Evaluating ProDirect Manipulation in Hour of Code

Quan Do
Department of Computer Science
Williams College
USA

Kiersten Campbell
Department of Computer Science
Williams College
USA

Emmie Hine
Department of Computer Science
Williams College
USA

Dzung Pham
Department of Computer Science
Williams College
USA

Alex Taylor
Department of Computer Science
Williams College
USA

Iris Howley
Department of Computer Science
Williams College
USA

Daniel W. Barowy
dbarowy@cs.williams.edu
Department of Computer Science
Williams College
USA

Abstract

We examine whether augmenting traditional coding environments with prodirect manipulation improves several learning measures. Prodirect manipulation is a novel user interaction model that provides a bidirectional link between code and outputs. Instead of reasoning abstractly about the output a program might produce, users instead directly manipulate outputs (e.g., using a keyboard and mouse). Program text is then updated to reflect the change.

We report the effects on learning using a prodirect manipulation environment versus a standard development environment for more than one hundred middle school students. To conduct the study, we built SWELL, a programming language with prodirect manipulation features. We conclude that within the context of an Hour-of-Code course, prodirect manipulation does not offer a significant advantage. We also make several observations regarding the way students interact with SWELL, which may inform future language design for this age group.

CCS Concepts • **Software and its engineering → General programming languages**; • **Human-centered computing → Graphical user interfaces**.

SPLASH-E '19, October 25, 2019, Athens, Greece

ACM ISBN 978-1-4503-6989-3/19/10…$15.00
https://doi.org/10.1145/3358711.3361623

Keywords computer science education, direct manipulation, prodirect manipulation

ACM Reference Format:
Quan Do, Kiersten Campbell, Emmie Hine, Dzung Pham, Alex Taylor, Iris Howley, and Daniel W. Barowy. 2019. Evaluating ProDirect Manipulation in Hour of Code. In *Proceedings of the 2019 ACM SIGPLAN SPLASH-E Symposium (SPLASH-E '19), October 25, 2019, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3358711.3361623

1 Introduction

Most software is written iteratively [14]. When a program does not work as intended, the problem is diagnosed and the code is modified. Such revision requires that a programmer understand the connection between a program's possible behaviors and its source code. Research suggests that novice programmers struggle to develop this connection [7, 9, 12, 16, 20, 27, 31].

There has long been interest in incorporating *direct manipulation*, which represents values as manipulable objects, into programming environments [8, 11, 18, 19]. Such an interface paradigm, called *prodirect manipulation*, combines both the interactivity of direct manipulation and the flexibility of a programming language. A prodirect manipulation-equipped programming environment lets users directly alter program outputs with a mouse or keyboard, which instantly updates the corresponding code.

This paper explores prodirect manipulation's potential within the context of computer science education. The extra fluidity afforded by such an environment facilitates experimentation and observation, potentially helping novice programmers to develop the connection between code and its behavior [33, 36]. In this paper, we ask whether prodirect manipulation lives up to its promise as a tool for teaching new programmers.

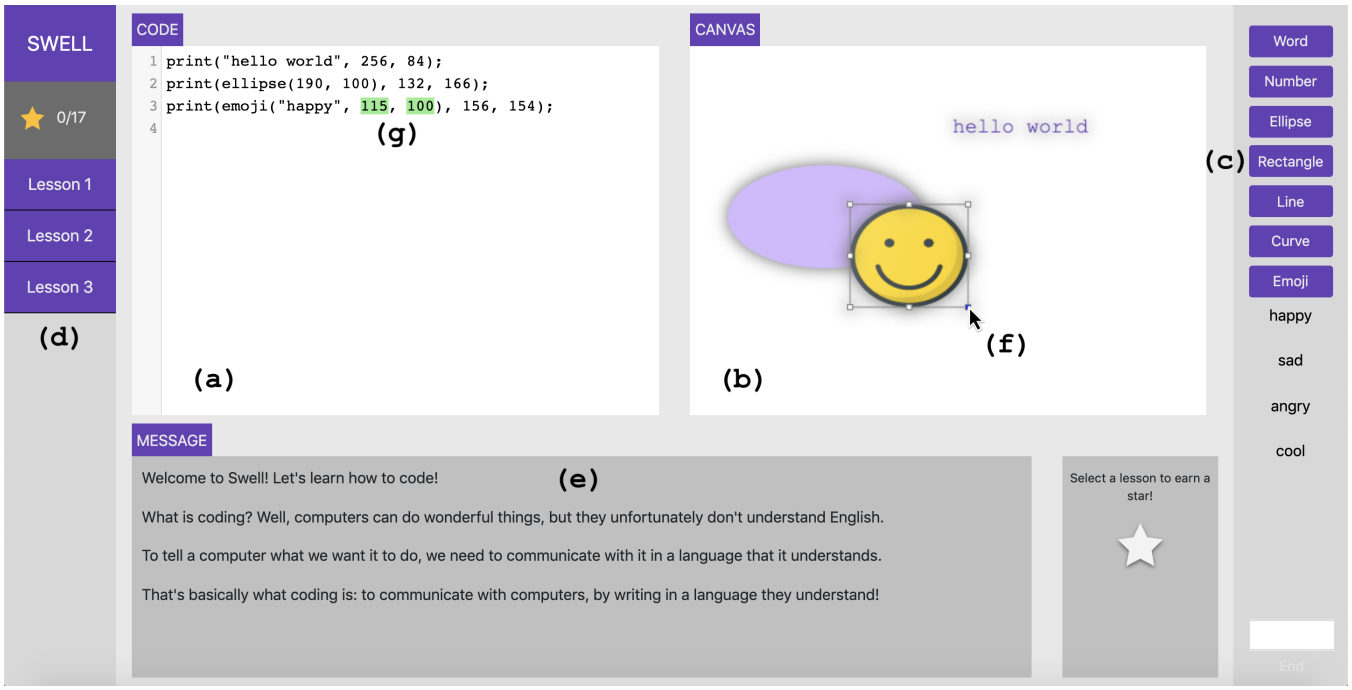

Figure 1. The web-based SWELL introductory programming environment. (a) Code editor where students write programs to render objects on a canvas. (b) The canvas where objects are rendered. With PDM enabled, students can drag and resize all rendered objects, as well as directly alter the values of string and number objects. Such manipulations update the corresponding program text in the code editor. (c) Students also have access to a palette of shapes that generate the appropriate `print()` statements in the code editor. (d) Each student can progress through the Hour-of-Code session at their own pace, by navigating through a sidebar to the left of the interface. (e) At the bottom of the interface is a message area that prompts students to progress through checkpoints, while also displaying error messages. Directly manipulating objects (f) updates the corresponding code (g) in the code window.

1.1 Contributions

Our work makes the following contributions:

- SWELL, a simple programming language designed for education. We augment SWELL with a web-based IDE featuring prodirect manipulation (§2).
- Two Hour of Code-style self-paced courses for middle-school students, one with and one without lessons utilizing prodirect manipulation. (§3.1).
- A user study (§3.2) conducted using SWELL (§4). We show that, despite its promise, direct manipulation does not improve short-term learning outcomes, student enthusiasm, or willingness to study CS.
- Finally, we offer some guidance for researchers considering in situ studies of first-time programmers (§4.3).

2 Design of SWELL

The SWELL introductory programming environment consists of two parts: a simple programming language and a graphical user interface augmented with point-and-click prodirect manipulation operations. We first motivate our use of the Hour of Code paradigm, and then we describe the system top-down, starting with the user interface.

2.1 Hour of Code

Hour of Code is a "one-hour introduction to computer science, designed to demystify 'code,' to show that anybody can learn the basics, and to broaden participation in the field of computer science," initially launched in 2013 [13]. Since then, interest in early CS education has grown tremendously, and Hour of Code has been promoted both by luminaries like US President Barack Obama and major corporations such as Apple and Microsoft [15].

Because of the widespread popularity of Hour of Code, and because it is a familiar and non-disruptive way of introducing computer science topics into a classroom, we found that teachers were highly receptive to the idea of letting us run an Hour of Code-style study. In fact, the combined effects of pressure from school districts to teach computer science and the dearth of qualified teaching personnel meant that teachers were eager for us to take over their classrooms for an hour at a time.

There are many Hour of Code-style tutorials. We modeled SWELL after one of the most popular ones, Khan Academy's Hour of Code programming tutorial[1]. This tutorial engages participants using a real programming language (Javascript and ProcessingJS) to draw shapes on screen, and is targeted at students ages 8 and up. Combining such tutorials with prodirect manipulation is a natural fit, as operations can be added unobtrusively. In fact, as of the time of this writing, the question "how do you draw on the right side of the code [*sic*]" is the highest-ranked comment in the "Drawing with Code" tutorial [17]. Implementing such operations requires prodirect manipulation, a feature not available in Khan Academy's tutorial.

2.2 User Interface

Figure 1 shows SWELL's web-based graphical user interface. SWELL allows programmers to interact with their code in two ways: either by editing code in the *Code* editor, or by manipulating program outputs that appear in the *Canvas* window. The *Code* editor (shown in Figure 1(a)) works like an ordinary text input box. SWELL comes equipped with built-in rendering functions so that programmers can print text and other shapes on the canvas (Figure 1(b)).

SWELL was designed to provide live feedback. Our interface has no *Run* button. After a suitable pause in typing, any code with valid syntax is executed immediately. Syntactically invalid code is highlighted in the editor using a squiggly red underline, reminiscent of Microsoft Word's spell-checker, and an error message is displayed in the *Message* window. The interface also comes with a clickable palette of functions that generate code for a set of predefined shapes, text, and numbers (Figure 1(c)).

Every object in the *Canvas* can be directly manipulated. Supported manipulation operations include *selecting*, *dragging*, *resizing*, and, if the objects are string or number literals, value *editing*. Manipulations automatically update the corresponding code fragment in the program text. For example, as the emoji object on the canvas is resized in Figure 1(f), the associated arguments for its dimensions are updated, and changed text is highlighted (in green) in Figure 1(g) to draw attention to the update.

Finally, the leftmost side of the screen hosts the interface for navigating programming lessons and a counter for the number of completed lessons. Each completed lesson earns the user a star. Users can freely move between lessons and checkpoints (sub-lessons) using a sidebar to the left of the code editor (Figure 1(d)). We developed a pair of curricula (where a curriculum is a sequence of lessons) for the SWELL environment. One curriculum uses prodirect manipulation and the other does not. Both curricula have the same conceptual structure and roughly the same number of checkpoints,

[1]https://www.khanacademy.org/hourofcode

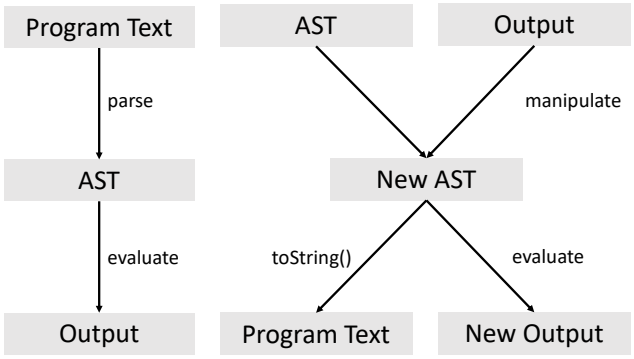

Figure 2. (a) In a traditional language, program text is parsed and the resulting AST is then either evaluated or compiled, eventually producing outputs. (b) In a PDM environment, this process can be run backward: manipulating an output changes the AST. When the changed AST is re-evaluated, new outputs and program text are produced.

and the learning goals are the same. We discuss our curricula in more detail in section 3.1.

2.3 Language

The SWELL programming language is a text-based, imperative, lexically-scoped, dynamically-typed language similar to Javascript. Unlike prior research in prodirect languages [11], which use Lisp-like syntax and functional semantics, we opted for a Javascript-like language. We adopt the imperative approach for two reasons. First, since our goal is to evaluate prodirect manipulation in an Hour of Code setting, our language and user interface are designed to mimic popular Hour of Code environments which are also imperative. This choice simplifies comparisons between SWELL and those environments. Second, we built a programming environment around a textual language instead of a block-based language like Scratch [6, 24] because conventional programming languages use textual syntax. Evidence shows that introductory programming education using block syntax neither helps nor hinders a student's eventual transition to a conventional language like Java, suggesting that the choice does not matter much [38].

The SWELL language was co-designed with live programming and direct manipulation in mind. We found that co-design was necessary to avoid pitfalls that occur when interacting with these features. For example, early versions of SWELL had both `while` and `for` constructs, which we later removed. We discovered that a typical convention for writing loops—adding the loop conditional after sketching out the control construct itself—sends a live programming environment into an infinite loop. A `while` loop with no conditional and no loop body is valid, so it will execute. One

alternative design suggested by colleagues makes a missing loop conditional evaluate to `false` by default. However, even here, a partial conditional expression can evaluate to `true`, which again would produce an infinite loop. Instead, we chose a simple, bounded `repeat(n)` construct, which is sufficient for educational purposes. Users can still write infinitely-looping programs if they so choose, using recursion. `repeat(n)` simply makes it harder to inadvertently produce infinite loops.

SWELL supports a number of primitive types, including decimal numbers, strings, arrays, and shapes (including emojis[2]). For consistency, all primitives can be rendered using an overloaded `print()` function. Outputs must be placed on the canvas at x and y locations, which can be supplied as arguments to the `print()` function. To simplify the experience for newcomers, coordinates may be omitted when printing, in which case objects are placed in a default location. *Dragging* an object from the default location inserts coordinates if they are missing. Figure 1 shows `print` being used with `string` literal, `ellipse`, and `emoji` constructors.

Similar to mainstream imperative languages, SWELL offers variable declaration and assignment as well as function abstraction using the `fun` keyword. For control flow, SWELL includes conditionals in the form of `if/else` statements, as well as the aforementioned `repeat(n)` looping construct.

2.4 Prodirect Manipulation

In the SWELL web interface, users can directly edit objects already drawn on the canvas by *selecting*, *dragging*, *resizing*, and *editing* them.

A PDM-equipped development environment establishes a bidirectional mapping from program text to output. The mapping from program text to output is simply the process of interpreting or compiling a program. SWELL is currently interpreted. Figure 2(a) shows the output of the parser and evaluator for a typical programming language.

The mapping from output back to program text involves user interface events like mouse clicks or key presses. As shown Figure 2(b), a manipulation operator (like *edit*) functions as an AST transducer: operators take in an AST and a change to the output and produce a new AST. The new AST is re-evaluated, producing a new output. At the same time, new program text is generated from the updated AST. Although SWELL's semantics are not whitespace-sensitive, SWELL is careful to preserve the user's idiosyncratic use of whitespace in the updated program by using a whitespace-sensitive parser. We do this to avoid disruptive code transformations.

In SWELL, all side effects that appear in the *Canvas* are represented explicitly by objects stored in the Javascript heap to facilitate direct manipulation. Each object implements an `Effect` interface, which informs the runtime how to draw the object on the *Canvas* as well as how to respond to direct

[2]Which were very popular among 5th and 6th grade students.

CODE

```
1 i = 1;
2 print(i, 100, 100);
3
```

(a) Before PDM.

CODE

```
1 i = 2;
2 print(i, 100, 100);
3
```

(b) After PDM.

Figure 3. A scenario where PDM localizes the change to a variable assignment in the program text. (a) Before PDM, a variable is assigned the value 1, then printed. (b) After the *edit* PDM operation, wherein the user directly edits the canvas output to 2, the runtime finds an update such that the program output is now 2.

manipulations. For example, as a user *drags* an ellipse across the screen, the `modifyDrag` method updates the object's x and y coordinates in the AST. When the screen is updated, the new parameters are used to redraw the object.

More concretely, consider the *edit* operation performed in Figure 3. Variable `i` is first assigned the value `1` and then rendered on the canvas using a `print()` statement (Figure 3(a)). The user selects the number by clicking on it in the *Canvas*, and then changes its value to `2` by pressing the `delete` and `2` keys. The `modifyText` method triggers an update to the value, which in turn updates the assignment statement in the *Code* window. SWELL highlights the update in green to draw attention to the changed portion of the program (Figure 3(a)).

Since the value to be updated may be the result of a compound expression, like `i=1+j`, there are often many possible changes that could produce the user's manipulation. This problem, a generalization of the *view update problem* from database literature, can be solved using constraint satisfaction techniques [4]. Consequently, the effect of an edit on a program is sometimes ambiguous, and necessitates heuristics that attempt to infer the intent of the programmer in order to avoid counterintuitive results [11, 25].

We sidestep these complexities by requiring that edits must not admit multiple satisfying program updates. This means that an updatable expression must be a literal value or a variable bound to a literal value. As a result, manipulations are either possible and unambiguous, or are not possible. We favor this approach over other program synthesis techniques whose semantics may appear opaque even to expert users.

Instead, we adopt a simple mechanism for determining which values must be updated in a program: dynamic dispatch. When a manipulation operation is triggered on an object, a message is sent from the affected `Effect` to the AST node that generated the `Effect`. If the AST node represents a variable, the message is forwarded to the expression that assigned the variable. If an update message is sent to an expression that admits multiple satisfying edits, the SWELL

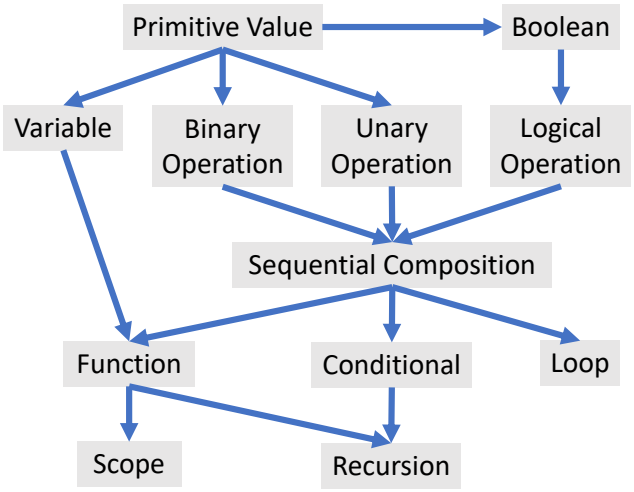

Figure 4. The concept map: a topic dependence graph that includes key programming concepts students should learn in an introductory course. An arrow pointing from *A* to *B* means that *B* conceptually builds on *A*. This graph informs our curriculum design.

user interface prevents the user from manipulating the object in the *Canvas*. Otherwise, the value is a literal, and it is modified. Since our goal was to develop a PDM environment adequate for the purposes of CS education, we found this simplification expressive enough to develop a rich curriculum around prodirect manipulation.

3 Study Design

To study the effect of prodirect manipulation in early CS education, we developed two curricula using SWELL: one curriculum that leads students through conventional software development lessons (i.e., a small programming challenges after a short lesson), and another curriculum that interleaves lessons with direct manipulation demonstrations (see §3.1). Experiments were conducted in 5th and 6th grade classrooms and were patterned after Hour of Code workshops (see §3.2); students were randomly assigned to a curriculum at the classroom level. Four schools from three school districts participated in the study, ranging from students living in an affluent college town to students in low-income rural and urban areas (see §3.3).

3.1 Hour-of-Code Curricula

Hour of Code workshops are customarily structured as hour-long self-paced activities, facilitated by software learning modules. We implemented our curricula as an Hour-of-Code module on top of the SWELL interface, drawing inspiration from Khan Academy's Hour of Code. Using this model introduces an important caveat: because our intervention is such a short duration, only large effects will produce statistically significant differences between treatment and control. In effect, we are asking whether direct manipulation makes a "big difference" for learning. We discuss this aspect of the study more in §4.

One SWELL Hour of Code session consists of multiple lessons, each consisting of multiple checkpoints. Each checkpoint has a specific instructional goal. By following on-screen prompts, students are instructed on a programming concept, culminating in a *checkpoint*, a test to assess student understanding. If a student fails a test, a prompt suggests that they revisit an earlier part of their lesson by clicking a button.

We developed two curricula: one allowing students to work through the Hour of Code using PDM, while the other disables all PDM operations (i.e., students must write code conventionally, using a keyboard). Each curriculum is a treatment in our experiment. Because we expected students to converse with their peers[3], we randomly assigned treatments at the classroom level. At least one classroom from each district was assigned to each of the two treatments.

The purpose of the two treatments is to answer the following research questions:

- *RQ1:* Does PDM improve comprehension of introductory programming concepts?
- *RQ2:* Does PDM increase excitement or willingness to engage in CS instruction in the future?

The order of instructional topics in our curricula are informed by a topic dependence graph shown in Figure 4 that we call the *concept map*. Each node in the concept map is a key programming concept that students should learn, and its in-edges denote prerequisite concepts. For example, to learn about functions, a student must first master sequential composition of instructions and the use of variables. Our lesson plan is therefore a simple topological ordering of the concepts we want to teach.

Figure 5 shows a sample checkpoint. Before every checkpoint, SWELL leads students through a short tutorial. At each step of the tutorial, an instruction box is superimposed over either the *Code* editor or the *Canvas* to draw student attention to a new concept. Instructions prompt students to make a change (either by editing code or by performing a manipulation) and to observe the effect of the change on the output, or in the case of PDM, the program text. Many tutorials give students the opportunity to play freely with the concept just learned before progressing. Each checkpoint ends with a test of the knowledge covered in that checkpoint. For the PDM treatment, direct manipulation is disabled for the test in order to assess comprehension.

[3]We were warned of this possibility ahead of the experiment by our school partners, and indeed, preventing 5th and 6th graders from conversing would have been an exercise in futility.

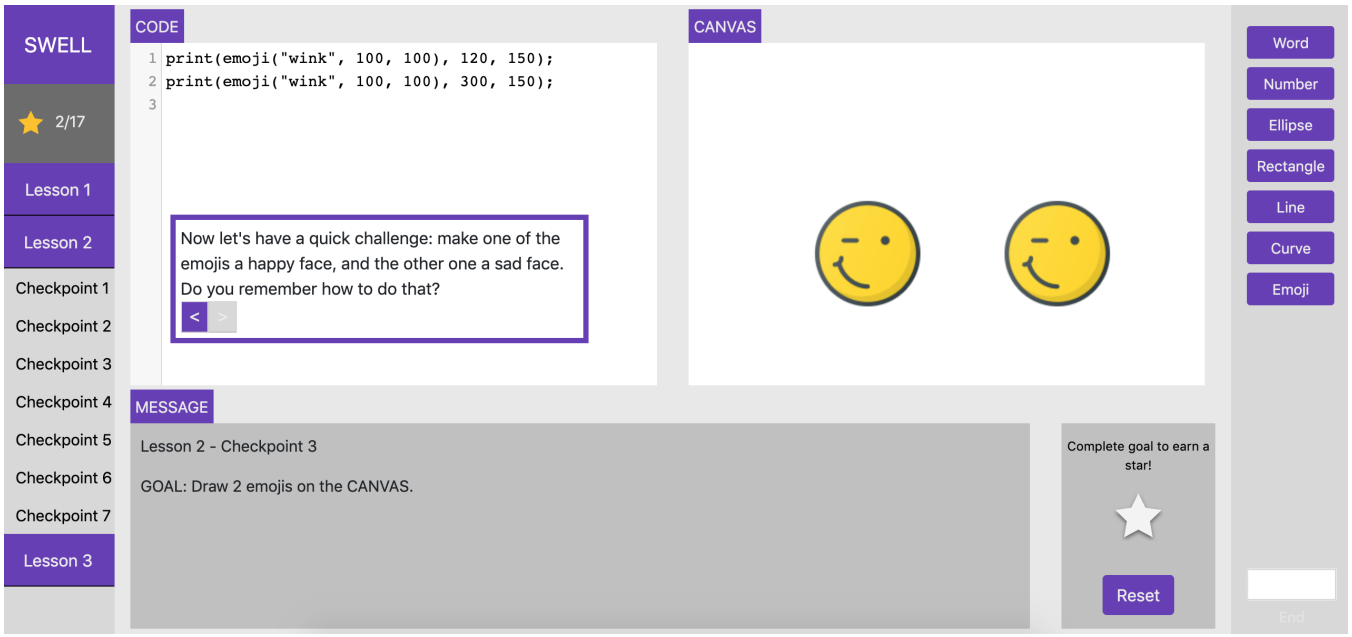

Figure 5. A checkpoint in which a student is being tested on function calls. The student can cycle through past tutorial instructions to review what they did. The checkpoint is only marked as completed once the student provides a correct answer.

3.2 Experimental Procedure

Each classroom session begins with a short introduction of the hour's activities by an Hour of Code facilitator. Our facilitators were college-age student volunteers who were not necessarily CS majors or part of our research staff. Students were instructed to follow the on-screen prompts. Facilitators walked around the room, answering questions when students requested help by raising their hands.

Students first completed a short pre-experiment survey designed to gauge excitement level and prior experience. For both surveys, the order of the questions and the order of response options are randomized. The study also concluded with a post-experiment survey to measure the change in excitement level and interest. The pre-test includes the following two questions:

- How excited are you about programming? (*not excited*, *excited*, and *very excited*)
- Have you programmed before? (*yes*/*no*)

The post-test includes the following two questions:

- How excited are you now about programming? (*not excited*, *excited*, and *very excited*)
- How interested are you in learning more about programming? (*not interested*, *interested*, and *very interested*)

After the pre-survey, students begin work on lessons. All student activities, including survey responses, are logged to a remote server. Such logging includes students' keyboard and mouse inputs as well as program text. This information allows us to reconstruct a trace of student activity for any given point. The information that we log includes:

- current lesson and checkpoint;
- all student code (i.e., partial solutions), and whether that code successfully parses;
- direct manipulation events;
- and timestamps for all events.

3.3 School Demographics

Our experiment drew from 5th and 6th grade students across three school districts from widely varying socioeconomic backgrounds. Our school partners requested that we not identify them in this study, however, they permitted us to describe their demographics. We use free and reduced lunch enrollment rate as a proxy for economic status [35]. Schools in more affluent communities tend to have lower free and reduced lunch enrollment rates. Schools A and B are urban schools (69% free and reduced lunch in 2014-2015); school C is a rural school (45.8% free and reduced lunch in 2014-2015); school D is an affluent suburban school in a college town (21% free and reduced lunch in 2014-2015) [1]. Each Hour-of-Code session took place during regular class time (during either the regular math period or during the library period) and spanned roughly one hour. In addition to college-age facilitators, teachers assigned to the regular class period were also present; because of their prior familiarity with the students, their primary role was to help maintain student focus during the study.

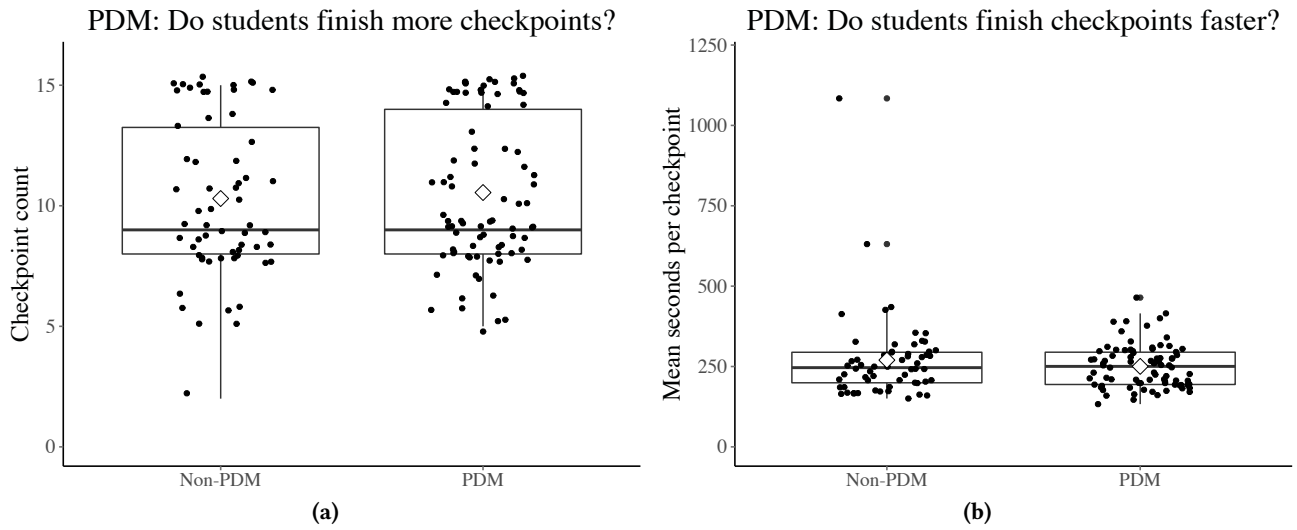

Figure 6. (a) The number of SWELL Hour of Code checkpoints completed by each student. (b) Mean seconds for each student to complete a checkpoint. In both cases, there is no significant difference between the PDM treatment and the non-PDM control. The mean is denoted with a diamond symbol.

4 Evaluation

After dropping incomplete observations (i.e., traces from students who did not complete the post-survey), our experimental data contains traces for 114 students: 30 sixth graders from school A, 29 fifth graders from school B, 26 fifth graders from school C, and 29 fifth graders from school D. 54/114 students were assigned to the PDM treatment. Overall, 80 students reported some previous experience with programming, and 34 reported no experience. On average, the students spent 40.7 minutes working on our modules, with a median of 40.3 minutes. They progressed through an average 10.3 out of 15 checkpoints, with a median of 9 checkpoints.

Instruction is structured in a sequential fashion such that upon finishing a checkpoint, the student is prompted to proceed to the next checkpoint in the sequence. There are two instances of students backtracking to an immediately previous checkpoint; some backtracking is expected since a number of checkpoints have optional prompts that encourage students to revisit previous checkpoints. No students skipped any checkpoints.

Below, we refer to the prodirect manipulation treatment as *PDM* while we refer to the non-prodirect manipulation control as *non-PDM*.

4.1 Does PDM improve comprehension of introductory programming concepts?

Since we cannot directly observe student comprehension, we instead operationalize learning in terms of the number of checkpoints students completed and the total time taken. *Checkpoint count* is the number of checkpoints completed by a student during an Hour of Code session. *Total time* is the time elapsed between the end of the pre-survey and the beginning of the post-survey. To make traces from different treatments directly comparable, we exclude student activities for the two checkpoints present in the PDM treatment but absent in the non-PDM treatment. These extra checkpoints include some early tasks to familiarize students with PDM itself.

Checkpoint counts. Do students in the PDM treatment complete more checkpoints than control students? Figure 6a strongly suggests that there is no difference in outcomes between the PDM and control treatments. Using the Shapiro-Wilk test, we conclude that checkpoint count distributions are not normally distributed ($p = 3.1 \times 10^{-6}$ for the PDM group and $p = 3.6 \times 10^{-4}$ for the non-PDM group). Therefore, to test the null hypothesis that there is no difference between median checkpoint counts, we use the unpaired two-sample Wilcoxon rank sum test. The test fails to reject the null hypothesis with $p = 0.68$, which means that there is no significant difference between treatments.

Checkpoint completion rates. Because the total time students spend in the SWELL learning module varies—some students finished all of the lessons and others gave up early—we also normalize checkpoint counts by total time. *Checkpoint rate* is the total time a student spent in the module divided by their checkpoint count. As with checkpoint counts, Figure 6b suggests that there is no difference between treatments, even when accounting for time. We test the null hypothesis that there is no difference between median checkpoint rates using the Wilcoxon rank sum test, as rates are also not normal (Shapiro-Wilk $p = 3.0 \times 10^{-11}$). The test fails to reject the null hypothesis with $p = 0.84$, revealing that there is no significant difference between treatments for checkpoint rates.

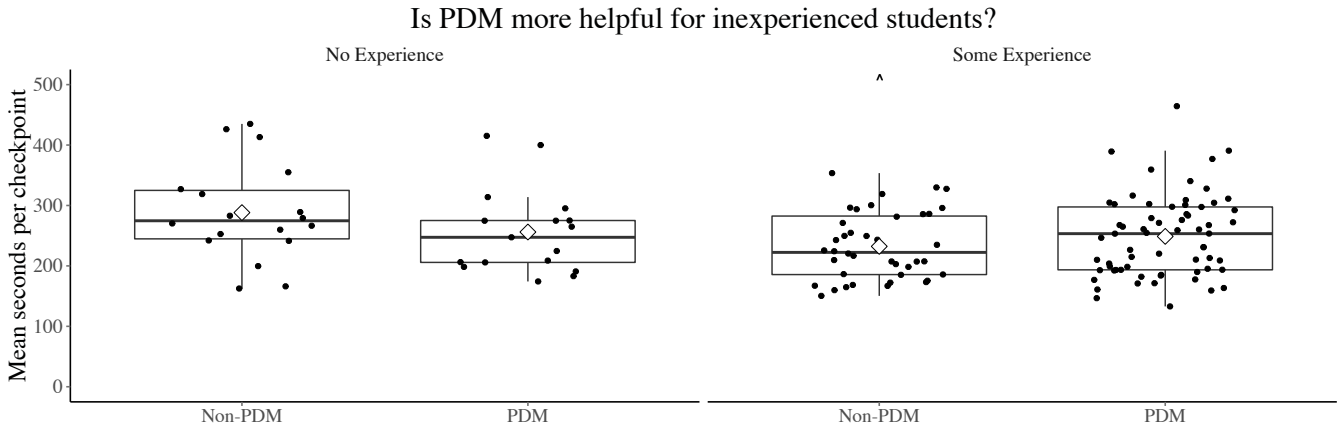

Figure 7. Mean seconds per checkpoint, when conditioning on prior programming experience. Before participating, students indicated whether they had "no experience" or "some experience." There is no significant difference between PDM and non-PDM groups for inexperienced students. ˆ denotes the presence of outliers that exceed 500 seconds.

Programming experience. In our pre-survey, students noted whether they had prior experience. We wanted to know if conditioning the data on prior programming experience revealed any differences. These differences may be hidden when viewing the population in aggregate. Specifically, we want to know whether inexperienced students benefit more from direct manipulation than experienced students. Figure 7 shows that even though there is a small positive effect for inexperienced students, there is no significant difference. Using the Wilcoxon rank-sum test, we test the null hypothesis that inexperienced students exposed to PDM do no better than their non-PDM peers. The test fails to reject the null hypothesis with $p = 0.20$.

Summary. We conclude that for Hour of Code-length interventions, prodirect manipulation does not have a large effect on learning. Anecdotally, a number of our research staff observed that there was a qualitative difference between students who received the PDM treatment and those who did not. A number of students in the PDM group remarked to us, with excitement, that SWELL was "cool," and among students who finished all of the checkpoints, students in the PDM group more enthusiastically engaged in "free play" using SWELL. We explore differences in attitudes more quantitatively in the next section.

4.2 Does PDM increase excitement or willingness to engage in CS instruction in the future?

A tool that does not improve learning outcomes, but instead generates excitement or willingness to study computer science, is also a compelling result. Therefore, we investigate whether direct manipulation increases excitement or willingness to engage in coding activities in the future.

Excitement. Our pre-test and post-test surveys ask students to rate their excitement levels before and after the session. The options are *not excited*, *excited*, and *very excited*. The majority of the students (73.6%) either maintain their excitement levels (31 stayed at *excited* and 29 stayed at *very excited*) or become more excited after the session (24 increased their excitement level from *excited* to *very excited*). It is clear that, for most students, Hour of Code activities are met with high levels of enthusiasm. However, we'd like to know whether PDM increases this enthusiasm level over the control group.

To facilitate comparison, we categorize changes in excitement levels as either *positive* or *negative*. We call these changes *excitement outcomes*. A negative outcome is when a student's excitement level either decreases or remains at *not excited*. All other outcomes are positive. To determine whether direct manipulation correlates with positive outcomes, we fit a logistic regression model to predict the excitement outcome (dependent variable) given a treatment (independent variable). We used a Wald test, which measures the probability that the outcome can be explained by factors (regression coefficients) other than the treatment (i.e., the null hypothesis). The test produces a z score of 0.83[4] with $p = 0.405$, which means that PDM is slightly, but not significantly, correlated with positive increases. However, since p is much greater than the typical significance threshold of $\alpha = 0.05$, we cannot rule out the possibility that the result is due to random chance.

It should be noted that post-excitement generally increased, and that this was true regardless of treatment. Again using a Wald test (on the intercept coefficient), we obtain a z score of 3.68 with $p = 0.0002$, which is strongly significant. So although the PDM treatment does not significantly improve excitement outcome, the outcome of an Hour of Code session is already good.

[4] z scores are a form of normalized standard deviation. $|z| > 2$ is generally considered rare.

Willingness. In our post-survey, we also asked students whether they were interested in learning more about programming (the options were *not interested*, *interested*, and *very interested*).

As with excitement, we wanted to know whether PDM predisposed students to want to study more programming than the control. A Wald test produces a z score of 0.68 with $p = 0.49$ (well over $\alpha = 0.05$). Again, this is a small positive increase that is not significant, and we cannot conclude that prodirect manipulation significantly alters the outcome.

Similar to excitement, outcomes are favorable irrespective of the treatment. 88.5% of students stated that they were either *interested* or *very interested* in learning more about programming.

Summary. Although we are unable to establish a positive link between prodirect manipulation and increases in affective measures, it is clear that in general, students respond favorably to Hour of Code activities using SWELL. This enthusiasm was noted by teachers in our study cohort, several of whom asked that we return to repeat the workshop. Since these repeat visits provide no additional insight, we do not describe them here, except to note anecdotally that student engagement remained high on our return.

4.3 Observations and Future Work

In this section, we make several qualitative observations regarding how students approach our lessons and interact with the SWELL interface. We hope that these observations will help inform future studies on programming language technology for educational purposes.

Our first observation is that very few students had any exposure to the literal nature of language syntax prior to our study. Even in the cases where checkpoint prompts provide sample code to copy, students make many mistakes when reproducing code. In particular, students frequently leave out parentheses for function calls, commas to separate arguments, and quotes for string literals. Although we could adopt a Scratch-like approach that makes syntax errors impossible, we are not yet ready to abandon traditional syntax. Students in our study eventually learned how to manipulate syntax even without explicit instruction. In the future, we plan to add tutorials that focus on diagnosing and correcting syntax errors to address this early roadblock.

Another observation is that students are generally insensitive to error messages. For example, in SWELL, parser failures underline problematic syntax with a red squiggle and print an error message to the *Messages* window. Instead of trying to understand the feedback, students typically sought help from the facilitators in the room. Indeed, this behavior is a possible confound in our study, since the effect of a facilitator's intervention may mask the effects of direct manipulation. Although we were aware of this possibility before running our study, we decided to provide in-person assistance as the alternative struck us as irresponsible. Our middle school collaborators hosted our study with the presumption that it would also provide educational value. Regardless, one improvement to our study would have been to log student-teacher interactions to help establish the strength of the effect of teacher assistance.

Finally, in hindsight, we note that for short interventions like ours, small effects are likely to be expected. Prior work demonstrates that effect sizes are small for Hour of Code [30]. In that study, a 3.4% increase in respondents reporting that they liked computer science after participating in Hour of Code was deemed a noteworthy improvement. Because the study benefitted from a very large sample size (8,040 students), such a small effect was significant at the $p = 0.01$ level.

Consequently, it is hard not to wonder about some of the small effects observed in this study. For example, there is a small improvement for the PDM group over the control group when looking at inexperienced students (see Figure 7). Using the basic bootstrap (with 100,000 replications), the expected median speedup for inexperienced students using PDM compared to the control is one checkpoint, with a 95% confidence interval of $(-1.5, 3.0)$ checkpoints. While this result is inconclusive, we think that it is suggestive, and we note that this small improvement corresponds to an increase of 6%—on par with the effects from the Hour of Code study mentioned earlier. Longer interventions (e.g., a semester-long class or after-school computer science program) or a larger study may still reveal significant differences.

5 Related Work

Here, we examine the current state of direct manipulation, both as a feature in programming language design and as a mechanism in educational software.

Direct Manipulation. Popularized by the first Apple Macintosh computers, direct manipulation interfaces represent computer values, tasks, and actions as visible objects that users can select and modify. Modifications produce instant feedback [2, 34]. Well-known uses are the *desktop metaphor* for operating system file operations and *touch interfaces* for phone and tablet UIs.

Direct manipulation in programming. Adding direct manipulation to a programming language presents numerous technical challenges. While mapping direct manipulation operations to code transformations is straightforward for simple programs, the semantics of these operations are less clear when applied to complex programs composed of multiple parts. Research that combines direct manipulation with programmatic systems have focused on techniques to handle ambiguities, how to frame the problem as a type of bidirectional programming, applications of PDM in developing

document and web contents, as well as optimizations for such PDM systems [8, 11, 18, 23, 26, 37]. The term *prodirect manipulation*, which we use to describe the technique utilized in SWELL, was coined to encompass a vision of a language runtime that synthesizes programs from output manipulations in real-time [10].

Direct manipulation in educational software. While direct manipulation frequently appears in educational software, to our knowledge, there is no prior work evaluating prodirect manipulation for educational purposes. Prior programming-related direct manipulation work focuses on assisting with or abstracting away language syntax. For example, the Scratch, BlockyTalky, and Reduct environments utilize blocks-based programming languages; primitives can be dragged, dropped, and combined using a point-and-click interface, eliminating the possibility of syntax errors [3, 22, 24]. The goal is to help students focus on learning programming language semantics. Other environments, such as Pencil Code and Code.org App Lab, can toggle between textual and block representations of code [5]. Such dual-mode editors are potentially helpful for students migrating from blocks-based to text-based languages. Another system, ALVIS Live! requires students to write code, but provides a direct manipulation via a palette of program fragments (like a button to insert code that creates an array), as well as a debugger-like stepper and memory visualizer [19]. ALVIS Live!'s stepper is similar to PLTutor and Omnicode, neither of which utilize direct manipulation for program construction [21, 28]. Finally, direct manipulation is used to construct algorithm visualizations and animations via mouse gestures [32].

Existing uses of direct manipulation in programming education focuses more on teaching the abstract models behind programs, an approach called *comprehension-first* pedagogy [28]. In contrast, for SWELL, our emphasis is on building programs, a strategy called *construction-first* pedagogy. We believe that prodirect manipulation addresses the criticism from comprehension-first proponents that the constructionist approach forces students to learn semantics by burdensome trial-and-error, literally by adjusting parameter values and observing what happens. SWELL tackles this burden not by removing language syntax, but instead by encouraging students to intuit semantics by directly manipulating outputs.

6 Conclusion

Prodirect manipulation is at the frontier of programming environments, combining programmatic systems with direct manipulation interfaces. Despite a growing body of research that formalizes the technology, there has been little work exploring its application in education. SWELL presents an effort to harness a prodirect manipulation interface to teach introductory programming. Although we do not find a large effect on educational outcomes, we believe that the technology still holds promise for educational purposes. We quote Seymour Papert, inventor of the LOGO programming language, who observed in his foreword to *Mindstorms* that the long-term effects of interventions are hard to quantify. "If any 'scientific' educational psychologist had tried to 'measure' the effects of [my] encounter, [they] would probably have failed. It had profound consequences but, I conjecture, only very many years later." [29] Whether prodirect manipulation is one of those long term interventions remains unclear. However, with computer science skills now more important than ever, we think that prodirect manipulation remains a promising technique for improving CS education.

SWELL is available at http://swell-lang.org.

Acknowledgments

The authors thank Molly Polk, Jennifer Swoap, and Paula Consolini at the Center for Learning in Action at Williams College for assistance organizing Hour of Code workshops. We also thank the many teachers and staff at the schools that participated in this study. Finally, we thank the students in WSP CS 11, which ran during January 2019 at Williams College, for helping to facilitate the workshops.

References

[1] Annie E. Casey Foundation. 2019. *Free and reduced price lunch enrollment rates by school district.* The Annie E. Casey Foundation, USA.

[2] Inc. Apple Computer. 1992. *Macintosh Human Interface Guidelines.* Addison-Wesley Publishing Company, USA.

[3] Ian Arawjo, Cheng-Yao Wang, Andrew C. Myers, Erik Andersen, and François Guimbretière. 2017. Teaching Programming with Gamified Semantics. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17).* ACM, New York, NY, USA, 4911–4923. https://doi.org/10.1145/3025453.3025711

[4] Francois Bancilhon and Nicolas Spyratos. 1981. Update Semantics of Relational Views. *ACM Transactions on Database Systems (TODS)* 6 (12 1981), 557–575. https://doi.org/10.1145/319628.319634

[5] David Bau, D. Anthony Bau, Mathew Dawson, and C. Sydney Pickens. 2015. Pencil Code: Block Code for a Text World. In *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15).* ACM, New York, NY, USA, 445–448. https://doi.org/10.1145/2771839.2771875

[6] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Commun. ACM* 60, 6 (May 2017), 72–80. https://doi.org/10.1145/3015455

[7] Piraye Bayman and Richard E. Mayer. 1982. *Novice Users' Misconceptions of BASIC Programming Statements.* Technical Report. https://eric.ed.gov/?id=ED238395

[8] Benjamin E. Birnbaum and Kenneth J. Goldman. 2005. Achieving Flexibility in Direct-Manipulation Programming Environments by Relaxing the Edit-Time Grammar. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL-HCC '05).* IEEE Computer Society, Washington, DC, USA, 259–266. https://doi.org/10.1109/VLHCC.2005.15

[9] A. Böttcher, V. Thurner, K. Schlierkamp, and D. Zehetmeier. 2016. Debugging students' debugging process. In *2016 IEEE Frontiers in Education Conference (FIE).* 1–7. https://doi.org/10.1109/FIE.2016.7757447

[10] Ravi Chugh. 2016. Prodirect Manipulation: Bidirectional Programming for the Masses. In *Proceedings of the 38th International Conference on*

Software Engineering Companion (ICSE '16). ACM, New York, NY, USA, 781–784. https://doi.org/10.1145/2889160.2889210

[11] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 341–354. https://doi.org/10.1145/2908080.2908103

[12] M Clancy. 2004. Misconceptions and attitudes that infere with learning to program. *Computer Science Education Research* (01 2004), 85–100.

[13] Code.org. 2019. Hour of Code: Frequently Asked Questions. https://hourofcode.com/us. Accessed: 2019-07-10.

[14] M. Cusumano, A. MacCormack, C. F. Kemerer, and B. Crandall. 2003. Software Development Worldwide: the State of the Practice. *IEEE Software* 20, 6 (Nov 2003), 28–34. https://doi.org/10.1109/MS.2003.1241363

[15] Rip Empson. 2014. Obama, Celebrities, Politicians And Tech Co's Come Together To Launch Coding Education Push. https://techcrunch.com/2013/12/08/obama-celebrities-politicians-and-tech-cos-come-together-to-launch-computer-science-education-push/. (08 01 2014). Accessed: 2019-07-10.

[16] Anne Fay and Richard Mayer. 1988. Learning LOGO: A cognitive analysis. *Teaching and Learning Computer Programming: Multiple Research Perspectives* (1988).

[17] greenlanturn. 2019. Making drawings with code. https://www.khanacademy.org/computing/computer-programming/programming/drawing-basics/pt/making-drawings-with-code. Accessed: 2019-07-10.

[18] Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. 2014. Programming by Manipulation for Layout. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 231–241. https://doi.org/10.1145/2642918.2647378

[19] Christopher D. Hundhausen, Sean F. Farley, and Jonathan L. Brown. 2009. Can Direct Manipulation Lower the Barriers to Computer Programming and Promote Transfer of Training?: An Experimental Study. *ACM Trans. Comput.-Hum. Interact.* 16, 3, Article 13 (Sept. 2009), 40 pages. https://doi.org/10.1145/1592440.1592442

[20] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying Student Misconceptions of Programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*. ACM, New York, NY, USA, 107–111. https://doi.org/10.1145/1734263.1734299

[21] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, New York, NY, USA, 737–745. https://doi.org/10.1145/3126594.3126632

[22] Annie Kelly, Lila Finch, Monica Bolles, and R Benjamin Shapiro. 2018. BlockyTalky: New programmable tools to enable students' learning networks. *International Journal of Child-Computer Interaction* 18 (04 2018). https://doi.org/10.1016/j.ijcci.2018.03.004

[23] Bum chul Kwon, Waqas Javed, Niklas Elmqvist, and Ji Soo Yi. 2011. Direct Manipulation Through Surrogate Objects. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 627–636. https://doi.org/10.1145/1978942.1979033

[24] John H. Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. 2008. Programming by Choice: Urban Youth Learning Programming with Scratch. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 367–371. https://doi.org/10.1145/1352135.1352260

[25] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 127 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276497

[26] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional evaluation with direct manipulation. *PACMPL* 2, OOPSLA (2018), 127:1–127:28. https://doi.org/10.1145/3276497

[27] T. Michaeli and R. Romeike. 2019. Current Status and Perspectives of Debugging in the K12 Classroom: A Qualitative Study. In *2019 IEEE Global Engineering Education Conference (EDUCON)*. 1030–1038. https://doi.org/10.1109/EDUCON.2019.8725282

[28] Greg L. Nelson, Benjamin Xie, and Andrew J. Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/3105726.3106178

[29] Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA.

[30] Rachel S. Philips and Benjamin PC Brooks. 2017. *The Hour of Code: Impact on Attitudes Towards and Self-Efficacy with Computer Science*. Technical Report. https://code.org/files/HourOfCodeImpactStudyJan2017.pdf

[31] Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. 2004. Self-efficacy and Mental Models in Learning to Program. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '04)*. ACM, New York, NY, USA, 171–175. https://doi.org/10.1145/1007996.1008042

[32] Jeremy Scott, Philip J. Guo, and Randall Davis. 2014. A direct manipulation language for explaining algorithms. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*. 45–48. https://doi.org/10.1109/VLHCC.2014.6883020

[33] Ben Shneiderman. 1987. Designing the User Interface: Strategies for Effective Human-Computer Interaction.

[34] Ben Shneiderman. 1997. Direct Manipulation for Comprehensible, Predictable and Controllable User Interfaces. In *Proceedings of the 2nd International Conference on Intelligent User Interfaces (IUI '97)*. ACM, New York, NY, USA, 33–39. https://doi.org/10.1145/238218.238281

[35] Tom Snyder and Lauren Musu-Gillette. 2015. Free or reduced price lunch: A proxy for poverty? https://nces.ed.gov/blogs/nces/post/free-or-reduced-price-lunch-a-proxy-for-poverty. Accessed: 2019-09-11.

[36] Bret Victor. 2012. *Learnable Programming*. Technical Report. http://worrydream.com/LearnableProgramming/

[37] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. 2012. Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 16, 11 pages. https://doi.org/10.1145/2393596.2393614

[38] David Weintrop and Uri Wilensky. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Transactions on Computing Education* 18 (10 2017), 1–25. https://doi.org/10.1145/3089799