Text-Based vs. Block-Based and Structural Editors **Epic Literature Review**

Reading Reflection

Discuss in groups

- Based on the readings for today, come up with:
  - 3 task-audience combinations for which you'd instantiate a language in a non-projectional editor
  - 3 task-audience combinations for which you'd instantiate a language in a projectional editor

Projectional editors and…

- …folk theories! :)
- I didn’t hear too much folk theory stuff coming out in Tuesday’s discussion—well done, team!
- But projectional editors are a common site of folk theories, so we’re going to go over some evidence.

Us vs. Them

Evidence That Computer Science Grades Are Not Bimodal

Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada
patitsas,mcraig,sme@cs.toronto.edu and jesse.berlin@mail.utoronto.ca

ABSTRACT

Although it has never been rigourously demonstrated, there is a common belief that CS grades are bimodal. We statistically analyzed 778 distributions of final course grades from a large research university, and found only 5.8% of the distributions passed tests of multimodality. We then devised a psychology experiment to understand why CS educators believe their grades to be bimodal. We showed 53 CS professors a series of histograms displaying ambiguous distributions and asked them to categorize the distributions. A random half of participants were primed to think about the fact that CS grades are commonly thought to be bimodal; these participants were more likely to label ambiguous distributions as "bimodal". Participants were also more likely to label distributions as bimodal if they believed that some students are innately predisposed to do better at CS. These results suggest that bimodal grades are instructional folklore in CS, caused by confirmation bias and instructor beliefs about their students.

1 INTRODUCTION

inform their practice [13], and these beliefs may or may not be based on empirical evidence.

1.1 Explanations of Bimodality

A number of explanations have been presented for why CS grades are bimodal, all of which begin with the assumption that this is the case.

1.1.1 Prior Experience

A bimodal distribution generally indicates that two distinct populations have been sampled together [5]. One explanation for bimodal grades is that CS1 classes have two populations of students: those with experience, and those without it [1].

High school CS is not common in many countries, and so students enter university CS with a range of prior experience. However, this explanation fits students into two bins. Prior experience is not as simple as "have it" vs. not – there is a large range on how much prior experience students can have programming, and practice with non-programming languages like HTML/CSS could also be beneficial [21].

Although it has never been rigourously demonstrated, there is a common belief that CS grades are bimodal. We statistically analyzed 778 distributions of final course grades from a large research university, and found only 5.8% of the distributions passed tests of multimodality. We then devised a psychology experiment to understand why CS educators believe their grades to be bimodal. We showed 53 CS professors a series of histograms displaying ambiguous distributions and asked them to categorize the distributions. A random half of participants were primed to think about the fact that CS grades are commonly thought to be bimodal; these participants were more likely to label ambiguous distributions as "bimodal". Participants were also more likely to label distributions as bimodal if they believed that some students are innately predisposed to do better at CS. These results suggest that bimodal grades are instructional folklore in CS, caused by confirmation bias and instructor beliefs about their students.

What stood out for us is that at both UBC and UToronto, the CS faculty would routinely assert that their CS grades are bimodal – and we now had evidence to the contrary.

Our results support Lister's argument that CS grades are generally not bimodal, and that the perception of bimodality comes from instructors expecting their grades to be [17].

If you plan to spend any additional years in CS at all, I highly recommend reading the whole paper:
https://dl.acm.org/doi/10.1145/2960310.2960312

Here's a picture of a rainbow so you can find this slide and therefore this link later!
https://www.johnentwistlephotography.com/

Text- vs. Structure-Based Editors: The lit review

https://abc7news.com/society/end-of-the-decade-googles-top-trends-of-the-2010s/5749300/

Going to focus on the about the last 10 years of the literature, since the editors available have changed a fair amount.
But we'll also take a look at a lit review that covers prior work.

Goals for the upcoming flood of data

- That we all leave recalling a few of the key insights that have come up with repeatedly
- That we know we don't have to rely on folk theories!
    - This is the sneaky secondary agenda for today, to remind ourselves that even as we're learning how to evaluate PLs' effects on programmers, we can also learn a lot from the body of knowledge that researchers have already built up in this space!

Computers & Education 142 (2019) 103646

Contents lists available at ScienceDirect

Computers & Education

journal homepage: www.elsevier.com/locate/compedu

Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms

David Weintrop[a,*], Uri Wilensky[b]

[a] College of Education, College of Information Studies, University of Maryland, College Park, USA
[b] Center for Connected Learning and Computer-based Modeling, Northwestern University, USA

ARTICLE INFO

Keywords:
Evaluation of CAL systems
Interactive learning environments
Programming and programming languages
Secondary education
Teaching/learning strategies

ABSTRACT

Block-based programming languages are becoming increasingly common in introductory computer science classrooms across the K-12 spectrum. One justification for the use of block-based environments in formal educational settings is the idea that the concepts and practices developed using these introductory tools will prepare learners for future computer science learning opportunities. This view is built on the assumption that the attitudinal and conceptual learning gains made while working in the introductory block-based environments will transfer to conventional text-based programming languages. To test this hypothesis, this paper presents the

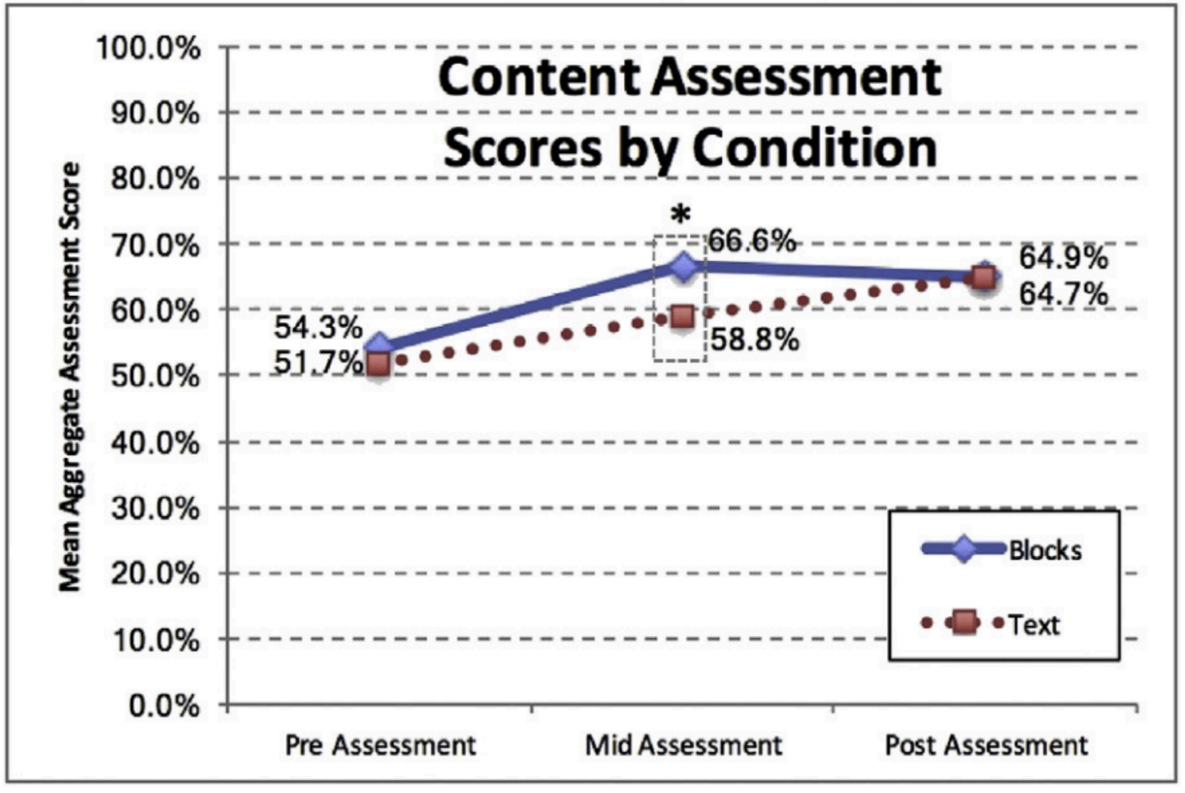
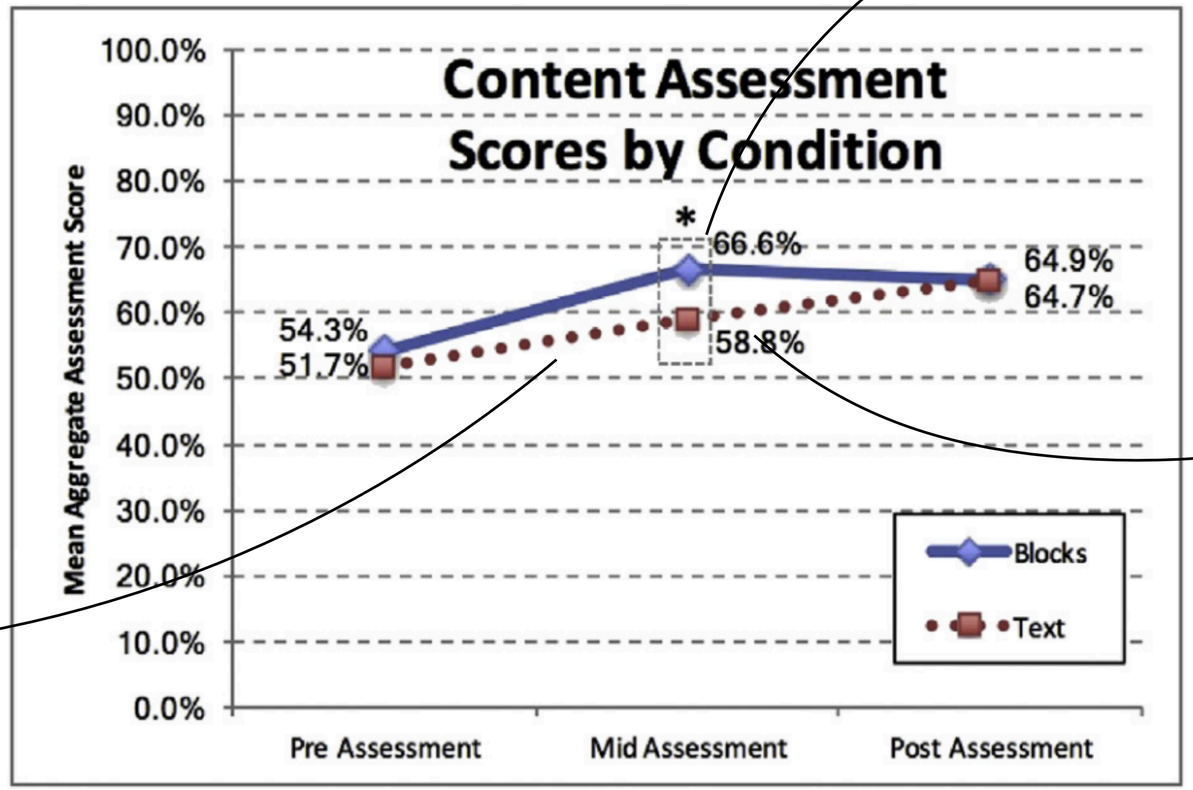


Fig. 3. The mean scores for students in the two conditions on the three administrations (Pre, Mid, and Post) of the Commutative Assessment.

Statistically significant



Students using either blocks- or text-based programming environments for the same language (pencil.cc)

At this point, all students switch to text-based Java

No particular pedagogic approach to the transition. They just switched.

Fig. 3. The mean scores for students in the two conditions on the three administrations (Pre, Mid, and Post) of the Commutative Assessment.

(53) = 2.03, p = 0.04, d = 0.58). This means that after 5 weeks, students learning to program in a block-based environment performed significantly better on a content assessment than peers using an isomorphic text-based environment.

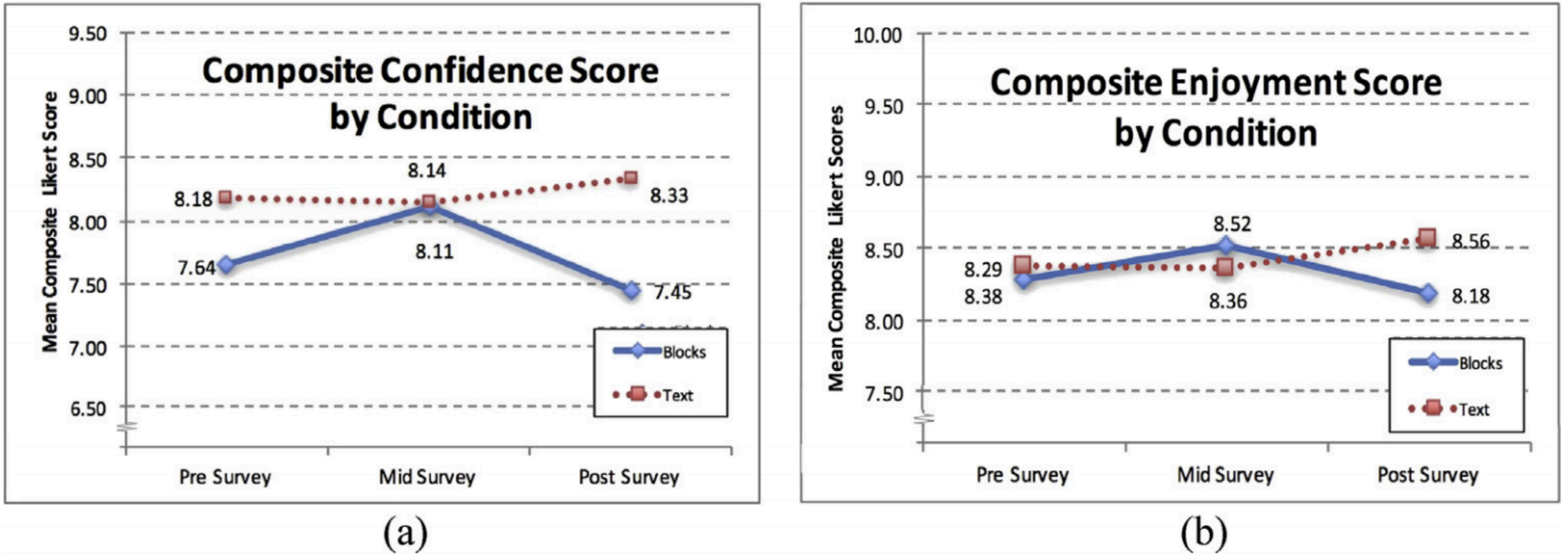


Fig. 4. Composite scores of students' confidence (a) and enjoyment (b) in programming at three points in the study.

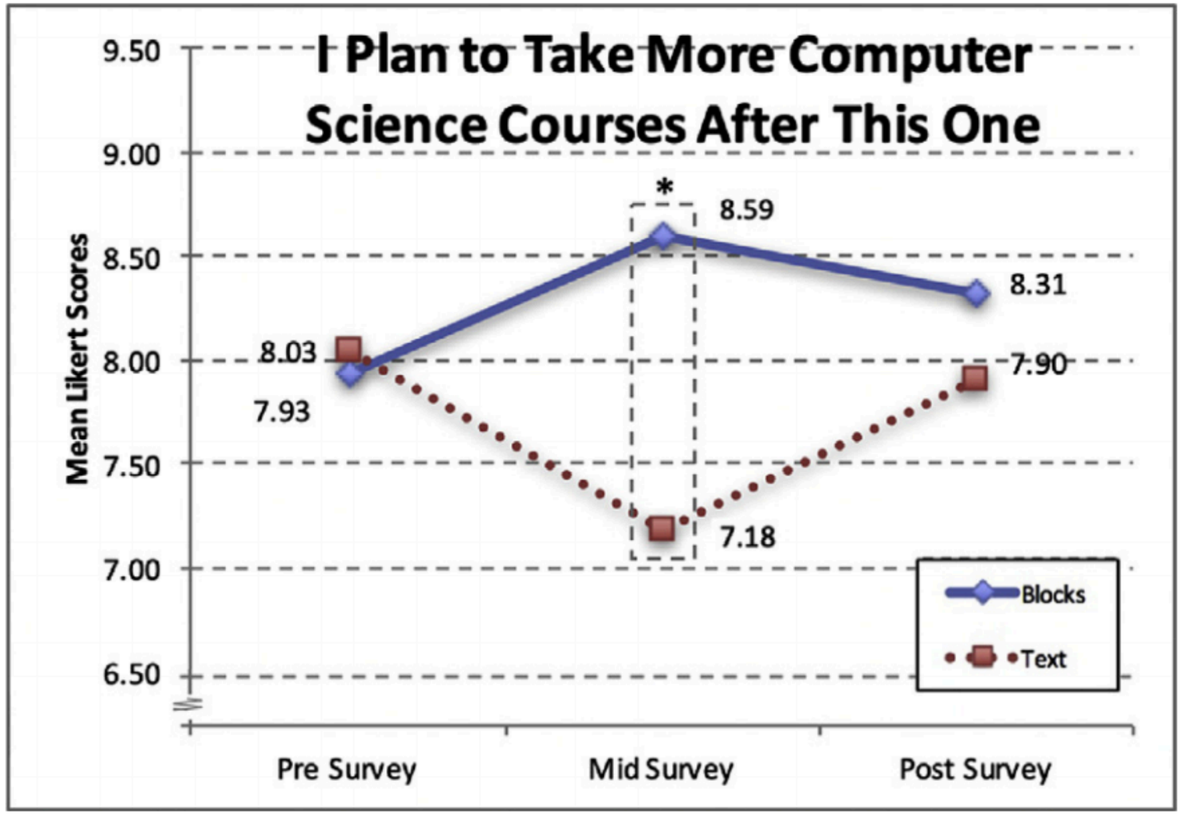


Fig. 5. The mean responses scores, grouped by condition, for the statement: I plan to take more computer science courses after this one.

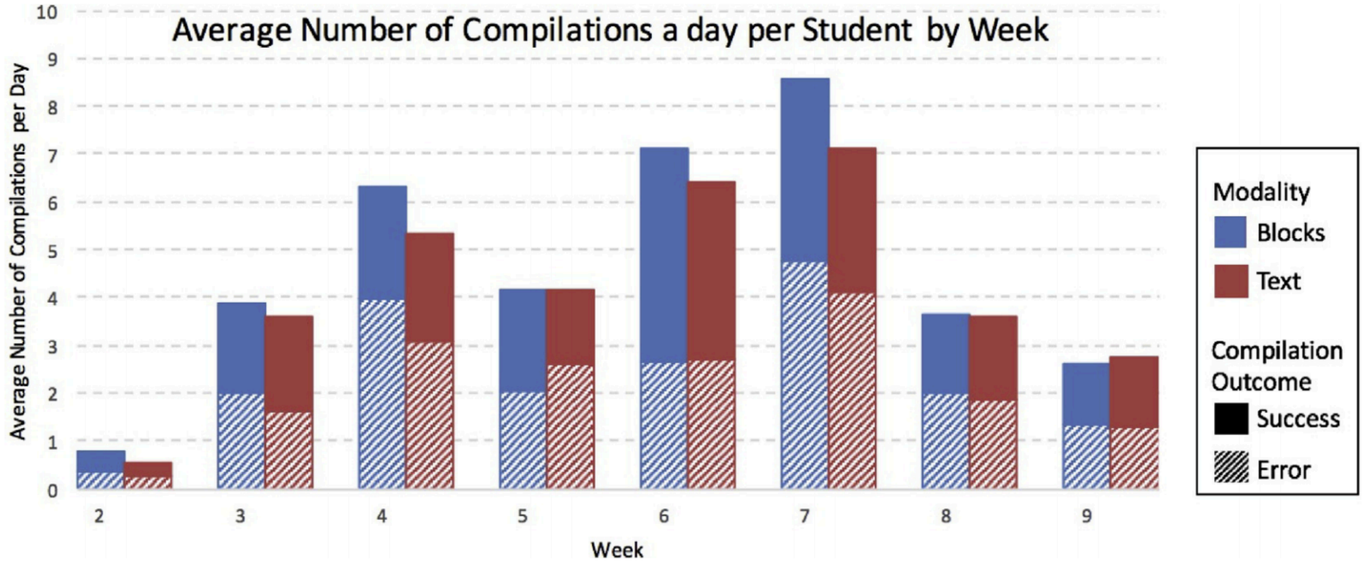


Fig. 6. The average number of `javac` calls per student per day grouped by week. The solid portion of each bar represents successful calls; the striped portions represent erroneous calls.

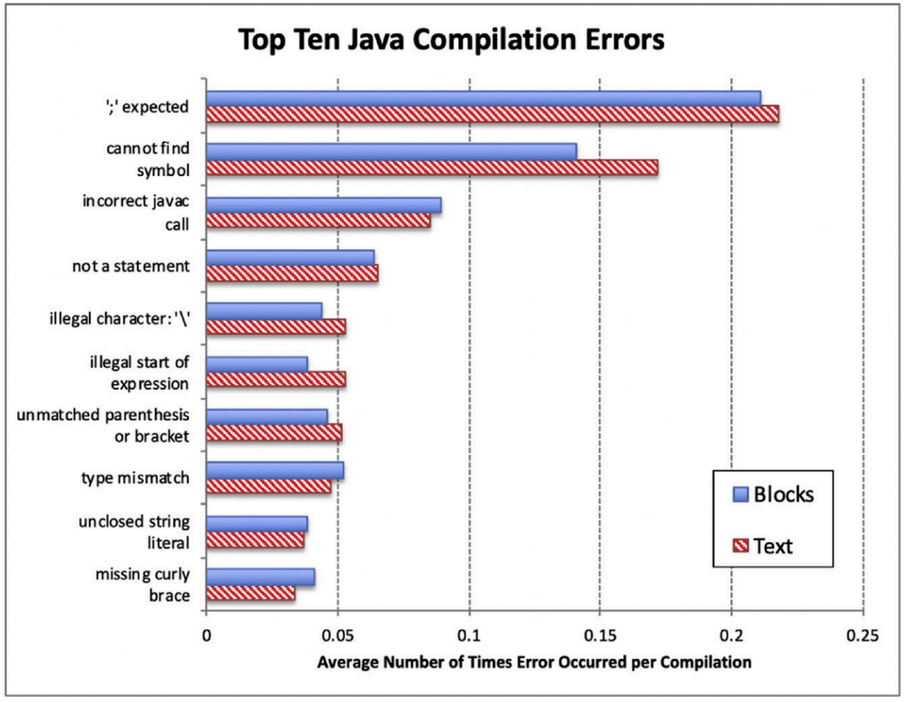


Fig. 7. The ten most frequently encountered Java errors, grouped by condition.

Table 1
High-level descriptive patterns of failing compilations and errors over the course of the 10 weeks.

	Failed `javac` calls per student	Compilation errors per student	Compilation errors per failed `javac` call
Blocks	75.11	165.78	2.23
Text	69.55	164.26	2.21

Table 2
The frequency of successful compilations with a given Levenshtein distance from the last successful compilation of the same program.

	Levenshtein Distance								
	0	1	2	3	4	5–10	11–25	26–100	> 100
Blocks	7.00	3.37	5.70	1.33	2.37	4.30	3.52	6.56	3.33
Text	6.16	3.00	5.58	1.23	2.13	3.77	3.48	5.87	2.55

Basically, programming approach once they switched to Java was the same.
(Differences not statistically significant.)

Block-based Comprehension: Exploring and Explaining Student Outcomes from a Read-only Block-based Exam

David Weintrop
University of Maryland
College Park, MD, USA
weintrop@umd.edu

Heather Killen
University of Maryland
College Park, MD, USA
hkillen@umd.edu

Talal Munzar
University of Maryland College Park, MD, USA
tmunzar@terpmail.umd.edu

Baker Franke
Code.org
Seattle, WA, USA
baker@code.org

ABSTRACT

The success of block-based programming environments like Scratch and Alice has resulted in a growing presence of the block-based modality in classrooms. For example, in the United States, a new, nationally-administered computer science exam is evaluating students' understanding of programming concepts using both block-based and text-based presentations of short programs written in a custom pseudocode. The presence of the block-based modality on a written exam in an unimplemented pseudocode is a far cry from the informal, creative, and live coding contexts where block-based programming initially gained popularity. Further, the design of the block-based pseudocode used on the exam includes few of the features cited in the research as contributing to positive learner experiences. In this paper, we seek to understand the implications of the inclusion of an unimplemented block-based pseudocode on a written exam. To do so, we analyze responses from over 5,000 students to a 20 item assessment that included both block-based and text-based questions written in the same

programming tools into K-12 classrooms. While some block-based programming environments have a long history in formal educational contexts (e.g. Alice), other block-based tools were specifically designed for informal learning spaces (e.g. Scratch). As part of the transition of block-based programming into K-12 classrooms, the modality is starting to be used in ways quite distinct from how it was initially designed. Nowhere is this clearer than when it comes to assessment.

Many introductory computer science courses assess student knowledge through written exams that ask students questions about specific syntactic features of a programming language and evaluate student comprehension of programs. While not ideal, such questions lend themselves well to the multiple-choice question format and thus can be graded quickly and objectively. As a result, written, multiple choice assessments are common in introductory computing contexts.

The rise of block-based programming environments in classrooms presents an interesting challenge for educators. What

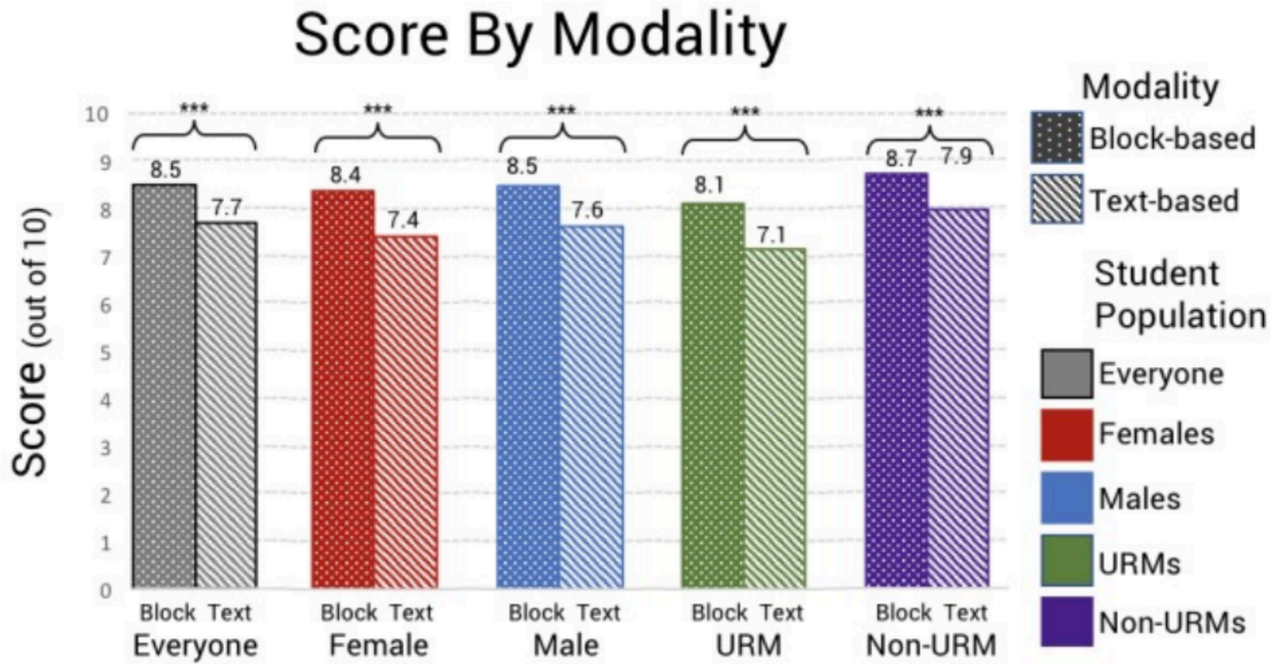


Figure 3. Average scores on the assessment by modality.

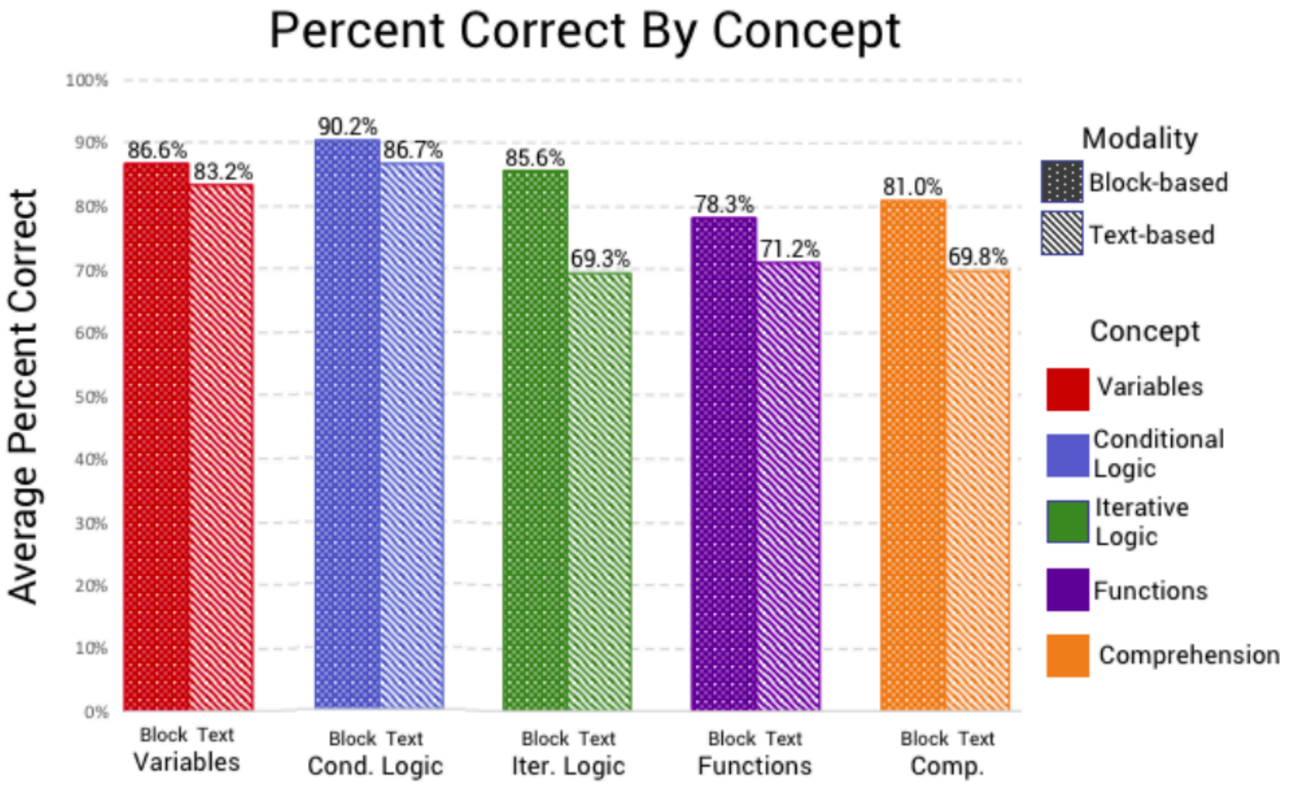


Figure 4. Average number of students who answered a question correctly grouped by concept and modality

ARTICLE IN PRESS

International Journal of Child-Computer Interaction ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

International Journal of Child-Computer Interaction

journal homepage: www.elsevier.com/locate/ijcci

How block-based, text-based, and hybrid block/text modalities shape novice programming practices

David Weintrop [a,*], Uri Wilensky [b]

[a] Teaching & Learning, Policy & Leadership, College of Education and College of Information Studies, University of Maryland, 3942 Campus Dr. Suite 2226D, College Park, MD 207421427, USA

[b] Center for Connected Learning and Computer-Based Modeling, Learning Sciences and Computer Science, Northwestern University, 2120 Campus Dr. Evanston, IL, 60208, USA

ARTICLE INFO

Article history:
Received 10 February 2017
Received in revised form 9 April 2018
Accepted 30 April 2018
Available online xxxx

Keywords:
Design
Modality
Programming Environments
Computer Science Education
Block-based Programming

ABSTRACT

There is growing diversity in the design of introductory programming environments. Where once all novices learned to program in conventional text-based languages, today, there exists a growing ecosystem of approaches to programming including graphical, tangible, and scaffolded text environments. To date, relatively little work has explored the relationship between the design of novice programming environments and the programming practices they engender in their users. This paper seeks to shed light on this dimension of learning to program through the careful analysis of novice programmers' experiences learning with a hybrid block/text programming environment. Specifically, this paper is concerned with how novices leverage the various affordances designed into programming environments and programming languages to support their early efforts to author programs. We explore this relationship through the construct of modality using data from a study conducted in a high school computer science classroom in which students spent five weeks working in block-based, text-based, and hybrid block/text programming environments. This paper uses a detailed vignette of a novice writing a program in the hybrid environment as a way to characterize emerging programming practices, then presents analyses of programming trends from the full study population to speak to the generality of the practices

Non-chart but still interesting…

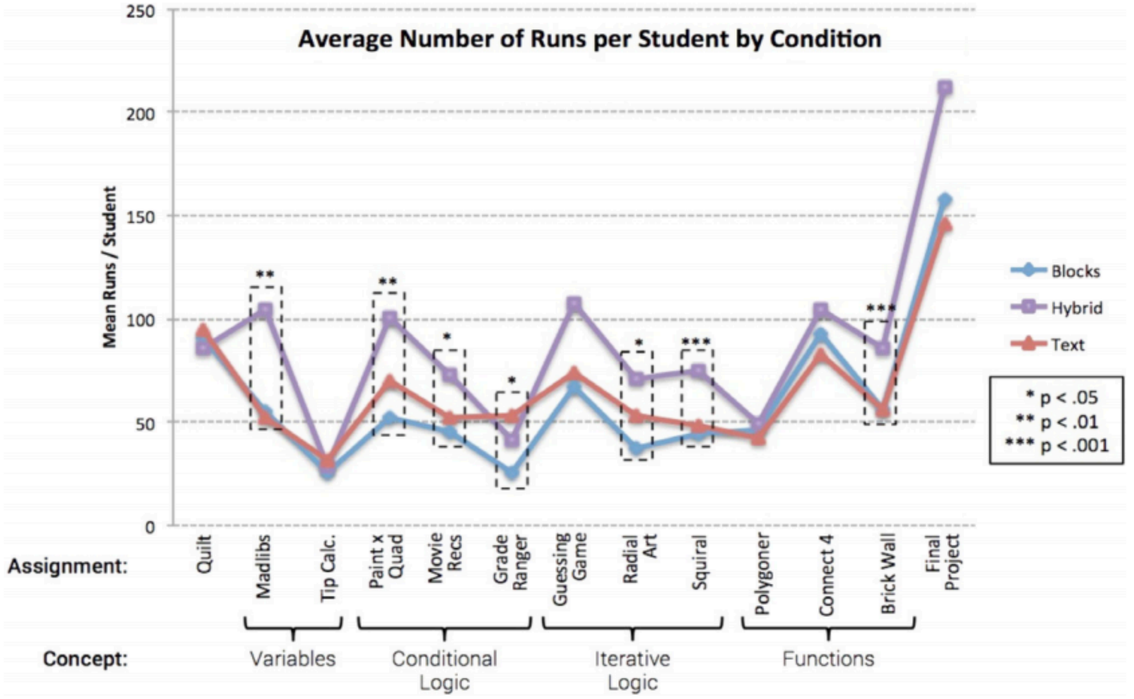


Fig. 5. The average number of runs by students for each project chronologically, broken down by condition.

At the same time, the Blocks modality makes it easy for students to quickly add commands to their program because dragging-and-dropping is faster than typing in commands one character at a time. As a result, on average, students in the Blocks condition produced programs that were longer in length than their Text and Hybrid peers. On 10 of the 13 assignments in the 5-week curriculum the Blocks students produced the longest programs on average, with students in the Hybrid condition producing the longest programs in the other three assignments. Running an ANOVA calculation for each of the assignments, four were found to have statistically significant differences across conditions at the $p < .05$ level: Tip Calculator ($F(2, 82) = 4.78$, $p = .01$) , Grade Ranger ($F(2, 71) = 5.26$, $p = .01$), Radial Art ($F = (2, 83) = 3.51$, $p = .03$) and Connect 4 ($F(2, 87) = 2.90$, $p = .05$). In all but the Connect 4 assignment,

to accomplish relative to the other assignments.[3] The fact that we see a difference in conditional logic is another piece of evidence towards the larger trend of modality affecting students' learning and use of those constructs [41,58]. In this case, we are using program length as a rough proxy for ease of composition given that all conditions had the same time on task. The fact that programs can be assembled more easily contributes to students running their

Evaluating CoBlox: A Comparative Study of Robotics Programming Environments for Adult Novices

David Weintrop[1], Afsoon Afzal[2], Jean Salac[3], Patrick Francis[4], Boyang Li[4], David C. Shepherd[4], Diana Franklin[3]

[1]University of Maryland, College Park, Maryland, United States
[2]Carnegie Mellon University, Pittsburgh, Pennsylvania, United States
[3]University of Chicago, Chicago, Illinois, United States
[4]ABB Corporate Research, Raleigh, North Carolina, United States

weintrop@umd.edu, afsoona@cs.cmu.edu, {salac, dmfranklin}@uchicago.edu,
{patrick.francis, boyang.li, david.shepherd}@us.abb.com

ABSTRACT

A new wave of collaborative robots designed to work alongside humans is bringing the automation historically seen in large-scale industrial settings to new, diverse contexts. However, the ability to program these machines often requires years of training, making them inaccessible or impractical for many. This paper rethinks what robot programming interfaces could be in order to make them accessible and intuitive for adult novice programmers. We created a block-based interface for programming a one-armed industrial robot and conducted a study with 67 adult novices comparing it to two programming approaches in widespread use in industry. The results show participants using the block-based interface successfully implemented robot programs faster with no loss in accuracy while reporting higher scores for usability, learnability, and overall satisfaction. The contribution of this work is showing the

is finding that automation does not necessarily replace workers, but it does change the nature of the work [9].

Collaborative robots, which are intended to work safely alongside humans, exemplify this trend [12,22,27]. Collaborative robots take advantage of "the interplay between machine and human comparative advantage [that] allows computers to substitute for workers in performing routine, codifiable tasks while amplifying the comparative advantage of workers in supplying problem-solving skills, adaptability, and creativity" [9]. In order to support new challenges that emerge from being placed in smaller factories and given a wider variety of tasks, these new robots must be safe, efficient and, support quick reprogramming.

While the design of the machines themselves has resulted in more powerful and flexible robots with a greater set of

CoBlox

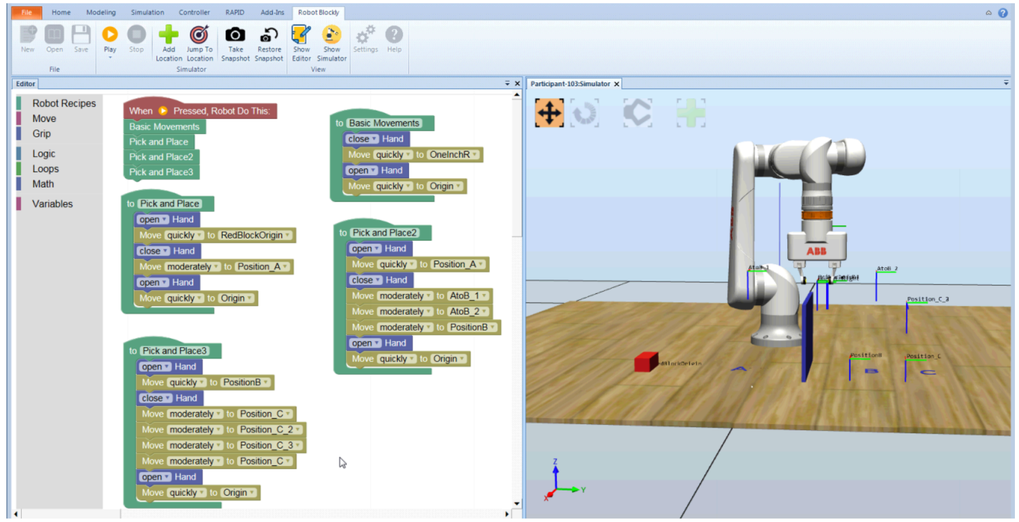


Figure 1. The CoBlox programming environment. The left side of the environment contains the block-based robot programming interface for Roberta, shown on the right.

Flex Pendant

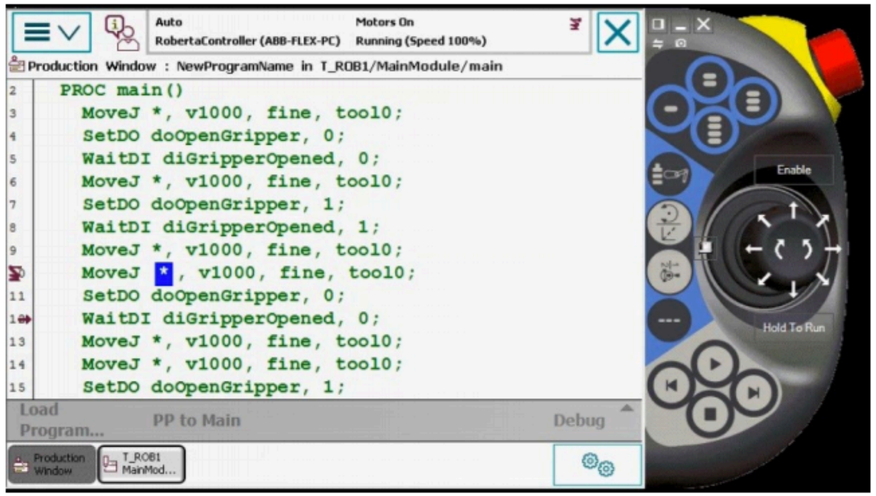
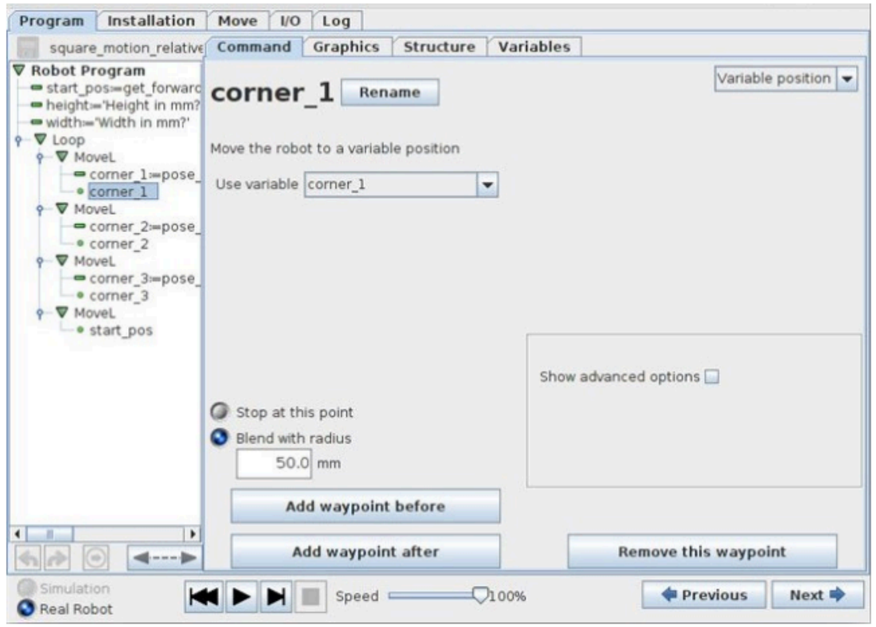


Figure 5. The virtual version of ABB's Flex Pendant programming interface.

Polyscope



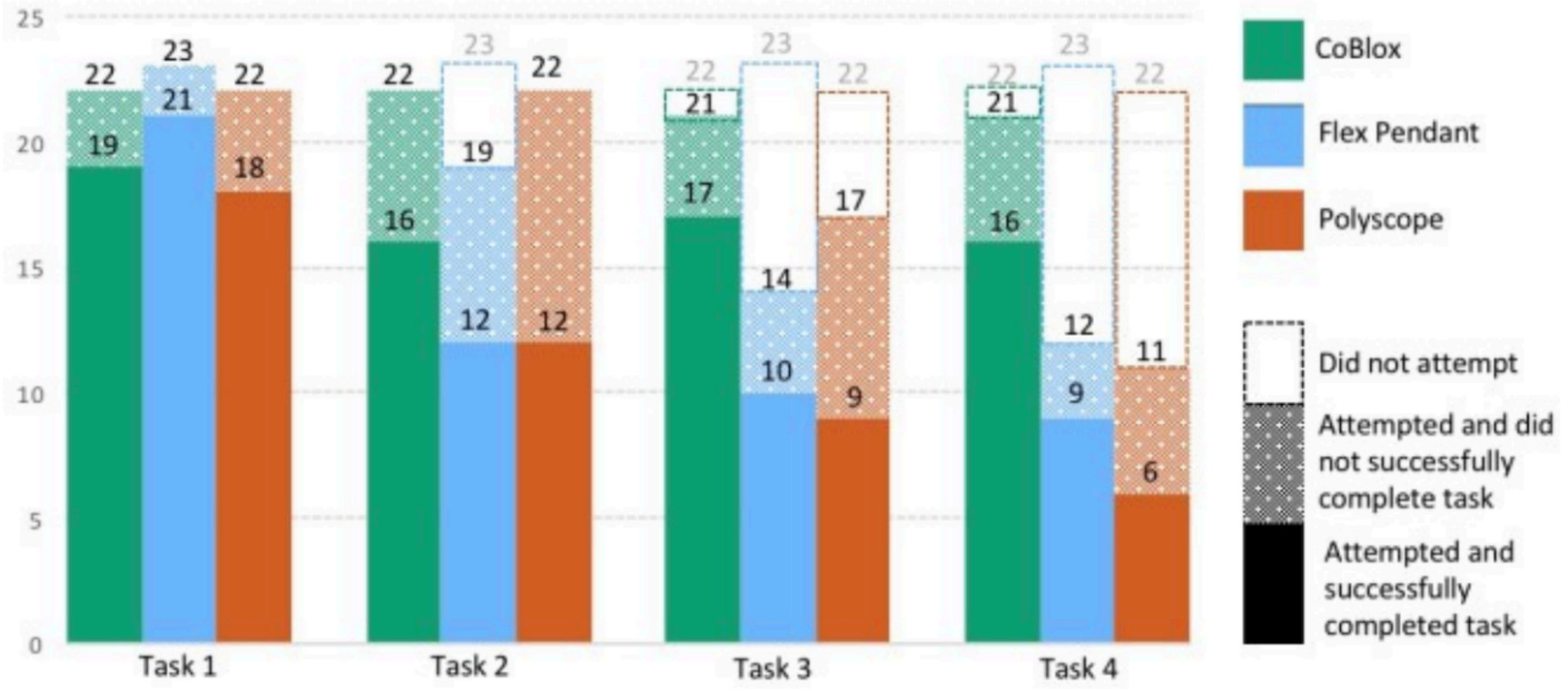


Figure 7. Number of participants that attempted and completed each task, grouped by condition.

	Time on Task (in seconds)			
Condition	**Task 1**	**Task 2**	**Task 3**	**Task 4**
CoBlox	438.36	843.64	481.43	621.29
Flex Pendant	1679.08	1003.32	506.93	605.00
Polyscope	940.73	1398.59	801.76	653.09

Table 1. Time-on-task in seconds for each condition, including only participants that attempted each task.

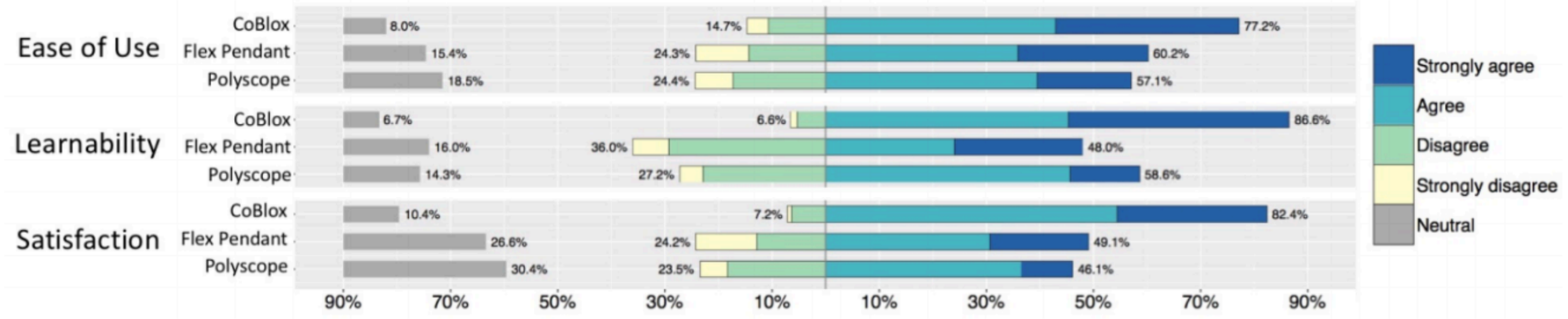


Figure 8. Composite scores for three attitudinal dimensions for the three conditions based on responses to the post survey. The differences between the three conditions are statistically significant for all three categories.

Criteria	CoBlox	Flex Pendant	Polyscope
Faster Task Completion	✓		
More Correct			
Easier to Use	✓		
Easier to Learn	✓		
Higher Satisfaction	✓		

Table 2. Summary of the comparative findings

The paper you read

Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms

DAVID WEINTROP, University of Chicago
URI WILENSKY, Northwestern University

The number of students taking high school computer science classes is growing. Increasingly, these students are learning with graphical, block-based programming environments either in place of or prior to traditional text-based programming languages. Despite their growing use in formal settings, relatively little empirical work has been done to understand the impacts of using block-based programming environments in high school classrooms. In this article, we present the results of a 5-week, quasi-experimental study comparing isomorphic block-based and text-based programming environments in an introductory high school programming class. The findings from this study show students in both conditions improved their scores between pre- and postassessments; however, students in the blocks condition showed greater learning gains and a higher level of interest in future computing courses. Students in the text condition viewed their programming experience as more similar to what professional programmers do and as more effective at improving their programming ability. No difference was found between students in the two conditions with respect to confidence or enjoyment. The implications of these findings with respect to pedagogy and design are discussed, along with directions for future work.

CCS Concepts: • **Social and professional topics** → **Professional topics**; **Computing education**; **K-12 education**;

Additional Key Words and Phrases: Block-based programming, programming environments, design

ACM Reference format:
David Weintrop and Uri Wilensky. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Trans. Comput. Educ.* 18, 1, Article 3 (October 2017), 25 pages.
https://doi.org/10.1145/3089799

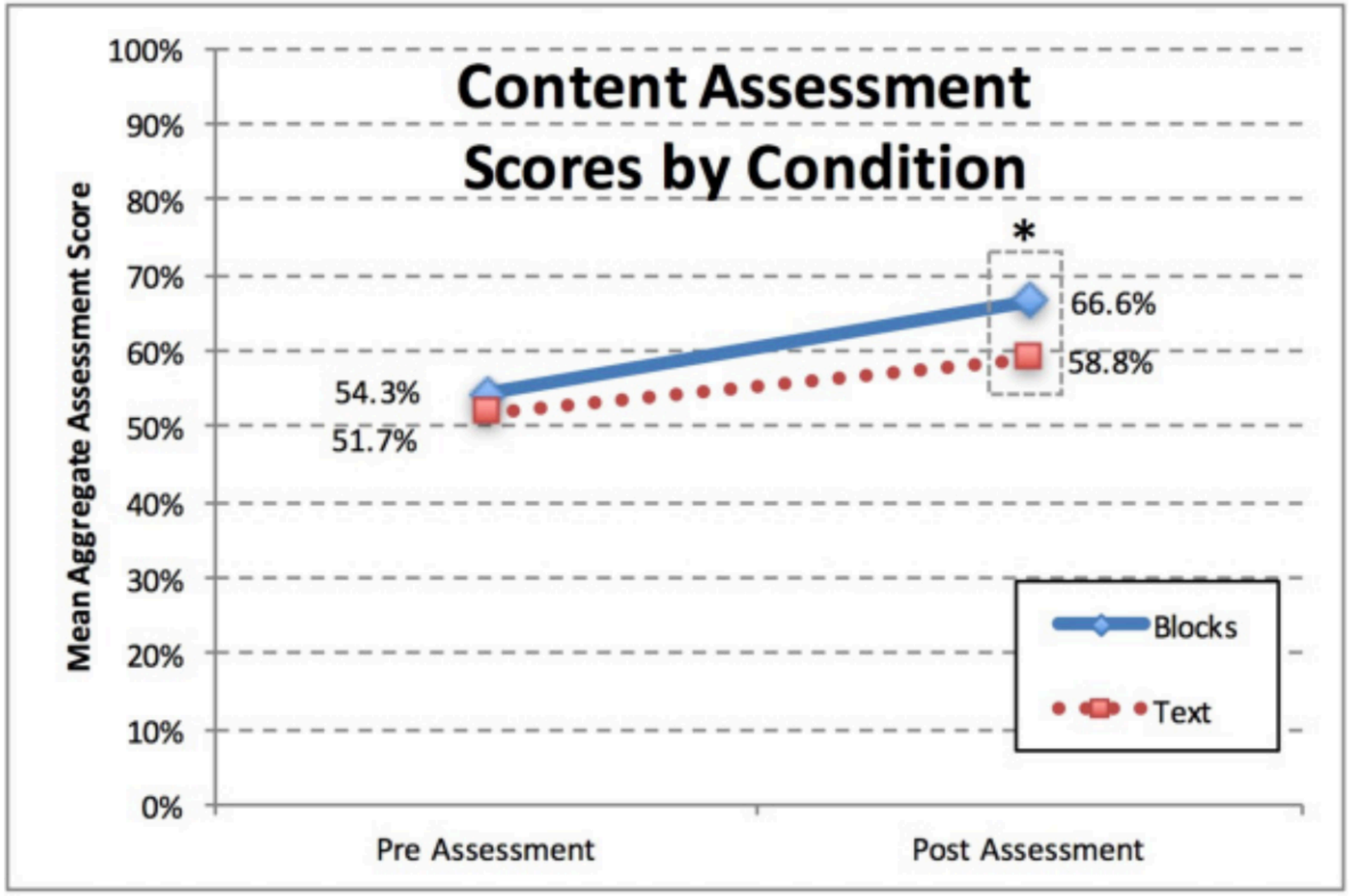
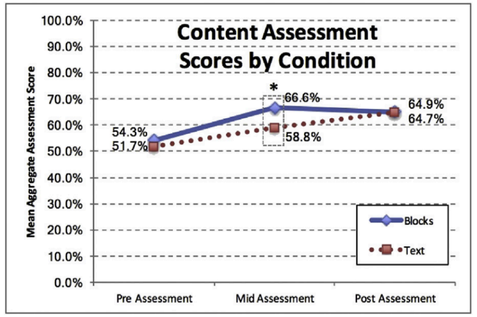


Fig. 4. Student Commutative Assessment scores by condition over time.



ores for students in the two conditions on the three administrations (Pre, Mid, and Post) of the Commutative Assessment.

Prefix of

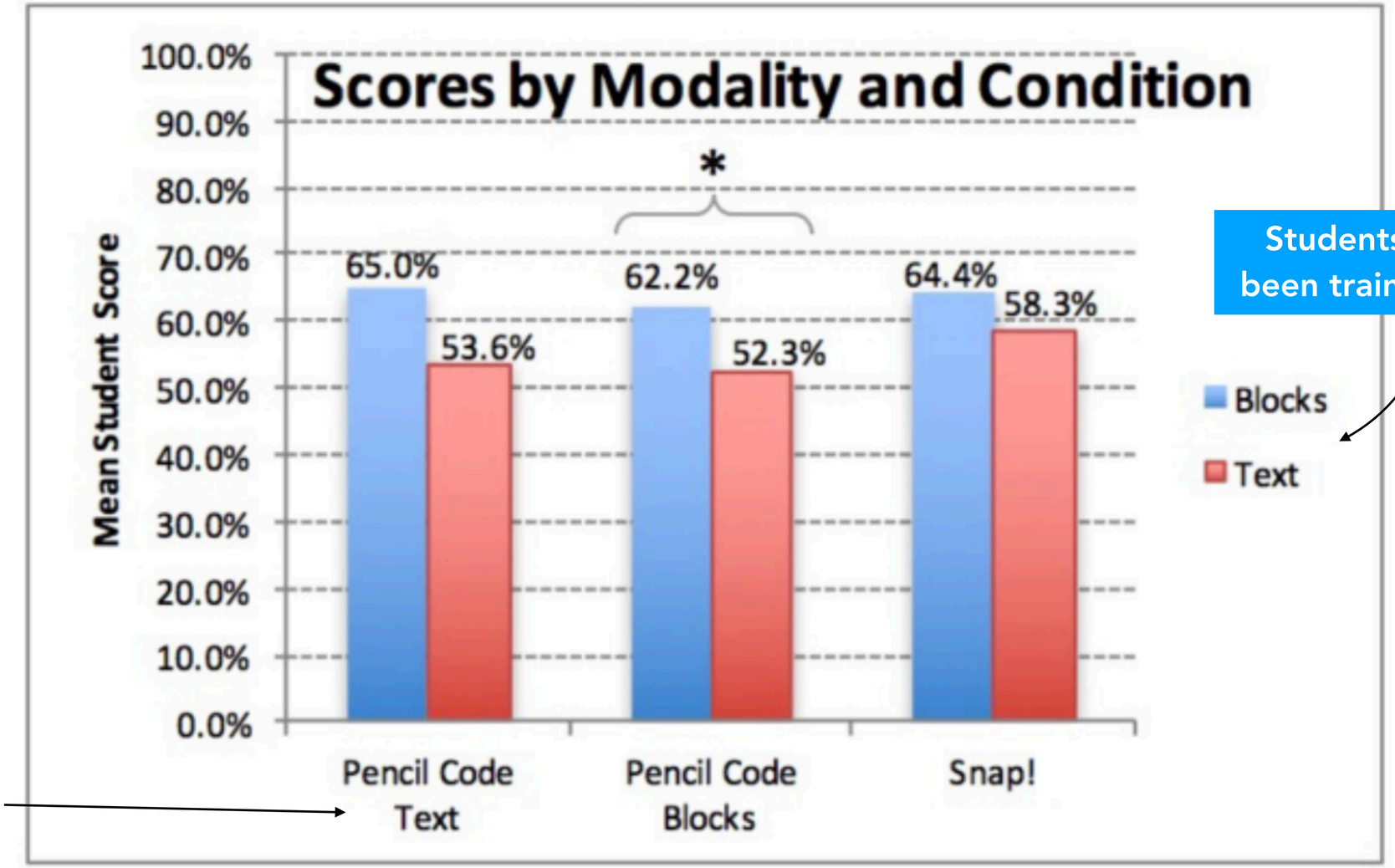


Fig. 5. Student scores on the Commutative Assessment grouped by modality and condition.

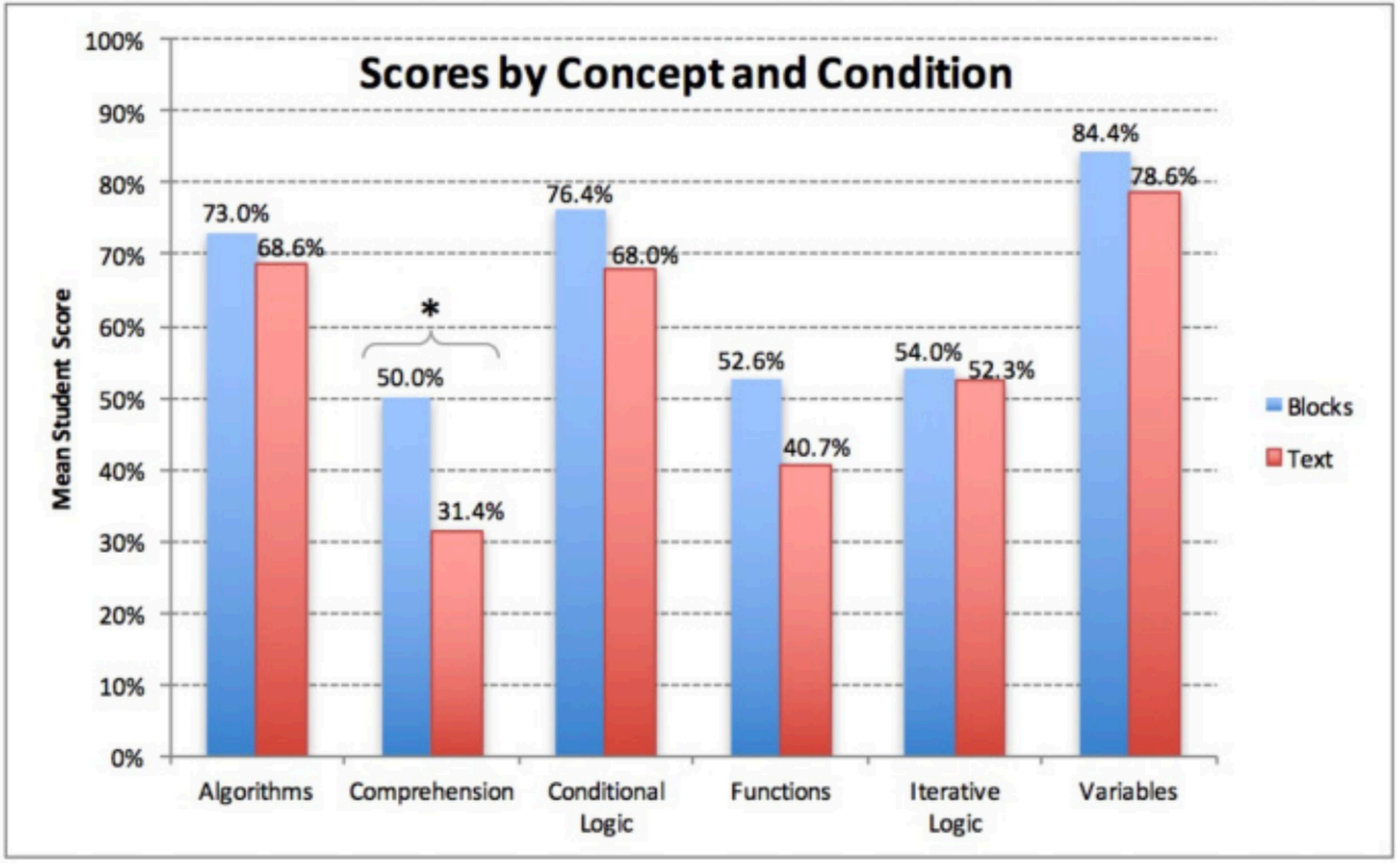


Fig. 6. Student performance on the midpoint administration of the Commutative Assessment grouped by condition and concept.

Table 1. Distribution of Ease-of-Use Responses

		Variables		Loops		Conditional Logic		Functions	
		Blocks	Text	Blocks	Text	Blocks	Text	Blocks	Text
Likert Response	1 (Very Easy)	13	10	11	3	14	8	5	2
	2	7	4	6	7	8	6	6	5
	3	4	5	3	9	2	6	9	7
	4	2	4	3	4	0	2	3	6
	5	0	2	1	3	2	3	1	2
	6	1	2	3	1	0	2	1	3
	7 (Very Hard)	0	1	0	1	1	1	2	3

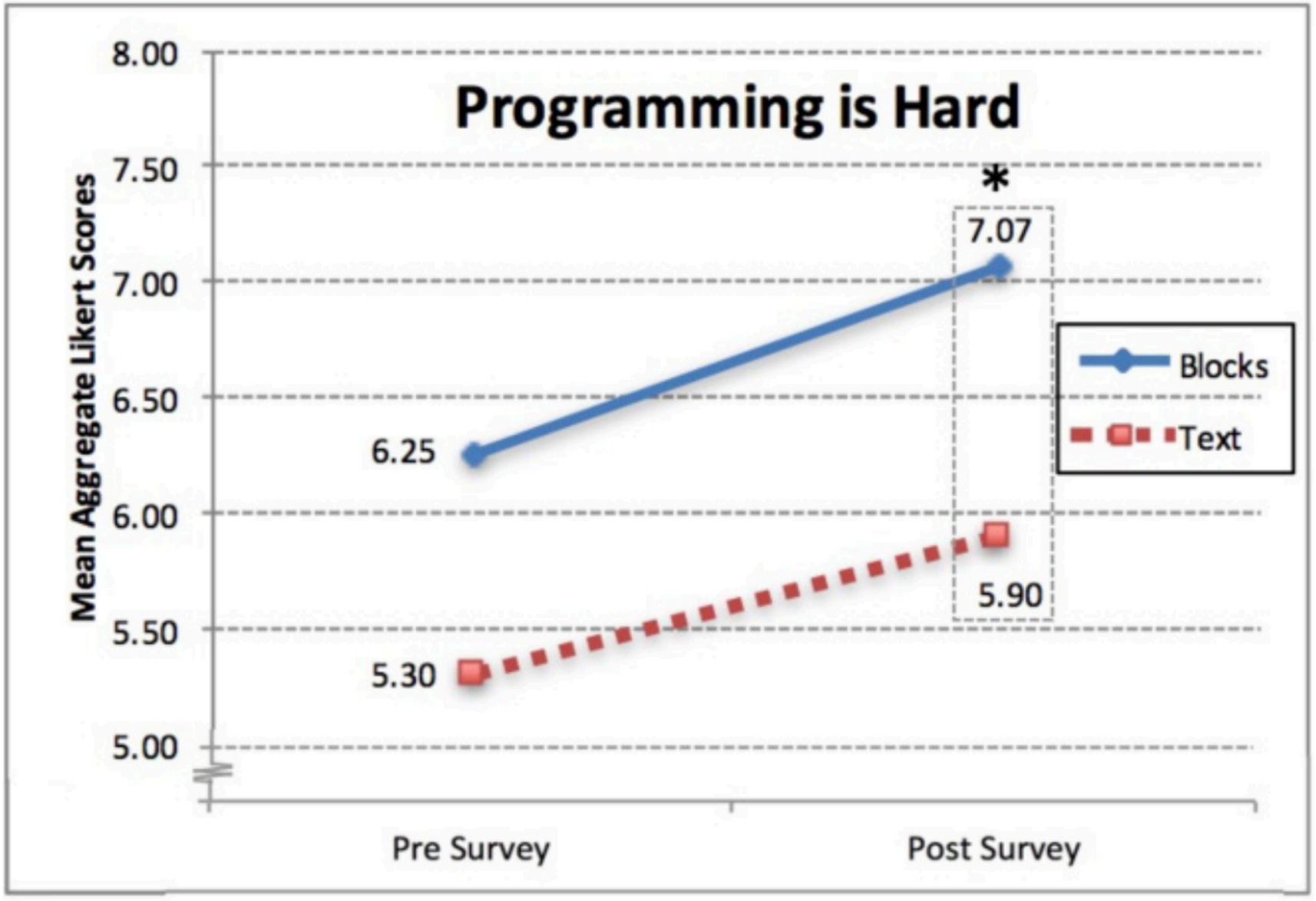


Fig. 9. Average responses to the Likert statement: Programming is hard.

Between a Block and a Typeface: Designing and Evaluating Hybrid Programming Environments

David Weintrop
UChicago STEM Education
University of Chicago
dweintrop@uchicago.edu

Uri Wilensky
Center for Connected Learning and Computer-based Modeling
Northwestern University
uri@northwestern.edu

ABSTRACT

The last ten years have seen a proliferation of introductory programming environments designed for learners across the K-12 spectrum. These environments include visual block-based tools, text-based languages designed for novices, and, increasingly, hybrid environments that blend features of block-based and text-based programming. This paper presents results from a quasi-experimental study investigating the affordances of a hybrid block/text programming environment relative to comparable block-based and textual versions in an introductory high school computer science class. The analysis reveals the hybrid environment demonstrates characteristics of both ancestors while outperforming the block-based and text-based versions in certain dimensions. This paper contributes to our understanding of the design of introductory programming environments and the design challenge of creating and evaluating novel representations for learning.

reviewed environments [10]. Further, we expect this trend to continue as a growing number of libraries are making it easy to develop environments that incorporate a block-based programming interface [12]. This growth in popularity can be seen both in informal environments as well as in classrooms where a growing number of curricula, like Exploring Computer Science [29] and the Beauty and Joy of Computing [13] utilize block-based programming.

Until recently, block-based and text-based programming environments have been distinct. An environment used either one modality or the other. As a result, learners trying to migrate from a block-based environment to a more conventional text-based programming language had few environmental supports to facilitate the transition. Multiple approaches have been developed to mitigate this transition cost. One approach is pedagogical, relying on teachers to assist learners in moving between modalities. An alternative

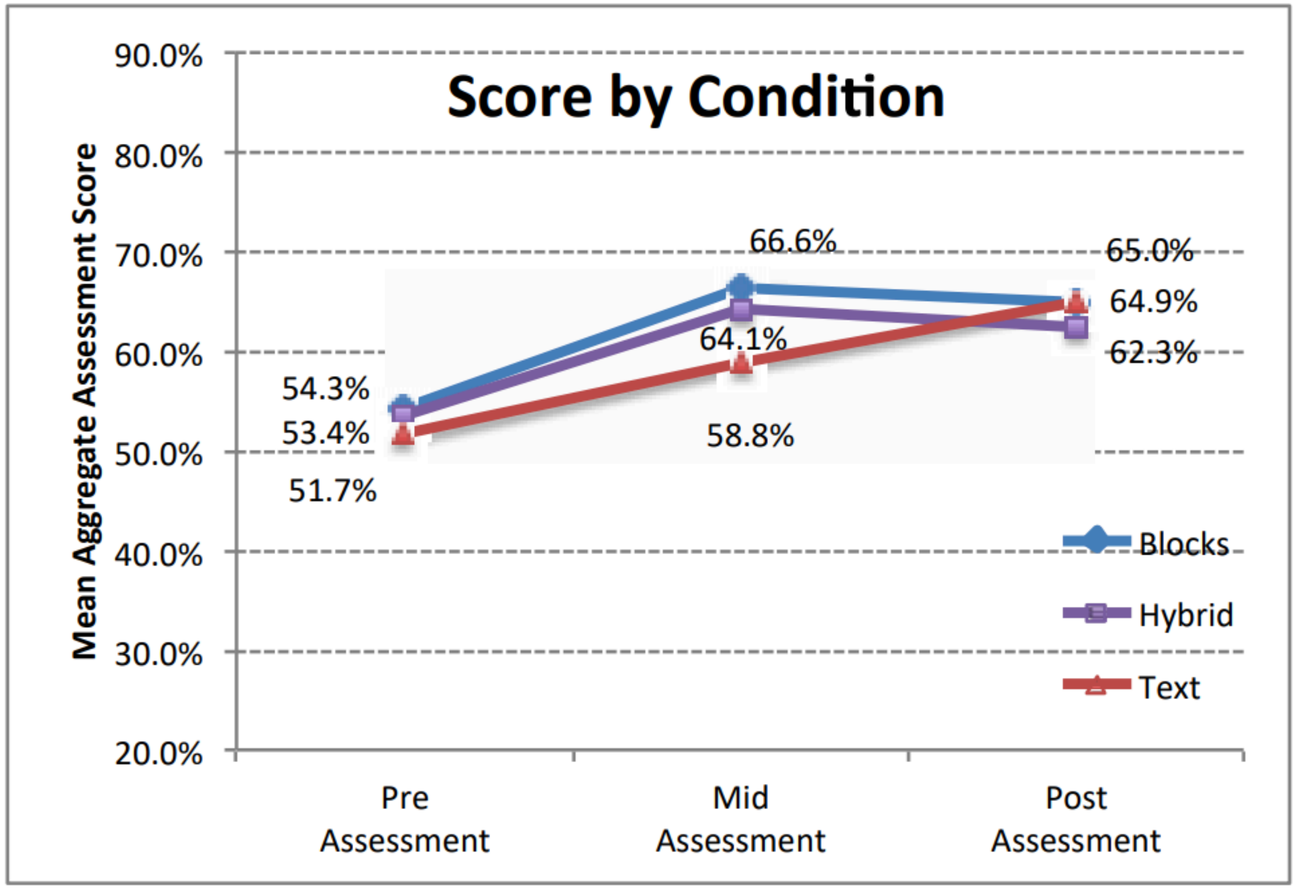


Figure 3. Content scores by condition over time.

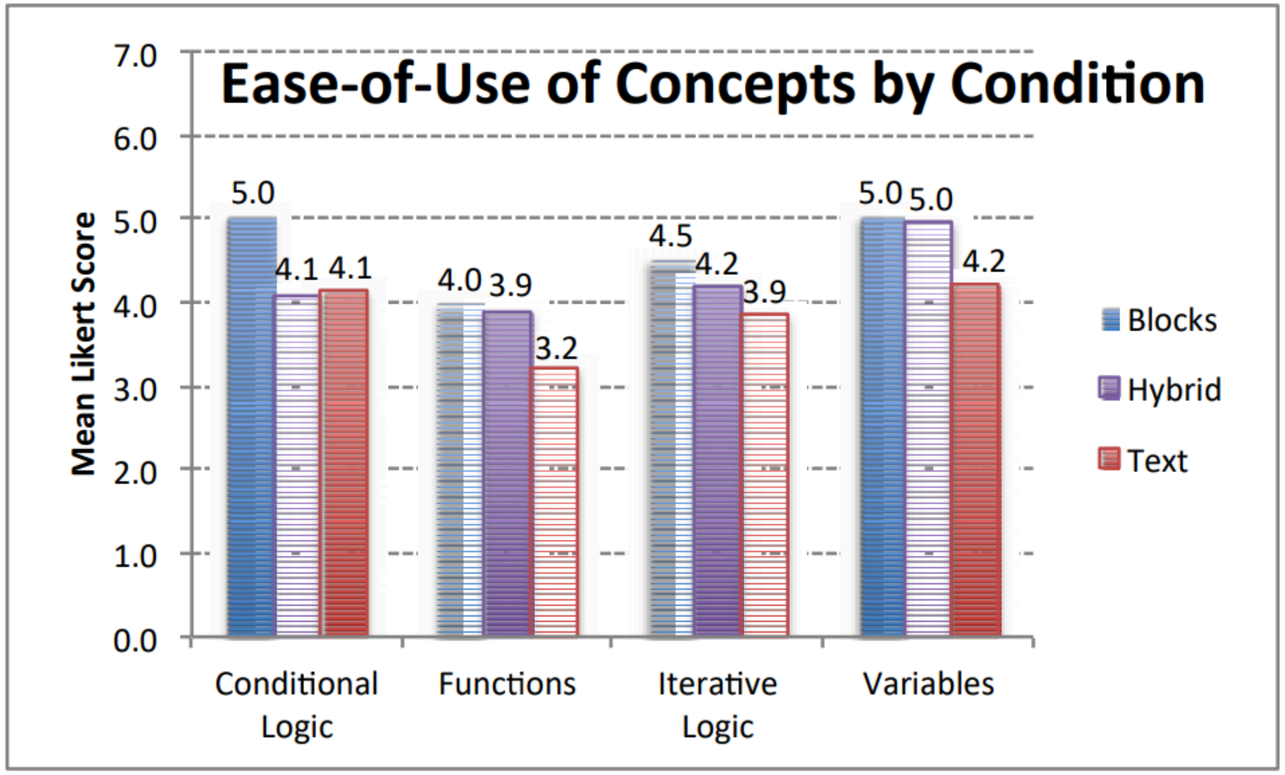


Figure 4. Student reported ease of using concepts in Pencil.cc.

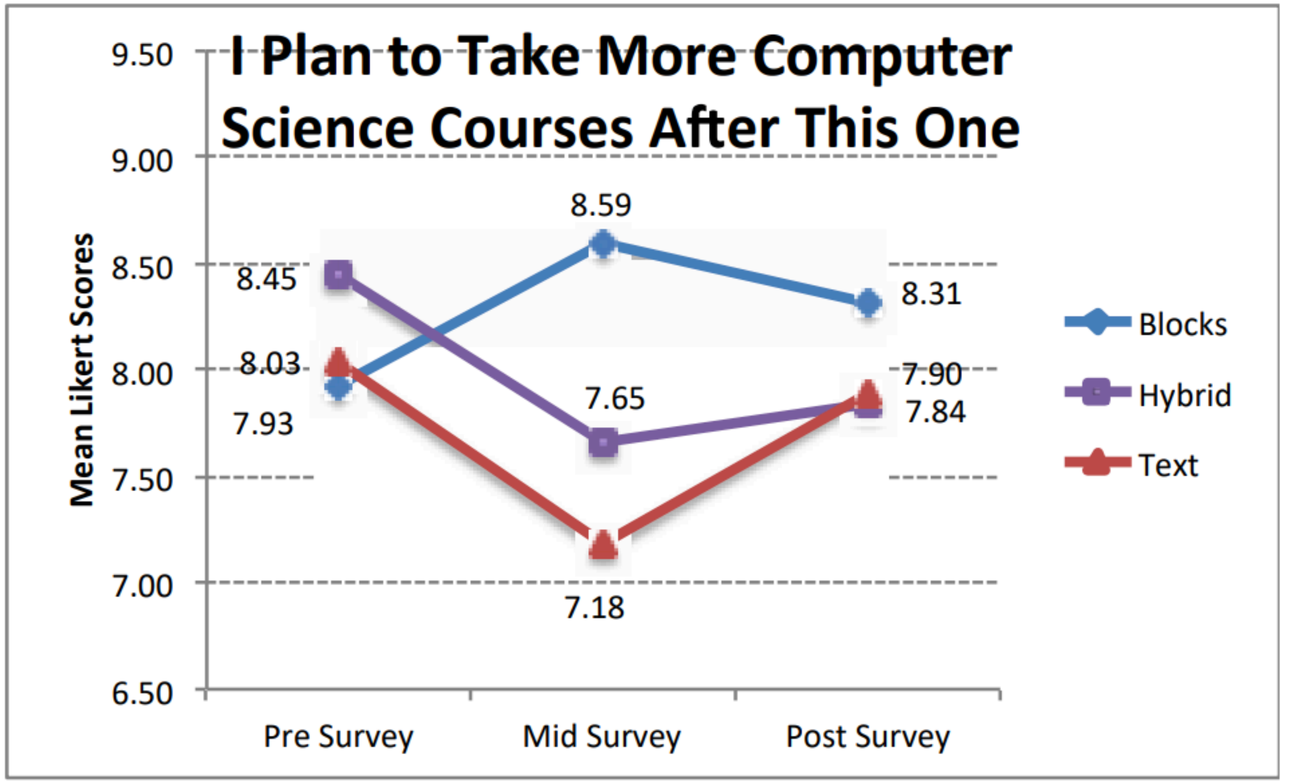


Figure 5. Average responses to the Likert statement: I plan to take more computer science courses after this one.

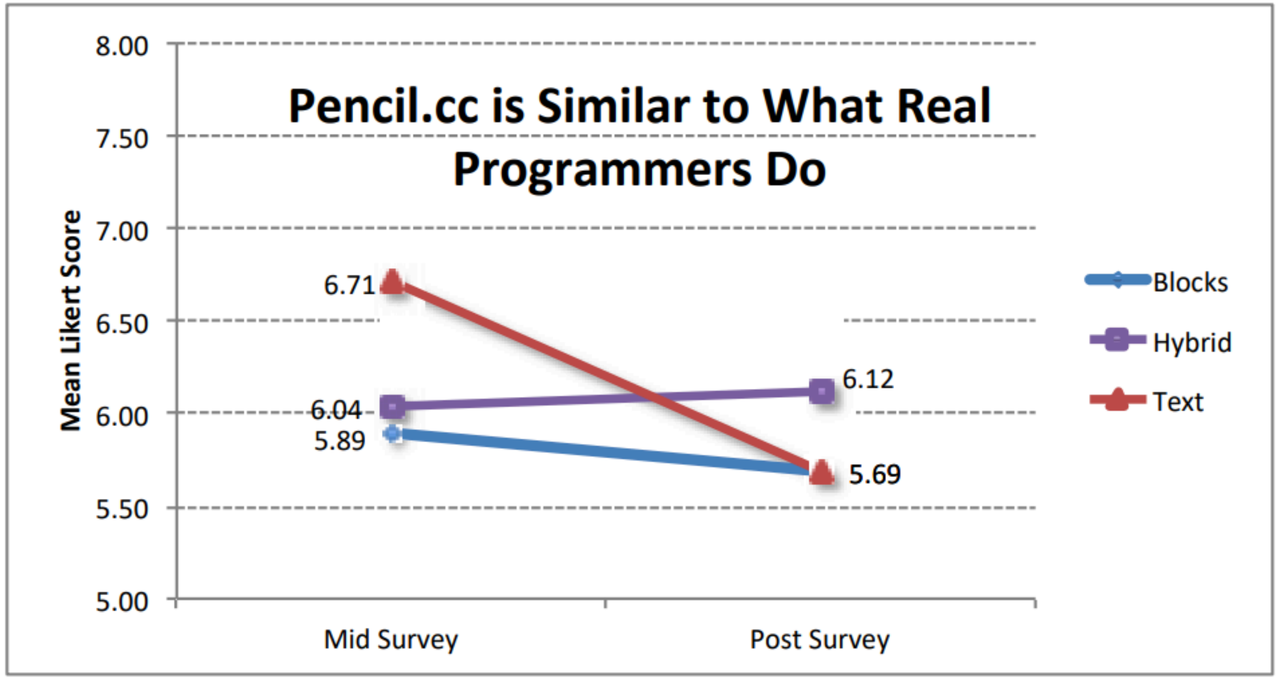


Figure 6. Student responses to the authenticity prompt: Pencil.cc is similar to what real programmers do.

From Blocks to Text and Back: Programming Patterns in a Dual-modality Environment

David Weintrop
University of Chicago
UChicago STEM Education
Chicago, IL, USA 60637
dweintrop@uchicago.edu

Nathan Holbert
Teacher's College, Columbia University
Department of Mathematics, Science, and Technology
New York City, NY, USA 10027
holbert@tc.columbia.edu

ABSTRACT

Blocks-based, graphical programming environments are increasingly becoming the way that novices are being introduced to the practice of programming and the field of computer science more broadly. An open question surrounding the use of such tools is how well they prepare learners for using more conventional text-based programming languages. In an effort to address this transition, new programming environments are providing support for both blocks-based and text-based programming. In this paper, we present findings from a study investigating how learners use a dual-modality environment where they can choose to work in either a blocks-based or text-based interface, moving between them as they choose. Our analysis investigates what modality learners choose to work in, and if and why they move from one representation to the other within a single project. We conclude with a discussion of design implications and future directions for this work. This work contributes to our understanding of the affordances of blocks-based programming environments and advances our knowledge on how best to utilize them.

CCS Concepts

Human-centered computing→Visualization • Social and professional topics→Computer science education

General Terms

provides syntactic information through the visual shape of commands and allows users to author programs by dragging-and-dropping block-shaped commands together. As more, and younger, learners are introduced to programming, the blocks-based approach is becoming the de facto standard for introductory programming environments and for early exposure to computer science (CS) more broadly.

Despite widespread use, open questions remain about the blocks-based modality and its fit in conventional CS education. More specifically, it is unclear how well such tools prepare students for future CS learning opportunities or how best to transition learners from blocks-based introductory tools to more conventional text-based languages [19]. One proposed solution involves the creation of dual-modality interfaces that allow learners to seamlessly shift back-and-forth between blocks-based and textual representations [3, 7, 12, 16]. In addition to allowing the user to decide what modality to work in, such tools also provide an opportunity for learners to see each representation of code "side-by-side," which can highlight structural similarities as well as syntactic differences [22]. While recent work has offered insight into perceived supports offered by blocks-based environments, and in the ways learners transition from blocks to text, less is known about the particular conceptual resources mobilized by each representation. In other words, when novices have a choice between blocks and text, which modality do they choose? Why? And how does this

HSC: High School Condition
GC: Graduate Condition (enrolled in a graduate level course on the design of educational learning environments (mean age of 29))

novices learning to program. During the course of the studies, students overwhelming used the blocks-based modality for the programming assignments. Participants in the HSC used the block modality 92% of the time, while the GC participants used the block modality 91% of the time. This suggests, at least at a high level, that the blocks-based modality is not more developmentally appropriate for one age over the other. However, the distribution of time spent in the two modalities was not uniform across the student population. Instead, some students worked almost exclusively in blocks, while other preferred text, and a third group moved between the two. In other word, the choice of modality is not driven by age, but instead, by some other factor.

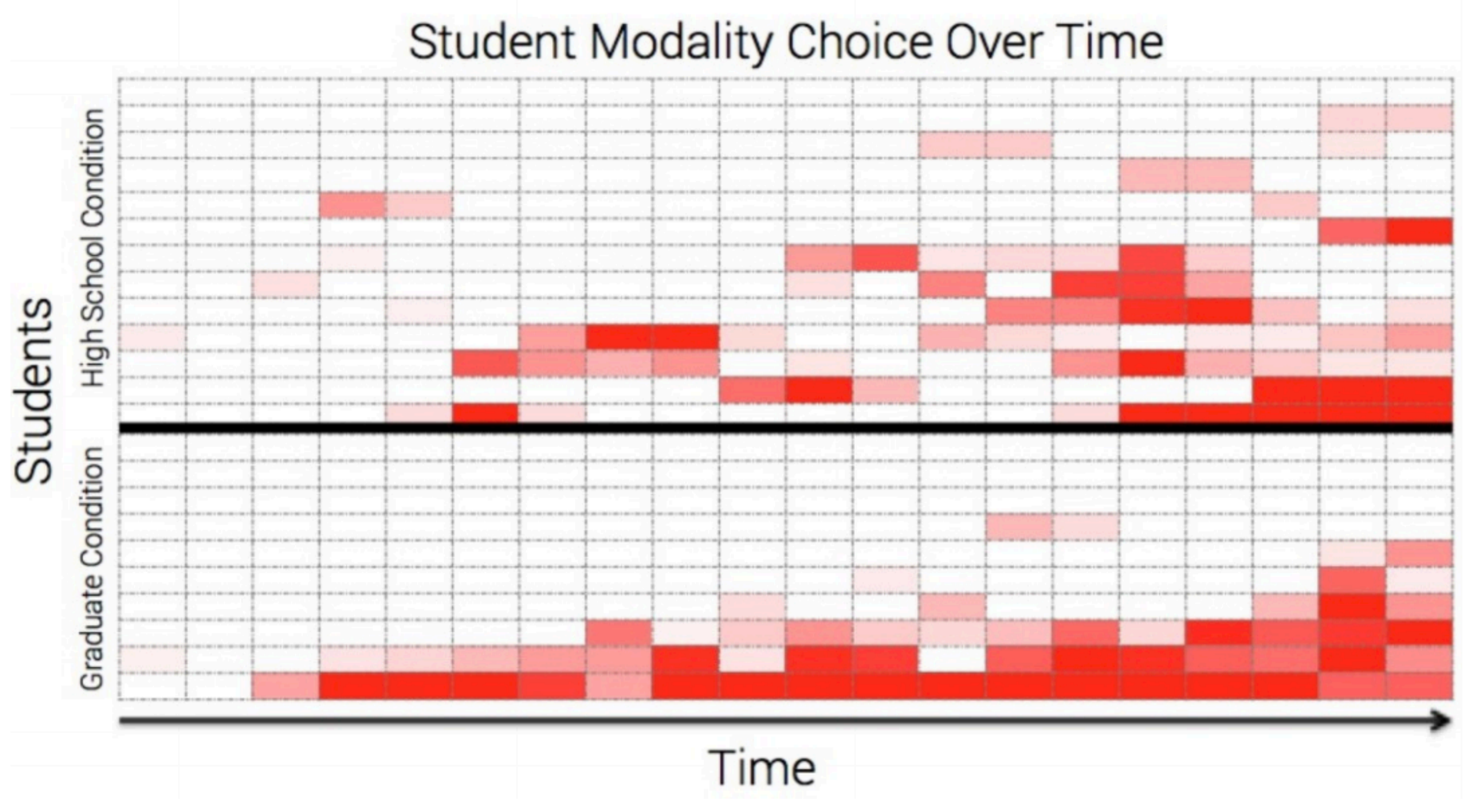
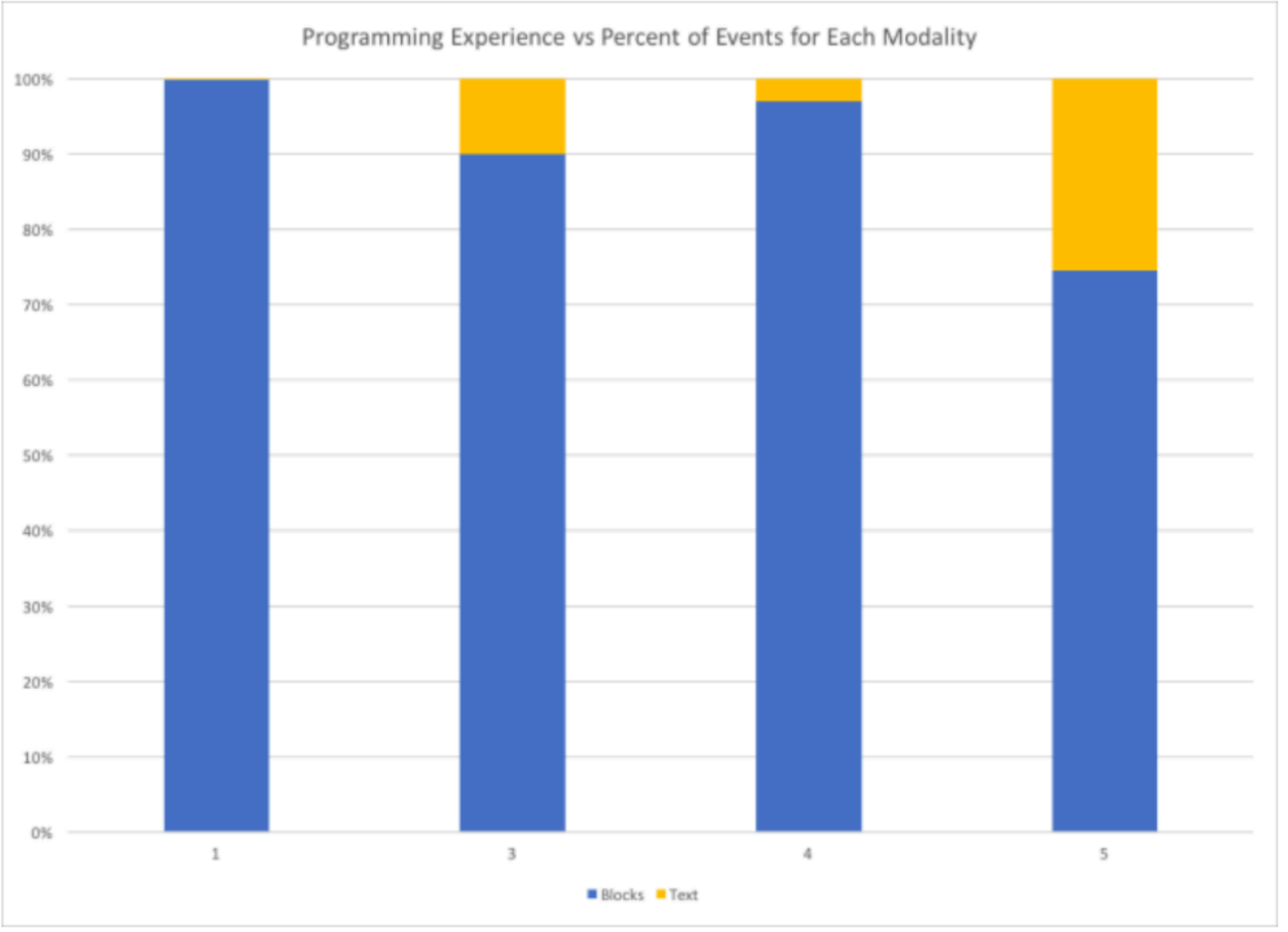


Figure 3. Student modality choice over time – the darker the square, the more time spent in the text interface.


Programming Experience vs Percent of Events for Each Modality
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
1
3
4
5
Blocks
Text

NORTHWESTERN UNIVERSITY

Modality Matters: Understanding the Effects of Programming Language Representation in High School Computer Science Classrooms

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

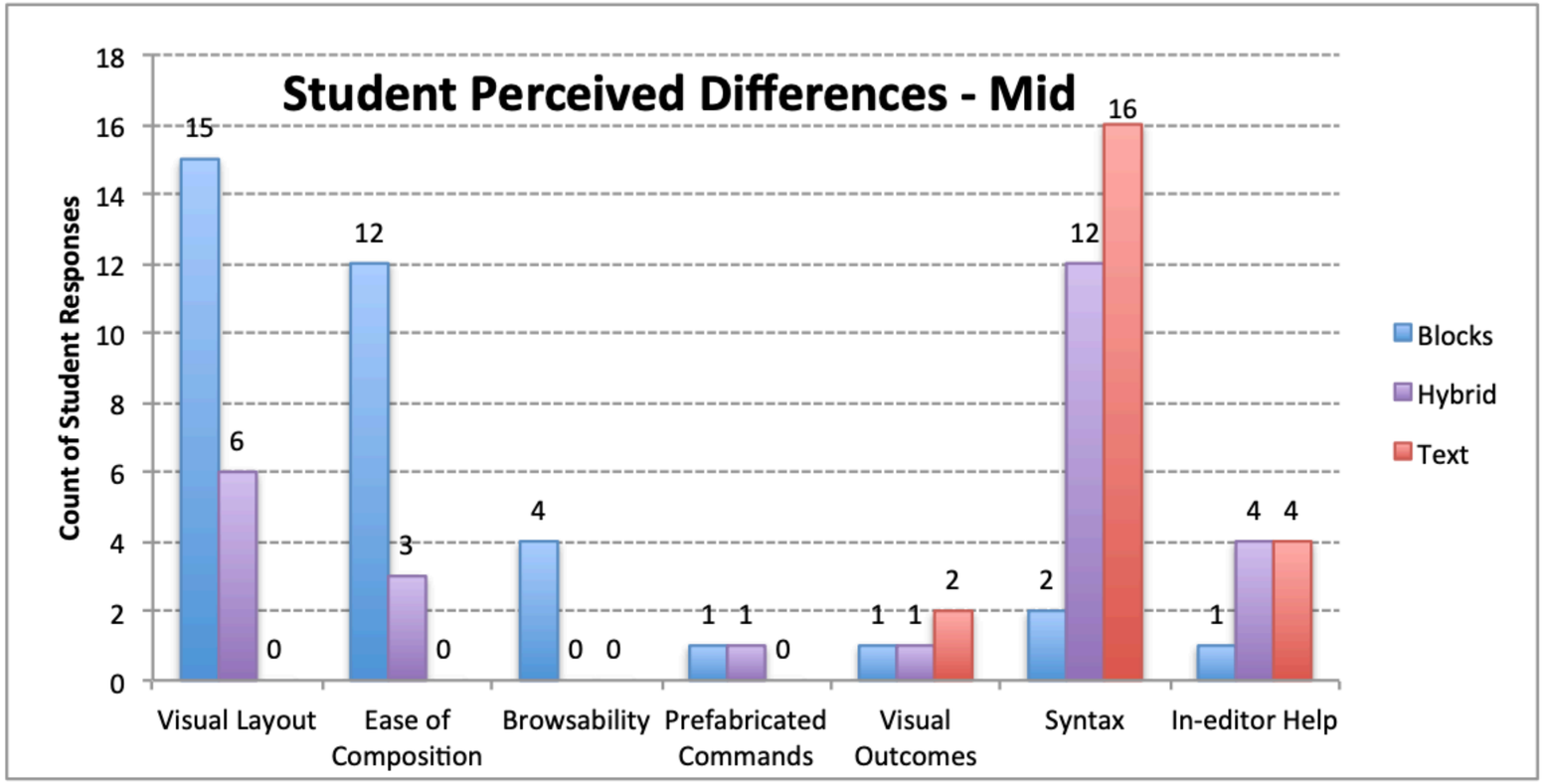


Figure 5.1. Student reported differences between Pencil.cc and Java at the midpoint of the study.

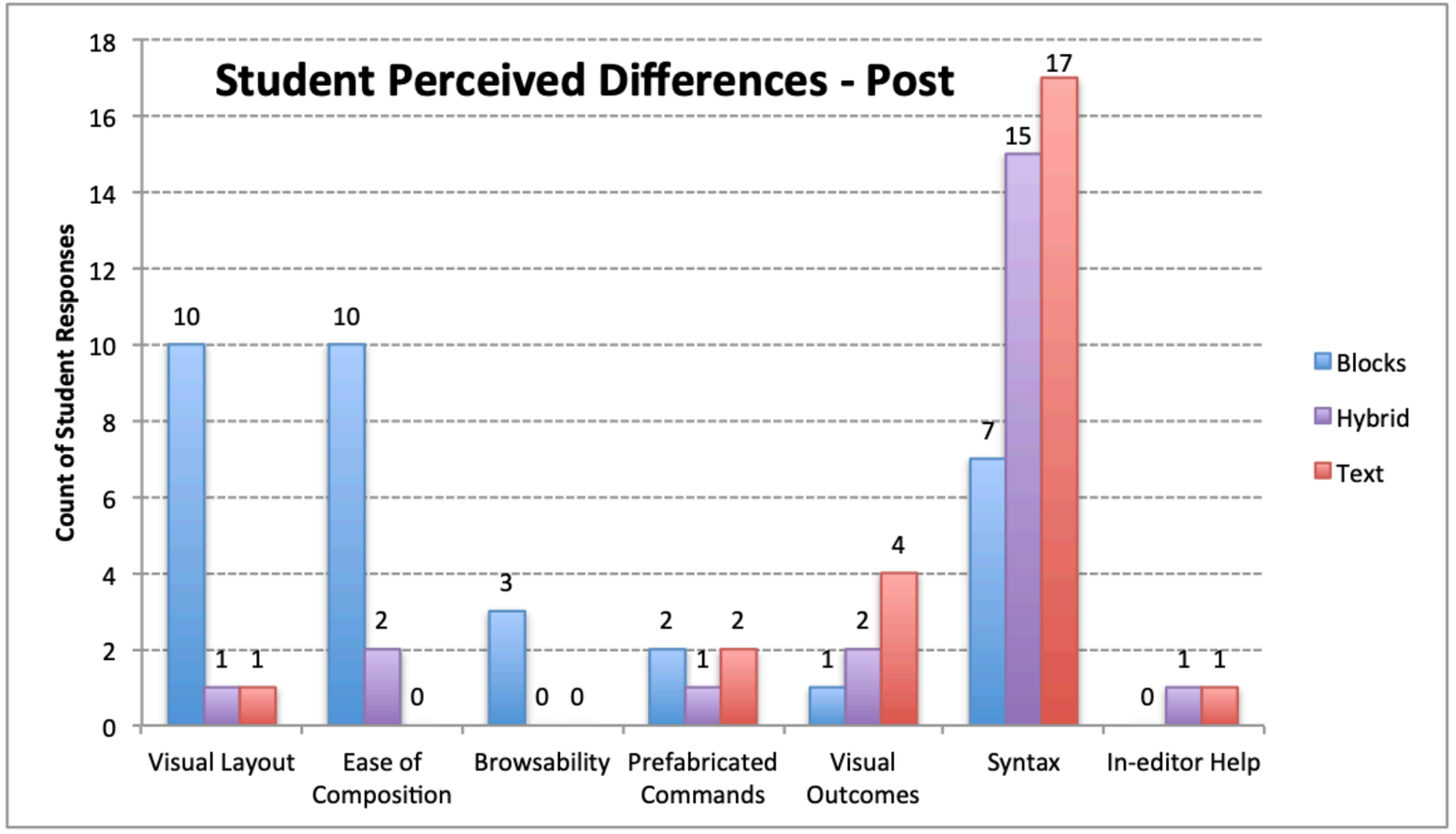


Figure 5.2. Student reported differences between Pencil.cc and Java at the conclusion of the study.

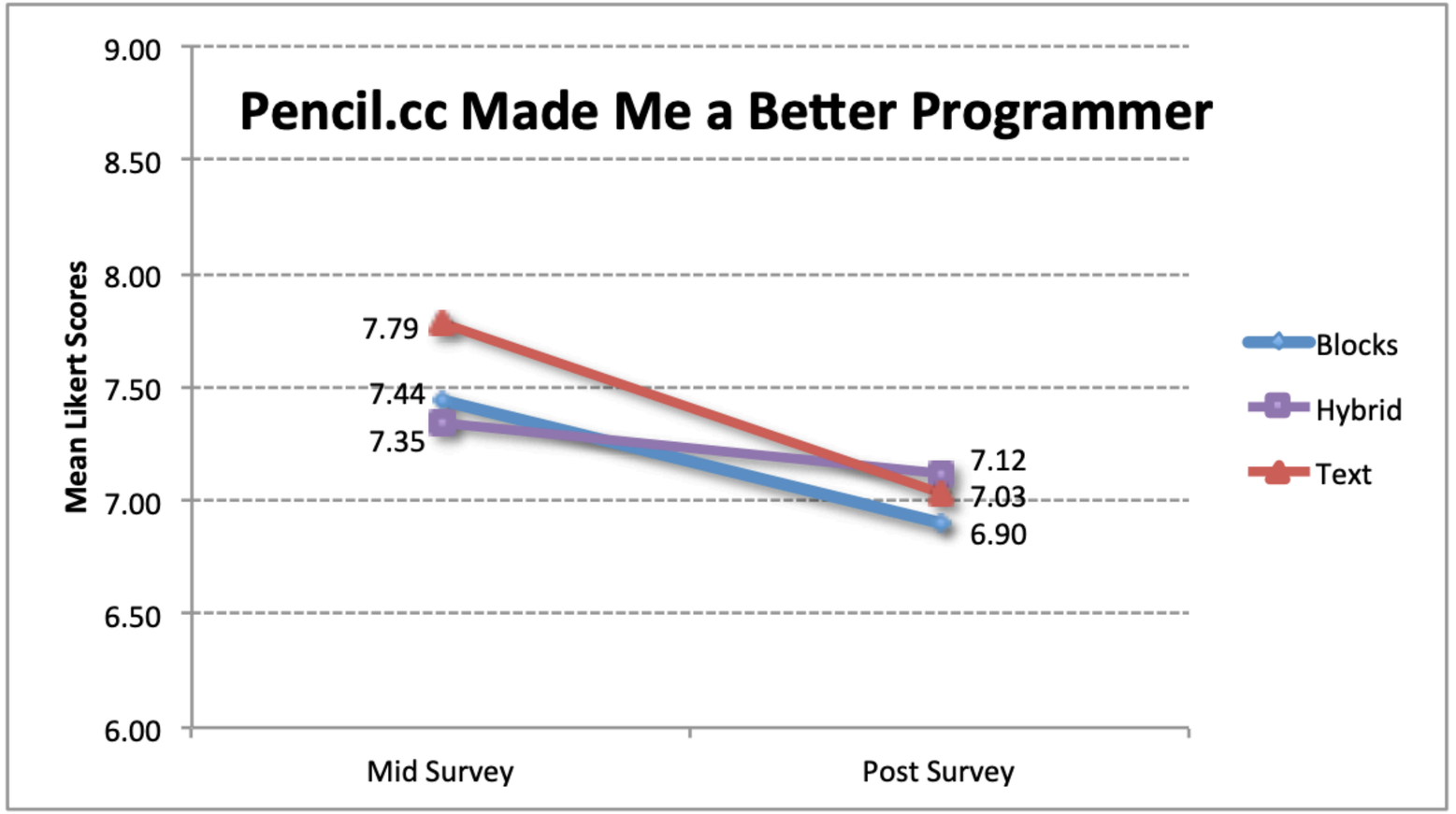


Figure 5.4. Student responses to the prompt: Pencil.cc made me a better programmer.

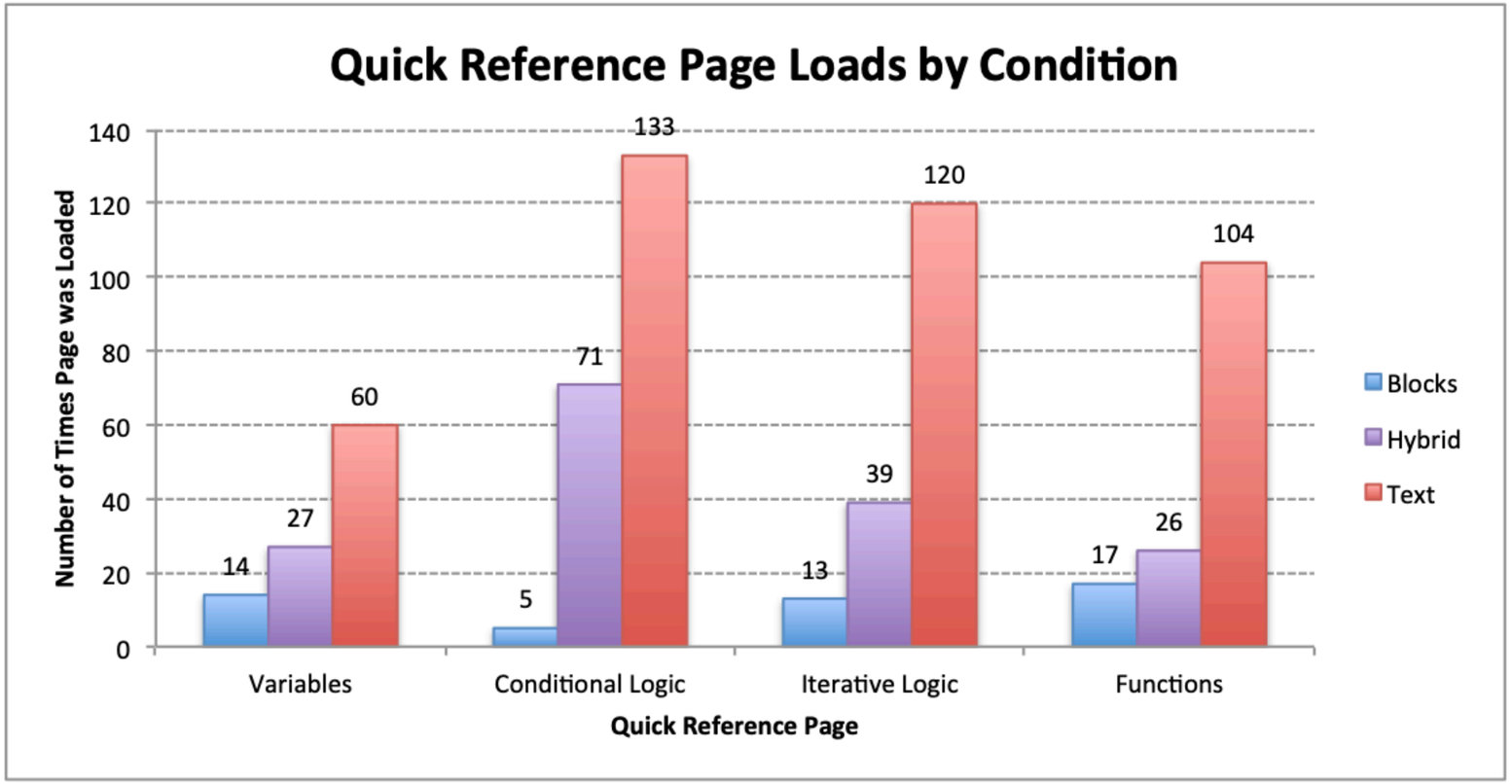


Figure 7.11. The number of times the Quick Reference pages were loaded for the four concepts covered in the introductory curriculum, grouped by Condition.

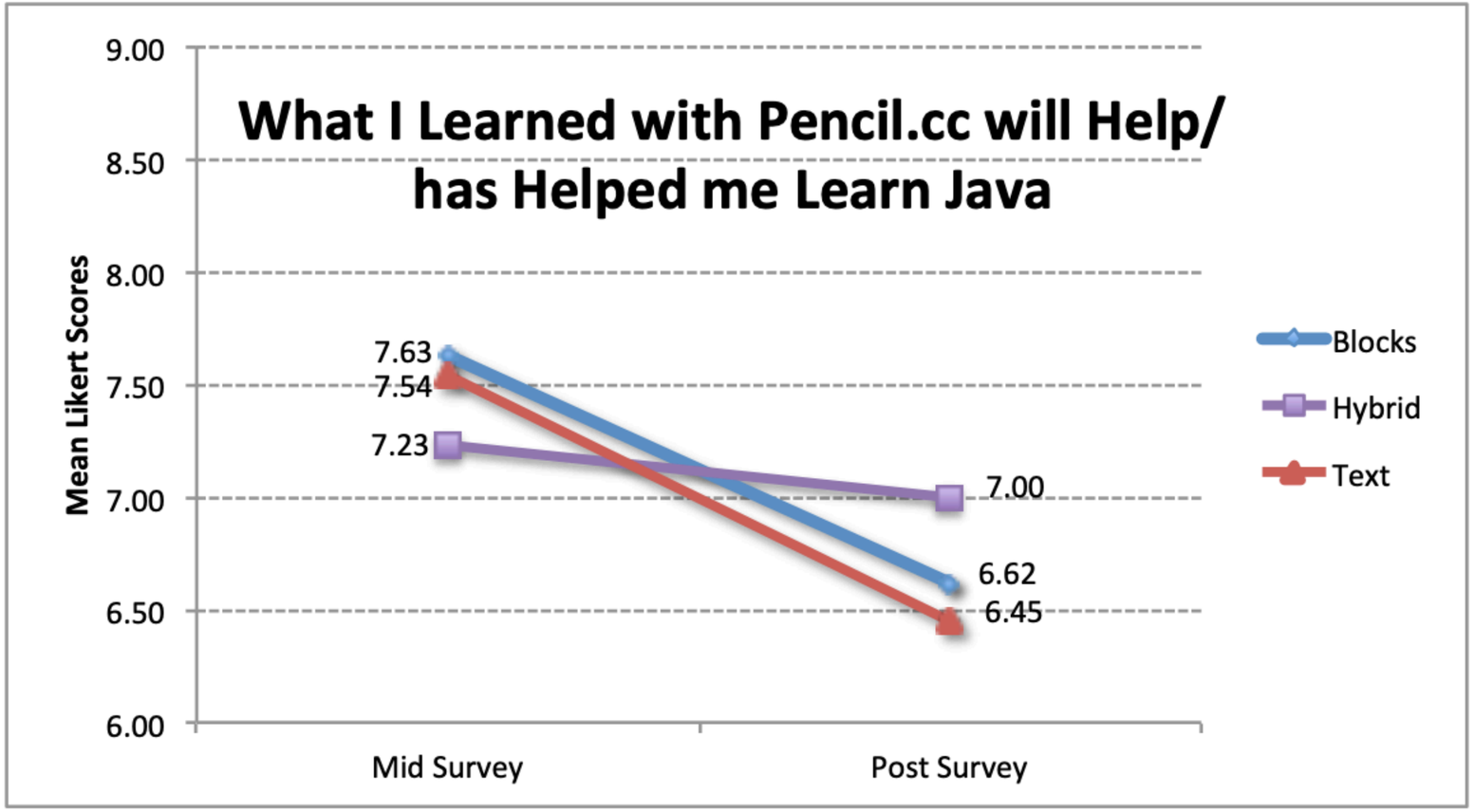
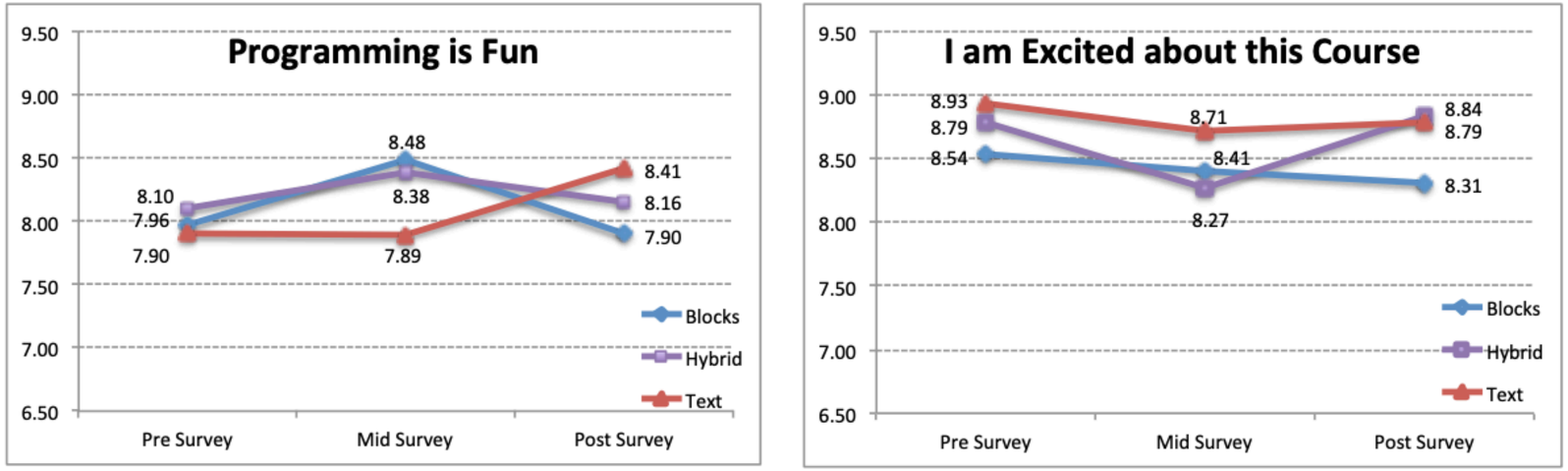


Figure 8.1. Student responses to whether or not they thought their time spent working in Pencil.cc was helpful for learning Java.



(a) (b)

Figure 8.6. Average student responses to the Likert prompt Programming is Fun (a) and I am Excited about this Course (b) grouped by condition.

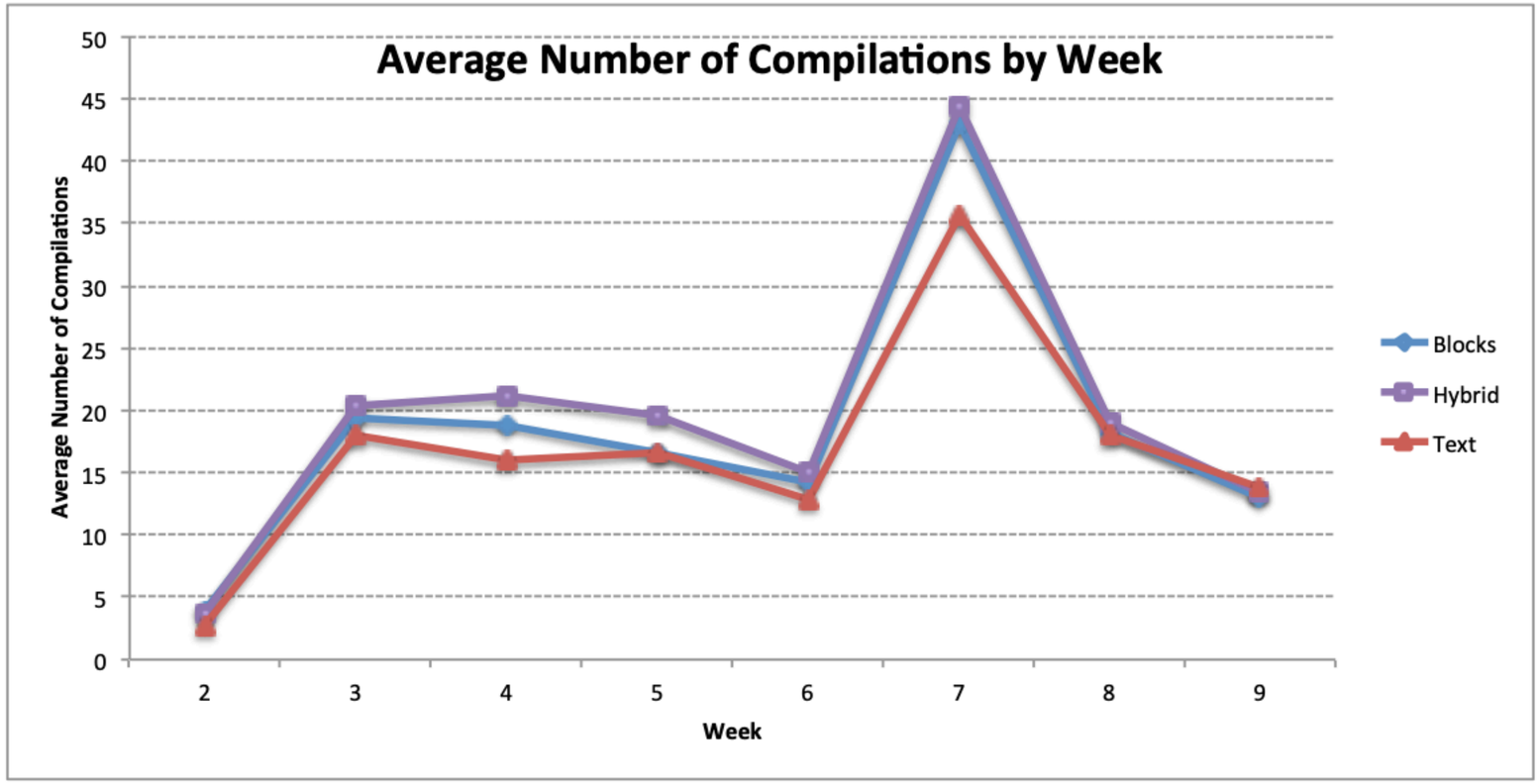


Figure 8.9. The average number of compilations of Java programs by student by week.

Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs

David Weintrop
Northwestern University
2120 Campus Drive, Suite 332
Evanston, Illinois 60628
dweintrop@u.northwestern.edu

Uri Wilensky
Northwestern University
2120 Campus Drive, Suite 337
Evanston, Illinois 60628
uri@northwestern.edu

ABSTRACT

Blocks-based programming environments are becoming increasingly common in introductory programming courses, but to date, little comparative work has been done to understand if and how this approach affects students' emerging understanding of fundamental programming concepts. In an effort to understand how tools like Scratch and Blockly differ from more conventional text-based introductory programming languages with respect to conceptual understanding, we developed a set of "commutative" assessments. Each multiple-choice question on the assessment includes a short program that can be displayed in either a blocks-based or text-based form. The set of potential answers for each question includes the correct answer along with choices informed by prior research on novice programming misconceptions. In this paper we introduce the Commutative Assessment, discuss the theoretical and practical motivations for the assessment, and present findings from a study that used the assessment. The study had 90 high school students take the assessment at three points over the course of the first ten weeks of an introduction to

qualifiers describing under what conditions a given language is the best choice. These so called 'language wars' have been raging for as long as computer science has been taught, with little in the way of consensus emerging and with potentially detrimental effects [58]. Much work has been done attempting to empirically answer the question of which text-based language is best for novices, or at least identify features that make a language more or less accessible to beginners. While there is much to show for this effort, an alternative to conventional text-based languages is emerging in novice programming classrooms that brings a new dimension to introductory tools. Graphical blocks-based programming tools like Scratch [49], Blockly [23], and Alice [13] are becoming commonplace in introductory programming contexts, with a growing number of new curricula utilizing blocks-based programming tools in their materials, including the CS Principles project, the Exploring Computer Science program, and the materials being developed by code.org. The introduction of blocks-based programming environments changes the landscape of introductory tools, replacing questions of syntactic features of textual languages with the larger question of if text-

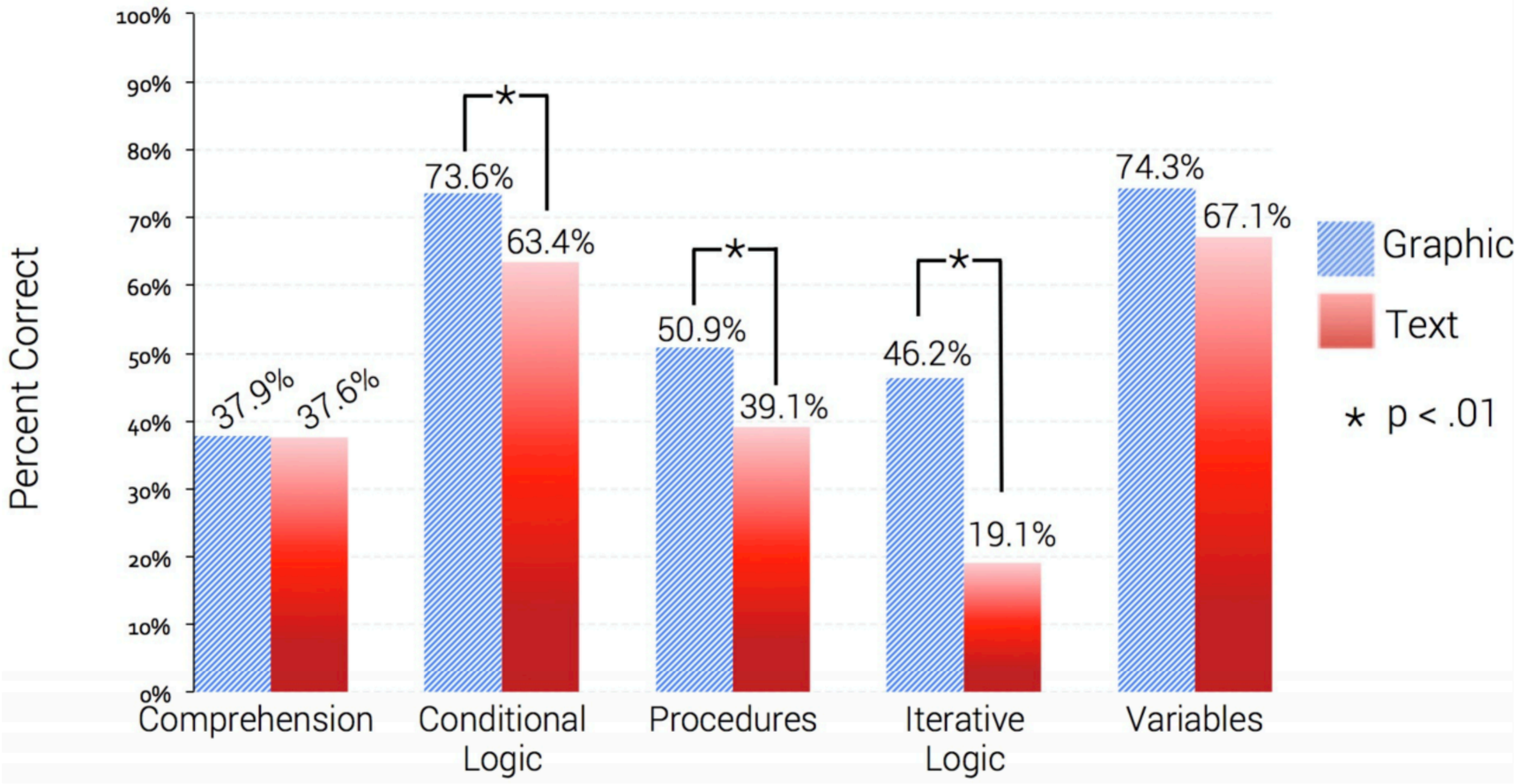


Figure 4. Student performance on the Commutative Assessment grouped by modality and concept.

Comparing Textual and Block Interfaces in a Novice Programming Environment

Thomas W. Price
North Carolina State University
890 Oval Drive
Raleigh, NC
twprice@ncsu.edu

Tiffany Barnes
North Carolina State University
890 Oval Drive
Raleigh, NC
tmbarnes@ncsu.edu

ABSTRACT

Visual, block-based programming environments present an alternative way of teaching programming to novices and have proven successful in classrooms and informal learning settings. However, few studies have been able to attribute this success to specific features of the environment. In this study, we isolate the most fundamental feature of these environments, the block interface, and compare it directly to its textual counterpart. We present analysis from a study of two groups of novice programmers, one assigned to each interface, as they completed a simple programming activity. We found that while the interface did not seem to affect users' attitudes or perceived difficulty, students using the block interface spent less time off task and completed more of the activity's goals in less time.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.1.7 [**Programming Techniques**]: Visual Programming

been evaluated in classrooms [24, 25, 26], summer camps [21, 31] and after-school programs [22].

In this paper, we will use the term Block-Based Programming Environment (BBPE) to refer to those environments that allow users to construct and execute computer programs by composing atomic blocks of code together to produce program structure. These code blocks may additionally have slots, which can be filled by other blocks; for example, a function call block may have slots for each of its parameters. These blocks may represent high-level structures, such as methods or loops, or low-level operators such as multiplication or equality comparison. An example is shown in Figure 1. There exist a variety of programming environments which use the block metaphor, but here we limit our use of the term BBPE to those that use *procedural* languages. For a more thorough introduction to one BBPE, see [29].

Much work has gone into the evaluation of BBPEs. Previous studies have identified what programming concepts students use in BBPEs [22], measured learning gains from classes based on BBPEs [24, 26], and investigated the ease of transitioning from these environments to textual program-

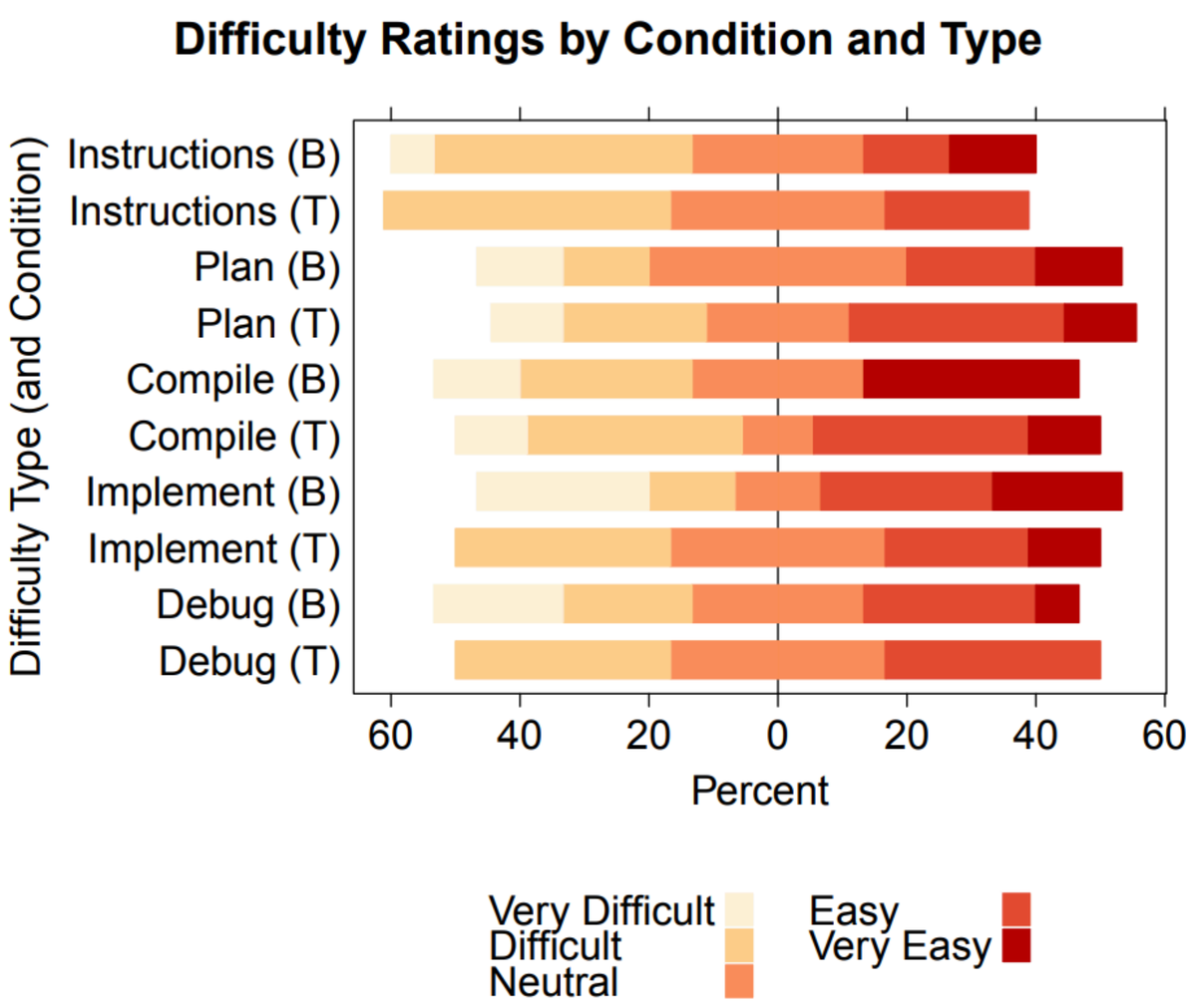


Figure 4: The distribution of difficulty ratings given by students in each condition. The questions, in order, asked users to rate their difficulty understanding instructions, deciding what to do, getting the program to run (compile), implementing a solution and figuring out what went wrong (debugging).

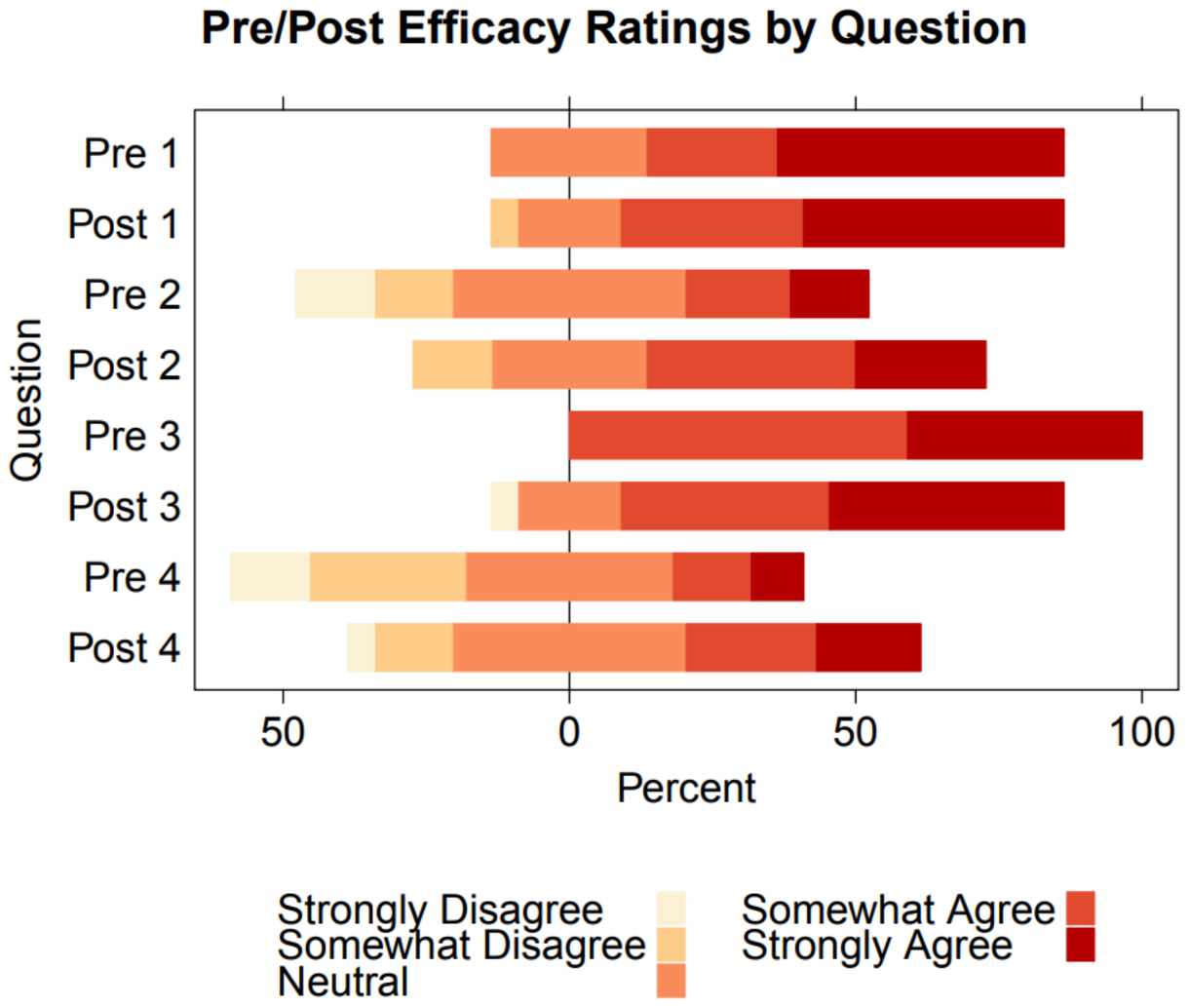


Figure 6: The distribution of ratings given by students in the pre- and post-survey for each Efficacy item (see Figure 1 for the items' text).

Value	Block	Text	p	d
Total	2273.9 (596.4)	2208.0 (427.1)	0.851	–
Idle	407.2 (238.9)	793.5 (368.3)	0.002	1.27
Active	1866.8 (617.4)	1414.5 (463.1)	0.014	0.82

Table 3: Average total, idle and active time in seconds for both groups (with standard deviations). The differences in idle and active time are significant, and Cohen's d is given.

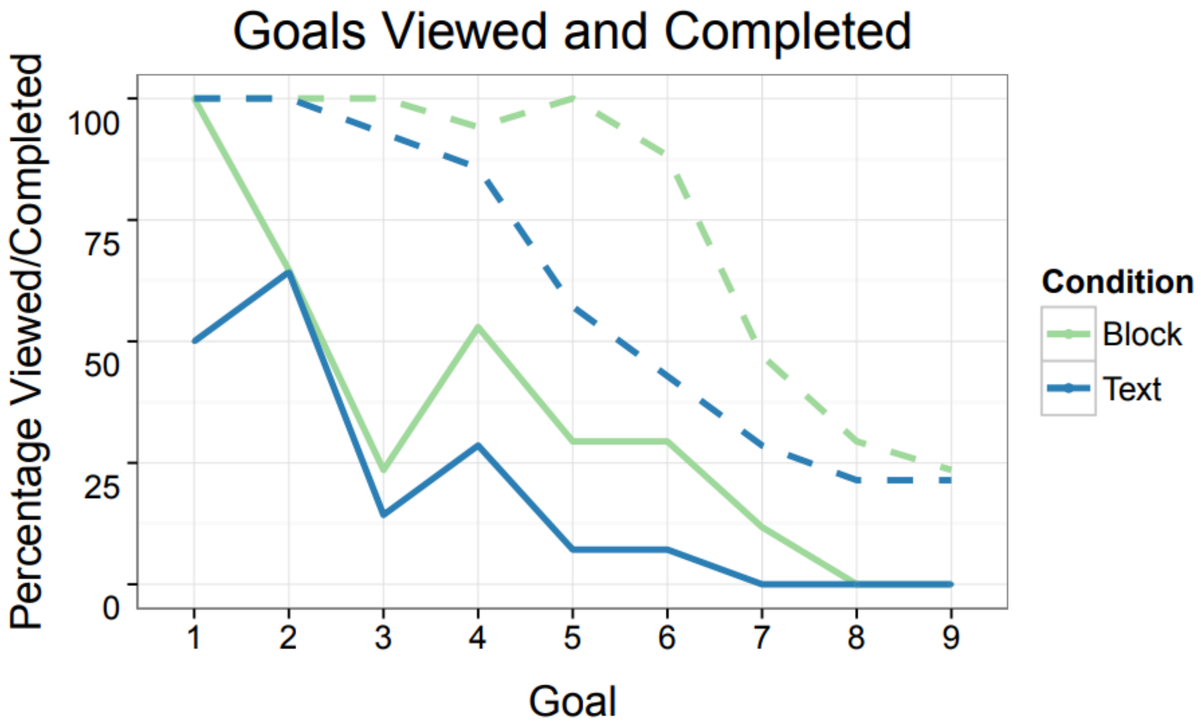


Figure 7: For each condition, the solid line shows the percentage of students who completed each goal. The dashed line shows the percentage of students who viewed each goal for at least 10 seconds.

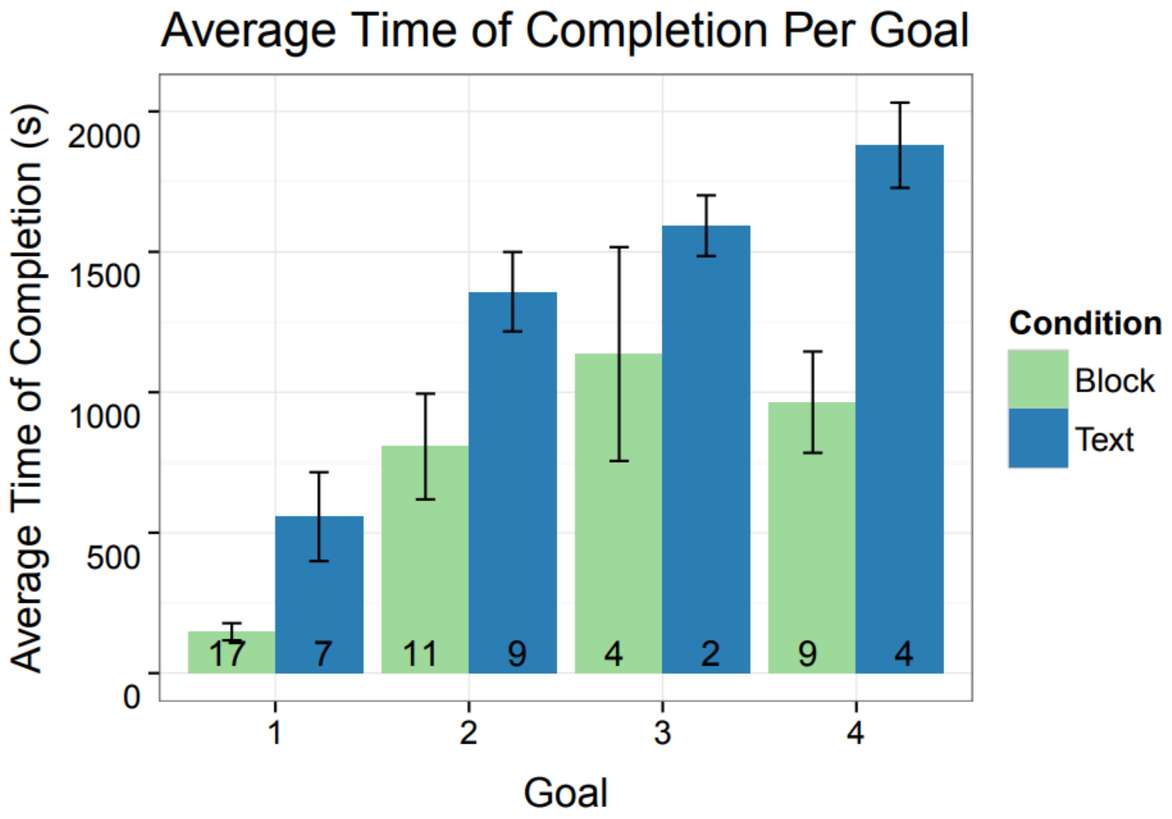


Figure 8: The average total time spent in the activity before completing each goal, with bars indicating standard error. The numbers at the bottom of each bar indicate the number of students who completed the goal. Values are not strictly increasing, in part because goals could be completed out of order. Goals 5-9 are not shown, as at most 1 student from the Text group completed them.

Empirical Comparison of Visual to Hybrid Formula Manipulation in Educational Programming Languages for Teenagers

Roxane Koitz
Graz University of Technology, Graz, Austria
rkoitz@ist.tugraz.at

Wolfgang Slany
Graz University of Technology, Graz, Austria
wolfgang.slany@tugraz.at

Abstract

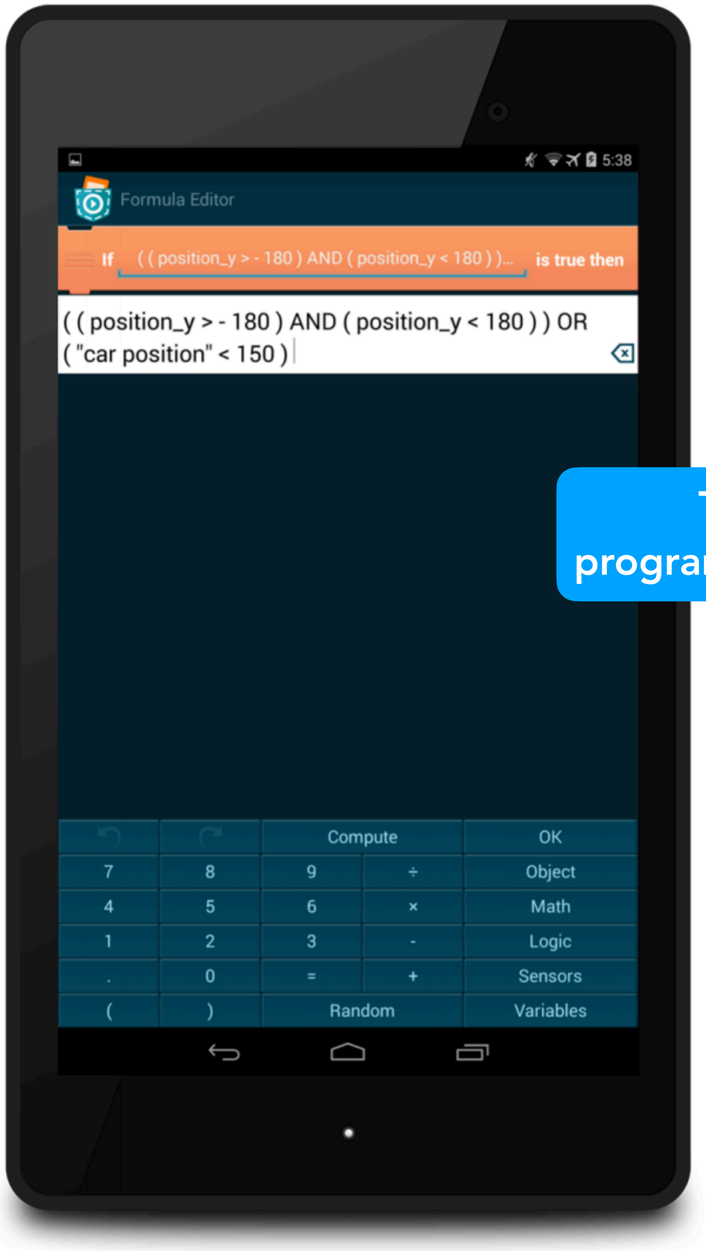
Visual programming environments hold great potential for end-user programming, as they, e.g., aim at diminishing the syntactical burden and enabling a focus on the semantic aspects of coding. Hence, graphical approaches have gained attention in the context of K-12 computer science education. Scratch, as being the prime example, is a visual educational language, where even formulas are composed utilizing Lego-style blocks. However, graphical creation and manipulation of complex and nested formulas can become overly cumbersome. Thus, we propose a hybrid approach employing visual creation and textual representation of formulas. In order to evaluate the method, a usability study has been conducted, comparing Scratch to our mobile programming

1. Introduction

"Computational Thinking" refers to the thought process of formulating and solving problems that involve abstraction, algorithmic thinking, the application of mathematical concepts, and the comprehension of problems of scale. Those fundamental skills are, however, not only relevant for computer scientists, but should be part of every child's analytical ability [22]. Several initiatives and organizations have been launched such as "code.org"[1] or "made with code"[2] with the specific goal to expose more children and adolescents to programming. The idea is to spark interest in computer science early on and make them not only consumers of digital content, but rather creators.

Visual Programming Languages (VPL) have been

(a) Scripts View



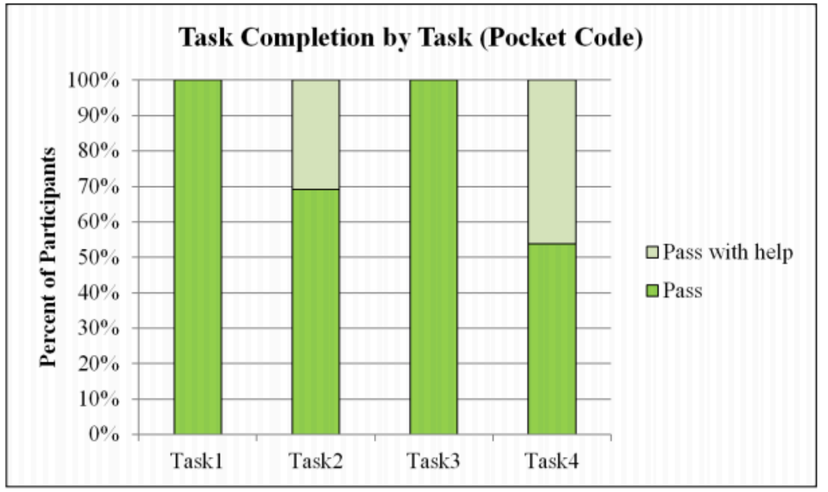
(b) Formula Editor View

This one's about programming on your phone!

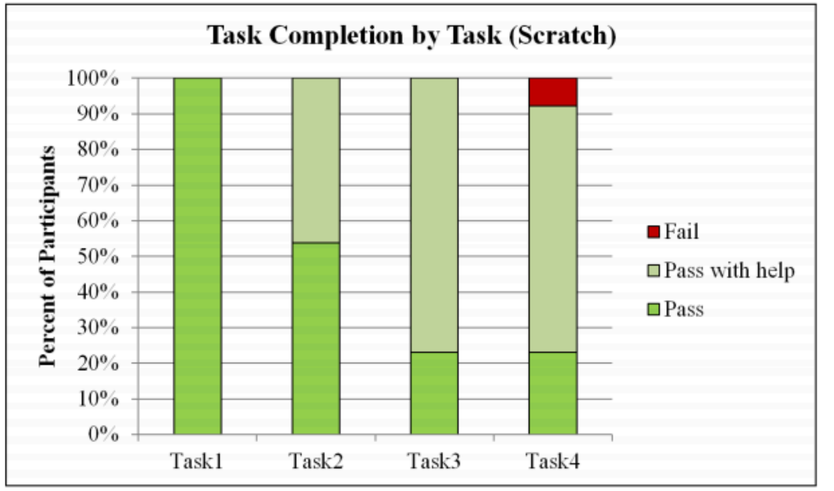
Figure 4: Pocket Code

Hybrid

Structure Only

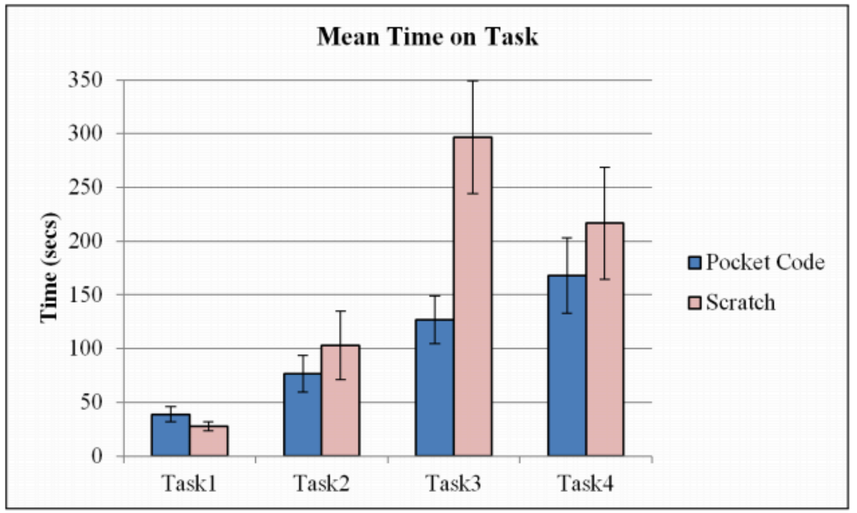


(a) Pocket Code Task Completion

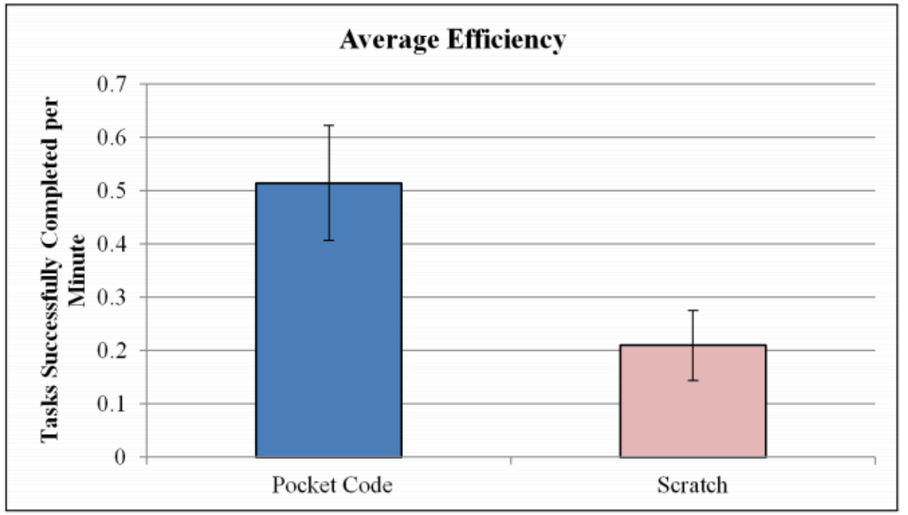


(b) Scratch Task Completion

Figure 6: Task Completion Rate by Application and Task

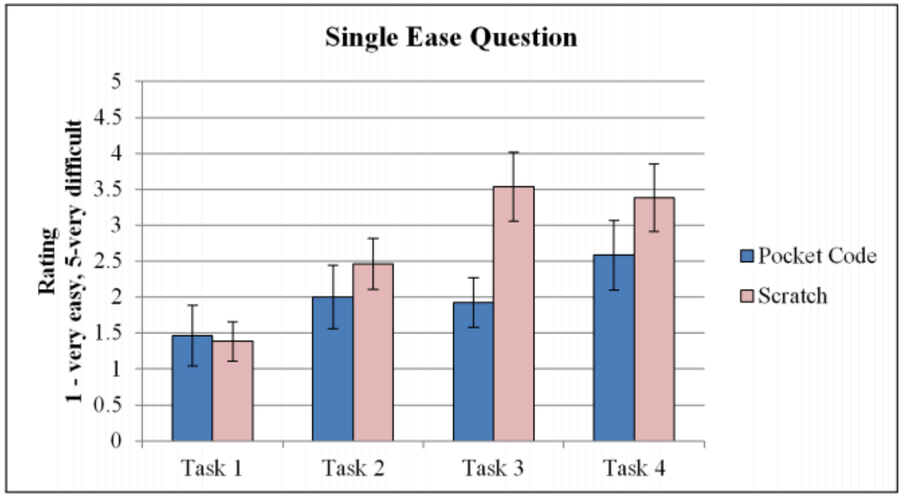


(a) Mean Time on Task

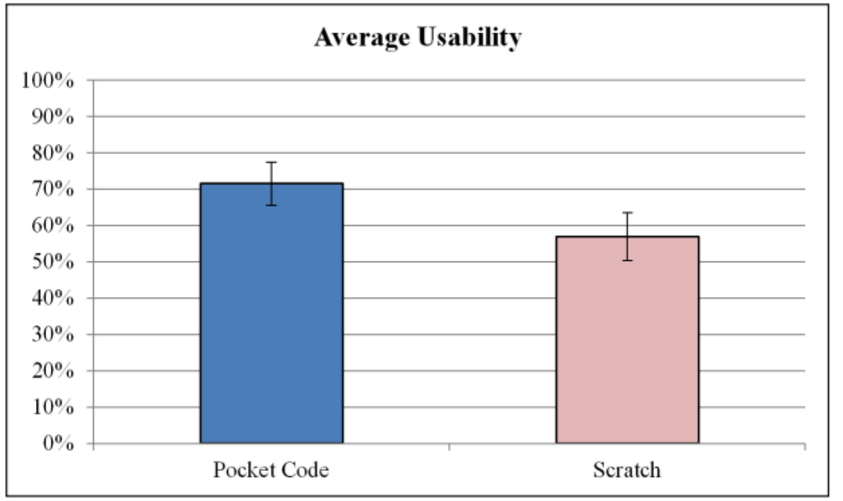


(b) Average Efficiency

Figure 7: Efficiency Results
(Error bars represent the 95% confidence interval)



(a) Singe Ease Question (1-very easy to 5-very difficult)



(b) Singe Usability Score as an Average Percentage of Time on Task, Task Completion, and SUS-Rating

Figure 8: SEQ Scores and Single Usability Score
(Error bars represent the 95% confidence interval)

End-User Experiences of Visual and Textual Programming Environments for Arduino

Tracey Booth and Simone Stumpf

City University London

{tracey.booth.2, simone.stumpf.1}@city.ac.uk

Abstract. Arduino is an open source electronics platform aimed at hobbyists, artists, and other people who want to make things but do not necessarily have a background in electronics or programming. We report the results of an exploratory empirical study that investigated the potential for a visual programming environment to provide benefits with respect to efficacy and user experience to end-user programmers of Arduino as an alternative to traditional text-based coding. We also investigated learning barriers that participants encountered in order to inform future programming environment design. Our study provides a first step in exploring end-user programming environments for open source electronics platforms.

Keywords: End-user programmers, Arduino, Visual Programming

1 Introduction

Open source hardware platforms such as Arduino [23] and Raspberry Pi [32] have reinvigorated interest in hacking and tinkering to create interactive electronics-based projects. These platforms present an opportunity for end users to move beyond being

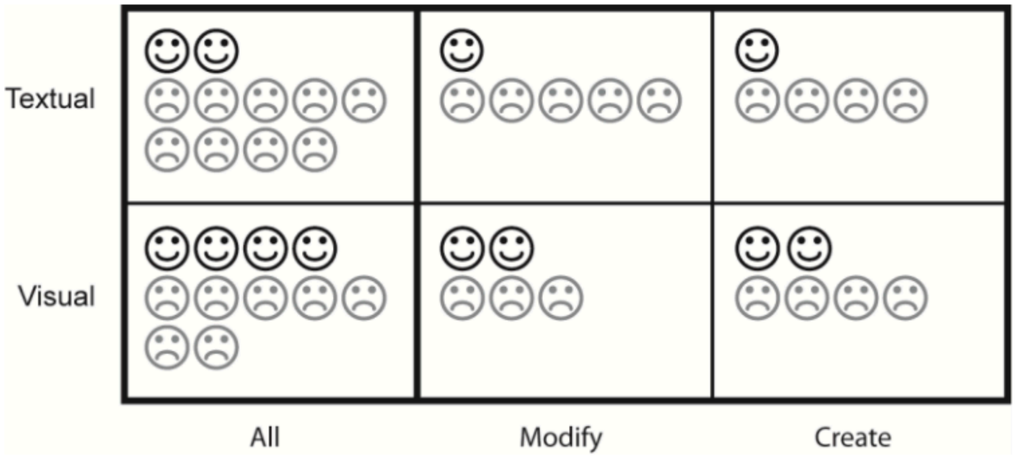


Fig. 4. Participant completion using programming environments for all tasks (left) and for create/modify task types (right)

demanding (Figure 5; a higher score indicates either higher workload or decreased performance). Although this may seem initially perplexing we found that participants carried out vastly more mouse clicks and mouse movements in the visual environment, which may explain this perceived cost.

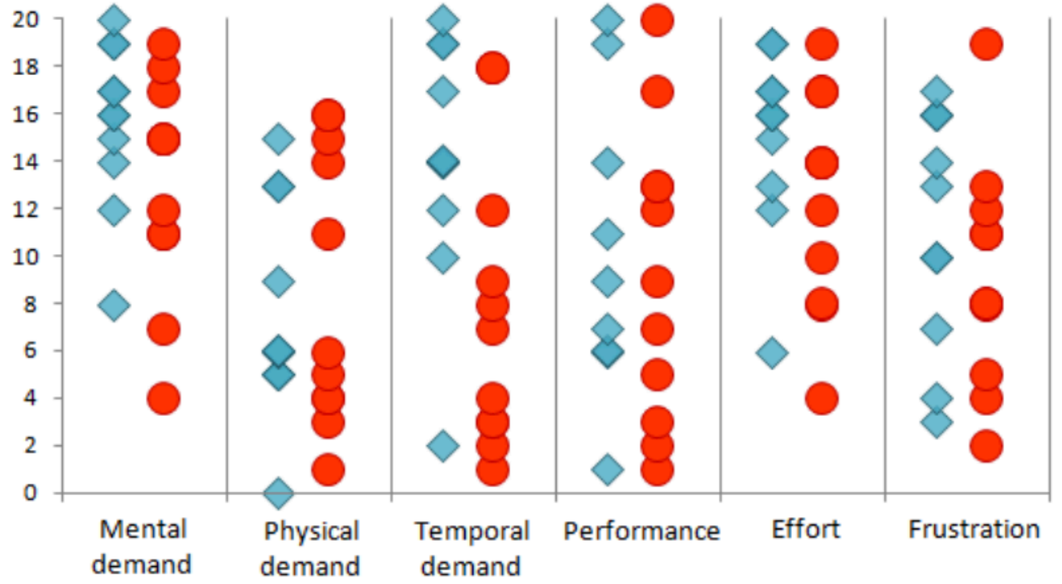


Fig. 5. Participants' TLX scores for textual (diamonds) and visual (circles) environments. The mean score was always higher for the textual environment, except relating to Physical demand.

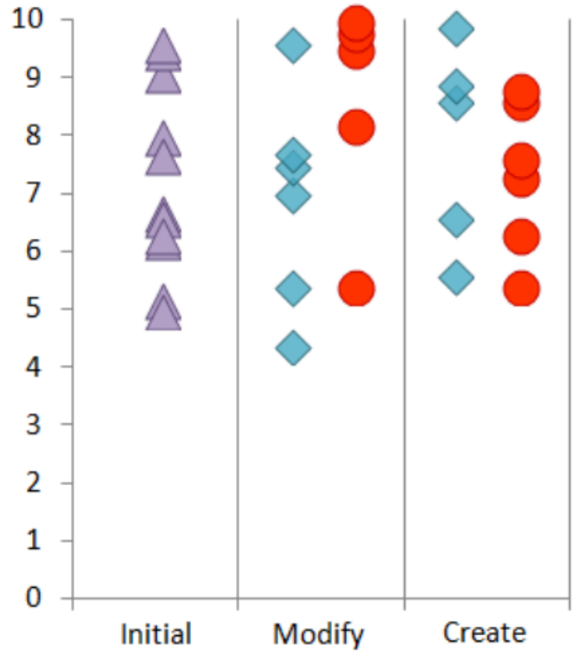


Fig. 6. Participants' self-efficacy scores initially (triangles) and after completing a task using the textual environment (diamonds) and visual environment (circles). Mean score for visual environment is higher than textual environment in the modify task but lower in the create task.

Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment

Yoshiaki Matsuzawa
Graduate School of Informatics,
Shizuoka University
3-5-1 Johoku, Nakaku, Hamamatsu
Shizuoka, Japan
matsuzawa@inf.shizuoka.ac.jp

Takashi Ohata
Graduate School of Informatics,
Shizuoka University
3-5-1 Johoku, Nakaku, Hamamatsu
Shizuoka, Japan
ohata@sakailab.info

Manabu Sugiura
Keio University
5322 Endo, Fujisawa
Kanagawa, Japan
manabu@sfc.keio.ac.jp

Sanshiro Sakai
Graduate School of Informatics,
Shizuoka University
3-5-1 Johoku, Nakaku, Hamamatsu
Shizuoka, Japan
sakai@inf.shizuoka.ac.jp

ABSTRACT

In the past decade, improvements have been made to the environments used for introductory programming education, including by the introduction of visual programming languages such as Squeak and Scratch. However, migration from these languages to text-based programming languages such as C and Java is still a problem. Hence, using the OpenBlocks framework proposed at the Massachusetts Institute of Technology, we developed a system named BlockEditor,

Categories and Subject Descriptors

D.1.7 [**Programming Techniques**]: Visual Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques; K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Design, Human Factors, Measurement

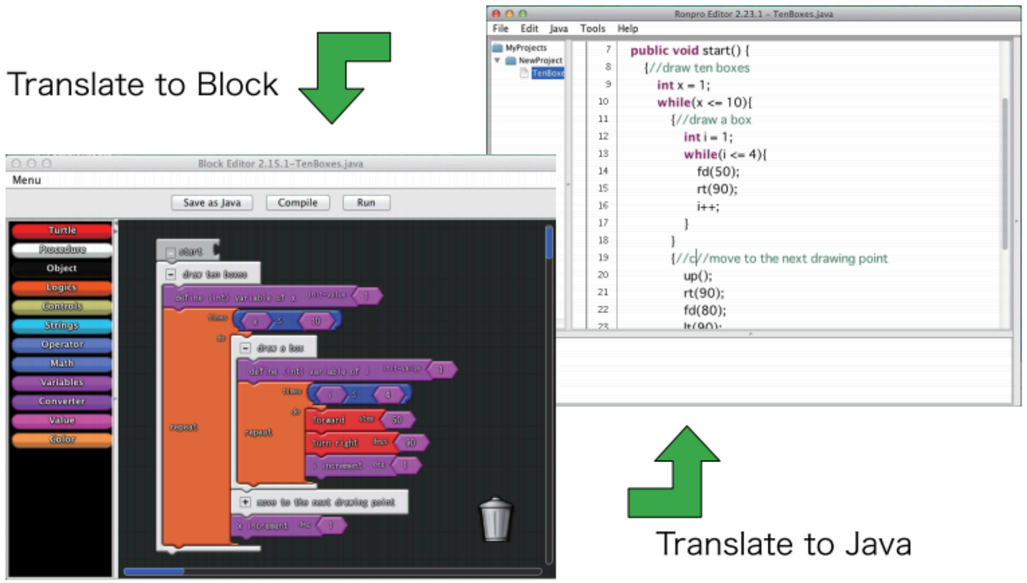


Figure 1: Overview of the proposed environment.

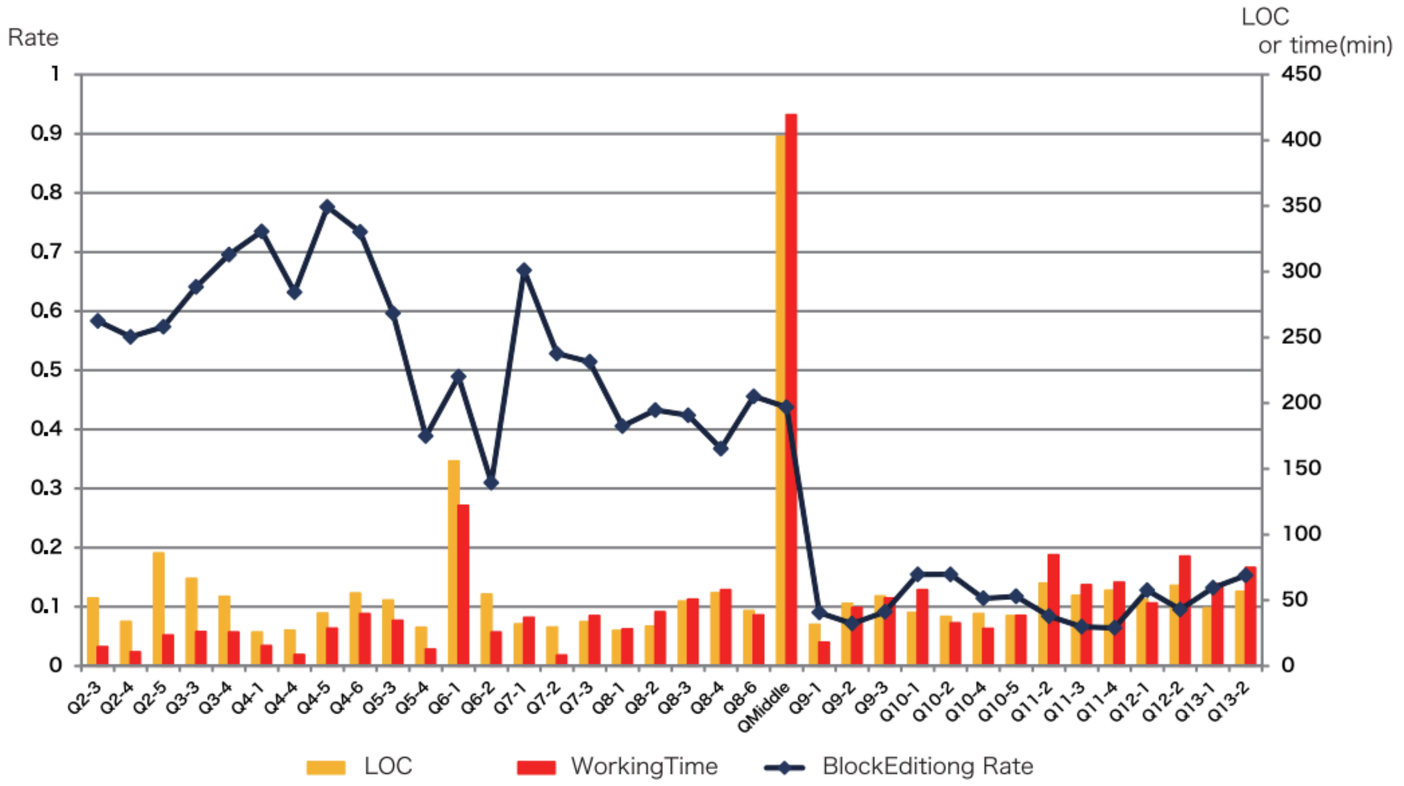


Figure 3: Relative rate of working with BlockEditor, total working time, and the lines of code (LOC) for each assignment.

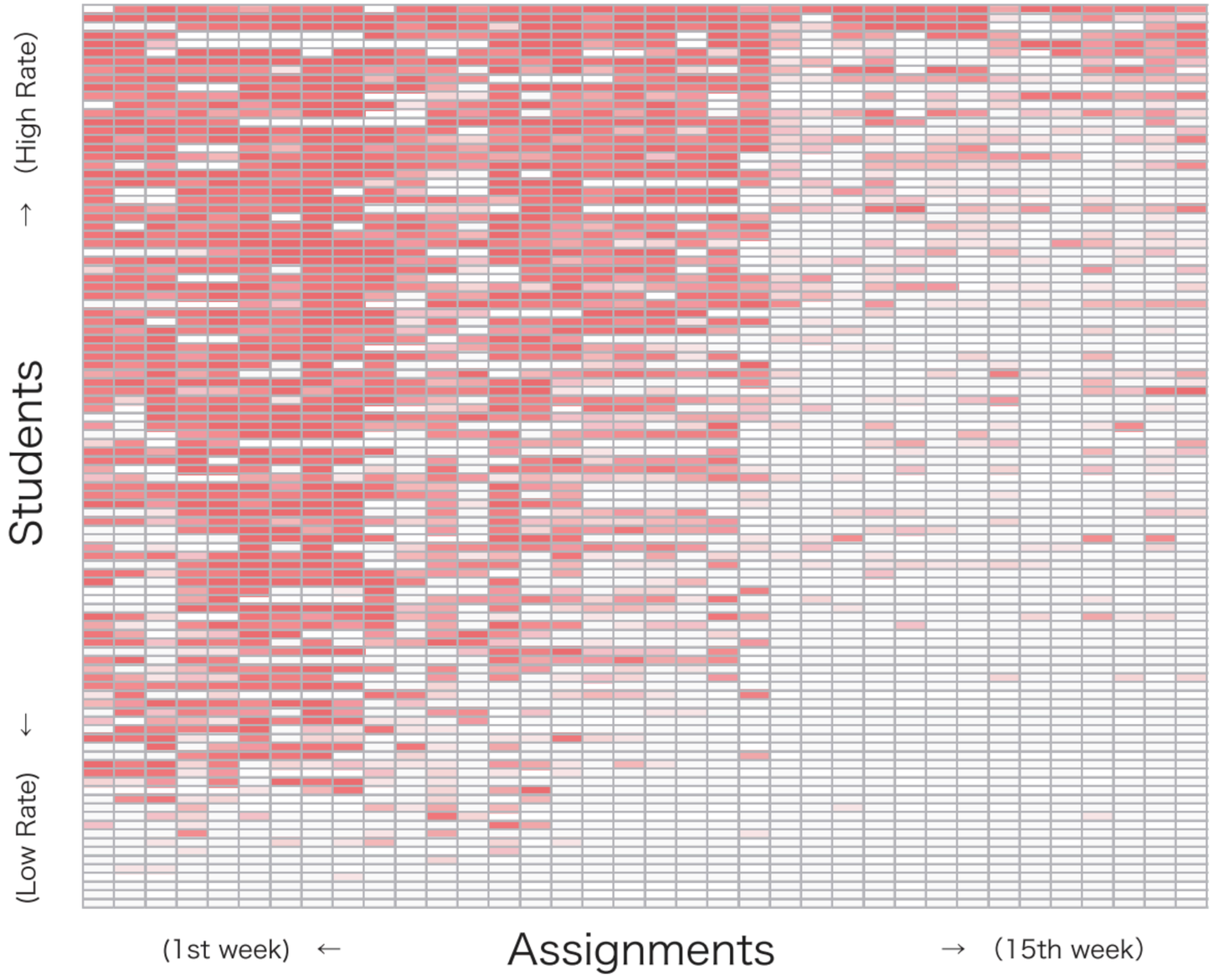
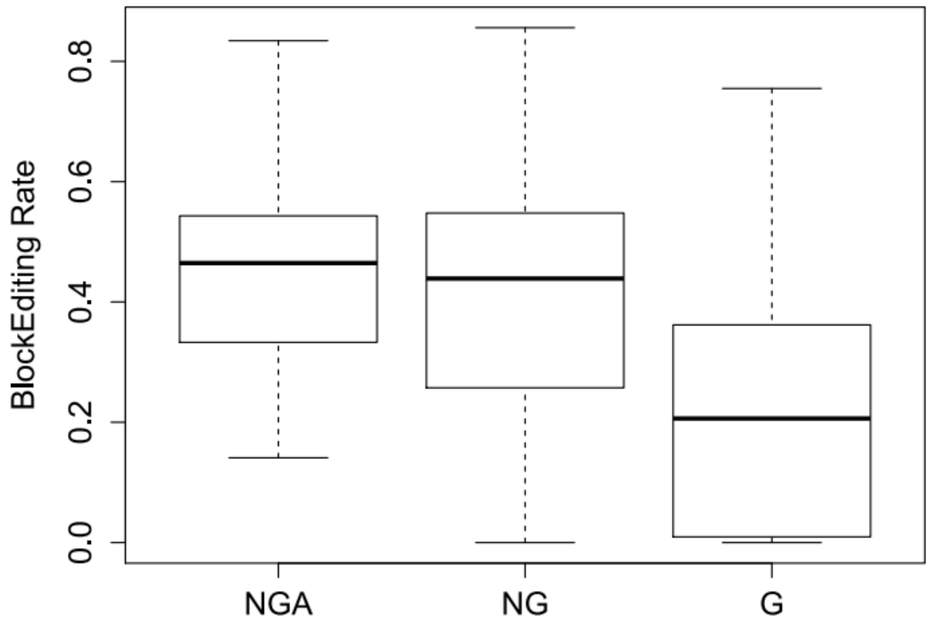


Figure 4: Grid representation of relative rate of working with BlockEditor for each student (color intensity indicates the rate of BlockEditor use).



NGA: Not good at all
NG: Not good
G: Good

Figure 5: Distributions of rates of working with BlockEditor, by self-evaluated programming skill level.

Block-based versus text-based programming environments on novice student learning outcomes: a meta-analysis study

Article *in* Computer Science Education · January 2019

DOI: 10.1080/08993408.2019.1565233

4 authors, including:

Zhen Xu
University of Florida
5 PUBLICATIONS **7** CITATIONS
SEE PROFILE

Albert D. Ritzhaupt
University of Florida
100 PUBLICATIONS **1,181** CITATIONS
SEE PROFILE

Block-based versus text-based programming environments on novice student learning outcomes: a meta-analysis study

Article *in* Computer Science Education · January 2019

DOI: 10.1080/08993408.2019.1565233

CITATIONS
6

READS
346

4 authors, including:

Zhen Xu
University of Florida
5 PUBLICATIONS **7** CITATIONS

SEE PROFILE

Albert D. Ritzhaupt
University of Florida
100 PUBLICATIONS **1,181** CITATIONS

Slight caveat here. All the works they include cover "block-based" editors, but they use this to include non-projectional editors that have blocks that snap together. Also, not all of the comparisons are against text-based programming environments.

Study name	Hedges's g	Standard error	Variance	Lower limit	Upper limit	Z-Value	p-Value
Cilliers, Calitz, and Greyling (2005)	0.463	0.169	0.028	0.132	0.794	2.743	0.006
Garlick and Cankaya (2010)	-0.329	0.162	0.026	-0.646	-0.012	-2.031	0.042
Gul, Asif, Ahmad, and Ahmad (2017)	0.768	0.243	0.059	0.292	1.244	3.160	0.002
Lewis (2010)	0.223	0.280	0.078	-0.325	0.771	0.798	0.425
Lin and Yang (2009)	-0.154	0.274	0.075	-0.691	0.383	-0.562	0.574
Mladenovic, Boljat, and Žanko (2017)	0.983	0.196	0.038	0.599	1.367	5.020	0.000
Mladenovic, Rosic, and Mladenovic (2016)	0.584	0.414	0.172	-0.228	1.396	1.409	0.159
Okita (2014)	-0.547	0.316	0.100	-1.167	0.073	-1.729	0.084
Ruf, Mühling, and Hubwieser (2014)	0.139	0.265	0.070	-0.379	0.658	0.527	0.598
Weintrop and Wilensky (2017a; 2017b)	0.272	0.257	0.066	-0.232	0.776	1.058	0.290
	0.245	0.164	0.027	-0.078	0.567	1.488	0.137

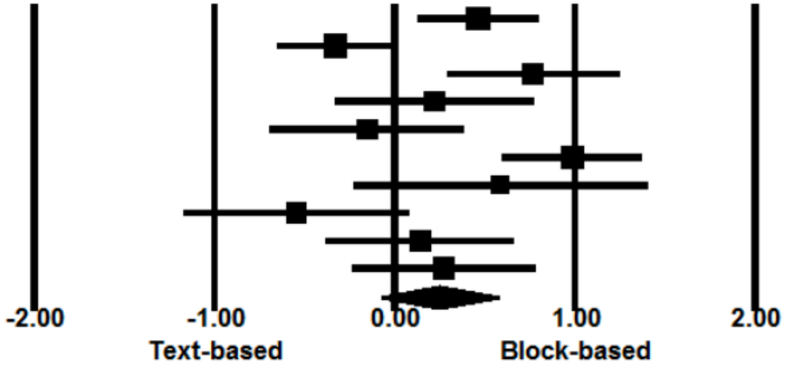


Figure 3. Forest plot of effect size details in cognitive model by study.

Study name	Hedges's g	Standard error	Variance	Lower limit	Upper limit	Z-Value	p-Value
Daly (2011)	2.609	0.506	0.256	1.618	3.600	5.159	0.000
Garlick and Cankaya (2010)	-0.331	0.169	0.028	-0.661	-0.000	-1.960	0.050
Mladenovic, Rosic, and Mladenovic (2016)	-0.229	0.407	0.166	-1.027	0.568	-0.564	0.573
Price and Barnes (2015)	0.268	0.354	0.125	-0.425	0.961	0.758	0.449
Ruf, Mühling, and Hubwieser (2014)	0.044	0.265	0.070	-0.476	0.563	0.166	0.868
Saito,Washizaki, and Fukazawa (2017)	-0.067	0.243	0.059	-0.544	0.410	-0.275	0.783
Weintrop and Wilensky (2017a; 2017b)	-0.160	0.256	0.066	-0.662	0.342	-0.625	0.532
	0.195	0.247	0.061	-0.289	0.678	0.790	0.429

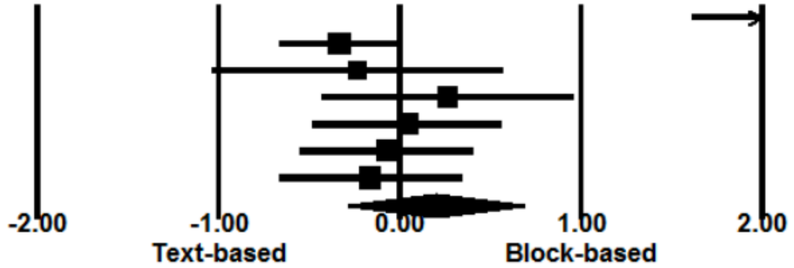
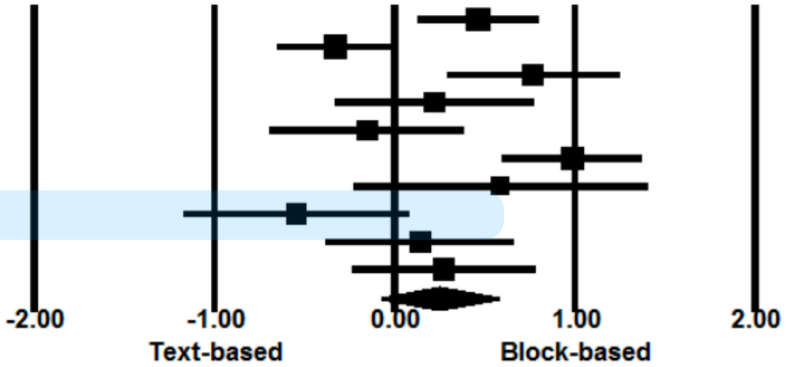


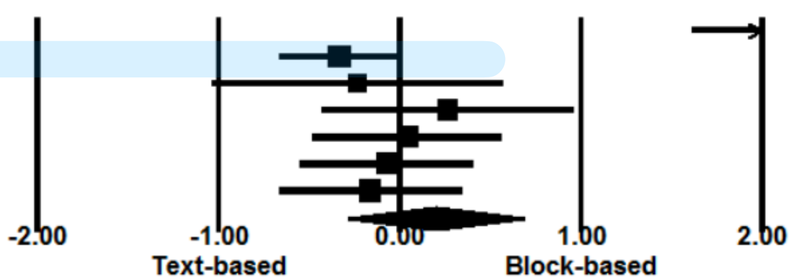
Figure 4. Forest plot of effect size details in affective model by study.

Mostly I'm not re-covering the same ground that's already covered in the meta-analysis, but I wanted to pull out the most negative studies in each category just to show that there really are works that find negative outcomes! (That is, show that textual editors can outperform projectional editors.)

Study name	Hedges's g	Standard error	Variance	Lower limit	Upper limit	Z-Value	p-Value
Cilliers, Calitz, and Greyling (2005)	0.463	0.169	0.028	0.132	0.794	2.743	0.006
Garlick and Cankaya (2010)	-0.329	0.162	0.026	-0.646	-0.012	-2.031	0.042
Gul, Asif, Ahmad, and Ahmad (2017)	0.768	0.243	0.059	0.292	1.244	3.160	0.002
Lewis (2010)	0.223	0.280	0.078	-0.325	0.771	0.798	0.425
Lin and Yang (2009)	-0.154	0.274	0.075	-0.691	0.383	-0.562	0.574
Mladenovic, Boljat, and Žanko (2017)	0.983	0.196	0.038	0.599	1.367	5.020	0.000
Mladenovic, Rosic, and Mladenovic (2016)	0.584	0.414	0.172	-0.228	1.396	1.409	0.159
Okita (2014)	-0.547	0.316	0.100	-1.167	0.073	-1.729	0.084
Ruf, Mühling, and Hubwieser (2014)	0.139	0.265	0.070	-0.379	0.658	0.527	0.598
Weintrop and Wilensky (2017a; 2017b)	0.272	0.257	0.066	-0.232	0.776	1.058	0.290
	0.245	0.164	0.027	-0.078	0.567	1.488	0.137



Study name	Hedges's g	Standard error	Variance	Lower limit	Upper limit	Z-Value	p-Value
Daly (2011)	2.609	0.506	0.256	1.618	3.600	5.159	0.000
Garlick and Cankaya (2010)	-0.331	0.169	0.028	-0.661	-0.000	-1.960	0.050
Mladenovic, Rosic, and Mladenovic (2016)	-0.229	0.407	0.166	-1.027	0.568	-0.564	0.573
Price and Barnes (2015)	0.268	0.354	0.125	-0.425	0.961	0.758	0.449
Ruf, Mühling, and Hubwieser (2014)	0.044	0.265	0.070	-0.476	0.563	0.166	0.868
Saito,Washizaki, and Fukazawa (2017)	-0.067	0.243	0.059	-0.544	0.410	-0.275	0.783
Weintrop and Wilensky (2017a; 2017b)	-0.160	0.256	0.066	-0.662	0.342	-0.625	0.532
	0.195	0.247	0.061	-0.289	0.678	0.790	0.429



British Journal of Educational Technology *Vol 45 No 5 2014* 844–862
doi:10.1111/bjet.12101

The relative merits of transparency: Investigating situations that support the use of robotics in developing student learning adaptability across virtual and physical computing platforms

Sandra Y. Okita

Sandra Y. Okita is an assistant professor of education and technology in the Department of Mathematics, Science, and Technology at Teachers College, Columbia University. Her research involves using innovative technologies (robots, agents and avatars, virtual reality environments) as a threshold to learning, instruction and assessment. Other areas include self-other monitoring, learning-by-teaching and metacognition in the domain of math, biology and science. Address for correspondence: Professor Sandra Y. Okita, Department of Mathematics, Science, and Technology, Teachers College, Columbia University, 525 West 120th Street, Box 8, New York, NY 10027-6696, USA. Email: okita@tc.columbia.edu; so2269@columbia.edu

Abstract

This study examined whether developing earlier forms of knowledge in specific learning environments prepares students better for future learning when they are placed in an unfamiliar learning environment. Forty-one students in the fifth and sixth grades learned to program robot movements using abstract concepts of speed, distance and direction. Students in high-transparency environments learned visual programming to control robots (eg, organizing visual icons), and students in low-transparency environments learned syntactic programming to control robots (eg, text-based coding). Both groups received feedback and models of solutions during the learning phase. The assessment midway showed students in both conditions learned equally well when solving problems using familiar materials. However, a difference emerged when students were asked to solve new problems, using unfamiliar materials. The low-transparency group was more successful in adapting and repurposing their knowledge to solve novel problems that required the use of unfamiliar high-transparency materials. Students in

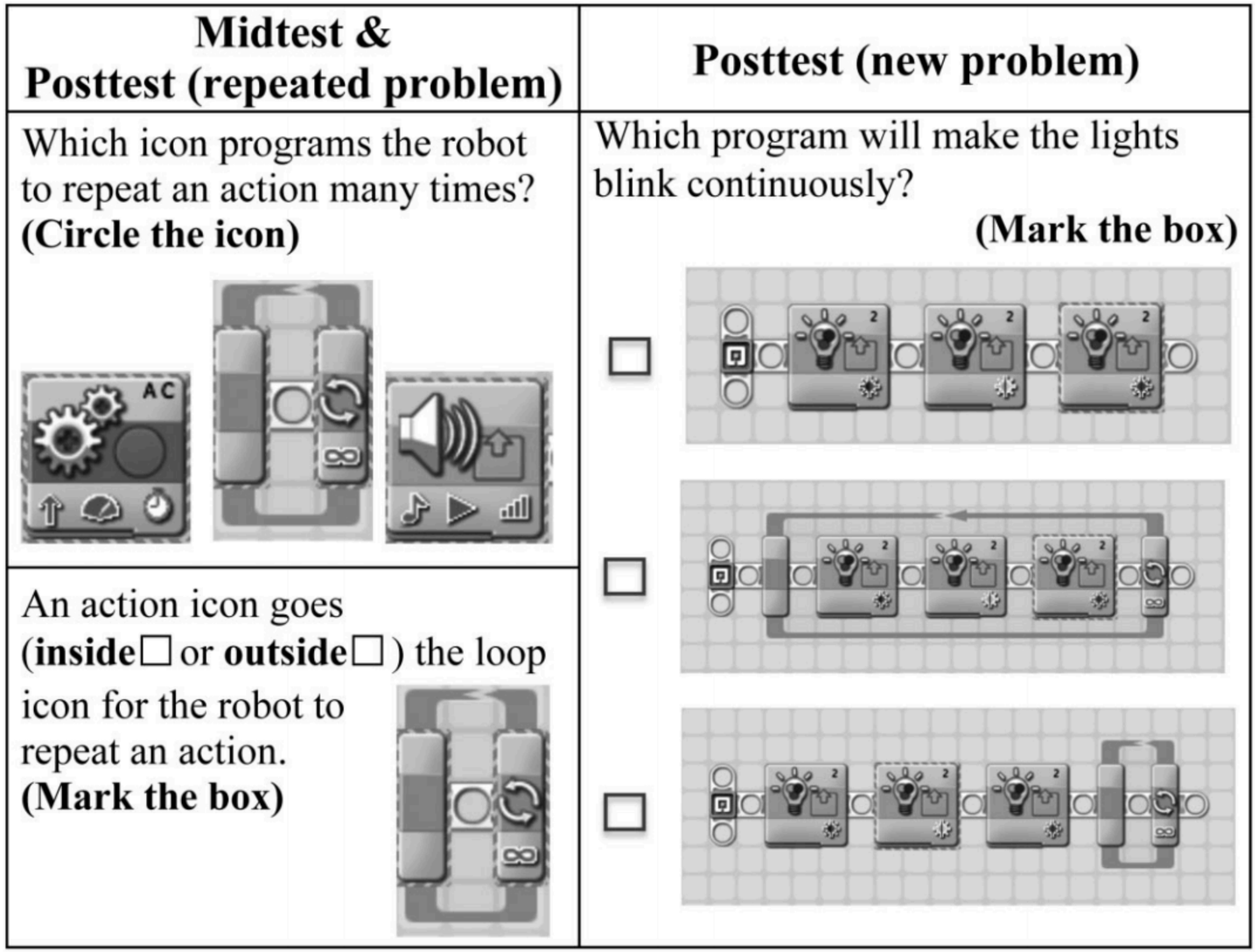


Figure 6: Example of virtual programming problems (virtual platform) on the midtest and posttest

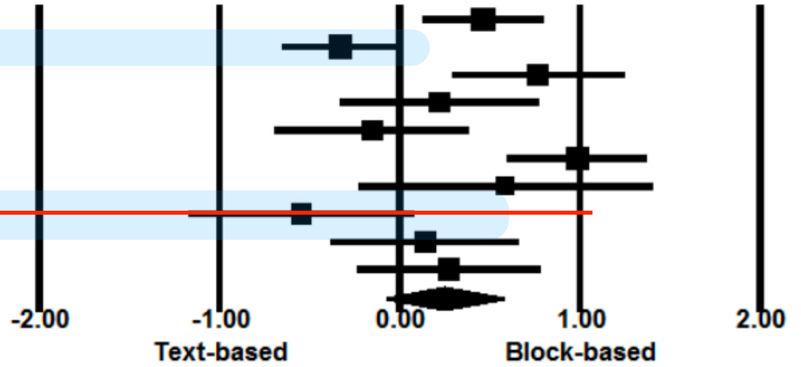
Oops, those are “blocks," but that’s not a structure editor. Next paper…

Mostly I'm not re-covering the same ground that's already covered in the meta-analysis, but I wanted to pull out the most negative studies in each category just to show that there really are works that find negative outcomes! (That is, show that textual editors can outperform projectional editors.)

Study name	Hedges's g	Standard error	Variance	Lower limit	Upper limit	Z-Value	p-Value
Cilliers, Calitz, and Greyling (2005)	0.463	0.169	0.028	0.132	0.794	2.743	0.006
Garlick and Cankaya (2010)	-0.329	0.162	0.026	-0.646	-0.012	-2.031	0.042
Gul, Asif, Ahmad, and Ahmad (2017)	0.768	0.243	0.059	0.292	1.244	3.160	0.002
Lewis (2010)	0.223	0.280	0.078	-0.325	0.771	0.798	0.425
Lin and Yang (2009)	-0.154	0.274	0.075	-0.691	0.383	-0.562	0.574
Mladenovic, Boljat, and Žanko (2017)	0.983	0.196	0.038	0.599	1.367	5.020	0.000
Mladenovic, Rosic, and Mladenovic (2016)	0.584	0.414	0.172	-0.228	1.396	1.409	0.159
Okita (2014)	-0.547	0.316	0.100	-1.167	0.073	-1.729	0.084
Ruf, Mühling, and Hubwieser (2014)	0.139	0.265	0.070	-0.379	0.658	0.527	0.598
Weintrop and Wilensky (2017a; 2017b)	0.272	0.257	0.066	-0.232	0.776	1.058	0.290
	0.245	0.164	0.027	-0.078	0.567	1.488	0.137

Statistics for each study

Hedges's g and 95% CI

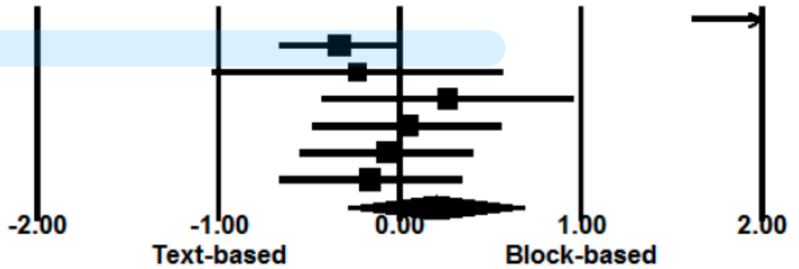


Awesome, 2 birds, 1 stone

Study name	Hedges's g	Standard error	Variance	Lower limit	Upper limit	Z-Value	p-Value
Daly (2011)	2.609	0.506	0.256	1.618	3.600	5.159	0.000
Garlick and Cankaya (2010)	-0.331	0.169	0.028	-0.661	-0.000	-1.960	0.050
Mladenovic, Rosic, and Mladenovic (2016)	-0.229	0.407	0.166	-1.027	0.568	-0.564	0.573
Price and Barnes (2015)	0.268	0.354	0.125	-0.425	0.961	0.758	0.449
Ruf, Mühling, and Hubwieser (2014)	0.044	0.265	0.070	-0.476	0.563	0.166	0.868
Saito,Washizaki, and Fukazawa (2017)	-0.067	0.243	0.059	-0.544	0.410	-0.275	0.783
Weintrop and Wilensky (2017a; 2017b)	-0.160	0.256	0.066	-0.662	0.342	-0.625	0.532
	0.195	0.247	0.061	-0.289	0.678	0.790	0.429

Statistics for each study

Hedges's g and 95% CI



Using Alice in CS1 – A Quantitative Experiment

Ryan Garlick
University of North Texas
Dept. of Computer Science and Engineering
garlick@unt.edu

Ebru Celikel Cankaya
University of North Texas
Dept. of Computer Science and Engineering
ecelikel@cse.unt.edu

ABSTRACT

We present the results of a 2-semester study of using the 3-D graphical programming environment Alice to introduce programming fundamentals during the first two weeks of CS1.

One cohort of students was taught basic programming constructs via traditional pseudocode, while a second group used Alice. A student survey was collected, along with performance metrics on a common quiz and first exam.

Students using Alice scored lower than those taught with pseudocode on common performance metrics and responded less-favorably to Alice in a survey. Anecdotal evidence of using Alice with younger students was more positive.

Categories and Subject Descriptors

K.3.2 **[Computer and Information Science Education]**: Computer Science Education.

General Terms

Measurement, Experimentation.

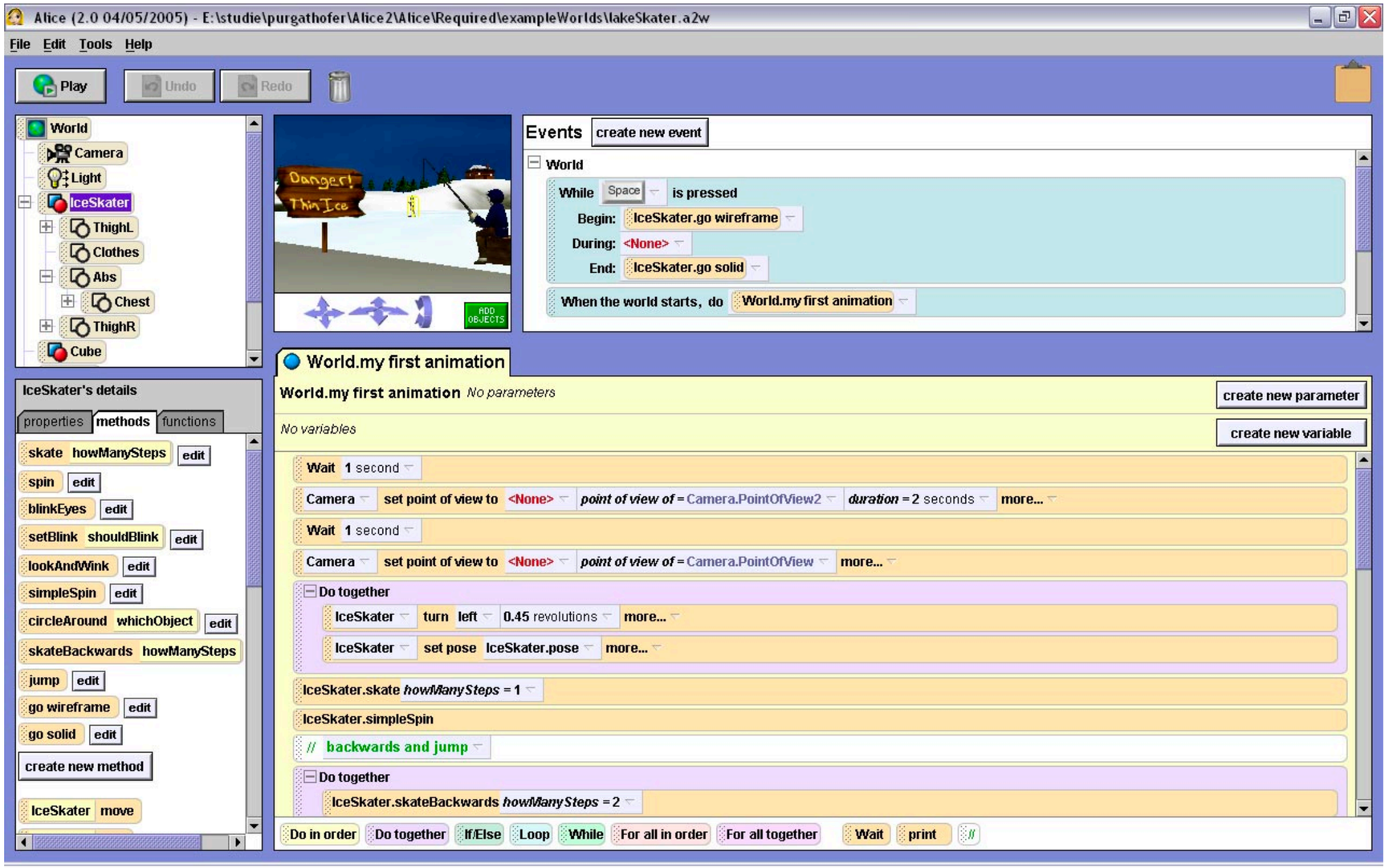
While proposed solutions have been numerous, empirical studies of what works and what does not are less common. Other studies have compared CS1 students who took a CS0 course to those who did not [7, 9]. Drawing conclusions about the influence of CS0 content seems difficult, as exposure to any programming related material in an additional course is likely to improve CS1 performance.

This paper aims to measure the effect of using a brief introduction to programming fundamentals via Alice versus a control group using traditional pseudocode (hereafter the pseudocode cohort). The comparison period involved the first two weeks of each CS1 class.

Three CS1 instructors participated in this study (including the authors), which encompassed 5 class sections, 2 semesters, and over 150 students.

From the Alice website [1]: *[Alice] allows students to learn fundamental programming concepts in the context of creating*

Alice vs. Pseudocode



3.1 Alice Class Sections

Two CS1class sections (82 total students) were presented with instruction in Alice. A very brief introduction to programming was presented in pseudo-code style statements. However, the core concepts of variable creation, looping, control statements and functions were presented via examples given in Alice, with discussion of how Alice was enabling a graphical representation of logical constructs.

Students worked through the tutorials included with the Alice software and were provided with links to additional Alice resources and tutorials. Two weeks of classroom lecture were spent covering these topics, introducing Alice, and working through examples in class, followed by a transition to Java.

The homework assignment was as follows:

Create an animation in the Alice environment. Your topic may be anything that you choose. Special consideration will be given to interactivity, complexity, and creativity. Your animation must contain the following elements:

- *You must create a new variable, function, and method*
- *You must have interactivity in your animation that is controlled by the user.*
- *You must have a loop in the program.*

Since the public will vote on the best animations, you are encouraged to learn extended techniques to make your program more sophisticated. Prizes will be awarded to the best submissions.

3.2 Traditional Pseudocode Class Sections

Two separate CS1 class sections (72 total students) were introduced to programming concepts via traditional pseudocode. This involved two weeks of slides and discussion related to pseudocode. The same topics: looping, control statements, and functions were introduced by tracing through pseudocode algorithms rather than via Alice. The pseudocode cohort also transitioned to Java.

The homework assignment in these sections was to create a pseudo code algorithm to calculate GPA.

3.3 Homework, Assignments and Quizzes

A common quiz was given to both the Alice and pseudocode cohorts. It included a given algorithm for finding the average of a group of numbers using a loop and conditional statements presented as blocks with arrows drawn between them to indicate program flow. The assignment was designed to provide a mix of pseudocode and the "tiles" dragged into the Alice environment to form programming constructs. Students were then asked to create a "make change" program, as illustrated by this excerpt from the quiz:

"Create a process for determining the correct number of quarters, dimes, nickels and pennies to give as change. For example, if the change amount is .82 (it will always be .99 or less), your process should end up with quarters = 3, dimes = 0, nickels = 1, pennies = 2".

Students were told they could use any method to solve the problem – drawing flowchart-style blocks similar to the development window in Alice, writing pseudo-code, or any other method that presented a coherent solution to the problem. This flexibility was designed to minimize any possibility of bias to the advantage or disadvantage of either teaching cohort employed in the experiment.

The mean quiz grade was lower among students in the Alice cohort, however the data did not quite establish statistical significance to a P-value of 0.05 (P=.0527).

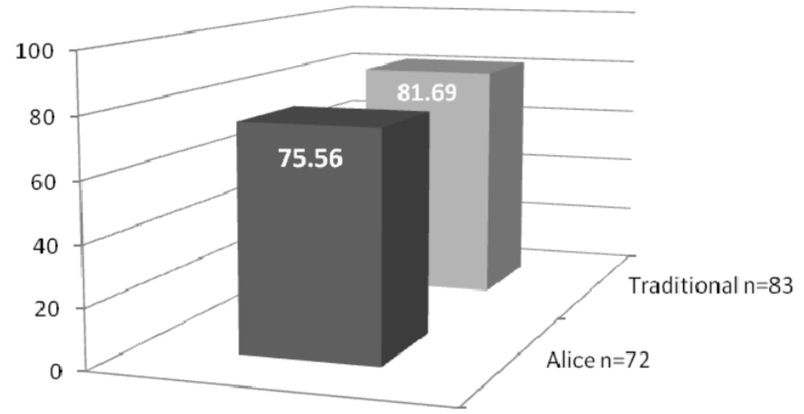


Figure 2. Mean Quiz Grade By Cohort

A common first exam was given 2-3 weeks after the Alice / pseudocode instruction. While largely based on Java syntax, the exam included questions that tested the ability to recognize the output of given programs and analyze existing program logic.

All exams were graded by a single instructor without knowledge of the students' cohort. Mean exam grades were significantly higher among students in the traditional cohort (P=.0291).

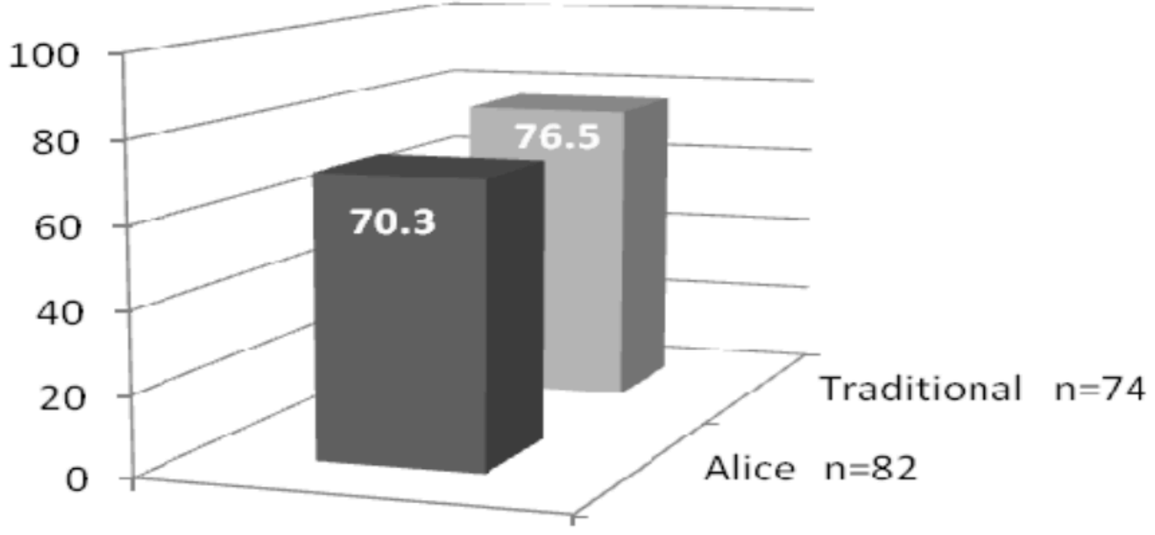
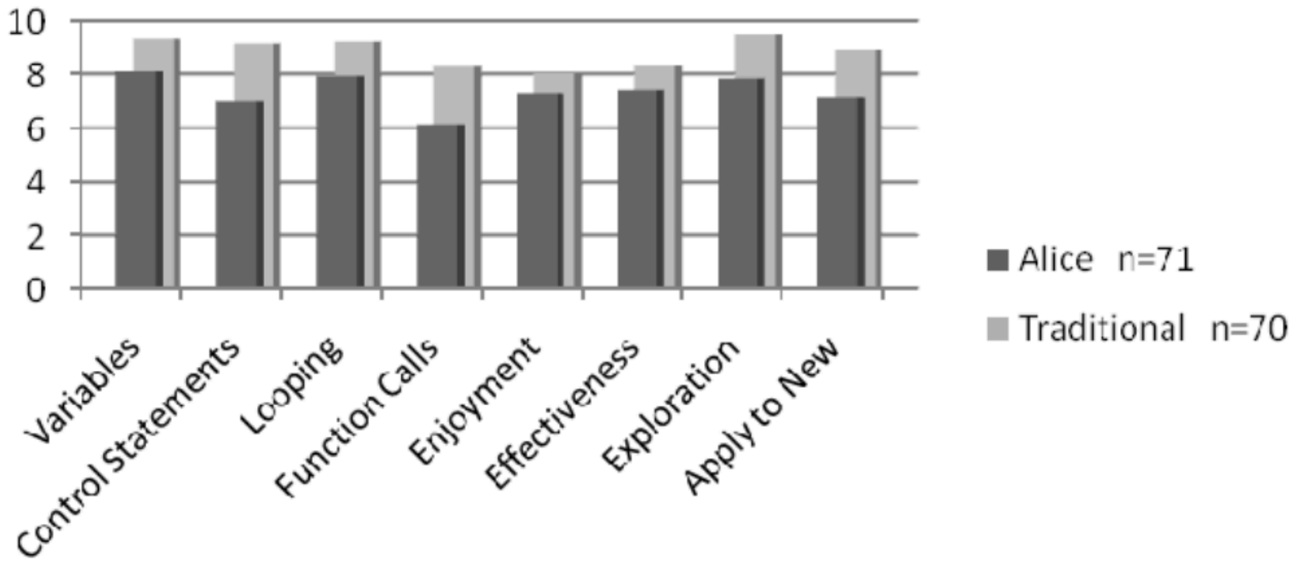


Figure 3. Mean First Exam Grade by Cohort


10
8
6
4
2
0
Variables
Control Statements
Looping
Function Calls
Enjoyment
Effectiveness
Exploration
Apply to New
Alice n=71
Traditional n=70

Also one of the papers y'all read.

Efficiency of Projectional Editing: A Controlled Experiment

Thorsten Berger
Chalmers | University of Gothenburg, Sweden

Markus Völter
independent / itemis Stuttgart, Germany

Hans Peter Jensen, Taweesap Dangprasert
IT University of Copenhagen, Denmark

Janet Siegmund
University of Passau, Germany

ABSTRACT

Projectional editors are editors where a user's editing actions directly change the abstract syntax tree without using a parser. They promise essentially unrestricted language composition as well as flexible notations, which supports aligning languages with their respective domain and constitutes an essential ingredient of model-driven development. Such editors have existed since the 1980s and gained widespread attention with the Intentional Programming paradigm, which used projectional editing at its core. However, despite the benefits, programming still mainly relies on editing textual code, where projectional editors imply a very different—typically perceived as worse—editing experience, often seen as the main challenge prohibiting their widespread adoption. We present an experiment of code-editing activities in a projectional editor, conducted with 19 graduate computer-science students and industrial developers. We investigate the effects of projectional editing on editing efficiency, editing strategies, and error rates—each of which we also compare to conventional, parser-based editing. We observe that editing

and directly change the AST with their editing gestures. This concept is different from parser-based editing, where users change the concrete syntax (characters in a text buffer), and a parser then matches the syntax against a grammar definition to construct the AST. Projectional editing, also known as structured editing or syntax-directed editing, is not a new idea; early references go back to the 1980s and include the Incremental Programming Environment [32], GANDALF [35], and the Synthesizer Generator [39]. Work on projectional editors continues today: Intentional Programming [44, 18, 45, 14] is its most well-known incarnation. Other contemporary tools [20] are the Whole Platform [9], Más [3], Onion, and MPS [4]. The latter is the instrument of this work. Most of these projectional editors are used in language workbenches—tools for developing and composing languages [20, 21].

Projectional editors have two main advantages, both resulting from the absence of parsing. First, they support notations that cannot easily be parsed, such as tables, diagrams or mathematical formulas—each of which can be mixed with the others and with textual notations [45, 51]. Second, they support various ways of language composition [19], typically

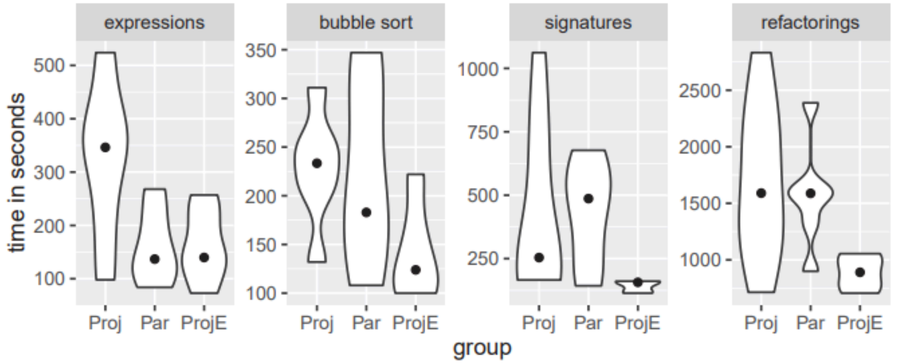


Figure 2: Task completion times in seconds (violin plots)

Proj: Projectional (inexperienced MPS users)
Par: Parser (text editor)
ProjE: Projectional + Experts (experienced MPS users)

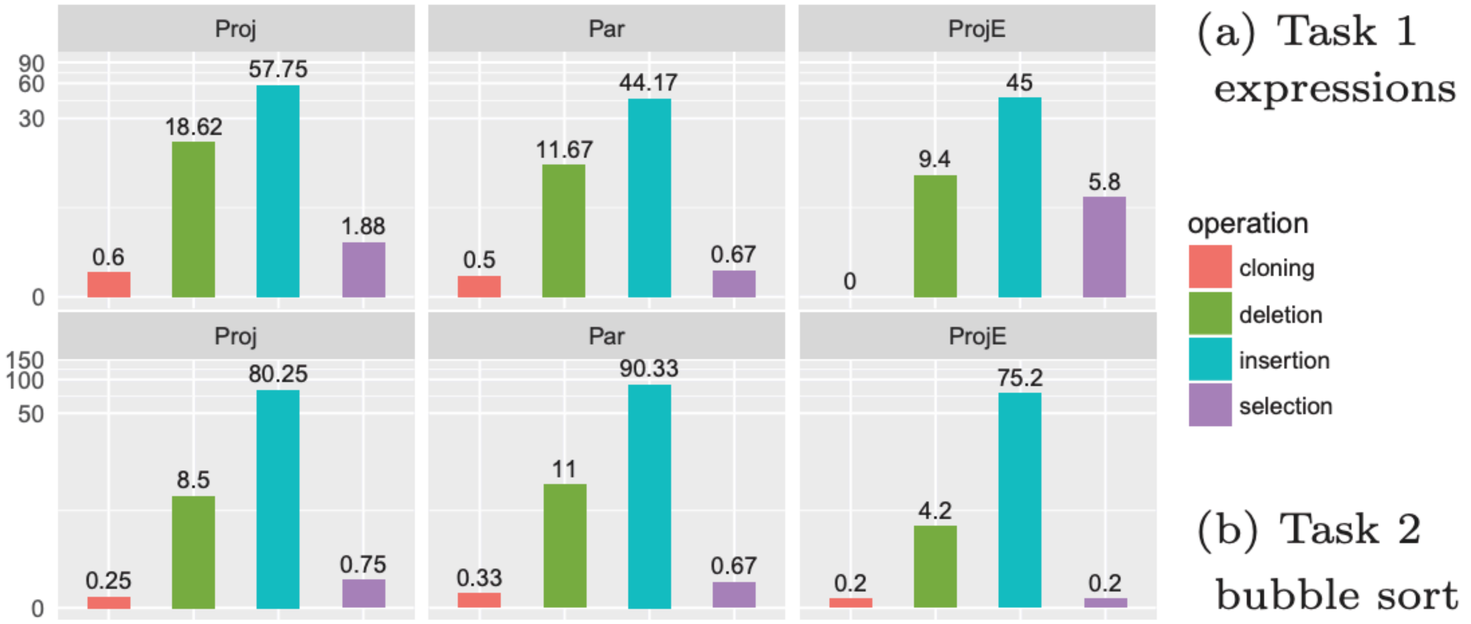
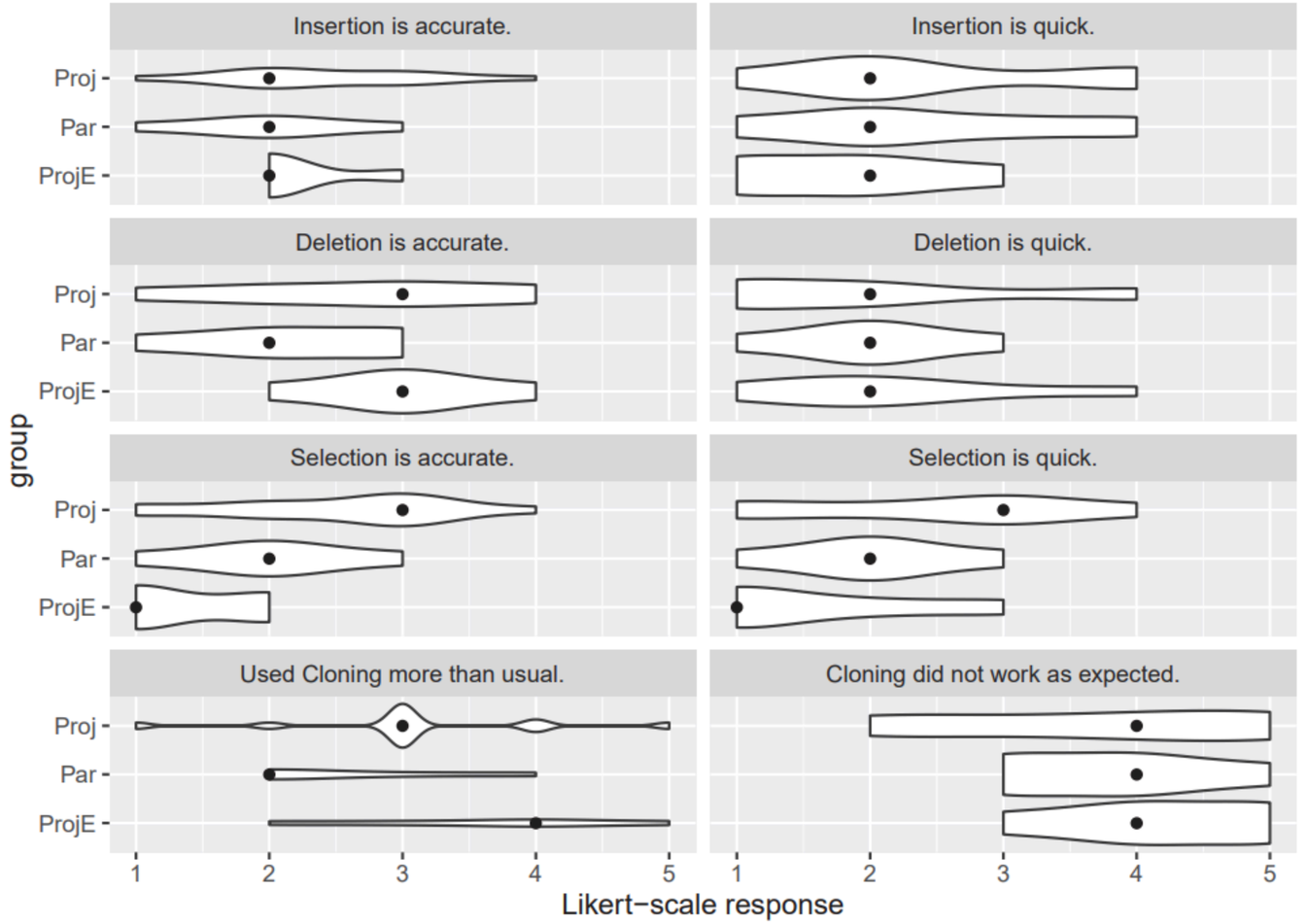


Figure 3: Average use of basic-editing operations per participant. Y-axis in log(y+1) scale.



1: strongly agree, 2: agree, 3: neutral, 4: disagree, 5: strongly disagree

Figure 4: Opinions about basic editing

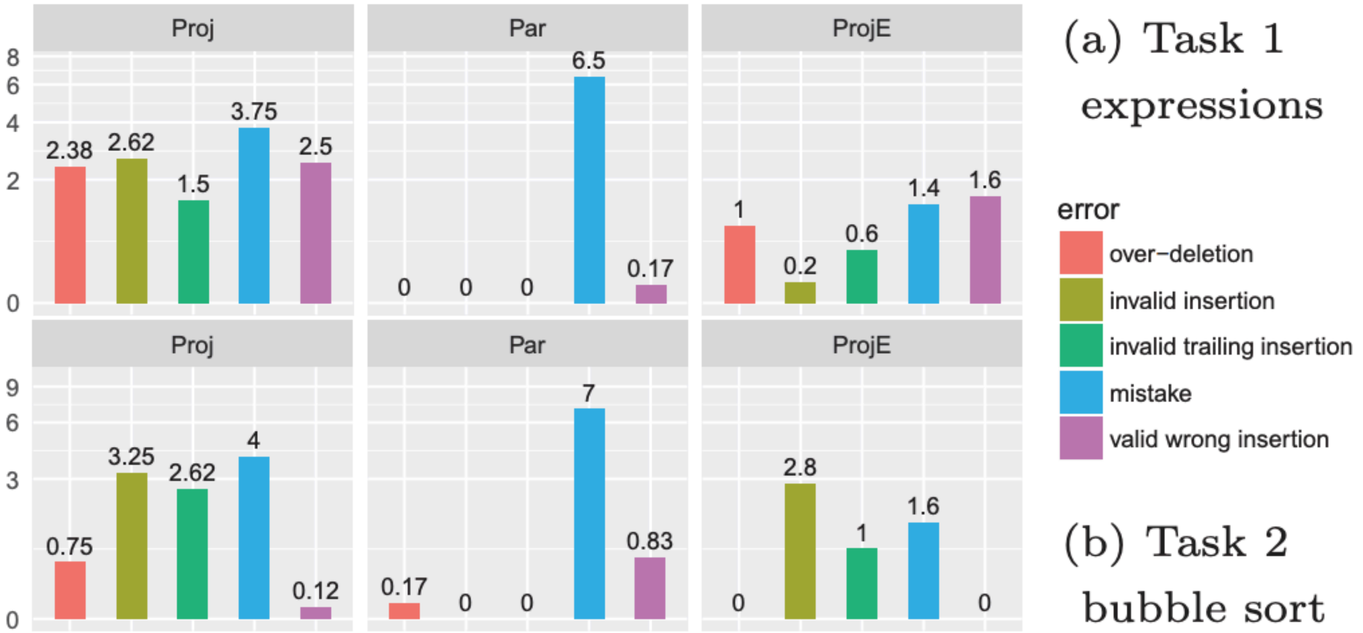


Figure 5: Average occurrence of errors per participant. Y-axis in log(y+1) scale.

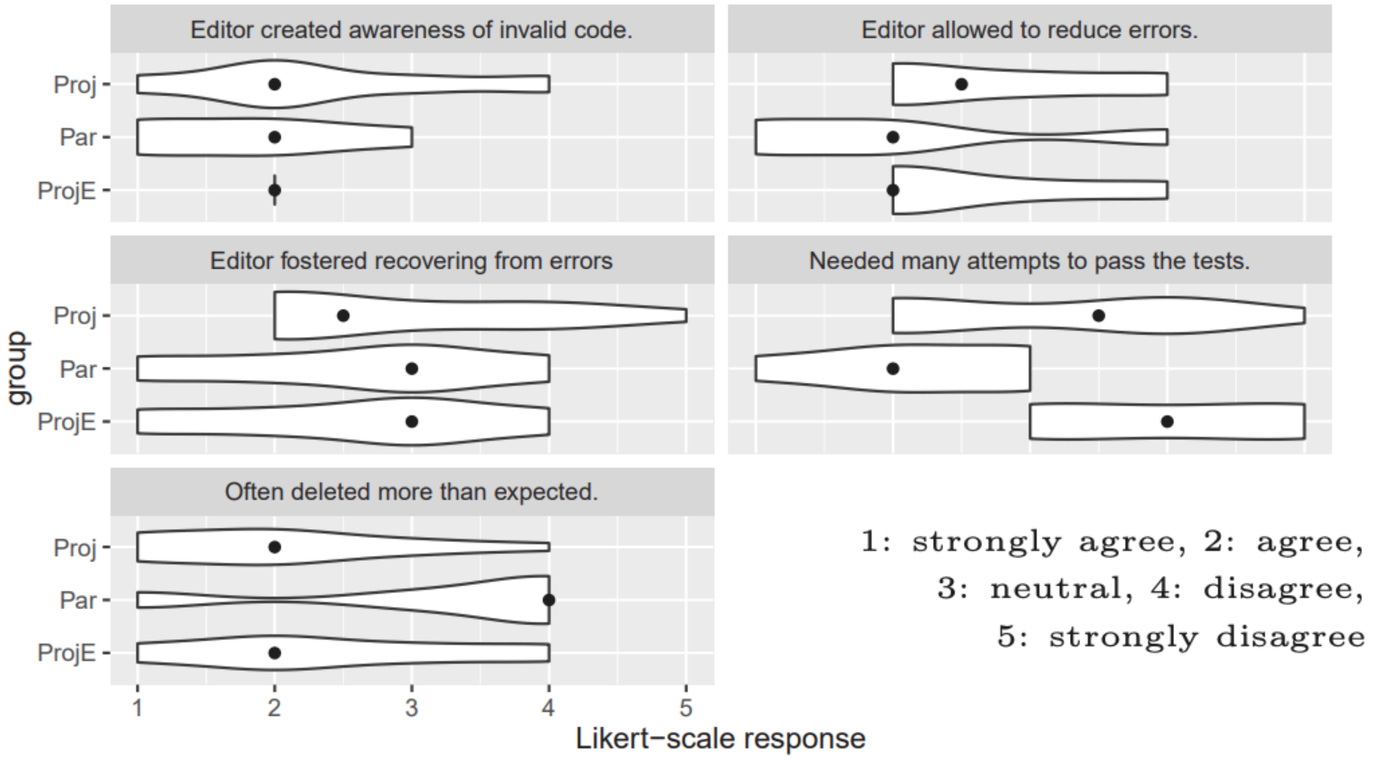


Figure 6: Opinions about errors

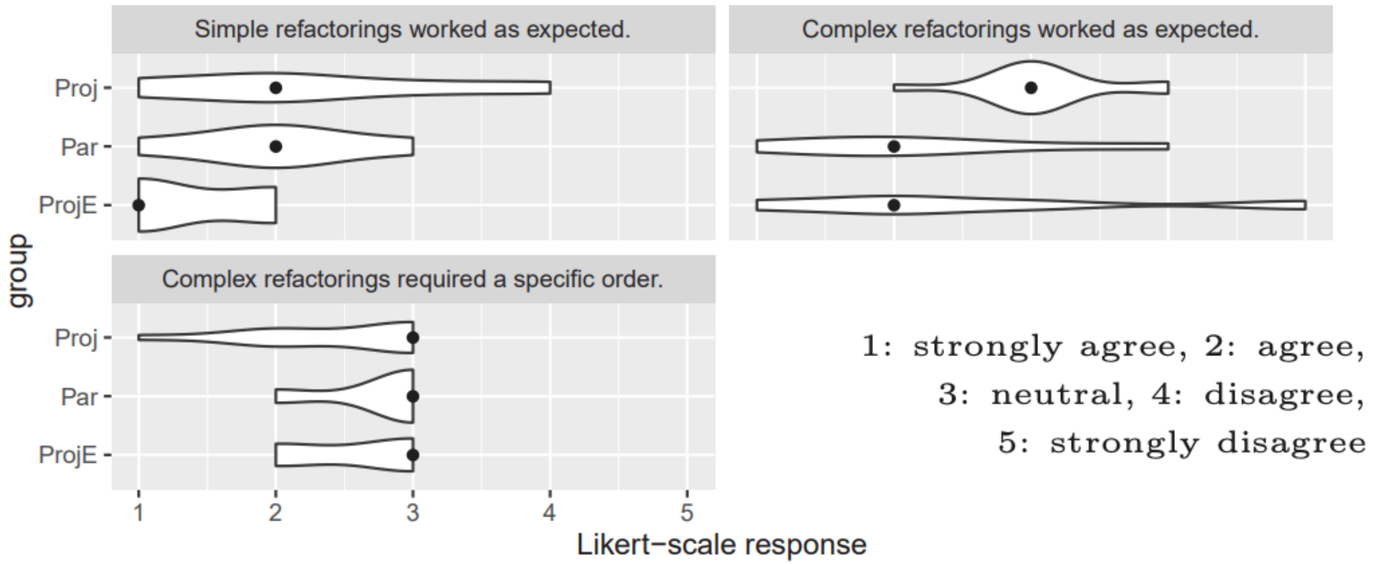


Figure 7: Opinions about refactoring operations

What's the Effect of Projectional Editors for Creating Words For Unknown Languages? A Controlled Experiment

Niklas Hollmann, Thorben Roßenbeck, Mark Kunze, Liron Türk, Stefan Hanenberg
Paluno - The Ruhr Institute für Software Technology
University of Duisburg-Essen, Essen, Germany
niklas.hollmann|thorben.rossenbeck|mark.kunze|liron.tuerk@stud.uni-due.de
stefan.hanenberg@uni-due.de

1 Introduction

Projectional editors (aka. structure editors, structural editors, block-based editors, etc.) are quite an old technique that was probably according to Gomolka and Humm [4] first articulated by 1971 [6]. The goal of a projectional editor is to provide tool support for writing documents that follow a given structure. Spoken in terms of grammars, it means a projectional editor for a given grammar permits to write only valid words of the language defined by the grammar: it gives users the ability to fill in all nodes in the syntax tree by traversing the tree manually. Hence, users either write incomplete words or complete and valid words of the language. For example, for a simple grammar <Start>→ "X" ("A" | "B" "C" | "D")$^+$ "Y" users is given the ability to start with the "X" [something] "Y" and then to decide what to do with [something]. Then, users just have the option to insert an "A", "B" or "D". In

were more recently introduced to a larger audience (see also [16]).

While the motivation for such kind of editing is plausible, it is not that clear what the effect of such kind of editors is. A recent controlled experiment by Berger et al. [1] revealed quite mixed results for the comparison of projectional and text-based editing for non-trained users and for programming related tasks: in average the text editor group even had a positive measurable benefit compared to the projectional editor group. However, the result assumed one thing: the people working on the tasks were familiar with the underlying syntax.

We believe projectional editors have a large benefit – but rather not in situations where the language is known. I.e. we do not think that the main benefit is in the pure editing process. We think the main benefit of projectional editors is in situa-

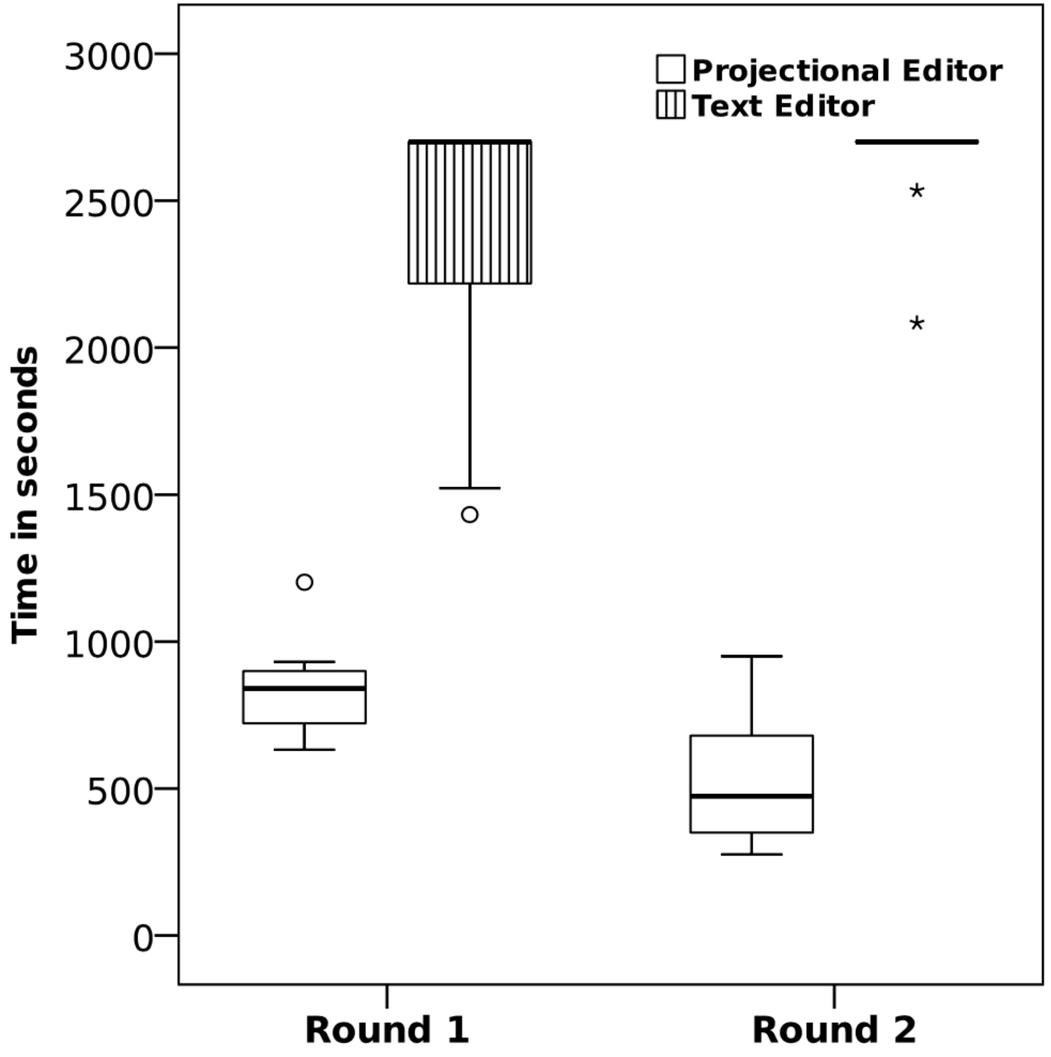


Figure 2. Boxplot for round 1 and 2 and projectional vs. text editor

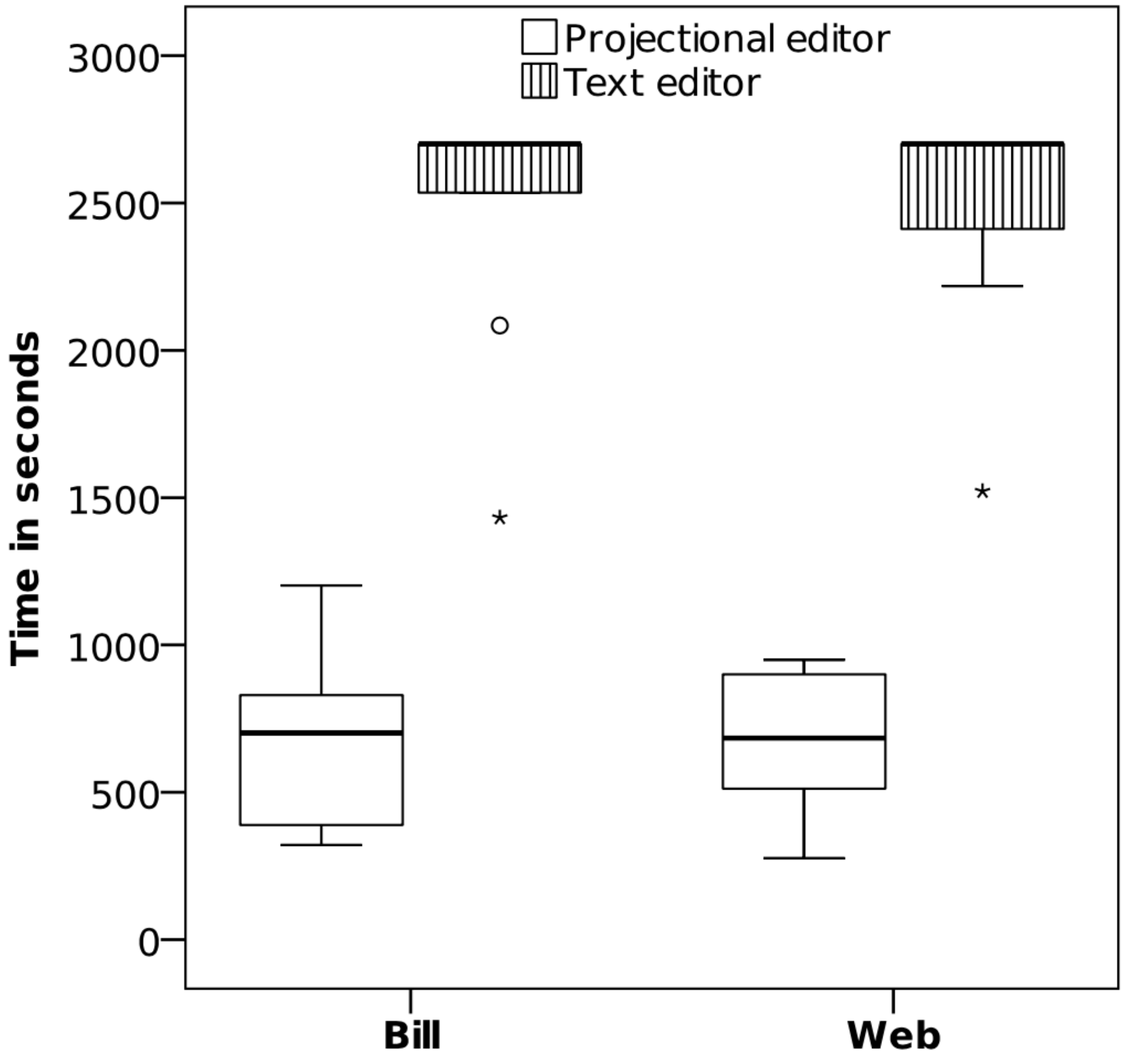


Figure 3. Boxplot for each task and projectional vs. text editor

A Few High-Level Takeaways

- For familiar languages—e.g., in a course setting where language becomes familiar over time—text-based and structure-based editors are surprisingly similar
  - I said surprising, but in some ways this makes sense; when we hold the language stable, it's basically the same task just done in mildly different styles
- Over time, given the option of transitioning smoothly between the two, users start using text more than at the beginning (though not always more than structure editing mode)
- Beginners have fonder feelings about CS when they start with structure editors vs. with text editors
- Structure editors aren't a good substitute for pseudocode
- For *unfamiliar languages*—e.g., domain-specific languages that will be used once a year—structure editors are more efficient

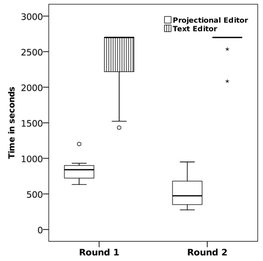


Figure 2. Boxplot for round 1 and 2 and projectional vs. text editor

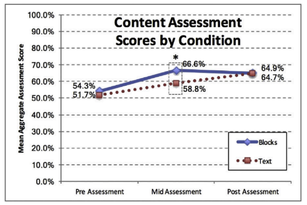


Fig. 3. The mean scores for students in the two conditions on the three administrations (Pre, Mid, and Post) of the Commutative Assessment.

It's super cool that you now know your biases on this topic! I hope this is useful self-knowledge! :)

Colorful puzzle-looking editors look like kids' toys to me, and I refuse to believe they're real programming.

I don't care that people learn just as many computing skills with them, can transfer knowledge to other programming environments, or that I can use the same programming languages and write the same programs in both kinds of environments!!! These environments feel restrictive to me, and I can't take them seriously!!!

Why did we spend all this time on this?

- Not because this course needs a bunch of data on projectional editors in particular, although it's convenient that we already have a lot of human factors studies of them.
- Perhaps a little bit more because of all the strong opinions programmers hold about them.
- Primarily because one of the biggest goals of this course is that you won't rely on folk theory in your PL and programming environment design decisions.
  - Our own intuitions and experiences are *awesome* for helping us brainstorm, giving us the ideas that we'll eventually prototype and put in front of users.
  - But reliance on folk theories, the tenets of various PL design factions, and personal experiences is how we got to the messy languages we have now!
  - …and hopefully you're taking this class because you think we can do better! :)

Why did we spend all this time on this?

- So how do we do better?
  - Surprisingly often, you can look to the literature to see if there's support for your folk theory!
    - There's a lot of research already out there
  - And when there's no research out there already?
    - By the end of this class, you'll have the tools you need to design and execute the research yourself!

Why did we spend all this time on this?

- And what should we do about folk theories?
  - *Don't* ignore them
    - I know, I know, I just spent all this time talking about how folk theories can be dangerous, lead us down bad design paths
  - *Do* see them as a great source of hypotheses
    - A community of practice often *does* observe important features of their domain before "science" catches on
    - Going to steal James C. Scott's definition of metis: "a wide array of practical skills and acquired intelligence in responding to a constantly changing natural and human environment."
  - *Don't* trust them blindly
    - Just don't take them as fact!
    - A hypothesis is just a hypothesis. We'll start making decisions with it once it's been supported or not supported

Final project stuff!

- Please form groups!
  - We have a survey going out today to let us know about your group formation status
  - We have a spreadsheet where you can include group information
- You can find the full guidelines for the final project in the Project Presentations and Project Writeups links at the bottom of the calendar.
  - That document is also an interactive FAQ! Feel free to add your questions there if anything is unclear.
  - Or feel free to swing by office hours if you want to double check whether your group's plan is looking good.

Final project stuff!

- Timeline things:
  - Remember that the IRB never provides retroactive approval for publishing data collected from unapproved activities
  - If you think there's *any* chance at all that you want to publish on the work you'll do during the course project:
    - get your CITI training
    - submit your IRB proposal
    - expect the IRB to get back to you within about 2 weeks
  - Even if you're not going through the IRB process, remember that recruiting study participants and running sessions is time consuming! It's good to start ramping up now!

Final project stuff!

- Timeline things:
  - Around 6 weeks before the end of the semester, we're going to switch over to having all of our homework supporting the final project
    - (Basically writing up some chunks of the final writeup for early feedback)
  - If you wait until that point to start doing the work to be written up, it will already be too late!

Final project stuff!

- Recommended formula:
  - A small-scale need finding study
  - A prototype of a programming language, tool, or environment that meets one or more of the needs identified in the study

Final project stuff!

- Other formulas…
  - …totally possible, but if you're going to try something else, then pay close attention to the next slide

Final project stuff!

- How not to get an A on the final :(

- This is a pretty laid back mostly-PhD course. Goal here is for everyone to get an A.

- Here are some things that will cause not-an-A:
  - The work doesn't implement a novel programming language, programming environment, or other programming tool
  - The work doesn't include a need-finding or formative user study
  - The design of the language, environment, or tool isn't **shaped by the findings from the earlier user study**. (If your initial design isn't supported, then don't write up the initial design! Write up the design after iterating based on the user study!)

- If you're worried that your work might fall into one of the three pitfalls above, come talk to me! I have OH every week!

Goal for next reading

- Prepare to write a program slicer! Understand the basics in preparation for writing your own.