Comparing the Effectiveness of Online Learning Approaches on CS1 Learning Outcomes

Michael J. Lee and Amy J. Ko
Information School | DUB Group
University of Washington
{mjslee, ajko}@uw.edu

ABSTRACT
People are increasingly turning to online resources to learn to code. However, despite their prevalence, it is still unclear how successful these resources are at teaching CS1 programming concepts. Using a pretest-posttest study design, we measured the performance of 60 novices before and after they used one of the following, randomly assigned learning activities: 1) complete a Python course on a website called Codecademy, 2) play through and finish a debugging game called Gidget, or 3) use Gidget's puzzle designer to write programs from scratch. The pre- and post-test exams consisted of 24 multiple choice questions that were selected and validated based on data from 1,494 crowdsourced respondents. All 60 of our novices across the three conditions did poorly on the exams overall in both the pre-tests and post-tests (e.g., the best median post-test score was 50% correct). However, those completing the Codecademy course and those playing through the Gidget game showed over a 100% increase in correct answers when comparing their post-test exam scores to their pre-test exam scores. Those playing Gidget, however, achieved these same learning gains in half the time. This was in contrast to novices that used the puzzle designer, who did not show any measurable learning gains. All participants performed similarly within their own conditions, regardless of gender, age, or education. These findings suggest that discretionary online educational technologies can successfully teach novices introductory programming concepts (to a degree) within a few hours when explicitly guided by a curriculum.

Categories and Subject Descriptors
K.3.2 Computer Science Education: Introductory Programming, D.2.5 Testing and Debugging.

General Terms
Design; Human Factors; Measurement.

Keywords
Programming; debugging; educational game; computing education; learning outcomes; Gidget; Codecademy.

1. INTRODUCTION
In recent years, major efforts such as the Hour of Code and CS Education Week events have attracted millions of people, including celebrities and even the U.S. president, to try programming using many of the discretionary learning resources available for free online [4]. These resource include tutorial websites such as Codecademy [14] and CodeSchool [17], open-ended creative environments such as Scratch [49] and Alice [19,21], and educational games such as Gidget [30] and LightBot [46]. Users of these systems report that they enjoy these informal resources more than traditional coursework because they allow for flexibility in how they learn, they provide a better sense of retention of the material [7], and they are more motivating, engaging, and interesting than traditional classroom courses [20]. Some of these attitudes can be attributed to these resources' use of game mechanics such as scaffolded materials, structured mastery learning, concrete goals, and extrinsic incentives such as badges [77]. Furthermore, these online resources allow users to learn about programming in a safe environment at their own pace, which gives them the opportunity to clear up any of their negative misconceptions about programming or their ability to learn it, to something more positive [12].

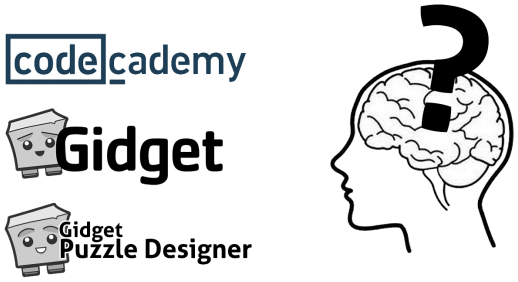

Figure 1. We examined if novice programmers produced measurable learning outcomes after using the three different types of discretionary, online learning tools shown here.

Although all these resources are undoubtably useful at attracting, exposing, and engaging new people in computer programming, few (if any) of these online resources report anything beyond the number of users that have signed up for their services and how many activities their users have completed. We do not know how long it takes learners to complete (or quit) the activities, if they ever come back, or, most importantly, what they are learning, if anything. This lack of evaluation makes it unclear how useful these tools are beyond merely engaging learners for a brief period of time, which resources are actually successful at teaching coding, or what parts of these resources contribute to success or failure. Without this knowledge, we risk designing instructional tools that do not actually instruct learners [28].

To investigate the learning outcomes of these online resources, we conducted a pretest-posttest experiment using three types of online educational technologies (see Figure 1), comparing the learning gains of each. We specifically compared the Python course on Codecademy [14], a debugging puzzle game called Gidget [30], and the open-ended creative environment found in Gidget called the Puzzle Designer [41], which is analogous to other creative development environments such as Scratch [49] and Alice [19,21]. We recruited learners aged 18 and above through Mechanical Turk.

ICER '15, August 9–13, 2015, Omaha, Nebraska, USA.
 ISBN 978-1-4503-3630-7/15/08…$15.00.
http://dx.doi.org/10.1145/2787622.2787709

In the rest of this paper, we discuss prior work on educational technologies, detail our test and study design, and discuss our results and their implications for online computing education.

2. RELATED WORK

This paper explores three major areas of work in educational technologies designed to teach beginners programming online: 1) open-ended, creative environments, 2) massively open online courses (MOOCs), and 3) educational games. Although these resources may differ in the way they deliver content and engage their users, there is little doubt that online learning will continue to be a major medium in 21st century computing education. This requires us not only to know about how these different instructional approaches perform in isolation, but also how they compare.

Open-ended, creative environments are largely unstructured and allow users to explore, tinker, and create content that is meaningful for themselves. These attributes align with constructivist theories of learning through hands-on experience [66] and constructionist ideas of learning through construction of meaningful projects [52]. Exemplars of these kinds of environments include Scratch [49] and Alice [19,21]. Prior work has shown that summer camps using these resources are great at engaging their users [10,73], but all of these reports required instructional scaffolding by teachers for learners to succeed.

MOOCs and self-paced learning resources such as CodeSchool [17], Codecademy [14], edX [25], and Khan Academy [35] attract millions of users and are an increasingly popular way for people to learn new skills such as programming. Many people view these approaches as connectivist learning, which is related to social learning theory (learning through social interactions and experiences). However, as Connolly and Stansfield [18] have suggested, many of these resources simply replicate the traditional classroom experience and may be too focused trying to deliver materials over the web rather than on teaching and learning, or motivating and engaging students [64].

Educators have considered games to be a beneficial platform in supporting student learning [55] and have pushed for more educational games to teach STEM (science, technology, engineering, and mathematics) subjects [32,56]. Games have been designed using a range of learning approaches, some constructivist (allowing learners to participate and experiment in non-threatening scenarios), some experiential (learning by doing), and some situated (providing relevant context or setting; for multiplayer, learning takes place alongside social interaction and collaboration). Video games are able to engage learners over extensive periods of time and can also motivate learners to replay the game repeatedly until they have mastered it [36]. This includes educational games such as CodeHunt [16], LightBot [46], CodeCombat [15], and Gidget [30]. Researchers have taken advantage of this interest to improve educational games to be more fun, informative, and educational, prompting more people to use games for both education and entertainment [29,54,76]. Some research focuses on creating games that directly try to teach a skill or subject such as computer programming [24,42,43,44], others focus on adding game-like features to existing teaching systems such as intelligent tutors [34,51], and some focus more generally on creating frameworks for effective evaluation [1,61]. Several works have also attempted to identify the specific parts of games that motivate [48] and attract people to pursue computing education [28,29,48].

Researchers and educators have evaluated the efficacy of many of these systems in isolation (e.g., Scratch [3,27], Alice [68], and edX [9]), and in comparison with other similar systems (e.g. Scratch vs. Karel [59]). However, only a few have examined how to effectively measure the outcomes of educational games [31,61], and very little is known about how online educational games actually compare to other technologies such as MOOCs and open-ended creative environments in teaching their users introductory programming concepts [13]. We aim to address this gap by comparing the learning outcomes of an open-ended creative environment, a self-paced MOOC, and an educational game.

3. METHOD

The goal of our study was to examine the extent to which adult novices of any age showed measurable learning gains after using one of three online learning technologies. To do this, we first selected three learning activities that are representative of the types of discretionary, online resources that people currently use to learn programming: 1) an online tutorial system using a web-based IDE, where learners go through a didactic, structured curriculum, 2) an educational game using an IDE, where learners go through a problem-based, structured curriculum, and 3) an open-ended creation IDE, where there is no planned curriculum and learners acquire skills by creating with code. Next, we developed a test designed to measure one's knowledge of different introductory programming concepts before and after completing one of the learning activities.

Our null hypothesis was:

> H_0: There is no difference in learners' *post-test performance* among the conditions after completing their assigned learning activity.

In the rest of this section, we describe our three learning activities in more detail, explain the design of our pre-post test, and discuss the experiment designed to test our hypothesis.

3.1 Learning Activities

3.1.1. Activity 1: Codecademy Course

Codecademy [14] is a popular online interactive tutorial website that offers free courses in multiple programming languages (see Figure 2). It has had over 24 million users who have completed over 100 million exercises [67]. For our study, learners participated in the introductory "Python Language Skills" course. According to the Codecademy website, over 2.5 million users are enrolled in this course designed for beginners. The website also states that the Python course takes an estimated 13 hours to complete.

Codecademy's course interface consists of a two-pane window split vertically on the screen (see Figure 2). The left pane consists of instructions, examples, and hints for the user to follow. For each activity, it contains a numbered list of explicit instructions for the user to follow (e.g., "01. Set the variable *my_varaiable* equal to the

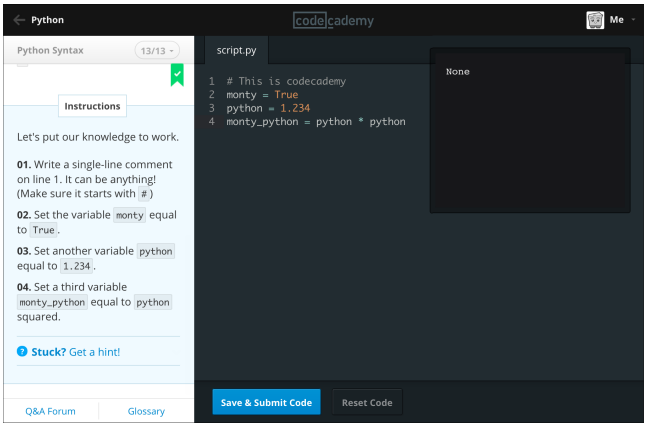

Figure 2. A Codecademy lesson, where users follow step-by-step instructions entering code into a virtual terminal.

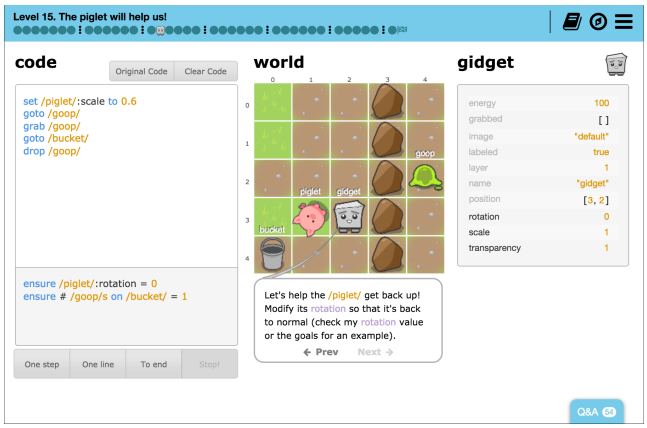

Figure 3. The Gidget game. Where players help a robotic character fix its code to complete 37 missions.

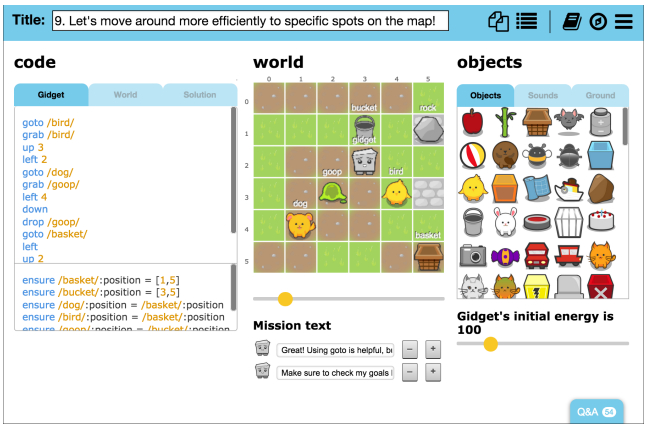

Figure 4. The Gidget Puzzle Designer, which players can use to create their own Gidget levels using a blank canvas.

value *10*" and "02. Click the *Save & Submit* button to run your code."). The right pane is an IDE for users to type in and execute their code, with an overlay on the upper-right corner that shows console output on code execution. In case learners need assistance, there is a "Stuck? Get a hint!" button below the list of instructions on the left pane that users can click to view more help text. The hints are typically explicit instructions (e.g., "All you need to do is type *3* after the equals sign on line 8.") or closely related examples (e.g., "Make sure you're setting your variable like this: *the_machine_goes = 'Ping!' "*.). Finally, the bottom-most area of the left pane includes two buttons that opens up a new browser tab: one labeled "Q&A forum" where fellow Codecademy users can post and answer questions, and another one labeled "glossary," that goes to a dictionary of Python commands and concepts.

The introductory Python course has a total of 12 modules covering the following topics: syntax, variables, mathematical and logical operator, strings, conditionals, control flow, functions, lists and dictionaries, and advanced concepts (e.g., classes and file input/ output). Each module is split into two parts. The first part is designed to teach a specific concept or set of concepts and consists of several activities that subsequently build on the previous activity. The second part of the section is an exercise to practice combining the first part to build something interesting. For example, in the case of the syntax module (where learners are introduced to variable assignment and the use of mathematical operators), the second part of the module tasks users to fill in variables with values to calculate gratuity for a meal.

To ensure that the concepts covered by Codecademy and the Gidget game (described below) were as close as possible, we asked learners to complete only the first 8 of 12 modules before taking our post-test. Although learners would not be tested on these extra advanced concepts on the post-test exam, finishing them would have given them additional practice with many of the previously learned concepts. We asked learners to keep track of the time they spent using Codecademy so that they could report their total time after taking the post-test exam. Since the Codecademy website states it takes around 13 hours to complete the 12 modules in the Python course, we informed our tutorial condition participants it would take approximately 10 hours to complete their assigned 8 modules before they started their activity.

3.1.2. Activity 2: Gidget Game

Gidget is a web application (see Figure 3) that has been played by thousands of people worldwide, with nearly half of its users being female [30,40]. The game was specifically designed to teach and appeal to both youth [41] and adult [12] novices, presenting debugging tasks as puzzles and using an imperative Python-like programming language. For this study, we asked learners to play through the entire game, which takes approximately 5 hours [41]. In each level of the game, the player must identify the level goals (written as test cases), inspect the given code, then modify and execute it until it satisfies all the level's test cases. Following the mastery learning paradigm [53], each of the game's levels is designed to be passable only if the learner has grasped a particular concept in the game's programming language.

Gidget's interface consists of three vertically-split sections (see Figure 3). The left section consists of the IDE to type in code, the list of goals (written as test cases that are checked after code execution), and the execution buttons. The execution buttons allow the player to control the level of execution (e.g., one compiled instruction, all instructions on one line of code like a breakpoint debugger, or the entire program), or halt the program. The central section shows the graphical representation of all the characters and objects in the game world, and also includes a large speech bubble where the game's protagonist provides detailed explanation of the execution of each statement in the program, highlighting changes in the runtime environment. This serves as the game's primary instructional content, explicitly teaching the language syntax and semantics. Finally, the right section updates after each instruction, showing the current runtime state of all the game's current objects and their respective variables.

The game had a total of 7 modules totaling 37 levels, with each module containing a set of levels focusing on a related set of programming keywords or concepts. The game covered exactly the same topics as the Codecademy modules listed earlier, excluding Python dictionaries and the "advanced topics," which we asked Codecademy participants to skip so the two learning activities would be as similar as possible. Each module was split into two parts, where each level in the first part (between 3-5 levels) had a specific learning objective to familiarize the player with a specific programming concept. The second part of each module included two assessment levels where learners did not have to edit code, but had to answer a question that cumulatively tested the concepts for that module. This was found in prior work to improve adult players' engagement and subsequent level completion speed [44].

Gidget included several ways for players to receive help. After signing up, the game presents the player with a tutorial highlighting and explaining the different parts of the interface and the sequence of steps players should take to proceed through each level. The game also features an in-game reference guide, providing explanations and examples of each command in the language. The

reference guide was available as a standalone help guide or as tooltips that appeared when hovering over tokens in the code editor. This was further enhanced by the inclusion of the Idea Garden [11], which analyzed the players' code in real-time and presented context sensitive suggestions if the player requested it, and AnswerDash [2], which allowed players to click on any part of the interface to ask questions about it or read responses to others' queries. Finally, the game's code editor provided keystroke-level feedback about syntax and semantics errors, underlining erroneous code in red and explaining the problem in Gidget's speech bubble.

Based on prior studies [40,41], we told our game condition participants that Gidget would take about 5 hours to complete before they started the activity. We required learners to complete all the levels before taking the post-test. For this specific condition, we automatically logged the time learners took to complete the game.

3.1.3. Activity 3: Gidget Puzzle Designer

The Gidget Puzzle Designer (GPD) is an integrated development environment used to create and edit Gidget levels (see Figure 4). It is normally unlocked after finishing the Gidget game. However, for our study, participants were given access to the GPD without any prior experience playing the Gidget game. This was to mirror other open-ended, creation-oriented learning environments like Scratch [49], Alice [19,21], and others [37], where users are free to explore and tinker to make their own projects.

The interface for the GPD is a modified version of the regular Gidget game interface, allowing modification of previously un-editable code such as the starting world code, the level goals, the dimension of the world grid, and Gidget's introductory dialogue and emotional state at the beginning of the level. In addition, the status pane on the rightmost section is replaced by a tabbed inventory of available characters and objects, ground tiles, and sounds that the learner can use to populate and enrich their programs.

All of the same help tools available in the Gidget game are also available in the GPD. This includes the syntax highlighting, tooltips, dictionary, and Idea Garden suggestions. In addition to the help systems, the learners had access to view and edit all of the regular game levels, giving them pre-designed puzzles to modify for creative purposes. These examples also included the solution (i.e., learners could see both the incorrect code and the correct code) for each level. The assessment levels from the end of each module were excluded, and all the default editable levels were listed in sequential order without indicating which module they belonged to. Similar types of help and examples are available in both Scratch and Alice to help bootstrap learner engagement.

Unlike Codecademy and the Gidget game, the GPD did not have a curriculum or sequence of steps to follow. Therefore, to help orient our GPD users, we showed them a list of directions before they started with their activity. First, we told them their task was to "Use a creative canvas tool to create multiple stories for a robotic character named Gidget." This is based on several works, primarily by Kelleher et al. [38,39], which shows that adding storytelling elements to open-ended creative environments can significantly increase users' engagement [33,60,72]. Second, we told them about the various help features available (see previous paragraph and section 3.1.2), and how to access them. Third, we asked them to "create, explore, and play with the website for at least *several hours* to get the full learning experience" with the activity, to mirror the ideal case of a learner first engaging with an open-ended, creative online environment. For this condition, we automatically logged the time learners spent in the GPD, and collected records of all the levels they created.

3.2 Knowledge Test for C1 Concepts

In order to measure how much participants learned and what they learned, we created and validated a test designed to be taken before and after the learning activities. We adopted this pre-test/post-test design as it widely used in both educational and non-educational contexts to measure change resulting from experimental treatments [6,13,23]. Although we spent considerable time creating and validating the test, its description will mostly be limited to this section as it is not the main contribution of this paper.

First, we determined which concepts to test by comparing the topics that are taught commonly in introductory programming courses [22,26,45,69,78] to the set of concepts that were covered in our Codecademy and Gidget game activities. We chose a total of eight concepts: basics (i.e., variables, mathematical operators, relational operators, Booleans), logical operators, selection statements (i.e., conditionals), arrays, indefinite loops (i.e., while), definite loops (i.e., for), function parameters, and function returns.

We modeled our test questions after Allison Tew's dissertation work on the FCS1, a programming language-independent test using pseudo-code [69]. In her studies, Tew showed that testing introductory programming students in the classroom with their native course language and in pseudo-code were strongly correlated [70] and has the extra benefit of demonstrating transfer of learning [8]. We generated pseudo-code questions using the examples, descriptions, and two-page pseudo-code guide Tew provided [69]. Questions used a verbose style adapted from guides for programmers published by Whitford [75] and Shackelford [62]. To minimize confounding factors in syntax design, we followed the latest evidence on syntax learnability, excluding semi-colons and curly braces, indenting code blocks, upper-casing reserved words, and closing program blocks with explicit keywords [63] (see Figures 5 and 6 for examples).

Based on guidelines and examples from Tew's dissertation, we designed 5 multiple choice questions for each of the concepts

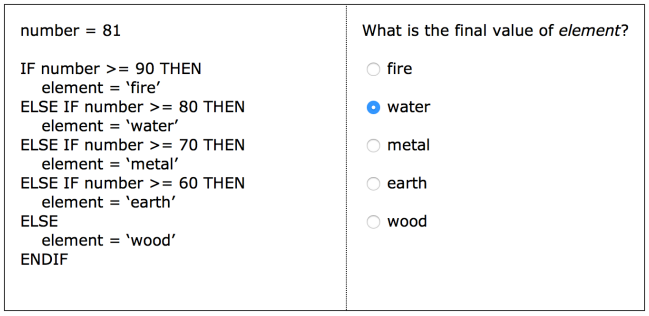

Figure 5. Screenshot of an "if/else" pseudo-code question from the pre- & post-tests with its answer choices.

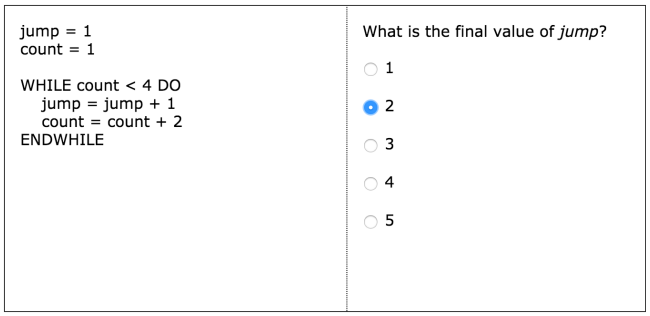

Figure 6. Screenshot of a "while" pseudo-code question from the pre- & post-tests with its answer choices.

Table 1. Demographic summary.

	Tutorial n=20	Game n=20	Canvas n=20
Gender (male : female)	10 : 10	11 : 9	11 : 9
Age (min, median, max)	18, 23, 35	18, 25, 41	19, 23, 29
Max education: high school	5%	0%	0%
Max education: some college	10%	10%	5%
Max education: college degree	85%	85%	90%
Max education: master degree	0%	5%	5%

covered in our learning activities, for a total of 40 questions. All questions had one correct response and four incorrect distractors. We designed distractors to deliberately test for common programming misconceptions [5,71].

To validate our 40 questions, we recruited people on Mechanical Turk (MTurk). MTurk is an online marketplace where individuals aged 18 and over (called "Turkers") can receive micro-payments for doing small tasks. Our Turkers were paid 2 cents to answer one pseudo-code question, indicate their experience with programming, and optionally provide their email address. No additional demographic information was collected. Each Turker could answer up to an additional 39 questions for 2 cents each. In these cases, each additional question would be new, and the Turker did not have to re-enter their answers for programming experience or their e-mail address (if provided previously). To mitigate ordering effects, questions were randomly sequenced each time a participant took the survey. Answer choices for questions that did not require a specific order were randomly arranged as well.

To identify problems with our questions and answer choices, we ran two rounds of pilot tests, with each question getting at least 3 responses for each iteration of testing. We corrected issues dealing with ambiguous/confusing wording, inappropriate distractors, syntax errors, and typos. To achieve this, we looked for data anomalies (e.g., nobody getting the answer correct, or everyone choosing the same answer) and requested open-ended feedback from our respondents. We then ran a full test with 1,494 Turkers and had a total of 8,011 responses to our questions (approximately 200 responses per question). The majority of our Turkers only answered one question, with 11% completing 3 or more questions.

To avoid ceiling and floor effects and to maximize discriminability of the assessment, we categorized our data by splitting responses by the Turkers' self-reported programming experience. We categorized *novices* as those who responded "never" to all of the following statements: 1) "taken a programming course," 2) "written a computer program," and 3) "contributed code towards the development of a computer program." All other respondents were considered *experienced* programmers. For our finalized list of exam questions, we selected the top 3 questions for each concept (for a total of 24) with highest variance between novice and experienced programmers (that is, those that novices tended to get incorrect and those with experience tended to get correct).

3.4 Participants and Procedure

The independent variable in our experiment was the instructional approach, which had three levels: 1) *tutorial* (complete the introductory Python programming tutorial on Codecademy), 2) *game* (play through the Gidget game), or 3) *canvas* (use the GPD to create Gidget levels). To help participants make an informed decision about the time commitment required to participate in our study, we told them that they were allowed seven days to complete

Table 2. Summary statistics pre-test and post-test scores.

	Tutorial n=20	Game n=20	Canvas n=20
Minimum score on pre-test	2	0	3
Median score on pre-test	5	5	5.5
Maximum score on pre-test	8	6	9
Minimum score on post-test	6	4	3
Median score on post-test	12	10	5
Maximum score on post-test	18	16	9
Percent increase between median pre-test and post-test scores	140%	100%	-9.1%

their assigned task, and provided an estimate of the number of hours their task would take (10 hours for the tutorial condition, 5 hours for the game condition, and open-ended for the canvas condition). We emphasized these hours were estimates, and that they could potentially take more or less time than what was listed.

We recruited our participants from Mechanical Turk, specifically sampling adults who self-reported that they had no experience with programming (see previous section). We also required participants to be U.S. residents to minimize English language barriers with the instructions and activities. Participants were compensated $10 for completing their assigned task. This amount was carried over from a previous study [44] and adjusted to account for the extra time required for the pre-test and post-test.

We sent participants an e-mail with a link that randomly assigned them to a condition and redirected them to the web-based pre-test. Each link was uniquely associated with a specific e-mail address, so that we could identify the owner of each test. Like our pilot study (see section 3.2), we randomly ordered our finalized collection of 24 questions to minimize ordering effects, also randomizing the order of the answers, where appropriate. The test only showed one question at a time (see Figures 5 and 6) and it was not possible to go back to a previous question. Each question required a response before being able to move onto the next question. There was a progress indicator on the top of the page showing participants how many questions remained. The system automatically logged each answer choice and the total time to complete the exam(s).

The pre-tests and post-tests were identical across all conditions. There was only one exception to this: the post-test for those in the tutorial condition had two additional questions for the participants to report how many modules they completed, and the time they spent to complete their Codecademy activity. The introductory text for the pre-test briefly explained that participants would be answering coding questions and that they should try their best even though they might not be familiar with the content. The introductory text for the post-test briefly explained that the questions were written in another, related programming language that covered the same concepts available in the learning activity they had completed.

Our study was a between-subjects design, with an even split of 20 people each among the three conditions. Our participants did not differ significantly by gender, age, or education (see Table 1). Consistent with other studies about the demographics of MTurk workers [57], we found that our participants were well-educated, with the majority reporting that they had at least a bachelor's degree (see Table 1).

Table 3. Percent increase between pre & post -test scores. Groupings with a mean >= 100% are in bold. Groupings with a mean >= 150% are also *italicized* in *red*.

Question + Concept (actual question ordered randomly)		(posttest - pretest) / pretest		
		Tutorial	**Game**	**Canvas**
Q1	basics	**175%**	60%	-40%
Q2	basics	**120%**	60%	0%
Q3	basics	**100%**	50%	0%
Q4	logical operators	***175%***	**133.3%**	-66.7%
Q5	logical operators	***125%***	**120%**	-40%
Q6	logical operators	***150%***	**166.7%**	-20%
Q7	if / else	100%	100%	20%
Q8	if / else	100%	80%	0%
Q9	if / else	80%	100%	0%
Q10	arrays	75%	100%	-33.3%
Q11	arrays	60%	50%	-40%
Q12	arrays	100%	50%	-33.3%
Q13	while	***225%***	***166.7%***	-60%
Q14	while	***333.3%***	***266.7%***	0%
Q15	while	***266.7%***	***125%***	0%
Q16	for	**100%**	**66.7%**	-50%
Q17	for	**200%**	**133.3%**	0%
Q18	for	**80%**	**100%**	-40%
Q19	function parameters	***140%***	***166.7%***	25%
Q20	function parameters	***233.3%***	***200%***	0%
Q21	function parameters	***233.3%***	***200%***	25%
Q22	function return	**166.7%**	***200%***	-50%
Q23	function return	**80%**	***166.7%***	-33.3%
Q24	function return	**125%**	***160%***	0%

4. RESULTS

We provide quantitative results comparing the learning outcomes from our three groups. Throughout this analysis, we use non-parametric Chi-Squared and Wilcoxon rank sums tests with α=0.01 confidence, as our data were not normally distributed. For post-hoc analyses, we use the Bonferroni correction for three comparisons: ($\alpha/3 = 0.0033$).

4.1 Better Post-Scores with Tutorial & Game

Overall, participants did poorly on the pre-test exams, with a median score of 5 out of 24 questions correct (20.8%) across all three conditions (see Table 2). This was expected, as we had selected the questions most difficult for novices from our original set. We compared the pre-test scores across the conditions and found no significant difference, confirming that all of our participants' programming knowledge was roughly equivalent prior to the learning activities.

Participants also did poorly on the post-tests, with the highest median score among the conditions being 12 out of 24 questions correct (50%). However, comparing the post-test scores across the conditions reveal that there is a significant difference in learning gains between conditions ($\chi^2(2,N=60)=27.03, p<.01$). Post-hoc analysis with Bonferroni correction revealed that two conditional pairs were significantly different: the tutorial vs. canvas conditions ($W=226, Z=-5.00$, $p<.01/3$) and the game vs. canvas conditions ($W=272.5, Z=-3.72$, $p<.01/3$). The scores on the post-test between the tutorial and game conditions did not show a significant difference. Based on these findings, we reject our null hypothesis (see section 3).

These results indicate that though all the participants had approximately the same programming knowledge during the pre-test, participants from the *tutorial* and *game* condition performed significantly better on their post-test, and that their degree of improvement was also significantly greater than that of the *canvas* condition. As seen in Table 2, the effect sizes of learning gains were 140% and 100% increase in scores for the *tutorial* and *game* conditions, respectively, whereas the median score from the *canvas* condition did not change significantly (and were actually 9.1% worse). Since participants had little programming knowledge to start with and there was no difference in demographics, the learning activities are likely the primary cause of the increase in scores for the tutorial and game condition participants.

4.3 Differences in Percent Increase of Scores

Although we had a relatively small sample size of 20 participants per condition, we found consistent patterns, particularly in the tutorial and game conditions, where participants made large percent gains answering questions correctly in their post-tests compared to their pre-tests (see Tables 3 and 4). As we saw in section 4.1, the tutorial and game condition participants performed much better than their canvas condition counterparts. This was particularly true for the basic concepts (i.e., variables, mathematical operators, relational operators, Booleans), logical operators, while loops, for loops, function parameters, and function returns, where participants increased their rate of correct answers by at least 100% in their post-test compared to their pre-test.

Tutorial and game condition participants made the largest improvements (greater than or equal to 150% increase) with *while loop* and *function parameters* concepts. Tutorial condition participants also made these large improvements answering questions about *logical operators*, while the game condition participants also made similarly large improvements answering questions about *function return*s. These results indicate that the tutorial and game conditions' learning activities were successful in helping their participants learn about all the concepts we tested for.

Canvas condition participants did not do well compared to their counterparts. Although we know from section 4.1 that the canvas condition participants did not do significantly worse on their post-tests compared to their pre-tests overall, Table 3 shows that they struggled answering many of the post-test questions, actually performing worse on many concepts in the post-test, despite encountering the identical questions.

Table 4. Summary statistics for activity times.

	Tutorial n=20	**Game** n=20	**Canvas** n=20
Minimum time on pre-test	20 min	22 min	20 min
Median time on pre-test	25.5 min	28 min	26 min
Maximum time on pre-test	33 min	31 min	41 min
Minimum time on activity	7.0 hours	3.61 hours	1.25 hours
Median time on activity	9.25 hours	4.76 hours	1.94 hours
Maximum time on activity	14.0 hours	7.22 hours	2.98 hours
Minimum time on post-test	23 min	29 min	19 min
Median time on post-test	35 min	34 min	24 min
Maximum time on post-test	55 min	42 min	35 min

These results indicate that online, educational tutorial and game resources can be successful at teaching users about programming concepts, but that open-ended creative resources in discretionary settings, at least in solitary, are likely not. Tutorial and game condition participants' scores indicate that there are large, measurable learning outcomes (see bolded text in Table 2), and that these learning activities might teach certain concepts better than others (see above and italicized text in Table 3).

4.3 More Time on Exams for Tutorial & Game

During the pre-test, participants from all conditions spent roughly the same amount of time on their exams (see Table 4). However, when we examine the time they spent on their post-test, there is a significant difference in time spent by condition ($\chi^2(2,N=60)=17.87,p<.01$). Doing post-hoc analysis with Bonferroni correction, we found that the tutorial participants spent significantly more time on the post-test than the canvas condition ($W=288.5$, $Z=-3.29,p<.01/3$); the same was true of the game vs. canvas conditions ($W=263.5,Z=-3.96$, $p<.01/3$). The time spent on the post-test between the tutorial and game conditions did not show a significant difference.

4.4 Differences on Learning Activity Time

Each of the three learning activities had largely different estimated times for completion (10 hours for Codecademy, 5 hours for Gidget, and open-ended for the GPD). Examining the time spent on each task (see Table 4), we see that there was indeed a significant difference in time participants spent on their respective activities ($\chi^2(2,N=60)=52.34$, $p<.01$). With a post-hoc analysis with Bonferroni correction, we found that all pairwise comparisons were significantly different: tutorial vs. canvas ($W=210$, $Z=-5.40$, $p<.01/3$), tutorial vs. game ($W=211$, $Z=-5.37$, $p<.01/3$), and game vs. canvas ($W=210$, $Z=-5.40$, $p<.01/3$).

Combined with the large difference on exam performance in the post-test and pre-test (from section 4.1), this suggests that the *game* condition was the most efficient of the three conditions at improving participants' post-test scores. Examining Table 4 reveals that the tutorial condition participants took nearly twice as long as the game condition participants to complete their assigned learning activity that covered the same materials (see sections 3.1.1 and 3.1.2), but performed similarly in their post-test (from section 4.2), further demonstrating that the game condition participants were most efficient at improving their post-test scores.

4.5 No Demographic Differences in Test Scores

We found that there were no significant differences in learning gains within the groups by gender. This indicates that males and females all performed similarly within their respective conditions.

Next, we used a simple linear regression for each condition's pre-test and post-test to predict test scores based on age. No significant correlation was found between test scores or age for any of the conditions in either of the tests. This indicates everyone performed similarly within their respective conditions, regardless of their age.

For completeness, we also examined if prior education (as measured in Table 1) had an effect on pre-test and post-test scores by condition. We found no significant differences within groups by education. This indicates everyone, regardless of education, performed similarly within their respective conditions.

4.6 No Demographic Differences in Test Time

We examined if gender had any effect on the time participants spent on the pre-tests and post-tests by condition. We found that there were no significant differences within the groups by gender. This indicates that males and females all spent a similar amount of time on their tests within their respective conditions.

Next, we used a simple linear regression for each condition's pre-test and post-test to predict the time spent on tests based on age. No significant correlation was found between the time spent on tests and age for any of the conditions in either of the tests. This indicates everyone spent a comparable amount of time on their tests within their respective conditions regardless of their age.

Finally, we examined if education had any effect on the time spent on the pre-tests and post-tests by condition. We found no significant differences within groups by participants' level of education. This indicates everyone, regardless of their level of education, spent a similar amount of time on their tests within their respective conditions.

5. DISCUSSION

Our findings show that online discretionary resources for computing education such as tutorial websites and games can be successful in teaching adult novices programming concepts without the need for additional external help. Even with relatively small sample sizes, we were able to see large differences in the time players spent on the learning activities, the exams, and their exam scores. All participants performed consistently within their own groups, without any significant differences in the time they spent on either the pre-tests or post-tests, the time on their learning activities, or on their exam scores. This consistency is also reflected in participants' demographics, which showed no differences between males or females, people of different ages, or level of education, within all conditions. This is particularly important for online discretionary learning, because our results indicate that all of our learning activities were gender-neutral, with everyone performing at equal levels within their respective conditions, which does not typically happen in programming-related classroom settings [58,74].

We found that participants in the tutorial and game conditions significantly increased their overall post-test scores by over 100% in comparison to their pre-test scores (see Table 2). These participants showed considerable gains for similar questions (see Table 3), suggesting that the learning activities from both the conditions taught similar concepts and also taught them equally well. Although these participants showed improvements across all the concepts we tested (see Table 3), the highest increases were in: basics, logical operators, while loops, for loops, function parameters, and function returns. Moreover, participants from the tutorial condition appeared to do slightly better on logical operator questions while participants from the game condition did slightly better on the function return questions. We examined the instruction of these two concepts in both Codecademy and the Gidget game, but did not find anything obviously different from the modules teaching those specific concepts from the other concepts within the same learning activity. Like the rest of the interactions within those groups, Codecademy had its users follow step-by-step instructions entering code into its IDE, and the Gidget game required participants to look through, diagnose, and fix broken code like every other level. None of these findings were true for the canvas participants, indicating this condition's learning activity failed to teach the same concepts even though all the necessary help resources were available to users.

We did not find any significant difference in the time participants spent on their pre-test exams. However, participants in the tutorial and game conditions spent significantly more time on their post-tests compared to their canvas condition counterparts. This suggests that those in the tutorial and game condition found more reason to concentrate and take their time on their respective post-test exams, possibly because they were better equipped to answer the questions correctly. Conversely, without clear goals or instruction in the Gidget Puzzle Designer, participants in the canvas condition likely did not learn the concepts necessary for them to engage successfully with the post-test.

Unfortunately, we found that none of the conditions led to substantial learning of the programming concepts (i.e., the highest median score for a post-test was 12 out of 24 questions correct). Although it is understandable that novices scored poorly on the pre-test since it was their first time seeing programming code, examining the overall scores for both the pre-test and post-test exams may give the impression that the exams were too difficult. However, novices performing badly on programming-related concepts they recently learned is not uncommon [5,47,50,65]. Comparable scores were also reported by Tew who administered a similar pseudo-code test to students who had taken entire introductory programming courses [70]. Since our test questions were generated by combining scores from a crowdsourced group of adult novices and experienced programmers, our results suggest that there is a major gap in programming knowledge between beginners and those with more experience, and that one short exposure with code, whether it be an online tutorial, online game, or even a formal high school or university class [70], may be enough to show some learning outcomes, but is still far from mastery of the subject.

There was a large difference in the time people spent on their respective learning activities. Learners spent the most time on the Codecademy course and spent the least amount of time using the Gidget Puzzle Designer. The time participants spent on the Codecademy and Gidget game tasks are not surprising, given that they are close to the developers' estimated time to complete the activity. It is also not too surprising that participants spent the least amount of time on the Gidget Puzzle Designer task. Without any clear goals or instructions, the Gidget Puzzle Designer participants likely lacked the motivation required to go beyond tinkering with the interface a bit. This means that goals are important for engagement with an activity, and that without proper motivation, people are likely to disengage with the activity. This is particularly worrisome for discretionary learning resources, because one negative/boring encounter with programming might cause a lasting impression where a learner decides that computer science is not for them based on this one experience. More generally, it may be that open-ended, but solitary creative learning tasks such as these fail to engage online with more substantial extrinsic motivators, such as teachers, online community, and more directed creative tasks.

5.1 Threats to Validity

Our study has several limitations that may limit its generalizability. First, we recruited all of our participants through MTurk from the USA. Our participants were all adults, aged between 18 to 41 years old. They were also highly educated, with 51 of our 60 participants having a college degree or higher. This may have introduce a sampling bias, which may limit the generalizability of our results to the particular adult populations found on MTurk who want to learn how to code. Since both Codecademy and Gidget were designed for people of all ages, future work could examine a wider demographic, including those without college degrees, youth, and people living outside of the USA.

We gave participants up to 7 days to complete their assigned activity. Although we asked them to refrain from using any other resources to learn or practice programming, participants could have potentially learned coding concepts from other places, even if it was unintentional. Learning or practicing programming concepts outside of the assigned task could have potentially affected exam outcomes, but could have happened in all conditions. However, unlike the numerous resources to get guidance for Python, there are no external resources to get explicit help for Gidget.

Participants spent significantly different times on their task by condition. Although the numbers for the Codecademy and Gidget game are similar to the estimates given by their developers, the extra time they had may have contributed to their performance gains over the GPD. However, we see that Gidget players performed just as well as Codecamy players, even though they spent significantly less time. There is also no evidence that GPD players would have learned anything more by being forced to play longer since the GPD did not provide any guided direction. Codecademy and Gidget users might not have been as successful on their post-tests if they spent less time and/or did not finish their task. However, our data shows that unlike the GPD players, these users were engaged enough to continue through their entire task for hours, and showed improvements in all the concepts that their assigned tasks covered.

Part of our Codecademy data was reliant on self-reported data that participants provided, including the time they spent on the task and how far they got in the course. Although we asked participants to stop at the "advanced concept" modules so the learning interventions were as similar as possible, we had no way of enforcing that since Codecademy is a third-party website.

Finally, there was an economic incentive for participants to participate in the study. We tried to minimize this effect as much as possible. We believe that the economic incentive in our study was minimal, as usage data for both Gidget and Codecademy show that thousands and millions of people have used these systems without being paid to play.

6. CONCLUSIONS & FUTURE WORK

We investigated the learning outcomes of three different types of programming resources designed for beginners. By comparing the test scores of learners before and after their respective learning activities, we found that the learners who took a Codecademy course and the learners who played through the Gidget game showed considerable improvement in their test scores. Though this was true of both cases, learners who played the Gidget game were able to match the post-test performance of learners who completed the Codecademy tutorial, in approximately half the time. Furthermore, we found that participants from both of these groups also spent more time on the post-test exam, suggesting that they found reason to try harder the second time taking the exam. In contrast, those who were assigned to create programs from scratch using the Gidget Puzzle Designer spent approximately the same amount of time on their pre-test and post-test exams, and did not show significant improvements in their post-test exam scores. In addition to these differences, we found that performance by demographics was consistent within all the conditions, meaning our learning activities had a similar impact regardless of gender, age, or level of education.

These findings raise many questions for future work. How might creativity-oriented online learning environments such as Scratch better support learners? Is it really the case that creative environments need a teacher, or can they be designed to teach effectively without human teachers? How might these findings apply to other forms of discretionary computing education such as MOOCs that are modeled more like traditional classroom settings? With the rising interest in learning to program, and the proliferation of resources to do so, we believe that knowledge about the interaction between the design of these resources and engagement and learning will be essential for learner retention and effective pedagogy.

7. ACKNOWLEDGEMENTS

We thank our participants. This work was supported in part by the National Science Foundation (NSF) under grants CNS-1240786, CNS-1240957, CNS-1339131, CCF-0952733, CCF-1339131, IIS-1314399, IIS-1314384, and OISE-1210205. Any opinions, findings, conclusions or recommendations are those of the authors and do not necessarily reflect the views of NSF.

8. REFERENCES

1. Aleven, V., Myers, E., Easterday, M., & Ogan, A. (2010). Toward a framework for the analysis and design of educational games. *IEEE DIGITEL*, 69-76.
2. Answerdash. http://www.answerdash.com. Accessed: 2015-03-26.
3. Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From scratch to "real" programming. *ACM TOCE*, 14(4), 25.
4. Beres, D. (2014). Obama Writes His First Line Of Code. Retrieved 2015-02-08, from http://www.huffingtonpost.com/2014/12/09/obama-code_n_6294036.html
5. Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human–Computer Interaction*, 1(2), 133-161.
6. Bonate, P.L. (2000). Analysis of pretest-posttest designs. *CRC Press*.
7. Boustedt, J., Eckerdal, A., McCartney, R., Sanders, K., Thomas, L., & Zander C. (2011). Students' perceptions of the differences between formal and informal learning. *ACM ICER*, 61–68
8. Bransford, J.D., Brown, A.L., & Cocking, R.R. (1999). How people learn: Brain, mind, experience, and school. *National Academy Press.*
9. Breslow, L., Pritchard, D.E., DeBoer, J., Stump, G.S., Ho, A. D., & Seaton, D.T. (2013). Studying learning in the worldwide classroom: Research into edX's first MOOC. *Research & Practice in Assessment*, 8(1), 13-25.
10. Bruckman, A., Biggers, M., Ericson, B., et al. (2009). Georgia computes!: Improving the computing education pipeline. *ACM SIGCSE Bulletin*, 41(1), 86-90.
11. Cao, J., Kwan, I., White, R., Fleming, S.D., Burnett, M., & Scaffidi, C. (2012). From barriers to learning in the Idea Garden: An empirical study. *IEEE VL/HCC,* 59-66.
12. Charters, P., Lee, M.J., Ko, A.J., & Loksa, D. (2014). Challenging stereotypes and changing attitudes: the effect of a brief programming encounter on adults' attitudes toward programming. *ACM SIGCSE*, 653-658.
13. Chumley-Jones, H.S., Dobbie, A., & Alford, C.L. (2002). Web−based learning: Sound educational method or hype? A review of the evaluation literature. *Academic medicine*, 77(10), S86-S93.
14. Codecademy. http://www.codecademy.com. Accessed: 2015-03-26.
15. Code Combat. http://www.codecombat.com. Accessed: 2015-03-26.
16. Code Hunt. http://www.codehunt.com. Accessed: 2015-03-26.
17. Code School. http://www.codeschool.com. Accessed: 2015-03-26.
18. Connolly, T.M., & Stansfield, M.H. (2006). Enhancing eLearning: Using Computer Games to Teach Requirements Collection and Analysis. WG HCI & UE of the Austrian Computer Society.
19. Cooper, S., Dann, W., & Pausch, R. (2000). Alice: a 3-D tool for introductory programming concepts. *J. of Computing Sciences in Colleges*, 15(5), 107-116.
20. Cross, J. (2006). Informal learning: rediscovering the natural pathways that inspire innovation and performance. *San Francisco, CA: Pfeiffer.*
21. Dann, W.P., Cooper, S., & Pausch, R. (2011). Learning to Program with Alice. *Prentice Hall Press*.
22. Deitel, H., & Deitel, P. (2005). C++: How to program (5th ed.). *Upper Saddle River, NJ: Prentice Hall.*
23. Dimitrov, D.M., & Rumrill, Jr, P.D. (2003). Pretest-posttest designs and measurement of change. *Work: A Journal of Prevention, Assessment and Rehabilitation*, 20(2), 159-165.
24. Eagle, M., & Barnes, T. (2009). Experimental evaluation of an educational game for improved learning in introductory computing. *ACM SIGCSE Bulletin*, 41(1), 321-325.
25. edX. https://www.edx.org. Accessed: 2015-03-26.
26. Felleisen, M., Findler, R. B., Flatt, M., & Krishnamurthi, S. (2001). How to design programs: An introduction to programming and computing. *Cambridge, MA: MIT Press.*
27. Franklin, D., Conrad, P., Boe, B., et al. (2013). Assessment of computer science learning in a scratch-based outreach program. *ACM SIGCSE*, 371-376.
28. Garris, R., Ahlers, R., & Driskell, J.E. (2002). Games, motivation, and learning: A research and practice model. *Simulation & Gaming*, 4, 441–467.
29. Gee, J.P. (2014). What video games have to teach us about learning and literacy. *Macmillan*.
30. Gidget. http://www.helpgidget.org. Accessed: 2015-03-26.
31. Hamari, J., Koivisto, J., & Sarsa, H. (2014). Does gamification work? – a literature review of empirical studies on gamification. *HICSS*, 3025-3034.
32. Hays, R.T. (2005). *The effectiveness of instructional games: A literature review and discussion* (Technical Report 2005-004). Naval air warfare ctr. training systems division. Orlando, FL.
33. Ivala, E., Gachago, D., Condy, J., & Chigona, A. (2013). Enhancing student engagement with their studies: a digital storytelling approach. *Creative Education*, 4(10), 82.
34. Kapp, K.M. (2012). The gamification of learning and instruction: game-based methods and strategies for training and education. *San Francisco, CA: Pfeiffer.*
35. Khan Academy. http://www.kahnacademy.com. Accessed: 2015-03-26.
36. Kirriemuir, J., & McFarlane, A. (2004). Literature Review in Games and Learning. Report 8, NESTA, Futurelab, Bristol.
37. Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM CSUR*, 37(2), 83-137.
38. Kelleher, C., & Pausch, R. (2007). Using storytelling to motivate programming. *Comm. of the ACM*, 50(7), 58-64.
39. Kelleher, C., Pausch, R., & Kiesler, S. (2007). Storytelling Alice Motivates Middle School Girls to Learn Computer Programming. *ACM CHI*, 1455-1464.
40. Lee, M.J. (2015). Teaching and Engaging with Debugging Puzzles. *PhD dissertation, University of Washington.*
41. Lee, M.J, Bahmani, F., Kwan, I., Laferte, J., Charters, P., Horvath, A., Luor, F., Cao, J., Law, C., Beswetherick, M., Long, S., Burnett, M., & Ko, A.J. (2014). Principles of a Debugging-First Puzzle Game for Computing Education. *IEEE VL/HCC*, 57-64.
42. Lee, M.J., & Ko, A.J. (2011). Personifying programming tool feedback improves novice programmers' learning. *ACM ICER,* 109-116.
43. Lee, M.J., & Ko, A.J. (2012). Investigating the role of purposeful goals on novices' engagement in a programming game. *IEEE VL/HCC,* 163-166.
44. Lee, M.J., Ko, A.J., & Kwan, I. (2013). In-game assessments increase novice programmers' engagement and level completion speed. *ACM ICER*, 153-160.
45. Lewis, J., & Loftus, W. (2005). Java software solutions (Java 5.0 version): Foundations of program design (4th ed.). *Boston, MA: Addison Wesley.*
46. Lightbot. http://www.lightbot.com. Accessed: 2015-03-26.

47. Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, et al. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4), 119-150.
48. Malone, T.W. (1981). What Makes Things Fun to Learn? A Study of Intrinsically Motivating Computer Games. *Palo Alto, CA: Xerox*.
49. Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. ACM TOCE, 10(4), 16.
50. McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.D., Laxer, C., Thomas, L., Utting, I. & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, 33(4), 125-180.
51. McNamara, D., Jackson, G., Graesser, A. (2009) Intelligent tutoring and games. *Artificial Intelligence in Education,* 1–10.
52. Papert, S., & Harel, I. (1991). Situating constructionism. *Constructionism*, 36, 1-11.
53. Pear, J.J. (2004). Enhanced feedback using computer-aided personalized system of instruction. In W. Buskist, V.W. Hevern, B.K. Saville, & T. Zinn, (Eds.), *Essays from excellence in teaching* (Chapter 11).
54. Prensky, M. (2003). Digital game-based learning. *Computers in Entertainment*, 1(1), 21-21.
55. Prensky, M. (2006). Don't Bother Me, Mom, I'm Learning!: How Computer and Video Games Are Preparing Your Kids for 21st Century Success and How You Can Help. *Saint Paul, Paragon House*.
56. Randel, J.M., Morris, B.A., Wetzel, C.D., & Whitehill, B.V. (1992). The effectiveness of games for educational purposes: A review of recent research. *Simulation & Gaming*, 23(3), 261-276.
57. Ross, J., Irani, L., Silberman, M., Zaldivar, A., & Tomlinson, B. (2010). Who are the crowdworkers?: shifting demographics in mechanical turk. *ACM CHI*, 2863-2872.
58. Rubio, M.A., Romero-Zaliz, R., Mañoso, C., & Angel, P. (2015). Closing the gender gap in an introductory programming course. *Computers & Education*, 82, 409-420.
59. Ruf, A., Mühling, A., & Hubwieser, P. (2014). Scratch vs. Karel: impact on learning outcomes and motivation. *ACM WiPCSE*, 50-59.
60. Ryokai, K., Lee, M.J., & Breitbart, J.M. (2009). Children's storytelling and programming with robotic characters. *ACM Creativity & Cognition*, 19-28.
61. Shute, V.J., Ventura, M., Bauer, M., & Zapata-Rivera, D. (2009). Melding the power of serious games and embedded assessment to monitor and foster learning. *Serious games: Mechanisms and Effects*, 295-321.
62. Shackelford, R. L. (1997). Introduction to computing and algorithms. *Boston, MA: Addison Wesley*.
63. Sime, M., Green, T., & Guest, D. (1976). Scope marking in computer conditionals: A psychological evaluation. *International Journal of Man-Machine Studies*, 9, 107–118.
64. Soflano, M., Connolly, T.M., & Hainey, T. (2015). An Application of Adaptive Games-Based Learning based on Learning Style to Teach SQL. *Computers & Education*.
65. Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9), 850-858.
66. Steffe, L.P., & Gale, J. E. (Eds.). (1995). Constructivism in education. *Hillsdale, NJ: Lawrence Erlbaum,* 159.
67. Summers, N. (n.d.) Codecademy surpasses 24 million unique users for its free online coding courses. *The Next Web*. Retrieved 23 April 2014.
68. Sykes, E.R. (2007). Determining the effectiveness of the 3D Alice programming environment at the computer science I level. *Journal of Educational Computing Research*, 36(2), 223-244.
69. Tew, A.E. (2010). Assessing fundamental introductory computing concept knowledge in a language independent manner. *PhD dissertation, Georgia Institute of Technology.*
70. Tew, A.E., & Guzdial, M. (2011). The FCS1: A language independent assessment of CS1 knowledge. *ACM SIGCSE*, 111-116.
71. Thompson, S.M. (2006). An Exploratory Study of Novice Programming Experiences and Errors. *Thesis., University of Victoria, Victoria*.
72. Umaschi, M. (1997). Soft toys with computer hearts: Building personal storytelling environments. *ACM CHI*, 20-21.
73. Webb, H. C., & Rosson, M. B. (2011). Exploring careers while learning Alice 3D: a summer camp for middle school girls. *ACM SIGCSE*, 377-382.
74. Werner, L L., Hanks, B., & McDowell, C. (2004). Pair-programming helps female computer science students. *Journal on Educational Resources in Computing*, 4(1).
75. Whitfort, T. (n.d.). Pseudo code guide [web page]. http://ironbark.bendigo.latrobe.edu.au/subjects/PE/2005s1/other_resources/pseudocode_ guide.html.
76. Williamson, B. (2009). Computer games, schools, and young people: A report for educators on using games for learning. *Bristol: Futurelab*.
77. Young, J. (2008). "Badges" earned online pose challenge to traditional college diplomas. *Chronicle of Higher Education*.
78. Zelle, J. M. (2004). Python programming: An introduction to computer science. *Wilsonville, OR: Franklin Beedle*.