Dual-Modality Instruction and Learning

A Case Study in CS1

Jeremiah Blanchard
Department of CISE
University of Florida
Gainesville, FL, USA
jjb@eng.ufl.edu

Christina Gardner-McCune
Department of CISE
University of Florida
Gainesville, FL, USA
gmccune@ufl.edu

Lisa Anthony
Department of CISE
University of Florida
Gainesville, FL, USA
lanthony@cise.ufl.edu

ABSTRACT

In college-level introductory computer science courses, students traditionally learn to program using text-based languages which are common in industry and research. This approach means that learners must concurrently master both syntax and semantics. Blocks-based programming environments have become commonplace in introductory computing courses in K-12 schools and some colleges in part to simplify syntax challenges. However, there is evidence that students may face difficulty moving to text-based programming environments when starting with blocks-based environments. Bi-directional dual-modality programming environments provide multiple representations of programming language constructs (in both blocks and text) and allow students to transition between them freely. Prior work has shown that some students who use dual-modality environments to transition from blocks to text have more positive views of text programming compared to students who move directly from blocks to text languages, but it is not yet known if there is any impact on learning. To investigate the impact on learning, we conducted a study at a large public university across two semesters in a CS1 course (N=673). We found that students performed better on typical course exams when they were taught using dual-modality representations in lecture and were provided dual-modality tools. The results of our work support the conclusion that dual-modality instruction can help students learn computational concepts in early college computer science coursework.

CCS CONCEPTS

• **Social and professional topics~CS1** • *Human-centered computing~Visualization* • Software and its engineering~Integrated and visual development environments

KEYWORDS: Computer science education, CS1, blocks-based programming environments, programming languages, novice programmers, dual-modality programming environments.

SIGCSE '20, March 11–14, 2020, Portland, OR, USA.

ACM ISBN 978-1-4503-6793-6/20/03...$15.00.
DOI: https://doi.org/10.1145/3328778.3366865

ACM Reference Format: Jeremiah Blanchard, Christina Gardner-McCune, & Lisa Anthony. 2019. Dual-Modality Instruction and Learning: A Case Study in CS1. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (SIGCSE '20), March 11-14, Portland, OR, USA. ACM, NY, NY. 7 pages. https://doi.org/10.1145/3328778.3366865

1 Introduction

When learning computer science, students must develop competency in computational thinking [37], an understanding of the fundamentals of programming and computer science theory, and also the ability to engage in abstraction, all while working in a language's semantic structure and syntax [32]. Drawing on the Neo-Piagetian framework of how students learn to program, students must develop expertise in programming by moving from the *sensorimotor stage* (in which they know what a program does, but not how it does it) to later *concrete* and *formal operational stages* (in which they can understand programs by breaking them into chunks and considering their abstract function) [20]. Then, students must be able to translate their ideas into code that runs within the target environment.

Computer science instructors and education researchers have recognized the positive role that appropriate scaffolding can play in programming instruction to help address these challenges; this has motivated the development of instructional programming environments [18]. In particular, blocks-based environments, such as Scratch and Alice, were developed by researchers in computer science education to decouple the learning of syntax from programming, computational thinking, and computer science theory [10, 16, 23]. Eliminating syntax errors may help reduce students' cognitive load, allowing them to master computational thinking skills without needing to master syntax at the same time [21]. While blocks-based environments have shown promise in improving learning by scaffolding programming by abstracting away syntax [11, 24], they do not currently address syntax challenges students must ultimately face when transitioning to production language environments. Dual-modality environments provide multiple representations of programming language constructs (in both blocks and text) and allow students to transition between them freely [1, 3]. These environments may be able to help bridge concepts learned in blocks to text – helping students delve into syntax by providing blocks and text representations of the same program, not only scaffolding learning of syntax, but also associating blocks and text representations, thereby scaffolding learner chunking and abstraction.

In this paper we present a study focused on answering the question, "**How do dual-modality programming environments and dual-modality instruction support learning of programming knowledge, as compared to traditional (text-based) approaches to instruction in CS1 courses?**" We investigated the use of dual-modality instruction and student learning in a study at a large public university across two 16-week semesters in an introductory computer science (CS1) course (n=673). Both semesters were taught by the same instructor. The first semester (n=248), acting as a control group, was taught using traditional, text-based instruction. The second semester (n=425), acting as an intervention group, was taught using dual-modality instruction and a dual-modality IDE plugin we developed. We measured participant learning via the SCS1 [27], a computer science inventory assessment, and course examination questions, which were classified as either definitional / code reading or code writing [30]. We also conducted surveys, both weekly and at the beginning, midpoint, and end of instruction, about student perceptions of blocks, text, and dual-modality instruction. Our hypothesis was that dual-modality instruction and tools would help students better chunk and abstract sections of code, and that, as a result, students in the intervention group would score higher on exam questions and the SCS1 than those in the control group.

2 Background

In college-level introductory computer science courses, students traditionally learn to program using text-based languages which are common in industry and research [31]. Even for languages with simple structure, syntax errors and semantics present challenges for learners [35]. Blocks-based programming environments have become more common in introductory computing courses in K-12 schools in part to address these challenges [14, 36]. However, there is evidence that students may face difficulty moving to text-based programming environments from blocks-based environments [22]. Some CS education community members believe that bi-directional dual-modality environments (**Figure 1**), which provide multiple representations of programming language constructs (in both blocks and text), show promise because they may provide a bridge for students between blocks and text representations while also scaffolding chunking and abstraction [9].

2.1 Bi-Directional Dual-Modality Environments

Modern bidirectional dual-modality environments, such as the Droplet Editor used by Pencil Code and App Lab on Code.org, allow students to move between blocks (**Figure 1a**) and text (**Figure 1b**) program representations freely and on demand [1–3]. Researchers have suggested that dual-modality instruction may facilitate learning in computing by providing a blocks-to-text scaffold for programming. Prior work suggests that dual-modality environments may alleviate frustration and improve student perceptions of programming [6, 33]. In addition, Weintrop found that dual-modality environments provide some of the affordances of both blocks and text when such environments are used by high school students: they helped foster student confidence (like blocks) but were also perceived as authentic programming experiences (like text languages) [33]. Later, Weintrop and Holbert found that, even after moving to work in text, learners in dual-modality environments returned to blocks when new constructs were introduced, suggesting the blocks helped scaffold learning when working with those constructs [34]. Dual-modality environments may also support student confidence in text languages. Bau et al. conducted a study with eight public middle school students across four after-school sessions who had no prior programming experience [4]. Students were permitted to freely use blocks and text modes. On the first day, all students used blocks most of the time; they progressed to using text more often each day. By the last day, they were working in text 95% of the time, of their own volition. Later, our own prior work with middle school students showed that some students who used dual-modality environments to transition from blocks to text had more positive views of text programming compared to students who moved directly from blocks to text [6]. However, despite these benefits to perception and confidence, it is not yet known if dual-modality environments and instruction help students learn programming concepts more effectively than traditional text-based approaches.

One challenge to using dual-modality representations in instruction is that they have been tied to specific use cases and platforms. For example, Pencil Code and Code.org are website-based sandboxes that allow students to program in the browser without any additional tools, with Pencil Code supporting three different languages [5, 17]. However, students in introductory programming classes often use an Integrated Development Environment (IDE) as is common in industry, thus bringing additional authenticity to the learning experience. We aimed to evaluate learning and use of dual-modality representations in realistic learning contexts such as these.

2.2 Development of Expertise

The Piagetian model of development envisioned stages of cognitive development tied to age [8]. Neo-Piagetian models do not tie cognitive development to particular ages; rather, they suggest that the stages of cognitive development are repeated for every new area of expertise or domain we learn [28]. Thus, people cognitively progress through these stages multiple times across their life spans. Lister proposed, and Teague expanded, the Neo-Piagetian framework to account for cognitive development in the

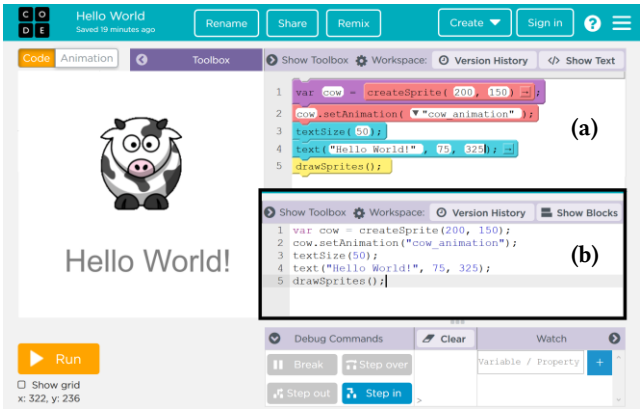

Figure 1: Droplet, Code.org; blocks (a) with text overlaid (b).

Table 1. Topics by Semester & Instructional Mode			
Topic	Ctrl	Intv	Intv. Mode
Introduction to CS	✓	✓	Blocks & Text
Variables / Arithmetic	✓	✓	Blocks & Text
Control (Selection, Loops)	✓	✓	Blocks & Text
Data Types / Arrays	✓	✓	Blocks & Text
Functions / Static Methods	✓	✓	Blocks & Text
Software Engineering	✓	✓	Blocks & Text
Classes & Objects	✓	✓	Blocks & Text
Algorithm Complexity	☒	✓	Blocks & Text
Inheritance / Interfaces	✓	✓	Text Only
Data Structures	✓	☒	-
Data Streams & Recursion	✓	✓	Text Only
Propositional Logic	☒	✓	Text Only
Generic Programming	✓	☒	-
Memory & Paradigms	✓	✓	Text Only

domain of programming [20, 30]. Lister and Teague proposed four stages of reasoning development in programming. In the **sensorimotor** stage, students know programs produce a result, but not why. At the **pre-operational** stage, students understand lines of code and can read simple programs but have difficulty thinking abstractly about code [20]. In the later **concrete operational** stage, students can reason about familiar, real-world situations; they can apply abstractions to help them read more complex code and write code. Finally, in the **formal operational** stage, they can reason about unfamiliar, hypothetical situations. This model of learning to program is supported by empirical evidence from Corney et al. [12] and think-aloud studies by Teague et al. [29].

The development of chunking and abstraction methods is also tied to the progression from stage to stage in the Neo-Piagetian model of development. Psychology literature has established that humans have limited short-term working memory [25], so experts employ chunking as a mechanism to recall information and ideas [13]. Information is broken into chunks that are stored in long-term memory; these chunks can be recalled as a single concept in working memory, reducing the number of unique ideas that must be in working memory at a particular moment in time, and thereby reducing cognitive load [13].

3 Methods

In this study, we collected data via several assessments, including class examinations and the SCS1 [27], which students took at the end of the course just before the final examination. The course covered all concepts tested by the assessments. We also collected student responses to several surveys throughout the intervention semester to help us understand the mechanisms behind any effects we might see. The same teacher, the first author, taught all of the CS1 students in this study. The course consisted of two large weekly lecture meetings and a weekly small lab meeting.

3.1 Study Design

As noted previously, our study was conducted over the span of two semesters in the CS1 course. Both semesters used the same lecture and lab format. The first semester, acting as a control group, was taught using traditional, text-based instruction; the second semester, acting as an intervention group, was taught using dual-modality instruction and a dual-modality IDE plugin we developed for the study:

1. Students in the **control group** (n=248) were provided with standard development tools, including **IntelliJ IDEA**, an Integrated Development Environment (IDE). All lecture slides and assignment descriptions used only text programming representations.
2. Students in the **intervention group** (n=425) were provided with **IntelliJ IDEA** and the **Droplet Dual-Modality IDE Plugin for Java** we built, which presented text and blocks representations of the code they wrote and allowed them to move freely between representation modes. They were also instructed in the plugin's use in lab sessions. 66.7% of the course (8 of 12 topics) used blocks and text representations on assignment descriptions and lecture slides (**Table 1**). The remaining topics were not represented in the blocks construct models we used (e.g., inheritance) or were non-programming topics (e.g., ethics and version control).

In general, the topic ordering between the semesters was the same, but some topics were replaced as part of typical course content adjustment in preparation for later courses. In the control semester, introductory Data Structure and Generics were covered, while in the intervention semester, Algorithm Complexity and Propositional Logic were covered (**Table 1**).

3.2 Dual-Modality Instruction & IDE Plugin

To present content via dual-modality instruction, we developed slides which incorporated blocks and text representations. These slides included animations transitioning from blocks to text representations. These materials were used for lectures introducing content, while the plugin we developed was used for projects. When we began this study, there were no existing dual-modality tools for standalone IDE-based development, so we developed a standalone plugin for IntelliJ IDEA based on the Droplet editor (**Figure 2**) [1, 38]. The plugin we developed enables instructors to more easily incorporate dual-modality instruction into courses [7]. In this version of the plugin, we targeted Java, which is common in introductory courses, including

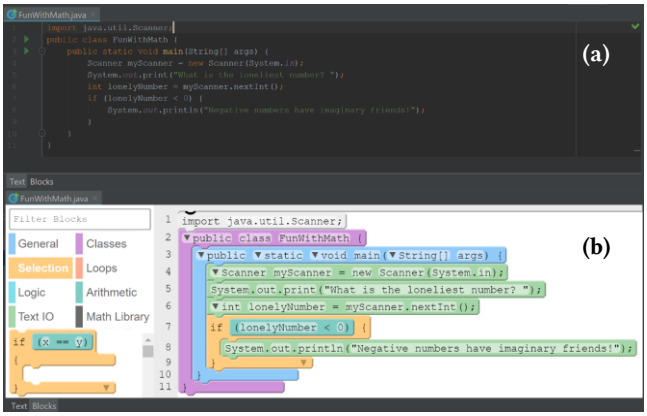

Figure 2: Droplet Java Plugin – (a) text and (b) blocks

Table 2. Demographic Groups by Condition		
Demographic Group	**Control**	**Intervention**
Men	66.7%, n=96	74.9%, n=188
Women	33.3%, n=48	25.1%, n=63
Asian	27.8%, n=40	30.7%, n=77
Black / African American	11.1%, n=16	5.2%, n=13
Hispanic / Latino	20.8%, n=30	24.7%, n=62
Native American	0.0%, n=0	0.4%, n=1
Native Hawaiian / Pacific Is.	0.7%, n=1	0.4%, n=1
White, Non-Hispanic / Latino	49.3%, n=71	52.6%, n=132
Other	0.7%, n=1	1.6%, n=4
Multiple Racial Backgrounds	7.6%, n=11	15.5%, n=39

in the CS1 course at the host university. This helped us avoid confounding factors, allowing us to use the standard development environment and programming language for the course in our study. Of the 425 students in the study, 88.9% (n=378) installed and used the plugin. The remaining 11.1% (n=47) of students did not install or use the plugin, but still received dual-modality instruction in lectures.

3.3 Participants

This study's data collection was classified as exempt by our Institutional Review Board (IRB). However, we still asked students who completed the SCS1 and demographic survey to explicitly consent to having their scores and demographic data included in the study. No compensation was provided to participants, but students who completed the SCS1 and demographic survey received extra credit in the course. Students who did not participate were offered an alternative assignment to earn the same amount of extra credit. Students who did not complete the SCS1 or demographic survey received the same instruction and in-class programming assignments and assessments, as these are part of ordinary classroom activity assigned by the instructor.

The CS1 course is the first required course in the programming sequence at the host university; as such, this class has a mix of students with prior experience and those with none. Transfer students with programming coursework usually have CS1 waived as an equivalency via transfer credit. 58.1% of the control group (n=144) and 59.1% (n=251) of the intervention group opted-in to the demographic survey. 36.8% (n=53) of students in the control group and 40.2% (n=101) in the intervention group had some prior programming experience; 19.9% (n=29) of the control and 22.1% (n=94) of the intervention groups had taken the AP Computer Science or AP Computer Science Principles courses in high school.

Participants came primarily from the young college student age range (18-22) and represented diverse ethnic, racial, and gender backgrounds in both the control and intervention groups due to the course's size and the university's demographics (**Table 2**). While the populations are similar, there are some notable differences. There was a higher proportion of men in the intervention group, and more intervention students indicated they were from white, Asian, or from multiple racial backgrounds.

3.4 Data Collection

We collected several types of data which varied by condition.

3.4.1 Assessments and Demographic Surveys. We collected exam question scores from each participant via the university's learning management system (LMS). These examinations were purely text-based in both semesters, using the same framework and modeled from exams in previous terms. The midterm exams were non-cumulative, and the final was cumulative. There were two midterm exams which were broken into two sections: a multiple choice / short answer section with **definitional** and **code reading** questions, and a free response pseudocode section requiring **code writing**. As is typical for this course, the final exam had only definitional and code reading questions; since code writing questions require significant time for students to complete and instructors to grade, it is logistically difficult to fit a cumulative final assessment into the allotted final examination blocks of two hours and also to have grading completed in time for grade submissions.

We proctored the SCS1 at the end of the semester for both groups and recorded participant responses. In addition, we collected demographic data from participants at the end of the same computer-based survey. Participants completed assessment and demographic questions in a dedicated room over a period of one-hour fifty-five minutes. The SCS1 was voluntary, so a subset of students opted to participate in this part of the study (n=395).

3.4.2 Perception Surveys and Usage Logs. We collected answers to Likert-scale perception survey questions weekly, as well as download logs for online resources such as lecture slides. The perception surveys were given at the beginning, midpoint, and end of the semester. A secure server was also used to collect logs of how students used the plugin itself during the semester.

3.4.3 Bias Control. Neither the first author, who was also the instructor, nor the other authors, had access to information about who took or planned to take the SCS1 during either semester. Instead, this information was controlled by the teaching assistants until after final grade submission. Once final grades had been submitted for the course, the teaching assistants shared the assessment and participation data with the instructor.

3.5 Data Analysis

To investigate student learning, we examined scoring on course exam questions and the SCS1. We broke down our analysis according to question type, which we also associate with the Neo-Piagetian stages of development. This paper focuses on the assessment data and what it reveals about student learning of programming between these two different conditions. Our future work will involve analysis of the survey responses and their correlation with assessments.

3.5.1 SCS1 Questions. Students in both semesters were offered the option to sit for the SCS1 exam at the end of the semester for extra credit. All students in both conditions took the same exam, whose questions are categorized by type into definitional, tracing, and code

Table 3. List of Topics in Common by Exam		
Midterm 1	Midterm 2	Final
Instructions	Classes	Instructions
Arithmetic	Encapsulation	Arithmetic
Selection	Overloading	Data Types
Data Types	Inheritance	Functions
Functions	Overriding	Arrays
Arrays		Loops
References		Versioning
		Data Streams

completion questions [27]. We computed overall scores on the SCS1 as well as scores by question type.

3.5.2 Definitional and Code Reading Questions. For the midterm and final exam definitional and code reading questions, not all question topics and formats appeared across semesters due to exam date variation. To eliminate these differences as a confounding factor, we identified a subset of questions for each midterm and final exam that were in common across semesters (**Table 3**). While the first midterm and final exam had nearly or exactly the same number of questions, the second midterm exams differed in length: the control group's exam was shorter. For the first midterm exam, 10 of 16 (62.5%) questions from the control semester overlapped with 10 of 15 (66.7%) questions from the intervention semester, while for the second midterm, 5 of 10 (50%) questions in the control overlapped with 5 of 16 (33.3%) in the intervention term. Finally, on the final exam, 11 of 16 (68.8%) questions overlapped between the exams.

3.5.3 Code Writing Questions. The code writing exam questions were isometric variants of one another; that is, they required employing the same skills and tested the same concepts. For example, on the first midterm in both classes, the code writing section required students to write and invoke simple methods and engage in console I/O, while on the second midterm, this section required writing and extending classes, overloading, and overriding. As such, we were able to directly compare the results. As with the definitional / code reading section, we calculated percentage scores for each exam before comparison.

3.5.4 Analysis Tests. Once we had collected the scores from all of the assessments for overlapping questions, we compared the scores in the control group to those in the intervention group. As the scores did not follow a normal distribution, we employed the non-parametric two-tailed Mann-Whitney U test [26] to compare the groups. Further, we calculated the eta-squared (η^2) value to identify the effect size and report it with our findings.

4 Findings

We compared student exam question performance on two midterm exams, the final exam, and the SCS1. In this section we outline our findings for each assessment.

4.1 SCS1

The results from the SCS1 assessment did not differ significantly between the groups in our study (**Table 4**). This was true on the exam as a whole, as well as scores we computed by question type (definitional, tracing, and code completion). This may be related to the attributes of the specific SCS1 questions, which we discuss in Section 5.

Table 4. Results Summary (Scores as Percent)					
	Ctrl μ, σ	Intv. μ, σ	P-val.	Z	η^2
E1 - Reading	58.3, 17.3	85.4, 13.9	<.001	-16.4	0.41
E1 - Writing	68.9, 25.0	76.1, 24.4	<.001	-4.3	0.03
E2 - Reading	72.7, 19.8	76.4, 18.6	<.001	-2.8	0.01
E2 - Writing	68.0, 29.5	68.7, 25.8	0.826	-0.2	0.00
Final Exam	65.8, 18.3	72.0, 15.5	<.001	-4.1	0.03
SCS1 - All	51.6, 18.9	50.1, 18.0	0.46	-0.7	0.00
SCS1 - Def.	58.8, 19.4	57.5, 20.1	0.64	-0.5	0.00
SCS1 - Trac.	52.1, 21.2	49.9, 21.4	0.25	-1.2	0.00
SCS1 - Comp.	43.8, 26.1	43.0, 23.3	0.79	-0.3	0.00

4.2 Midterm Exams

Code Reading / Definitional. For the questions on topics shared between the exams that focused on definitions and code reading, students in the intervention group scored higher in both midterm exams, and for both exams this difference was statistically significant (α=0.05): first midterm (Z=-16.4, p<.001, η^2=0.41), and second midterm (Z=-2.8, p<.001, η^2=0.01).

Code Writing. For the code writing sections of the exams, the first exam showed significant differences between the conditions, but the second exam did not. For the first exam, students in the intervention group scored significantly higher (α=0.05) than those in the control group (Z=-4.3, p<.001, η^2=0.03), while for the second exam, means were not statistically different (Z=-0.2, p=.826, η^2=0.00).

4.3 Final Exam

Code Reading / Definitional. The final exam had only code reading and definitional questions for logistical reasons as explained in Section 3.4. On this exam, the intervention group scored higher, and again the result was significant (α=0.05) when comparing the scores from the control and intervention groups (Z=-4.1, p<.001, η^2=0.03).

5 Discussion

Code reading and writing are tied to the Neo-Piagetian model's concrete-operational and formal operational stages, especially for more complex tracing. We had hypothesized that, as dual-modality instruction supports and scaffolds student learning of abstraction and chunking – which are critical to these later Neo-Piagetian stages – that students would learn more effectively, and as a result score higher on assessments.

In our study, we found that the SCS1 did not help us distinguish between our control and intervention group, despite our large sample size. This was true even for the subset of questions identified as "code-completion" questions by the assessment's authors [27] and stands in contrast to the course exams' code writing questions. This may be due in part to the fact that, while the code completion questions on the SCS1 are multiple choice and scored as either "right" or "wrong", the course exam

questions were free response pseudocode questions. These free response questions can be graded with a rubric to award partial credit, allowing the scores to capture more nuance regarding student understanding. The results may also be due in part to the SCS1's difficulty. The authors of the SCS1 determined in their work that the SCS1 questions overwhelmingly skewed to *hard* levels of difficulty, and most questions they classified as *fair*, but not *good*, in their effectiveness at discriminating between students of different ability levels [27]. Also, the course exam questions were developed to address the specific topics covered in the course, including some topics that were not covered by the SCS1 (e.g., object-oriented programming), though there were no SCS1 topics *not* covered in the course.

When we compared scores from questions covering shared topics across semesters on in-class assessments, scores between the groups differed significantly on all assessments except for the code writing section of the second exam. It is notable that the subject of the second exam's code writing section was inheritance. This topic was not covered by the dual-modality instruction, and instead was taught in text, as the Droplet Editor does not have visualizations for inheritance relationships. As such, we would expect students to score about the same on such a topic, and indeed, this is what we found. The second exam's code reading and definitional section also contained topics that were not covered in blocks / text dual modes as they were not accounted for in Droplet's model. As such, we would expect to see less of a difference between the groups in the second midterm and final exams. In line with these expectations, the average scores in the second midterm's code reading section and final exam (which contained only code reading) were closer between the control and intervention groups, though the difference was still statistically significant, while the averages on the second midterm's code writing section were very close, without statistically significant difference. Further, the second midterm exam's definitional / code reading section had fewer questions in the control semester, but the exam was given in the same amount of time, giving students more time per question. Despite this, students in the intervention semester scored higher than students in the control semester.

6 Limitations

We designed this study to fit within existing coursework and several practical limitations arose as a result. As there is just a single CS1 course at the host institution, the control and intervention conditions were in different semesters, so the population was not randomly assigned. However, we did collect and examine demographic information to minimize confounding factors. As a result, we cannot draw a direct inference of impact, only an association. The exams themselves were also not identical across conditions, both to maintain the integrity of the examinations for fairness, and because some course topics changed between the two semesters. The second midterm exam differed in length as well, with the control group having an exam with fewer questions (and thus more time per question) compared to the intervention group.

7 Future Work

The work outlined in this paper invites several avenues for future work. While we have speculated that differences in question formatting and evaluation caused the SCS1 results to differ from the class exams, further investigation could help determine whether or not this is the case and help guide future research in the use and development of concept inventories and other assessments. We also noted that the Droplet editor, and more broadly most bi-directional dual-modality environments, do not support blocks representations of certain object-oriented programming language concepts such as inheritance, which may have contributed to similarities between the control and intervention groups on the inheritance-focused code writing questions. Other environments, such as BlueJay and Greenfoot [15, 19], offer ways to represent object-oriented relationships that could be used to improve bi-directional dual modality editor representations. We also touched on connections between our study and the Neo-Piagetian model [20], especially as it relates to question types; future work could expand on this to examine how, and if, support for learning within bi-directional dual-modality environments differs depending on the cognitive stages in the Neo-Piagetian model and, by extension, level of student experience. Finally, we have limited this paper to an examination of the assessment data collected in this study; we plan to examine student perceptions of their experiences through survey responses in detail in future work.

8 Conclusion

In this paper, we have presented an analysis of test scores from a study of 673 college students who completed a CS1 course. This study analyzed data from two 16-week semesters, each with different students. The first semester, with 248 students, served as a control group and was taught using traditional, text-based instruction, while the second semester, with 425 students, was taught using dual-modality instruction and making a dual-modality IDE plugin available to students. We examined student scores on in-class assessments, including midterm and final exams, separating definitional and code reading sections from code writing sections. Our study provides the first evidence that students who learned via dual-modality instruction and tools scored higher on course assessments, indicating that they developed a deeper understanding of programming concepts and better skill in programming. The findings of this study will help educators in college-level computer science courses when deciding on programming environments to help students learn and will offer guidance to researchers in developing new approaches to CS instruction.

ACKNOWLEDGMENTS

We extend our thanks to Aishat Aloba and Shaghayegh Esmaeili for their support in conducting this study and to Amanpreet Kapoor for his support in analyzing the data collected.

REFERENCES

[1] Bau, D. 2015. Droplet, a blocks-based editor for text code. *Journal of Computing Sciences in Colleges*. 30, 6 (2015), 138–144.

[2] Bau, D.A. 2015. Integrating Droplet into Applab -Improving the usability of a blocks-based text editor. *2015 IEEE Blocks and Beyond Workshop* (2015), 55–57.

[3] Bau, D.A. and Bau, D.A. 2014. A Preview of Pencil Code. *Proceedings of the 2nd Workshop on Programming for Mobile & Touch - PROMOTO '14* (2014), 21–24.

[4] Bau, D.A., Bau, D.A., Pickens, C.S. and Dawson, M. 2015. Pencil Code: Block Code for a Text World. *Proceedings of the 14th International Conference on Interaction Design and Children* (2015), 445–448.

[5] Bau, D. and Bau, D.A. 2014. A Preview of Pencil Code. *Proceedings of the 2nd Workshop on Programming for Mobile & Touch - PROMOTO '14*. (2014), 21–24. DOI:https://doi.org/10.1145/2688471.2688481.

[6] Blanchard, J., Gardner-McCune, C. and Anthony, L. 2019. Effects of Code Representation on Student Perceptions and Attitudes Toward Programming. *2019 IEEE Symposium on Visual Languages and Human-Centric Computing* (2019), 127–131.

[7] Blanchard, J., Gardner-McCune, C. and Anthony, L. 2019. Amphibian: Dual-Modality Representation in Integrated Development Environments. *Proceedings of the 2019 IEEE Blocks and Beyond Workshop (Blocks and Beyond)* (2019).

[8] Brainerd, C.J. 1978. *Piaget's Theory of Intelligence*. Prentice Hall.

[9] Brown, N.C.C., Mönig, J., Bau, A. and Weintrop, D. 2016. Panel: Future Directions of Block-based Programming. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (2016), 315–316.

[10] Cooper, S., Dann, W. and Pausch, R. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*. 15, 5 (2000), 107–116. DOI:https://doi.org/10.1145/1953163.1953243.

[11] Cooper, S., Dann, W. and Pausch, R. 2003. Teaching objects-first in introductory computer science. *ACM SIGCSE Bulletin*. 35, 1 (2003), 191–195. DOI:https://doi.org/10.1145/792548.611966.

[12] Corney, M., Teague, D., Ahadi, A. and Lister, R. 2012. Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions. *14th Australasian Computing Education Conference* (2012), 77–86.

[13] Gobet, F., Lane, P.C.R., Croker, S., Cheng, P.C.-H., Jones, G., Oliver, I. and Pine, J.M. 2001. Chunking mechanisms in human learning. *TRENDS in Cognitive Sciences*. 5, 6 (2001), 236–243. DOI:https://doi.org/10.1016/S1364-6613(00)01662-4.

[14] Goode, J. and Margolis, J. 2011. Exploring Computer Science: A Case Study of School Reform. *ACM Transactions on Computing Education*. 11, 2 (2011), 1–16. DOI:https://doi.org/10.1145/1993069.1993076.

[15] Henriksen, P. and Kolling, M. 2004. Greenfoot: Combining Object Visualization with Interaction. *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '04*. (2004), 73–82. DOI:https://doi.org/10.1145/1028664.1028701.

[16] Hsu, K.C. 1996. *Developing Programming Environments for Programmable Bricks*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.

[17] Kalelioğlu, F. 2015. A new way of teaching programming skills to K-12 students: Code.org. *Computers in Human Behavior*. 52, (2015), 200–210. DOI:https://doi.org/10.1016/j.chb.2015.05.047.

[18] Kelleher, C. and Pausch, R. 2005. Lowering the Barriers to Programming: a survey of programming environments and languages for novice programmers. *ACM Computing Surveys*. 37, 2 (2005), 83–137. DOI:https://doi.org/10.1145/1089733.1089734.

[19] Kölling, M. 2008. Using BlueJ to Introduce Programming. *Reflections on the Teaching of Programming*. Springer-Verlag. 98–115.

[20] Lister, R. 2011. Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. *Proceedings of the Thirteenth Australasian Computing Education Conference* (2011), 9–18.

[21] Lister, R. 2011. COMPUTING EDUCATION RESEARCH: Programming, syntax and cognitive load. *ACM Inroads*.

[22] Malan, D.J. and Leitner, H.H. 2007. Scratch for budding computer scientists. *ACM SIGCSE Bulletin*. 39, 1 (2007), 223–227. DOI:https://doi.org/10.1145/1227504.1227388.

[23] Maloney, J., Rusk, N., Burd, L., Silverman, B., Kafai, Y., Resnick, M., Rusk, N., Silverman, B. and Resnick, M. 2004. Scratch: A sneak preview. *Proceedings - Second International Conference on Creating, Connecting and Collaborating Through Computing*. (2004), 104–109. DOI:https://doi.org/10.1109/C5.2004.1314376.

[24] Meerbaum-Salant, O. 2010. Learning computer science concepts with scratch. *ICER '10: Proceedings of the Sixth international workshop on Computing education research*. (2010), 69–76. DOI:https://doi.org/10.1145/1839594.1839607.

[25] Miller, G.A. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*. 63, 2 (1956), 81–97.

[26] Nachar, N. 2008. The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution. *Tutorials in Quantitative Methods for Psychology*. 4, 1 (2008), 13–20. DOI:https://doi.org/10.20982/tqmp.04.1.p013.

[27] Parker, M.C. and Guzdial, M. 2016. Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment. *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (2016), 93–101.

[28] Teague, D. 2015. *Neo-Piagetian Theory and the Novice Programmer*. Ph.D. Dissertation. Queensland University of Technology, Brisbane, Australia.

[29] Teague, D., Comey, M., Ahadi, A., Lister, R., Corney, M., Ahadi, A. and Lister, R. 2013. A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers. *Proceedings of the Fifteenth Australasian Computing Education Conference* (2013), 87–95.

[30] Teague, D. and Lister, R. 2014. Programming: reading, writing and reversing. *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*. (2014), 285–290. DOI:https://doi.org/10.1145/2591708.2591712.

[31] Tharp, A.L. 1982. Selecting the "right" programming language. *Proceedings of the thirteenth SIGCSE technical symposium on Computer science education* (1982), 151–155.

[32] The Joint Task Force on Computing Curricula - ACM/IEEE-Computer Society 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*.

[33] Weintrop, D. 2016. *Modality Matters: Understanding the Effects of Programming Language Representation in High School Computer Science Classrooms*. Ph.D. Dissertation. Northwestern University, Evanston, IL.

[34] Weintrop, D. and Holbert, N. 2017. From Blocks to Text and Back: Programming Patterns in a Dual-modality Environment. *Proceedings of the 48th ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. (2017). DOI:https://doi.org/10.1145/3017680.3017707.

[35] Weintrop, D. and Wilensky, U. 2015. To block or not to block? That is the question. *Proceedings of the 14th International Conference on Interaction Design and Children*. (2015), 199–208. DOI:https://doi.org/10.1016/j.jtcvs.2015.01.023.

[36] Weintrop, D. and Wilensky, U. 2016. Bringing blocks-based programming into high school computer science classrooms. *Annual Meeting of the American Educational Research Association* (2016).

[37] Wing, J.M., Jeannette, V. and Wing, J.M. 2006. Computational thinking. *Communications of the ACM*. 49, 3 (2006), 33–35. DOI:https://doi.org/10.1145/1118178.1118215.

[38] IntelliJ IDEA: *https://www.jetbrains.com/idea/*.