize in Chapter 3.

As the above review of novice interfaces demonstrates, anecdotal support for the efficacy of many of these environments exists, yet experimental or quantitative support is sparse and sometimes contradictory. While there are many novice interfaces that utilize dual-representations in some manner, none explicitly reference two particularly relevant theoretical frameworks that may give us insight into the benefits and dangers of dual-representations: *Dual Coding*, and *Cognitive Load Theory*.

Dual Coding

Originally proposed by Paivio (1986), dual coding asserts that humans posses two different cognitive systems for encoding information: one for the processing of nonverbal objects, events, and representations, and another for processing language (verbal) information. These systems (or “pathways”) are largely independent, but information coded in one system can be activated by the retrieval of information stored in the other (Clark & Paivio, 1991). Thus, presenting a viewer with information in both verbal and visual representations activates different sections of the brain, improving recall (Paivio, 1975; Paivio & Lambert, 1981). A line of subsequent experimental evaluations of dual coding theory, surveyed by Paivio (1983), has provided empirical support for this theory.

Richard Mayer and colleagues have continued to verify and expand this line of inquiry. For example, Mayer and Anderson (1991) found that combining animation and narration resulted in improved problem solving ability. They presented participants with little prior mechanical knowledge several different configurations animation and narration describing the function of a bicycle pump. Novices who were presented with animation while hearing a description of bicycle pump function performed better on a post-instruction problem-solving transfer test than those that viewed animation only, words only, and narration and animation separately. This line of research has informed the development of design principles which attempt to take advantage of these cognitive mechanisms.

Mayer and Moreno (1998) list these principles, which include 1) presenting an explanation in diagrams and words is superior to using just a verbal or textual explanation, 2) presenting corresponding pictures and words simultaneously is more effective than presenting them separately, 3) presenting words as auditory rather than text while viewing information is superior, and 4) the beneficial effects of combined visual and verbal dual representations are largely confined to students with little experience with the material being taught.[2]

[2] For those interested, Mayer (2003) surveys and lists further empirical support for each principle. In his book *Multimedia Learning*, he also discusses the pedagogical implications and applications of these principles in the greater context of multimedia learning (Mayer, 2001).

Furthermore, visual representations may have significant value to the learner in their own right. As Larkin and Simon (1987) pointed out, well-constructed diagrammatic (visual) representations may have certain inherent advantages over equivalent sentential (verbal/textual) descriptions of the same system. After constructing both diagrammatic and sentential representations of various physical and geometric systems, they analyzed these representations in terms of the cost of search, recognition, and inference, concluding that properly constructed diagrams can make these tasks much more efficient ("zero-cost").

Cognitive Load Theory

While Paivio and Mayer have both provided evidence for the pedagogical value of multiple representations, Cognitive Load Theory (CLT) draws attention to the cognitive cost (load) that representations impose during complex cognitive tasks. Before moving on, I should define two concepts integral to a discussion of cognitive load: *working memory* and *long-term memory*.

It is generally agreed that human working memory—the place in the brain where information is temporarily stored and manipulated—is relatively constrained (Baddeley, 2001; Baddeley, 1992; Sweller, Merrienboer, & Paas, 1998). Representations of systems with a large number of interacting elements require a certain amount of working memory resources on the part of the learner—the cognitive load—in order to be understood and learned (Sweller et al., 1998; Pass, Renkl, Sweller, 2003). CLT divides this cognitive load into three key categories: *intrinsic* load, *extraneous* load, and *germane* load.

Intrinsic load represents the amount of working memory required that is intrinsic to the material being learned. In a programming context, the amount of working memory required to comprehend an entire method or function of code is the intrinsic load of that method or function.

This load cannot be reduced without removing important elements of the system being presented to the learner, or by drawing on pre-existing *schemas* possessed by the learner. Schemas are constructs that allow the learner to reference large amounts of information in long-term memory, an area of the human brain dedicated to the persistent storage of a vast quantity of memories (Sweller et al., 1998). A schema may reference long-term memory from a single construct in working memory (Chi, Glaser, & Rees, 1981). Thus, as the learner brings more schemas to bear while learning a particular system, the cognitive load required to process the whole system is lowered. More experienced programmers might have a schema for control statements that allows them to focus on the conditionals (A > 10) of the statement rather than the form (IF, THEN, ELSE) of the statement, effectively creating one concept out of the several statements required to define an IF statement.

Germane load is cognitive load directed toward a learner’s schema development, and therefore useful. This kind of load can be fostered by the designer of the instructional framework; in a programming context this might involve the insertion of inline "comments" describing a block of code in higher-level terms.

Extraneous load is the extra cognitive load imposed by the teaching method or presentation. It is not directly useful to learning, and can be viewed as the “overhead” of whatever means of presenting the material is used. For example, presenting a novice programmer with a program with cryptic variable names and poor commenting could be an example of extraneous load; these additions to the program are unnecessary and require additional effort to decipher.

In order for learning to occur, the total of these three types of cognitive load must not exceed the learner's capacity, which can vary with experience and motivation (Paas, Tuovinen, Merriënboer, & Darabi, 2005). Also, in order for germane load to be generated and schema acquisition (learning) to take place, learners must be presented with realistically complex cognition tasks (Merriënboer, Kester, & Paas, 2006). Efforts at lowering cognitive load largely have the goal of reducing extraneous cognitive load and using the freed load to foster germane load and therefore learning, or at least bringing total load of the learning task within the capabilities of the learner (Merriënboer & Sweller, 2005). In addition to reducing extraneous cognitive load, efforts at reducing *intrinsic* cognitive load have focused on the separation of various components of the system and presenting them to the learner as whole units that may be learned and then integrated into a larger picture of the system (Pollock, Chandler, & Sweller, 2002; Wouters, Paas, & Merriënboer, 2008).

Furthermore, research on the extent of the effects of cognitive load has shown an Expertise Reversal Effect. Essentially, efforts to reduce cognitive load for novice learners can have a negative impact on more experienced learners (Sweller et al., 1998). For example, at some point, increasing the amount of commenting and other cognitive aids makes comprehending code *more* difficult for more experienced learners (Yeung, Jin, & Sweller, 1998). For a survey of research on the expertise reversal effect, see (Kalyuga, Ayres, Chandler, & Sweller, 2003).

Extraneous cognitive load can come from several sources. These sources can include 1) mean-ends analysis, in which the learner is required to remember the beginning, end, and intermediate states of a problem while trying to solve it (Sweller, 1988) 2) split attention, in which the learner must integrate two different, physically separated sources of information (Tarmizi & Sweller, 1988), and 3) temporal dis-contiguity, where the learner must combine two sources of information that are seen some time apart (Ginns, 2006).

Sweller et al. (1998) presents an overview of some common techniques for reducing some of these kinds of cognitive load, while fostering germane load, in a greater pedagogical context. These guidelines include 1) providing a learner with problems that do not contain an end goal ("goal-free") and instead asking for the results of intermediate calculations. 2) Providing partially-worked examples of problems and allowing the learner to complete the problems and 3) presenting the learner with partially-worked problems in a semi-random order. While these guidelines may be applied in the greater context of curriculum design and instructional order in order to reduce cognitive load from means-end analysis, they do not address two of the other sources of extraneous cognitive load particularly applicable to novice programming interfaces: the split-attention and temporal contiguity effects.

Split attention refers to the fact that physically separated representations (like a diagram and explanatory text) must be integrated by the learner before either representation makes sense (Sweller et al., 1998). This integration task increases cognitive load, as the learner is required to hold one representation in memory while searching for corresponding parts in the other representation before learning can even begin (Ginns, 2006). Tarmizi and Sweller (1998) produced a well-known study examining the effect of integrated and split geometry worked-examples, noting that students who used integrated geometric examples outperformed those who used the more typical examples that contained geometric diagrams separate from the corresponding explanation.

These findings have been supported in a variety of contexts. In simple mathematical inference tasks, middle-school aged children presented with integrated sources of problem information outperformed classmates presented with split sources of information (Mwangi & Sweller, 1998). A reading comprehension study spanning children ages 5-18 as well as children with high and low English capability also noted the increase in cognitive load (and corresponding lower comprehension scores) for novice learners presented with vocabulary definitions and explanatory information separate from the text (Yeung, Jin, & Sweller, 1998).

Temporal contiguity can be considered to be similar to split-attention, but in the time dimension as opposed to the spatial dimension. Much like integrating spatially separated information sources, the task of integrating information sources separated by time requires that the learner hold one representation in memory while searching for connections to subsequent information, which takes cognitive resources away from actually learning the material (Mayer, 2001). From an animation perspective, Wouters, Paas, and Merriënboer (2008) summarize several empirical studies which indicate that animations of high-complexity may actually increase cognitive load, since the viewer must track several changing objects in two dimensions while also attending to the introduction and removal of new and old objects. For those interested in a larger picture of split-attention and temporal contiguity research, Ginns (2006) has an excellent meta-analysis of the effect size of 37 empirical split-attention and 13 temporal contiguity studies.

Although CLT has yet to be explicitly applied to the development of novice programming interfaces, it may prove to be particularly relevant to this space. A number of the interfaces noted in the review at the beginning of this chapter feature a diagrammatic component (representing program structure or state), which may or may not be integrated with a separate source of information (a code window). In many cases, these diagrammatic representations may be directly manipulated by the user in order to create or edit the algorithm (code) solution. Furthermore, these representations may change (animate) during runtime. Thus, a thoughtful examination of how these two theories of learning could interact may give us insights into managing this interaction and improving the efficacy of dual-representation novice programming environments.

According to dual-coding theory, dual representations may significantly assist learner retention and comprehension of material (Paivio, 1987; Paivio & Lambert, 1981). However, the benefits of dual-representations come at a cost—the split-attention effect caused by these spatially separated representations increase cognitive load. One line of research aimed at mitigating split-attention effects focuses on integrating disparate representations, placing both sources of information in the same space (Kablan & Erden, 2008; Chandler & Sweller, 1991).

Another line of research has looked at the effects of signaling on spatially separated representations, which might have significant bearing on novice programming environments. Signaling denotes the technique of offering viewer visual cues that indicate what portions of the representation are connected or should be attended to (Mayer & Moreno, 2003). The beneficial effects of signaling have been demonstrated in several contexts. In one experiment, Jamet, Gavota, and Quaireau (2008) showed participants a diagram on areas of the brain in conjunction with explanative narration. Learners who viewed diagrams that changed color in sync with the narration performed better on closely-related learning tasks than learners who viewed a static diagram of the brain areas while listening to the same explanation. Craig, Gholson, and Driscoll (2002) found a similar, stronger effect. Students who listened to an explanation while viewing diagrams that animated and changed color outperformed those who viewed static-representations on retention, transfer-of-learning, and text-matching tasks. As discussed by the authors of both papers, cuing may have the potential to mitigate split-attention CLT effects by eliminating the requirement that a user search one representation while elements from the other in memory (Jamet et al., 2008; Craig et al., 2002).

One key study on cuing provides evidence that dual-representation programming environments may benefit significantly from cuing support. Kalyuga, Chandler, and Sweller (1999, experiment 2) examined the possibility of reducing the split-attention effect through color-based cuing. They developed a labeled circuit diagram and accompanying (physically separate) explanation that referenced elements in the diagram. They presented two versions of this representation to two randomly selected groups of beginning electrician trainees. One group received an unmodified version of the representation, while the other group received a version that was augmented with color-coding cuing support which allowed a participant to click on any section of the text and view all the elements mentioned in the selected text highlighted in the diagram with the same individual colors in both text and diagram. This second group of students who received the color-cued version of the learning material significantly outperformed the "normal" representation group on a subsequent multiple-choice comprehension test, while reporting marginally lower subjective ratings of mental effort.

These results indicate that careful application of color-based highlighting mechanisms may reduce the amount of effort devoted to searching and integrating elements in separate representations, allowing the interface to retain the dual-representation while minimizing the increased cognitive load from the additional representation.

In application, it seems that novice programming interfaces (like ALVIS LIVE!) with separate textual and graphical representations of program state may benefit from two different modifications to the interface. The first is the addition of color correspondence highlighting to both representations, which would reduce the required effort to search for and integrate corresponding elements in each representation, lowering cognitive load and thus facilitating learning. The second is the use of highlighting to draw attention to complete units within the learner's code, in an attempt to effectively reduce intrinsic cognitive load by allowing the learner to focus on a single complete (yet still interacting) unit (Pollock et al., 2002; Wouters et. al, 2008) in order to quickly develop a schema for that unit (Chi et al., 1981).

The benefits of this augmented interface must be compared against the risk of increasing the student's cognitive load by subjecting them to two different highlighting schemes simultaneously. Work by Wallen, Plass, & Brunken (2005) indicates that while selection-level annotations (those annotations that allow the learner to select important elements in a text) are beneficial to the learner, simultaneous presentation of multiple types of annotations in the same interface dramatically increases cognitive load in textual processing tasks. This indicates that presenting the user with two different types of code highlighting might inadvertently make learning more difficult. My pilot experiment results described in the following section seem to support this hypothesis.

CHAPTER 3

CORRESPONDENCE HIGHLIGHTING DESIGN

Much of my work is based on the ALgorithm VIsualization Storyboarder (ALVIS LIVE!), developed by Dr. Hundhausen at Washington State University and described in (Hundhausen et al., 2006). Dr. Hundhausen designed ALVIS LIVE! to be a supportive educational programming environment for novices in a classroom environment. Novices using ALVIS LIVE! (Figure 5) create algorithmic solutions to problems in SALSA, an English-like pseudocode used to define their solutions.

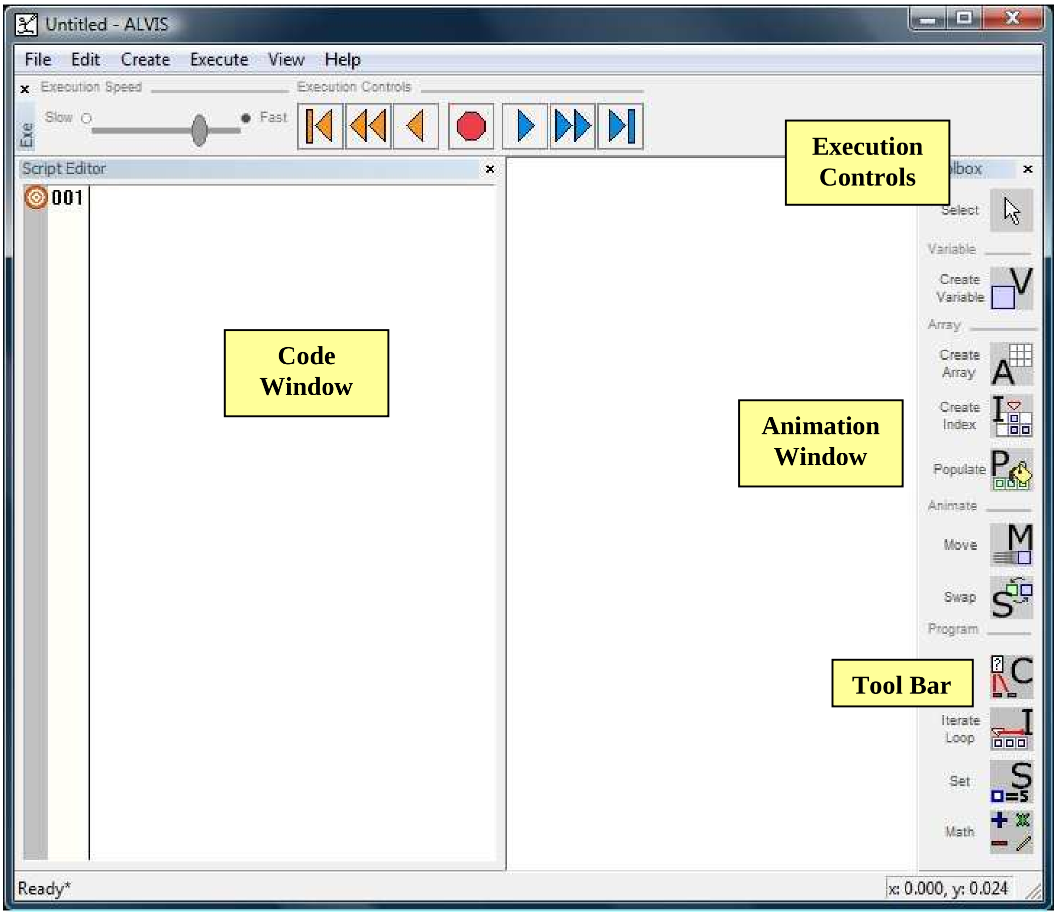
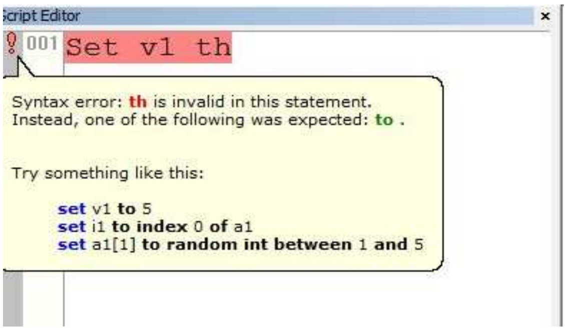


Figure 5. Existing ALVIS LIVE! interface.

Original ALVIS LIVE! Interface

The ALVIS LIVE! interface consists of two windows – the Code window and the Animation window. As a student creates a solution to a program, the same solution is reflected in both windows. Users may program complete algorithmic solutions in either the script window or the animation window. As the user enters SALSA commands in the text window, ALVIS LIVE! provides contextual suggestions and syntax error checking via a “help bubble” (Figure6). A green execution arrow sits in the left margin of the Script window, indicating which line has just been “executed” in the live ALVIS LIVE! environment.

One key element of the ALVIS LIVE! interface is its ability to represent the program state while the programmer enters commands in the script window. As each new command is entered into the script window, the line is “executed”, and the animation window is updated to indicate the new state of the program. Additionally, ALVIS LIVE! supports the step-by-step execution of program code, allowing the user to “play” their algorithm, as well as step forward and backward through instructions via the Execution Controls. At each step, the animation window updates to reflect the state of the program at the position of the execution arrow.



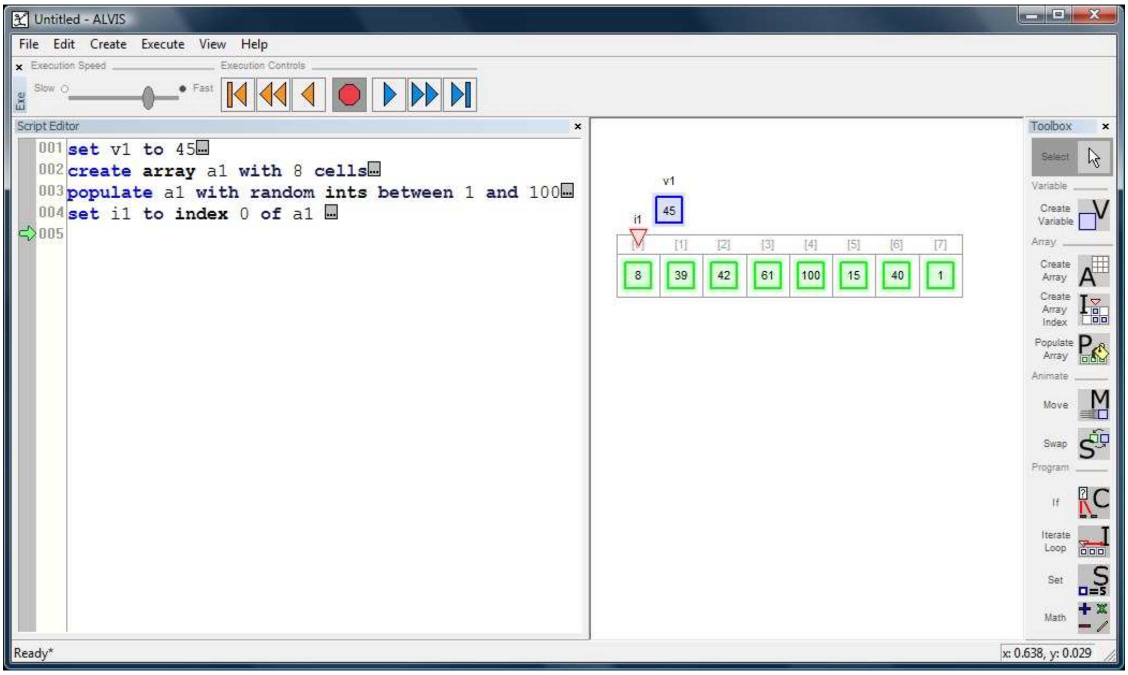
```
Set v1 th
```

Syntax error: **th** is invalid in this statement.
Instead, one of the following was expected: **to** .

Try something like this:

```
set v1 to 5
set i1 to index 0 of a1
set a1[1] to random int between 1 and 5
```

Figure 6. ALVIS LIVE! contextual suggestion initiated by user mistyping



```
set v1 to 45
create array a1 with 8 cells
populate a1 with random ints between 1 and 100
set i1 to index 0 of a1
```

Figure 7. Partial algorithmic solution constructed via Direct manipulation

In order to build algorithmic solutions by direct manipulation of graphical objects, the user may create variables, iterators and arrays via the toolbar to the right of the animation window. Furthermore, the user may create branching and iterative statements by directly manipulating existing elements in the animation window. To illustrate how a user may create a loop structure via direct manipulation, let's assume a novice programmer has created the ALVIS LIVE! program illustrated in Figure 7.

At this point, let us assume the user wants the program to iterate through all the array positions. In order to construct a while loop that iterates through the array, the user selects the Iterate Loop tool (Figure 8), then drags the array iterator (v1) from its position at the start of the array and drops it in the last cell of the array (Figure 9). This generates the WHILE loop block in the Script window, and resets the iterator to the beginning of

Figure 8. The Iterate Loop Tool Button

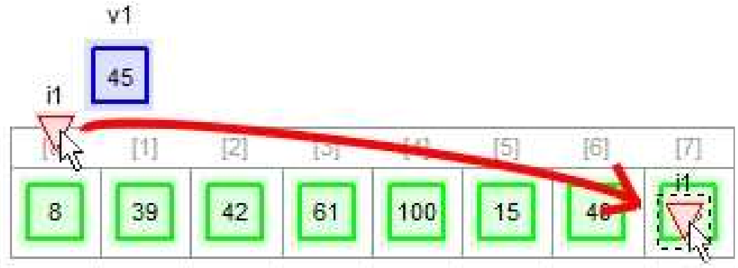
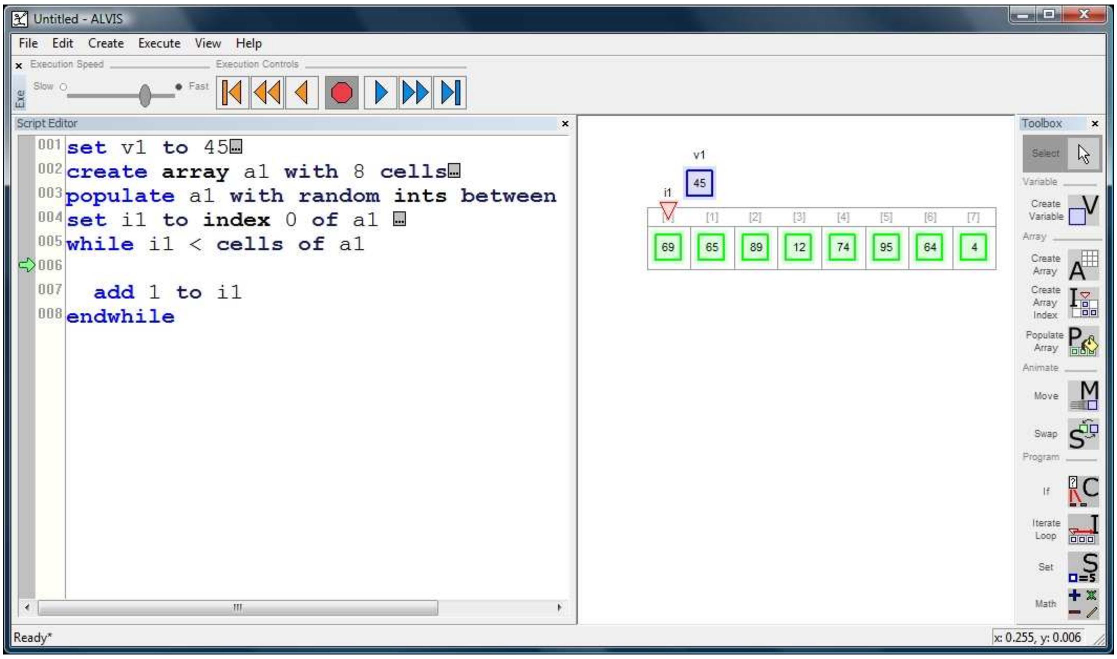


Figure 9. User drags iterator from start of array a1 to the last element in the array



```
set v1 to 45
create array a1 with 8 cells
populate a1 with random ints between
set i1 to index 0 of a1
while i1 < cells of a1

  add 1 to i1
endwhile
```

Figure 10. WHILE code is generated in the Script Window. The iterator icon is moved to the starting position.

the array (the correct position for the iterator at the start of the while loop). Figure 10 shows the program with the new WHILE loop structure in the ALVIS LIVE! interface. Via a combination of tool buttons and direct manipulation, a user may construct a complete algorithmic solution in this fashion.

As mentioned in the previous chapter, this version of the ALVIS LIVE! interface was described and experimentally evaluated in (Hundhausen et al., 2006), which showed novice programmers that used direct manipulation to construct algorithmic solutions were able to develop solutions with greater speed and accuracy then those developing solutions only via the script window. Furthermore, when all students were required to construct algorithmic solutions in the text-only ALVIS LIVE! environment, those that initially learned SALSA via direct manipulation produced scores that were significantly higher than in cases where training occurred with a text-only version of the environment.

Novice programmer Errors in ALVIS LIVE!

As described in the previous chapter, my thesis builds on previous studies of novice programming in the ALVIS LIVE! environment. Initial studies used the English-like SALSA psudeocode as its *target language* – the language in which users of ALVIS LIVE! construct algorithmic solutions. These SALSA studies showed significant benefits to learning by direct manipulation; however subsequent modifications of the target language used in the ALVIS LIVE! environment produced the ambiguous results described below.

As a follow up study to the experiment described in Chapter 2, Hundhausen et. al repeated the same experimental protocol, utilizing a C-like target language (dubbed "C*b*," prounounced "see flat") for the ALVIS LIVE! programming language instead of

Table 1
The Find Max Program as implemented in SALSA and Cb

Find Max Program in SALSA	Find Max Program in C*b*
`set v1 to 0`	`int v1 = 0;`
`create array a1 with 8 cells`	`int a1[8];`
`populate a1 with random ints between 1 and 100`	`populate( a1, 1, 100 );`
`set i1 to index 0 of a1`	`int i1 = set_as_index( a1, 0 );`
`while i1 < cells of a1`	`while (i1 < num_cells( a1 )) {`
`    if a1[i1] > v1`	`    if( a1[i1] > v1 ) {`
`        set v1 to a1[i1]`	`        v1 = a1[i1];`
`    endif`	`    }`
`    add 1 to i1`	`    i1++;`
`endwhile`	`}`

Note. The find max program iterates through an array filled with randomly generated values, and stores the largest value found in a temporary storage variable.

the English-Like SALSA pseudo-code. Table 1 provides an example of the syntactical differences between the two target languages.

With this substitution, no transfer-of-training effects were observed. This lack of effect prompted us to take a closer look at the types of problems encountered by users both SALSA and Cb ALVIS LIVE! versions. A subsequent qualitative review of screen recordings from both the previous “SALSA” study and this “C*b*” study yielded insight into four classes of novice programming errors:

Missing code block delimiters: In both the SALSA and C*b* studies, some novice programmers failed to correctly manage the scope of loops and `if` statements, missing or confusing block delimiting operators (either brackets, '{' and '},' in the case of C*b*, or “ENDIF” and “ENDWHILE” tokens in the case of SALSA). Even in the DM condition, in which DM tools automatically generated code blocks delimiters, participants mistakenly deleted the closing ENDIF or ENDWHILE in some cases. Additionally, participants in the C*b* study also exhibited confusion when nesting IF statements within WHILE statements, possibly due to the fact that both if and while structures in Cb delimit the end of code blocks with a bracket '}' token .

Statement(s) placed in wrong code block: Participants exhibited confusion when selecting the location at which to place otherwise-correct code. Some participants, intending to set a particular variable within the body of an if statement, instead placed the command after the ENDIF block delimiter but before the ENDWHILE command. In such cases ALVIS LIVE! dutifully executed the statement once on every iteration of the loop body (instead of only when the if structure's condition evaluated to true). In many such cases the participant ultimately deleted all or some of the involved WHILE and IF structures, making them vulnerable again to the kinds of *Missing Code Block Delimiter* errors described above.

Referencing the wrong variable: In some cases, particularly when using the Text-Only interface, participants initially referenced either a variable that did not exist, or the wrong variable entirely. Frequently, these errors were later noticed and resolved by the user, but many errors remained uncorrected, ultimately impacting the correctness of participant’s algorithmic solutions.

Ignoring array index variables: In both studies, some novice programmers failed to attend to array index variables when attempting to reference elements within arrays. After creating the index variable, they chose to access each variable cell directly by hard-coding the cell position (e.g., a[3]) rather than by utilizing a loop and referencing the array via the index variable (e.g., a[i]). Since many of these participants used a series of IF structures to mimic the functionality of a loop, their solutions were not technically incorrect. Unfortunately, these participants had sidestepped working through a significant programming concept: problem solving with iteration.

Based on the analysis of novice errors presented above, I can identify two different features of a given environment interface that may assist novice programmers in minimizing these kinds of errors: the interface *constraints* present, and the syntactical and logical *highlighting* schemes used by the interface.

Interface Constraints

Observed participant behavior in both of these studies anecdotally supports the notion that constraints in the ALVIS LIVE! interface may limit the kinds of programming errors I observed. Indeed, participants using the DM version of ALVIS LIVE! in both studies were prevented by the interface from iterating past the end of an array. Additionally, the DM interface paradigm required that participants select existing variable icons when setting or modifying variable values. In contrast, participants utilizing the Text-Only versions of ALVIS LIVE! could easily iterate past the end of the array, as well as attempt to assign values to variables not declared at that point in the program. Thus, to a certain degree the constraints already present in the ALIVS interface limit other kinds of errors of similar severity.

Furthermore, increasing the interface constraints in targeted areas may also limit some of the errors discussed in the previous previously. For instance, additional user interface constraints could be put in place to prevent participants from creating an IF or WHILE statement without a corresponding ENDIF or ENDWHILE statement; likewise, constraints could be developed to prevent users from deleting just the start or end of a WHILE or IF block of code. Lumping IF/ENDIF and WHILE/ENDWHILE statements together and forcing the novice to treat the statements as an entire syntactical unit, much like Alice (Powers et al., 2007) and traditional structure editing environments (Miller, Pane, Meter, & Vorthmann, 1994) directly address novice errors related to missing code block delimiters. Furthermore, allowing users in the Text-Only interface only to reference existing variables would at least prevent some instances of participants referencing a non-existent variable.

Highlighting

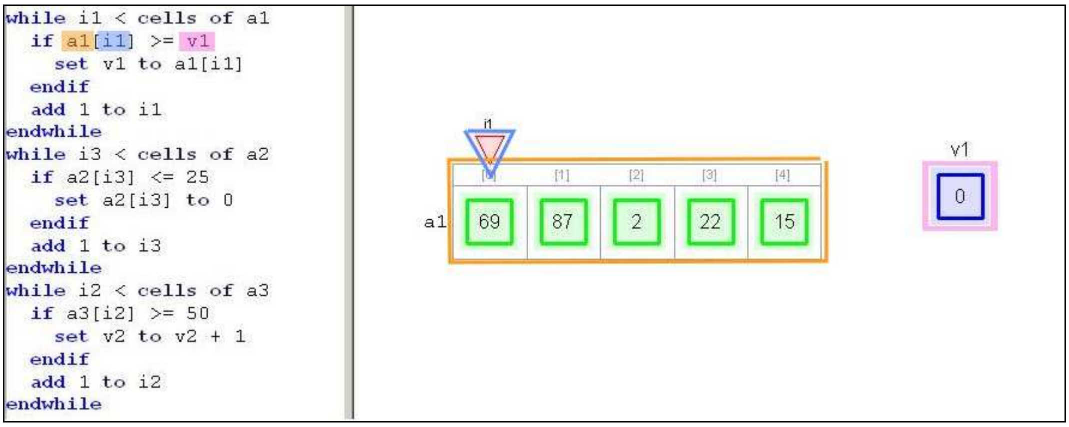
While both versions of ALVIS LIVE! used in previous studies (DM and Text-Only) supported a form of syntax error highlighting that notified the user when a particular line contains syntax errors, ALVIS LIVE! syntax checking occurred only on a line-by-line basis; it did not notify the user of missing end brackets/block delimiters. Extending the existing highlighting in ALVIS LIVE! may prove beneficial to novices prone to the kind of errors mentioned above. An interface that highlights the location of variables in both the text and animation windows could indicate the existence (or lack thereof) of declared variables and alert the user to variables set in the wrong code block. Highlighting the index variable identifier in both the text and graphical representations may draw attention to the existence and location of array index variables, reducing the effort associated with searching and integrating the variables in both representations (Kalyuga et al., 1999) and ultimately reinforcing the connections between the verbal and visual stores in the brain (Paivio, 1986). In a different context, highlighting iteration and comparison statement pairs as a “complete” semantic component—i.e., IF/ENDIF and WHILE/ENDWHILE pairs are highlighted as a unit—may help novices to recognize the IF and WHILE -blocks as complete, encapsulated semantic units. This recognition in turn may limit the number of “orphaned” IF/ENDIF and WHILE/ENDWHILE statement pairs while also heightening novices' awareness of the code within the blocks, thus preventing errors involving correct statements placed in the wrong code block.

Since increased interface constraints and increased variable identifier notification (highlighting) may have a bearing on novice programmer success, I determined that each should be examined separately. For the scope of this study, I focused on the least invasive of the two: highlighting. My goal was to develop a highlighting scheme that assists beginning programmers by strategically highlighting pertinent portions of the text and animation window contents to improve the "coupling" of those two representations. Furthermore, as discussed in Chapter 2, existing literature on CLT and Dual-Coding Theory indicate that highlighting the same elements in the code and animation windows of ALVIS Live! may allow the ALVIS Live! interface to retain the educational benefits of its simultaneous code and animation views, while reducing the extraneous cognitive load associated with split-representations and the need for learners to search for and integrate elements in both windows. For the purposes of this study, I decided to implement highlighting in the "SALSA" ALVIS LIVE! interface described in (Hundhausen et al., 2006).

A New Highlighting Interface

In developing a proposed highlighting scheme for the ALVIS LIVE! interface, I believe that the types of errors mentioned in the previous section can be avoided by novice programmers who have overcome the following two cognitive hurdles:

1. Recognition of IF/ENDIF and WHILE/ENDWHILE code blocks as complete semantic units.
2. Understanding the relationship between variables, arrays, and iterators referenced in the text editing window and their representation in the animation window.



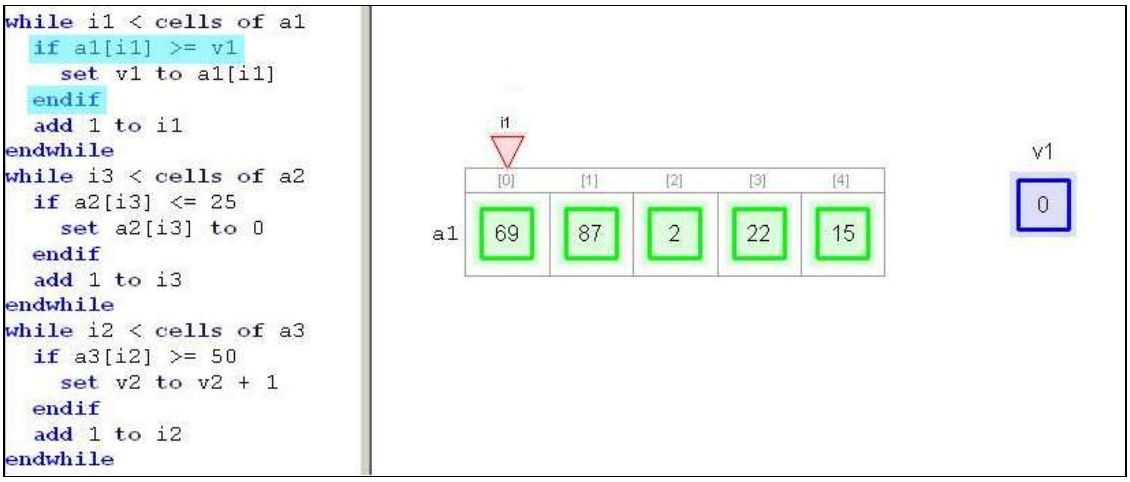
```
while i1 < cells of a1
  if a1[i1] >= v1
    set v1 to a1[i1]
  endif
  add 1 to i1
endwhile
while i3 < cells of a2
  if a2[i3] <= 25
    set a2[i3] to 0
  endif
  add 1 to i3
endwhile
while i2 < cells of a3
  if a3[i2] >= 50
    set v2 to v2 + 1
  endif
  add 1 to i2
endwhile
```

Figure 11. Example of Correspondence Highlighting in the ALVIS LIVE! Interface.

To this end, I developed two basic forms of highlighting: *correspondence highlighting* and *semantic highlighting*.

Correspondence highlighting. One method of drawing attention to the location of variable identifiers present in the text and animation windows is to highlight the variable identifier in both the text window and the animation window with a color unique to that particular identifier. This would allow multiple identifiers referenced on a singe line—like a comparison involving array cells, an array index, and a variable—to be distinguished from each other in both the Code and Animation windows, as in shown in Figure 11 above.

Semantic highlighting. By highlighting the initial statement and the following block delineators, as well as the counter increment statement in the case of loops, semantic highlighting draws the user’s attention to the existence and location of the initial IF/WHILE statement as well as appropriate block delineator (ENDIF/ENDWHILE) statements. Since conditionals and iterative statements are not represented in the DM window, this kind of highlighting can only apply to the contents of the Code Window, as shown in Figure 12 below.



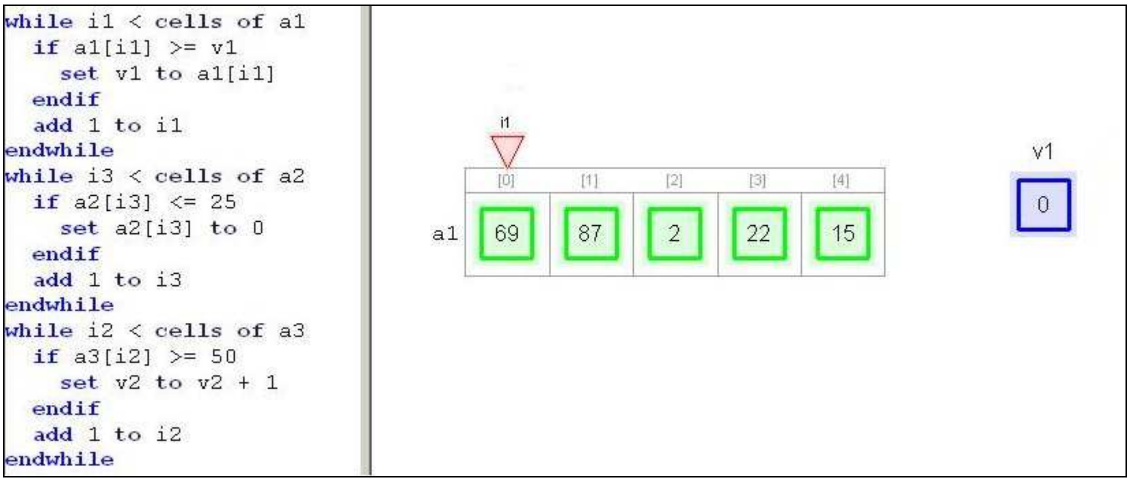
```
while i1 < cells of a1
  if a1[i1] >= v1
    set v1 to a1[i1]
  endif
  add 1 to i1
endwhile
while i3 < cells of a2
  if a2[i3] <= 25
    set a2[i3] to 0
  endif
  add 1 to i3
endwhile
while i2 < cells of a3
  if a3[i2] >= 50
    set v2 to v2 + 1
  endif
  add 1 to i2
endwhile
```

Figure 12. Example of Semantic Highlighting Condition In the ALVIS LIVE! Interface.

Pilot Study

Since Semantic and correspondence highlighting attempt to address potentially overlapping cognitive issues, implementing both types in the same ALVIS LIVE! interface would make it more difficult to empirically evaluate the effect of each type of highlighting on programmer performance. Thus, I wanted to determine the most effective highlighting scheme, and then to implement and evaluate that scheme first.

To identify the most effective highlighting combination, I developed a simple paper highlighting comprehension test to guide the selection and implementation of a highlighting scheme. This test consisted of a series of static “screenshots” of a SALSA program in the ALVIS LIVE! interface, paired with 10 questions to evaluate participants’ understanding of the relationship between the Code and Animation windows of the program, as well as their comprehension of the code. The SALSA code and basic ALVIS LIVE! interface in each version of the test remained unchanged across treatments; only the addition of one or more highlighting schemes to the screenshot changed. (See the Appendix for an example test used in the pilot study
)



```
while i1 < cells of a1
  if a1[i1] >= v1
    set v1 to a1[i1]
  endif
  add 1 to i1
endwhile
while i3 < cells of a2
  if a2[i3] <= 25
    set a2[i3] to 0
  endif
  add 1 to i3
endwhile
while i2 < cells of a3
  if a3[i2] >= 50
    set v2 to v2 + 1
  endif
  add 1 to i2
endwhile
```

Figure 13. Example of the No Highlighting pilot condition

The four treatments were as follows:

No highlighting. In this condition, participants were presented with the SALSA program via "screenshots" of the unmodified ALVIS LIVE! Interface. Figure 13 above shows part of the comprehension test from this condition.

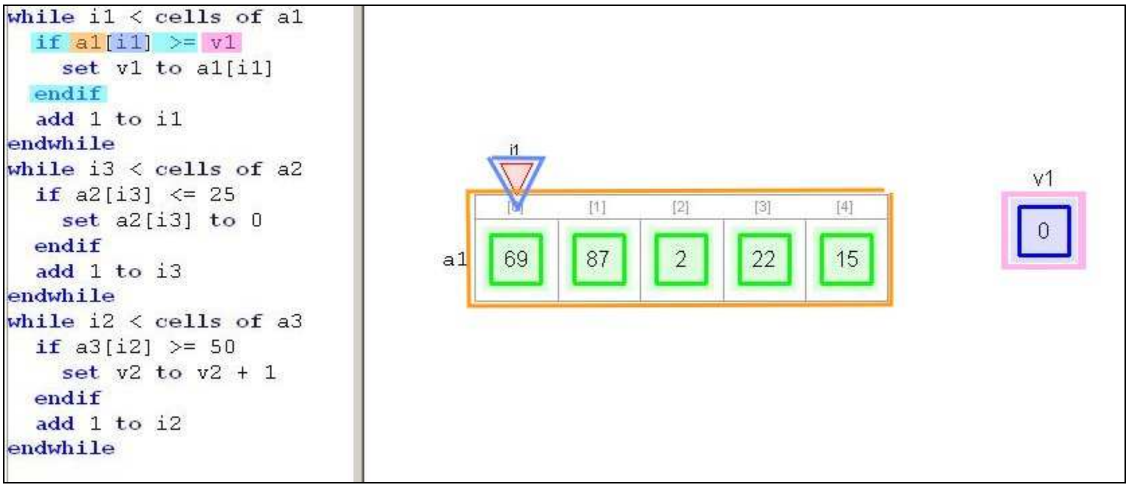
Correspondence highlighting. In this condition, participants were presented with the same "screenshots" of the ALVIS LIVE! Interface, except that variables in the line(s) being asked about are highlighted in the textual window as well as the "graphical" interface. Figure 11 above shows part of the comprehension test from this condition.

Semantic highlighting. In this condition, participants were presented with the same "screenshots" of the ALVIS LIVE! Interface, except that when questions dealt with "WHEN" or "IF" blocks, these blocks were highlighted in the text window. Figure 12 above shows part of the comprehension test from this condition.

Combination of correspondence and semantic highlighting. In this condition, participants were presented with the same "screenshots" of the ALVIS LIVE! Interface, with both correspondence and semantic highlighting treatments presented simultaneously. Figure 14 below shows part of the comprehension test from this condition.

Procedure

I recruited 68 participants from Washington State University's fall 2007 offering of Computer Science 111, an introductory programming class oriented toward new computer science students with no prior programming experience. After giving these students a standard programming pretest to assess their programming experience and skill, I randomly divided participants into four groups, and gave them the highlighting comprehension test. Participants were instructed to sit away from each other.. The highlighting comprehension test was timed. The two dependent variables for this pilot study were highlighting questionnaire score, and time on task for the highlighting questionnaire.



```
while i1 < cells of a1
  if a1[i1] >= v1
    set v1 to a1[i1]
  endif
  add 1 to i1
endwhile
while i3 < cells of a2
  if a2[i3] <= 25
    set a2[i3] to 0
  endif
  add 1 to i3
endwhile
while i2 < cells of a3
  if a3[i2] >= 50
    set v2 to v2 + 1
  endif
  add 1 to i2
endwhile
```

Figure 14. Example of Both Correspondence and Semantic Highlighting pilot condition.

Results

Table 2 presents mean score and time-on-task statistics for all four groups along with standard deviation and N for these statitics. To further explore that data, an ANOVA was used. Descriptive statistics (skew and kurtosis) as well as Levene's test results indicate that the assumptions of normality and equal variance were met. Furthermore, we took care to isolate participants so as to meet the assumption of independence of errors. Based on these indicators it seemed appropriate to use a 2 × 2 factorial ANOVA (Presence/absence of semantic highlighting × presence/absence of correspondence highlighting).

With respect to score, the main effect of correspondence highlighting was non-significant, *F*(1, 66) = 0.266, p = 0.61. The effect due to the semantic highlighting was

Table 2

Means for Time and Score factors.

Treatment (Time)	**M**	**SD**	**N**
Both	20.77	6.24	15
Correspondence	15.94	4.37	17
Semantic	20.94	8.24	16
None	20.21	6.61	19

Treatment (Score)	**M**	**SD**	**N**
Both	7.00	3.02	15
Correspondence	5.64	2.71	17
Semantic	7.43	3.82	16
None	6.00	2.81	19

non-significant, F(1, 66) = 3.38, p = 0.07. The interaction was also non-significant, F(1, 66) = 0.03, p = 0.96.

Analysis of the factors with respect to time on task yielded similar results. The main effect of correspondence highlighting was non-significant, F(1, 66) = 2.21, p = 0.14; the effect due to semantic highlighting was non-significant, F(1, 66) = 2.28, p = .10; and the interaction was non-significant, F(1, 66) = 1.65, p = 0.20. Although the differences in time on task between correspondence and semantic highlighting groups was not significant, both neared significance. While the effect of semantic highlighting was not significant with respect to score, it neared significance. An examination of the mean time-on-task results (Table 2) showed that the semantic highlighting group also took approximately 33% longer than the correspondence highlighting group to reach this score. This time difference also neared significance, which suggests that while the semantic highlighting group had an almost-significant score benefit, it might be attributable to the increased time on task. This implies that if the correspondence highlighting group spent the same amount of time developing a solution as students in other groups, they might produce higher accuracy scores then their peers in other groups.

This suspicion, based on the pattern of results, led us to take a closer look at how the ANOVA deals with variance within groups. We noted that the lack of significance might be due to the possibility that one highlighting treatment posed a significant influence on time to completion or score (hence the non-significant F values), while the other contributed variance to the analysis that masked the effect of the treatment. Given that the logic of the ANOVA involves combining the within-subjects (error) variance of all four conditions, one or two conditions with larger variance could potentially obscure prominent effects from other conditions. Indeed, an examination of mean score and variance from all four conditions (Table 3) indicates that the semantic highlighting condition without a commensurate impact on mean differences. It is possible that the condition contributed more variance to the ANOVA than the correspondence highlighting inherent variability of novice student performance as well as the less-than-ideal presentation of the dynamic ALVIS LIVE! interface through a static paper representation may have created excessive variance that obscured significant effects in the ANOVA. Given that the semantic highlighting group exhibited more variance than the correspondence highlighting group, I examined the differences with a series of independent t-tests which do not combine variance across groups.

After comparing semantic and correspondence highlighting groups against the control group (no highlighting) with respect to score and time on task variables with t-tests, only the correspondence highlighting condition revealed a significant difference on the time on task metric (df = 34, *T* = 2.25, p = .031). The other comparisons were non-significant.

Based on the sole significant effect of the correspondence highlighting treatment on time on task, we elected to use the correspondence highlighting treatment, which elicited the most significant effect in student performance (lower time), while also being consistent with the theoretical assertion that effective cuing reduces search effort and time (and therefore extraneous

Table 3

Means and variance for score and time by condition

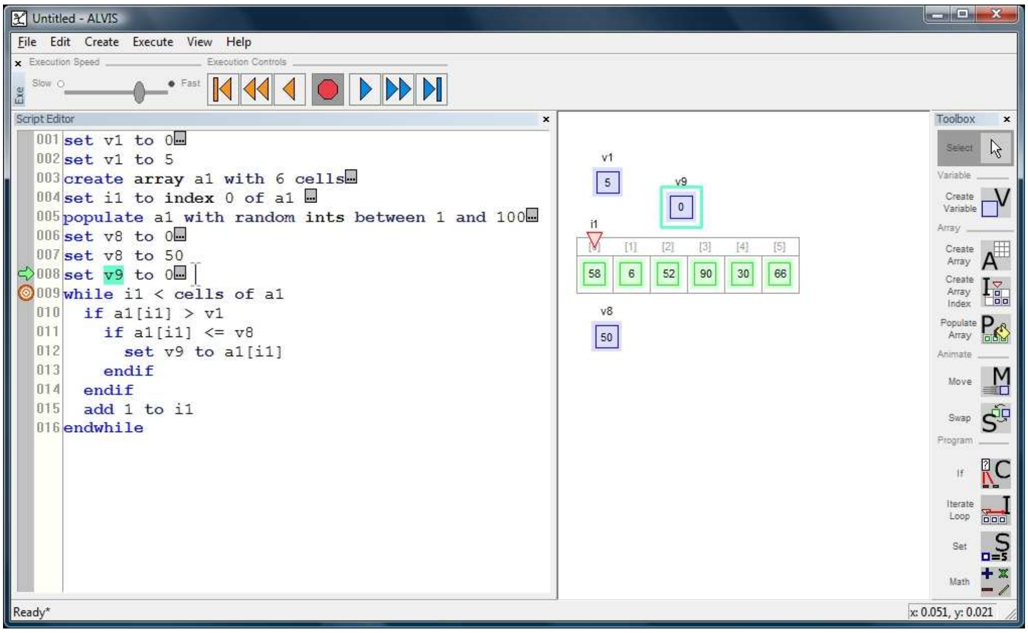
Treatment		Score		Time	
	N	**M**	**Variance**	**M**	**Variance**
Both	15	7.00	8.53	20.77	36.37
Correspondence	17	5.64	6.93	15.94	18.02
Semantic	16	7.43	13.75	20.94	59.98
None	19	6.00	7.47	20.21	41.40

cognitive load) in a dual-representation environment.

Final design: Correspondence Highlighting in ALVIS LIVE!

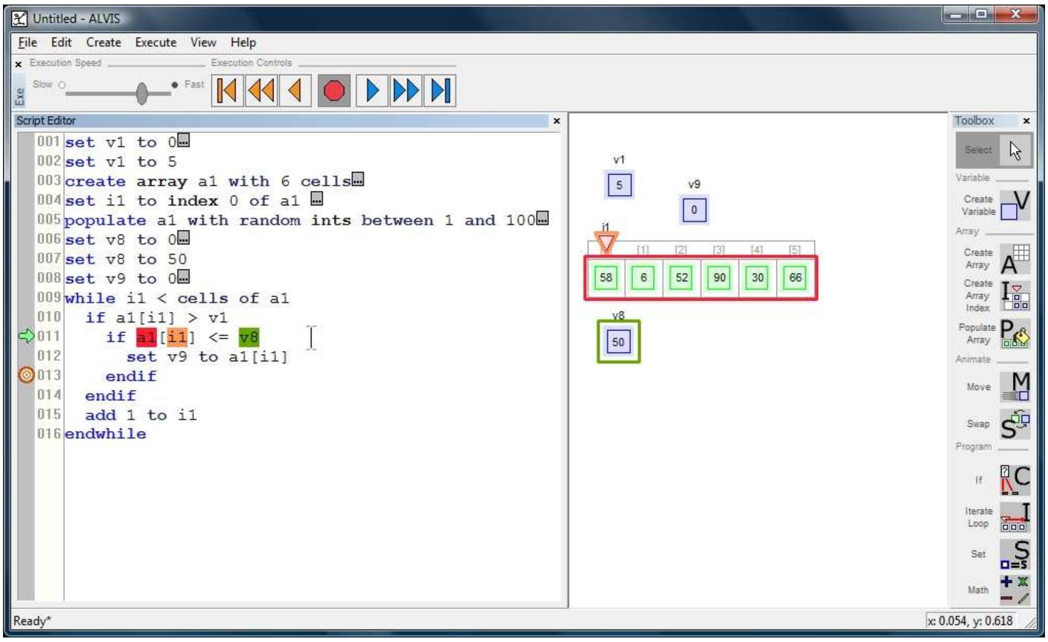
The final version of ALVIS LIVE! used for the purposes of this research consisted of the original SALSA version of ALVIS LIVE! described in Chapter 3 of this document and by Hundhausen et al. (2006), with the addition of correspondence highlighting to the interface. Since the ALVIS LIVE! programming environment supports both text and direct manipulation as forms of editing, different user actions in the text and animation windows trigger appropriate correspondence highlighting behavior.

Correspondence highlighting is triggered from within the text window by placing a line of code in focus. A user may do this by clicking within the line, moving the cursor down to the line via keyboard commands, by highlighting any portion of the line, or by executing that line (via the execution controls at the top of the window). When a line is in focus, any identifier (variable, array, or iterator name) present in the line is highlighted with a unique color, while its corresponding graphical representation is simultaneously highlighted the same color (Figure 15). If the line contains multiple identifiers, each is highlighted in its own unique color in both the text and animation windows (Figure 16). This highlighting persists as long as 1) the identifiers are not deleted by the student and 2) the line stays in focus. If an identifier is deleted in the line, its highlighted representation is removed from the animation window. If focus changes (IE, the user moves to another line) the highlighted elements in the previous line disappears, and the elements in the new line are highlighted in both the text and animation windows. Additionally, when a program is executed (via the execution controls) the line being executed is considered “in focus”.



```
set v1 to 0
set v1 to 5
create array a1 with 6 cells
set i1 to index 0 of a1
populate a1 with random ints between 1 and 100
set v8 to 0
set v8 to 50
set v9 to 0
while i1 < cells of a1
  if a1[i1] > v1
    if a1[i1] <= v8
      set v9 to a1[i1]
    endif
  endif
  add 1 to i1
endwhile
```

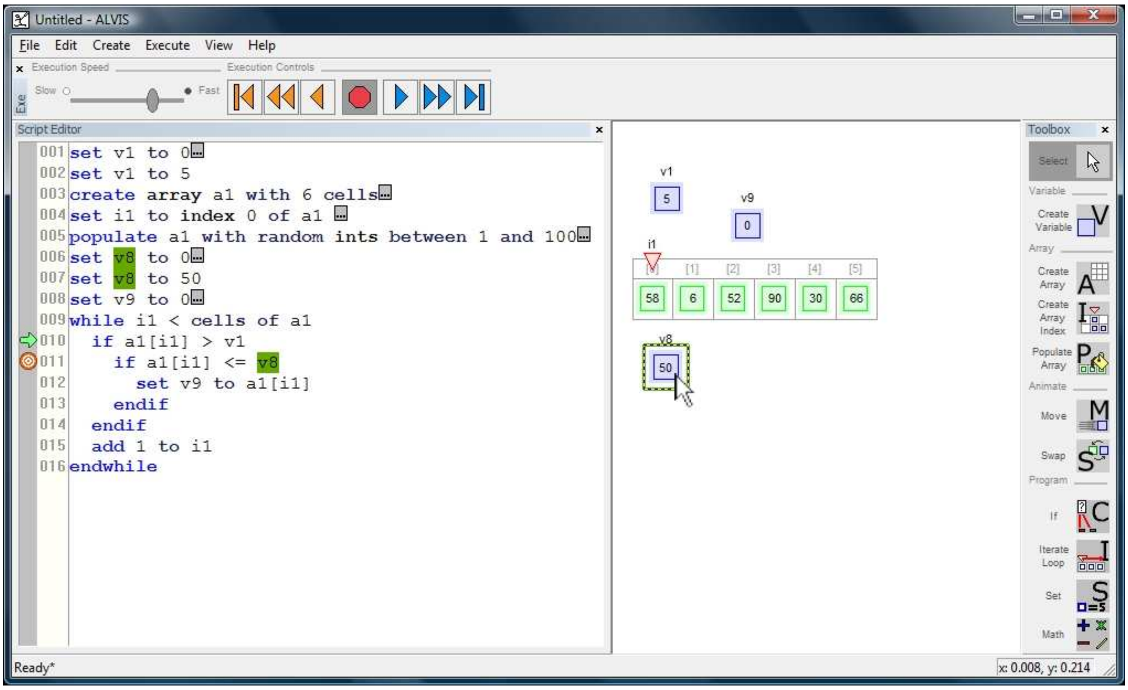
Figure 15. Selecting a single variable in text window of ALVIS LIVE!. When the user selects, edits, or executes a line of code that contains an identifier, the identifier is ighlighted with a unique color in the text and animation windows.



```
set v1 to 0
set v1 to 5
create array a1 with 6 cells
set i1 to index 0 of a1
populate a1 with random ints between 1 and 100
set v8 to 0
set v8 to 50
set v9 to 0
while i1 < cells of a1
  if a1[i1] > v1
    if a1[i1] <= v8
      set v9 to a1[i1]
    endif
  endif
  add 1 to i1
endwhile
```

Figure 16. Selecting a line of code in the text editing window of ALIVS. When the user selects a line of code, every identifier present in the line is highlighted in both the text and animation windows.

Correspondence highlighting is triggered in the animation window by selecting a pre-existing element or creating a new element. In the animation window, when a variable, array, or array index representation is selected by the user, that element is highlighted in the animation window, while *every* reference to the identifier in the text window is also highlighted (Figure 17). This highlighting persists as long as the element in the animation window remains selected.



```
set v1 to 0
set v1 to 5
create array a1 with 6 cells
set i1 to index 0 of a1
populate a1 with random ints between 1 and 100
set v8 to 0
set v8 to 50
set v9 to 0
while i1 < cells of a1
  if a1[i1] > v1
    if a1[i1] <= v8
      set v9 to a1[i1]
    endif
  endif
  add 1 to i1
endwhile
```

Figure 17. Selecting a singe variable in the animation window of ALIVS. When the user selects an identifier in the animation window, that identifier is highlighted in the animation window. Additionally, every reference to that identifier is highlighted in the text window.

CHAPTER 4

EXPERIMENTAL EVALUATION

To evaluate the effect of correspondence highlighting on novice programmer performance, I conducted an experimental study with the following hypotheses:

H1: *Students who use the ALVIS LIVE! DM interface with correspondence highlighting will be able to create algorithmic solutions significantly more quickly and accurately than students who use a DM interface without highlighting.*

H2: *Students who use the ALVIS LIVE! Text-Only interface with correspondence highlighting will be able to create algorithmic solutions significantly more quickly and accurately than students who use a Text-Only interface without highlighting.*

In order to test these hypotheses, I conducted an experiment utilizing a between subjects design with four conditions: A Text-Only interface (condition T), a Text-Only interface supporting correspondence highlighting (condition T-H), a Direct Manipulation (DM) only interface (condition D), and a Direct Manipulation interface supporting correspondence highlighting (condition D-H). Much of the protocol used in this experiment is similar to that described in (Hundhausen, 2006), as I was also interested in replicating the results of that study, which showed that the Direct Manipulation promoted a transfer-of-training effect to the Text-only interface.

Students in the T condition used a version of ALVIS LIVE! that allowed them to edit SALSA code through the Text Window. Students could view but could not interact with elements in the Animation Window, and the direct manipulation toolbar was not present in this version of ALVIS LIVE!. These students used this version of ALVIS LIVE! for all three tasks.

The T-H group used an almost identical version of ALVIS LIVE! as the T condition, except this interface supported correspondence highlighting as described in the previous section. These students used this version of ALVIS LIVE! for all three tasks.

For the first two tasks, students in the D condition used a version of ALVIS LIVE! to code solutions to common algorithmic problems via the Direct Manipulation tools and direct interaction with graphical elements in the Animation Window. While students could use this interface to view, select, and delete individual lines in the Text Window, they were prevented from typing or editing code in this window. For the third task, this group switched to the T (Text-Only, No Highlighting) interface used by students in the T condition.

The D-H group used an almost identical version of ALVIS LIVE! as the D condition, except this interface also supported correspondence highlighting as described in the previous section. After completing the second tasks, this group completed the third task with the T-H (Text-Only, with Highlighting) interface used by students in the T-H condition.

Effectiveness of each interface treatment was assessed by recording two dependent variables—time on task and semantic accuracy.

Participants

I recruited 51 students out of the Spring, 2008 offering of CptS 121 and CptS 111, the introductory computer science courses at Washington State University. Participants were recruited in the fourth week of the semester, before they had received instruction on arrays (the topic of the experiment’s tasks). Most participants received course credit for their participation.

Materials

All experiment sessions were conducted in a computer lab containing machines utilizing 3 Gigahertz processors, 1 GB of RAM, and running Windows XP Professional. The 17 inch monitors used were set to a resolution of 1280x1024. Each lab computer was equipped with Morae Recorder©, which was used to record the screens of participants for the duration of each task. These recordings were later used to calculate their time on task, as well as reconstruct corrupted or missing task solutions.

Study Tasks

Participants in each condition completed three isomorphic tasks which dealt with array traversal and manipulation. For each task, participants were required to create and initialize an array with random values. After this initialization, the participant was then required to construct an algorithm that fulfilled the requirements of Find and Replace, Find Max, and Count tasks. The tasks were semantically isomorphic to each other, so that I could use the same universal grading criteria established in previous studies by Hundhausen et al. (2006).

In the Find and Replace task, any array values less than 25 were to be replaced with the value 0.

A correct solution to this task in SALSA code looked something like this:

```
set v1 to 0
create array a1 with 7 cells
populate a1 with random ints between 1 and 100
set i1 to index 0 of a1
while i1 < cells of a1
  if a1[i1] < 25
    set a1[i1] to 0
  endif
  add 1 to i1
endwhile
```

In the Count task, the participants were to iterate through the array and store in a variable the tally of array values greater than or equal to 50. A correct solution to this task in SALSA code looked something like this:

```
set v1 to 0
create array a1 with 7 cells
populate a1 with random ints between 1 and 100
set i1 to index 0 of a1
while i1 < cells of a1
  if a1[i1] >= 50
    set a1[i1] to 0
  endif
  add 1 to i1
endwhile
```

For the Find Max task, participants were required to iterate through the array and store the largest value present in the array. A correct solution to this task in SALSA code looked something like this:

```
 set v1 to 0
create array a1 with 7 cells
populate a1 with random ints between 1 and 100
set i1 to index 0 of a1
while i1 < cells of a1
  if a1[i1] > v1
    set v1 to a1[i1]
  endif
  add 1 to i1
endwhile
```

Completed solutions to all three tasks were saved by each participant and collected after the conclusion of each study session.

Procedure

The experiment was conducted during five lab sessions which lasted an average of two hours and thirty minutes each, and included an average of 10 participants. Students were randomly assigned to conditions, with the order of tasks counterbalanced to guard against task order effects.

In each session, participants began with a 15 minute pre-test of basic programming competency. They then spent 15 minutes working through an informationally-equivalent written tutorial specific to the version of ALVIS LIVE! they used for the first two tasks. Those using a

Text-Only version of ALVIS LIVE! (regardless of highlighting condition) were instructed on typing SALSA commands in the text window, those using the Direct Manipulation version (again, regardless of highlighting condition) were instructed on how to use the animation window tools to create the same code. The tutorials for the Highlighting conditions (D-H and T-H) were almost identical to their DM or Text-Only counterparts with two exceptions: highlighting condition tutorials contained illustrations demonstrating the appearance of highlighting and a brief explanation of the correspondence highlighting scheme. Following the tutorial, participants began working on the three programming tasks. Before each task, participants were instructed to set up their screen recording software and begin the task by opening the appropriate version of ALVIS LIVE!, working as quickly and accurately as possible. After 35 minutes, or whenever they were finished, participants were instructed to save their work, close the ALVIS LIVE! interface, and stop their screen recording.

For the third task, participants utilizing a DM interface (D or D-H conditions) were instructed to use the equivalent “text-only” interface (T or T-H condition). Those participants who completed the first two tasks with an interface that supported correspondence highlighting retained the highlighting during the “text-only” third task. They were not provided a tutorial or given instruction in the use of this new interface. After completing the third task, participants filled out an exit questionnaire.

Dependent Variables

Since this experiment was designed to compliment prior work by Hundhausen et al. (2006) in developing the ALIVS treatments used in the D and T conditions, I utilized an identical method of measuring the dependent variables.

To measure the accuracy of each task, I divided each isomorphic task into eight semantic components that had to be present and correctly implemented for the solution to be correct: (a) create array; (b) populate array; (c) create array index; (d) index visits each array cell; (e) loop terminates; (f) correct comparison; (g) correct change; (h) correct result. Each element maps to a specific line or block of code of a correct solution. I scored each solution from 0 to 8 based on the number of semantic elements correctly implemented. The Table 4 contains examples of semantic elements a-g as scored in a correct implementation of the Count task. The last element h (correct result) is used to differentiate between solutions that otherwise would be identical.

Table 4
Example graded Count task solution with associated semantic elements.

Example Code	**Semantic element**
`set v1 to 0`	
`create array a1 with 7 cells`	(a) create array
`populate a1 with random ints between 1 and 100`	(b) populate array
`set i1 to index 0 of a1`	(c) create array index
`while i1 < cells of a1`	(d) index visits each array cell, (e) loop terminates
`if a1[i1] >= 50`	(f) correct comparison
`set a1[i1] to 0`	(g) correct change
`endif`	(f) correct comparison
`add 1 to i1`	(d) index visits each array cell,
`endwhile`	(e) loop terminates

To measure time on task, I reviewed the screen recordings, noting the time at which each participant started and stopped the task. I defined the task start as the point in the recording which the participant opens the ALVIS LIVE! interface, and the task end as the point at which the participant last selects the “Save” or “Save as” option from the file menu. Since ALVIS LIVE! automatically saves students’ solutions in the event of a crash, I did not subtract time in instances where the interface crashed. Students merely opened up ALVIS LIVE! again and continued with their programs.. In the few instances that a participant took longer than the allotted 35 minutes to complete a task, I stopped the recording at 35 minutes and scored their solution at that point for the programming accuracy portion of the analysis.

Results and Discussion

Quantitative Assessment. Tables 5 and 6 present the means and standard deviations of the four conditions with respect to the two dependent measures. Figures 18 through 23 present line graphs of this data. As these plots suggest, there was a large amount of variance in the data, with all four conditions appearing to perform similarly on all three tasks. In particular, all conditions not only have similar accuracy scores and times across task, but also a similar trend: accuracy scores trended upward from task to task, while times on task trended downward.

Table 5

Means for score by condition and task

Condition	N	Task 1		Task 2		Task 3	
		M	SD	M	SD	M	SD
D	12	5.75	2.49	7.00	1.70	6.08	2.23
D-H	11	6.09	2.16	5.72	2.05	6.18	1.99
T	13	5.46	2.22	5.23	2.65	5.76	2.31
T-H	13	5.38	2.46	5.76	2.61	5.84	2.37

Table 6

Means for time by condition and task

Condition	N	Task 1		Task 2		Task 3	
		M	SD	M	SD	M	SD
D	12	18.62	9.54	9.48	9.66	15.56	9.42
D-H	11	15.69	8.15	11.94	5.92	15.58	7.34
T	13	20.61	8.93	13.22	8.24	9.07	6.80
T-H	13	20.89	9.52	12.28	10.21	10.92	8.01

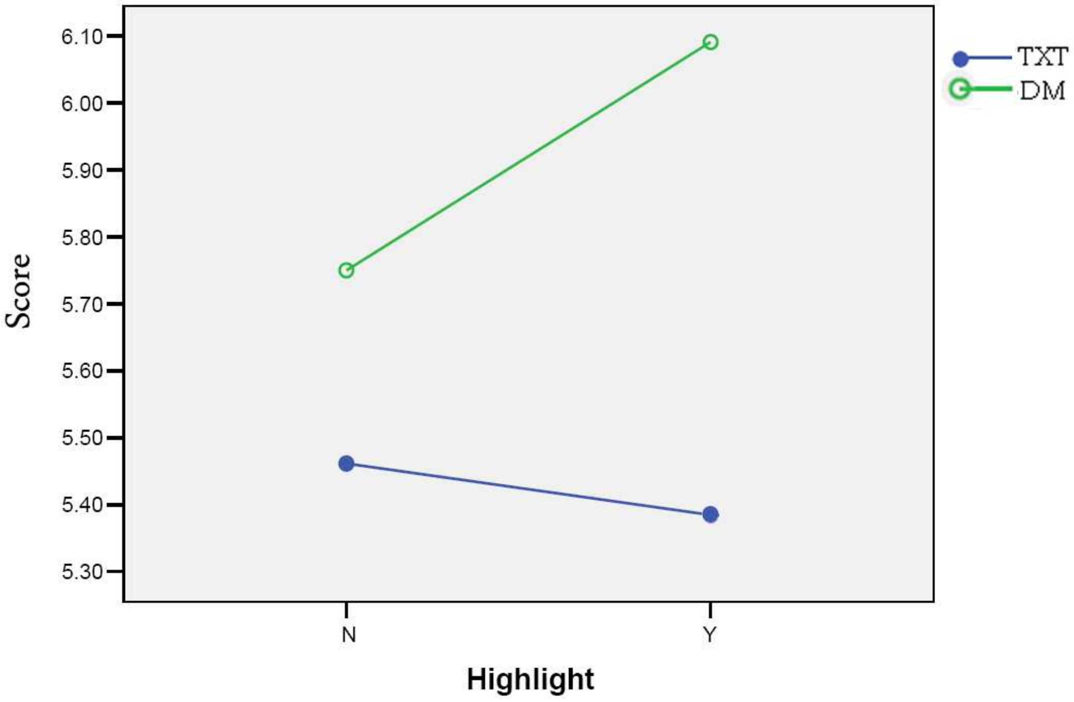


Figure 18. Mean score for Task 1, by interface and highlighting conditions.

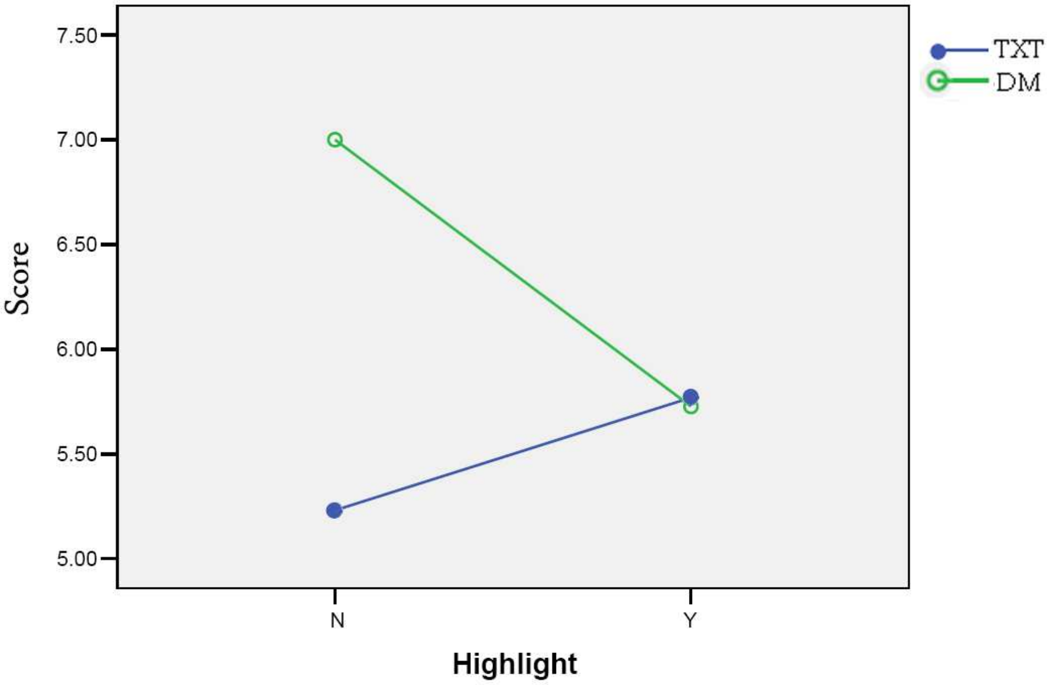


Figure 19. Mean score for Task 2, by interface and highlighting conditions.

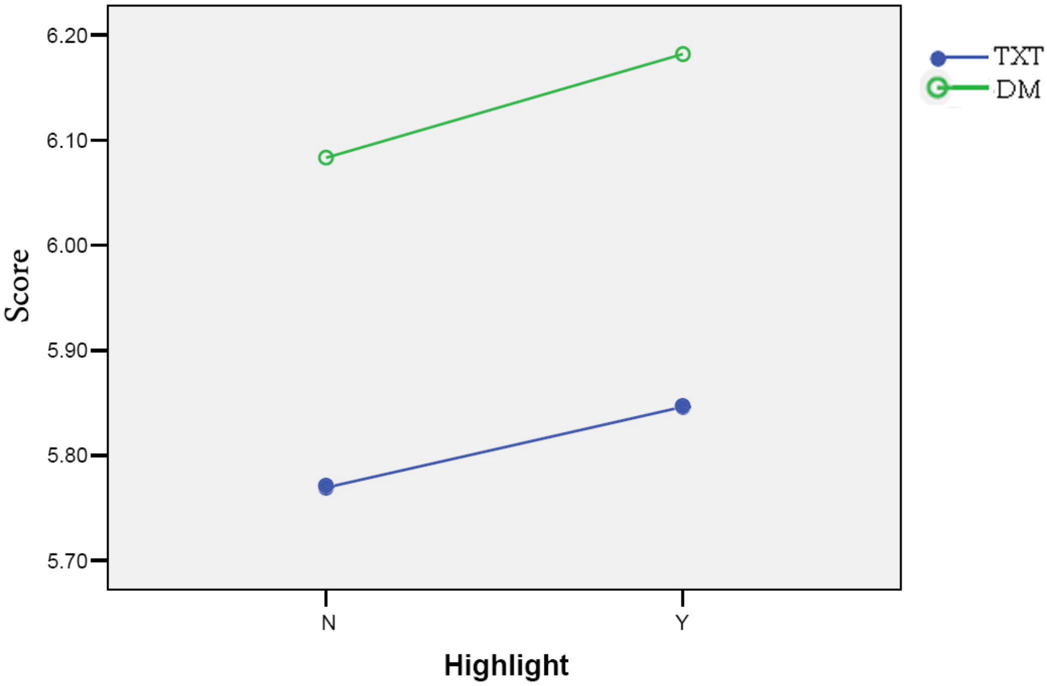


Figure 20. Mean score for Task 3, by interface and highlighting conditions.

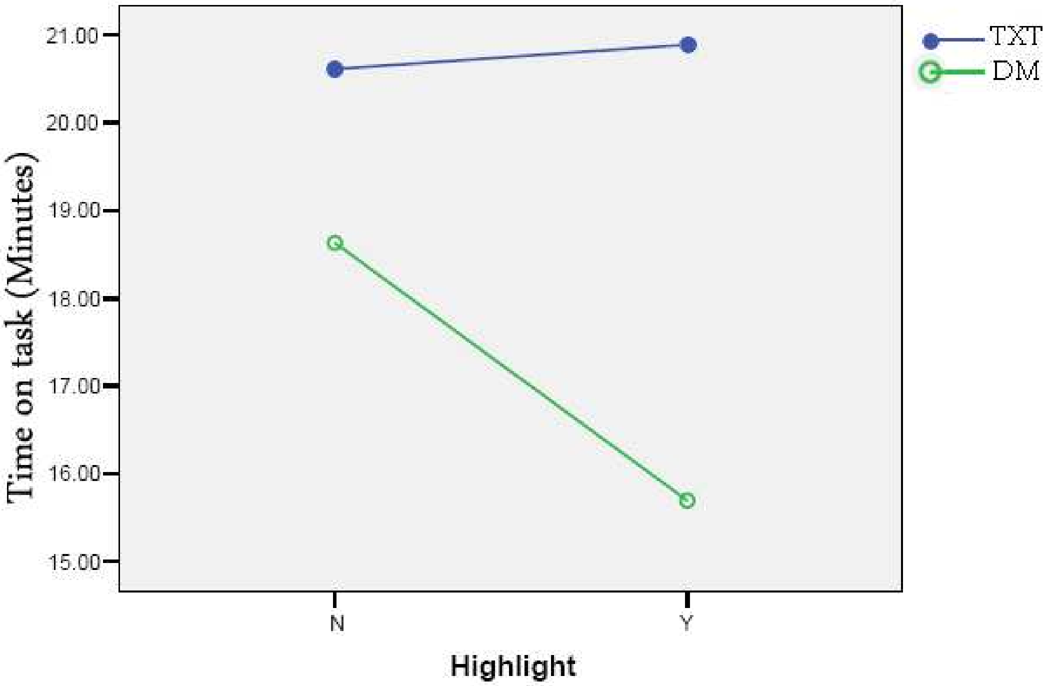


Figure 21. Mean time on task for Task 1, by interface and highlighting conditions.

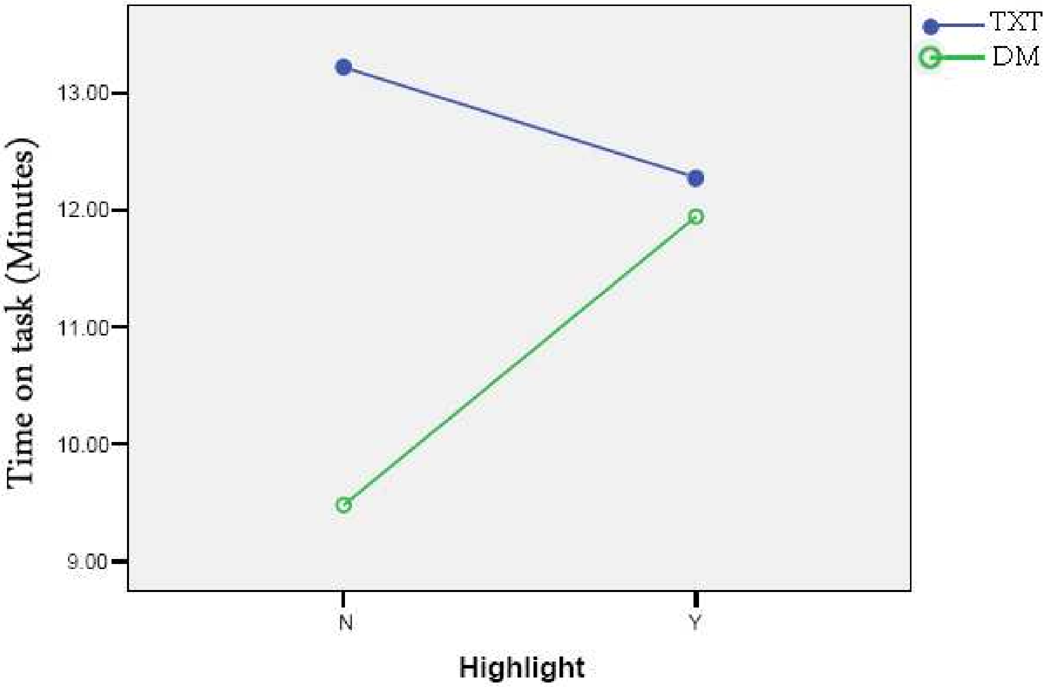


Figure 22. Mean time on task for Task 2, by interface and highlighting conditions.

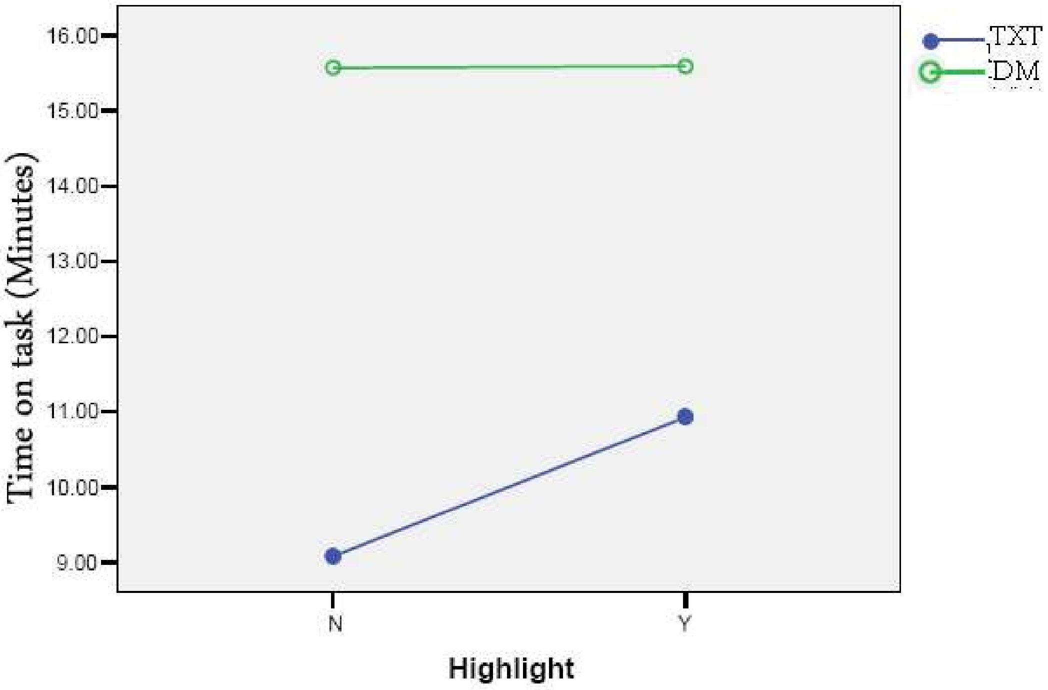


Figure 23. Mean time on task for Task 3, by interface and highlighting conditions.

Table 7

ANOVA Results for Dependent Measures

Source of Variation (Score)	**DF**	**F-Value**	**P-Value**
Interaction (DM, Text)	1	12.52	0.07
Highlighting (Yes, No)	1	0.10	0.79
Interaction x Highlighting	1	0.44	0.57
Interaction x Task	2	0.21	0.82
Highlighting x Task	2	0.22	0.82
Interaction x Highlighting x Task	2	0.81	0.45
Source of Variation (Time)			
Interaction (DM, Text)	1	0.00	0.99
Highlighting (Yes, No)	1	0.03	0.88
Task (1, 2, 3)	2	2.69	0.29
Interaction x Highlighting	1	0.07	0.81
Interaction x Task	2	7.89	0.11
Highlighting x Task	2	0.52	0.66
Interaction x Highlighting x Task	2	0.50	0.61

In order to determine whether there existed significant differences among the conditions, with respect to the two dependent measures, I ran a 2 × 2 x 3 ANOVA with interaction method, highlighting presence, and task number as the main effects. Table 7 presents the results of that ANOVA. As the table indicates, there were no significant differences with respect to score or time, although the interaction method almost reached significance with respect to score. Thus, the quantitative experimental results failed to confirm my theoretically-grounded hypotheses regarding the efficacy of a dual-representation environment. Participants who used the correspondence highlighting interface performed no better than those who did not use it.

Qualitative assessment. While we failed to find quantitative evidence of the superiority of correspondence highlighting, student responses to the highlighting interface regardless of the particular interface method (Direct manipulation or Text-only) were encouraging. Students that used a version of ALVIS LIVE! with highlighting (D-H or T-H) were asked to answer highlighting-specific questions on their exit questionnaires. One of the questions in the exit questionnaire asked students in the D-H and T-H conditions to rate the helpfulness of correspondence highlighting on a 10-point scale (where 0 indicates that highlighting was no help at all and 10 indicates that highlighting was extremely helpful). The average rating on this question from students in the D-H and T-H conditions was 6.85 and 6.80, respectively. On an open-ended question regarding the helpfulness of highlighting in constructing their solutions, respondents offered comments like:

- “The highlighting was helpful because it showed when objects in the visualization were located in the code.”
- "It helped to identify which part of the function was being affected, making it very easy to work with."
- "The variable highlighting was helpful in that it made it clear which name corresponded to which variable."
- "I would… have to change the variables and highlighting made the task easy; visually, it eliminated a lot of frustration."

Thus, we see that anecdotally at least, most participants recognized some benefit from the use of correspondence highlighting to tie the two representations together and cut down on search.

In trying to reconcile the anecdotal support for correspondence highlighting with the lack of effect it seemed to have on novice programmer performance, I can identify at least one threat to validity, and three alternative explanations of my results, as discussed below.

Threats to Validity. As in Hundhausen et al.'s (2006) prior ALVIS LIVE! experiment, I attempted to control for student abilities by administering a pre-test of the specific programming knowledge that would be needed to complete study tasks. A statistical comparison of the four conditions' pre-test scores (Table 8) yielded no significant differences between the groups (df = 3; F = 0.34; *p* = *0.*79). Unsurprisingly, when I used the pre-test as a covariate to moderate the effects of programming ability on the experimental analysis, I similarly found no significant effects.

However, in contrast to the prior ALVIS LIVE! experiment, I did not administer a background questionnaire to screen participants for prior programming experience. Moreover, while I administered this experiment in the fourth week of the semester before the class was exposed to arrays and loop structures (the primary subject of the experimental tasks), the prior evaluation of ALVIS LIVE! occurred in the second week of the semester (Hundhausen et al., 2006). It is therefore possible that participants in this experiment had more programming experience than participants in the previous ALVIS LIVE! study. My failure to control for prior programming experience can be seen as a threat to the validity of my results.

Table 8

Means for pre-test scores by condition

Condition	Mean	SD	N
D	5.50	1.44	12
D-H	5.72	1.27	11
T	5.00	2.58	13
T-H	5.76	2.77	13

Alternative explanations. It is possible that the experimental tasks selected were not complex enough for split-attention to hinder my participants' performance. Cogntive Load Theory posits that cognitive load can become an issue when the tasks presented are realistically complex (Sweller et al., 1998; Merriënboer et al., 2006). If the cognitive tasks are too simple, it's possible that the cognitive load present in the task and presentation is well within the learner's capability and any attempts to reduce it further will have no effect on learner performance. Given the relative simplicity of the three programming tasks used in the study, my correspondence highlighting scheme may not have yielded a performance advantage.

Alternatively, if participants were advanced enough in their programming skills, correspondence highlighting may have *negatively* impacted their performance. Research on the extent of the effects of cognitive load has shown an *expertise reversal effect*, in which efforts to reduce cognitive load for novice learners negatively impacts more experienced learners (Sweller et al., 1998). Particularly in instances where learners have no difficulty integrating separate sources of information, additional information or cuing mechanisms can be redundant, forcing the learner to filter supportive information or material, and ultimately having *negative* effects on cognitive load measures and learner performance (Kalyuga, 2003). In a programming context, this means that at some point increasing the amount of commenting and other cognitive aids makes comprehending code *more* difficult for more experienced programmers. Thus, correspondence highlighting (along with the supportive ALVIS LIVE! programming environment) may have been "overkill" for these participants, reducing their ability to perform regardless of the interface version used.

Another alternative explanation is that participants' programming skills were too advanced, rendering *both* the DM interface and the highlighting unnecessary. Indeed, the results of Hundhausen et al.'s (2006) study were not replicated; the DM interface showed no significant advantage in this follow-up study. Since I used an identical pre-test to that administered in my prior study, I was able to compare the pre-test scores from that study with those from this study. The pre-test scores in the previous experiment were indeed lower (*M* = 4.94) and had a smaller standard deviation (1.67), than those in this experiment (*M* = 5.49, *SD* = 2.12). However, according to a two-tailed *t*-test, there were no significant differences in pre-test performance (*df* = 1, *T* = 1.26, *p* = 0.211). This may indicate that the impact of both direct manipulation and correspondence highlighting are sensitive to an aspect of prior experience that was not well-gauged by the programming pre-test.

CHAPTER 5

SUMMARY AND FUTURE WORK

Although many dual-representation novice programming environments have been developed, few have been empirically evaluated, and none has explored the design implications of Dual-Coding and Cognitive Load Theories. To address this gap, I have surveyed the existing Dual-Coding and Cognitive Load literature and identified a promising design modification to dual-representation novice programming interfaces: correspondence highlighting. I implemented correspondence highlighting in the ALVIS LIVE! programming environment, such that when an element in the Code Window is selected, ALVIS LIVE! uses color to highlight the corresponding graphical element in the Animation Window, and vice versa. In an experimental study, correspondence highlighting failed to provide a significant performance advantage over a version of ALVIS LIVE! without correspondence highlighting. The study also failed to replicate the Direct Manipulation performance advantage reported previously by Hundhausen et al. (2006). The lack of significant statistical results yielded by the study opens up several directions of future research.

Program complexity and cognitive load. One explanation of our lack of effect is that the tasks selected for this study did not require enough cognitive load to make split-attention effects an issue. As the existing CLT literature suggests, more cognitive load may be generated by more complex programs. Future research with dual-representation novice programming environments may fruitfully explore the effects of incrementally increasing cognitive load for novice programmers using a dual-representation environment like ALVIS LIVE!. By providing novice programmers increasingly complex programming tasks—like implementing various sorting algorithms and using multiple arrays—with the aim of incrementally increasing cognitive load, researchers may be able to observe the increasing performance impact of the split-attention effect and the utility of efforts to mitigate it through techniques like correspondence highlighting. This would provide useful insight into how the cognitive capacity of novice programmers evolves with experience and how the behavior of a novice programming environment may evolve with the learner to better facilitate learning as tasks increase in complexity

Individual differences. One direction is to explore possible individual differences that might contribute to one's ability to benefit from both direct manipulation and correspondence highlighting. Here, I wonder whether individual learning styles like visual vs. auditory (Felder & Silverman, 1988) might play a pivotal role, particularly in the early stages of programming education. I would recommend that future studies administer pre-tests of learning styles and other relevant psychometric factors to explore this space.

Eye tracking studies. Another direction for future research is to use eye tracking technology to explore the effects of split-attention in a dual-representation programming context. Indeed, knowing what representations participants look at, and when they look at them, could provide valuable insight into how participants successfully enlist and integrate dual representations in programming tasks. Observing eye movement behavior as participants develop increasingly complex algorithmic solutions could give us even more insight into the evolving needs of novice programmers as they gain experience and tackle more difficult programming tasks. Furthermore, observing how novice programmers adjust,, or don't adjust, to dual representations with correspondence highlighting may also help gauge what effect, if any, cuing treatments have on split-attention issues for dual representations. Such insight, in turn, could ultimately help us to improve the design of dual representation interfaces for computer programming.

Nunc scripsi totum, pro Christo da mihi potum.

Bibliography

Alice.org. (2009). *What is Alice?*. Retrieved March 26, 2009, from http://alice.org/index.php?page=what_is_alice/what_is_alice

Baddeley, A. (1992). Working memory. *Science*, *255*(5044), 556-559.

Baddeley, A. D. (2001). Is working memory still working? *The American Psychologist*, *56*(11), 851-64.

Barnes, D. J. (2002). Teaching introductory Java through LEGO MINDSTORMS models. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education* (pp. 147-151). Cincinnati, Kentucky: ACM.

Beaubouef, T., & Mason, J. (2005). Why the high attrition rate for computer science students: some thoughts and observations. *SIGCSE Bull.*, *37*(2), 103—106.

BlueJ.org (2009) *BlueJ – What is BlueJ*. Retrieved March 26, 2009, from http://www.bluej.org/about/what.html

Brown, P. H. (2008). Some field experience with Alice. *J. Comput. Small Coll.*, *24*(2), 213-219.

Bureau of Labor Statistics, U.S. Department of Labor. (2009). *Occupational Outlook Handbook, 2008-09 Edition, Computer Software Engineers*. Retrieved March 26, 2009, from http://www.bls.gov/oco/ocos267.htm.

Byrne, P., & Lyons, G. (2001). The effect of student attributes on success in programming. *SIGCSE Bull.*, *33*(3), 49-52.

Carlisle, M. (2009). *RAPTOR - Flowchart Interpreter*. Retrieved March 26, 2009, from http://raptor.martincarlisle.com/.

Carlisle, M. C., Wilson, T. A., Humphries, J. W., & Hadfield, S. M. (2005). RAPTOR: a visual programming environment for teaching algorithmic problem solving. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education* (pp. 176-180). St. Louis, Missouri, USA: ACM.

Chandler, P., & Sweller, J. (1991). Cognitive load theory and the format of instruction. *Cognition & Instruction*, *8*(4), 293.

Cohoon, J.M., Chen, L. (2003, March). Migrating Out of Computer Science. *Computing Research News*, Vol. 15 (No. 2) 2-30. Retrieved November 12, 2008, from http://www.cra.org/CRN/articles/march03/cohoon.chen.html

Cooper, S., Dann, W., & Pausch, R. (2000). Alice: a 3-D tool for introductory programming concepts. In *Proceedings of the fifth annual CCSC northeastern conference.* (pp. 107-116). New Jersey, United States: Consortium for Computing Sciences in Colleges.

Craig, S. D., Gholson, B., & Driscoll, D. M. (2002). Animated Pedagogical Agents in Multimedia Educational Environments: Effects of Agent Properties, Picture Features, and Redundancy. *Journal of Educational Psychology*, *94*(2), 428-34.

Farley, S. (2006). *The design and evaluation of a direct manipulation interface for novice programmers*. Unpublished masters thesis, Department of Electrical Engineering and Computer Science, Washington State University, Pullman, WA.

Felder, R. M., & Silverman, L. K. (1988). Learning and Teaching Styles in Engineering Education. *Engineering Education*, *78*(7), 674-81.

Ginns, P. (2006). Integrating information: A meta-analysis of the spatial contiguity and temporal contiguity effects. *Learning and Instruction*, *16*(6), 511-525.

Giordano, J. C., & Carlisle, M. (2006). Toward a more effective visualization tool to teach novice programmers. In *Proceedings of the 7th conference on Information technology education* (pp. 115-122). Minneapolis, Minnesota, USA: ACM.

Henriksen, P., & Kölling, M. (2004). Greenfoot: Combining Object Visualization with Interaction. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (pp. 73-82). Vancouver, BC, CANADA: ACM.

Howles, T. (2007). Preliminary results of a longitudinal study of computer science student trends, behaviors and preferences. *J. Comput. Small Coll., 22*(6), 18—27.

Hundhausen, C., & Douglas, S. (2000). SALSA and ALVIS: a language and system for constructing and presenting low fidelity algorithm visualizations. In *The 2000 IEEE International Symposium on Visual Languages, 2000.* (pp. 67-68). Los Alamitos, Ca: IEEE Computer Society Press.

Hundhausen, C., & Brown, J. (2005). What you see is what you code: a radically dynamic algorithm visualization development model for novice learners. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing,* (pp. 163-170). Dallas, TX: IEEE Computer Society Press.

Hundhausen, C., Farley, S., & Brown, J. (2006). Can Direct Manipulation Lower the Barriers to Programming and Promote Positive Transfer to Textual Programming? An Experimental Study. In *2006 IEEE Symposium on Visual Languages and Human-Centric Computing,*. (pp. 157-164). Piscataway, NJ: IEEE Computer Society Press.

Hundhausen, C. D., Brown, J. L., & Farley, S. (2006). Adding procedures and pointers to the ALVIS algorithm visualization software: a preliminary design. In *Proceedings of the 2006 ACM symposium on Software visualization* (pp. 155-156). Brighton, United Kingdom: ACM.

Jacobsen, C. L., & Jadud, M. C. (2005). Towards concrete concurrency: occam-pi on the LEGO mindstorms. *SIGCSE Bull., 37*(1), 431-435.

Jamet, E., Gavota, M., & Quaireau, C. (2008). Attention guiding in multimedia learning. *Learning and Instruction*, *18*(2), 135-145.

Kablan, Z., & Erden, M. (2008). Instructional efficiency of integrated and separated text with animated presentations in computer-based science instruction. *Computers & Education*, *51*(2), 660-668.

Kalyuga, S., Ayres, P., Chandler, P., & Sweller, J. (2003). The Expertise Reversal Effect. *Educational Psychologist, 38*(1), 23-31.

Kalyuga, S., Chandler, P., & Sweller, J. (1999). Managing split-attention and redundancy in multimedia instruction. *Applied Cognitive Psychology*, *13*(4), 351-371.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv., 37*(2), 83-137.

Kelleher, C., Pausch, R., & Kiesler, S. (2007). Storytelling alice motivates middle school girls to learn computer programming. *In Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 1455-1464). San Jose, California, USA: ACM.

Klassner, F., & Anderson, S. (2003). LEGO MindStorms: not just for K-12 anymore. *Robotics & Automation Magazine, IEEE*, *10*(2), 12-18.

Klassner, F. (2002). A case study of LEGO Mindstorms' suitability for artificial intelligence and robotics courses at the college level. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education* (pp. 8-12). Cincinnati, Kentucky: ACM.

Kölling, M. (2009). *Greenfoot Tutorial*. Retrieved March 26, 2009, from http://www.greenfoot.org/doc/tutorial/tutorial.html.

Kouznetsova, S. (2007). Using BlueJ and Blackjack to teach object-oriented design concepts in CS1. *J. Comput. Small Coll.*, *22*(4), 49-55.

Larkin, J., & Simon, H. (1987). Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science*, *11*(1), 100, 65.

Lorenzen, T., & Sattar, A. (2008). Objects first using Alice to introduce object constructs in CS1. *SIGCSE Bull.*, *40*(2), 62-64.

Mayer, R. E. (1981). The Psychology of How Novices Learn Computer Programming. *ACM Comput. Surv.*, *13*(1), 121-141.

Mayer, R. E. (2001). *Multimedia Learning*. Cambridge, England: Cambridge University Press.

Mayer, R. E. (2003). The promise of multimedia learning: using the same instructional design methods across different media. *Learning and Instruction*, *13*(2), 125-139.

Mayer, R. E., & Anderson, R. B. (1991). Animations need narrations: An experimental test of a dual-coding hypothesis. *Journal of Educational Psychology*, *83*(4), 484-90.

Mayer, R. E., & Moreno, R. (1998). A cognitive theory of multimedia learning: Implications for design principles. In Naryana (Ed.)*, Electronic Proceedings of the CHI'98 Workshop on Hyped-Media to Hyper-Media: Toward Theoretical Foundations of Design, Use and Evaluation.* Retrieved March 23, 2009, from http://www.unm.edu/~moreno/PDFS/chi.pdf

Mayer, R. E., & Moreno, R. (2003). Nine Ways to Reduce Cognitive Load in Multimedia Learning. *Educational Psychologist*, *38*(1), 43-52.

van Merriënboer, J. J. G., & Sweller, J. (2005). Cognitive load theory and complex learning: Recent developments and future directions. *Educational Psychology Review*, *17*(2), 147-177.

Miller, P., Pane, J., Meter, G., & Vorthmann, S. (1994). Evolution of novice programming environments: the structure editors of Carnegie Mellon University. *Interactive Learning Environments, 4*(2), 140-158.

Moskal, B., Lurie, D., & Cooper, S. (2004). Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education* (pp. 75-79). Norfolk, Virginia, USA: ACM.

Mullins, P., Whitfield, D., & Conlon, M. (2009). Using Alice 2.0 as a first language. *J. Comput. Small Coll., 24*(3), 136-143.

Mullins, P. M., & Conlon, M. (2008). Engaging students in programming fundamentals using alice 2.0. In *Proceedings of the 9th ACM SIGITE conference on Information technology education* (pp. 81-88). Cincinnati, OH, USA: ACM.

Mwangi, W., & Sweller, J. (1998). Learning to Solve Compare Word Problems: The Effect of Example Format and Generating Self-Explanations. *Cognition & Instruction, 16*(2), 173.

Paas, F., Renkl, A., & Sweller, J. (2003). Cognitive Load Theory and Instructional Design: Recent Developments. *Educational Psychologist, 38*(1), 1-4.

Paas, F., Tuovinen, J. E., van Merriënboer, J. J. G., & Darabi, A. A. (2005). A motivational perspective on the relation between mental effort and performance: Optimizing learner involvement in instruction. *Educational Technology Research & Development, 53*(3), 25-34.

Paivio, A. (1975). Coding distinctions and repetition effects in memory. In G.H. Bower (Ed.) , *The psychology of learning and motivation*. (Vol. 9, pp. 179-214). New York: Academic Press.

Paivio, A. (1983). Empirical case for dual coding. In J. Yuille (Ed.)*, Imagery, memory, and cognition: Essays in honor of Allan Paivio* (pp. 307-332). Hillsdale, NJ: Lawrence Erlbaum Associates.

Paivio, A. (1986). *Mental Representations: A Dual Coding Approach*. Oxford, England: Oxford University Press.

Paivio, A., & Lambert, W. (1981). Dual Coding and Bilingual Memory. *Journal of Verbal Learning and Verbal Behavior, 20*(5), 532-39.

Patterson, A., Kölling, M., & Rosenberg, J. (2003). Introducing unit testing with BlueJ. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education* (pp. 11-15). Thessaloniki, Greece: ACM.

Pollock, E., Chandler, P., & Sweller, J. (2002). Assimilating complex information. *Learning and Instruction, 12*(1), 61-86.

Powers, K., Ecott, S., & Hirshfield, L. M. (2007). Through the looking glass: teaching CS0 with Alice. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education* (pp. 213-217). Covington, Kentucky, USA: ACM.

Ragonis, N., & Ben-Ari, M. (2005). On understanding the statics and dynamics of object-oriented programs. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education* (pp. 226-230). St. Louis, Missouri, USA: ACM.

Ramalingam, V., LaBelle, D., & Wiedenbeck, S. (2004). Self-efficacy and mental models in learning to program. *SIGCSE Bull., 36*(3), 171-175.

Sharad, S. (2007). Introducing Embedded Design Concepts to Freshmen and Sophomore Engineering Students with LEGO MINDSTORMS NXT. In *The 2007 IEEE International Conference on Microelectronic Systems Education.* (pp. 119-120). San Diego, CA: IEEE Computer Society Press.

Soloway, E., Jackson, S. L., Klein, J., Quintana, C., Reed, J., Spitulnik, J., et al. (1996). Learning theory in practice: case studies of learner-centered design. In *Proceedings of the SIGCHI conference on*

Human factors in computing systems: common ground (pp. 189-196). Vancouver, British Columbia, Canada: ACM.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science, 12*(2), 257-285.

Sweller, J., van Merrienboer, J., & Paas, F. (1998). Cognitive Architecture and Instructional Design. *Educational Psychology Review*, *10*(3), 251-296.

Tarmizi, R. A., & Sweller, J. (1988). Guidance during Mathematical Problem Solving. *Journal of Educational Psychology*, *80*(4), 424-36.

Van Merriënboer, J. J. G., Kester, L., & Paas, F. (2006). Teaching complex rather than simple tasks: balancing intrinsic and germane load to enhance transfer of learning. *Applied Cognitive Psychology, 20*(3), 343-352.

Wallen, E., Plass, J. L., & Brünken, R. (2005). The Function of Annotations in the Comprehension of Scientific Texts: Cognitive Load Effects and the Impact of Verbal Ability. *Educational Technology Research & Development*, *53*(3), 59-72.

Wiedenbeck, S. (2005). Factors affecting the success of non-majors in learning to program. In *Proceedings of the first international workshop on Computing education research.* (pp. 13-24). Seattle, WA, USA: ACM.

Wilson, B. C., & Shrock, S. (2001). Contributing to success in an introductory computer science course: a study of twelve factors. *SIGCSE Bull., 33*(1), 184—188.

Wouters, P., Paas, F., & van Merriënboer, J. J. G. (2008). How to optimize learning from animated models: A review of guidelines based on cognitive load. *Review of Educational Research*, *78*(3), 645-675.

Xinogalos, S., Satratzemi, M., & Dagdilelis, V. (2007). Teaching java with BlueJ: a two-year experience. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education* (pp. 345-345). Dundee, Scotland: ACM.

Yeung, A. S., Jin, P., & Sweller, J. (1998). Cognitive load and learner expertise: split-attention and redundancy effects in reading with explanatory notes. *Contemporary Educational Psychology*, *23*(1), 1-21.

Zweben, S. (2008, May) Ph.D. Production Exceeds 1,700; Undergraduate Enrollment Trends Still Unclear. *Computing Research News*, Vol. 20 (No. 3) 6-17 Retrieved December 10, 2008, from http://www.cra.org/CRN/articles/may08/taulbee.html

Appendix:

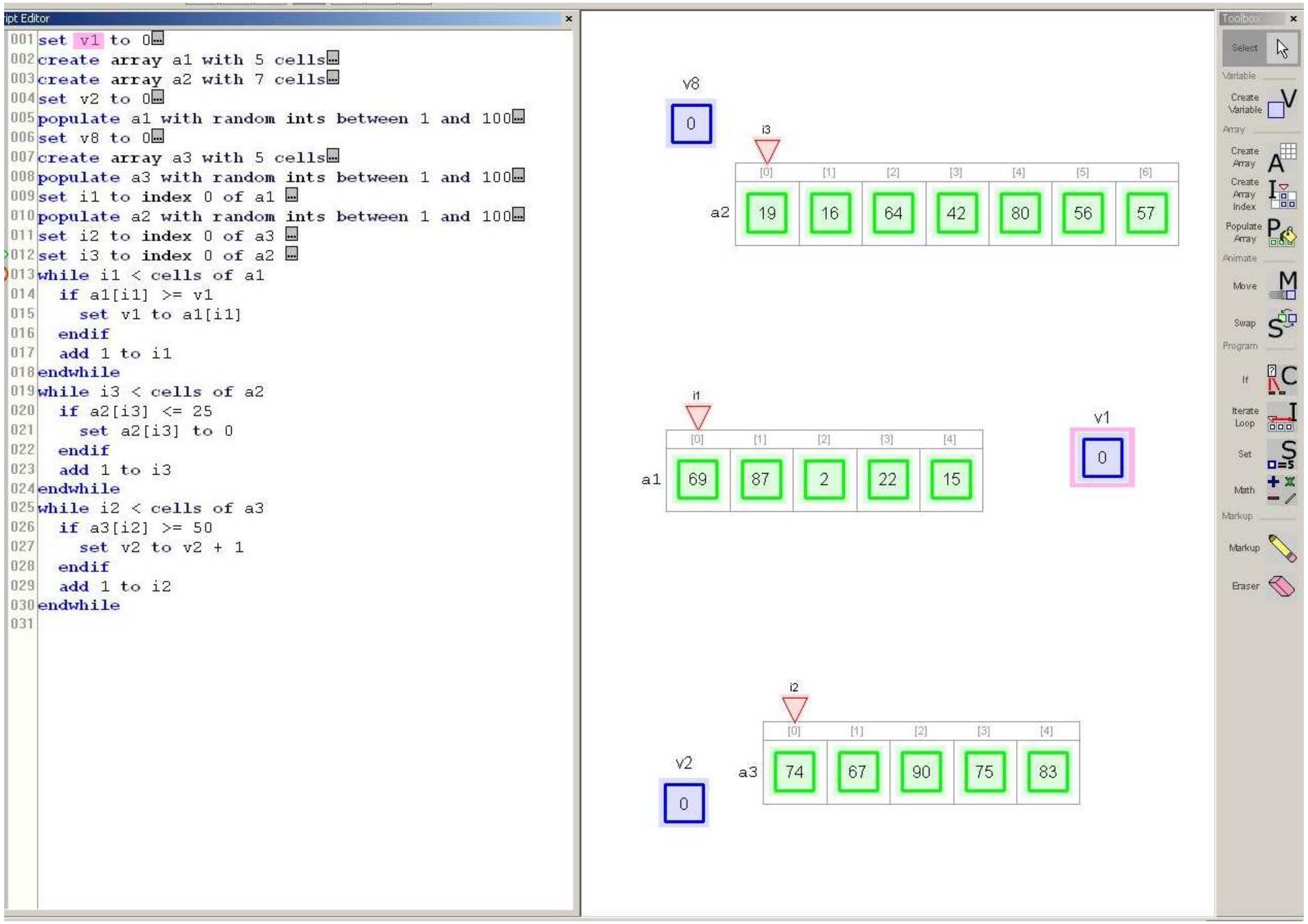
ALVIS Learning Environment Pilot Study 1
Version C1

Participant Code: ________________________ (Supplied by your TA)
Date: _________

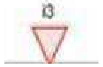
Please answer each question to the best of your ability with the information given. If you have any questions, feel free to ask the researcher.

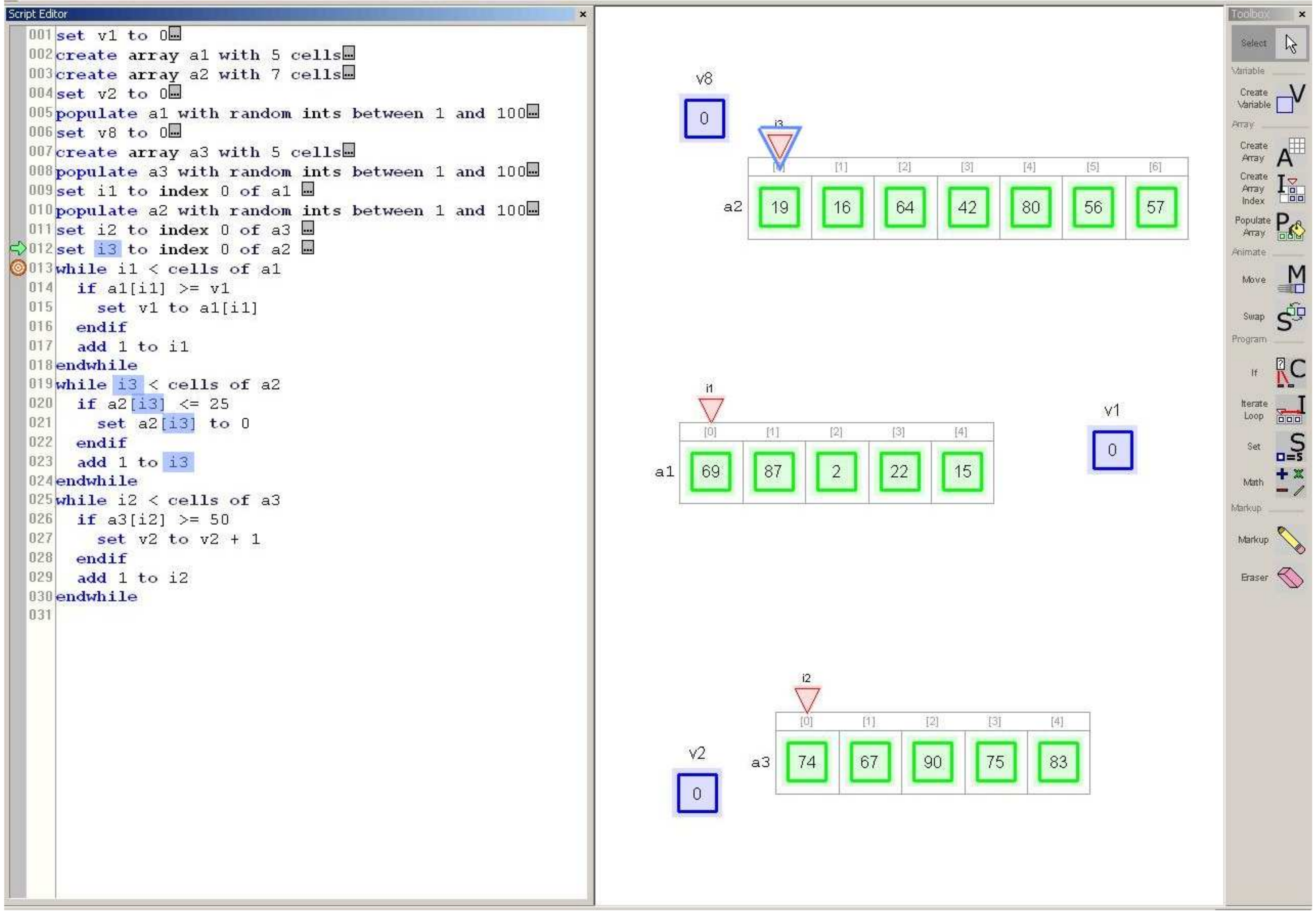
1. Time you started this questionnaire: ____:_____ AM __ PM __

2. Locate and circle the graphical representation of variable `v1` in the *right-hand* pane of the image that follows.

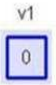
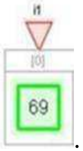


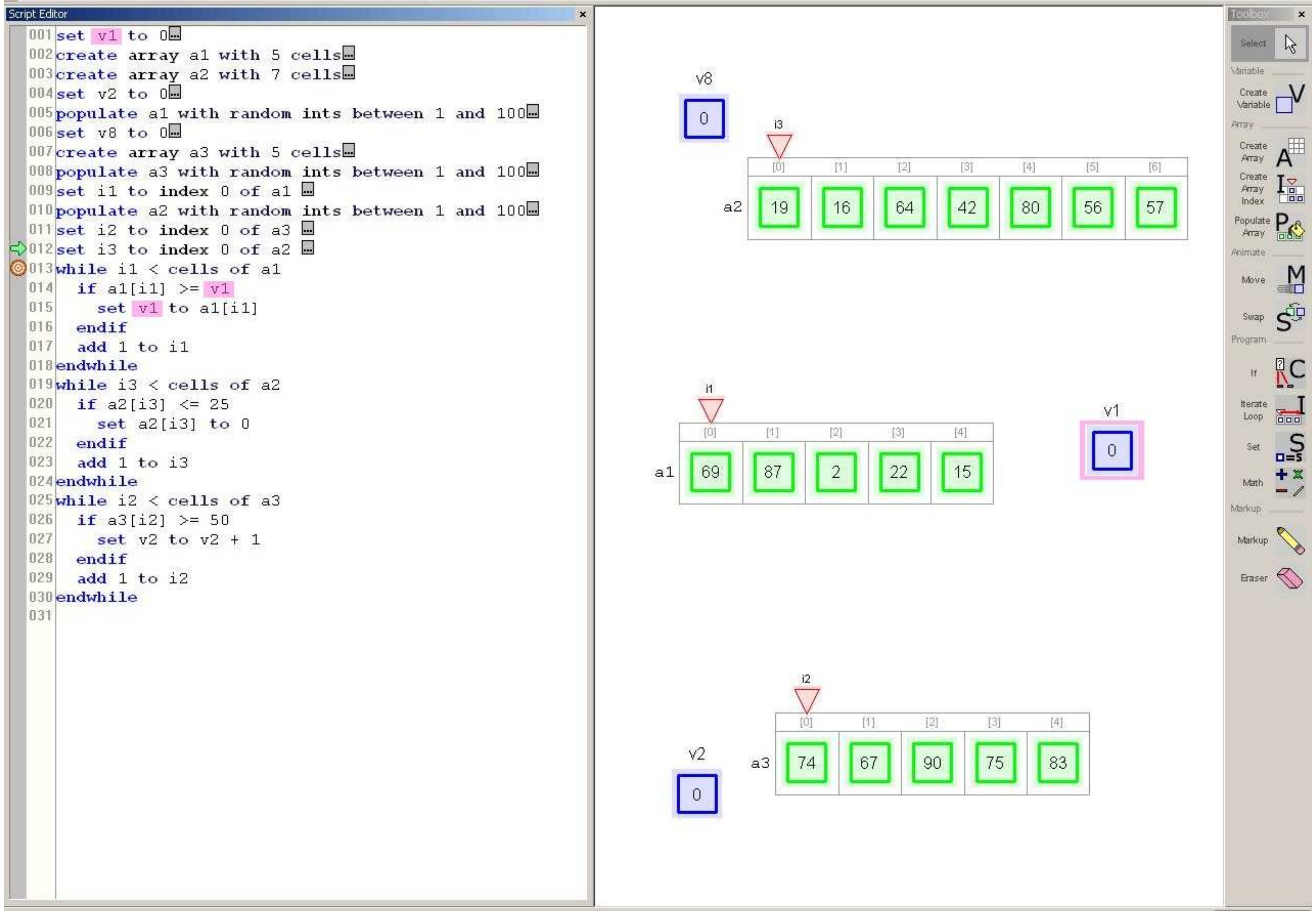
```
set v1 to 0
create array a1 with 5 cells
create array a2 with 7 cells
set v2 to 0
populate a1 with random ints between 1 and 100
set v8 to 0
create array a3 with 5 cells
populate a3 with random ints between 1 and 100
set i1 to index 0 of a1
populate a2 with random ints between 1 and 100
set i2 to index 0 of a3
set i3 to index 0 of a2
while i1 < cells of a1
  if a1[i1] >= v1
    set v1 to a1[i1]
  endif
  add 1 to i1
endwhile
while i3 < cells of a2
  if a2[i3] <= 25
    set a2[i3] to 0
  endif
  add 1 to i3
endwhile
while i2 < cells of a3
  if a3[i2] >= 50
    set v2 to v2 + 1
  endif
  add 1 to i2
endwhile
```

3. Locate and underline every reference to  in the code of the *left-hand* pane of the following image.

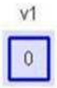


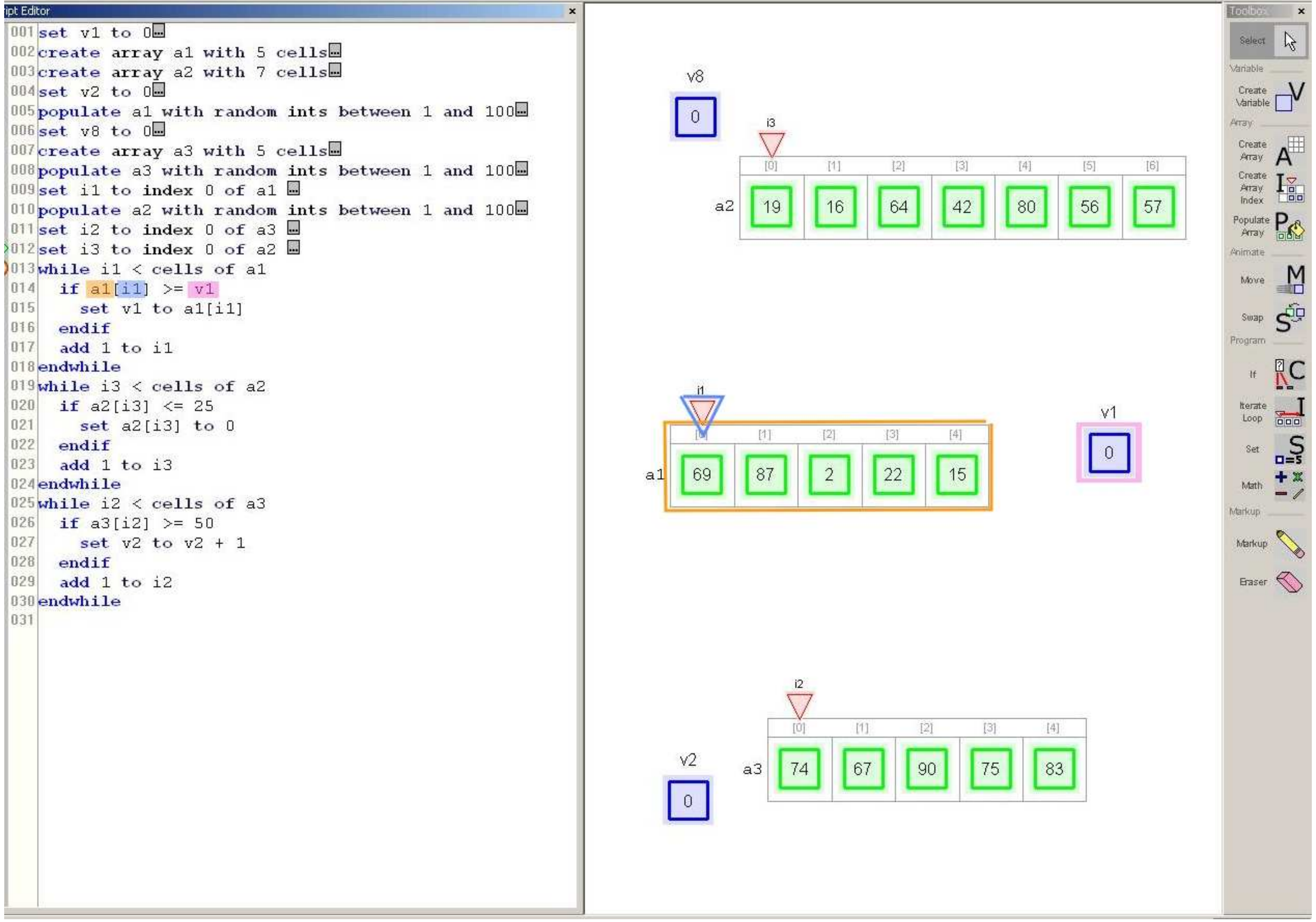
```
set v1 to 0
create array a1 with 5 cells
create array a2 with 7 cells
set v2 to 0
populate a1 with random ints between 1 and 100
set v8 to 0
create array a3 with 5 cells
populate a3 with random ints between 1 and 100
set i1 to index 0 of a1
populate a2 with random ints between 1 and 100
set i2 to index 0 of a3
set i3 to index 0 of a2
while i1 < cells of a1
  if a1[i1] >= v1
    set v1 to a1[i1]
  endif
  add 1 to i1
endwhile
while i3 < cells of a2
  if a2[i3] <= 25
    set a2[i3] to 0
  endif
  add 1 to i3
endwhile
while i2 < cells of a3
  if a3[i2] >= 50
    set v2 to v2 + 1
  endif
  add 1 to i2
endwhile
```

4. On the screen shown on the following page, locate and underline the code that compares  with .



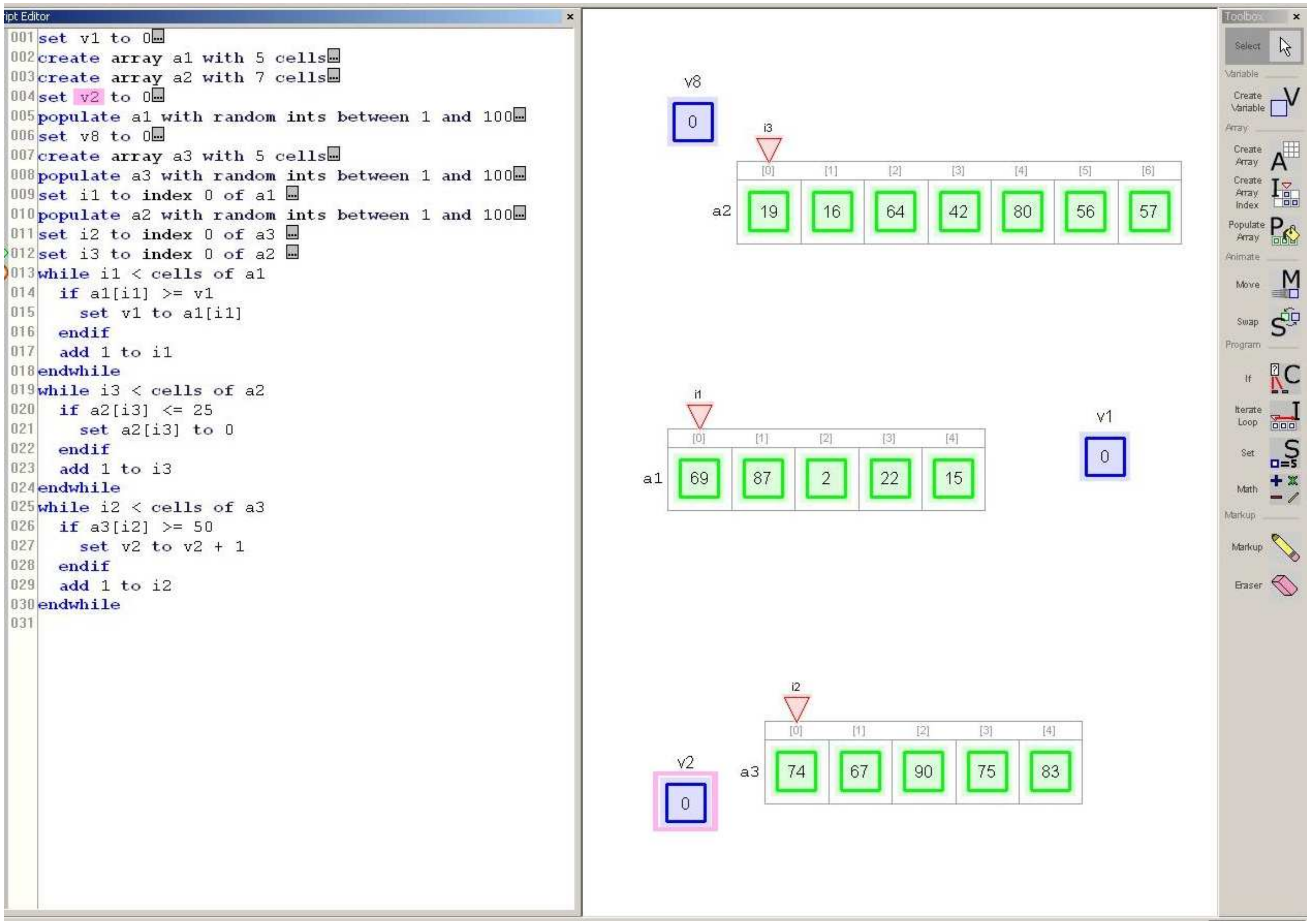
```
set v1 to 0
create array a1 with 5 cells
create array a2 with 7 cells
set v2 to 0
populate a1 with random ints between 1 and 100
set v8 to 0
create array a3 with 5 cells
populate a3 with random ints between 1 and 100
set i1 to index 0 of a1
populate a2 with random ints between 1 and 100
set i2 to index 0 of a3
set i3 to index 0 of a2
while i1 < cells of a1
  if a1[i1] >= v1
    set v1 to a1[i1]
  endif
  add 1 to i1
endwhile
while i3 < cells of a2
  if a2[i3] <= 25
    set a2[i3] to 0
  endif
  add 1 to i3
endwhile
while i2 < cells of a3
  if a3[i2] >= 50
    set v2 to v2 + 1
  endif
  add 1 to i2
endwhile
```

5. In the program below, locate the same code you underlined in the previous problem, but **change it** so that after it is executed,  will be unchanged (IE, change it so that `set v1 to a1[i1]` never runs). Write this code under the screenshot.



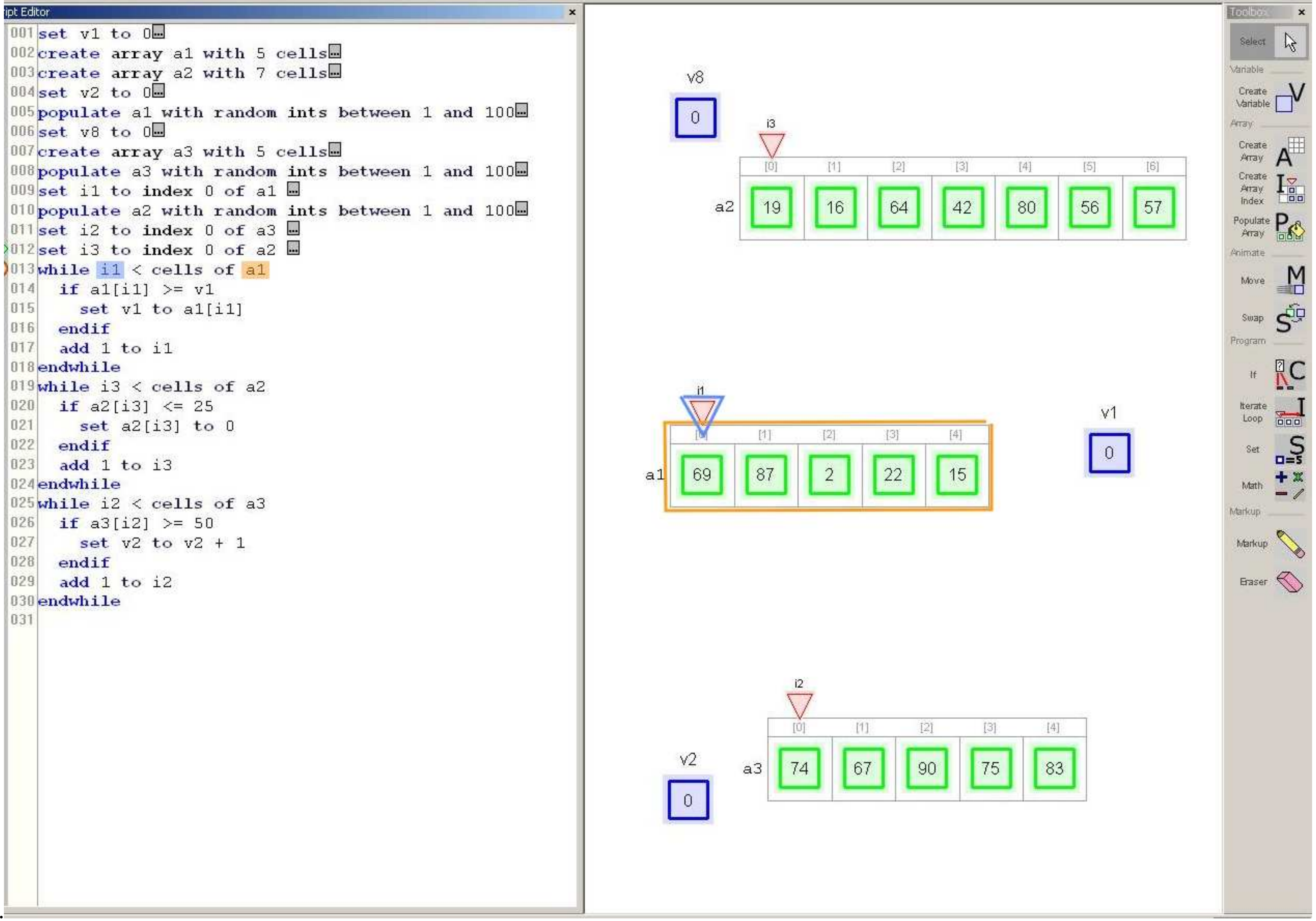
```
set v1 to 0
create array a1 with 5 cells
create array a2 with 7 cells
set v2 to 0
populate a1 with random ints between 1 and 100
set v8 to 0
create array a3 with 5 cells
populate a3 with random ints between 1 and 100
set i1 to index 0 of a1
populate a2 with random ints between 1 and 100
set i2 to index 0 of a3
set i3 to index 0 of a2
while i1 < cells of a1
  if a1[i1] >= v1
    set v1 to a1[i1]
  endif
  add 1 to i1
endwhile
while i3 < cells of a2
  if a2[i3] <= 25
    set a2[i3] to 0
  endif
  add 1 to i3
endwhile
while i2 < cells of a3
  if a3[i2] >= 50
    set v2 to v2 + 1
  endif
  add 1 to i2
endwhile
```

6. If line 4 of the program below were changed to `set v2 to 3,` what would the graphical representation of v2 look like after this line (and only this line) was executed? Draw the new v2 next to the old v2:



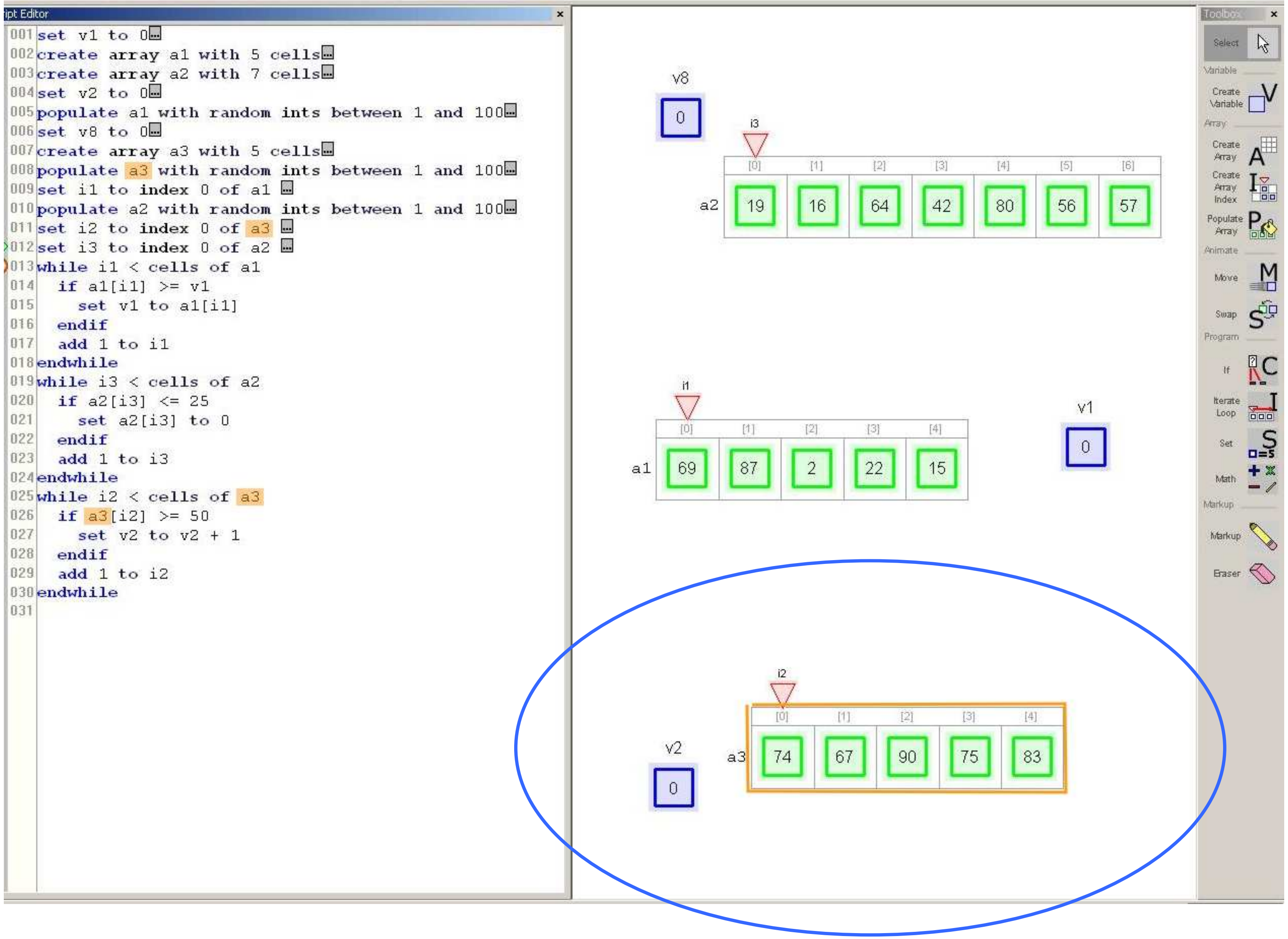
```
001 set v1 to 0
002 create array a1 with 5 cells
003 create array a2 with 7 cells
004 set v2 to 0
005 populate a1 with random ints between 1 and 100
006 set v8 to 0
007 create array a3 with 5 cells
008 populate a3 with random ints between 1 and 100
009 set i1 to index 0 of a1
010 populate a2 with random ints between 1 and 100
011 set i2 to index 0 of a3
012 set i3 to index 0 of a2
013 while i1 < cells of a1
014   if a1[i1] >= v1
015     set v1 to a1[i1]
016   endif
017   add 1 to i1
018 endwhile
019 while i3 < cells of a2
020   if a2[i3] <= 25
021     set a2[i3] to 0
022   endif
023   add 1 to i3
024 endwhile
025 while i2 < cells of a3
026   if a3[i2] >= 50
027     set v2 to v2 + 1
028   endif
029   add 1 to i2
030 endwhile
031
```

7. On the following screen, draw the **new** location of `i1` in the *right-hand* pane of the following image after the loop in lines 13-18 of the program has been run 3 times.



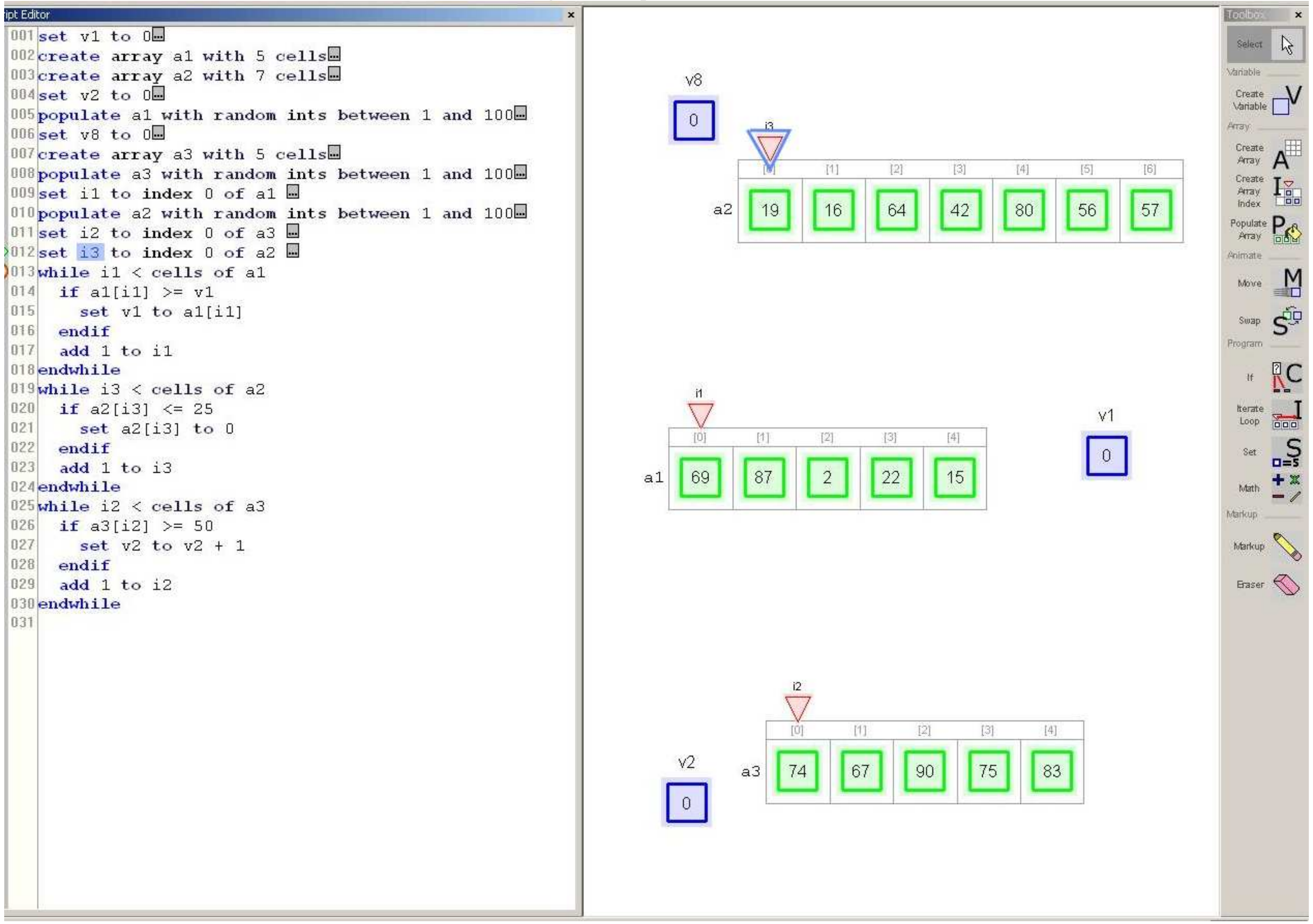
```
001 set v1 to 0
002 create array a1 with 5 cells
003 create array a2 with 7 cells
004 set v2 to 0
005 populate a1 with random ints between 1 and 100
006 set v8 to 0
007 create array a3 with 5 cells
008 populate a3 with random ints between 1 and 100
009 set i1 to index 0 of a1
010 populate a2 with random ints between 1 and 100
011 set i2 to index 0 of a3
012 set i3 to index 0 of a2
013 while i1 < cells of a1
014   if a1[i1] >= v1
015     set v1 to a1[i1]
016   endif
017   add 1 to i1
018 endwhile
019 while i3 < cells of a2
020   if a2[i3] <= 25
021     set a2[i3] to 0
022   endif
023   add 1 to i3
024 endwhile
025 while i2 < cells of a3
026   if a3[i2] >= 50
027     set v2 to v2 + 1
028   endif
029   add 1 to i2
030 endwhile
031
```

8. Locate and underline the line of code in the *left-hand* pane that creates the array shown in the circled portion of the following image. (Note: The circled portion may contain more elements than the array. Please underline only the code that creates the *array*).



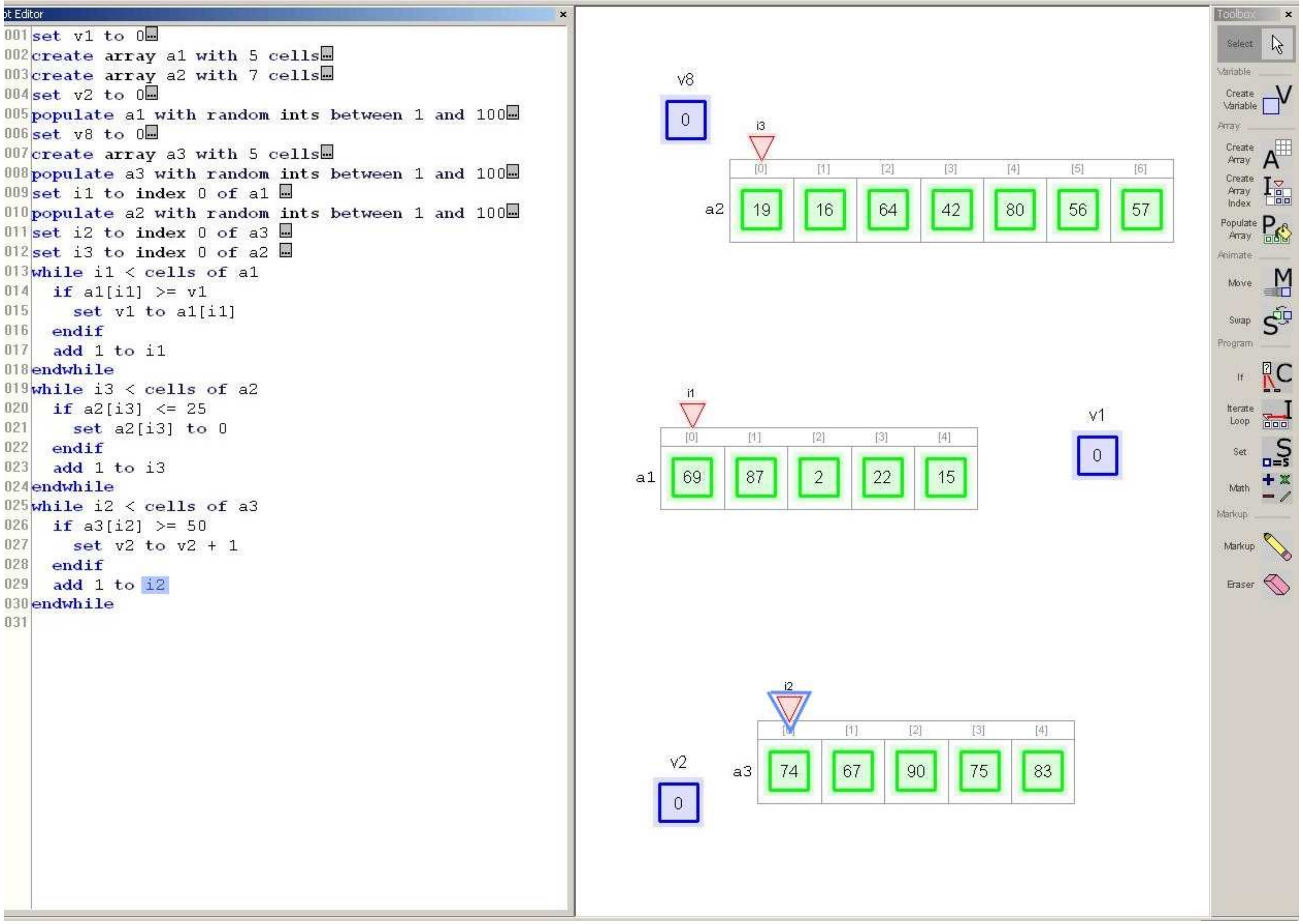
```
set v1 to 0
create array a1 with 5 cells
create array a2 with 7 cells
set v2 to 0
populate a1 with random ints between 1 and 100
set v8 to 0
create array a3 with 5 cells
populate a3 with random ints between 1 and 100
set i1 to index 0 of a1
populate a2 with random ints between 1 and 100
set i2 to index 0 of a3
set i3 to index 0 of a2
while i1 < cells of a1
  if a1[i1] >= v1
    set v1 to a1[i1]
  endif
  add 1 to i1
endwhile
while i3 < cells of a2
  if a2[i3] <= 25
    set a2[i3] to 0
  endif
  add 1 to i3
endwhile
while i2 < cells of a3
  if a3[i2] >= 50
    set v2 to v2 + 1
  endif
  add 1 to i2
endwhile
```

9. If a new line containing `set i3 to cells of a2 – 2` was inserted right after line 12 of this program, what would be the new location of i3 right after this line was executed? Draw the new location of i3 in the *right-hand* pane.



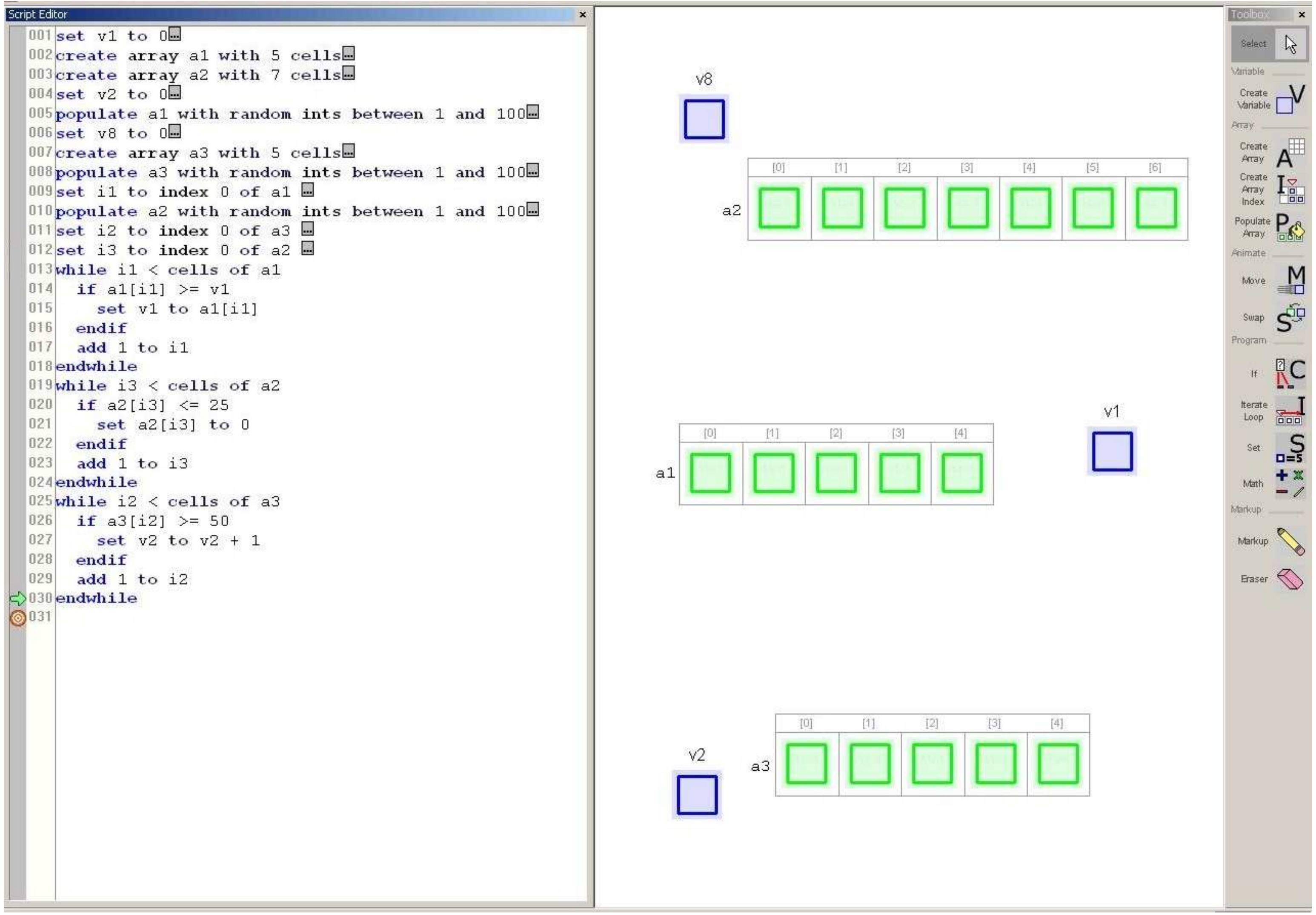
```
001 set v1 to 0
002 create array a1 with 5 cells
003 create array a2 with 7 cells
004 set v2 to 0
005 populate a1 with random ints between 1 and 100
006 set v8 to 0
007 create array a3 with 5 cells
008 populate a3 with random ints between 1 and 100
009 set i1 to index 0 of a1
010 populate a2 with random ints between 1 and 100
011 set i2 to index 0 of a3
012 set i3 to index 0 of a2
013 while i1 < cells of a1
014   if a1[i1] >= v1
015     set v1 to a1[i1]
016   endif
017   add 1 to i1
018 endwhile
019 while i3 < cells of a2
020   if a2[i3] <= 25
021     set a2[i3] to 0
022   endif
023   add 1 to i3
024 endwhile
025 while i2 < cells of a3
026   if a3[i2] >= 50
027     set v2 to v2 + 1
028   endif
029   add 1 to i2
030 endwhile
031
```

1. If line 29 were removed from the following program, where would i2 be after the loop in lines 25-30 of the program has run 5 times? Draw the new location of i2 in the *right-hand* pane.



```
001 set v1 to 0
002 create array a1 with 5 cells
003 create array a2 with 7 cells
004 set v2 to 0
005 populate a1 with random ints between 1 and 100
006 set v8 to 0
007 create array a3 with 5 cells
008 populate a3 with random ints between 1 and 100
009 set i1 to index 0 of a1
010 populate a2 with random ints between 1 and 100
011 set i2 to index 0 of a3
012 set i3 to index 0 of a2
013 while i1 < cells of a1
014   if a1[i1] >= v1
015     set v1 to a1[i1]
016   endif
017   add 1 to i1
018 endwhile
019 while i3 < cells of a2
020   if a2[i3] <= 25
021     set a2[i3] to 0
022   endif
023   add 1 to i3
024 endwhile
025 while i2 < cells of a3
026   if a3[i2] >= 50
027     set v2 to v2 + 1
028   endif
029   add 1 to i2
030 endwhile
031
```

11. When the program finishes execution, what will the *right-hand* graphical pane look like? Please draw the new location of the array iterators and the values contained in the variables and arrays.



```
set v1 to 0
create array a1 with 5 cells
create array a2 with 7 cells
set v2 to 0
populate a1 with random ints between 1 and 100
set v8 to 0
create array a3 with 5 cells
populate a3 with random ints between 1 and 100
set i1 to index 0 of a1
populate a2 with random ints between 1 and 100
set i2 to index 0 of a3
set i3 to index 0 of a2
while i1 < cells of a1
  if a1[i1] >= v1
    set v1 to a1[i1]
  endif
  add 1 to i1
endwhile
while i3 < cells of a2
  if a2[i3] <= 25
    set a2[i3] to 0
  endif
  add 1 to i3
endwhile
while i2 < cells of a3
  if a3[i2] >= 50
    set v2 to v2 + 1
  endif
  add 1 to i2
endwhile
```

12 Write the time you finished this questionnaire: ____:_____ AM __ PM __