Enabling Independent Learning of Programming Concepts through Programming Completion Puzzles

Kyle J. Harms, Noah Rowlett, Caitlin Kelleher
Department of Computer Science & Engineering
Washington University in St. Louis
St. Louis, Missouri, United States
{kyle.harms, noah, ckelleher}@wustl.edu

***Abstract*—Many novice programming environments use puzzle-like approaches to help novice programmers acquire new programming skills independently. Yet, little is known about 1) how puzzles can support effective learning of programming skills and 2) how learning programming using a puzzle-based approach compares to more a traditional tutorial style approach. We conducted a pair of studies to explore these two questions. First, we report lessons learned on the design of programming completion puzzles, their interface within a novice programming environment, and the design of a puzzle curriculum drawn from our first, formative study. We then report on a second study that compared the learning effectiveness of programming puzzles and tutorials. The results suggest that puzzles are a promising approach for introducing programming concepts within novice programming environments. Puzzle users performed 26% better on transfer tasks compared to tutorial users, while taking 23% less time to complete the learning materials.**

Keywords—novice programming; independent learning; programming puzzles; completion problems, completion strategy

I. Introduction

Opportunities to learn computer programming in formal educational settings are limited for many people, especially in grades K-12. In response, educators and researchers have created programming environments that can be used in informal settings [1]–[3]. Since classroom opportunities are limited, these systems often include resources designed to help users learn on their own. One of the challenges in creating these learning resources is that they must support users in learning both the user interface mechanics and programming concepts.

Traditionally, novice programming environments have often attempted to support independent learning using tutorials [4]–[6]. Much of the research on tutorials has focused on improving successful completion by guiding users through a series of steps while minimizing errors [7]–[9]. While completing tutorials may help with user interface mechanics, it is not clear how well novice programmers learn programming concepts via tutorials.

More recently, some programming environments have begun to use a more puzzle-like approach to facilitate independent learning [2], [10]–[12]. Often these environments present some type of challenge with code pieces and encourage users to solve the challenge using the code pieces. While the specific inspirations for many of these environments' puzzle-like approaches are unclear, the methods are similar to Van Merriënboer's programming completion strategy [13].

Van Merriënboer demonstrated that high school students who practiced programming by completing partial programs were later able to construct better programs than students who practiced by writing programs from scratch [14]. Completing a partial problem is known as the completion strategy or as a completion problem [14], [15]. Completion problems are partially worked examples that require a user to complete the remaining key steps [16]. The completion strategy has also been found effective in other educational domains [15], [17]. We hypothesize that programming puzzles can also leverage the completion strategy and consequently better support the learning of programming concepts.

Researchers have also proposed the use of programming puzzles to support learning [10], [18], [19]. However, we are unaware of any work that empirically evaluates the effectiveness of programming puzzles to support independent learning.

In this paper, we present our work developing programming completion puzzles that facilitate learning for middle school children. Specifically, we iteratively developed and tested a puzzle interface and curriculum within a blocks-based programming environment. We report our lessons learned on how to construct both the interface support and curriculum to encourage learning. We then compare the learning effectiveness of tutorials and puzzles on a set of four programming concepts. Participants who learned using programming puzzles spent 23% less time in the learning phase and performed 26% better on transfer tasks than participants who learned using tutorials.

II. Related Work

Prior work has investigated tutorials, tutors, and puzzle-like programming environments. We discuss how prior work supports learning independently in the context of programming.

A. Tutorials

Prior research has mostly investigated improving tutorial presentation and mechanics. Tutorial presentation can take the form of static text [20], pictures [21], or video [8], [22] or can even be interactive [5]–[7], [9], [23]. Generally, these systems have focused on improving tutorial completion by providing additional scaffolding or support by using videos [22], onscreen annotations [7], [9], synchronizing user progress with the tutorial [7], [8], [23], or viewing other users' tutorial experiences [24]. Some researchers have used gamification to help motivate users to complete tutorials [25], [26]. Researchers also found that reconstructing code via a tutorial enabled users to perform better on transfer tasks than directly inserting the code [5]. Beyond this, it is unclear how well tutorials will perform when teaching beyond mechanics.

B. *Tutors*

Unlike tutorials, programming tutors typically focus on providing students with programming practice alongside classroom instruction. During practice, many programming tutors focus on helping identify student programming errors [27]–[30]. Further, cognitive tutors [31] and intelligent tutoring systems [27] try to provide sufficient practice by modeling students' knowledge and suggesting additional practice for un-mastered concepts. These types of tutors have been found to be effective with classroom instruction [32].

Outside of the classroom, online tutors try to provide a classroom-like experience [33], [34]. However, many students often start but fail to finish online instruction [35]. It is not clear whether or not systems designed to complement the classroom or for extra practice are effective when used alone.

C. *Puzzle-Like Systems*

Puzzle-like systems frequently present some form of problem and require users to formulate a solution to solve that problem [36]. In the context of programming, many puzzle systems give users an objective that requires them to program an object's path through obstacles on a grid [2], [10], [37]–[39]. Frequently, puzzle-like programming environments use tutorials as part of their initial user experience [2], [10], [37], [38]. Introductory tutorials typically take the form of tutorial levels that introduce the puzzle's mechanics. To encourage learning, some systems use debugging [10] or competition [40], [41]. However, beyond these few systems many puzzle-like systems simply appear to encourage learning through some form of the completion strategy [13] including CORT [18] and Parson's puzzles [19]. We are not aware of any work that explores the learning effectiveness of programming puzzles.

III. Formative Evaluation

We broke our formative work into two parts: 1) designing the programming puzzles format and interface, and 2) creating a programming puzzle curriculum.

A. *Programming Puzzle Format & Interface*

We chose to implement our programming completion puzzles within the novice programming environment, Looking Glass [3]. Looking Glass is a blocks-based programming environment that enables users to author 3D animated stories by dragging and dropping program statement tiles. When executed, programs in Looking Glass are 3D animations.

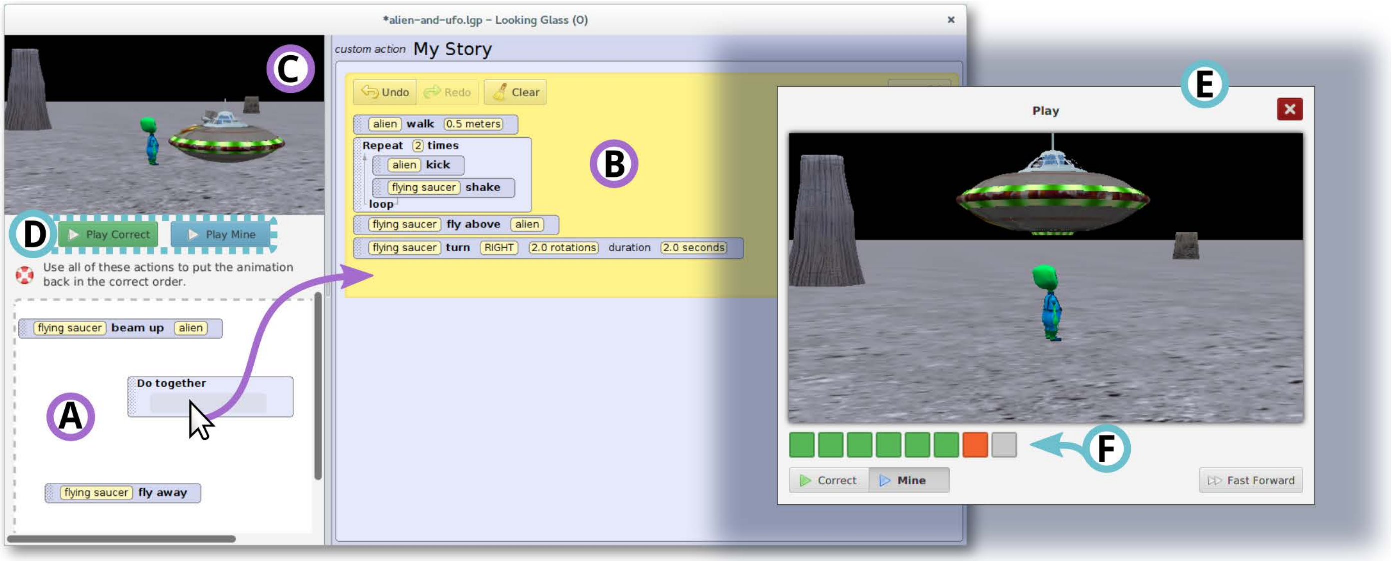
We designed the puzzle format and interface through ten iterations of formative testing. We conducted our formative testing with 23 users between the ages of 10 and 15 years at the St. Louis Science Center. Each session lasted 30 minutes. We compensated participants with a $5 gift certificate.

We used a variety of methods to iteratively design and develop our puzzle format and interface. Our methods included mockups, paper prototyping, Wizard of Oz testing, and working prototypes.

See Fig. 1 for our final puzzle interface. Users complete the puzzles by dragging statements from the unused statement bin (Fig. 1.A) into the puzzle workspace (Fig. 1.B). Users must use all of the statements in the bin to complete the puzzle. If users need to view the correct program output or view the puzzle's output, they click the appropriate play button (Fig. 1.D). This brings up the program output overlay pane (Fig. 1.E). When the user is viewing the puzzle output, the puzzle correctness indicator (Fig. 1.F) will show the user's current progress. The puzzle is correct, when all of the indicator's dots are green.

B. *Puzzle Format & Interface Lessons Learned*

We share some of the features of our puzzle interface and lessons learned in developing an effective puzzle format that facilitates independent learning.



Use all of these actions to put the animation back in the correct order.

```
alien walk 0.5 meters
Repeat 2 times
    alien kick
    flying saucer shake
loop
flying saucer fly above alien
flying saucer turn RIGHT 2.0 rotations duration 2.0 seconds
```

```
flying saucer beam up alien
Do together
flying saucer fly away
```

Fig. 1. Our programming completion puzzle interface. Users drag statements from the unused puzzle statements bin (A) into the puzzle workspace (B). (C) shows the initial starting state of the program. Users view the correct program output or their puzzle output by clicking the appropriate play button (D) which brings up the program output overlay (E). When viewing the puzzle's output users receive feedback on their progress from the correctness indicator (F).

1) Only show the user's work in the program's output.

Initially, we attempted to implement the completion strategy for programming by taking a complete program and scrambling the order of all the statements. Using a typical (drag-n-drop) block-based code editor, users would then only need to reorder the statements into the correct order to complete the puzzle. Through testing, we realized that when participants viewed the program's output, the output showed both the parts of the program that the user had reordered and the parts that they had not yet reordered. Mixing both elements in the output made it impossible for users to tell where their changes ended and where the scrambled code began.

To address this we switched from the reorder strategy to the place-in-order strategy. Users now place the statements in the correct order by dragging the unused statements from a bin and placing them into the correct order in the puzzle workspace. This ensures that only the work that the user has completed is shown in the program's output.

2) Limit the editable dimensions of the puzzle.

During our early testing using the reorder strategy, we found that users changed the code beyond reordering the statements. In a typical code editor, a programmer can change the code in several dimensions: she may add new statements, and edit, move, or delete existing statements. Early in our formative testing, we discovered that using a code editor gives users too many dimensions to manipulate. When everything in a puzzle is editable, the user thinks they need to change all of these dimensions. Many editable dimensions can make these puzzles extremely difficult to complete successfully. In our final version, we limited editing the puzzles to moving statements. Increasing the number of dimensions may provide later opportunities for increasing puzzle difficulty.

3) When executing a program, limit distractions and focus the user's attention on the program's output.

To complete a programming puzzle, a user needs to know what the correct output looks like and how it compares to the puzzle's output. We initially tried allowing users to execute and view both the correct output and the puzzle output side-by-side. Unfortunately, users spent most of their time looking back and forth between both outputs, missing key moments. Any distractions, including looking at the actual code, caused users to miss important details in the program output that would help them to complete the puzzle. We focus the user's attention by solely showing the output in an overlay pane that obscures the actual code as shown in Fig. 1.E. Users can view the correct output or the puzzle output, but not both at the same time. Without distractions, the user can carefully study the nuances of program's behavior and later tie the behavior to the appropriate puzzle statements.

4) Provide the user with ambiguous and incremental feedback towards the puzzle's solution.

When using the typical code editor for the puzzles, the user only received a notification that the puzzle was complete once every statement was in the correct order. Prior to receiving this notification, many participants believed they had correctly completed the puzzle, leading many to terminate the puzzle prematurely. We developed and tested several incremental progress indicators to provide feedback throughout puzzle editing. Unfortunately, early versions were easy to game; participants noticed they could quickly rearrange the statements until the progress bar increased without actually having to try to complete the puzzle. To limit this type of behavior and to encourage engagement, we increased the ambiguity within our feedback indicator. We increased the ambiguity by reducing the ability for users to make a direct mapping between the code and the progress indicator.

Fig. 1.F shows our final incremental and ambiguous progress indicator. Throughout our formative testing, all users used a top-down approach to solve the puzzles. Our progress indicator is designed to provide feedback for the first error in the puzzle, which supports this top-down approach. This indicator shows a new dot each time a statement is executed. A green dot means that the currently executing statement is correct. However, when the first incorrect statement in the puzzle (the *fly above* method in Fig. 1) is executed a new orange dot is added. Any executing statements that follow the incorrect statement are added as gray dots. For additional ambiguity, we group all similarly colored dots together; all green dots are grouped together, followed by all orange dots, and then the gray dots. Through testing, we found that this feedback mechanism seems to strike the right balance between incremental and ambiguous feedback.

TABLE I. PROGRAMMING PUZZLE CURRICULUM

ID	Programming Concept	Difficulty	No. Statements
1	Sequential execution	Easy	7
2	Repeated execution	Challenging	10
3	Parallel execution	Challenging	9
4	Repeated & Parallel	Easy	9
5	Parallel{ Repeated }	Challenging	8
6	Repeated{ Parallel }	Challenging	10

C. Programming Puzzle Curriculum

After developing the puzzle format and interface, we then moved to creating the actual content for the programming completion puzzles. We based our puzzle curriculum on concepts found in the middle school level (2) of the Computer Science Teacher's Association's curricular standards [42]. We selected four concepts at the introductory level that are useful for authoring 3D animations: repeated execution (*Repeat*), parallel execution (*Do Together*), parallel nested within repeated execution, and repeated nested within parallel execution.

We recruited 21 participants between the ages of 10 and 15 years from the Academy of Science of St. Louis mailing list. Each session lasted 90 minutes. Afterwards, we compensated participants with a $10 gift certificate.

We created eight iterations of our puzzle curriculum with the final version containing six puzzles. See Table I for the puzzle curriculum. We designed the puzzles to expose users to the four new concepts and vary in difficulty to help keep users engaged. Fig. 1 shows a partially completed puzzle for lesson 4.

D. Curriculum Lessons Learned

In the following section, we discuss some of the lessons we learned while creating our puzzle curriculum.

1) Author puzzle programs with motivating scenarios.

Completion problems are often described as partial worked examples [16] and so we initially created our early puzzles to be examples more focused on demonstrating concept behavior. However, this resulted in puzzles that users did not find compelling. For example, users frequently lacked the motivation to complete our initial repeat puzzle that simply had an object move up and down three times. From informal interviews, we realized that we needed to provide users with puzzles that excited them.

One approach we found that motivated users was to author puzzles with a problem-solution scenario. These puzzles contain a compelling initial scene with a scenario that contains a problem and resolution. For example, the puzzle in Fig. 1 shows a initial scene of an alien on a planet. After viewing the correct output, users realize that they have to help the alien leave the planet by repairing his broken spaceship. Users are motivated to finish the puzzle in order to help the alien escape the planet.

2) Author puzzle programs with memorable segments.

We found during testing that if an animation or part of an animation was not memorable, then users had to continuously flip between the correct output and puzzle output to know what to work on next. To limit this thrashing, a user should be able to remember the part of the puzzle they are working on without the need to continuously review the correct output. A simple approach is to author puzzle programs so that there are memorable segments. The alien example above contains three memorable segments: the alien fixing the broken spaceship, the alien entering the spaceship, and the spaceship leaving the planet. We also suggest using exaggerated or over-the-top animations which can be easier to remember.

3) Provide a challenge without being tricky.

Over the course of our formative sessions, we found that participants sometimes had very different reactions to difficult puzzles. In some cases, completing a difficult puzzle was a very rewarding experience. In others, participants remained frustrated, even after finding the correct solution. Participants' descriptions of these puzzles reflect these two kinds of reactions. Participants often described rewarding puzzles as *challenging* and frustrating puzzles as *tricky*. Through examining a collection of tricky and challenging puzzles, we noticed that what differentiates these puzzles is the source of their difficulty.

Tricky puzzles pull users' mental resources away from the puzzle's programming concept and instead expend their resources on extraneous details. Frequently, these extraneous details require participants to observe subtle details in the animation or notice subtle differences between statements. For example, an early puzzle included small rotations of a character's joint that participants struggled to perceive when viewing the puzzle output. In another case, a puzzle included nearly identical statements, one moved 1.0 meters and the other 1.2 meters. Participants frequently swapped these two statements in the puzzle and perceived the resulting output as matching the correct one. While participants typically solve tricky puzzles, they often finish with a sense that they have wasted a lot of time on unimportant details.

In contrast, we found that users really enjoyed challenging puzzles. Frequently, users commented that they liked the challenging puzzles best, "I thought this one was a little challenging, but I liked it!" The source of difficulty in challenging puzzles was the puzzle's new programming concept. These puzzles were still difficult for users, but this type of difficulty feels authentic because users finish with a new skill; spending time on it feels justified.

However, we note that too many challenging puzzles can be overwhelming for users. During one formative testing session, we replaced all of our easy puzzles with challenging ones. In this session, users no longer enjoyed the puzzles. Providing difficulty variation between different puzzles seemed to result in the best user experience.

4) Leave the users with a positive impression.

The last few statements of a puzzle leave a lasting impression on users. Frequently, when users recall their whole experience completing a task, their experience is shaped by the end experience of the task [43]. Even though users enjoy the challenging aspects of puzzles, we end all puzzles with easier statements. Not only does the user feel like they had a rewarding experience due to accomplishing the challenging portion, but they also walk away from the puzzle with a confidence boost. We believe this helps motivate users, as noted by some users, "I just wanna do puzzles!"

IV. Summative Evaluation

After our formative evaluation, we conducted a between subjects evaluation to assess the effectiveness of learning new programming concepts with programming puzzles compared to tutorials. After conducting a literature review of current widely-used novice programming systems, we observed that several offered the option to also use some form of tutorial [1], [2], [10]. We primarily based our tutorial condition on Scratch's step-by-step video and text tutorials [1].

See Fig. 2 for our implementation of these tutorials. Every step in the tutorial instructs the user to insert a statement into their program. For each step there is a looping screencast video showing how to complete the step as well as text directions instructing the user what statement they need to insert.

From our evaluation, we intended to answer the following questions: 1) Do puzzles require a different time and mental

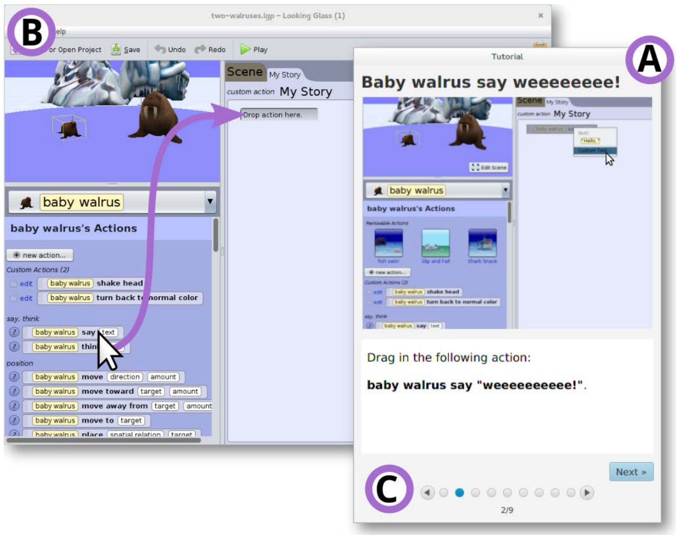

A
B
C
Tutorial
Baby walrus say weeeeeeee!
Drag in the following action:
baby walrus say "weeeeeeeeee!".
Next »
2/9
baby walrus
baby walrus's Actions
Scene
My Story
Drop action here.

Fig. 2. The tutorial condition. The tutorial window (A) shows a video and text description of each step. Users complete the step in the programming editor (B) and then manually advance the tutorial (C).

investment than tutorials? 2) Are puzzles more motivating than tutorials? and 3) Do puzzle users show more evidence of learning than tutorial users?

A. Participants

We recruited 34 participants between the ages of 10 and 15 for two separate experiments that lasted two hours. This paper presents only the first experiment. We recruited participants through the Academy of Science of St. Louis mailing list. The Academy of Science is a not-for-profit organization dedicated to science outreach within the St. Louis metropolitan area. This mailing list is well known by community members and was further forwarded around local schools and other groups.

We screened participants for minimal (less than 3 hours total) programming experience. Upon arriving to the study location, we interviewed all participants for prior programming experience. We removed 7 participants due to screening failures discovered during the interviews. With screening failures removed, we analyzed the data for 27 participants (gender: 12 female, 15 male; age: $M = 11.59$, $SD = 1.60$). We compensated participants with a $10 gift certificate.

B. Materials

We developed programs and survey materials that participants used during training and transfer study phases.

1) Familiarization Tasks

For each study phase, we developed a phase familiarization task. The familiarization task introduced participants to the format of the tasks for the phase and also introduced participants to the interface mechanics of the programming environment. Each task's format was identical to the training or transfer tasks.

2) Training Tasks

We used the puzzles developed during our formative testing for the content of our six training tasks. For our puzzle condition, we used the puzzles in their original form. However, for the tutorial condition we authored tutorials that produced the same programs as the puzzles but within a typical code editor.

3) Transfer Tasks

To measure learning, we wanted to know if given a novel problem, can users identify and correctly use the programming constructs needed to solve that problem. In response, we developed four transfer tasks, one for each of the challenging programming concepts introduced in our curriculum (See Table I). We asked participants to complete the transfer tasks in a typical code editor.

The transfer tasks are complete programs that contain correctly ordered method invocations along with a description and video of the correct program output. Each program is missing the required control flow constructs necessary to create the desired output. To complete a transfer task a user must 1) determine the appropriate constructs, 2) locate the constructs within the programming environment, and 3) drag the construct blocks into the existing code and move the appropriate statements into the construct blocks. No additional method invocations are necessary to complete these tasks. This setup is intended to allow participants to spend their time demonstrating their mastery of the programming concepts rather than writing entire programs from scratch.

4) Surveys

We used two surveys for our evaluation: a self-developed task survey and the Intrinsic Motivation Inventory's Task Evaluation Questionnaire (TEQ). The TEQ is a 22 Likert scale item questionnaire with four subscales: interest/enjoyment, perceived competence, perceived choice, and pressure tension [44]. Participants rate each item on a scale from 1 - not true at all, to 7 - very true.

Our task survey was comprised of a multiple-choice question and five Likert scale items: a mental effort scale and four enjoyment scales. The mental effort survey is a well validated and reliable unidimensional scale from 1 - extremely easy (very, very low mental effort), to 9 - extremely difficult (very, very high mental effort) [45], [46]. The perceived enjoyment scale contains four, seven item Likert scale items: enjoyable – unenjoyable, exciting – dull, pleasant – unpleasant, interesting – boring [47], [48]. Lastly, we included the *again again table*, which asks participants whether they would like to do this activity again with the following choices: yes, maybe, no [49].

C. Methods

We conducted our evaluation with four multi-user sessions. Each participant attended one of the two-hour sessions. Even though multiple participants attended each session, each participant worked independently. The study took place in a typical computer lab environment. We seated participants to minimize viewing other participants' screens.

We randomly assigned the participants to the puzzle or tutorial condition (puzzle: 14, tutorial: 13). Participants first completed the training phase followed by the transfer phase. Following the transfer phase, the participants completed a separate experiment (not presented). After completion of the separate experiment, participants were free to author their own programs. See Fig. 3 for an overview of the basic study procedure for this experiment.

1) Training Phase

The training phase consisted of the following activities: the familiarization task, six training tasks with task surveys, followed by the TEQ survey. We gave participants a maximum of 7 minutes to finish the familiarization task and each training task. Following each task, we asked participants to complete a task survey. During our pilot studies, almost all participants completed each task within 7 minutes (this is roughly the mean plus a standard deviation). We allowed participants to ask any questions during the familiarization task. We also helped participants complete the familiarization task if they needed assistance with the directions or interface mechanics. For the

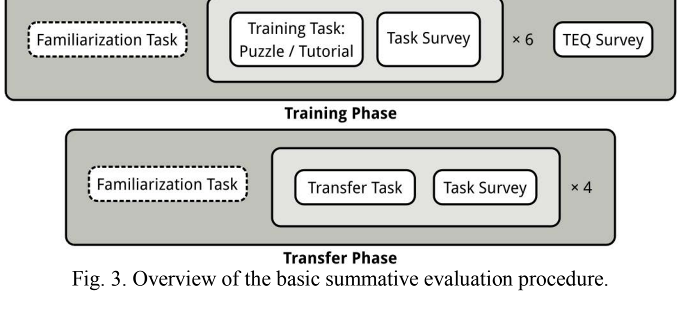


Fig. 3. Overview of the basic summative evaluation procedure.

actual training tasks, participants were not allowed to ask questions and worked independently.

Once participants completed the familiarization task, they were given the six training tasks in the curriculum order. Based on the participant's condition they completed all tasks using either the puzzles or tutorials. Following each training task, participants completed the task survey. After completing all training tasks, we asked participants to complete the TEQ.

2) Transfer Phase

The transfer phase consisted of the familiarization task and four transfer tasks with task surveys. We gave participants a maximum of 6 minutes to complete the familiarization and all transfer tasks. After each task, we asked participants to complete the task survey. We used a Latin squares design to administer the transfer tasks to control for learning effects. Like the training phase, we assisted participants if necessary during the familiarization task; we provided no assistance for the four transfer tasks. After each transfer task, we asked participants to complete the task survey.

V. Analysis

We collected log data from all tasks, scored the transfer tasks and analyzed the surveys.

A. Log Data

We gathered timing, event data, and code edits from the logs.

B. Transfer Task Scoring

We developed a grading rubric for each of the transfer programs based on the prior work of Harms et al. [5]. Using the rubric, we assigned a numerical score to each transfer task. For every transfer task, roughly three fourths of the points were dedicated to correct usage of the programming concept. When assigning points, we gave each attribute of the programming construct one point. The remaining points were assigned for carefully following the directions so that the program's output was identical to the correct output.

See Table II for a partial example of the scoring for a repeat statement in the repeat transfer task. The last two criterion: *no additional statements* and *animation is correct*, may seem identical, however they capture different attributes. Some participants will not use the repeat block and instead insert duplicate code statements. In this situation, the animation is correct, but solution is not (total score: 1 point).

We carefully designed each transfer program rubric to assign credit for each attribute once. Not all tasks have multiple solutions like the repeat task. For example, when evaluating the parallel transfer task, we gave a single point for correctly using a parallel construct and for having a correct animation. For this task, the user must use a parallel *Do Together* block; no other solutions exist to produce a correct animation without using the parallel construct.

TABLE II. TRANSFER TASK SCORING EXAMPLE

Scoring Rubric	Points
Has repeat block.	1
Repeat has correct index.	1
Correct statements inside of repeat.	1
No additional statements inserted.	1
The animation is correct.	1
Total	5

C. Task Surveys

We analyzed the results for the mental effort scale, the four perceived enjoyment scales, and the *again again table*.

D. TEQ Survey

The interest/enjoyment and perceived competence subscales of the TEQ are reliable, Cronbach's $\alpha = .91$, .72 respectively. The subscales for perceived choice (Cronbach's $\alpha = .58$) and pressure/tension (Cronbach's $\alpha = .62$) are not reliable. We have only included the interest/enjoyment and perceived competence subscales in our analysis.

VI. Results

In this section, we address the results of our evaluation in terms of answering our research questions.

A. Do puzzles require a different time and mental investment than tutorials?

We looked at time and mental investment by examining three factors from the training phase data: time, mental effort, and exposure to the programming concept. We wanted to know if there was any difference in completion time between the conditions. We also wanted to know if puzzles required more mental effort than tutorials and if puzzles exposed users to the appropriate concepts at the same rate as tutorials. We found that puzzle participants overall took 23% less time, required 50% more mental effort, and were exposed to programming concepts equivalently to tutorial participants.

1) Puzzle and Tutorial Task Time

Overall puzzle participants spent 23% less time completing the puzzles compared to the tutorial participants. See Table III for training task time for the puzzle and tutorial conditions. We conducted a MANOVA using Wilk's lambda statistic. There was a significant effect of condition (puzzle or tutorial) on the training task time, $\Lambda = .25$, $F(6, 20) = 9.90$, $p < .001$. However, separate univariate ANOVAs revealed that this effect was only significant for the sequential (1) $F(1, 25) = 41.44$, $p < .001$, the parallel (3) $F(1, 25) = 15.72$, $p < .001$, and the repeated & parallel (4) $F(1, 25) = 8.76$, $p < .01$ tasks. There was no significant difference for the repeated (2) $F(1, 25) = 0.19$, $p = .67$, the parallel with nested repeat (5) $F(1, 25) = 2.45$, $p = .13$, and the repeat with nested parallel (6) $F(1, 25) = 0.37$, $p = .55$ tasks. The total time participants spent completing all training tasks for the puzzle ($M = 21.89$, $SD = 6.05$) compared to the tutorial ($M = 28.31$, $SD = 5.05$) was also significant $t(24.73) = 3.00$, $p < .01$.

We were curious why some training tasks took less time for the puzzles, but not all. We hypothesized that puzzle difficulty and longer program execution due to loops might be a factor. We observed that all of the non-significant training tasks (2, 5, and 6) all contained repeated execution and were all rated difficult. Difficult puzzles require users to continuously iterate and check their work many times since the solution is not immediately obvious to them. In contrast, tutorial users were led through the correct assembly of the programs and did not need

TABLE III. TRAINING TASK RESULTS

Task	Concept	Puzzle Time (min)	Tutorial Time (min)	Puzzle Mental Effort	Tutorial Mental Effort	Puzzle Exposure	Tutorial Exposure
1	Sequential execution	$M = 2.45$, $SD = 0.99$***	$M = 5.02$, $SD = 1.10$***	$M = 2.86$, $SD = 1.41$	$M = 2.62$, $SD = 1.26$	100%	100%
2	**Repeated execution**	$M = 5.42$, $SD = 1.48$	$M = 5.64$, $SD = 1.10$	$M = 5.07$, $SD = 2.23$**	$M = 2.53$, $SD = 1.23$**	92.9%	100%
3	**Parallel execution**	$M = 3.19$, $SD = 1.12$***	$M = 4.78$, $SD = 0.99$***	$M = 4.00$, $SD = 2.32$*	$M = 2.15$, $SD = 1.34$*	100%	100%
4	Repeated & Parallel	$M = 2.80$, $SD = 1.24$**	$M = 4.21$, $SD = 1.24$**	$M = 3.29$, $SD = 1.73$	$M = 2.31$, $SD = 1.60$	100%	100%
5	**Parallel{ Repeated }**	$M = 3.86$, $SD = 1.88$	$M = 4.78$, $SD = 1.06$	$M = 5.21$, $SD = 2.72$*	$M = 2.85$, $SD = 2.12$*	71.4%	76.9%
6	**Repeated{ Parallel }**	$M = 4.18$, $SD = 1.76$	$M = 3.86$, $SD = 0.75$	$M = 4.64$, $SD = 2.44$*	$M = 2.62$, $SD = 2.33$*	78.6%	92.3%

Challenging tasks. *** $p < .001$, ** $p < .01$, * $p < .05$

to continuously check the output to validate their work. Repeated execution would also cause the program execution time to increase. Typically, each executing statement in Looking Glass takes one second to execute. We note however, that the number of statements for the training programs was relatively consistent as shown in Table I.

For all of the challenging tasks (2, 3, 5, and 6) puzzle users spent significantly more time executing their program than tutorial users. There was a significant effect of condition (puzzle or tutorial) on the program execution time, $\Lambda = .50$, $F(6, 20) = 3.48$, $p < .05$. Separate univariate ANOVAs revealed this was significant for tasks (2) $F(1, 25) = 18.07$, $p < .001$, (3) $F(1, 25) = 6.23$, $p < .05$, (5) $F(1, 25) = 7.52$, $p < .05$, and (6) $F(1, 25) = 11.59$, $p < .01$. Yet, of tasks 2, 3, 5, and 6, task 3 is the only task that does not contain a loop. We think this suggests that the extra time puzzles users spent in tasks 2, 5, and 6 is may be due to the extra time necessary to execute and view the output of the loops.

2) *Puzzle and Tutorial Mental Effort*

Puzzle participants rated the challenging tasks as requiring more mental effort than the tutorial participants. We conducted a MANOVA using Wilk's lambda statistic, there was a significant effect of condition on the mental effort rating, $\Lambda = .49$, $F(6, 20) = 3.47$, $p < .05$. Separate univariate ANOVAs revealed that this was significant for all the challenging tasks, (2) $F(1, 25) = 13.49$, $p < .01$, (3) $F(1, 25) = 6.26$, $p < .05$, (5) $F(1, 25) = 6.30$, $p < .05$, and (6) $F(1, 25) = 4.87$, $p < .05$. However, it was not significant for the easy tasks, (1) $F(1, 25) = .22$, $p = .64$ and (4) $F(1, 25) = 2.32$, $p = .14$.

3) *Puzzle and Tutorial Concept Exposure*

Participants were exposed approximately equally to the new programming concepts regardless of whether they used tutorials or puzzles during the training tasks. When completing a puzzle, participants may not have discovered the correct solution and thus lacked exposure to the programming concept. If a tutorial user failed to complete the tutorial correctly, they may also have lacked exposure. To measure concept exposure we analyzed each participant's training tasks to see if at some point during the puzzle or tutorial the code was in a state where the desired programming concept was correctly demonstrated. See Table III for the exposure rates for the puzzle and tutorials conditions. There are no significant differences between the conditions and exposure for all tasks: (1) $\chi^2(1) = .04$, $p = .85$, (2) $\chi^2(1) = 0$, $p = 1$, (3) $\chi^2(1) = .04$, $p = .85$, (4) $\chi^2(1) = .04$, $p = .85$, (5) $\chi^2(1) = 0$, $p = 1$, and (6) $\chi^2(1) = .21$, $p = .64$.

B. *Are puzzles more motivating than tutorials?*

We used the training tasks surveys and the TEQ to help us determine whether the puzzles are more motivating than tutorials. We found no significant effects for these measures between the puzzle and tutorial conditions suggesting no difference in motivation between the conditions.

The TEQ showed no significant changes between the conditions. For the TEQ interest/enjoyment subscale (1 = not true, 7 = very true), the puzzle ($M = 5.55$, $SD = 1.03$) and tutorial ($M = 5.64$, $SD = 1.42$), participants rated that they enjoyed their experience. For the perceived competence subscale, the puzzle ($M = 5.19$, $SD = .87$) and tutorial ($M = 5.94$, $SD = .94$) participants also rated that they felt competent. We conducted a MANOVA using Wilk's lambda statistic. There was no effect of condition on the interest/enjoyment and perceived competence subscales $\Lambda = .84$, $F(2, 24) = 2.34$, $p = .12$.

The training task surveys showed no differences between conditions. Participants in both conditions rated each task as enjoyable, exciting, pleasant, and interesting. They also rated that they would like to do the task again. For the perceived enjoyment survey we conducted a MANOVA using Wilk's lambda statistic for the effect of condition on the scales: enjoyable – unenjoyable $\Lambda = .79$, $F(6, 20) = .91$, $p = .51$, exciting – dull $\Lambda = .55$, $F(6, 20) = 2.78$, $p < .05$, pleasant – unpleasant $\Lambda = .72$, $F(6, 20) = 1.32$, $p = .30$, and interesting – boring $\Lambda = .67$, $F(6, 20) = 1.67$, $p = .18$. The exciting – dull scale was the only significant scale, however, separate univariate ANOVAs revealed no differences between the conditions. The *again again table* also revealed similar non-significant results, $\Lambda = .85$, $F(6, 20) = .59$, $p = .74$. These results suggest that puzzles are no more or less motivating that tutorials. However, we note that these results do not align with our formative evaluation observations; we discuss this further in the discussion section.

C. *Do puzzle users show more evidence of learning than tutorial users?*

Lastly, we wanted to know whether users learn programming concepts more effectively using puzzles or tutorials. Overall, we found that puzzle participants performed 26% better on the transfer tasks, required equivalent mental effort to complete the transfer tasks, and took 23% less time than tutorial participants. Next, we share the results of the transfer task data for performance, code edits, and mental effort. See Table IV for a summary of the transfer task results.

Puzzle participants performed better on the repeated and repeated{parallel} transfer tasks. We conducted a MANOVA using the Wilk's lambda statistic for the effect of condition on the transfer task performance scores $\Lambda = .62$, $F(4, 22) = 3.43$, $p < .05$. Separate univariate ANOVAs reveal that the puzzle participants performed significantly better than the tutorial participants for the repeated $F(1, 25) = 6.65$, $p < .05$, and

TABLE IV. TRANSFER TASK RESULTS

Transfer Task	Performance (%) – Puzzle	Performance (%) – Tutorial	Task Time (min) – Puzzle	Task Time (min) – Tutorial	Inserts – Puzzle	Inserts – Tutorial
Repeated	*M* = 89.5, *SD* = 18.8*	*M* = 65.6, *SD* = 28.7*	*M* = 3.75, *SD* = 1.32*	*M* = 4.94, *SD* = 1.07*	*M* = 3.4*	*M* = 4.4*
Parallel	*M* = 88.1, *SD* = 24.8	*M* = 76.9, *SD* = 34.4	*M* = 2.16, *SD* = 0.97	*M* = 2.87, *SD* = 1.46	*M* = 2.8	*M* = 2.4
Parallel{ Repeated }	*M* = 92.1, *SD* = 14.8*	*M* = 68.5, *SD* = 32.9*	*M* = 3.17, *SD* = 1.09	*M* = 3.96, *SD* = 1.77	*M* = 2.2	*M* = 2.8
Repeated{ Parallel }	*M* = 77.9, *SD* = 32.1	*M* = 65.4, *SD* = 31.5	*M* = 3.70, *SD* = 1.28*	*M* = 4.83, *SD* = 1.23*	*M* = 2.4**	*M* = 4.3**

** $p < .01$, * $p < .05$

repeated{ parallel } $F(1, 25) = 5.98$, $p < .05$ transfer tasks. However, there was no difference between the conditions for the parallel $F(1, 25) = .95$, $p = .34$ and parallel{ repeated } $F(1, 25) = 1.03$, $p = .32$ transfer tasks.

During the evaluation, we observed that some participants completed the transfer tasks in a trial and error fashion. If participants used significantly more edits than the minimum required, this would suggest they may have completed the task using trial and error. For some transfer tasks the tutorial users made more insertions than the puzzle users. We conducted a MANOVA for the difference between the conditions for the number of insertions participants performed $\Lambda = .63$, $F(4, 22) = 3.28$, $p < .05$. Separate univariate ANOVAs revealed that this is significant for the repeated $F(1, 25) = 5.51$, $p < .05$ and repeated{ parallel } $F(1, 25) = 12.08$, $p < .05$ transfer tasks. However, there are no significant differences between the conditions for move edits, $\Lambda = .93$, $F(4, 22) = .39$, $p = .82$. This suggests that in some tasks, the puzzle participants made more purposeful edits than the tutorial participants.

Additionally, in all transfer tasks puzzle participants took less time than the tutorial participants in completing the transfer tasks $\Lambda = .68$, $F(4, 22) = 2.62$, $p = .06$, however this difference is not significant. Mental effort between conditions is also non-significant $\Lambda = .98$, $F(4, 22) = .13$, $p = .97$. These results suggest that puzzles can help users perform in similar tasks better or at least equivalently to tutorials without significant differences in task completion time or mental effort.

VII. DISCUSSION

Overall, our results suggest that programming puzzles are a promising way to support the learning of programming concepts. Participants who used the puzzles required less time to learn and performed 26% better on the transfer tasks. Yet, puzzle participants rated the mental effort required to complete the training tasks 50% higher than tutorial participants. This difference in mental effort provides a potential explanation for the performance difference.

Tutorial participants could follow the instructions without engaging deeply with the programming concept at the core of each task, while the programming puzzles required participants to think about the behavior of each statement within a program and how to combine them to achieve a given outcome. The puzzle format also intentionally removes unnecessary code authoring barriers to focus learners on practicing directly with the programming concepts. This is in direct contrast to the tutorials which attempt to teach both the interface mechanics and the programming concepts simultaneously by demonstration. While the puzzles omit the interface mechanics, we believe that this approach better prepares learners for using programming concepts in a novel setting, like the transfer tasks.

While puzzle participants generally outperformed tutorial participants, we did not see significant differences for all tasks. Returning to the prior work on the problem completion effect may provide ways to further increase learning performance [15]. Specifically, we note that problem completion strategies often pair completion problems with worked examples [15]. We explored presenting examples alongside the puzzles in formative testing, but found that many participants preferred not to use the examples. Investigating how best to incorporate examples so that participants actively engage with example content is an interesting direction for future work.

Finally, we note that we were surprised by the similarity in attitudinal results between tutorial and puzzle participants. In formative tests where participants interacted with both the puzzles and tutorials, we observed a marked preference for the puzzles. As one user noted, "I like the puzzles more [than the tutorials]… so it was fun." A within-subjects study comparing attitudes towards the tutorials and puzzles may be better able to measure our observed preference. However, the lack of attitudinal difference between puzzles and tutorials may suggest that a motivating context, such as storytelling, plays a larger role in supporting motivation in early programming experience than the presentation of learning materials. After the initial appeal of the system has worn off, it is possible that the learning experience will begin to have a larger impact on motivation.

VIII. CONCLUSION

In this paper, we presented our programming completion puzzles as an effective learning alternative to tutorials. While additional work is necessary to improve this strategy further, we believe that these puzzles can help users learn programming concepts independently. We also hope that this work inspires novice programming environment authors to actively utilize strategies that facilitate effective learning, ultimately resulting in broader access and learning opportunities for children to learn computer programming.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grant No. 1054587.

REFERENCES

[1] "Scratch," *Scratch*. [Online]. Available: https://scratch.mit.edu/.

[2] "Code.org," *Code.org*. [Online]. Available: http://code.org/.

[3] "Looking Glass." [Online]. Available: http://lookingglass.wustl.edu/.

[4] K. J. Harms, J. H. Kerr, and C. L. Kelleher, "Improving Learning Transfer from Stencils-based Tutorials," in *Proc. Interaction Design and Children*, New York, NY, USA, 2011.

[5] K. J. Harms, D. Cosgrove, S. Gray, and C. Kelleher, "Automatically Generating Tutorials to Enable Middle School Children to Learn Programming Independently," in *Proc. Interaction Design and Children*, New York, NY, USA, 2013.

[6] L. Bergman, V. Castelli, T. Lau, and D. Oblinger, “DocWizards: A System for Authoring Follow-me Documentation Wizards,” in *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2005, pp. 191–200.

[7] C. Kelleher and R. Pausch, “Stencils-based Tutorials: Design and Evaluation,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2005, pp. 541–550.

[8] S. Pongnumkul, M. Dontcheva, W. Li, J. Wang, L. Bourdev, S. Avidan, and M. F. Cohen, “Pause-and-play: Automatically Linking Screencast Video Tutorials with Applications,” in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2011, pp. 135–144.

[9] C.-Y. Wang, W.-C. Chu, H.-R. Chen, C.-Y. Hsu, and M. Y. Chen, “EverTutor: Automatically Creating Interactive Guided Tutorials on Smartphones by User Demonstration,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2014, pp. 4027–4036.

[10] M. J. Lee, F. Bahmani, I. Kwan, J. LaFerte, P. Charters, A. Horvath, F. Luor, J. Cao, C. Law, M. Beswetherick, S. Long, M. Burnett, and A. J. Ko, “Principles of a debugging-first puzzle game for computing education,” in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2014, pp. 57–64.

[11] M. Eagle and T. Barnes, “Experimental Evaluation of an Educational Game for Improved Learning in Introductory Computing,” in *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, New York, NY, USA, 2009, pp. 321–325.

[12] “Blockly Games.” [Online]. Available: https://blockly-games.appspot.com/.

[13] J. J. G. Van Merrienboer and M. B. M. De Croock, “Strategies for Computer-Based Programming Instruction: Program Completion Vs. Program Generation,” *J. Educ. Comput. Res.*, vol. 8, no. 3, Jan. 1992.

[14] J. J. G. Van Merriënboer, “Strategies for Programming Instruction in High School: Program Completion vs. Program Generation,” *J. Educ. Comput. Res.*, vol. 6, no. 3, pp. 265–285, Jan. 1990.

[15] F. G. Paas, “Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach,” *J. Educ. Psychol.*, vol. 84, no. 4, pp. 429–434, 1992.

[16] J. Sweller, P. Ayres, and S. Kalyuga, *Cognitive Load Theory*. Springer, 2011.

[17] X. Zhu and H. A. Simon, “Learning Mathematics From Examples and by Doing,” *Cogn. Instr.*, vol. 4, no. 3, pp. 137–166, 1987.

[18] S. Garner, “A tool to support the use of part-complete solutions in the learning of programming,” in *Proceeding de conférence*, 2001.

[19] D. Parsons and P. Haden, “Parson’s Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses,” in *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, Darlinghurst, Australia, Australia, 2006, pp. 157–163.

[20] S. Palmiter, J. Elkerton, and P. Baggett, “Animated demonstrations vs. written instructions for learning procedural tasks: a preliminary investigation,” *Int J Man-Mach Stud*, vol. 34, no. 5, pp. 687–701, 1991.

[21] F. Grabler, M. Agrawala, W. Li, M. Dontcheva, and T. Igarashi, “Generating photo manipulation tutorials by demonstration,” in *ACM SIGGRAPH 2009 papers*, New Orleans, Louisiana, 2009, pp. 1–9.

[22] P.-Y. Chi, S. Ahn, A. Ren, M. Dontcheva, W. Li, and B. Hartmann, “MixT: Automatic Generation of Step-by-step Mixed Media Tutorials,” in *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2012, pp. 93–102.

[23] J. Fernquist, T. Grossman, and G. Fitzmaurice, “Sketch-sketch Revolution: An Engaging Tutorial System for Guided Sketching and Application Learning,” in *Proc. User Interface Software and Technology*, New York, NY, USA, 2011.

[24] B. Lafreniere, T. Grossman, and G. Fitzmaurice, “Community Enhanced Tutorials: Improving Tutorials with Multiple Demonstrations,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2013, pp. 1779–1788.

[25] W. Li, T. Grossman, and G. Fitzmaurice, “GamiCAD: A Gamified Tutorial System for First Time Autocad Users,” in *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2012, pp. 103–112.

[26] W. Li, T. Grossman, and G. Fitzmaurice, “CADament: A Gamified Multiplayer Software Tutorial System,” in *Proc. SIGCHI*, New York, NY, USA, 2014.

[27] H. Ramadhan, “An Intelligent Discovery Programming System,” in *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990’s*, New York, NY, USA, 1992, pp. 149–159.

[28] W. Sack, E. Soloway, and P. Weingrad, “From PROUST to CHIRON: Its design as iterative engineering: Intermediate results are important,” *Comput.-Assist. Instr. Intell. Tutoring Syst. Shar. Goals Complement. Approaches Lawrence Erlbaum Assoc. Hillsdale NJ*, pp. 239–274, 1992.

[29] E. Soloway, E. Rubin, B. Woolf, J. Bonar, and W. L. Johnson, “MENO-II: An AI-Based Programming Tutor.,” Aug. 1983.

[30] J. R. Anderson and E. Skwarecki, “The Automated Tutoring of Introductory Computer Programming,” *Commun ACM*, vol. 29, no. 9, pp. 842–849, Sep. 1986.

[31] A. T. Corbett and J. R. Anderson, “Student modeling and mastery learning in a computer-based programming tutor,” in *Intelligent Tutoring Systems*, Eds. Springer Berlin Heidelberg, 1992, pp. 413–420.

[32] W. Jin, “Pre-programming Analysis Tutors Help Students Learn Basic Programming Concepts,” in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, New York, NY, USA, 2008.

[33] “Khan Academy.” [Online]. Available: http://www.khanacademy.org.

[34] “Codecademy.” [Online]. Available: http://www.codecademy.com/.

[35] J. Moody, “Distance Education: Why Are the Attrition Rates so High?,” *Q. Rev. Distance Educ.*, vol. 5, no. 3, pp. 205–210, 2004.

[36] T. Dong, M. Dontcheva, D. Joseph, K. Karahalios, M. Newman, and M. Ackerman, “Discovery-based Games for Learning Software,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2012, pp. 2083–2086.

[37] “Lightbot.” [Online]. Available: http://lightbot.com/.

[38] “Code Avengers.” [Online]. Available: http://www.codeavengers.com/.

[39] “CodeCombat.” [Online]. Available: http://codecombat.com/.

[40] K. Hartness, “Robocode: using games to teach artificial intelligence,” *J Comput Small Coll*, vol. 19, no. 4, pp. 287–291, 2004.

[41] J. Long, “Just for Fun: Using Programming Games in Software Programming Training and Education--A Field Study of IBM Robocode Community,” *J. Inf. Technol. Educ.*, vol. 6, pp. 279–290, 2007.

[42] D. Seehorn, S. Carey, B. Fuschetto, I. Lee, D. Moix, D. O’Grady-Cunniff, B. B. Owens, C. Stephenson, and A. Verno, “CSTA K–12 Computer Science Standards: Revised 2011,” ACM, New York, NY, USA, 2011.

[43] A. Cockburn, P. Quinn, and C. Gutwin, “Examining the Peak-End Effects of Subjective Experience,” in *Proc. SIGCHI*, New York, NY, USA, 2015.

[44] “Intrinsic Motivation Inventory,” *Self-Determination Theory*. [Online]. Available: http://www.selfdeterminationtheory.org/questionnaires/10-questionnaires/50.

[45] F. G. W. C. Paas, J. J. G. Van Merrienboer, and J. J. Adam, “Measurement of cognitive load in instructional research,” *Percept. Mot. Skills*, vol. 79, no. 1, pp. 419–430, Aug. 1994.

[46] F. Paas, J. E. Tuovinen, H. Tabbers, and P. W. M. Van Gerven, “Cognitive Load Measurement as a Means to Advance Cognitive Load Theory,” *Educ. Psychol.*, vol. 38, no. 1, pp. 63–71, 2003.

[47] H. Van der Heijden, “User acceptance of hedonic information systems,” *MIS Q.*, pp. 695–704, 2004.

[48] M. Igbaria, J. Iivari, and H. Maragahh, “Why do individuals use computer technology? A Finnish case study,” *Inf. Manage.*, vol. 29, no. 5, pp. 227–238, Nov. 1995.

[49] J. C. Read, “Validating the Fun Toolkit: an instrument for measuring children’s opinions of technology,” *Cogn. Technol. Work*, vol. 10, no. 2, pp. 119–128, May 2007.