Can Direct Manipulation Lower the Barriers to Programming and Promote Positive Transfer to Textual Programming? An Experimental Study

Christopher D. Hundhausen, Sean Farley, and Jonathan Lee Brown
Visualization and End User Programming Laboratory
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164-2752 USA
{hundhaus, sfarley, jbrown}@eecs.wsu.edu

Abstract

Novices face many barriers when learning to program, including the need to learn both a new syntax and a model of computation. By constraining syntax and providing concrete visual representations on which to operate, direct manipulation programming environments can potentially lower these barriers. However, what if the learning goal of the novice is to be able ultimately to program in conventional textual languages, as is the case for introductory computer science students? 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, we designed a new direct manipulation programming interface for ALVIS Live!, a novice programming environment. We then conducted an experimental study that compared the programming outcomes promoted by the new direct manipulation interface to those promoted by ALVIS Live!'s textual programming interface. We found that the direct manipulation interface not only led to significantly better initial programming outcomes, but also to significant positive transfer to the textual interface. Our results show that direct manipulation interfaces can provide novices with a "way in" to traditional textual programming.

1. Introduction

Learning how to program is challenging for many students, as evidenced by the high attrition rates commonly seen in introductory computer science courses [1]. In our own introductory computer science course, for example, we have observed 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?

Past computer science education research suggests a multitude of factors might contribute to the high attrition rates of introductory computer science courses, including (a) individual student differences, (b) a lack of a sense of community, (c) deficient pedagogical approaches, and (d) inadequate novice programming environments.

An extensive line of research, comprehensively reviewed in [2], has addressed the issue of inadequate novice programming environments. In an attempt to lower the barriers to programming, this line of research has explored a variety of alternative novice programming environment features, including algorithm visualization (e.g., [3]), drag-and-drop editing (e.g., [4]), programming by direct manipulation and gestures (e.g., [5]), and support for the declarative constructs that novices appear to use independently of programming environments [6].

In conjunction with our development of an alternative "studio-based" pedagogical approach to teaching novices how to program [7], we have been contributing to the above line of research through our development of a novice programming environment called ALVIS Live! [8], 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—that is, by using a set of tools to directly lay out program objects in an animation window.

While a recent experimental study of the ALVIS Live! environment [9] showed that its "live" editing and visualization model enables novices to program significantly more accurately than they could without any programming environment at all, we were dismayed by the extent to which 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 would be suggested by past research [10], we found that iterative constructs proved to be a key stumbling block for participants in our study.

Given that study participants appeared to benefit from ALVIS Live!'s direct manipulation interface for creating program objects, we wondered whether a direct manipulation interface might also address the programming difficulties we observed in our study. This led to the following research question:

RQ1: *Can a direct manipulation interface for programming iteration, conditionals, and assignment statements in ALVIS Live! 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 paper, we address both of these questions by presenting and experimentally evaluating a new direct manipulation programming interface for the ALVIS Live! 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 Live!'s animation window. Coupled with ALVIS Live!'s existing direct manipulation interface for object creation, the new interface makes it possible for ALVIS Live! users to specify, without having to type in any textual code, the kinds of single procedure, array iterative algorithms that students explore in the first five weeks of our algorithms-first introductory computer science course [7].

Given that the ultimate purpose of ALVIS Live! 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. Our concern for the issue of transfer 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, our new direct manipulation interface not only facilitated significantly faster and more accurate programming than the text-based ALVIS Live! interface, but also promoted positive transfer to textual programming.

The remainder of this paper is organized as follows. After reviewing related work in Section 2, we describe our new direct manipulation ALVIS Live! interface in Section 3. Then, in Section 4, we present the design and results of our experimental evaluation. Finally, in Section 5 we summarize our contributions, and outline directions for future research.

2. Related Work

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 in code. A large body of work, some of which is surveyed in [11], shares our 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 [12] 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 [13] 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 [14], children can program robots by dragging iconic representations of commands from a palette onto a workspace, where they can be wired together to create a program. In Stagecast Creator [5], children specify simulations by demonstrating graphical rewrite rules in a grid-based world. In Tinker [15], 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.

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 Live!, many of these environments generate visual representations of program execution (e.g., [3, 16]). In addition, some of these environments enable the learner to specify a program at least partly by direct manipulation. For example, ALICE [16] and JPie [4] provide drag-and-drop code editors that prevent syntax errors. In RAPTOR [17], 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.

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 [12] 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*. [18] 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. The experimental comparison presented here differs from these two evaluations in that it focused not only on programming performance, but also on the extent to which programming in a direct manipulation interface facilitates positive transfer-of-training to the kind of textual programming interface that computer science students must ultimately use.

3. New Direct Manipulation Interface

We developed the new direct manipulation interface to ALVIS Live! (henceforth "the new DM interface") through a user-centered design process. Using the original ALVIS Live! as a starting point, we first constructed a preliminary low fidelity prototype of the new DM interface. The prototype consisted of a series of static screens that were created by doctoring screenshots of the original ALVIS Live!. To test and refine the interface, we ran a "wizard of oz" prototype study, for which we recruited 15 volunteers out of the Fall, 2005 offering of the introductory computer science course at Washington State University. The design gradually evolved into its final form through five design iterations, each of which consisted of input from three to five participants.

Figure 1 presents an annotated screenshot of the new DM interface, which is identical to the original ALVIS Live! interface [8], except that it contains four new programming tools—"If," "Iterate Loop," "Set," and "Math" (see bottom of Toolbox in Figure 1). Just as in the original ALVIS Live!, users of the new DM interface program algorithms in a pseudocode-like language called "SALSA." They can do so either by typing SALSA commands directly into the Script Window, or by using the Toolbox Tools to specify commands by placing and directly manipulating objects in the Animation Window. Through such direct manipulation, SALSA code is dynamically inserted into the Script Window on the left.

As in the original ALVIS Live, the focal point of the new DM interface is the green Execution Arrow, which marks the line of code that was most recently executed, and that is currently being edited. On every keystroke or direct manipulation action, that line of code is re-executed, and the graphics in the Animation Window are dynamically updated to reflect that execution.

3.1 Sample Programming Session

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.

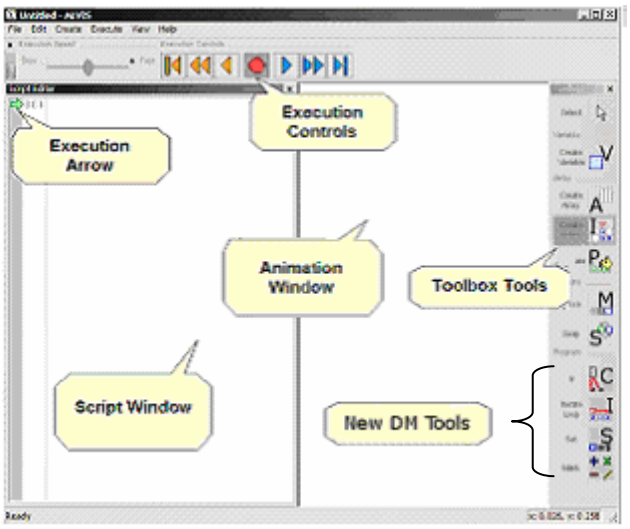
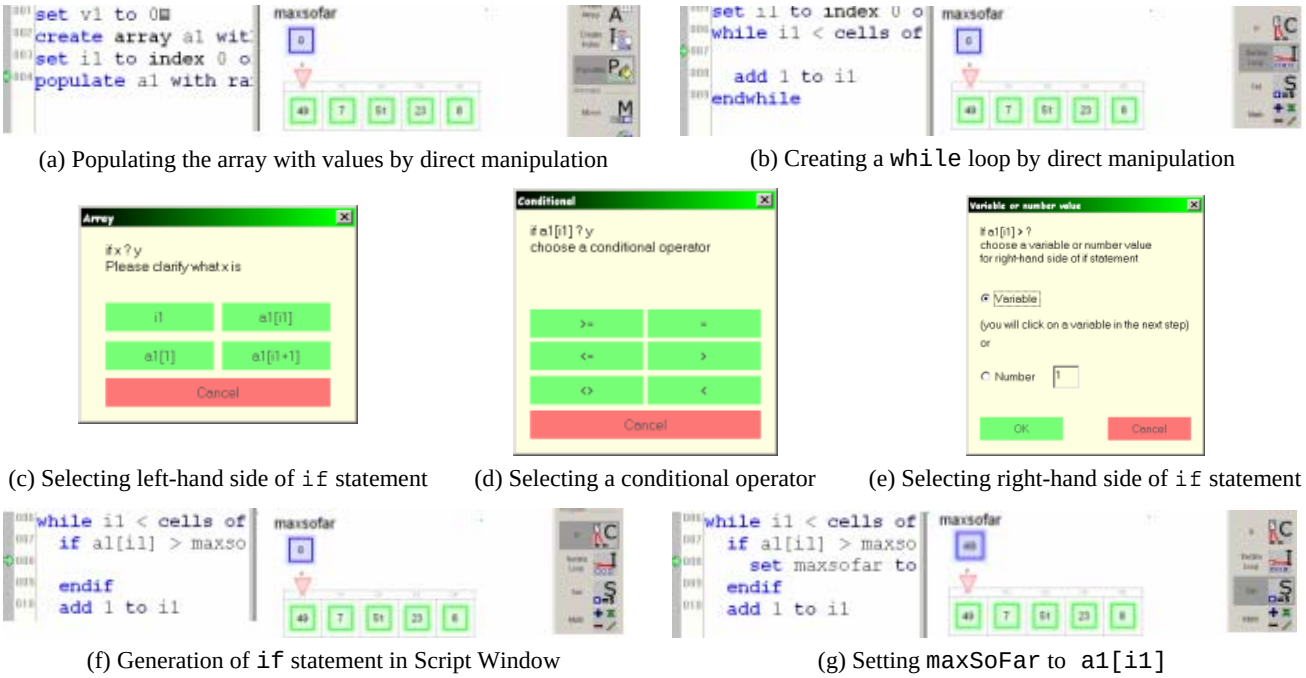


Figure 1. Annotated Snapshot of the new direct manipulation interface to ALVIS Live!. The only difference between this new interface and the original ALVIS Live! interface is the addition of four new Toolbox Tools: If, Iterate Loop, Set, and Math.

To code this algorithm, we must first create the algorithm objects. We need an array named `a1`, a variable named `maxSoFar`, and an array index named `i1`. Each of these can be created using the DM tools. For example, we can create the variable by clicking on the Create Variable tool to activate it, and then clicking in the Animation Window. A variable will appear at the cursor's location. We can create the array in a similar fashion, using the Create Array tool. To create the array index, the same process is followed, but here we click on the array cell where we want the index to be positioned (the left-most cell). We can modify object attributes, such as name or array size, by right clicking on the object, which provides us with a properties window. As each object is created, appropriate SALSA code appears in the Script Window.

Next, we need to populate the array with numbers. To do this, we first click on the Populate tool, and then click on the array, and we see the array fill with integer values (Figure 2a).

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 need to construct a loop to iterate through the array. To do this, we select the Iterate Loop tool, position the cursor on the `i1` index, press the mouse down, and drag the i1 index to the last array cell. When we release the button, a `while` loop skeleton appears in the Script Window consisting of a `while` statement, an index increment statement, and an `endwhile` statement (Figure 2b). A blank line is also created, and the caret is placed here to indicate that the next line of code will be placed here (unless we



(a) Populating the array with values by direct manipulation

(b) Creating a `while` loop by direct manipulation

(c) Selecting left-hand side of `if` statement

(d) Selecting a conditional operator

(e) Selecting right-hand side of `if` statement

(f) Generation of `if` statement in Script Window

(g) Setting `maxSoFar` to `a1[i1]`

Figure 2. Snapshots from a Sample Session with the New ALVIS Live! Direct Manipulation Interface

move the caret by clicking elsewhere in the Script Window).

In order to compare the array values to `maxSoFar`, we now need an `if` statement. We first click on the If tool, and then click on the first cell of the array. An array component selection window appears with several possible interpretations of this gesture (Figure 2c); we recognize `a1[i1]` as the correct left-hand side of our `if` statement, and click on it.

Next, we are prompted to select a conditional from the if conditional menu (Figure 2d), and we select ">." The final window asks us to decide if the right-hand side of the `if` statement should be a variable or a number (Figure 2e). Choosing variable, we use the mouse to click on `maxSoFar`; our completed `if` statement appears in the text window (Figure 2f). As with the `while` loop, `if` and `endif` lines appear, with a blank line in between, on which the editing caret is now positioned.

Finally, we use the Set tool to generate a statement that assigns the current array value to `maxSoFar` if the conditional test is true. We first select the Set tool, and then click on `maxSoFar`—the variable to be set. We are asked if we want to set `maxSoFar` to a variable or a number. We select "variable," and use the mouse to click on an array cell. 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 2g). As with the original ALVIS Live!, we can now explore the algorithm further by using the Execution Controls to step through our code. 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.

4. Experimental Evaluation

To evaluate the new DM interface, we conducted an experimental study with two main hypotheses:

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

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

To test these hypotheses, we 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 Live! for all three experimental tasks. In this software version, the only way to program was by entering textual SALSA code via the keyboard. In contrast, in the DM condition, for the first two experimental tasks, participants used a version of ALVIS Live! 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*—which we will explain further in Section 4.4. In addition, to ensure that participants in each condition were equally matched with respect to general programming knowledge, we administered a multiple-choice pretest. A follow-up posttest, isomorphic to the pretest, was used to gauge any gains in programming knowledge; however, its analysis is beyond the scope of this paper.

4.1 Participants

We 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. Participants received course credit for their participation.

4.2 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.

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.

In the DM condition, participants used the DM version of ALVIS Live! (see Section 3) for the first two tasks, and the text only version of ALVIS Live! (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 only version of ALVIS Live! for all three tasks.

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.

4.3 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, each of which included 10 to 12 participants. In each study session, participants first completed a 20-minute pre-test of conceptual programming 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.

4.4 Measuring the Dependent Variables

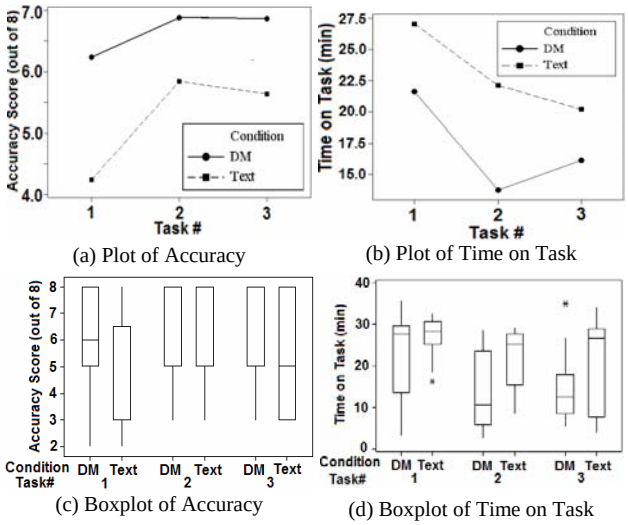
To measure time on task, we reviewed the screen recordings, noting the time at which each participant started and stopped the task. To measure programming accuracy, we identified the key semantic elements of a correct solution to each task. Because our three tasks were isomorphic, each task solution 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 mapped to a line of code or essential property of a correct solution. We gave each algorithm solution a score of 0 to 8 based on the number of correct semantic elements it contained.

4.5. Results

Before analyzing our data, we first verified that the two conditions were equally matched with respect to prior programming ability. To do this, we used a paired sample t-test with Wilks-Satterwhait correction (because we did not want to assume equal variances) to test for a significant difference between the two conditions' programming pretest scores. The test showed that the Text and DM conditions did not vary significantly (df = 27, T = 0.61, p = 0.727). In addition, using normal probability plots, we confirmed that our accuracy and time-on-task data were normally distributed.

Figure 3(a) and (b) plot each condition's accuracy and task time means on a task-by-task basis; Figure 3(c) and (d) present box plots of these same data. While the boxplots indicate a substantial amount of variance in the data—a hallmark of novice performance—these plots suggest that the DM condition constructed more accurate programs in all three tasks, with the biggest 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 get an overall indication of whether a statistically significant difference existed between the conditions with respect to accuracy and time-on-task across all three tasks, we first 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 presents the results of the ANOVA for accuracy and time-on-task. As predicted by H1, the overall effect of condition on both accuracy and time-on-task was significant, with the DM condition completing the three tasks in significantly less time than the Text condition (least square means:



(a) Plot of Accuracy (b) Plot of Time on Task

(c) Boxplot of Accuracy (d) Boxplot of Time on Task

Figure 3. Mean Accuracy and Time on Task Plots by Task. In the boxplots, the boxed regions delineate the middle 50% for each condition; the asterisks denote outliers; and the horizontal lines mark the medians.

Source of variation	DF	F-value	P-value
Accuracy			
Condition (DM, Text)	1	5.69	0.023
Task # (1, 2, 3)	2	8.77	< 0.0001
Condition × task #	2	1.48	0.236
Subject (condition)	32	5.93	< 0.0001
Time-On-Task			
Condition (DM, Text)	1	6.30	0.017
Task # (1, 2, 3)	2	10.07	< 0.0001
Condition × Task #	2	0.86	0.428
Subject (condition)	32	3.16	< 0.0001

Table 1. ANOVA Results for Accuracy and Time-On Task

DM—17.16 min; Text—23.07 min), but scoring significantly higher (least square means: DM—6.66 out of 8; Text—5.24 out of 8).

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 2, reinforce the general repeated-measure ANOVA results. Task 1 accuracy and time-on-task differences were significant at the 95% confidence inter-

Task #	Dependent Measure	DF	T-value	P-value
1	Accuracy	30	2.85	0.004
1	Time-On-Task	23	2.01	0.028
2	Accuracy	31	1.65	0.055
2	Time-On-Task	30	2.89	0.004
3	Accuracy	28	1.38	0.089
3	Time-On-Task	27	1.54	0.068

Table 2. DM vs. Text T-Test Results by Task

val, indicating that the DM interface was significantly easier to learn. This difference held up reasonably wellin 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 results suggest marginal significance, 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.

4.6 Discussion

Our results would appear to provide reasonable empirical support for both of our hypotheses. Not only did the new DM interface significantly improve programming speed and accuracy, as compared to the text-only interface, but it may also have 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. Even though they were found to be only marginally significant, we 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?

We find two different theoretical orientations helpful in explaining our predicted results. With respect to H1, which correctly predicted that the DM interface would support faster, more accurate programming, we believe cognitive load theory (see, e.g., [19]) provides a plausible explanation: namely, that 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* [20], 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 predicted a transfer-of-training effect, we think dual coding theory (see, e.g., [21]) 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.

5. Summary and Future Work

As we 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, we have presented a new direct manipulation interface to our ALVIS Live! 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.

In future research, we plan to pursue two complementary directions. First, despite the empirical evidence that our new DM interface was an improvement over a text-only interface, we 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. We believe that imposing additional gestural constraints throughout our DM interface will greatly improve the learnability and usability of the interface. Indeed, as Mudugno *et al.* [18] learned in their development of a demonstrational interface with much in common with

ALIVS Live!, “seemingly small details of the system can greatly alter the system’s effectiveness” (p. 278).

Second, we would like to strengthen and expand upon the empirical case that direct manipulation programming interfaces can promote positive transfer-of-training to textual programming. To that end, we will perform a post-hoc video analysis of participants’ programming sessions, in order to gain further qualitative insights into the ways in which they proceeded with each interface. In addition, we would like to run a follow-up study 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 in introductory computer science courses.

6. Acknowledgments

Dr. Nairanjana Dasgupta of the Department of Statistics at Washington State University designed and carried out the statistical analyses presented in Section 4.5. This research is funded by the National Science Foundation under grant nos. 0406485 and 0530708.

7. References

[1] T. Beaugouef and J. Mason, "Why the high attrition rate for computer science students: Some thoughts and observations," *SIGCSE Bulletin 37* (2), 2005, pp. 103-106.

[2] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys 37* (2), 2005, pp. 83-137.

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

[4] B. Birnbaum and K. Goldman, "Achieving Flexibility in Direct-Manipulation Programming Environments by Relaxing the Edit-Time Grammar," in *Proceedings 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Press, Los Alamitos, 2005, pp. 251-258.

[5] D. C. Smith, A. Cypher, and J. Spohrer, "KidSim: Programming agents without a programming language," *Communications of the ACM 37* (7), 1994, pp. 54-67.

[6] J. F. Pane, B. A. Myers, and L. B. Miller, "Using HCI techniques to design a more usable programming system," in *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. IEEE Computer Society, Los Alamitos, 2002, pp. 198-206.

[7] C. D. Hundhausen and J. L. Brown, "Personalizing and discussing algorithms within CS 1 Studio Experiences: An observational study," in *Proc. 2005 International Computing Education Research Workshop*. ACM Press, New York, 2005, pp. 45-56.

[8] C. D. Hundhausen and J. L. Brown, "What You See Is What You Code: A radically dynamic algorithm visualization development model for novice learners," in *Proceedings 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society, Los Alamitos, 2005, pp. 140-147.

[9] C. D. Hundhausen and J. L. Brown, "An experimental study of the impact of feedback self-selection on novice programming," *Journal of Visual Languages and Computing*, Under review.

[10] E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive strategies and looping consructs: an empirical study," *Communications of the ACM 26* (11), 1983.

[11] A. Cypher, "Watch What I Do: Programming by Demonstration." Cambridge, MA: The MIT Press, 1993.

[12] M. Burnett and H. Gottfried, "Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures," *ACM Transactions on Computer-Human Interaction 5* (1), 1998, pp. 1-33.

[13] J. T. Stasko, "Using Direct Manipulation to Build Algorithm Animations by Demonstration," in *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems, Programming by Demonstration*. ACM Press, New York, 1991, pp. 307-314.

[14] Lego Systems, Inc., "Lego Mindstorms Robotics Invention System," http://mindstorms.lego.com, 1998.

[15] H. Lieberman, "Tinker: A programming by demonstration system for beginners," in *Watch What I Do: Programming by Demonstration*, C. Cypher, Ed. MIT Press, Cambridge, MA, 1993.

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

[17] M. Carlisle, T. Wilson, J. Humphrieis, and S. Hadfield, "RAPTOR: A visual programming environment for teaching algorithmic problem solving," in *Proc. ACM SIGCSE 2005 Symposium*. ACM Press, New York, 2005, pp. 176-180.

[18] F. Mudgno, A. Corbett, and B. Myers, "Graphical representation of programs in a demonstrational visual shell—an empirical evaluation," *ACM Transactions on Computer-Human Interaction 4* (3), 1997, pp. 276-308.

[19] J. J. G. van Merrienboer and J. Sweller, "Cognitive load theory and complex learning: Recent developments and future directions," *Educational Psychology Review 17* (2), 2005, pp. 147-177.

[20] E. L. Hutchins, J. D. Hollan, and D. A. Norman, "Direct manipulation interfaces," *Human-Computer Interaction 1* (4), 1985, pp. 311-338.

[21] A. Paivio, "The empirical case for dual coding," in *Imagery, Memory, and Cognition: Essays in Honor of Allan Paivio*, J. C. Yuille, Ed. Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.