Pinpoint: A Record, Replay, and Extract System to Support Code Comprehension and Reuse

Wengran Wang[1], Gordon Fraser[2], Mahesh Bobbadi[1], Benyamin T. Tabarsi[1],
Tiffany Barnes[1], Chris Martens[1], Shuyin Jiao[1], Thomas Price[1]
[1]*North Carolina State University, Raleigh, USA*
[2]*University of Passau, Passau, Germany*
wwang33@ncsu.edu, gordon.fraser@uni-passau.de, mbobbad@ncsu.edu, btaghiz@ncsu.edu,
tmbarnes@ncsu.edu, crmarten@ncsu.edu, sjiao2@ncsu.edu, twprice@ncsu.edu

***Abstract*—Block-based programming environments, such as Scratch and Snap*!*, engage users to create programming artifacts such as games and stories, and share them in an online community. Many Snap*!* users start programming by reusing and modifying an example project, but encounter many barriers when searching and identifying the relevant parts of the program to learn and reuse. We present Pinpoint, a system that helps Snap*!* programmers understand and reuse an existing program by isolating the code responsible for specific events during program execution. Specifically, a user can record an execution of the program (including user inputs and graphical output), replay the output, and select a specific time interval where the event of interest occurred, to view code that is relevant to this event. We conducted a small-scale user study to compare users' program comprehension experience with and without Pinpoint, and found suggestive evidence that Pinpoint helps users understand and reuse a complex program more efficiently.**

I. Introduction

Block-based programming environments, such as Scratch [26] and Snap*!* [16], provide novice-friendly features such as block-based editors, and visual, interactive output. They engage programmers by allowing them to easily create artifacts that feel interesting and meaningful to them [30], such as games, apps, animations, or stories.

A common way for users to get started in Scratch or Snap*!* is by reusing and modifying another programmer's work as an example project [28]. Reusing such example projects allows programmers to create artifacts that go beyond their own abilities, while maintaining a sense of ownership over their work [31]. Such example-centric programming [3] is commonly seen when learners *remix* (i.e., copy and modify) other projects in Scratch's online community [22], or when students learn new APIs by exploring examples [37].

However, program reuse requires not only knowledge of the programming language and APIs in the example [36], but also skills such as code navigation and code comprehension [13]. Many Scratch or Snap*!* users are beginner programmers without strong prior programming experience [9], and can easily encounter barriers when identifying, understanding, and integrating features from example programs [36]. This suggests a need to help such users navigate complex examples, identify and understand which code is responsible for specific features, and integrate them into their own program.

In this work, we present Pinpoint, a system that helps Snap*!* programmers understand and reuse an existing program by isolating the code responsible for specific events during program execution. Specifically, a user can record an execution of the program (including user inputs and graphical output), replay the output, and select a specific time interval where the event of interest occurred. Pinpoint then identifies and displays the code responsible for creating that event, including the needed set-up code. Unlike prior systems to support example playback and understanding (e.g. [6, 14]), which only highlight a relevant line of code, Pinpoint presents users with complete, executable *code slices* [40] that demonstrate specific functionality. We use dynamic program slicing [1] to create such code slices for selected time intervals, and then further subdivide them according to different aspects of functionality (e.g., movement, cloning, etc.) by creating slice profiles [29].

We evaluated Pinpoint in a user study with 17 programmers with various levels of programming background. We found suggestive evidence that Pinpoint improved users' ability to integrate example code into their own projects. Follow-up interviews revealed specific ways that Pinpoint helped, including allowing users to relate specific example code to output and helping to focus on relevant parts of the example code. Our primary contributions are: 1) A set of design goals for a system that helps Snap*!* programmers identify responsible code for a program behavior, based on a formative study; 2) The design and implementation of the Pinpoint system, which isolates a specific code slice responsible for certain event; and 3) A user study for evaluating the impact of Pinpoint to help with reuse tasks.

II. Related Work

Pinpoint is based on Snap*!*, a block-based, graphical programming environment [16] that engages novice programmers and end users to easily build visual, interactive apps and stories [12]. The Snap*!* programming editor includes one or more programmable sprites, each of which represents an actor on the Snap*!* stage, and carries out its own code instructions written by the user.

Snap*!* was originally based on Scratch [26], a highly-popular novice programming environment. A core feature of Scratch and Snap*!* are their online communities, which are

978-1-6654-4214-5/22/$31.00 ©2022 IEEE

built on the culture of remixing [8], where users can click the "Remix" button to make a copy and modify to start their own version [22], allowing them to collaborate creatively [27] to share ideas and learn from each other [31]. A large portion of Scratch projects are remixed projects [28], but as many projects are relatively complex to remix [22], remixers may not always learn from reuse. Amanullah and Bell found that remixers rarely understand the original program, and rarely transfer concepts from the remixed program to their own future programs [2]. Khawas et al. also found that the remixers rarely used cloning correctly or included new procedures [22]. Furthermore, remixers frequently delete sprites [22], showing a need to extract particular functionalities from programs.

A. Code Reuse

Remixing is an example of code reuse, which refers to the process of identifying useful components of the example code and integrating them into one's own program [18]. Programmers reuse code examples for different purposes, such as cloning and owning existing software [10], exploring ideas, understanding implementation details, and debugging their own code [38]. Learning from code examples before or while making a similar program has been shown to help students not only complete the program faster [41], but also perform better in a concept-related post-test [33], and effectively learn how to use APIs later in their own code [19].

Holmes et al. conducted four case studies on programmers' process of code reuse, and characterized the reuse process into two stages: 1) *locating and selecting* and 2) *integrating* [18]. During the *locating and selecting* stage, programmers need to navigate through a complete example program to find relevant areas of interest [18]. This process can be challenging for both experienced and novice programmers. For example, Ko et al. found that in this selection stage, software developers begin by searching for relevant information, but they often make use of limited and misrepresented cues in the program or the environment, which would cause failed searches [24]. Similarly, Gross et al. conducted an observational study for 14 novice programmers to identify code responsible for a target functionality and found that they engage in a cyclic search process of 1) generating assumptions based on a search target in the code or output, and then 2) reading and searching code to adjust or expand the potential code region relevant to the target functionality. These programmers frequently made false assumptions and failed 59% of the code identification tasks [13]. These results suggest that programmers need support that helps them make more accurate assumptions when relating functionality to a relevant code segment.

During the *integration* stage of code reuse, programmers need to adapt and integrate the selected code into their own program [18]. During this process, programmers may directly copy a subset of an example code to their own code, or may reimplement a functionality by themselves after reading and learning an example [37]. Prior work has identified many barriers programmers encounter when integrating example code [36]. For example, Wang et al. analyzed 44 novice programmers' example integration process and found that these programmers encounter barriers in *understanding* how to integrate an unfamiliar code block into their own context, *mapping* the functionality of a part of an example to their own code, and *modifying* the example to fit their own needs [36]. Wang et al. also found that students prefer smaller code examples with few or no unfamiliar code blocks [36]. This shows the need to craft examples into smaller, comprehensible code segments so that students may understand a specific segment before integrating it into their own code.

B. Supporting Code Comprehension & Reuse

Code comprehension refers to the process of programmers building a mental model of how a piece of code works [13, 35]. Von Mayrhauser defined that a key cognitive process during code comprehension is generating a hypothesis of the causal effect from a code segment to its output [35]. Programmers of different levels may all form an incorrect hypothesis, but experts discard questionable hypotheses and form correct ones more quickly than novices [35].

Prior work has developed tools to support program comprehension for programming education and end users. For example, Python Tutor visualizes stack traces for students to see internal data representations of the program state [15]. However, it is not designed for complex user input and graphical output of games and apps. Whyline in Alice [7] helps users ask why and why not questions for debugging their own code [23]. However, it can only answer object-specific questions such as "Why did Pacman resize .5?", but not "object-relative" [23] questions such as "Why did Pacman resize after the Ghost moved", which were frequently asked by Alice programmers [23].

Some prior work applied record/replay systems to help users understand or debug programs [6, 14]. Timelapse is a record/replay-based tool for debugging web applications, which points to the users the lines of code responsible for a point of interest during the recorded trace [6]. Similarly, Gross et al. developed a record/replay tool to help users in Looking Glass record and select the timeframe of interest during the playback. The system then highlights the code responsible for the timeframe [14]. However, both interfaces only highlight the lines of code responsible for the selected time frame in the output but do not extract an *executable* code slice from the program. In contrast, Pinpoint directly addresses the learning barriers Snap*!* programmers encounter when reading and understanding code examples that are long and include multiple sprites, leveraging techniques including static [40] and dynamic code slicing [1]. Pinpoint allows users to select a time interval in the recorded replay and inspect a decomposition of the original program, which only includes the part of the code responsible for the desired output, helping users learn their desired functionality in a targeted executable code example.

III. System Design Goals & Formative Study

Before introducing the Pinpoint system, we first present the design goals. To develop these design goals, we con-

ducted a formative, think-aloud study with six students in our university's introduction to engineering course (a required prerequisite for all CS courses) to investigate their code comprehension experience. The six students had various levels of programming experience. During the study, we asked the students to spend two minutes exploring the code of a mid-sized Snap*!* programming project (4 sprites, 57 code blocks), with the goal of being able to explain how that code achieves the output on the stage. The project was a simplified version of a space-invaders style game, explained in more detail in Section IV-A. Using thematic analysis [4], two researchers transcribed the audio recordings of students' think-aloud utterances and conducted open-coding on the transcripts, while referencing corresponding screen recordings for context. They next discussed and sorted the open codes into three high-level themes, which revealed patterns of students' code comprehension experience when reading an unfamiliar, complex program.

1) Mapping from code to its runtime behavior: While reading the code, students frequently make hypotheses about the effect of a piece of code on the output, e.g., "*this is the code for when the bullet touches the enemy, they disappear.*"[P1]. However, these hypotheses were frequently erroneous. Prior work has found that such false hypotheses could lead to errors and inefficiencies in code maintenance tasks [23]. Therefore **Design Goal 1** is to help students better map a code segment to its runtime behavior.

2) Bottom-up, linear reading for the whole program: Several students read the code linearly from the top of the first sprite to the bottom of the last sprite (4/6[1]). Instead of starting from the output of the code and relating functionalities to code (top-down [5, 35]), several students' learning approach was primarily bottom-up [13, 35], where they read code first, and then related the code to its output. As the students were trying to understand the whole program, our video data shows that only one student was able to completely read through the project within two minutes, perhaps due to the length of the program and the different task of being asked to explain the output. Therefore, **Design Goal 2** is to help students find and focus on the most important/relevant code for their goal.

3) Not running or modifying code: Several students either ran the code only once (2/6) or did not run the code at all (3/6), which may explain students' misconceptions about its output, perhaps due to the complexity of the project and not knowing where to start. Therefore, **Design Goal 3** is to present users with relevant, executable code examples that are small and specific enough to run and modify.

IV. The Pinpoint System

A. The Pinpoint Design

To illustrate the use of Pinpoint, assume that a user wants to create a game with a character that shoots a bullet, based on Space Invaders game (shown in Figure 1), which includes this desired feature. In Space Invaders, the player controls a ship (blue) and tries to destroy a group of enemy ships (red) by shooting bullets with the space key. Space Invaders also has other features that add code complexity: the enemy ships also move and fire randomly toward the player, and the player can dodge left and right. The player wins when all enemies are destroyed. The user wants to identify the relevant shooting code and incorporate it into their own program. Using Pinpoint, they can do the following steps, whose numbers correspond to those highlighted in Figure 1.

1) Step 1: Record an execution: Our first design goal is to help users visually map the code to its runtime behavior. Pinpoint allows students to record their program execution by pressing the green flag to start recording and then pressing the stop button when they want the recording to stop. Users can view a replay of their program execution by clicking the "NEXT" button, or re-record by pressing the green flag again. While recording, Pinpoint creates an execution trace that records all user interactions (e.g., key presses), program states (e.g., variable values and sprite positions), and code executed. This is used to completely reproduce the program execution.

2) Step 2: Select an event: Our second design goal is to help users find and focus on the most important/relevant code for their goal. To do that, Pinpoint uses the slider bar for students to navigate through the recorded execution trace (Figures 1 – 2). The slider bar is automatically annotated with key events during program execution, such as clone creation/deletion and user inputs (e.g., ←, which refers to when the left arrow key is pressed). A student can select the start and end frame (each corresponding to the Snap*!* stage at the time index) to identify an event they want to explore. For example, an event of interest could be "when the space key is pressed, the bullet shoots out and destroys the enemy". For this, a user could select the time interval shown in Figure 1-2.

3) Step 3: Inspect the code: Our third goal is to present users with relevant, executable code examples that are small and specific enough to run and modify. To do that, as the user selects a time interval in the slider bar, Pinpoint automatically updates the relevant code slice to show only the code necessary to 1) set up the relevant event (e.g., moving the sprite to a starting location) and 2) carry out the event (cf. Section IV-B).

To help students understand the extracted code, Pinpoint includes the following 3 features: 1) **"How" questions.** One way to improve code comprehension is asking students to explicitly track changes to variables while reading code [39]. However, as code slices may include multiple variables and implicit properties (e.g., a sprite's position, size, and appearance), it is difficult to track all changes. Therefore, the user can filter the code using the menu tabs above each sprite to select questions, such as "How does the enemy change its position?". This will show only the code relevant to movement and relevant control structures. These "How" questions can be inspected one at a time, since students may be interested in only one aspect of an event (e.g., how the bullet was destroyed) but not others (e.g., how the bullet moves). We discuss the implementation of "how" questions in Section IV-B. 2) **Highlights for executing**

[1]4 among 6 students. Among the 2 other students, one started from another sprite, perhaps because it had the least number of code blocks; one student ran the code multiple times, but it was unclear how the student read the code, as they did not think aloud to verbalize their thoughts although prompted.

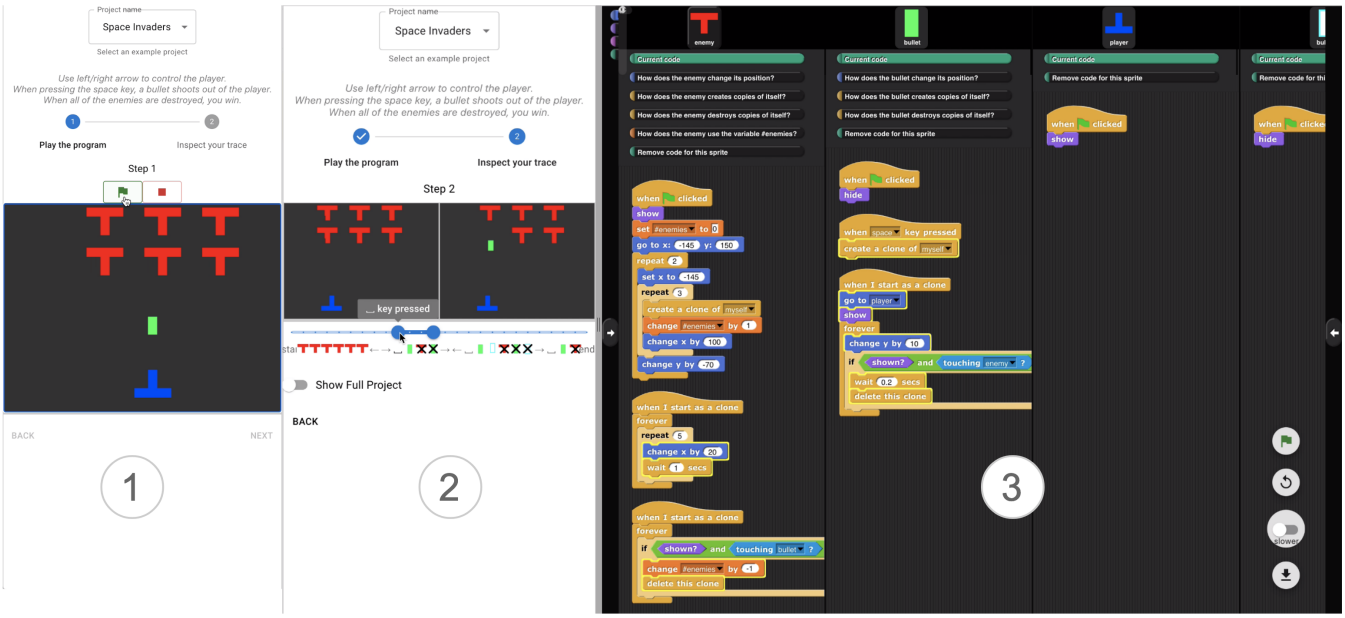

Fig. 1. Pinpoint users can 1) record a program execution (including user input and graphical output), 2) replay a recording and select a time interval where an event has occurred, and 3) inspect an executable code slice relevant to the event, where the code executed inside the selected time interval is highlighted.

blocks. Helping users quickly navigate to the key blocks for a code example has been shown to improve code comprehension [20]. Therefore, Pinpoint highlights the executing blocks in yellow for a selected interval, while the code blocks that are not highlighted are required for the program to execute (e.g., setup code). 3) **The "show full project" button**. To allow users to compare the simplified code slice with the original program, students can toggle the "show full project" button to view the complete original program.

Pinpoint provides an augmented Snap*!* editor, with the following features: 1) It places all sprites into different columns, as a selected event could impact several sprites (e.g., in "bullet shooting enemy"). 2) To allow users to both browse and execute/modify the code when needed, it uses toggle switches to open/close the code palette and Snap*!* stage. Users can add/delete/modify code blocks to tinker with any example code to verify what the example does (and if it matches their query) and explore changes that might alter its behavior.

B. Pinpoint Implementation

During Step 3 of the user experience (SectionIV-A), Pinpoint uses dynamic slicing [1] to find the code relevant for the selected time interval. It then uses static slicing [40] on top of the dynamic slice to automatically generate the "how" questions, which students can use to further isolate a code slice for the specific properties and variables of interest.

1) Using dynamic slicing to generate code corresponding to students' selected time interval: The program slice for a selected time interval (i.e., an executable subset of the program) is created using the program dependence graph (PDG) for the program being inspected. The PDG is a directed graph that consists of nodes which represent the program statements, and directed edges between these nodes which indicate control-flow, data-flow, or temporal (wait) dependencies. Pinpoint uses LitterBox [11] to create an inter-procedural PDG which includes all scripts of the program, as well as their dependencies caused, for example, by broadcast or clone events. In general, program slicing consists of a backwards traversal of the PDG starting from a chosen slicing criterion (e.g., target statement). In Pinpoint, the code executed in the selected time interval serves as a slicing criterion, such that the dynamic slice represents the subgraph consisting of nodes that (1) are covered in the execution trace, and (2) are reachable in a backwards traversal from any nodes executed within the selected time interval. The result, for example shown in Figure 1, is a subset of the code which was either executed in the selected time or sets up the start state of the time interval.

2) Using static slicing [40] to generate "How" questions: In order to generate "How" questions, Pinpoint further slices the program based on different sprite attributes (position, direction, size, visibility, sound effect, volume, layer, appearance, and clone status). To achieve this, Pinpoint creates the *slice profile* [29] for the dynamic slice of the current selection, i.e., it separates the slice into further sub-slices, each of which contains only code relevant to one attribute. For each attribute, the union slice [21] is created as the union of the backward slices of all statements that use (read) the attribute with the forward slices of the statements that define (write) the attributes within these backward slices. The answer to a "How" question is given by the union slice for the attribute underlying the question, as for example shown in Figure 2.

V. Methods

To collect formative data on how students use Pinpoint, and understand the strengths and weaknesses of the Pinpoint

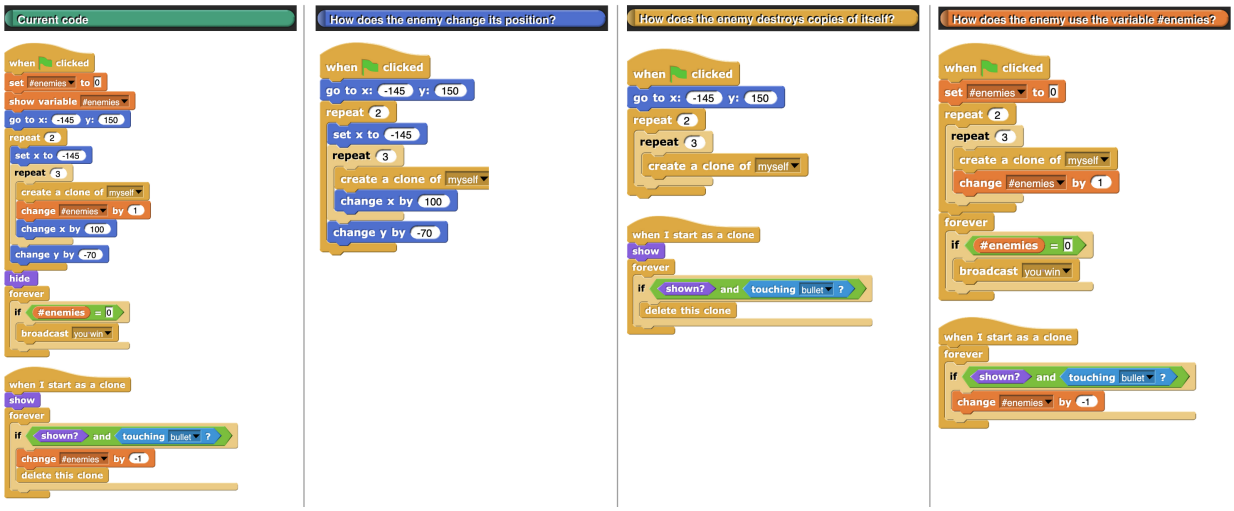

Fig. 2. Users can also trace changes to individual variables by selecting "How" questions on different variables and attributes.

system, we recruited a small group of students for a within-subject comparison study. We used the following research questions to direct our user study and analysis:

1) What is the impact of Pinpoint on students' ability to reuse example programs?
2) What are students' perceptions of their reuse experience?

A. Participants and Study Design

The goal of the study was to see how Pinpoint supported students in a standard reuse task: The students' goal is to create their own relatively simple target project, given a more complex example project implementing similar, but also additional, functionality as a reference.

1) Population: We recruited 17 participants from an introductory engineering course (5) and introductory computer science (12) course, including 11 males and 6 females; 10 Asian, 3 African American, 3 White and 1 Hispanic/Latino. Participants reported various levels of prior CS experience: 2 had no prior programming experience, 3 had taken only some tutorials, 11 reported taking or having taken 1-2 programming courses; 1 reported taking or having taken 3-4 programming courses. 10 students had not worked with Snap*!* or Scratch before, 6 "a few times", and 1 reported working with Snap*!* or Scratch "more than a few times".

2) Procedure: We used a within-subject design to understand how students use Pinpoint compared to a standard Snap*!* interface, through a 90-minute 1-on-1 study over Zoom. Students first completed informed consent, a pre-survey, Snap*!* tutorial (to familiarize them with the interface), and an 8-minute Snap*!* programming pre-test, which measured students' prior knowledge in Snap*!* programming, including questions about loops, conditional statements, variables, concurrency, cloning, and message passing. They then completed two reuse tasks: Space Invaders and Catch the Dots. To explore examples, all students used Pinpoint in one task and the standard Snap*!* interface in the other (details below). In both tasks, the students were first given a description of the target task, including a working version that they could play (without seeing the code). They were then shown the corresponding example project, and were asked to identify two key features from that example that would help them complete the target task. Then, they were given up to 20 minutes to explore the example project and complete the target task, the goal of which is to reduce ceiling effects and force students to explore the example efficiently. In the first 5 minutes, students were encouraged to read the example program without directly starting to program, as learning an example prior to programming has been shown to be beneficial to students [33]. During reuse tasks, students can also refer back to the example program at any time. As both Pinpoint and standard Snap*!* show the example in a separate interface, students can read the example project in a different tab or browser. After each reuse task, we conducted semi-structured interviews to ask students' reusing experience and the challenges they encountered.

To moderate possible effects of task difficulty, we randomized which assignment (Space Invaders or Catch the Dots) the students used Pinpoint on (i.e., a random half of students used Pinpoint first and half used it second; and independently, a random half used it on Space Invaders, and half used it on Catch the Dots). Before using Pinpoint, students learned a short tutorial on the system using a third task with distinct features from both the Space Invaders and the Catch the Dots program. To reduce the effect of learning from the first task, we designed the requirements of the two tasks to use distinct code patterns, so completing one task would not give away the solution to the next. For example, even though the two projects both have functionalities for collision detection, the implementation was different (one used forever-if blocks, and the other one used repeat-until block).

B. Materials: Two Reuse Assignments

Each of the assignments included two parts: the learning part, where students learn an example program; and the reusing

part, where students use parts of the example program to build their target program. We created two sets of example-target project pairs: Space Invaders / Rain Game and Catch the Dots / Flower Collection Game. The two example tasks were chosen as they represent the type of programs users may prefer to reuse in Snap*!*: they include engaging elements such as user interactions and multiple-sprite interactions; they were representative of the size of programs users may remix on Scratch, and were relatively high in code quality — which has been shown to enable more high-quality code reusing [17]. For each assignment, students were given starter code that has all sprites' images; one completed feature (move/rotate with key); and a set of unconnected blocks that would likely be needed for a correct solution (to reduce block searching time). To allow two tasks to include distinct features, the code for the overlapping feature in the two assignments (moving/rotating an actor with arrow key) was already provided.

1) Space Invaders – Rain Game: In one of the tasks, students first explore the Space Invader game from Section IV-A. The program includes 78 blocks across 5 sprites. They next complete a Rain Game with 7 distinct features, all having analogs in the space invader example: 1) when the space key is pressed, a water drop shows and goes to the location of the cloud; 2) The water drop can move slowly downwards; 3) Can create 3 clones of trees; 4) Tree clones are created with different x coordinates; 5) when water hits a tree, water disappears; 6) when water hits a tree, tree grows; and 7) when water hits the lower edge, water disappears. Additionally, Space Invaders also includes many other features, such as using a variable to track number of enemy clones, that are not needed for the Rain Game, requiring the student to select relevant code.

2) Catch the Dots – Flower Collection Game: In the other task, students first explore another example program (Catch the Dots), which is similarly complex as the Space Invader Program (4 sprites, 85 blocks). Students next use the Catch the Dots program to complete a Follower Collection Game, which includes two sprites, a flower and a pot. The project includes the following 7 distinct features that have analogs in the Catch the Dots program: 1) When the green flag is clicked, a flower shows at random position and point towards the pot. 2) When the green flag is clicked, set the score to 0; 3) The flower can move towards the pot; 4) When touching the pot body, the flower returns; 5) When touching the opening of the pot, the flower disappears and the score increases by 1; 6) The flower can restart its movement after it disappears or returns to the edge; 7) When the score equals 5, the pot says "you win", and the game stops. Similar to the Space Invaders program, Catch The Dots game also includes many other features that are not needed for the Rain Game, requiring the student to select relevant code. Note that Flower Collection and Rain Game both include collision detection functionality, but this was implemented differently in the example programs.

VI. Data Collection and Analysis

To understand in what ways Pinpoint was helpful for students, and investigate students' strategies when using Pinpoint, we conducted the following data collection and analyses based on our online tutorial sessions.

1) Pretest: To understand students' programming knowledge, we collected students' scores on the 8-minute pretest they completed prior to the two reusing tasks.

2) Task Performance: We collected the amount of time it took to complete programming tasks and their features using screen recordings and log data. As students were allowed for at least 20 minutes[2] to read and write code, but some completed the task before the 20-minute time limit (7 for task 1, 7 for task 2, with 5 students finishing early on both tasks), we used the following two measures for task performance:

2.1) *Task completion*: we analyzed students' code at the 20-minute timestamp to identify completed programming features. Tasks were graded by one researcher, blind to students' condition, on a binary rubric, where each feature was either complete (correct functionality) or incomplete/unattempted. Each task had 7 features, detailed in Section V-B.

2.2) *Time on task*: For the students who completed the assignments early, we measured the amount of time spent to complete the task features. In addition, we collected students' example reading time, capped at 5 minutes, as all students were suggested to use the first 5 minutes of their programming time to read and learn the example project and are required to start programming at the 5th minute, but they were also allowed to start earlier than the 5th minute, if they believed they had learned what was needed from the example program.

3) Qualitative Interview Analysis: We used thematic analysis [4] to conduct qualitative analysis to our interview data. As P12 consented to the study but did not consent to be audio-recorded, our interview data includes 16 students. We began by performing open coding on the first 6 interview transcriptions. Two researchers independently read through the interview transcripts and conducted open coding. They next met to discuss and merge codes and then arranged codes into high-level themes by finding commonalities between codes. They discussed and wrote down definitions of these themes, and compiled a codebook, which includes each theme and their definitions. One researcher then coded the remaining 10 transcriptions based on the codebook, updating the codebook to include 3 themes.

During the semi-structured interviews at the end of each student's programming session, we asked students whether they would prefer using Pinpoint or a standard Snap*!* interface for code comprehension. As part of students' response to this question (e.g., preferring Snap*!*, or Pinpoint, or mixed) is quantitative rather than qualitative, we report students' preference along with the other quantitative data in Section VII-A.

VII. Results and Discussion

A. RQ1: What was the impact of Pinpoint on students' ability to extract and reuse code from an example?

Participants' interview responses show that 15 out of 16 preferring Pinpoint to Snap*!*'s interface for example compre-

[2]One student requested to stop after only 17.5 minutes on Task 2, when using the standard Snap*!* interface.

hension (one had mixed feelings). This shows that students in general perceived Pinpoint to be helpful for code comprehension. Next, we discuss the impact of Pinpoint on students' task performance.

Students completed more features when using Pinpoint. We first conducted a within-subjects comparison of students' performance (i.e., # of features completed) on the task where they had Pinpoint versus when they did not. We found that students completed more of the 7 features in the task that uses Pinpoint ($M = 5.18$; $SD = 2.21$; Med=6) than when they did not ($M = 4.47$; $SD = 2.89$; Med=5). This difference was not significant according to a Wilcoxon signed-rank test[3] ($p = 0.056$; Cohen's $d = 0.28$). This shows that overall, students were able to complete an average of 0.7 more features (10% more of the task) when using Pinpoint, compared to not using Pinpoint.

Students improved significantly more from Task 1 to Task 2 when using Pinpoint second. One challenge with directly comparing students' performance on the two tasks (within-subjects) is that this does not control for time (Task 1 vs. Task 2). Students generally improved their performance in terms of features completed from Task 1 ($M = 4.53$) to Task 2 ($M = 5.11$), having had additional practice in Snap*!*. However, if Pinpoint is helpful, we would expect students to improve *more* when they transitioned from not having to having Pinpoint on Task 2 (the **Late** group), than when they lost access to Pinpoint from Task 1 to Task 2 (the **Early** group). To investigate this, we calculated students' *improvement* from Task 1 to Task 2 based on the number of completed features. We found that 6 students (2 Early; 4 Late) performed perfectly on both Tasks (i.e., a ceiling effect), likely due to higher prior programming experience[4]. Since the 6 students who completed both tasks perfectly could *not* have improved, we analyze their data separately below.

For the remaining 11 students (6 Early; 5 Late), we compared their improvement from Task 1 to Task 2 (Figure 3). We found that on average the Early group showed far less improvement ($M = -0.17$; $SD = 1.30$; Med=0) than the Late group ($M = 2.20$; $SD = 1.30$; Med=3). A Mann-Whitney U-test shows that the difference is significant ($W = 3.0$; $p = 0.031$) and the effect size (Cohen's $d = -1.92$) is quite large. Put another way, the Early group did just as well on Task 1 (with Pinpoint) as on their second task (after 20 minutes of practice, but without Pinpoint), while the Late group did much better on Task 2 (with both Pinpoint and practice), getting a median 3 out of 7 *more* features complete. It is worth noting that the Late group (after removing the 4 perfectly performing students) performed worse overall (see Figure 3), which may have meant they had more room to improve. However, given the large differences between Early and Late groups, it seems safe to conclude that for students *who did need help* with the reuse task, Pinpoint had a large impact on their performance.

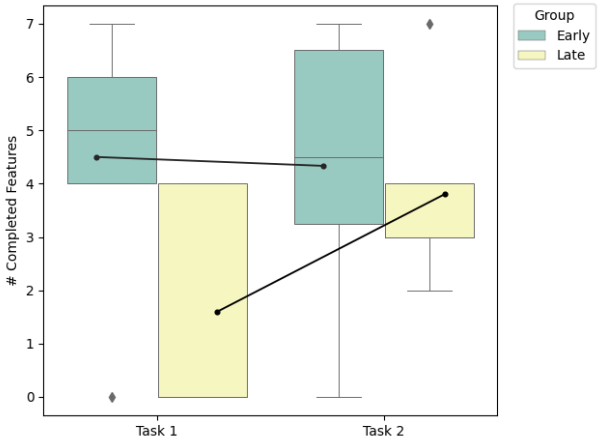

Fig. 3. For the 11 students without perfect performance, The Late group (shown in yellow) showed significantly more improvement than the Early group (shown in green).

For both of the above findings, it is also possible that the assignment the student did with Pinpoint impacted their performance somewhat; however, we found little difference between learners' performance in the Space Invaders assignment ($M = 4.59$; $SD = 2.90$; Med=6) and the Catch the Dots assignment ($M = 5.06$; $SD = 2.25$; Med=6).

When using Pinpoint, students used more time to read the example, and less time to complete the programming task. We also looked at how students spent their time when working on the reuse task. Considering all 17 students, we found that they spent on average about half a minute longer reading the example when using Pinpoint ($M = 254$ seconds; $SD = 62.0$; Med=289) than when using the standard Snap*!* interface ($M = 220$ seconds; $SD = 92.5$; Med=277), suggesting that Pinpoint may encourage students to spend more time reading and searching the example code before beginning to code. However, recall that this reading time still counted towards students' 20 minute time limit. Given that students generally completed *more* features when using Pinpoint, this suggests that the extra time for reading paid off. Finally, for the 6 students who completed both tasks perfectly, the time they spent improved from Task 1 to Task 2. We found that students completed Task 2 faster in both Early ($N = 2$; $M = 169.5$ seconds reduced; $SD = 29.0$; Med=169.5) and Late groups ($N = 4$, $M = 186.75$ seconds reduced; $SD = 199.3$; Med=147), but there was little difference between the groups. Therefore, Pinpoint may be most helpful for students with more difficulty in programming.

B. RQ2: What are students' perceptions of their reuse experience?

We discuss the key themes on participants' perceptions and mindsets during code reuse, based the qualitative interview analysis.

[3]We use non-parametric statistical tests, as our dependent variables were not normally distributed; however, we also report means, SDs and effect sizes for completeness.

[4]The 6 students who completed both tasks perfectly scored higher ($M = 7.83$, $SD = 1.33$) in the pretest than the 11 students who did not ($M = 6.36$, $SD = 1.75$).

1) Certainty: Certainty refers to students' self-perceived confidence when making decisions or generating hypotheses during code reuse. Participants expressed relatively high levels of certainty when using Pinpoint to find a code segment. For example, participants discussed that the slider bar "*makes it a lot easier finding certain codes*"[P13], as alternatively in Snap*!*, "*you may have to search through ... each sprite, to ... find code*"[P13]. 8 students discussed that the "How" questions helped them make better searches: "*most of these questions are questions I was having ... when I would be writing code without [Pinpoint]*"[P9]. Participants also expressed relatively high levels of certainty when using Pinpoint to hypothesize the runtime behavior of specific code segments: "*assigning an action to ... the code really helps solidify in your brain what you're supposed to do*"[P5].

Students discussed uncertainty when understanding unfamiliar code blocks both using and not using Pinpoint, e.g., "*it was harder to understand the repeat until [code block]*"[P13]. One also discussed uncertainty when trying to navigate through the slider bar to "*get it to the right time frame*"[P15].

2) Direction: Direction refers to the approach that participants employ to navigate through the example program. Many discussed a depth-first, top-down approach when using Pinpoint, where instead of learning the entire program, they concentrate on the desired code segment: "*it basically changes the scope, like instead of having to read through the whole code, you can just zoom in on the specific piece you want and look at that.*"[P2] Three students specifically mentioned that the code highlights were helpful for further focusing their attention on the most relevant code (i.e., code that ran, as opposed to setup code): "*I could see for specific actions, which one I had to focus on*"[P10], in contrast to their experience without Pinpoint: "*it kind of points it out for you... which section of the code is working versus me just looking at the code and not being able to recognize which portions are being used.*"[P6].

For the task using the standard Snap*!* interface, similar to the formative study (Section III) students in general described a breadth-first, bottom-up approach: "*I clicked on each [sprite] and see ... what happened*"[P4]. But some discussed negative feelings about their learning experience: "*I have not ... learned anything ... in detail about this program*"[P4].

3) Annotation: Annotation refers specifically to the mental process defined by Letovsky in the Code Cognition Model [25], where programmers mentally *manage and structure* hypotheses about how multiple code segments interrelates, and how they map to the program's runtime behavior. Some students, while using standard Snap*!*, discussed challenges with the annotation process: "*I was trying to understand ... (the) timeline, (as) I know there is a moment when I have to add when clones start, and then there is a moment I have to put ... clone myself.*"[P8]. But many discussed being able to build the connection of different code segments more easily when using Pinpoint: "*[Pinpoint] showed me what was being done by section and so ... when I ... was progressing through [the slider bar]. [Pinpoint] added more [code blocks], I was like, oh, okay, so that is what this additional section does.*"[P6]. In addition, students discussed that they perceived code dependencies more easily when using Pinpoint: "*I prefer ... looking at [Pinpoint] where the code is divided up. ... I think it is easier to look at how stuff are related that way, instead of just looking at it line by line.*"[P11], as for a specific event of interest, the full program include many irrelevant dependencies, which may distract students.

VIII. Conclusions and Future Work

In this paper, we present Pinpoint, a novel interface that helps Snap*!* programmers to understand and reuse an example program by isolating a specific code slice relevant to students' target event of interest. Our user study with 17 students provides suggestive evidence that Pinpoint improves students' ability to integrate code examples into their own projects. The students explained forming more confident hypothesis about a code segment's runtime behaviors, employing more focused, targeted example learning approach, and connecting different code segments more easily using Pinpoint. Overall, this suggests that Pinpoint achieved its design goals.

While these are promising results, our study and the Pinpoint system have limitations that we will address in future work. As a small-sample, within-subject factorial crossover design [34], our study suffers from variances caused by the wide range of students' programming backgrounds, the different order of tasks, and students being able to stop early at any time during example learning.

In addition, no participant explicitly mentioned running the code, and our observations also found limited testing/running behavior when using Pinpoint during the study. In future work, we may conduct specific interview questions on why students may choose to test or simply view the example code. The interview analysis also shows that users encountered barriers when trying to understand unfamiliar code blocks both when using and when not using Pinpoint. This shows that although Pinpoint allowed students to map a code segment to its functionality, it does not teach students the fundamental conceptual knowledge of a piece of code, which is an important subcomponent of robust API knowledge [32]. In the future, it may be helpful to point users to the documentation and tutorials inside Pinpoint, when they encounter unfamiliar blocks.

We may also further simplify the search experience: For example, one student noted in the interview that, as the reuse program becomes longer and more complex, the recorded trace can be long and difficult to navigate. Therefore, similar to how Pinpoint automatically generates the "How" questions to simplify search, we may intelligently generate higher-level questions regarding potential selections on the slider bar (e.g., auto-generating a "How to shoot the bullet" question), which users can click on and directly inspect a code slice, based on an auto-selected time interval.

IX. Acknowledgements

This work is supported by the National Science Foundation under Grant No. 1917885, and partially supported by the German Research Foundation under Grant FR 2955/3-1.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256, 1990.

[2] K. Amanullah and T. 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] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010.

[4] V. Braun and V. Clarke. Thematic analysis. 2012.

[5] R. Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18(6):543–554, 1983.

[6] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 473–484, 2013.

[7] S. Cooper, W. Dann, and R. Pausch. Alice: a 3-d tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116. Consortium for Computing Sciences in Colleges, 2000.

[8] S. Dasgupta, W. Hale, A. Monroy-Hernández, and B. M. Hill. Remixing as a pathway to computational thinking. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, pages 1438–1449, 2016.

[9] D. A. Fields, M. Giang, and Y. Kafai. Programming in the wild: trends in youth computational participation in the online scratch community. In *Proceedings of the 9th workshop in primary and secondary computing education*, pages 2–11, 2014.

[10] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 391–400. IEEE, 2014.

[11] G. Fraser, U. Heuer, N. Körber, E. Wasmeier, et al. Litterbox: A linter for scratch programs. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 183–188. IEEE, 2021.

[12] D. Garcia, B. Harvey, and T. Barnes. The beauty and joy of computing. *ACM Inroads*, 6(4):71–79, 2015.

[13] P. Gross and C. Kelleher. Non-programmers identifying functionality in unfamiliar code: strategies and barriers. *Journal of Visual Languages & Computing*, 21(5):263–276, 2010.

[14] P. A. Gross, M. S. Herstand, J. W. Hodges, and C. L. Kelleher. A code reuse interface for non-programmer middle school students. In *Proceedings of the 15th international conference on Intelligent user interfaces*, pages 219–228, 2010.

[15] P. J. Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584, 2013.

[16] B. Harvey, D. D. Garcia, T. Barnes, N. Titterton, D. Armendariz, L. Segars, E. Lemon, S. Morris, and J. Paley. Snap! (build your own blocks). In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 759–759, 2013.

[17] B. M. Hill and A. Monroy-Hernández. The cost of collaboration for code and art: Evidence from a remixing community. In *Proceedings of the 2013 conference on Computer supported cooperative work*, pages 1035–1046, 2013.

[18] R. Holmes, R. Cottrell, R. J. Walker, and J. Denzinger. The end-to-end use of source code examples: An exploratory study. In *2009 IEEE International Conference on Software Maintenance*, pages 555–558. IEEE, 2009.

[19] M. Ichinco, W. Y. Hnin, and C. 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.

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

[21] M. Kesselbacher and A. Bollin. Towards the use of slice-based cohesion metrics with learning analytics to assess programming skills. In *2021 Third International Workshop on Software Engineering Education for the Next Generation (SEENG)*, pages 6–10. IEEE, 2021.

[22] P. Khawas, P. Techapalokul, and E. 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.

[23] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158, 2004.

[24] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12):971–987, 2006.

[25] S. Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987.

[26] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010.

[27] A. Monroy-Hernández. Scratchr: sharing user-generated programmable media. In *Proceedings of the 6th inter-*

national conference on Interaction design and children, pages 167–168, 2007.

[28] A. Monroy-Hernández. *Designing for remixing: Supporting an online community of amateur creators*. PhD thesis, Massachusetts Institute of Technology, 2012.

[29] L. M. Ott and J. J. Thuss. Slice based metrics for estimating cohesion. In *[1993] Proceedings First International Software Metrics Symposium*, pages 71–81. IEEE, 1993.

[30] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[31] R. Roque, N. Rusk, and M. Resnick. Supporting diverse and creative collaboration in the scratch online community. In *Mass collaboration and education*, pages 241–256. Springer, 2016.

[32] K. M. Thayer. *Practical Knowledge Barriers in Professional Programming*. PhD thesis, 2020.

[33] J. G. Trafton and B. J. Reiser. *The contributions of studying examples and solving problems to skill acquisition*. PhD thesis, Citeseer, 1994.

[34] S. Vegas, C. Apa, and N. Juristo. Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering*, 42(2):120–135, 2015.

[35] A. Von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[36] W. Wang, A. Kwatra, J. Skripchuk, N. Gomes, A. Milliken, C. Martens, T. Barnes, and T. Price. Novices'

[41] R. Zhi, T. W. Price, S. Marwan, A. Milliken, T. Barnes, 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.

[37] W. Wang, A. Le Meur, M. Bobbadi, B. Akram, T. Barnes, C. Martens, and T. 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.

[38] W. Wang, Y. Rao, R. Zhi, S. Marwan, G. Gao, and T. 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, To be published*, pages 391–397, 2020.

[39] B. Xie, G. L. Nelson, and A. J. Ko. An explicit strategy to scaffold novice program tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 344–349. ACM, 2018.

[40] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

and M. 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.