Investigating the Impact of On-Demand Code Examples on Novices' Open-Ended Programming Projects

Wengran Wang[*]
North Carolina State University
Raleigh, USA

John Bacher
North Carolina State University
[†]Currently at Apple Inc.
Raleigh, USA

Amy Isvik
North Carolina State University
Raleigh, USA

Ally Limke
North Carolina State University
Raleigh, USA

Sandeep Sthapit
North Carolina State University
Raleigh, USA

Yang Shi
North Carolina State University
Raleigh, NC, USA

Benyamin T. Tabarsi
North Carolina State University
Raleigh, USA

Keith Tran
North Carolina State University
Raleigh, USA

Veronica Catété
North Carolina State University
Raleigh, USA

Tiffany Barnes
North Carolina State University
Raleigh, USA

Chris Martens
North Carolina State University
Raleigh, USA

Thomas Price
North Carolina State University
Raleigh, USA

ABSTRACT

Background and Context: Open-ended programming projects encourage novice students to choose and pursue projects based on their own ideas and interests, and are widely used in many introductory programming courses. However, novice programmers encounter challenges exploring and discovering new ideas, implementing their ideas, and applying unfamiliar programming concepts and APIs. Code examples are one of the primary resources students use to apply code usage patterns and learn API knowledge, but little work has investigated the effect of having access to examples on students' open-ended programming experience.

Objectives: In this work, we evaluate the impact of code examples on open-ended programming, through a study with 46 local high school students in a full-day coding workshop.

Method: We conducted a controlled study, where half of the students had full access to 37 code examples using an example browser system called Example Helper and the other half had 5 standard, tutorial examples.

Findings: We found that students who had access to all 37 code examples used a significantly larger variety of code APIs, perceived the programming as relatively more creative, but also experienced a higher task load. We also found suggestive evidence of a better post-assignment performance from the example group, showing that some students were able to learn and apply the knowledge they learned from examples to a new programming task.

[*]Currently at Apple Inc.

CCS CONCEPTS

• **Human-centered computing → Human computer interaction (HCI)**; **User studies**; **Empirical studies in HCI**.

KEYWORDS

open-ended programming, code examples, block-based programming, novice programming

ACM Reference Format:
Wengran Wang, John Bacher, Amy Isvik, Ally Limke, Sandeep Sthapit, Yang Shi, Benyamin T. Tabarsi, Keith Tran, Veronica Catété, Tiffany Barnes, Chris Martens, and Thomas Price. 2023. Investigating the Impact of On-Demand Code Examples on Novices' Open-Ended Programming Projects. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1 (ICER '23 V1), August 7–11, 2023, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3568813.3600141

1 INTRODUCTION

Open-ended programming projects encourage students to build artifacts of their own design, and are widely used in many introductory programming courses (e.g., [15, 17, 18]). However, novice programmers encounter many challenges when building open-ended projects, as they may be unsure what to make or what ideas are appropriate for the programming environment [50], or lack knowledge of the programming concepts and APIs necessary for implementing an idea [18].

Many students use code examples to incorporate API usage patterns into their open-ended projects, for example, by searching for them on the web and "patching them together" into their own code [24], or by "remixing" an existing project [25]. Instructors also sometimes provide a set of examples for students when giving an open-ended project assignment [29]. Prior work suggests examples may support students in open-ended projects by helping them understand what is possible and doable, and introducing new APIs they could use to get started [7, 47]. Some systems have been built to support example use during programming [22]. However, few studies have been conducted to study the effect of having access to these examples during open-ended project-making. While examples have been studied on closed-ended problems with instructor-defined goals [44, 48, 55], supporting open-ended programming represents a new challenge because they inherently include unique requirements for students, such as to express their ideas creatively [20], and to learn new concepts and APIs during the process of project-making [16]. Without experimental data, it is unclear whether having access to code examples helps students achieve these requirements.

In this work, we investigated three Research Questions: to what extent did having access to code examples affect students' RQ1) project complexity, RQ2) self-reported task load and creativity, and RQ3) learning outcomes. We ran a controlled study in a full-day coding workshop, with students recruited from local high schools. The examples were provided through an example support system called Example Helper [53], on which students can search, select, test, and use an example from a gallery of code examples. The experimental group had access to 37 on-demand examples in Example Helper, introducing new API usage patterns, while the control group had no access to novel examples[1]. During the workshop, students experienced an introduction to Snap*!*, and then complete a 2.5-hour open-ended programming activity, with pre and post-tests. The primary contribution of this work is a formal controlled study about the impact of code examples on novices' open-ended programming.

2 RELATED WORK

2.1 Open-Ended Programming

In this work, we use the term "open-ended projects" to mean projects where students partially or fully design the programming artifact they create, where they set their own goals that are meaningful to them [18]. This is in contrast to more traditional, "closed-ended" assignments, where students are given tasks with clear, defined goals, often accompanied by step-by-step instructions [13]. The interest-driven coding experience of open-ended projects has been shown to provide a sense of ownership and self-confidence in various engineering, science, and design classrooms [3, 11, 33]. In computer science classrooms, according to a survey on 134 introductory programming instructors in the UK, many instructors found that the creative experience of building an app or game by themselves, makes computing more interesting to novice students [5]. The BJC curriculum designers also highlight that through open-ended projects, students learn the important CS concepts and APIs "not by doing exercises on them, but by building things that need them", which sparks creativity and a sense of self-confidence [16].

However, multiple challenges arise when students freely choose what they want to build. Students may encounter difficulties exploring or generating ideas and may choose expedient or simple solutions, preventing them from committing to thoughtful work [24, 26]. In addition to the need for ideation, novices also encounter barriers, such as difficulties in applying knowledge of programming concepts into code implementation [18] and difficulties understanding and reusing unfamiliar APIs [25].

To overcome these challenges, some instructors provide students with a set of example programs when building open-ended projects, as examples offer new ideas, and demonstrate how unfamiliar APIs and idioms can be used [24, 29]. However, little work has investigated the impact of having access to these examples on students' open-ended programming. Additionally, although an example system can help students to better explore and integrate new ideas, example systems for open-ended programming has been relatively scarce – as students freely choose and define their own goals, it is difficult to anticipate content to offer to the students [45, 50].

2.2 Code Examples

Code examples demonstrate a sample solution for a programming task or subtask. Most frequently, they are used in closed-ended programming tasks, but recent work has explored open-ended settings as well.

2.2.1 Using code examples on closed-ended programming tasks. The "worked example effect" from Cognitive Load Theory [43] explains how examples support learning: reading and understanding a step-by-step demonstration of a problem (a "worked example" [43]) prior to solving a related problem helps learners to complete the latter problem more efficiently. From a theoretical perspective, the worked examples reduce the amount of mental resources needed for searching for a problem-solving schema, and therefore retain more mental resources for processing and learning the material [43]. Many prior studies have found that learning a worked example prior to solving a similar problem led to higher learning gains than solving both problems from scratch [23, 41, 44]. A meta-analysis conducted by Crissman on 62 studies across various disciplines found that learning a worked example prior to problem-solving produced an overall moderate, positive effect on students' final learning outcome [10]. These results also hold for programming worked examples. For example, Trafton and Reiser experimented with teaching undergraduate students Lisp programming by interleaved examples and solving an isomorphic problem, and found that compared to the group with only problem-solving pairs, the interleaved group achieved higher post-test scores [44]. These studies show strong evidence of the effectiveness of using worked example examples [43] with step-by-step instructions, presented before students solve a programming problem.

However, many students – especially when working on a programming problem, also use what we call *on-demand examples*, such as those found on the internet [14], in course resources (e.g., textbooks), or from an on-demand example system (discussed below). Unlike worked examples, students seek this help *during* problem-solving, and these examples may not have step-by-step instructions. However, less research has evaluated the effectiveness of these "on-demand" examples - and almost none in the context of open-ended programming. The Peer Code Helper [56] is an on-demand example system in Snap*!*. The system provides code examples as a sample solution of the same task, when students complete a programming assignment in Snap*!*. A study with 22 high school novice students

[1]To keep conditions comparable, the control group did have access to Example Helper, but it only provided 5 examples from an earlier tutorial. See Section 3 for details.

showed that these examples help students solve the task quicker than the group without these examples. However, unlike the learning effect commonly seen with worked examples, the two groups' post-test performances were not different [56]. Ichinco and Kelleher augmented the Looking Glass system [37] to include an example dialog, which presents code examples for a similar problem when students work on a closed-ended program completion task [21]. In a formative study with 9 pairs of children aged 10-15 using the example system, Ichinco and Kelleher found that students encountered difficulties understanding the examples and connecting the examples to their own tasks. Some students also felt distracted by looking at the examples [21]. This raises the question of whether the learning effect found from the structured learning of a worked example can transfer to the on-demand example learning scenarios, as students primarily focus on solving their current problem, and examples from a different context introduce additional barriers.

2.2.2 Using code examples for open-ended programming. Example use during open-ended programming is very different from that during closed-ended programming – unlike learning an example for solving a closed-ended programming problem, prior work has shown that example use during open-ended programming is usually "opportunistic" [7, 24], with the primary goal of saving time [6, 27, 39, 40]. For example, Rosson and Carroll investigated Smalltalk programmers' example use, and summarized a typical reusing process as "getting something to work with" by directly copying an example to their own workspace, and then testing the example code and "debug into existence" [39].

Prior work analyzing the phenomena of example use during open-ended programming has been focused on how examples were found and reused from the web (e.g., [7, 14, 25]. Specifically, prior work has noted a number of challenges programmers face when using examples during open-ended programming. For example, an analysis by Gao et al. on 18 programmers' web use during React programming found that programmers heavily relied on searching for and copying code examples from the web, but more than half of the examples were later deleted; with most of the kept examples modified to fit in programmers' own programs [14]. Analysis of Scratch's "remixing" projects also found that novices often misuse or ignore sophisticated APIs (e.g., cloning and procedures) when remixing an example project, and experience difficulties to understand example code and connect them with their own goals [2, 25]. These challenges show that while on-demand examples are commonly used for programmers, it is still unclear whether using examples can lead to more advanced API use, or better project-making outcomes for novice programmers.

We found only one controlled study, conducted by Ichinco et al. on the Example Guru system [22], to have evaluated the impact of having access to code example use during *open-ended* programming. The Example Guru uses pre-defined rules to find and prompt an example to the student during open-ended programming. In a study with 78 children aged 10-15, Ichinco et al. asked students to use 30 minutes to create two animations in Looking Glass, and found that students in the Example Guru group accessed more examples compared to another group that uses documentation, and applied more APIs from those examples into their own code [22]. Going beyond this work, we measure the impact of example use in a longer (3 hours), ecologically valid learning setting (a coding workshop classroom), where students made a project from start to finish, which they had the freedom to design and plan themselves. Further, we not only investigate the extent of API use from code examples, we also investigate how access to examples affects the complexity of students' projects, their post-task performance, and students' perceptions of task load and creativity.

3 METHODS

3.1 The Example Helper System

To investigate the impact of students' example use during open-ended programming, we aim to use an example system that allows students to browse, search, test, and copy examples from a set of different examples, which resembles how students search for and reuse examples on the web [7, 14]. Therefore, we chose the Example Helper system [53] from our prior work [53]).

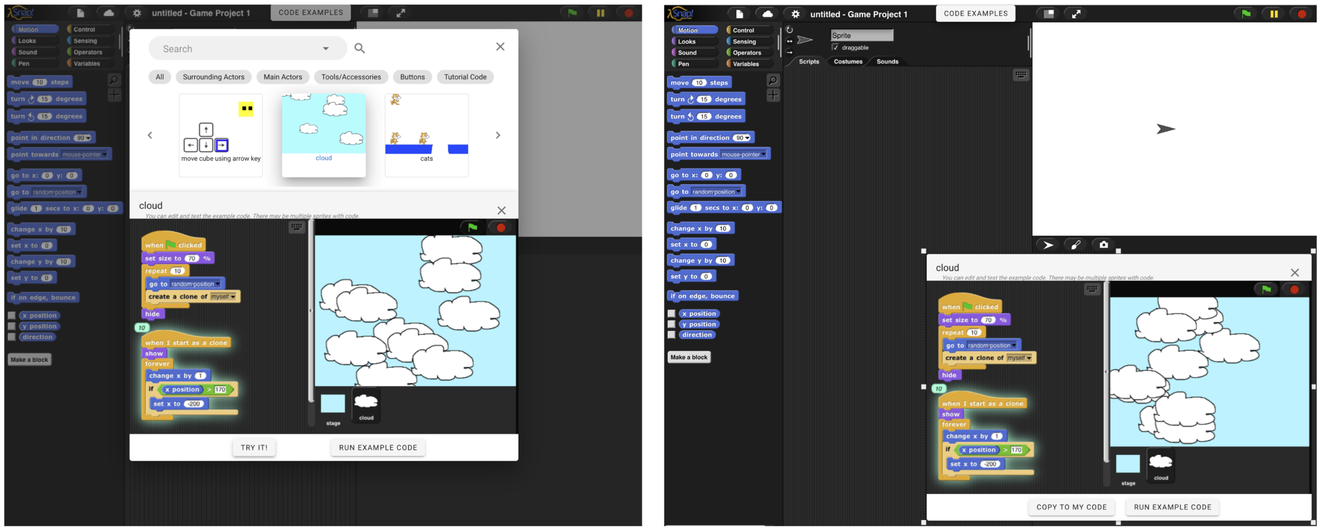
The Example Helper is an example gallery developed as an extension of Snap*!* [32], a block-based novice programming environment (Figure 1). When programming in Snap*!*, a student can click on the "show example" button at the top-center of the Snap*!* window to open a gallery of code examples (Figure 1-a). They can use the search box, or select different tags to find an example they need. They may also simply click through the left and right arrows to browse through different code examples, represented as gif animations. To inspect the code, they may click on the gif to see a sandbox window, shown in the lower half of Figure 1). Just like a standard Snap*!* window, the student may view code from different sprites (i.e., actors), edit the code, and run it to see its output on the right of the preview window. If they click on the "try it" button on the lower right of the preview window, a new sandbox will show up (Figure 1-right), where students can continue to work with the example, or click on the "copy to our code" button to copy the example code to their own code. A qualitative analysis on the log data of 46 undergraduate students using Example Helper has found that the system affords a variety of exploration and experimentation behaviors, showing that the system is suitable to be used to support open-ended programming.

3.2 Participants & Learning Context

To provide an authentic learning context, where students had enough time to build a complex project, we conducted a full-day coding workshop, which was repeated on 3 consecutive Saturdays with different participants. During each day, students learned Snap*!* programming and built an open-ended project.

3.2.1 Participants. We recruited 46 participants from multiple high schools from a state in the Southeastern region of the United States. Among them, 21 are in the control group, 25 are in the example group. The conditions are randomly assigned based on participants' pre-test scores, which we discuss in Section 3.2.3. The participants include 14 Females (6 in the control (C), and 8 in the example (E) condition) and 31 Males (14C, 17E[2]), and 1 (IE) who preferred not to disclose. The 46 students were split over each of the 3 consecutive Saturdays. There were 18 White (7C, 11E), 13 African American (6C,

[2]14C, 17E means there were 14 students in the control (C) group, and 17 in the example group.



```
when [flag] clicked
set size to 70 %
repeat 10
    go to random position
    create a clone of myself
hide

when I start as a clone
show
forever
    change x by 1
    if x position > 170
        set x to -200
```

Figure 1: The Example Helper interface [53]. Students can browse and search for examples in a gallery interface, and then test and modify them in a sandbox. They may also copy the example into their own workspace.

7E), 10 Asian (5C, 5E), 3 Hispanic or Latino (1C, 2E), 1 Caribbean Islander (IC), and 1 American Indian or Native Alaskan students (1C). Participants reported various levels of prior CS experience: 8 reported having no prior programming experience (5C, 3E), 22 reported having completed a few programming tutorials before (10C, 12E), 8 reported having taken one programming course (2C, 6E), and 8 reported having taken at least 2 programming courses (4C, 4E). We recruited participants from diverse prior experiences, as open-ended programming projects are common at all levels.

3.2.2 Learning Goals. To prepare students for building complex open-ended programming projects and to allow all students with different levels of background to learn the basics of Snap*!* and programming, we designed the workshop to cover a variety of CS concepts and Snap*!* API use-cases, that are useful to build visual, interactive programs

Specifically, the Snap*!* environment features building visual, interactive programs (such as games and apps) and commonly includes multiple sprites[3] that can interact with each other over time. Therefore, apart from general CS concepts such as loops and conditionals, our learning objectives also include more advanced Snap*!* concepts such as concurrency, which can be implemented in Snap*!* by using multiple scripts[4]; and Snap*!*-specific APIs such as cloning, which can create multiple copies of a sprite.

The curriculum and assessments for the coding workshop were developed around the following 3 learning goals, divided into 12 subgoals: 1) CS Concepts: understand and apply knowledge of basic CS concepts, including loops, conditionals, variables, and concurrency. 2) Understand and apply the various concepts and usage of Snap*!*'s cloning APIs, including: clones are copies of their parent sprite; clones inherit properties of the parent sprite when they are created; scripts under the header block "when I start as a clone" can run concurrently for each individual clones after they are created; a clone can be deleted using "delete this clone". 3) Understand and use other important Snap*!* APIs correctly, including message passing, randomness, appearance (e.g., graphic effects), touching, and stopping the Snap*!* program, we applied these learning goals to design the example content, introductory materials, and the pre and post-tests, discussed below.

3.2.3 Conditions. Participants were divided across three days, and during each day, a random half of the students were assigned to the control condition, while the other half were assigned to the example condition. To do the random assignment, we used an algorithm, which was administered each camp day after students had completed the pre-test. The algorithm generated a fully random assignment, and then re-generated the random assignment if the differences between the mean pre-test scores of the two groups exceeded a threshold. This meant that each student was equally likely to be assigned to each group, while ensuring pre-test scores were roughly equivalent in each group. This random group assignment mechanism was integrated directly into the ExampleAid programming system, so that the instructors and researchers in the classroom were unaware of which group a specific student was assigned to, to ensure the same learning experience for both groups. However, this mechanism did not ensure equal distribution of gender and race, as the groups are assigned randomly.

To understand students' example use in an authentic learning setting, conditions were assigned non-intrusively: all students had access to the Example Helper system throughout the day. In the morning, all students were introduced to Snap*!* programming and the Example Helper system; in the afternoon, students design and build an open-ended project in Snap*!*. All students had access to 5 "tutorial examples" throughout the day, which were used for introducing students to Snap*!* and the Example Helper system. However, the control group only had access to these 5 tutorial

[3]Sprites in Snap*!* are similar to objects in object-oriented programming environments.
[4]Scripts refers to a set of connected code blocks in Snap*!*.

examples, while the example group (i.e., Example Helper group) an additional set of 32 examples, with various different contexts [5].

3.3 Procedure

We followed the same procedure for each of the three Saturdays, we describe the timeline of one day's coding workshop below:

Morning: Tutorial and pre-test. Students arrived at the classroom at 9:30 am. After completing the pre-survey, they learned a 2-hour Snap*!* tutorial. The goal is to help students learn the 12 sub-learning goals, and become familiar with the Example Helper interface. Students completed a scaffolded closed-ended programming exercise, while making use of 5 tutorial examples from Example Helper. Afterward, students completed a 15-minute pre-test, which includes 12 questions, each corresponding to one of the 12 sub-learning goals. Afterward, they do a 30-minute lunch break.

Afternoon: Programming, post-survey, post-test, and post-assignment. Students planned and programmed an open-ended project in the afternoon. They first learned the project requirements: they should build a game with the theme "Animals and Nature", and follow requirements based on 2 dimensions: A) Game Experience dimension: they should build a complete game, that has 1) starting and 2) ending mechanism; 3) actor and 4) user interactions; 5) obstacles; 6) actors can gain score or lose life; and 7) artistic details (e.g., scrolling backdrop or graphical effects). B) Applying Programming Knowledge dimension – they should apply the programming knowledge and APIs they have learned in the morning (i.e., the 3 key learning goals) to their final projects.

After learning the requirements, students did a 20-minute planning session, where they describe the key mechanics they wanted to include in their projects.

After planning, students did open-ended programming in Snap*!* for 2 hours and 30 minutes. As discussed in Section 3, the control group had access to 5 tutorial examples, while the example group had access to all 37 examples. After programming, students were asked to use 30 minutes to complete a post-survey, a post-test, and a post-assignment. The post-survey includes questions on task load and creativity; the post-test includes 12 isomorphic questions as the pre-test; and the post-assignment (called "Catch The Egg"), is a close-ended task that combines knowledge from the 12 sub-learning goals, but is different from any of the code examples. The assignment asked students to build a game that has clones of eggs falling, and a cart at the bottom to catch the egg and increase their scores. The game should stop after a certain amount of time, and should print the user's final score.

3.4 Example Content Design

As explained in Section 3.3, all students had access to 5 tutorial examples, while the example group had access to an additional set of 32 examples. These examples were designed based on two main goals: 1) They needed to be self-contained, small in size, and demonstrate programming patterns commonly seen in novices' open-ended projects [46]. 2) Each example was intended to demonstrate one or more learning goals discussed in Section 3.2. Now, we will discuss the design of the 5 tutorial examples and the 32 additional example content:

3.4.1 5 Tutorial Examples. The goal of the 5 tutorial examples was to: 1) be used as demonstration and learning materials for the 2-hour Snap*!* tutorial; and 2) help students become familiar with the ExampleAid interface. During the tutorial task, students followed the instructors to complete a simple game where they could control a main character to travel through the stage and destroy multiple copies of enemies. These tutorials were built as 5 tasks. For each task, students were asked to learn, modify, and expand on one example to complete the task. The examples were themed around three sprites: a main character cat, a geometric shape representing an enemy, and a "start game" button that a user could press to start the game.

Each example-based learning task was designed following the Use-Modify-Create progression [30]. Students first used the example by following the instructor to change some parameters and selected blocks, and ran the example to understand how it worked (Use). Students could then modify the example based on a small goal that the instructor demonstrated (e.g., in Example 1, students were asked to change from animating the cat 3 times to animating the cat non-stop). Afterwards, students made discoveries for open-ended learning goals (e.g., in Example 1, students were asked to explore how to make the cat walk). At the end, the teacher showed students a working version of the final program in the digital handouts, which students could use to compare with their own program.

The 5 tasks started with using a loop to animate and walk the cat (Task 1), using conditionals and variables to change the cat's speed on key press (Task 2), and using concurrency to stop the cat when touching the wall (Task 3). Next, the students needed to use cloning to generate a set of enemies that went to random positions on the stage. When a user controlled the cat to touch an enemy, the enemy gradually disappeared (Task 4). Finally, students needed to program the game using message passing so that the game started when a player pressed a "start game" button and stopped when all enemies were destroyed (Task 5).

To allow space for "Create," the examples for each task did not include the final step. For example, in Example 1, the cat could change costume but could not walk non-stop. In Task 4, the enemy could show up at random positions, and the user could control the cat, but students needed to learn and program the cat-enemy interaction themselves. As a result, while all learning goals were introduced in the tutorial and the code for completing each task was included in the digital handouts, the 5 tutorial examples in the ExampleAid system only included 8 of the 12 learning goals (4 basic, 2 cloning, 2 other APIs).

3.4.2 32 Additional Examples. The 32 additional examples utilized various themes and characters to demonstrate the 12 learning goals. For instance, instead of using geometric shapes that appeared randomly, one example demonstrated cloning and randomness by enabling a group of bubbles to slowly follow mouse movement while changing transparency. All 12 learning goals were represented in at least one example, with many of them demonstrated by multiple examples.

[5]For example, Figure 1 shows an interface for an example group student – all tags had relevant code examples underneath the search bar. However, the control group only had code examples underneath the final tag "tutorial code".

3.5 Measures

We collected measures to evaluate students' project complexity (RQ1), their self-reported task load and creativity (RQ2), and outcomes on learning (RQ3), discussed below.

Pre-test. We designed the pre-test questions based on the 12 sub-learning goals under the 3 learning goals. Each question corresponds to one learning goal. In specific, the 4 CS concept questions were adapted from sample questions from the CS Concept Inventory [38]. Across both conditions, their pre-test score had a mean score of 7.11 (std = 2.48, min = 2, max = 11), meaning that an average student completed around 60% of the questions correctly. The pre-test scores had no significant difference between groups (p = 0.57) - The control group had 21 students, with mean = 7.0 (std = 2.035), and the experiment group had 25 students, with mean = 7.2 (std = 2.75). This suggests that the two groups had prior knowledge, which is consistent with students' pre-survey responses, indicating 82.6%(38/46) had at lease some level of prior programming experience.

Example usage statistics. The Example Helper system tracks log data on all actions students perform with examples, such as opening an example, or copying an example to their own code. Based on the log data, we collected the timestamps and the content of the examples the students have requested, we next compared the examples students requested and their final submitted project, to inspect how examples were integrated into students' projects. 44 among the 46 students have submitted their projects (20 in control, 24 in experimental)[6]. For consistency of comparison, our analysis of example usage statistics focuses on the 44 students.

Project complexity score (RQ1). To evaluate the complexity of the 44 submitted students' projects, we constructed a grading rubric with three dimensions: 1) Game Experience, which calculates how many of the 7 requirements (Section 3) a student's project achieved, to measure whether a student's game is complete and interesting. 2) Applying Programming Knowledge, which calculates how many of the 12 sub-learning goals were applied in a student's project. We translated the 12 sub-learning goals into loose patterns found in students' code. For example, when a student's project has at least one loop statement, we grade the "the loops concept" sub-goal as complete. 3) Number of Game Mechanics, which calculates the number of game mechanics students implemented successfully in their projects. To define "mechanics", we followed the approach by Cateté et al. on developing scientific rubrics [8]. Two researchers independently inspected 10 projects to generate a list of mechanics, each of which describes a distinct behavior of the game (e.g., an actor can move with an arrow key), that can be completed using a combination of programming concepts and Snap*!* APIs. They next discussed, merged, and verified the mechanics, generating a list of 26 mechanics. Next, the two researchers used the mechanic list to each grade 17 student's projects, and they adjusted the mechanic list to include extra mechanics, generating a final list of 37 mechanics. Based on the above grading rubric, an average student achieved 8.6/12 programming knowledge scores (std = 3.30), 4.77/6 game requirements (std = 2.20), and 6.57/37 game mechanics (std = 8.65). For each student, the overall project complexity score is calculated as the sum of the three scores (Game Experience, Applying Programming Knowledge, and Number of Game Mechanics), each score scaled to the range 0–1.

# Blocks (RQ1). A simple but common measure of project complexity is the size of the program (e.g., [1]). Therefore, for each student's submission, we calculated the number of blocks to indicate the size of the project, and the number of distinct blocks in students' projects to indicate the variety of APIs a student applied in their project.

Use rate difference (RQ1). One of the goals of Example Helper is to introduce students to new APIs and programming concepts that are relevant to their project goals, which students may not have previously seen. Therefore, we compared the usage of specific blocks and mechanics between the control and example group. For each block, we define the "use rate" as the proportion of students who used this block or mechanic in their project. For example, 66.7%(16/24) example group students and 20% (4/20) control group students used the block "delete this clone" at least once in their projects, so the block's use rate is 66.7% in the example group, and 20% in the control group. To compare the preference of block usage, we sorted the blocks based on the **"use rate difference"**, which indicates the use rate of a certain block in the example group, minus the use rate of the block in the control group. Therefore, a positive/negative use rate difference indicates that the block was more/less popular among the example group than the control group. Similarly, the use rate difference was also calculated for each of the Game Mechanics collected from the project complexity measurements.

Self-reported task load (RQ2). Prior work suggests that examples can reduce the mental effort necessary to learn programming [34], but this has not been explored for on-demand examples, which may play a different role. We therefore measured the task load based on students' 5 Likert-scale responses to the 6 NASA-TLX questions [19]:

(1) (Mental Demand) How mentally demanding was the task?
(2) (Physical Demand) How physically demanding was the task?
(3) (Temporal Demand) How hurried or rushed was the pace of the task?
(4) (Performance) How successful were you in accomplishing what you were asked to do?
(5) (Effort) How hard did you have to work to accomplish your level of performance?
(6) (Frustration) How insecure, discouraged, irritated, stressed, and annoyed were you?

The total task load score was calculated by the sum of scores from all 6 questions. All 46 students completed the task load survey, with an average total score of 18.02 (std = 3.03).

Creativity (RQ2). Another goal of on-demand examples is to enable students' creativity, both by supporting their existing goals, and suggesting new ideas they might pursue. We measured students' self-reported creativity based on students' 5 Likert-scale responses to 5 CSI questions [9][7]:

(1) (Result-Worth-Effort) What I was able to produce was worth the effort I had to exert to produce it.

[6]Two students chose to not submit their projects. As a voluntary coding workshop, we did not oblige students to submit their final projects if they chose not to.

[7]The last "collaboration" question was excluded, as students worked independently.

(2) (Expressiveness) I was able to be very expressive and creative while implementing our ideas.
(3) (Exploration) It was easy for me to explore many different ideas, options, designs, or outcomes during programming.
(4) (Immersion) Our attention was fully tuned to the programming activity, and I forgot about the system/tool that I was using.
(5) (Enjoyment) I was very engaged in the programming activity - I enjoyed this activity and would do it again.

The total creativity score was calculated by the sum of ratings for all 5 questions. All 46 students completed the CSI surveys, with an average score of 17.52 (std = 3.95).

Learning (RQ3). We measured learning in two ways: 1) a programming post-assignment; and 2) the normalized learning gain (NLG) [31] calculated from pre/post tests.

Because Example Helper is designed to help students learn to write open-ended programs (rather than trace or complete them), our primary measure of learning was a programming post-assignment. Since students in the control and experimental conditions had comparable performance on their pre-tests (see above), any difference in performance on this post-assignment was likely due to their condition. After they created their open-ended project (where the intervention took place), students were asked to complete an assignment called "Catch The Eggs," a small, closed-ended project with specific instructions and goals. Neither group had access to examples during this assignment, and we gave them a fixed 30 minutes to complete it. The goal of the post-assignment is to assess whether students who had access to examples learned from them, and can apply the knowledge more effectively, than the control group students. The post-assignment covered all 12 learning objectives discussed in Section 3.3-Learning Goals, which were introduced during the morning tutorial, and were also covered by the 32 additional examples (e.g., generating clones of a sprite, and start each clone at a random position). To avoid ceiling effects, the task was designed to be hard to complete entirely during that time. We, therefore, score the assessment as the number of objectives completed in that fixed time. Across both conditions, the mean is 3.84, meaning that students completed 3.84 / 8 objectives on average.

Note that the examples demonstrating each learning goal were distinct from the implementation required to complete the post-assignment. For example, an example that covers all 4 sub-goals under "cloning" includes a group of clouds moving left to right, disappearing when the cloud touches the edge, then reappearing from the left. However, in the catch-the-egg post-assignment, the 4 cloning concepts were tested by students programming the eggs falling from the top of the screen, and disappearing/reappearing when touching the shopping cart.

We also measured learning based on the changes between students' pre-test to post-test scores. We designed the post-tests to be isomorphic to the pre-test (12 questions), and calculated NLG as $(\text{post} - \text{pre})/(12 - \text{pre})$, according to [31]. Students' post-test scores had an average (mean = 6.67, std = 2.86, min = 0, max = 12), which was lower than the pre-test, which has an average of 7.11 (std = 2.48, min = 2, max = 11). Consequently, the NLG is negative (mean = -0.18, std = 0.70, min = -2.5, max = 1). This is likely due to the fatigue of the full-day coding workshop – 47.8% (22/46) students' scores dropped during the post-test, with 6 students dropping at least 3 points, and 7 dropped to lower than or equal to a score from random guessing (post-test score <= 3). In addition, instructors anecdotally reported off-task behaviors during the post-test, showing that many students chose not to meaningfully engage with the post-test, and randomly guessed their answers. For completeness, we still present the measure; however, it is likely not reflective of differences in learning between conditions. We discuss the implication of this result in Section 5.

4 RESULTS

We first briefly discuss the descriptive statistics for example use, and then present the results based on the research questions from Section 1 – investigating how code examples may impact 1) students' project complexity, 2) their self-perceived Task Load and Creativity, and 3) learning outcomes. We compare the control and Example Helper (example) groups' data based on the measures discussed in Section 3.5. We first use Shapiro-Wilk test to test sample normality; then we compare the corresponding measures between groups using Mann-Whitney-U test for non-normal samples, and Student's t-test for normal samples[8]. We conducted 14 total statistical tests.[9]. We present our findings below.

4.1 Descriptive statistics

We first collected data on example requests and integrations. We found that 1) most students from the example group requested (83.3%, 20/24) and integrated (79.8%, 17/24) at least one example. On average, example group students opened 2.76 examples (min = 0, max = 13, std = 3.07); and integrated 1.42 examples into their final code (min = 0, max = 5, std = 1.35). 2) Fewer students from the control group (40%, 8/20) have requested and integrated at least one example (25%, 5/20), and they requested and integrated fewer examples too: On average, control condition students opened 0.9 examples (min = 0, max = 4, std = 1.22); and integrated 0.3 examples into their final code (min = 0, max = 2, std = 0.55). Likely due to the greater variety of examples, the example group had a significantly higher number of example requests (p = 0.01, d = 0.82) and integrations (p = 0.001, d = 1.05). This suggests that any differences in the outcomes of these groups, reported later, were likely due to their use of examples, and our choice to limit the control condition to 5 tutorial examples served its purpose. When integrating examples, students typically build features on top of it by removing parts of the example code and adding their own project-specific code. Among the integrated examples, students in both groups removed on average 34.4% (min = 0, max = 77.8%, std = 0.24) of the example code, and added 60.5% (min = 0, max = 828.6%[10], std = 1.36) of their own code on top of it.

[8]The effect size based on the Mann-Whitney-U test is reported as r value, while the effect size based on the Student's t-test is reported as Cohen's d value.
[9]While our analysis includes multiple comparisons (e.g., project complexity score, task load, and Learning), we follow suggestions from [36] to report original p-values, as well as the number of tests, and discuss its implications, rather than adjusting alpha values.
[10]The maximum amount of added code is 828.6%, as some students build more complex code than the example.

Table 1: Project score statistics (RQ1) for the control and example groups, with the p-value and effect sizes (r or d).

1. project complexity

	game experience		applying knowledge		# game mechanic		total score		# blocks*		distinct # blocks*	
	control	example	control	example	control	example	control	example	control	example	control	example
n	20	24	20	24	20	24	20	24	20	24	20	24
mean	0.65	0.71	0.65	0.72	0.48	0.53	1.78	1.96	87.3	101.04	23.2	28.42
std	0.31	0.31	0.3	0.29	0.27	0.28	0.84	0.83	47.15	50.17	7.86	7.84
p-value		0.54		0.38		0.55		0.48		0.37		0.04
effect size		r=0.09		r=0.13		r=0.09		r=0.11		d=0.28		d=0.65

4.2 RQ1: Project Complexity

RQ1: How did have access to examples affect students' project complexity?

Summary: Our results suggest that having access to examples encouraged students to build projects with API calls (i.e., blocks) they otherwise would not have used, and may have encouraged students to create larger overall projects. Examples did not seem to affect overall project complexity, but they may have influenced the choices of mechanics students used to achieve that complexity.

Table 1 shows that compared to the control group, the example group built projects with higher overall size (# blocks), but this difference was not significant. However, the distinct number of blocks (distinct # blocks) was significantly higher in the experimental group (p = 0.04), with a large effect size (d = 0.65), showing that the example group used a greater variety of Snap*!* APIs.

We investigated the control and example groups' "use rate difference" (Section 3.5) in their choices of Snap*!* APIs, and found that **the additional types of blocks used by the example group primarily came from the code examples**: we found that the top 3 blocks that have the largest use rate difference between example and control groups were: "delete this clone" (66.7% v.s. 20%[11]), "pick random _ to _" (50% v.s. 10%), and the operator "_ < _"(54.17% v.s. 15%). These blocks are often demonstrated by the examples, appearing in 35.14%, 27.03%, and 24.3% of all 37 examples, respectively, with none appearing in the 5 tutorial examples. These 3 blocks accounted for much (but not all) of the difference between the two groups. These results suggest that the examples likely had their intended effect of encouraging students to explore Snap*!* features and new game mechanics in their project: cloning, randomness, and boundary checking (e.g., detecting if an actor reaches the edge of the screen). As we discuss further in Section 5, because student-designed projects can go in many directions, and instructors cannot cover all relevant topics for all students, this suggests on-demand examples may play a role in helping students go beyond what was explicitly taught in class to accomplish their own project goals.

Lastly, we analyzed each mechanic's use rate difference between the example and control group, and found that **the control group may have spent more effort in building the basic mechanics they learned from the morning tutorials**. Specifically, when inspecting each mechanic's use rate difference, we found that the mechanics with the highest positive use rate difference (i.e., more popular in the example group) frequently came from integrating one or more examples. For example, the "end game on collision" mechanic had the highest positive use rate difference (62.5% v.s. 35%), and can be integrated from examples, as 12 examples include collision (i.e., a sprite touching another), and another (non-overlapping) 3 examples include ways to stop the game. However, the mechanics with the lowest negative use rate difference (i.e., more popular in the control group) commonly came from the morning tutorials. For example, "having a sprite that can move with key press" is a mechanic demonstrated in the morning tutorial, and has the lowest negative use rate (50% v.s. 70%). Other mechanics used more commonly in the control group also had similar characteristics: the mechanics with the second and third lowest use rate difference are "rotate with mouse" (0% v.s. 15%) and "conversation between sprites" (8.3% v.s. 15%). These mechanics are similar to what they have learned from the morning tutorial, where students may have had more time to learn and practice these mechanics before the afternoon project-making session.

These results show that, while the example group incorporated more features demonstrated from examples, the control group spent more effort in successfully implementing the more basic features, which they learned from the morning tutorials. However, it is worth noting that when evaluating project complexity, our rubric weighted each feature with different levels of difficulties equally. Table 1[12] shows that the significant difference in block variety, and the difference in students' mechanic and API preferences, did not lead to a significant difference in project complexity: while the example group had a consistently higher average on the game experience, applying knowledge, and the number of mechanics scores, the differences were not significant, with small effect sizes.

4.3 RQ2: Creativity and Task Load

RQ2: How did have access to examples affect students' self-reported creativity and task load?

Summary: Our results were inconclusive. They suggest that having access to examples *may* have led to an increase in students' perceived creativity, as well as task load; however, these differences were not significant.

The example group reported a total creativity score higher than the control group (Table 2). This difference is not significant, although the effect size is medium (d = 0.38). Specifically, the example group's creativity rating was higher in all 5 subscales. On the other

[11]Meaning the block "delete this clone" was used by 66.7% (16/24) example group students and 20% (4/20) control group students.

[12]We used r to indicate the effect size when the data is not normally distributed, and we conducted a Mann-Whitney-U test. We used d to indicate the effect size when the data is normally distributed, and we conducted a Student's t-test.

Table 2: Statistics for the control and example groups, with the *p*-value and effect sizes.

	2. task load & creativity				3. learning			
	total task load score		total creativity score		post-assignment score		normalized learning gain	
	control	example	control	example	control	example	control	example
n	21	25	21	25	21	24	21	24
mean	17.43	18.52	16.71	18.2	3.05	4.71	-0.21	-0.17
std	2.97	2.93	3.81	3.86	2.92	3.06	0.58	0.77
p-value		0.23		0.21		0.07		0.43
effect size		*d*=0.36		*d*=0.38		*r*=0.26		*r*=0.12

hand, the example group also reported higher task load, but the difference was not significant, with a medium effect size (d = 0.36). This rating is consistent over 4/6 TLX questions: Physical and Temporal Demand, Performance, and Effort. While our results are inconclusive, they offer suggestive evidence that examples may have helped students feel more creative, but may also have increased students' self-reported task load, causing them to feel that they needed to spend more time and effort in building the projects. We discuss why on-demand examples may increase, rather than decrease, task load in Section 5.

4.4 RQ3: Learning

RQ3: How did have access to examples affect students' learning?

Summary: Our results were inconclusive. They suggest that having access to examples may have improved some students' performance on a later, closed-ended programming task *without* examples, but the difference was not significant.

We found that the example group performed more than 50% better than the control group on their post-assignment, shown by the post-assignment score in Table 2. This difference was not significant (p = 0.07), but had a medium effect size (r = 0.26). This suggests that the example group may have learned more API usage patterns from the code examples, and applied them to a different post-assignment without examples, but further evaluations are needed to confirm this finding. As a reminder, both groups performed similarly on a pretest, so it is likely that this difference was due to learning that occurred while making the open-ended project (with or without examples).

On the other hand, both groups produced negative normalized learning gains (NLG) (Table 2). This is likely due to the fatigue of the full-day coding workshop (discussed in Section 3.5). As many students did not engage meaningfully with the post-test, the NLG is likely not reflective of differences in learning between conditions. For these reasons, we do not place much emphasis on this measure; however, we do note that the score drop in the example group was smaller than that in the control group, although the difference was not significant (p = 0.43), with a small effect size.

5 DISCUSSION & LIMITATIONS

Our results give important insight into how on-demand examples may affect students' behavior and outcomes in an authentic classroom setting. While some of our results were inconclusive, we note that this uncertainty reflects the challenges of real-world classroom experiments, where the motivation and prior knowledge of students may vary considerably. Conducting fully randomized, controlled studies in such settings is challenging, but they also provide some of the most useful data, and our results have clear implications for the use of on-demand examples to support open-ended project creation in classrooms, which we discuss below.

On-demand examples can be useful for encouraging students to use new programming APIs that are relevant to their project, but they may not increase the *overall* size or complexity of a project. Findings for RQ1 (Project Complexity) showed that the example group may have used more mechanics demonstrated in the examples, while the control group may have spent more effort in successfully implementing more basic mechanics that were introduced from the morning tutorials. These results likely reflect that students had a limited amount of time to build their projects; they either spent this time building ideas they found in the examples, or ideas they had already seen, but examples did not change the total amount of coding students could do. These differences in students' choices of features show that examples may encourage more exploratory programming behaviors, but may also cause students to spend less time practicing some basic programming knowledge they have previously learned. Our results for RQ2 (Creativity and Task Load) suggest that this focus on practicing new versus old mechanics may also have implications for how creative students feel, and how much mental effort they expend, though these results were less clear. Together, these results suggest a potential trade-off, where on-demand examples may help students who want to explore and build a creative project, but may also be less appropriate for students who need more practice in basic programming knowledge, and may already be experiencing a high task load. This shows the importance of personalizing on-demand examples, so that students who need more practice may be prompted with examples that reinforce the knowledge and APIs they have learned, rather than introducing new knowledge.

It is worth noting that the 5 tutorial examples in the ExampleAid system only included 8 of the 12 learning goals (4 basic, 2 cloning, 2 other APIs). However, the additional set of 32 examples covered all 12 learning goals. Therefore, the example group, which had access to these 32 additional examples, had more exposure to the 12 learning goals. Although all 12 learning goals were demonstrated in the morning tutorial, those who achieved higher post-assignment scores may have benefited from using and applying the more challenging goals from the additional examples.

Furthermore, it is important to note that the increase in API usage may not have resulted in improved learning outcomes. Although the post-assignment indicated that the example group achieved slightly higher scores, the difference in scores between the pre-

and post-tests was minimal and not statistically significant. This suggests that while some students were able to apply what they learned from the examples to a different task, others still faced challenges when programming a similar task without access to examples. Previous research has shown that programmers often opportunistically rely on examples to quickly achieve functionality rather than fully grasping the underlying concepts [7]. However, it remains an open question in the literature whether this opportunistic use of examples actually facilitates learning or simply aids in task completion. Similarly, our study also revealed the difficulty of transferring knowledge acquired from examples to new tasks. This highlights the need for future research to explore methods that can help users effectively learn from examples instead of merely copying and relying on functional solutions.

Limitation 1: Generalizability. One limitation of the work is that we focused on one example system (Example Helper). While the findings may not generalize to systems that are drastically different to Example Helper, the design principles of the system are similar to systems developed from prior work (e.g. [22, 54]), where students may browse, select and copy code examples. Similar results may be achieved if instructors simply provide a set of code examples to students when they work on open-ended programming projects (e.g., [29]).

Limitation 2: Full-day coding workshop and worse post-test performance. Another limitation that impacted our results is the study duration of our workshops. Given that many students were unfamiliar with Snap*!*, the full-day coding workshop had a steep learning curve and lasted long hours. This had an impact on students' fatigue and engagement levels, as shown by instructor-reported off-task behaviors and students' negative NLGs. The NLGs, as a result, did not provide meaningful information on whether access to examples led to better code-tracing problem performances. Future work should investigate ways to reduce students' fatigue, for example, by breaking down the coding workshop into multiple days.

Limitation 3: Low example integration rate. Our findings indicate that, on average, students in the example group accessed 2.76 examples (min = 0, max = 13, std = 3.07) and incorporated 1.42 examples into their final code (min = 0, max = 5, std = 1.35). This limited integration can be attributed to the time constraints faced by students. With only 2 hours and 30 minutes allocated for programming a complex game, students may prioritize creating a basic prototype or utilizing examples that demonstrate specific aspects of the game and can be completed within a short timeframe. The low rate of example utilization may also stem from a lack of alignment between the available examples and students' programming objectives or difficulties in locating suitable examples. Future research should investigate the causes of low example integrations, and identify strategies to match examples with students' needs and explore effective approaches for searching and reusing examples. In addition, with the recent progress made in automated program analysis (e.g., [12, 42, 49, 51, 52]) and machine learning-based code example generation systems [4, 28, 35], it is essential for future code example systems to embrace these advancements. By incorporating these updates, such systems may effectively create more tailored and scalable examples.

6 CONCLUSION

In this work, we investigated the impact of having access to examples on students' open-ended programming experience. We found that having access to code examples allowed students to create projects with more variety of APIs, and include new mechanics introduced by the examples, although not significantly increasing their project complexity. We found suggestive evidence that examples may have led to an increase in students' perceived creativity and task load, although the differences were not significant. Finally, we found suggestive evidence of students having improved performance on a later, closed-ended programming assignment after having access to examples during open-ended programming. Our results show the potential for code examples to increase students' creativity and support API learning during open-ended programming, and implicate the importance of designing personalized code examples that can both increase creativity and reduce task load.

7 ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1917885.

REFERENCES

[1] Felix Adler, Gordon Fraser, Eva Gründinger, Nina Körber, Simon Labrenz, Jonas Lerchenberger, Stephan Lukasczyk, and Sebastian Schweikl. Improving readability of scratch programs with search-based refactoring. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 120–130. IEEE, 2021.

[2] Kashif Amanullah and Tim Bell. Evaluating the use of remixing in scratch projects based on repertoire, lines of code (loc), and elementary patterns. In *2019 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE, 2019.

[3] Moshe Barak. From "doing" to "doing with learning": Reflection on an effort to promote self-regulated learning in technological projects in high school. *European Journal of Engineering Education*, 37(1):105–116, 2012.

[4] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 500–506, 2023.

[5] Jonathan Black, Jo Brodie, Paul Curzon, Chrystie Myketiak, Peter W McOwan, and Laura R Meagher. Making computing interesting to school students: teachers' perspectives. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 255–260, 2013.

[6] A. F. Blackwell. First steps in programming: a rationale for attention investment models. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 2–10, 2002.

[7] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598, 2009.

[8] Veronica Cateté, Nicholas Lytle, and Tiffany Barnes. Creation and validation of low-stakes rubrics for k-12 computer science. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, pages 63–68, 2018.

[9] Erin Cherry and Celine Latulipe. Quantifying the creativity support of digital tools through the creativity support index. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 21(4):1–25, 2014.

[10] Jennifer K Crissman. *The design and utilization of effective worked examples: A meta-analysis.* The University of Nebraska-Lincoln, 2006.

[11] Peggy Cuevas, Okhee Lee, Juliet Hart, and Rachael Deaktor. Improving science inquiry with elementary students of diverse backgrounds. *Journal of Research in Science Teaching: the Official Journal of the National Association for Research in Science Teaching*, 42(3):337–357, 2005.

[12] Adina Deiner, Christoph Frädrich, Gordon Fraser, Sophia Geserer, and Niklas Zantner. Search-based testing for scratch programs. In *International Symposium on Search Based Software Engineering*, pages 58–72. Springer, 2020.

[13] Diana Franklin, Merijke Coenraad, Jennifer Palmer, Donna Eatinger, Anna Zipp, Marco Anaya, Max White, Hoang Pham, Ozan Gökdemir, and David Weintrop. An analysis of use-modify-create pedagogical approach's success in balancing

structure and student agency. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 14–24, 2020.

[14] Gao Gao, Finn Voichick, Michelle Ichinco, and Caitlin Kelleher. Exploring programmers' API learning processes: Collecting web resources as external memory. In Michael Homer, Felienne Hermans, Steven L. Tanimoto, and Craig Anslow, editors, *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2020, Dunedin, New Zealand, August 10-14, 2020*, pages 1–10. IEEE, 2020. doi: 10.1109/VL/HCC50065.2020.9127274. URL https://doi.org/10.1109/VL/HCC50065.2020.9127274.

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

[16] Paul Goldenberg, June Mark, Brian Harvey, Al Cuoco, and Mary Fries. Design principles behind beauty and joy of computing. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 220–226, 2020.

[17] David Gonzalez-Maldonado, Alex Pugnali, Jennifer Tsan, Donna Eatinger, Diana Franklin, and David Weintrop. Investigating the use of planning sheets in young learners' open-ended scratch projects. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, pages 247–263, 2022.

[18] Shuchi Grover, Satabdi Basu, and Patricia Schank. What we can learn about student learning from open-ended programming projects in middle school computer science. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, page 999–1004, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450351034. doi: 10.1145/3159450.3159522. URL https://doi.org/10.1145/3159450.3159522.

[19] Sandra G Hart and Lowell E Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. In *Advances in psychology*, volume 52, pages 139–183. Elsevier, 1988.

[20] Carol Hulls, Chris Rennick, Sanjeev Bedi, Mary Robinson, and William Melek. The use of an open-ended project to improve the student experience in first year programming. *Proceedings of the Canadian Engineering Education Association (CEEA)*, 2015.

[21] Michelle Ichinco and Caitlin Kelleher. Exploring novice programmer example use. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 63–71. IEEE, 2015.

[22] Michelle Ichinco, Wint Yee Hnin, and Caitlin L Kelleher. Suggesting api usage to novice programmers with the example guru. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 1105–1117, 2017.

[23] Jay Jennings and Kasia Muldner. When does scaffolding provide too much assistance? a code-tracing tutor investigation. *International Journal of Artificial Intelligence in Education*, 31(4):784–819, 2021.

[24] Mary Beth Kery and Brad A Myers. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–29. IEEE, 2017.

[25] Prapti Khawas, Peeratham Techapalokul, and Eli Tilevich. Unmixing remixes: The how and why of not starting projects from scratch. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 169–173. IEEE, 2019.

[26] Dimitra Kokotsaki, Victoria Menzies, and Andy Wiggins. Project-based learning: A review of the literature. *Improving schools*, 19(3):267–277, 2016.

[27] B. M. Lange and T. G. Moher. Some strategies of reuse in an object-oriented programming environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '89, page 69–73, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913019. doi: 10.1145/67449.67465. URL https://doi.org/10.1145/67449.67465.

[28] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. Using large language models to enhance programming error messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 563–569, 2023.

[29] Ally Limke, Alexandra Milliken, Veronica Cateté, Isabella Gransbury, Amy Isvik, Thomas Price, Chris Martens, and Tiffany Barnes. Case studies on the use of storyboarding by novice programmers. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, pages 318–324, 2022.

[30] Nicholas Lytle, Veronica Cateté, Danielle Boulden, Yihuan Dong, Jennifer Houchins, Alexandra Milliken, Amy Isvik, Dolly Bounajim, Eric Wiebe, and Tiffany Barnes. Use, modify, create: Comparing computational thinking lesson progressions for stem classes. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pages 395–401. ACM, 2019.

[31] Jeffrey D Marx and Karen Cummings. Normalized change. *American Journal of Physics*, 75(1):87–91, 2007.

[32] J Moenig and B Harvey. Byob build your own blocks (a/k/a snap!). *URL: http://byob.berkeley. edu/, accessed Aug*, 2012.

[33] Teresa M Morales, EunJin Bang, and Thomas Andre. A one-year case study: Understanding the rich potential of project-based learning in a virtual reality class for high school students. *Journal of Science Education and Technology*, 22(5): 791–806, 2013.

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

[35] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 1–5, 2022.

[36] Thomas V Perneger. What's wrong with bonferroni adjustments. *Bmj*, 316(7139): 1236–1238, 1998.

[37] Kris Powers, Stacey Ecott, and Leanne M Hirshfield. Through the looking glass: teaching cs0 with alice. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 213–217, 2007.

[38] Arif Rachmatullah, Bita Akram, Danielle Boulden, Bradford Mott, Kristy Boyer, James Lester, and Eric Wiebe. Development and validation of the middle grades computer science concept inventory (mg-csci) assessment. *EURASIA Journal of Mathematics, Science and Technology Education*, 16(5):em1841, 2020.

[39] Mary Beth Rosson and John M Carroll. Active programming strategies in reuse. In *European Conference on Object-Oriented Programming*, pages 4–20. Springer, 1993.

[40] Mary Beth Rosson and John M Carroll. The reuse of uses in smalltalk programming. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(3):219–253, 1996.

[41] Amir Shareghi Najar and Antonija Mitrovic. Examples and tutored problems: How can self-explanation make a difference to learning? In *International Conference on Artificial Intelligence in Education*, pages 339–348. Springer, 2013.

[42] Yang Shi, Krupal Shah, Wengran Wang, Samiha Marwan, Poorvaja Penmetsa, and Thomas Price. Toward semi-automatic misconception discovery using code embeddings. In *The 11th International Conference on Learning Analytics & Knowledge (LAK 21)*, 2021.

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

[44] John Gregory Trafton and Brian J Reiser. *The contributions of studying examples and solving problems to skill acquisition.* PhD thesis, Citeseer, 1994.

[45] Wengran Wang. The design and evaluation of automated examples to support creative, open-ended programming. 2023.

[46] Wengran Wang, Yudong Rao, Yang Shi, Alexandra Milliken, Chris Martens, Tiffany Barnes, and Thomas W. Price. Comparing feature engineering approaches to predict complex programming behaviors. *Educational Data Mining in Computer Science Education (CSEDM) Workshop @ EDM'20*, 2020.

[47] Wengran Wang, Yudong Rao, Rui Zhi, Samiha Marwan, Ge Gao, and Thomas Price. The step tutor: Supporting students through step-by-step example-based feedback. *ITiCSE'20 - Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 391–397, 2020.

[48] Wengran Wang, Rui Zhi, Alexandra Milliken, Nicholas Lytle, and Thomas W. Price. Crescendo: Engaging students to self-paced programming practices. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 859–865, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367936. doi: 10.1145/3328778.3366919. URL https://doi.org/10.1145/3328778.3366919.

[49] Wengran Wang, Gordon Fraser, Tiffany Barnes, Chris Martens, and Thomas Price. Execution-trace-based feature engineering to enable formative feedback on visual, interactive programs. *Educational Data Mining in Computer Science Education (CSEDM) Workshop @ EDM'21*, 2021.

[50] Wengran Wang, Archit Kwatra, James Skripchuk, Neeloy Gomes, Alexandra Milliken, Chris Martens, Tiffany Barnes, and Thomas Price. Novices' learning barriers when using code examples in open-ended programming. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, pages 394–400, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382144. doi: 10.1145/3430665.3456370. URL https://doi.org/10.1145/3430665.3456370.

[51] Wengran Wang, Chenhao Zhang, Andreas Stahlbauer, Gordon Fraser, and Thomas Price. Snapcheck: Automated testing for snap programs. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, pages 227–233, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382144. doi: 10.1145/3430665.3456367. URL https://doi.org/10.1145/3430665.3456367.

[52] Wengran Wang, Gordon Fraser, Mahesh Bobbadi, Benyamin T Tabarsi, Tiffany Barnes, Chris Martens, Shuyin Jiao, and Thomas Price. Pinpoint: A record, replay, and extract system to support code comprehension and reuse. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–10. IEEE Computer Society, 2022.

[53] Wengran Wang, Audrey Le Meur, Mahesh Bobbadi, Bita Akram, Tiffany Barnes, Chris Martens, and Thomas Price. Exploring design choices to support novices' example use during creative open-ended programming. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, pages 619–625, 2022.

[54] Wengran Wang, Yudong Rao, Archit Kwatra, Alexandra Milliken, Yihuan Dong, Neeloy Gomes, Sarah Martin, Veronica Catete, Amy Isvik, Tiffany Barnes, Chris Martens, and Thomas Price. A case study on when and how novices use code examples in open-ended programming. In *Proceedings of the 28th ACM Conference*

on Innovation and Technology in Computer Science Education V. 1, ITiCSE '23, New York, NY, USA, 2023. Association for Computing Machinery.

[55] Rui Zhi, Nicholas Lytle, and Thomas W Price. Exploring instructional support design in an educational game for k-12 computing education. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 747–752. ACM, 2018.

[56] Rui Zhi, Thomas W Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min Chi. Exploring the impact of worked examples in a novice programming environment. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 98–104. ACM, 2019.