Exploring the Impact of Worked Examples in a Novice Programming Environment

Rui Zhi, Thomas W. Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, Min Chi
North Carolina State University
Raleigh, NC
rzhi,twprice,samarwan,aamillik,tmbarnes,mchi@ncsu.edu

ABSTRACT

Research in a variety of domains has shown that viewing worked examples (WEs) can be a more efficient way to learn than solving equivalent problems. We designed a Peer Code Helper system to display WEs, along with scaffolded self-explanation prompts, in a block-based, novice programming environment called Snap*!*. We evaluated our system during a high school summer camp with 22 students. Participants completed three programming problems with access to WEs on either the first or second problem. We found that WEs did not significantly impact students' learning, but may have impacted students' intrinsic cognitive load, suggesting that our WEs with scaffolded prompts may be an inherently different learning task. Our results show that WEs saved students time on initial tasks compared to writing code, but some of the time saved was lost in subsequent programming tasks. Overall, students with WEs completed more tasks within a fixed time period, but not significantly more. WEs may improve students' learning efficiency when programming, but these effects are nuanced and merit further study.

ACM Reference Format:
Rui Zhi, Thomas W. Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, Min Chi. 2019. Exploring the Impact of Worked Examples in a Novice Programming Environment. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287385

1 INTRODUCTION AND BACKGROUND

In computing classrooms, students are often expected to learn by solving programming problems independently. An alternative approach is to show students worked examples (WEs), that guide the student through solving the problem, step by step. WEs have been studied in a variety of domains [9, 10, 20], and the results consistently show that replacing some or all practice problems with WEs results in higher learning efficiency, where students take less time to learn the same amount of content. One way WEs may increase learning efficiency is by reducing students' cognitive load, or mental effort when learning [18]. Though a number of studies have explored the impact of WEs in the domain of programming [2, 3, 12, 14, 22–24], few of these studies have specifically compared WEs to traditional problem solving [22], especially in novice programming environments [23].

In this work, we present the "Peer Code Helper" (PCH), which integrates WEs into the Snap*!* programming environment [4]. We evaluated the PCH and the impact of programming WEs during a high school summer camp with 22 students. We found that access to WEs did not significantly impact students' learning, though they may have impacted students' intrinsic cognitive load. Students with WEs on a problem completed more objectives on that problem, but this difference was not significant. Our results suggest that WEs allowed students to work more efficiently on the first few problem objectives, but they took longer to complete subsequent objectives that were not supported with WEs. The difference in the time taken on each subsequent objective between students with and without WEs closed over time, and difference in problem solving speed was still much less than the time saved by the WEs on the first few objectives. Our findings show that WEs may help students solve more objectives in the same amount of time, but further study is needed to confirm this result.

1.1 Cognitive Load Theory

Our work is based on Cognitive Load Theory (CLT), which describes the relationship between mental effort and learning [21]. CLT believes that all learning activities impose a *cognitive load* on working memory, and this cognitive load comes in three categories: intrinsic, extraneous, and germane. Intrinsic load comes from the inherent difficulty or effort required to accomplish the learning task. Extraneous load comes from the way information or tasks are presented. Germane load refers to the work or effort needed to create new, permanent knowledge. CLT posits that humans have limited working memory, and it suggests that students' learning can be impaired if instructional materials require students to hold excessive amounts of information in working memory. Since the intrinsic load and extraneous load are additive [19], extraneous load should be minimized to maximize germane load and thus learning. Motivated by CLT, we integrated WEs into Snap*!* with the goal of reducing students' extraneous cognitive load while working on open-ended programming problems.

1.2 Worked Examples

A worked example (WE) teaches students how to solve a problem by directly presenting a solution for the problem. It is an effective instructional strategy to help novices learn complex tasks and has been studied for decades [18]. Prior studies have explored the

SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287385

effect of WEs in different disciplines, including algebra [20], propositional logic proofs [9], and stoichiometry [10]. Empirical studies have shown that WEs are more beneficial to novice learners compared with traditional problem solving in learning algebra [20]. Sweller and Cooper found that students who were presented with WE problems solved similar problems quicker and with fewer errors than students who solved the same problems themselves [20]. McLaren et al. compared four different instructional designs in a stoichiometry tutor. They found that students in the WE condition performed similarly in the post-test but were more efficient in finishing intervention problems and with less cognitive load, compared to students in other conditions including problem solving, erroneous worked examples, and tutored problem solving [10]. In an intelligent tutoring system for teaching and practicing propositional logic problems in discrete math, Liu et al.'s [9] work shows that WEs are beneficial to undergraduate students in the first few problems they solve but less beneficial for later problems in the students' homework set.

Worked examples have also been studied in the domain of programming. Some studies suggest using WEs in programming for teaching novices. Pirolli and Anderson investigated how three novices learned recursive functions and found that they relied heavily on examples while writing new programs [14]. Informed by the results from cognitive science and educational psychology, Caspersen and Bennedsen proposed a model which treats WEs as one of the fundamental principles of programming education [2]. Vihavainen et al. extended Caspersen and Bennedsen's work and proposed a model, which also suggests using WEs in study materials and lectures [24].

Others have investigated the impact of WEs experimentally. Trafton and Reiser's study suggests interleaving WEs with similar practice problems to maximize students learning gains, compared to blocking WEs with problems, or solving equivalent problems by writing LISP programs [22]. Recently, Ericson et al. found that interleaving WEs with Parsons problems can improve programming learning efficiency compared to WEs paired with writing code or fixing code [3]. Patitsas et al. investigated the effect of comparing and contrasting multiple solutions, by presenting the different solutions (WEs) side by side. They found that students in the comparing and contrasting condition showed greater learning gains for procedural knowledge and flexibility than students in the WE condition [13]. However, to our knowledge, only Trafton and Reiser directly evaluate the effectiveness of WEs by comparing them to equivalent problem solving tasks, but their results may not apply to novice programming environment.

In novice programming environment, Ichinco et al. showed that novices struggle to use WEs to solve block-based programming problems [6]. For example, they found that novices have difficulty in understanding the examples and finding relevant blocks to finish the programming task. To limit the impact of this difficult transfer task, we hand-selected WEs and presented them in small steps with scaffolded self-explanation prompts in PCH. Merriënboer and Croock found that *incomplete* WEs, in which students are given part of a WE and asked to complete the rest, improved novice's programming performance and post-test scores, compared with those who only had the WEs as a reference[23]. Inspired by this, our PCH provides students the WEs as starter code for their own solutions once students have completed their self-explanation prompts. Motivated by results from these studies, we combined the incomplete WEs with WE steps and investigated the impact of *incomplete* WEs on novice programming. Our prior work with WEs in a novice programming game showed that when students are given only WEs, they may skip through them quickly without learning [26].

1.3 Self Explanation

Self-explanation is an instructional strategy which requires the learner to reflect and generate explanations about the learning material. It has been shown that combining self-explanation with WEs can be especially beneficial to students' learning [1]. A review of self-explanation shows that along with WEs, self-explanation prompts help students foster greater understanding [16] and improves students' problem solving performance [12]. Morrison et al. showed that students who self-explained subgoal labels perform better than those who were not given subgoal labels [12]. Self-explanation has different forms, including open-ended, menu-based, focused, scaffolded, and resource-based self-explanations [16]. Berthold found that scaffolded self-explanation prompts following WEs were more beneficial to students in terms of conceptual and procedural understanding than open-ended self-explanation prompts in multimedia environments [1]. Therefore, our PCH system provides scaffolded self-explanation prompts, that guide students to reflect on WE code.

2 METHOD

To investigate the impact of incomplete worked examples (WEs) on novice programmers' learning outcomes, we designed a "Peer Code Helper" (PCH) feature in Snap*!* to demonstrate the first few steps of a programming problem as structured WEs. We evaluated the impact of these PCH WEs as compared with traditional problem solving during a high school coding camp for girls in summer 2018. Specifically, we investigated the following research questions. How does having access to WEs during a programming problem impact: **RQ1**: Students' learning during the problem? **RQ2**: Students' perceived difficulty and cognitive load with respect to the problem? **RQ3**: Students' programming efficiency?

2.1 The Peer Code Helper

We designed and developed the Peer Code Helper (PCH) [1] interface to present interactive WEs in a block-based programming environment called Snap*!*, which is used extensively in the Beauty and Joy of Computing (BJC) AP Computer Science Principles curriculum [4]. Our design is based on the implementation of WEs [23] and self-explanation prompts in other domains [1] as described in the related work. PCH shows students the first few steps for solving the given problem. A step was designed to be a meaningful chunk of code to finish an objective of a problem (e.g. draw a square/circle), though the exact example code may not end up in the final solution. For each step, we first show some starting code ("before code"), without the step completed, and ask the student to run it and answer questions related to the function of the code via scaffolded self explanations, as shown in Figure 1. Then we show what the code looks like after the step has been completed ("after code") and

[1]Contact the first author to request a copy of the system.

ask them to run the new code and consider why each new piece of code was added. Students can compare the "before" and "after" code and reflect on the changes between the two examples. When showing the next example step, the previous "after" code becomes the new "before" code.

We use the Stencil technique [8] to direct students' attention to the example code by enclosing it within a transparent stencil and hiding other components. The example code is runnable but not editable. When asking questions about the code, we used a *scaffolded* self-explanation prompt to help the student reflect on the code because prior work suggests that scaffolded prompts are more effective than open-ended prompts [1]. Students must fill in the blanks to explain what the "before code" does and then how the new parts improve the "after code". Students can run the example code and view its visual output. Then students can reflect on the example code and its output, complete the self-explanation prompt, and click the "Submit" button to go to the next step. After finishing all the WE steps, each piece of example code, commented with its explanation, is provided to the student within the Snap*!* scripting window. Students can then edit the code to build their solution.

2.2 Participants & Procedure

We studied the effects of WEs in Snap*!* on the first 2 days of a 5-day computing summer camp with 22 female high school students, grades 8-9 (ages 13-15). Five students were excluded from the study, leaving 17 students with 6 who identified as African-American, 4 as White, 1 Hispanic, and 6 as multiple ethnicities. About half (9/17) self-identified as having some programming experience and the rest (8/17) as having none or very little.

We adapted 3 drawing-themed programming problems with increasing difficulty from the BJC curriculum [4]. The first problem, Daisy Design, is to write a program to draw a pattern of circles. Its common solution has ~7 lines of code using procedures and loops. The second problem, Spiral Polygon, is to draw a polygon in a spiral shape. Its common solution is 7 ~10 lines of code and uses procedures, loops, and variables. The third problem, Brick Wall, uses ~25 lines of code with procedures, loops, variables, and conditionals. Each problem has a set of goal objectives and several stretch objectives in case some students finished early.

The study used a matched pairs, switching replication design as shown in Table 1 to investigate the impact of WEs on problem 1, and whether or not it matters which problem receives WE support. In Step 0, the instructor led an introduction to programming in Snap*!*, including loops, variables, conditionals, and procedures. In Step 1, students took the pre-test and the pre-survey, and were randomly assigned to groups (E1 receiving WEs on problem 1, E2 with WEs on problem 2), with matched pairs according to pre-test scores. The pre-test, and isomorphic post-test1 and post-test2, were adapted from the block-based version of the Commutative Assessments [25], with 16 multiple-choice questions on variables, loops, procedures, and conditionals. Each topic has four types of questions: code tracing, code comprehension, code completion, and WEs (designed to exactly match PCH WEs). Next, the instructor demonstrated PCH (Step 2) and all the students started problem 1 (Step 3). Camp instructors followed a tutoring protocol to ensure that each student received the same quality and quantity of help, using an audio recorder to capture the dialogue of these help episodes, and record who they helped, the help duration (time), and the type of help (e.g. high-level planning, programming, debugging, interface related). Then students completed the post-test1 and the cognitive load instrument to measure their perceived cognitive load (Step 4). We used the CS Cognitive Load Component Survey (CS CLCS) [11] as our cognitive load instrument. On day 2, students repeated the process for problem 2 (Step 5 ~7). Additionally, they completed problem 3 as a performance post-test, without WEs (Step 8) and filled in post-surveys (Step 9).

During the study, we collected data including participants' interaction with the system (e.g. clicking a button, altering or running their program) and regular snapshots of the students' code with timestamps when they are modified. During the study, students were also able to ask for help from instructors and volunteers, and we also recorded the frequency and duration of the help requests that students made. After the study, we graded students' code to measure the code quality and students' programming performance.

Table 1: Study Outline

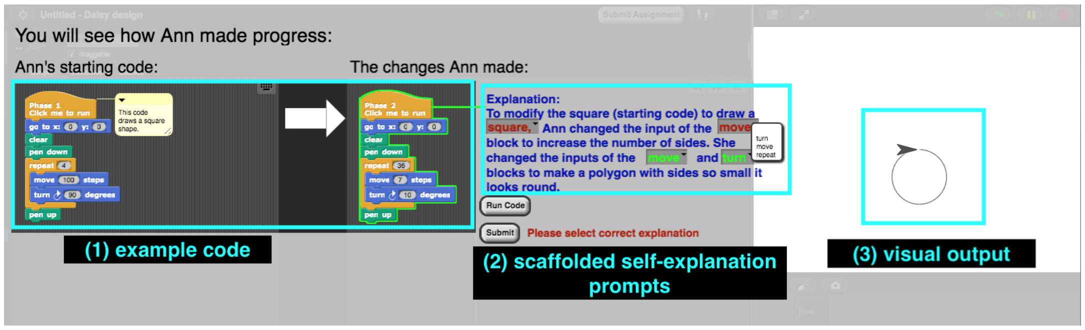
Step	Group E1	Group E2	Time
0	Snap*!* Introduction (taught by camp instructor)		90 minutes
1	Experience pre-survey + Knowledge pre-test		35 minutes
2	Introduce the Peer Code Helper		10 minutes
3	🟩 E1: Problem 1 (WEs)	E2: Problem 1 (no WEs)	45 minutes
4	Post-test1 + Cognitive load survey		25 minutes
Second Day			
5	Re-introduce the Peer Code Helper		5 minutes
6	E1: Problem 2 (no WEs)	🟩 E2: Problem 2 (WEs)	45 minutes
7	Post-test2 + Cognitive load survey		25 minutes
8	Problem 3 (Brick Wall, no WEs)		45 minutes
9	Demographics (post-survey) + Cognitive load survey		15 minutes

3 RESULTS

Our original data included 22 students, 10 in group E1 (receiving WEs on problem 1), and 12 in group E2 (receiving WEs on problem 2). However, five students were excluded from survey and assessment analysis because they were not present for the whole study (2) or encountered errors in Snap*!* that caused them to lose significant programming progress (3) . After excluding these students, there are 9 students in Group E1 and 8 students in Group E2.

3.1 RQ1: Assessments Analysis

For our analysis of pre- and post-tests, we exclude one additional student from Group E1 who did not fill in post-test1, leaving 8 students in each group. Test results for students' pre-test, post-test1 and post-test2 are shown in Table 2. We compared the pre-test scores between groups and found no significant difference ($t(13.96) = -0.4$, $p = .64$, $d = .24$). We then ran a 2x3 mixed ANOVA on students' test scores, with group (E1 v. E2) as a between-subjects factor and test time (pre-test, post-test1, and post-test2) as within-subjects factor. There was a main effect of test time ($F_{2,28} = 5.26$, $p < .05$, partial $\eta^2 = .27$), indicating students' test scores changed significantly from pre-test to the two post-tests as shown in table 2.



You will see how Ann made progress:

Ann's starting code:

```
Phase 1
Click me to run
go to x: 0 y: 0
clear
pen down
repeat 4
  move 100 steps
  turn ↻ 90 degrees
pen up
```

This code draws a square shape.

The changes Ann made:

```
Phase 2
Click me to run
go to x: 0 y: 0
clear
pen down
repeat 36
  move 7 steps
  turn ↻ 10 degrees
pen up
```

Explanation:
To modify the square (starting code) to draw a square, Ann changed the input of the move block to increase the number of sides. She changed the inputs of the move and turn blocks to make a polygon with sides so small it looks round.

turn
move
repeat

Please select correct explanation

Figure 1: Peer Code Helper Interface. Students can run the example code (1), reflect on the output (3), and explain the code (2). Examples are shown side by side. Fill-in-blanks answer choices are marked correct in green or incorrect in red.

We found statistically significant improvements for all students from both pre-test to post-test2 ($t(15) = 3.05$, $p < .01$, $d = .30$) and post-test1 to post-test2 ($t(15) = 2.49$, $p < .05$, $d = .24$), but not from pre-test to post-test1 ($t(15) = -0.86$, $p = .40$, $d = .08$). This suggests that most of students' learning occurred during the second problem.

Table 2: Mean (SD) pre-test, post-test1, and post-test2 scores

	Pre-test	Post-test1	Post-test2
Group E1 (N=8)	8.13 (4.22)	8.38 (3.62)	9.13 (3.83)
Group E2 (N=8)	7.13 (4.02)	7.50 (3.21)	8.50 (4.21)

There was no main effect of group, indicating that there was not a statistically significant difference in mean test score between the two groups ($F_{1,14} = 0.20$, $p = .66$, partial $\eta^2 = .014$). Additionally, there was no statistically significant interaction between group and test time ($F_{2,28} = 0.13$, $p = .88$, partial $\eta^2 = .009$), indicating that students in both groups had similar learning across the study.

We also analyzed the test score for each type of problem (code tracing, code comprehension, code completion, and WE) but found no statistically significant difference on any of the measurements above. To understand the impact of WEs on help requests, we ran an 2x3 mixed ANOVA on the number of help requests, with group as a between-subjects factor and problem as a within-subject factor. We found no significant main or interaction effects ($F_{2,28} = 2.15$, $p = .14$, partial $\eta^2 = .13$), indicating that WEs likely did not impact the number of help requests students made, and nor did the problem, despite increasing levels of problem difficulty.

3.2 RQ2: Cognitive Load Analysis

We used the CS CLCS [11] to measure students' cognitive load. We excluded data from one additional student from Group E1 for this analysis because that student did not answer the survey after finishing problem 3, leaving 8 students in each group. The average factor scores of cognitive load in each category are shown in Table 3.

We ran a 2x3 mixed ANOVA on each type of the cognitive load (CL) scores, with group as a between-subject factor and problem as within-subjects factor. We did not find any significant result for extraneous load or germane load. For the intrinsic load, there was not a significant main effect of group ($F_{1,14} = 0.10$, $p = .76$, partial $\eta^2 = .007$) or problem ($F_{2,28} = 1.78$, $p = .19$, partial $\eta^2 = .11$) factors, but there was a statistically significant interaction *between* group and problem on the intrinsic load scores ($F_{2,28} = 4.65$, $p < .05$, partial $\eta^2 = .25$). This indicates that the two groups experienced different levels of intrinsic CL on different problems, which may be due to the WEs. We investigated this further with post hoc t-tests and found that E1 students' intrinsic load on problem 1 was significantly lower than on problem 2 ($t(7) = -3.51$, $p < .01$, $d = 0.83$), but no such results were found for E2. Similarly, we found E2 students' intrinsic load on problem 2 was significantly lower than that on problem 3 ($t(7) = -4.52$, $p < .01$, $d = 1.04$), but no such results were found for E1. This suggests that both groups experienced lower intrinsic CL on problems *with WEs* than on the following, more difficult problem *without WEs*. Comparing the two groups, we find that the difference in intrinsic load between E1 and E2 is not statistically significant for problem 1 ($t(13.91) = 0.40$, $p = .69$, $d = 0.20$). However, it is significant for problem 2, with a large effect size ($t(13.19) = -2.33$, $p < .05$, $d = -1.16$). One interpretation of these results is that students experienced reduced intrinsic CL when using WEs. This would be consistent with results from work by Gerjets et al., who evaluated "modular worked examples," which, like our WEs, emphasized breaking a problem down into steps [5]. However, another interpretation is that students experienced *higher* intrinsic CL on problem solving that *followed* a WE, which made it seem harder by contrast.

Table 3: Mean (SD) factor score of cognitive load (IL - Intrinsic Load, EL - Extraneous Load, GL - Germane Load)

	Problem 1			Problem 2			Problem 3		
	IL	EL	GL	IL	EL	GL	IL	EL	GL
Group E1	4.3	3.5	6.6	**6.4**	3.3	6.7	5.3	3.4	7.9
N = 8	(2.4)	(3.7)	(3.2)	**(2.4)**	(2.8)	(2.4)	(3.3)	(2.5)	(2.6)
Group E2	4.9	3.4	8.6	**3.8**	2.7	8.0	6.3	4.9	7.6
N = 8	(2.6)	(2.6)	(1.6)	**(1.9)**	(1.7)	(2.1)	(2.8)	(3.5)	(2.3)

3.3 RQ3: Programming Efficiency

Previous research in other domains has shown that replacing problem solving exercises with worked examples (WEs) can increase learning efficiency by decreasing the *time* that students take to complete the exercises without affecting how much they learn. Similarly, RQ3 investigates the impact of WEs on the time taken to complete programming practice problems, or programming efficiency. However, because most students did not complete the *whole* problem within the allotted 45 minutes, we analyze completion time for smaller objectives within each problem.

To investigate students' programming efficiency, two researchers graded all students' submitted code by marking each problem's objectives as either complete or incomplete. For complete objectives, the researchers also marked the time when the student first completed the objective. The three problems (Daisy Design, Spiral Polygon and Brick Wall) had 9, 5 and 5 objectives, respectively. The two researchers reached an inter-rater reliability between 0.744 and 0.979 for first 15 student submissions (5 in each problem). Then the researchers discussed and resolved all the conflicts and graded the rest of student code separately. On the first problem, 2 out of 9 objectives were covered in the WEs (22%) and on the second problem, 1 out of 5 was (20%); we refer to these as "WE objectives." On each problem we removed some students from our analysis who lost their work due to errors in Snap*!*, leaving 9, 10, and 10 students in E1 and 10, 9, and 10 students in E2 on the three problems, respectively.

Figure 2 shows the average number of objectives completed for each group, over time. On both Daisy Design and Spiral Polygon, the group with WEs (E1 and E2, respectively), had more objectives completed at all times throughout the 45 minute work period, while on Brick Wall (with no WEs) there is little difference. These differences between groups is most apparent at the beginning of the curves, when students are completing the WE objectives, though the same trends holds, even if we do not include the WE objectives. On Daisy Design, it took the WE group (E1) a median 3.5 minutes to complete the WE portion of the problem. It took the non-WE group (E2) a median 22.5 minutes to accomplish the same WE objectives by writing code, with 20% of students never finishing the WE objectives in the 45 minutes given. A Mann-Whitney U Test shows that, even ignoring students who did not finish the objectives, this difference in time is significant ($W = 2$; $p = 0.001$). Similarly, on Spiral Polygon, the WE group (E2) took a median 2.5 minutes on the WE portion of the problem, and the non-WE group (E1) took 30.4 minutes, with 40% of students failing to achieve the objectives on 45 minutes. This difference in time is also significant ($W = 54$; $p < 0.001$). It is not surprising that the WE group, who only had to read code and answer questions, finished this content much faster (6-12 times faster), but the magnitude of the difference is worth noting. Based on these results, we can say the WEs saved students lots of time (19-24 minutes on a 45 minute problem), and allowed all students to finish the WE objectives (compared to only 80% and 60% for non-WE students).

However, looking past the WE objectives in Figure 2, we see that the differences between the two groups closes over time. On Daisy Design, the WE group (E1) end with a median 6 objectives complete (67%), compared to 5 objectives (56%) for the non-WE group (E2), and the difference is not significant ($W = 56.0$; $p = 0.382$). On Spiral Polygon, the WE group (E2) ends with a median 2 objectives complete (40%), compared to 1 objective (20%) for the non-WE group (E1), and the difference is not significant ($W = 31.5$; $p = 0.270$). This suggests that WEs may have impacted students' overall performance, but the difference was not significant. One possible explanation for the lack of a significant difference in programming is that while WEs save students time initially, it takes those students additional time to process the WE code they are given, and they work less efficiently at writing code after receiving a WE.

We investigated how the programming efficiency of students who just received WE code compares to those who just wrote equivalent code on their own. Starting from the time each student completed the WE objectives, we calculated how much additional time it took them to complete the next 1, 2 and 3 objectives. Some students finished the WE objectives slowly, or not at all, and ran out of time before they could complete additional objectives. However, for most objectives, enough students finished that we can still calculate the median time for each group (the time at which 50% of students finished and objective) or the first-quartile (Q1) time (when 25% of students finished). To do this, we make the simplifying assumption that the students who completed the WE objectives the slowest would have completed additional objectives *no quicker than* the other students in their group.

Table 4: The first-quartile (Q1) and median (Med) time (in minutes) taken by the WE and non-WE groups to complete 1, 2 and 3 additional objectives after the WE objectives, as well as the proportion that Comp. each objective

		WE Group			Non-WE Group		
	Addt'l Objs	Q1 time	Med time	Comp.	Q1 time	Med time	Comp.
Daisy	WE+1	7.5	9.5	100%	0.0	0.0	80%
	WE+2	14.2	24.8	78%	4.7	11.0	60%
	WE+3	23.3		44%	13.4		30%
Polygon	WE+1	12.0		44%	2.6		50%
	WE+2	13.0		44%	10.4		50%
	WE+3	29.4		44%	>39.6		20%

The results of this analysis are given in Table 4. For Daisy Design, we see that the median student in the WE-group (E1) took 9.5 minutes to complete their next objective, while the median student in the non-WE group (E2) took 0 minutes (they accomplished the next objective at the same time as the WE objectives). This occurred because the next objective was to move code into a procedure, and many students in the non-WE group simply wrote their code in a procedure from the beginning. The next two objectives also take the WE group longer. For Spiral Polygon, no additional objectives were completed by more than half of students, so we can only compute the Q1 time. Similarly, it takes the WE group much longer to accomplish one additional objective (Q1 = 12.0m) compared to the non-WE group (Q1 = 2.6m), but this gap quickly reduces for the second objective, and even reverses by the third (where not even 25% of non-WE of students had completed the third objective by 39 minutes). From this, we conclude that, after completing the WE-supported objectives, students with WEs initially worked less efficiently, especially since non-WE students could accomplish additional objectives *while* completing the WE objectives (e.g. putting code into a procedure). However, it also seems that this difference decreases over time. In addition, the difference never exceeded 15 minutes on Daisy Design, and 10 minutes on Spiral Polygon, which is still far less time than the WEs saved the median student on each

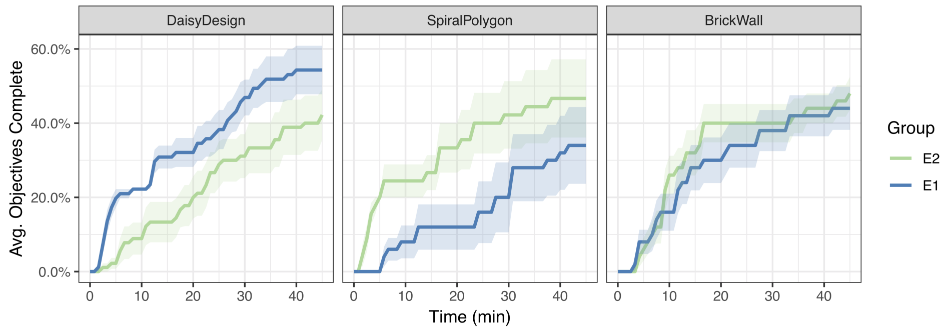


Figure 2: The average number of objectives completed by each group over time, with shading indicating ± 1 standard error.

problem (19 and 24 minutes, respectively). This helps to explain why the WE group accomplished more objectives on both problems, but the difference was not significant.

3.4 Post-survey Feedback

We asked students for feedback on the Peer Code Helper (PCH) at the end of the study. When asked if they would like to use the PCH on a future problem, 35% of students said yes and 12% said no, with the rest uncertain. We found that the 2 students said "no" had very high pre-test scores (over 75%), indicating that some more advanced students may not appreciate WEs, as found in previous work [7]. The participants who were uncertain about the PCH indicated that they preferred the challenge of working independently, saying, "it's good to have a challenge, but it's also nice... to make it a little bit easier.", and "I liked the ... feeling of success from figuring it out for myself, but sometimes the instructions are really helpful when I get stuck." Students' desire for independence is a common reason that they avoid help when programming, and overcoming this barrier can be very challenging [15]. However, many students wrote about how they appreciated the PCH, emphasizing how it helped them to "see how to go from one step to the next" and "know what to do... and... figure it out faster," and that the PCH "gave me somewhere to start coding from." When asked about their favorite parts of the PCH, students specifically mentioned how it gave "comments beforehand to help us understand how to begin the code." Paired with the other results presented here, this indicates that the PCH is an overall improvement to Snap*!*, which students generally prefer over purely independent programming.

4 DISCUSSION & CONCLUSION

We did not find significant differences in learning based on post-test scores. However, results suggest that most of students' learning occurred during problem 2. This may be because students solve problem 2 on second day of the study, after having more time to reflect and digest the concepts learned in problem 1. There was no difference in learning between groups on the WE problems. This is consistent with previous WE studies [10, 17], where students learned more efficiently with WEs, but they did not learn more, since both groups were exposed to the same learning content. The problems in our study contained core objectives (e.g. draw the daisy design), followed by more open-ended objectives (make the design colorful) which do not include new programming concepts. Since most students in both groups completed the core objectives, it is not surprising that there was no significant difference in their learning.

We found significant differences between the groups' intrinsic cognitive load for problem 2 but not for problem 1. It is interesting that we observed differences in *intrinsic* load, which is assumed to be inherent in the learning task, rather than *extraneous* load, which is assumed to be the result of our instructional design. This may be because WEs represent an inherently different learning task than problem solving. We also found both groups experienced higher intrinsic load on problems *without WEs* that followed problems *with WEs*. This may indicate that our design of WEs increased students' perceived difficulty of problem solving immediate following WEs.

Our analysis of programming efficiency suggests that initially, WEs save students considerable time in completing programming objectives (up to half of the time allotted), but that students take longer to complete later objectives. We hypothesize this is because students require time to process the WE code and begin to write their own code. However, the time saved is longer than the time students need to complete additional objectives, which allowed students who received WEs to complete more objectives overall on both problems 1 and 2, although this difference was not significant due to the low sample size.

Threats to Validity: This is an exploratory study with a small sample size, which makes it difficult to rule out possible differences between groups. All of the programming problems involved drawing patterns, and our design of WEs may not generalize to problems that lack visual output. The self-selected summer campers and may be more motivated to learn, but may have taken the assessments less seriously than other student populations. Participants' learning gains may also have come from repeated exposure to tests.

In conclusion, this study evaluates the impact of incomplete WEs, along with scaffolded self-explanation prompts, on learning, cognitive load, and efficiency in block-based programming activities in a high school summer camp. We found that WEs may have an effect on students' intrinsic cognitive load, but further research is needed how and why they have this effect. Our results also suggest that programming WEs may improve students' programming efficiency in the short term, but that students do require additional time to process WEs before they can construct their own code.

Acknowledgements: This material is based upon work supported by the National Science Foundation under grant 1623470.

REFERENCES

[1] Kirsten Berthold, Tessa HS Eysink, and Alexander Renkl. 2009. Assisting self-explanation prompts are more effective than open prompts when learning with multiple representations. *Instructional Science* 37, 4 (2009), 345–363.

[2] Michael E Caspersen and Jens Bennedsen. 2007. Instructional design of a programming course: a learning theoretic approach. In *Proceedings of the third international workshop on Computing education research*. ACM, 111–122.

[3] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research*. ACM, 20–29.

[4] Dan Garcia, Brian Harvey, and Tiffany Barnes. 2015. The beauty and joy of computing. *ACM Inroads* 6, 4 (2015), 71–79.

[5] Peter Gerjets, Katharina Scheiter, and Richard Catrambone. 2006. Can learning from molar and modular worked examples be enhanced by providing instructional explanations and prompting self-explanations? *Learning and Instruction* 16, 2 (2006), 104–121.

[6] Michelle Ichinco, Kyle J Harms, and Caitlin Kelleher. 2017. Towards Understanding Successful Novice Example User in Blocks-Based Programming. *Journal of Visual Languages and Sentient Systems* 3 (2017), 101–118.

[7] Slava Kalyuga, Paul Ayres, Paul Chandler, and John Sweller. 2003. The Expertise Reversal Effect. *Educational Psychologist* 38, 1 (2003), 23–31.

[8] Caitlin Kelleher and Randy Pausch. 2005. Stencils-based tutorials: design and evaluation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 541–550.

[9] Zhongxiu Liu, Behrooz Mostafavi, and Tiffany Barnes. 2016. Combining Worked Examples and Problem Solving in a Data-Driven Logic Tutor. In *International Conference on Intelligent Tutoring Systems*. Springer, 347–353.

[10] Bruce M McLaren, Tamara van Gog, Craig Ganoe, David Yaron, and Michael Karabinos. 2014. Exploring the assistance dilemma: Comparing instructional support in examples and problems. In *International Conference on Intelligent Tutoring Systems*. Springer, 354–361.

[11] Briana B Morrison, Brian Dorn, and Mark Guzdial. 2014. Measuring cognitive load in introductory CS: adaptation of an instrument. In *Proceedings of the tenth annual conference on International computing education research*. ACM, 131–138.

[12] Briana B Morrison, Lauren E Margulieux, and Mark Guzdial. 2015. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the eleventh annual international conference on international computing education research*. ACM, 21–29.

[13] Elizabeth Patitsas, Michelle Craig, and Steve Easterbrook. 2013. Comparing and contrasting different algorithms leads to increased student learning. In *Proceedings of the ninth annual international ACM conference on International computing education research*. ACM, 145–152.

[14] Peter L Pirolli and John R Anderson. 1985. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology/Revue canadienne de psychologie* 39, 2 (1985), 240.

[15] Thomas W Price, Zhongxiu Liu, Veronica Cateté, and Tiffany Barnes. 2017. Factors Influencing Students' Help-Seeking Behavior while Programming with Human and Computer Tutors. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 127–135.

[16] Marguerite Roy and Michelene TH Chi. 2005. The self-explanation principle in multimedia learning. *The Cambridge handbook of multimedia learning* (2005), 271–286.

[17] Ron JCM Salden, Kenneth R Koedinger, Alexander Renkl, Vincent Aleven, and Bruce M McLaren. 2010. Accounting for beneficial effects of worked examples in tutored problem solving. *Educational Psychology Review* 22, 4 (2010), 379–392.

[18] John Sweller. 2006. The worked example effect and human cognition. *Learning and instruction* 16, 2 (2006), 165–169.

[19] John Sweller. 2010. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational psychology review* 22, 2 (2010), 123–138.

[20] John Sweller and Graham A Cooper. 1985. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and instruction* (1985).

[21] John Sweller, Jeroen JG Van Merrienboer, and Fred GWC Paas. 1998. Cognitive architecture and instructional design. *Educational psychology review* 10, 3 (1998).

[22] J Gregory Trafton and Brian J Reiser. 1993. Studying examples and solving problems: Contributions to skill acquisition. In *Proceedings of the 15th conference of the Cognitive Science Society*. Citeseer, 1017–1022.

[23] Jeroen JG Van Merriënboer and Marcel BM De Croock. 1992. Strategies for computer-based programming instruction: Program completion vs. program generation. *Journal of Educational Computing Research* 8, 3 (1992), 365–394.

[24] Arto Vihavainen, Matti Paksula, and Matti Luukkainen. 2011. Extreme apprenticeship method in teaching programming for beginners. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 93–98.

[25] David Weintrop and Uri Wilensky. 2015. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs.. In *ICER*, Vol. 15. 101–110.

[26] Rui Zhi. 2018. Exploring Data-driven Worked Examples for Block-based Programming. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 294–295.