THE DESIGN AND EVALUATION OF A DIRECT MANIPULATION INTERFACE

FOR NOVICE PROGRAMMERS

By

SEAN FARLEY

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 2006
To the Faculty of Washington State University:

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

__________________________
Chair

__________________________

__________________________
THE LEARNING EFFECT AND EFFICACY OF DIRECT MANIPULATION PROGRAMMING

Abstract

by Sean Farley, M.S.
Washington State University
May 2006

Chair: Christopher Hundhausen

Novices face many barriers when learning to program, including the need to learn both a new syntax and a computation model. By constraining syntax and providing visual representations on which to operate, direct manipulation programming environments might lower these barriers. However, since the learning goal of the novice is to be able ultimately to program in conventional textual languages, can direct manipulation programming environments lower the barriers to programming, and, at the same time, promote positive transfer to textual programming? To address this question, I designed a new direct manipulation programming interface for ALVIS, a novice programming environment. I then conducted an experimental study that compared the programming outcomes promoted by the new direct manipulation interface to those promoted by the textual programming interface. I found that the direct manipulation interface not only led to better initial programming outcomes, but also to significant positive transfer to the textual interface. My results show that direct manipulation interfaces can provide novices with a “way in” to traditional textual programming.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. RELATED WORK</td>
<td>5</td>
</tr>
<tr>
<td>3. INTERFACE</td>
<td>9</td>
</tr>
<tr>
<td>4. EXPERIMENT</td>
<td>24</td>
</tr>
<tr>
<td>5. CONCLUSION</td>
<td>41</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>44</td>
</tr>
</tbody>
</table>
LIST OF TABLES

1. ANOVA Results for Task Accuracy.........................................................................................37
2. ANOVA Results for Time on Task ......................................................................................37
3. DM vs. Text T-Test Results by Task.....................................................................................38
LIST OF FIGURES

1. ALVIS 2.1 .............................................................................................................. 10
2. Prototype Screenshot 1 .......................................................................................... 12
3. Prototype Screenshot 2 .......................................................................................... 13
4. Prototype Screenshot 3 .......................................................................................... 14
5. Help Window for Array Creation ........................................................................ 17
6. Array Creation ...................................................................................................... 17
7. Array Population .................................................................................................. 18
8. Variable Creation .................................................................................................. 19
9. Index Creation ...................................................................................................... 19
10. While Loop Creation ........................................................................................... 21
11. Array Component Selection Dialog .................................................................... 22
12. Conditional Selection Dialog ............................................................................. 22
13. Variable or Literal Selection Dialog .................................................................... 22
14. If Statement Creation .......................................................................................... 23
15. Set Statement Creation ....................................................................................... 23
16. The Text-only ALVIS .......................................................................................... 29
17. Plot of time-on-task means by condition and task ............................................ 33
18. Plot of accuracy score means by condition and task .......................................... 33
19. Plot of Accuracy .................................................................................................. 34
20. Plot of Time on Task ........................................................................................... 34
21. Boxplot of Accuracy ............................................................................................ 35
22. Boxplot of Time on Task ..................................................................................... 35
CHAPTER ONE
INTRODUCTION

When students first begin to learn how to program, they are faced with a significant challenge—a learning curve that many will not successfully overcome. Attrition rates commonly seen in introductory computer science courses are quite high[1]. This phenomenon can be observed here at Washington State University, where introductory computer science courses have suffered an average attrition rate of 43 percent over the past three years (n = 1,001 enrolled students). A key question arises: Why do so many students not succeed as programmers?

Pitfalls for Novice Programmers

Past computer science education research suggests that several factors might contribute to the high attrition rates of introductory computer science courses, including (a) individual student differences (see e.g., [2]), (b) a lack of a sense of community (see e.g., [3], (c) deficient pedagogical approaches, and (d) inadequate novice programming environments. This thesis addresses the last of these factors.

An extensive line of research, comprehensively reviewed in [4], has addressed the issue of inadequate novice programming environments. In an attempt to lower the barriers to programming, my colleagues and I have explored a variety of alternative novice programming environment features, including algorithm visualization (e.g., [5]), drag-and-drop editing (e.g., [6]), programming by direct manipulation and gestures (e.g., [7]), and support for the declarative constructs that novices appear to use independently of programming environments [8].
A Direct Manipulation Programming Environment

Over the past three years, I have been involved in and contributing to the above line of research through the development of a novice programming environment called ALVIS [9], which supports both (a) up-to-the-keystroke syntactic and semantic feedback through a “live” algorithm editing and visualization model, and (b) an interface for generating object creation statements by direct manipulation, in which a set of tools can be used to directly specify and place program objects in an animation window.

While a recent experimental study of the ALVIS environment [10] showed that its “live” editing and visualization model enables novices to program significantly more accurately than they could without any programming environment at all, there remained evidence that study participants still struggled to develop correct programs. In particular, in their attempts to write array iterative algorithms, many participants had trouble constructing correct loops, and referencing array elements correctly within those loops through the use of array indexes. As has been suggested by past research [11], we found that iterative constructs proved to be a key stumbling block for participants in our study.

Research Questions

Considering that study participants appeared to benefit from ALVIS’s direct manipulation interface for creating program objects, I began to explore whether the direct manipulation interface could be expanded such that additional programming tasks could be performed solely through direct manipulation. Ultimately, my goal was an interface that could be used to construct simple algorithms entirely through direct manipulation. This ambition led to the following research question:
RQ1: Can a direct manipulation interface for programming iteration, conditionals, and assignment statements in ALVIS facilitate (a) faster and more accurate programming, and (b) positive transfer to textual programming?

This question, in turn, led to a second, related research question:

RQ2: What might such a direct manipulation interface look like?

In this thesis, I address both of these questions by presenting and experimentally evaluating a new direct manipulation programming interface for the ALVIS software. The new interface enables one to write iterative, conditional, assignment, and arithmetic operations through a combination of filling in dialog boxes and directly manipulating objects in ALVIS’s animation window. Coupled with ALVIS’s existing direct manipulation interface for object creation, the new interface makes it possible for users to specify, without having to type in any textual code, the kinds of single procedure, array iterative algorithms that students explore in the initial portions of many introductory computer science courses.

Transfer of Training

Given that the ultimate purpose of ALVIS is to train introductory computer science students to program in conventional textual languages, a key objective of the direct manipulation interface explored here is to facilitate positive transfer of training to textual programming. The term “transfer of training” refers to knowledge or abilities acquired in one area that can then be used to solve problems or perform tasks in another
area. Transfer of training is based on the Theory of Transfer of Learning, which was introduced by Thorndike and Woodworth in 1901 [12]. Thorndike and Woodworth explored how individuals would transfer learning gained in one context to a different but similar context. Their theory suggested that transfer of learning depends on the learning and transfer tasks being identical. Positive transfer of training was defined by Baldwin and Ford [13] as the degree to which individuals were able to apply in their jobs the skills gained in training situations. Holding states that “transfer of training occurs whenever the effects of prior learning influence the performance of a later activity [14].”

My focus on transfer of training separates the research presented here from past efforts to develop direct manipulation environments for the sole purpose of easing the programming task. As we shall see, in an experimental study, my new direct manipulation interface not only facilitated significantly faster and more accurate programming than the text-based ALVIS interface, but also promoted positive transfer to textual programming.

**Thesis Outline**

The remainder of this thesis is organized as follows. Chapter 2 discusses the relevant filed of research and reviews related work. Chapter 3 describes the ALVIS software and the design process for additions to this software. In Chapter 4 I present the design and results of my experimental evaluation. Finally, in Section 5 I summarize my contributions, and outline directions for future research.
CHAPTER TWO
RELATED WORK

I have established that learning programming can be difficult for novices, and identified the programming interface as the primary concern of this thesis. The style of interface which I have developed and tested employs “direct manipulation” to make novice programming easier to learn and perform.

Direct Manipulation

The fundamental feature of a direct manipulation interface is user control, the central concepts of which were defined by Shneiderman as being (a) visibility of actions and objects, (b) fast, reversible, and incremental actions, and (c) replacement of command language syntax with direct and visual manipulation of objects [15]. Shneiderman made these observations when GUI interfaces were just beginning to appear on the market. It is clear that the industry and the consumer have largely embraced the concept of direct manipulation since that time, as evidenced by the ascendancy of GUI operating systems, in which users have access to visual representations of files and directories which they can manipulate via drag and drop actions. A favorite example is the trash can icon located on some computer desktops. Rather than typing a command to delete a file, a user can drag a file to the trash icon and release it. Both the action and the objects are visible, and the action is both quick and reversible.

Direct manipulation concepts have been applied to programming as well. Certainly, we now use programming platforms that allow us to manipulate program file structure in the same way that we can organize general computer files, but this is not
exactly programming by direct manipulation. In order to qualify as offering direct manipulation programming, an interface must enable users to construct algorithms without having to type code in the traditional fashion.

The research presented here develops and experimentally evaluates a direct manipulation interface that enables novices to construct elementary, array-iterative, imperative algorithms without having to type code in the traditional fashion. A large body of work, some of which is surveyed in [16], shares my interest in transforming programming tasks that have traditionally been text-based into ones that can be performed by direct manipulation and demonstration. For example, Burnett and Gottfried [17] describe an extension to the Forms/3 spreadsheet environment that allows users to program graphical objects by direct manipulation, as opposed to by specifying formulas. Likewise, Stasko [18] presents an environment in which programmers can specify algorithm animations by direct manipulation, rather than by writing complex C code.

In order to make programming more accessible to children who are first learning to program, numerous novice programming environments have explored direct manipulation and demonstrational techniques. For example, in Lego Mindstorms [19], children can program robots by dragging iconic representations of commands from a palette onto a workspace, where they can be interfaced together to create a program. In Stagecast Creator [7], children specify simulations by demonstrating graphical rewrite rules in a grid-based world. HANDS [20], another programming system intended for children, represents computation with a character sitting at a table, who reacts to program events. Many tasks in HANDS can be specified more naturally than in typical
programming languages. In Tinker [21], novices specify examples through a combination of textual programming and direct manipulation of objects; with the user’s assistance, Tinker attempts to generalize the examples into Lisp procedures.

Wilcox et al [22] examined how continuous visual feedback, or “liveness,” might help with programming—specifically, how it might aide the debugging process. Participants in their study were asked to debug two programs, receiving immediate feedback for only one of the tasks. The results were mixed, but the researchers concluded that, while continuous visual feedback did not aide debugging in general, it was helpful with debugging in some situations.

Ko and Myers [23] sought to define the barriers in the programming environment which prevented novices from learning programming skills. They performed a study of beginning programmers using Visual Basic.NET, and identified six types of barriers, which included design, selection, coordination, use, understanding, and information. They suggested that a more learner-centric approach to programming design could be devised with these barriers in mind.

Most closely related to the direct manipulation programming interface explored here is a family of novice programming environments that have been used to teach the imperative programming paradigm commonly explored in undergraduate computer science courses. Like ALVIS, many of these environments generate visual representations of program execution (e.g., [5, 16]). In addition, some of these environments enable the learner to specify a program at least partly by direct manipulation. For example, ALICE [24] and JPie [6] provide drag-and-drop code editors that prevent syntax errors. In RAPTOR [25], the user writes algorithms by laying out a
flowchart by direct manipulation; however, the commands within each element of the
flowchart (e.g., conditional and assignment statements) must still be specified textually.
In Jeliot [5], program execution steps are animated, while the source code displayed. The
animation can be executed continuously or step-by-step.

While numerous direct manipulation and demonstrational programming interfaces
have been developed, few have been subjected to experimental evaluation in order to
determine whether they actually ease the programming task. In one of the few such
evaluations, Burnett and Gottfried [17] compared the speed and accuracy with which
users could construct graphics using (a) direct manipulation and (b) textual formulas
within the Forms/3 spreadsheet environment. Their results indicated that users could
perform programming tasks significantly faster and more accurately with the direct
manipulation interface. In another study, Mudugno et al. [26] compared the accuracy
with which users could create (by demonstration) and comprehend desktop file
manipulation programs written in a comic strip-based and text-based representational
language. Participants were able to construct significantly more accurate programs using
the comic strip-based language, and were able to better comprehend the comic strip-
based programs they generated.

My research most resembles these last two experimental comparisons, but in an
effort to distinguish my research from others in the field, my experiment goes further, and
tries to identify and evaluate positive transfer-of-training that might occur when users
first use a direct manipulation interface, and then are asked to program with a textual
interface.
CHAPTER THREE

INTERFACE

Introduction to ALVIS

Central to my thesis is the software named ALVIS (ALgorithm VIsualization Storyboarder), which I will describe in this chapter. ALVIS is a programming environment intended for use by novice programmers. Users can program simple algorithms in a pseudocode-like language called “SALSA” in one of two ways: they can type the commands directly into a text window called the “Script Window,” or use direct manipulation tools to specify commands by placing and directly manipulating objects in what is called the “Animation Window.” For example, to create a variable, the user could type “set v1 to 0” into the Script Window, or select the Create Variable tool in the toolbox and then click in the Animation Window, where a visual representation of a variable will appear. In either case, the visual or textual counterpart is created at the same time: typing the SALSA code creates a visual object in the Animation Window, and creating the object with direct manipulation also creates the SALSA code in the Script Window (Figure 1).

Placed in the Script Window, in the left margin of the code text, is an “Execution Arrow” that indicates which line of code was most recently executed. The Execution Arrow can be advanced or reversed using the “Execution Controls” at the top of the ALVIS environment. The Execution Controls allow for stepping forwards or backwards
Figure 1: ALVIS 2.1

one line at a time, playing forwards or backwards at user defined speeds, advancing to the end of the code, or returning to the start. When a line of code is being edited, or is created via direct manipulation, the Execution Arrow moves to that line. As code is executed, the Animation Window updates to reflect the current state.

ALVIS 2.0

The previous iteration of ALVIS, known as ALVIS 2.0, provided direct manipulation tools for the creation of the basic data components, which include variables, arrays, and array indexes. Each of these can be created with the tools Create Variable,
Create Array, and Create Index respectively. Arrays can be populated with data using the Populate Tool. A Swap tool allows users to swap variables, and a Move tool allows users to physically move objects in the Animation Window. All of these tools exist in the new version of ALVIS. Other tools which were present in ALVIS 2.0 but have been removed are Glue, Say, Compare, and Flash.

**New Proposed Design**

In order to test the hypothesis, it was necessary for me to add direct manipulation tools such that a simple algorithm could be built strictly through direct manipulation. I examined several algorithms, and identified four potential tools as necessary. These were Accumulate, Iterate, If, and Set. In order to facilitate the use of these tools, it became clear to me that dialog windows would be required to elicit information from the user about the program statement. For instance, when using the If tool, the user would need to identify which conditional operator would be used. Finally, I devised a series of help windows to steer the user towards the right path, intending to employ these windows for both new and old tools.

**Design Process**

I developed the new direct manipulation interface for ALVIS through a user-centered design process. This approach is preferable to “armchair” design, in which a designer proceeds to implement a design without any user input. Typically, when armchair design is employed, assumptions are made about the users’ abilities and perceptions, resulting in a product that is unintuitive to the user. Using the original
ALVIS as a starting point, I first constructed a preliminary low fidelity prototype of the new direct manipulation interface. The prototype consisted of a series of static screens that were created by doctoring screenshots of the original ALVIS, examples of which can be seen in figures 2-4. To test and refine the interface, I ran a “wizard of oz” prototype study, for which I recruited 15 volunteers out of the Fall, 2005 offering of the introductory computer science course at Washington State University. In this type of study, subjects are presented with static screens representing interface states. Subjects describe their actions, and the facilitator shows them pictures which correspond to the state their actions would lead to, based upon the current design. For my study I used a
smart board, which allowed me to dynamically alter the screens if the participants’ actions led to unanticipated situations. The design gradually evolved into its final form through five design iterations, each of which consisted of input from three to five participants.

In a typical session, the participant was presented with the first screen, which is the initial ALVIS screen at startup. The participant was first asked to “explore” the interface for a few minutes. This exploration consisted of the participant describing an action to the facilitator, such as moving the mouse or pressing a button, and the facilitator responded either by describing a result or physically drawing information onto the screen.
After this introduction, the participant was asked to create the “find max” algorithm, in which the largest element of an array is identified. The first steps would be to create an array, a variable, and an array index, and screenshots exist for these, but only in one order. If the participant chose to create these objects in an order not represented by the screenshots, the facilitator drew the results by hand onto the screenshot currently being displayed.

Following the creation of objects, the participant would need to create a while loop, an if statement, and a set statement. Again, this was the anticipated order, but...
participants sometimes skipped the while loop, forcing the facilitator to improvise screenshots on the fly.

After finishing the find max algorithm, participants were asked to complete two more algorithms, but in these cases all objects were already created for them. The additional algorithms were needed to test the Accumulate tool, and to test situations in which a literal value was required.

Results

The Accumulate tool was an early casualty of the design process. The tool was captioned “Accum,” and this term was less than clear to most participants. The tool image used a “++,” but this derives from the C language, and is not always meaningful to beginning programmers. Most participants sought in vain for a “Math” tool. I decided that a Math tool would indeed be a better solution, capable of everything the Accumulate tool was, and more besides. The Math tool replaced the Accumulate tool during the design process, and subsequent testing sessions proved its worth.

The means of introducing literal values into program statements took on several forms during the design process. Few beginning programmers recognize what “literal value” means, and so I changed the term to “number.”

The Iterate Tool was initially captioned “Iterate,” but this term was not readily identifiable by the participants. After I added “loop” to the caption, participants had no trouble locating the tool when needed.

When selecting array components, participants had a wide range of approaches and expectations. Participants needed to select an array cell referenced by an index. To
achieve this, participants sometimes clicked on the index, sometimes on an array cell, and sometimes on the cell number bar above the array. I determined that an array component selection dialog would be needed to allow users to select the array component appropriate to their intentions, shown in figure 11 in the next section.

Walkthrough

To illustrate how the new DM interface works, we now step through a sample session in which we use the new interface to code the simple “Find Max” algorithm, which identifies the largest value in an array.

To code this algorithm, we begin by creating the array of numbers to be searched. First, we activate the Create Array tool in the Toolbox by clicking on it. At this point, an ALVIS help window appears with advice on how to create the array (Figure 5). These windows appear during direct manipulation programming in order to aid users with the interface, and can be disabled individually or globally by checking one of two check boxes at the bottom of the window. Positioning the cursor in the top center of the Animation Window, we now drag out an array with five columns. When we release the mouse button, the code segment `create array a1 with 5 cells` appears on the first line of the script editor; the execution arrow adjacent to this line of code indicates that it has just been executed (Figure 6).

Next, we need to populate the array with numbers. To do this, we first click on the Populate tool to activate it. A help window appears, suggesting that we click on an array cell. After clicking anywhere in the array, we see the array fill with values. Simultaneously, the SALSA code `populate a1 with random ints between`
Figure 5: Help window for array creation

![Help window for array creation]

Figure 6: Array creation

1 and 100 appears on the second line of the Script Window. The execution arrow advances to this line to indicate that it has just been executed (Figure 7). We now need to create the variable maxSoFar. We accomplish this by (a) clicking on the Create Variable tool, (b) positioning the cursor inside the Animation Window, and (c) clicking
to create a variable at the cursor’s location. At this point, the variable is named v1. We can change the variable’s name by double-clicking on the variable, which brings up a dialog box that allows us to modify the variable’s properties. Changing the name in the variable name field and clicking the accept button gives us our variable maxSoFar (Figure 8).

The next step is to create an index for array iteration. First, we click on the Create Index tool, which activates the tool. We then click on the left-most cell of the array. An inverted triangle appears above the cell, with the caption i1 (Figure 9). This is our array index.

Having created all of the necessary program objects we are now in a position to make use of the new direct manipulation tools to flesh out the algorithm. We first construct the loop in which we will visit each array cell. To create this, we select the Iterate Loop tool.
Figure 8: Variable creation

Figure 9: Index creation
Positioning the cursor on the \texttt{i1} index, we press the mouse down, and then drag the \texttt{i1} index to the last cell of the array. When we release the button, a three line while loop skeleton appears in the Script Window: a \texttt{while} statement, an index increment statement, and an \texttt{endwhile} statement (Figure 10). A blank line is also created, and the caret is placed on this line to indicate that the next line of code will be placed here (unless we move the caret to a different place by clicking somewhere else in the Script Window).

In order to compare the value of each array cell to the \texttt{maxSoFar} variable, we now need to create an \texttt{if} statement. To do this, we first click on the If tool, and then click on the array cell at which the \texttt{i1} index is presently positioned. An array component selection window appears with several possible interpretations of this gesture (Figure 11); we recognize \texttt{a1[i1]} as the correct left-hand side of our \texttt{if} statement, and click on it. Next, we are prompted to select a conditional from the \texttt{if} conditional menu (Figure 12), and we select “\texttt{>}.” The final window asks us to decide if the right-hand side of the \texttt{if} statement should be a variable or a number (Figure 13). Choosing variable, we use the mouse to click on \texttt{maxSoFar}; our completed \texttt{if} statement appears in the text window (Figure 14). As with the while loop, \texttt{if} and \texttt{endif} lines appear, with a blank line in between, on which the editing caret is now positioned. The last programming step involves using the Set tool to generate a statement that assigns the current array value to \texttt{maxSoFar} in the case that the current array value is greater than \texttt{maxSoFar}. We first select the Set tool, and then click on \texttt{maxSoFar} — the variable to be set. We are asked if we want to set \texttt{maxSoFar} to a variable or a number. We select “variable,” and use the mouse to click on the array cell on which \texttt{i1} is presently positioned, as this is the
variable to be assigned to maxSoFar. As before, we are presented with a window that
gives several possible interpretations of our gesture. We choose the most general of
these—a1[i1]—to complete our algorithm (Figure 15).

We can now explore the algorithm further by using the Execution Controls to step
through our code, one line at a time. As we do so, the execution arrow advances, and the
Animation Window is dynamically updated to reflect the execution results. We can also
execute to any point in the script simply by clicking on that line with the mouse.
Figure 11: Array component selection dialog

Figure 12: Conditional selection dialog

Figure 13: Variable or literal selector dialog
Figure 14: If statement creation

Figure 15: Set statement creation
CHAPTER FOUR
EXPERIMENT

Experiment

To evaluate the new direct manipulation interface, I conducted an experimental study with two main hypotheses:

H1: Novices who use the new ALVIS DM interface will be able to create algorithmic solutions significantly more quickly and accurately than students who use a text-only version of ALVIS in which code must be typed in.

H2: Students who use the new ALVIS DM interface will benefit from a significant transfer-of-training effect that will enable them to program in the text-only version of ALVIS more quickly and accurately than students who use the text-only version from the start.

To test these hypotheses, I conducted a between-subjects experimental study with two conditions defined by programming interface: Text and Direct Manipulation (DM). In the Text condition, participants used a text-only version of ALVIS for all three experimental tasks. In this software version, the only way to program was by entering textual SALSA code via the keyboard. The direct manipulation toolbar was removed from this version. In contrast, in the DM condition, for the first two experimental tasks, participants used a version of ALVIS with the new DM interface presented in the previous section, but without the ability to type in textual commands (text entry into the Script Window was turned off). Hence, participants in the DM condition had to use the
new DM tools to program their solutions to the first two tasks. For the third experimental task, participants in the DM condition switched to the text-only version of the software, thus enabling us to consider a transfer-of-training effect.

Programming outcomes were assessed according to two dependent measures—semantic accuracy and time on task. In addition, to gauge conceptual programming knowledge, we administered a multiple-choice pretest and posttest.

**Participants**

I recruited 34 students (29 male, 5 female; mean age 19.7) out of the Spring, 2006 offering of CptS 121, the introductory computer science course at Washington State University. Participants were recruited in the second week of the semester, before they had received instruction on programming. Students who self-reported any prior programming experience were excluded from the study. Participants received course credit for their participation.

**Materials and Tasks**

All participants worked on Pentium IV computers running the Windows XP operating system. Equipped with mice and keyboards, the computers had 1 GB of RAM and either a 15 or 18 inch LCD color display set to a resolution of 1024 × 768.

Participants were first asked to complete a questionnaire. A second questionnaire was administered at the end of the experiment, and the results were compared. Prior to working on the programming tasks, participants in both conditions completed an informationally-equivalent tutorial that introduced them to the software version they
would be using. The Text version of the tutorial asked students to type lines of code into the text editor, whereas the DM version of the tutorial had the students create the same code using the DM tools.

Participants in both conditions completed three isomorphic programming tasks: Find Max, Replace, and Count. In Find Max, participants were required to construct an algorithm to locate and identify the largest value in an array. In Replace, participants were required to construct an algorithm to find and replace array values smaller than 25. In Count, participants had to write an algorithm to count the number of array values larger than 50. These tasks were designed to be semantically isomorphic to each other, so that a universal grading system could be applied to the results, regardless of task. Correct SALSA solutions would look something like the following samples.

**Find Max**

create array a1 with 6 cells

populate a1 with random ints between 1 and 100

set maxSoFar to 0

set i1 to index 0 of a1

while i1 < cells of a1

    if a1[i1] > maxSoFar

        set maxSoFar to a1[i1]

    endif

    add 1 to i1

 endwhile
Replace

create array a1 with 6 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

Count

create array a1 with 6 cells
populate a1 with random ints between 1 and 100
set count to 0
set i1 to index 0 of a1
while i1 < cells of a1
    if a1[i1] > 50
        add 1 to count
    endif
    add 1 to i1
endwhile
In the DM condition, participants used the “DM” version of ALVIS for the first two tasks, and the “text” version of ALVIS (identical to the “DM” version, except that all of the DM Tools were removed) for the third task. In contrast, participants in the Text condition used the “text” version of ALVIS for all three tasks. Figure 16 shows the text version of ALVIS with the DM toolbar removed. Figure y in Chapter 3 shows the DM version, which included the DM toolbar. Note that, while typing was not permitted in the DM version, text operations like cut, paste, and delete were allowed on a line basis, and code navigation inside the text window was also possible.

We used Morae® Recorder to make lossless recordings of participants’ screens as they worked on tasks. These recordings allowed us to recreate participants’ work if needed, and to gauge their time on task.

Procedure

We used a background questionnaire to screen potential participants for prior programming experience; students who self-reported any prior programming experience were excluded from the study. The remaining students were randomly assigned to the two conditions. In order to guard against task order effects, we fully counterbalanced the order in which participants completed tasks within each condition. This meant that roughly six study participants (three per condition) performed each of the six possible task orderings.
The experiment was conducted during three two-hour- and-50-minute sessions. There were 10 participants present for the first session, and 12 for the other two. In each study session, participants first completed a 20-minute pre-test of conceptual knowledge. They then worked through a 15-minute tutorial specific to the software version they would initially use. Following the tutorial, participants were asked to start their screen recording and to begin their first task. Participants were instructed to complete each task as quickly as possible, without sacrificing accuracy, with the stipulation that each of the three tasks had to be completed in less than 35 minutes. After 35 minutes, or whenever they finished, participants were asked to save their work, stop their screen recording, and
move on to the next task. After completing the second task, DM participants were asked to complete the third task using the “text-only” interface; however, they were not provided with a tutorial for that interface. After finishing all three tasks, participants in both conditions completed a 20-minute post-test, and then filled out an exit questionnaire.

Measuring the Dependent Variables

To measure time on task, we reviewed the screen recordings, noting the time at which each participant started and stopped the task. In a few cases, this measure was not readily obtainable. Roughly 5% of the recording files were corrupt, and technical support on this issue was not forthcoming. To retrieve this data, we calculated a ratio of file size to time on task based on those files which were intact. This ratio was then applied to the problem files, providing us with reasonable estimates of time on task.

In one case, a student was found to have worked well past the 35 minute deadline in one of the tasks. Using the recording, that participant’s work was capped at 35 minutes. The solution achieved at that point was used for analysis purposes.

In three cases, participants in the DM group failed to switch to the text version of ALVIS for task three, as instructed. Data from these participants were dropped from our analyses.

To measure programming accuracy, we identified the key semantic elements of a correct solution to each task. Because our three tasks were isomorphic, each task solution consisted of the following set of eight semantic components: (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. As can be seen, a semantic
element essentially constituted a line of code in most cases. We gave each algorithm solution a score of 0 to 8 based on the number of correct semantic elements it contained. Here is an example of scoring, using the correct implementation of Find Max.

create array a1 with 6 cells—**semantic (a)**

populate a1 with random ints between 1 and 100—**sem. (b)**

set maxSoFar to 0

set il to index 0 of a1—**semantic (c)**

while il < cells of a1—**semantic (d) and (e)**

  if a1[il] > maxSoFar—**semantic (f)**

    set maxSoFar to a1[il]—**semantic (g)**

  endif

  add 1 to il—**semantic (d) and (e)**

endwhile

Semantics (a) through (c) all deal with object creation and initialization. These were the easiest for the participants to achieve and for the researcher to grade. Potential grading difficulties begin with semantic (d), in which the index must visit each array cell. Algorithms can fail this test if the participant doesn’t use an index, or if they do not increment the index properly, or if the loop terminates prematurely.

Semantic (e) is intended to penalize endless loops. Semantics (f) and (g) are the meat of the algorithm. Each of the tasks in the experiment required a comparison (f) and, if true, either a set statement or an accumulation (g). Finally, semantic (h) is awarded if the algorithm provides the correct result. This last semantic might seem a strange addition, but the purpose of a semantic is to differentiate between different participant
algorithms based on content. It is possible for two algorithms to receive exactly the same score on semantics (a) through (g), and yet only one of the algorithms obtains the correct result. For example, we observed that two participant solutions used no loop, but instead used hard coded statements for each array cell. One of the algorithms overlooked one of the cells. Except for semantic (g), these algorithms could obtain the same score.

Results

To provide a high level view of our results, Figure 17 graphically plots the mean accuracy scores by condition across all three tasks, along with the mean accuracy scores across both conditions by task number. Likewise, Figure 18 plots the mean task times by condition and by task. As these figures indicate, the participants in the DM condition not only constructed more accurate programs, but did so in less time. Moreover, as would be expected from practice, the accuracy of both conditions improved from Task 1 to Task 2, while the time-on-task for both conditions decreased from Task 1 to Task 2. These trends are not evident from Task 2 to Task 3, most likely because participants in the DM condition switched to the Text interface for Task 3.

Figure 19 and Figure 20 examine the results more closely by plotting each condition’s accuracy and task time means on a task-by-task basis; Figure 21 and Figure 22 present boxplots of these same data. In the boxplots, the boxed regions delineate the middle 50% for each condition; the asterisks denote outliers; and the horizontal lines mark the medians. While the boxplots indicate a substantial amount of variance in the data—a hallmark of novice performance—these plots clearly show that the DM condition constructed more accurate programs in all three tasks, with the biggest
Figure 17: Plot of time-on-task means by condition and task. The horizontal line marks the overall mean across conditions and tasks.

Figure 18: Plot of accuracy score means by condition and task. The horizontal line marks the overall mean across conditions and tasks.
Figure 19: Plot of Accuracy

Figure 20: Plot of Time on Task
accuracy difference occurring in the initial task. Similarly, in all three tasks, the DM condition took less time than the Text condition, with the biggest difference occurring in the second task.

To test for statistically significant differences, we ran a repeated measures analysis of variance (ANOVA) model with condition (Text vs. DM) and task number (1,
2, 3) as the main effects. The interaction between condition and task number, along with the subject effect within the conditions, was accounted for by the statistical model.

Table 1 and Table 2 present the results of the ANOVA for accuracy and time-on-task. As predicted by our hypotheses, the effect of condition on both accuracy and time-on-task was significant, with the DM condition completing the tasks in significantly less time than the Text condition, but scoring significantly higher.

In addition, we can see from the tables that there were significant effects due to both task number and subject. The task number effect can be seen as a practice effect; participants improved because they gained practice from task to task. The strong subject effect was also to be expected, given the large amount of variance that is typically seen in novice performance.

We next wanted to see to what extent differences existed between the conditions in the three individual tasks. To that end, we ran paired sample t-tests with Wilks-Satterwhait correction (because we did not want to assume equal variances). The results, presented in Table 3, reinforce the general repeated-measure ANOVA results. Task 1 accuracy and time-on-task differences were significant at the 95% confidence interval, indicating that the DM interface was significantly easier to learn. This difference held up reasonably well in the second task, with the accuracy difference reaching significance at the 94% confidence interval, and the time-on-task difference remaining significant at the 95% confidence interval. Finally, for the third task, in which the DM participants switched to the text interface, our confidence in the significant difference diminished
### Table 1. ANOVA Results for Task Accuracy

<table>
<thead>
<tr>
<th>Source of variation</th>
<th>DF</th>
<th>F-value</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition (DM, Text)</td>
<td>1</td>
<td>5.69</td>
<td>0.023</td>
</tr>
<tr>
<td>Task number (1, 2, 3)</td>
<td>2</td>
<td>8.77</td>
<td>&lt; 0.0001</td>
</tr>
<tr>
<td>Condition × task number</td>
<td>2</td>
<td>1.48</td>
<td>0.236</td>
</tr>
<tr>
<td>Subject (condition)</td>
<td>32</td>
<td>5.93</td>
<td>&lt; 0.0001</td>
</tr>
</tbody>
</table>

### Table 2. ANOVA Results for Time-on-Task

<table>
<thead>
<tr>
<th>Source of variation</th>
<th>DF</th>
<th>F-value</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition (DM, Text)</td>
<td>1</td>
<td>6.30</td>
<td>0.017</td>
</tr>
<tr>
<td>Task number (1, 2, 3)</td>
<td>2</td>
<td>10.07</td>
<td>&lt; 0.0001</td>
</tr>
<tr>
<td>Condition × task number</td>
<td>2</td>
<td>0.86</td>
<td>0.428</td>
</tr>
<tr>
<td>Subject (condition)</td>
<td>32</td>
<td>3.16</td>
<td>&lt; 0.0001</td>
</tr>
</tbody>
</table>
only slightly, with the p-value of the accuracy difference rising to 0.089, and the p-value of the time-on-task difference rising to 0.068.

Results from the pre and post-test were not significant. In fact, the participants, on average, scored lower on the post-test than on the pre-test. I cannot, therefore, claim that ALVIS improved conceptual knowledge inside of the three hour period of the experiment. My speculation is that the time frame was too short for measurable conceptual gains to occur, and that the participants, after nearly three hours, were fatigued and impatient to finish with the experiment, and were therefore careless.

Discussion

The results would appear to provide solid empirical support for both of my hypotheses. Not only did the new DM interface significantly improve programming speed and accuracy, as compared to the text-only interface, but it also promoted a positive transfer-of-training effect, enabling the DM participants to outperform the Text
participants in the final task, both with respect to time and accuracy. I find the time and accuracy differences between the DM and Text conditions in Task 3 to be especially notable, given that participants in the DM condition did not receive training in the text-only interface prior to their performing Task 3 with that interface. The question, then, is why? What is it about the DM interface that would produce these results?

There are two different theoretical orientations which I think are helpful in explaining the predicted results. With respect to H1, which correctly predicted that the DM interface would support faster, more accurate programming, cognitive load theory (see, e.g., [27]) provides a plausible explanation. Cognitive load theory, as defined by Sweller [28], states that optimum learning occurs when the load on working memory is kept to a minimum to best facilitate changes in long term memory. The new DM interface constrains the complexity of programming, thus reducing the intrinsic cognitive load of the programming task. Note that the same conclusion might also be reached by the idea of directness [29], which would predict that our new DM interface reduces the “information processing distance” between a novice’s programming goals and the interface mechanisms provided to accomplish those goals.

With respect to H2, which correctly predicted a transfer-of-training effect, I think dual coding theory (see, e.g., [30]) provides a possible causal explanation. Dual-coding theory posits that (a) pictures and words are encoded in different ways in memory; (b) referential connections can be built between each encoding of a given concept, and (c) a concept that is dually coded and has referential connections can be remembered more easily. Because the new DM interface makes continuously visible the textual commands to which direct manipulation actions give rise, we speculate that users of the new DM
interface were able to build referential connections between pictorial representations (as manifested in the Animation Window) and textual representations (as manifested in the Script Window) of their programming plans. According to dual coding theory, this ought to lead to improved recall of the commands, and hence the positive transfer-of-training effect we observed.

It would seem, then, that my between subjects experiment was a success. I have conducted a wide range of statistical analysis on the resulting data, and the results are mainly significant. I believe that, based on these results, I can claim with a reasonably high confidence level that my hypotheses are correct.
CHAPTER FIVE

CONCLUSION

As I have argued, a substantial amount of research has focused on developing novice programming environments that lower the barriers to programming by supporting alternative programming techniques such as direct manipulation and demonstration. Such techniques have been shown to hold promise in making programming easier to learn; however, little empirical research has explored whether such techniques actually promote a positive transfer-of-training effect to textual programming—an effect that would be especially useful for computer science students, who will ultimately have to program in text-based environments. As a preliminary step toward addressing this issue, I have presented a new direct manipulation interface for the ALVIS software, along with an experimental study that furnishes evidence that a direct manipulation programming interface has the potential not only to lower the initial barriers to programming, but also to promote a positive transfer-of-training to textual programming.

For future research, I suggest consideration of two complementary directions. First, despite the empirical evidence that the new DM interface was an improvement over a text-only interface, I believe that there are many aspects of it that can be improved. For example, when defining a loop in our DM interface, a user can freely drag an array index anywhere in the Animation Window, even though the index must ultimately land in an array cell in order to be valid. In this situation, the fact that the user’s gesture is unconstrained can lead not only to temporary confusion on the user’s part over what to do next, but also to gestures with ambiguous semantics. I believe that imposing additional
gestural constraints throughout the DM interface will greatly improve the learnability and usability of the interface. Indeed, as Mudugno et al. [26] learned in their development of a demonstrational interface with much in common with ALIVS, “seemingly small details of the system can greatly alter the system’s effectiveness” (p. 278).

Additional improvements to the ALVIS interface can be made as well. We now require that programmers use the Move tool to move objects in the Animation Window. However, users consistently try to move objects without first activating this tool. It should be possible to accommodate the user’s desire to move objects without resorting to a tool at all. The Swap tool, which is an older tool, should borrow from the new tools’ activation and construction processes. This would allow swap commands that can operate on referenced array cells. Similar enhancements can be made to all of the old tools, borrowing from the new tools’ functionality.

Second, I suggest that future research could strengthen and expand upon the empirical case that direct manipulation programming interfaces can promote positive transfer-of-training to textual programming. To that end, post-hoc video analysis of participants’ programming sessions could be performed, in order to gain further qualitative insights into the ways in which they proceeded with each interface. This investigation of the visual artifacts should provide researchers with valuable information about when and why students make programming errors. Further, this sort of analysis can uncover the precise moment of insight when a student discovers how to correctly proceed, and could indicate what element of the interface led to this insight. I propose that the video recordings can be essentially transcribed, such that most relevant information about when a student begins and ends a semantic component, makes an error,
or corrects an error, can be recorded in a spreadsheet. This would provide researchers with the temporal evolution of the code being studied. I must admit, however, that interpretation will have to play a part, especially with regards to the cause of both errors and insights.

Finally, I plan on being involved in a follow-up study in the Summer of 2006 that considers a version of ALVIS that supports more difficult programming tasks involving procedures and recursion, in order to better understand the extent to which direct manipulation programming interfaces can be leveraged for more advanced topics in introductory computer science courses.
BIBLIOGRAPHY


