To Block or not to Block, That is the Question: Students' Perceptions of Blocks-based Programming

David Weintrop
Northwestern University
2120 Campus Drive, Suite 332
Evanston, Illinois 60628
dweintrop@u.northwestern.edu

Uri Wilensky
Northwestern University
2120 Campus Drive, Suite 337
Evanston, Illinois 60628
uri@northwestern.edu

ABSTRACT

Blocks-based programming tools are becoming increasingly common in high-school introductory computer science classes. Such contexts are quite different than the younger audience and informal settings where these tools are more often used. This paper reports findings from a study looking at how high school students view blocks-based programming tools, what they identify as contributing to the perceived ease-of-use of such tools, and what they see as the most salient differences between blocks-based and text-based programming. Students report that numerous factors contribute to making blocks-based programming easy, including the natural language description of blocks, the drag-and-drop composition interaction, and the ease of browsing the language. Students also identify drawbacks to blocks-based programming compared to the conventional text-based approach, including a perceived lack of authenticity and being less powerful. These findings, along with the identified differences between blocks-based and text-based programming, contribute to our understanding of the suitability of using such tools in formal high school settings and can be used to inform the design of new, and revision of existing, introductory programming tools.

Categories and Subject Descriptors

D.1.7 [Visual Programming]. K.3.2 [Computer and Information Science Education]: Computer science education.

General Terms

Design, Human Factors, Languages

Keywords

Blocks-based Programming; High School Computer Science Education; Perceptions of Programming

1. INTRODUCTION

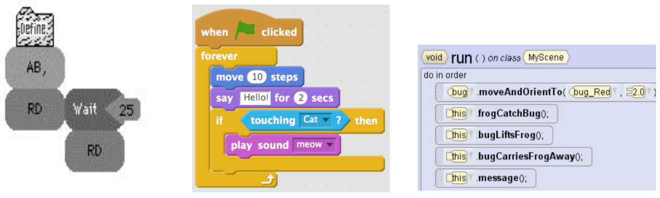
Computation is changing our world. Competencies and skills grounded in the ability to effectively use computational tools, and design and implement solutions that rely on computation, often collected under umbrella terms like "Computational Thinking," or "21st Century Skills," are now the focus of many new K-12 initiatives. This has resulted in new curricula and learning environments for introducing students to the field of computer science. Increasingly, these courses are turning to blocks-based visual programming tools to serve as students' first introductions to the practice of programming. Notably, the Exploring Computer Science Curriculum [13], the CS Principles course [1], and the materials produced by code.org for classrooms, all rely on the use of blocks-based programming. This trend is in part due to the general perception that blocks-based programming is easier for novice programmers. Despite the rise in prominence of blocks-based programming in formal settings, open questions remain as the strengths and drawbacks of this programming modality in classroom settings. Notably, little work has been done examining how learners perceive blocks-based programming interfaces and what they see as the utility of the approach relative to the more conventional text-based alternatives. Additionally, much of the work done on evaluating block-based programming has focused on younger learners and informal settings, contexts quite distinct from the high school classrooms where blocks-based programming is increasingly being used. This paper seeks to address these gaps in the literature by answering the following three research questions:

- Do high school students think blocks-based programming is easier than text-based programming and if so why?
- What do high school students perceive as the differences between blocks-based and text-based programming?
- What potential drawbacks to block-based programming do high school students identify?

We begin the paper with an introduction to blocks-based programming, trying to capture the current state and popularity of the programming approach. We then present our study design, a ten-week intervention in three sections of an introductory high school computer science course that followed students as they spend five weeks working in blocks-based tools then transitioned to a text-based programming language. Next, we present our findings, which include student identified strengths and weaknesses of blocks-based programming and reports on what they see as the major differences between blocks-based and text-base programming. Finally, we discuss the implications of these findings with respect to the design of introductory tools and the use of blocks-based programming in formal classroom settings.

2. BLOCKS-BASED PROGRAMMING

Blocks-based programming environments are a variety of visual programming languages that leverage a primitives-as-puzzle-pieces metaphor (Figure 1). In such environments, learners can assemble functioning programs using only a mouse by snapping together instructions and receiving visual (and sometime audio) feedback informing the user if a given construction is valid. Each block provides visual cues to the user on how and where the block can be used through the block's shape, its color (which is associated with categories of similar blocks), and the use of natural language label on the block to communicate its function. Along with the visual information depicted by each block, the construction space in which the blocks are used also provides various forms of scaffoldings including the grouping of similar blocks together, making it possible to easily browse the available set of blocks by category, and often providing a visual execution space in which the authored programs are enacted.



(A) LogoBlocks (B) Scratch (C) Alice

Figure 1. Three examples of blocks-based programming tools.

Blocks-based programming is a relatively recent addition to the long line of programming languages and environments designed explicitly with learners in mind (for reviews of this work, see: [8, 14, 22]). The earliest language designed explicitly for children, and a direct influence for blocks-based programming tools is the Logo programming language [10, 28]. The Logo language introduced a number of characteristics that feature prominently in blocks-based programming environments, notably, the use of egocentric motion commands like `forward` and `turn right`, the presence of onscreen avatars to carryout those commands (Logo had the turtle, while newer environments have sprites), and language primitives and syntax designed to be accessible to novices. Beyond features of the programming interface, the types of activities supported by blocks-based tools draw from the constructionist tradition that emphasizes learner-directed construction and exploration and the importance of learners creating public, sharable artifacts, often in the form of artwork, games, and interactive stories [15, 28].

In recent years, there has been a proliferation of programming environments that utilize a blocks-based approach. Well known block-based programming environments such as Scratch [29] and Alice [5] provide learners with open-ended, exploratory spaces designed to support creative activities like story telling and game making. With the rise in popularity of these and other similar tools, the number of activities a learner can engage with through blocks-based programming is growing increasingly diverse. For example, you can develop mobile applications with MIT App Inventor and Pocket Code [32], build and interact with computational models with DeltaTick [36], NetTango [19], Frog Pond [18] or StarLogo TNG [2], create artistic masterpieces with Turtle Art [3] or PicoBlocks, and play video games like RoboBuilder [34] and CodeSpells [9]. Similarly, informal computer science education initiatives are increasingly relying on blocks-based programming, including the activities provided as part of Code.org's Hour of Code and Google's Made with Code initiative. The rise of blocks-based tools is especially prominent in the design of programming tools for younger learners. A recent review of coding environments for children included 19 drag-and-drop tools among the 24 environments reviewed for learners under the age of eight, and 28 drag-and-drop environments out of the 47 total reviewed environments [8]. Further, we expect this trend to continue as a growing number of libraries are making it easy to develop environments that incorporate a blocks-based programming interface [12, 30]. With the growth of these environments, it is crucial to understand where they came from, if and why they work, and identify how learners perceive and interact with such tools and learn with such tools.

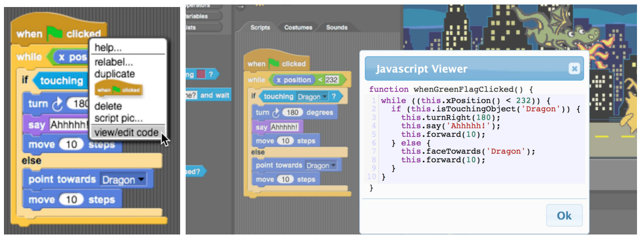
The blocks-based programming approach weaves together two historically distinct strands of research on ways to scaffold novice programmers. The first is the use of direct manipulation interfaces that present users with on-screen icons that depict the concepts or objects that the users will use to accomplish the desired goal. Programming in these environments takes the form of connecting the appropriate icons on screen. This approach has become popular for designing robotics kits such as Lego Mindstorms NXT-G [23] and the MiniBloq programming environment for the Arduino family of microcontrollers [31]. The second influence on blocks-based programming arose from the rise of structured editors [7], which are software authoring environments that use information about a programming language's underlying grammar to provide scaffolds to the users such as code-complete suggestions, syntax highlighting and real-time complication checking [27]. Similar to structured editors, blocks-based programming environments use the grammar of the language to support the act of programming by encoding the grammar of the language into the individual blocks through the name, shape and color ascribed to each block. The environments then allow the user to interact with these grammar elements (the blocks) directly though a drag-and-drop interface. In this way blocks-based tools provide the transparency and ease-of-use of direct manipulation interfaces with the scaffolds enabled by structured editors to create an introductory programming environment that is inviting and easy to use that also faithfully embodies the practice of programming and introduces learners to central ideas of programming.

Some have conceptualized blocks-based programming as serving as an introduction to programming that can lay the foundation for an eventual transition to text-based programming, but this remains an open empirical question that is only starting to be answered. A first step towards understanding if and how blocks-based programming prepares learners for future text-based programming is to identify what features of blocks-based programming learners find salient and how they perceive them relative to more conventional text-based programming.

3. METHODS AND PARTICIPANTS

The data presented in this paper are part of a larger study comparing blocks-based, text-based, and hybrid blocks/text programming environments at a selective enrollment public high school in a Midwestern city. We followed students in three sections of an elective introductory programming course for the first 10 weeks of the school year. Each class spent the first five weeks of the course working in a form of blocks-based programming environment. The students then transitioned to Java, a text-based programming language, for the next five weeks of the study and then continued with Java for the duration of the year. Two teachers participated in this study (one teacher taught two of the classes), both of whom have over five years of teaching high school computer science and have previously taught the course. Both teachers run a workshop-style class, doing little lecturing, instead having students spend class time working on assignments and asking for help when it is needed.

The three classes participating in our study used different modified versions of the Snap! programming environment during the first phase of the study [16]. The Snap! environment closely mirrors Scratch, but adds a few additional features (like first order functions) and was created with the goal of creating a "no ceiling" blocks-based programming environment [16]. The first class used a version of Snap! that added the ability to right-click on any block or script and open up a window showing a JavaScript implementation of what was clicked on (Figure 2). This served as a hybrid, blocks/text read only environment, as students were able to read, but not edit or write, text-based versions of the programs they constructed with the blocks. The second class used a version of Snap! that allowed students to read their programs, like the read-only condition, but also added the ability to define the behavior of new blocks in JavaScript. This served as a hybrid blocks/text read/write environment, as students could both read a text-based version of their own blocks, as well as define the behaviors of new blocks in JavaScript. The usual workflow for defining new blocks was for students to author the behavior with blocks, view the JavaScript equivalent, and then copy/paste the text into their new block. In this way, students in the read-write condition were usually not writing JavaScript from scratch, but instead doing more tinkering and tweaking of the textually defined behaviors. It is important to note in this condition, students were only writing small snippets of code (usually four lines or less) to define custom block behaviors and then integrating the text-defined custom blocks into larger scripts. Thus, this condition is quite different than a full text-based programming environment as block-based composition was still the predominant form of authoring, but is also distinct from fully blocks-based programming given the need to write some text-based code. Students in the third class used a version of Snap! that had no text-based features, so they never saw any text-based versions of their programs during class time. These three classes served as our three conditions for the study, which we abbreviate as: read-only, read-write, and graphical. All three classes followed the same curriculum based on UC Berkeley's Beauty and Joy of Computing course that covers topics including control structures, variables, and defining new functions. We chose this curriculum because it include the creation of new blocks early, so students in the read-write condition would get early exposure to authoring blocks in JavaScript.

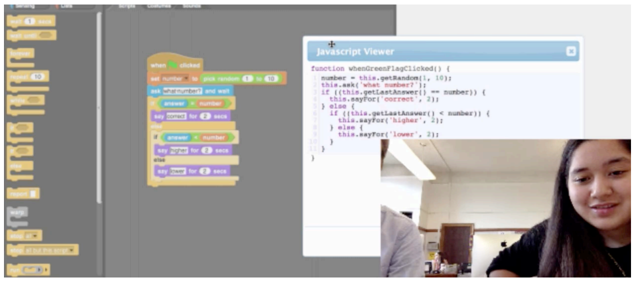


```
function whenGreenFlagClicked() {
while ((this.xPosition() < 232)) {
  if (this.isTouchingObject('Dragon')) {
    this.turnRight(180);
    this.say('Ahhhhh!');
    this.forward(10);
  } else {
    this.faceTowards('Dragon');
    this.forward(10);
  }
}
}
```

Figure 2. Side-by-side blocks and text in our version of Snap!

At the conclusion of the 5-week blocks-based introduction, the students transition to Java, following an objects-first curriculum designed around the Java Concepts: Early Objects textbook [20]. During the Java portion of the study, the topics covered in class included how to compile and run Java programs, simple data input and output, and the basics of defining and calling functions. It is worth noting this is a much more limited set of programming concepts than were covered in the 5-week Snap! curriculum.

A variety of data were collected as part of the study and used in the analysis presented below. Attitudinal surveys and content assessments were administered three times during the 10-weeks study: at the outset of the study (beginning of week 1), at the midpoint of the study after students had completed working with the blocks-based environments but before they had started with Java (end of week 5), and at the conclusion of the study after using Java for five week (end of week 10). All three surveys were administered online during class time. Additionally, a total of 27 semi-structured clinical interviews were conducted with students: nine during the first week of the study, ten at the midpoint (during weeks 5 and 6), and eight in the final week. The interviews took place in empty classrooms outside of class time, usually either during the student's lunch period or after school. For the interviews, the researcher sat alongside the student as they both faced a computer that had the Snap! programming environment on screen (Figure 3). Each interview was recorded using software that captures both the user sitting at the computer and what is being shown on screen. We will discuss details of the interview protocols later as part of our analysis.



```
function whenGreenFlagClicked() {
number = this.getRandom(1, 10);
this.ask('what number?');
if ((this.getLastAnswer() == number)) {
  this.sayFor('correct', 2);
} else {
  if ((this.getLastAnswer() < number)) {
    this.sayFor('higher', 2);
  } else {
    this.sayFor('lower', 2);
  }
}
}
```

Figure 3. A screen shot from an interview.

The school we worked with was chosen as it offers three sections of their Programming I course allowing us to carry out our three-condition study design with students from the same student population. A total of 90 students across three sections of the course participated in the study, which included 67 male students and 23 female students. The students participating in the study were 43% Hispanic, 29% White, 10% Asian, 6% African American, and 10% Multi-racial - a breakdown comparable to the larger student body. The classes included one student in eighth grade, three high school freshman, 43 sophomores, 18 juniors, and 25 high school seniors. Two-thirds of the students in these classes speak a language other than English in their homes.

4. FINDINGS

4.1 Is Blocks-based Programming Easier?

Our first research question asks if students think blocks-based programming is easier than text-based programming, and if so why. To answer this question, we will use data from the survey administered at the conclusion of our study, after students had spent five weeks working in Snap! and then another five weeks learning Java. On the survey we asked students to compare the two environments, specifically asking what they saw as the major difference between the two tools. We then analyzed each response, identifying which answers attended to ease-of-use as contributing to the difference between Snap! and Java. Of the 84 responses collected, more than half of students (58%) included ease-of-use as a major difference between the graphical and text-based environments. Table 1 shows the outcome of the coding of the responses for ease-of-use. The subscript numbers in the table show the breakdown by the three Snap! conditions.

In this analysis, we were careful to only include responses that clearly attended to a difference in difficulty between the two environments. For example, the response "[In Java] *there are no blocks to help out, it is basically done from scratch*" was coded as attending to ease-of-use, since the blocks "*help out*", while the response: "*Java is more writing as if it was a language, while Snap! you use logic to put blocks together*" was not coded as attending to ease-of-use because the student did not make it clear that this difference made one environment easier than the other. We included responses that mentioned the need to memorize commands in Java to mean that Snap! was easier due to the assumption that memorization contributes to difficulty. While many responses required some interpretation, others were very clear on which environment they found easier, giving responses like: "*Learning Java is more complicated than Snap!*" and "*Java is much easier for me than Snap!*" Additionally, two students attended to ease-of-use, but specifically said the two modalities were comparable: "*one is hard and the other is equally as hard.*"

Table 1. Student responses comparing Java to Snap! - coded for ease-of-use of the environment.

Perception	Count (Graph/Read-only/Read-write)
Text-based Programming is Easier	4 (0/1/3)
Blocks-based Programming is Easier	42 (14/15/13)
Comparable Difficulty	2 (0/1/1)
Did not attend to Difficulty	41 (13/13/14)

These data show that students found the blocks-based programming approach of Snap! to be easier than Java, thus supporting the general view of blocks-based tools being easier for novice programmers. There were no significant differences in responses across the three conditions of the study with the exception that three of the four responses that said Java was easier came from the read-write condition, where students were asked to write JavaScript along with compose in blocks. One explanation of this is that these students preferred the text-based programming in Java because composing text in Snap! required additional steps (creating new blocks and opening the editor), so students who were already comfortable with text-based programming may have found Java easier as it didn't require these additional steps.

4.2 Why is blocks-based programming easy?

Since our analysis shows that students perceive blocks-based programming as easier than text-based programming, we now move to the second part of our first research question, why? To answer this question we draw on the interviews we conducted during the first week of our study when students were initially introduced to the Snap! programming environment. We focus on these interviews as they give us insight into students' first impressions of the blocks-based and text-based representations. Data from later in the study, after students have experience using the two representations, will be incorporated later in our analysis. The protocol for these interviews starts with a series of questions asking about students' prior programming experience and their reasons for taking the course. From there, we introduced them to the Snap! programming environment and, if they had never seen blocks-based programming before, showed them how to write a basic program (i.e. how to drag-and-drop blocks together to control the onscreen sprite). The main portion of the interview had students try and write a simple program (programs differed depending on their prior experience, but ranged from having a sprite draw a square to a basic number guessing game for more experienced students). Having written the short program, we then opened up the text window to display a JavaScript implementation of the program they just authored (Figure 3.) and began a discussion about the differences between the two program representations.

Nine students were interviewed using this protocol, five male and four female. The students were distributed across the grades, with four grade 10 students, two grade 12 students, and one student each from grades, 8, 9, and 11. The students were chosen as they were representative of the larger sample with respect to grade, gender, and ethnicity, and had reported having little prior programming experience. From the nine interviews, four major reasons for blocks-based programming being easier emerged.

4.2.1 Blocks are Easier to Read

The first aspect of the blocks-based tools that students identified as helpful was the descriptive, easy-to-read labels on the blocks. "*Well, I mean, if you can read it…for humans this looks better, it's easier to understand.*" Despite its looking less like a text editor when compared with the text-based code, a number of students viewed the blocks-based representation as closer to English than its text-based counterpart. "*With blocks, it's in English, it's like pretty, like, more easier to understand and read,*" a second student highlighted this difference, saying: "*Java is not in English it's in Java language, and the blocks are in English, it's easier to understand.*" A third student explained: "[the blocks] *are basically a translation of what* [the JavaScript] *is doing, in, I guess, English for lack of better words. It is describing what* [the JavaScript] *is doing, but it's describing it in an English form...like a conversion.*" In calling the blocks a translation of the JavaScript, the student recognizes the equivalence of the two representations and identifies the blocks as being more easily read. Across these responses we see the blocks-based representation serving as an intermediary between English and conventional text-based programming, with students recognizing features of both in the blocks-based representation. Another way this difference appeared in our interviews was in students highlighting the lack of obscure punctuation in the block-based tool: "[the JavaScript] *is really confusing to understand with all the parenthesis and bracket and all of that.*" Of the nine student interviewed, seven mentioned the readability of the blocks as a feature that made them easier to use than the text-based alternative. That students find the natural language nature of Snap! helpful is supported by research on the design of novice programming languages that has found the strategy of drawing on learners' natural language knowledge to be effective [4].

4.2.2 Shape and Visual Layout of the Blocks

The second feature students identified that makes blocks-based programming easy is the visual nature of the blocks and the graphical cues that each block provides for how and where they can be used. Four of the nine students interviewed explicitly mentioned the shape of the blocks as being useful. For example, when our eighth grade student was asked why some blocks have rounded edges and others have diamond shaped edges, she explained that it was so "*the user knows that…they have a limited* *choice so that you don't make the mistake, because if all of* [the blocks] *were the same, it might not work. If* [the block is] *rounded or diagonal, they'll know the difference; they'll know that you can't put* [a diamond block] *in* [an oval slot]*, it's like a puzzle.*" A second student echoed this fact, when asked how he knew that Boolean blocks could be used with control structure blocks and numbers and mathematical operators worked with motion blocks he explained "*it's because of their outline,* [the Boolean blocks shape] *is the same as* [the control blocks inputs] *and then in motion, the* [oval input] *is the same as* [the mathematical blocks]." The shape was identified as being useful to see how blocks fit inside each other, as well as how sequences of blocks could be built, which was helpful for making sense of the resulting behavior. "*When* [the blocks] *are attached to each other, you know that the first one is going to affect the ones underneath it…everything is connected and it's easier to understand what is going on…I guess it's more intuitive too, because you can see how they all connect.*" Students said that these shape cues helped not only to see where blocks could be used, but also the larger idea of the importance of the sequence of commands, "[the environment] *teaches you that order is important*."

There is a potential drawback to the programming-primitives-as-puzzle-pieces metaphor stemming from the fact that in a puzzle, each piece has one specific place that it belongs. As one student said when talking about the blocks "*everything has its place*." This is not true with programming, as commands can be used in a variety of ways and in various places within a program to produce an infinite number of behaviors. One of our interviewees struggled due to holding this perspective, which became clear when he recounted his difficulty on the first class assignment. When asked if it was the blocks he struggled with, he answered that it was not the blocks themselves, but in not knowing "*the combinations that do something specific, like, I'm not exactly sure which blocks snap together to do something, like a specific action.*" The idea that particular sequences of blocks, when snapped together, produce a specific action calls to mind special combinations used in video games, where unique combinations of moves results in special actions that are different that the sum of the inputs used to produce them. As a result, this participant felt there was something he did not know, some knowledge beyond what is shown in the interface. After getting help from a neighbor, he said the program he eventually wrote made sense, but admitted to not knowing how to create it initially. We raise this issue with blocks-based interfaces as we fear it may be exacerbated by introductory activities that provide fill-in-the-missing-command style challenges, as they may reinforce the "*everything has its place*" perception of programming as opposed to the more accurate view of there being multiple ways to successfully achieve a desired programming outcome.

4.2.3 Easier to compose

A third advantage identified by students was how the act of composing a program was easier with blocks. This is in part due to the shape of the blocks that we just discussed, but also a product of a number of other features of the blocks-based modality. The first is that the act of dragging-and-dropping commands is easier and less error prone than having to type in commands character-by-character: "*If you type it, with like one word or one period or one something that's wrong it's going to mess everything up…it's just harder to write with the codes.*" Another student put it slightly differently saying: "*I like visualizing things more so with Snap, it's a lot easier than having to type everything in,*" continuing by saying how with text-based programing "*you have to be pretty precise with your punctuation, you have to type everything in.*" A third student succinctly put it, with blocks "*you don't end up making as much mistakes.*"

Along with the ease of composing valid programs, a number of students highlight how blocks make it easier to tinker with a program. "*You get to play around with* [blocks]*…because if you do it with writing, you like, have to erase everything or like start all over. It's not as easy to change and make new things. With blocks, you can just drag them and change what it's going to do*." This benefit can be seen when watching students compose programs, often taking a block or sets of blocks and putting them off to the side while trying new blocks in their script, only to ultimately return the removed blocks back into the script. Similarly, with blocks it becomes easy to compose complex statement in a non-linear order. For example, during her interview we asked one of our tenth grade students to write a program that required using a conditional statement to compare two numbers. The student built her statement in four discrete steps (Figure 4). First, she dragged out the green = comparison block. Second, she added the `answer` block to its left side and the `number` variable block to its right. Next, she dragged out the `if` block, placing her newly constructed comparator inside it, then finally completing the statement by nesting the `say` block inside the parent `if` block. This sequence of composition is quite different than how one conventionally goes about writing a conditional statement in a text-based language, where the left-to-right orientation imposed by the text editor makes it unnatural to start with anything other than the word `if`, making the approach this student took to building a conditional statements difficult. In this way, the blocks-based representation facilitates what Turkle and Papert [33] call epistemological pluralism, where the medium can support a variety of authoring approaches, including the traditionally favored planner mentality, as well as a bricolage orientation that emphasizes negotiation and rearranging of materials.

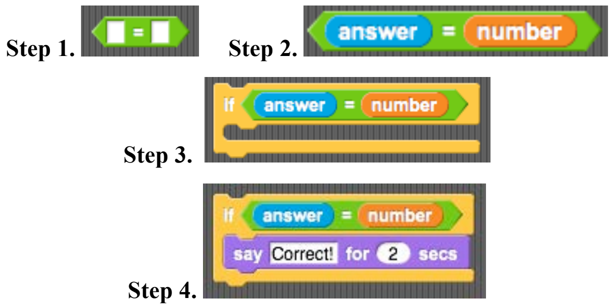


Figure 4. The sequence of steps followed by one student to assemble a conditional statement.

4.2.4 Blocks as Memory Aids

The final feature of blocks-based programming that was identified by four of the nine students interviewed, stems from the ease of finding block and understanding what they do through their organization within the programming environment. More specifically, how the blocks themselves alleviate the memorization that is required in text-based programming. "[The blocks] *kind of jog your memory, so you can see something and be like 'oh, I remember how to do that now', but with* [text-based programming] *you don't really have anything there to help you remember how to code something.*" As a second student put it: "[In JavaScript] *you need to like, know all the code words to draw something. Let's say you want to draw something, you need to type in a certain word to do that when in scratch you could just* *like, find the pen down block or something*." This last point is critical, blocks-based environments provide an easy and organized way to browse all the available blocks, making it possible to use the blocks themselves as a source of ideas, as one student put it: "*everything is here that you can do*." Another student focused on how easy it was to browse the available set of blocks as being a key reason blocks-based programming was easier, saying "*it's just because of the blocks and how they're separated into categories…so it's just much simpler to find the blocks and put them in to the pane*." The utility of the organization and ease of browsing of the blocks was evident throughout the interviews. For example, during an interview with a grade ten student, we asked if he could draw a square on the screen, he successfully did so, but relied on the `forever` block in his program. When asked how he would change his program so it would be possible to draw a second square next to the first, he opened, the Control category where looping blocks were stored, read through the blocks, and said "*I'm not really sure, I think it's in the tab somewhere though,*" showing how the organization of the blocks within the environment can support novices in constructing programs.

Recognizing the way that the graphical features of the blocks-based language support various cognitive aspects of the programming activity is important as a designer as it provides a powerful scaffold for learners. Viewing this characteristic of blocks-based tools through a distributed cognition lens [17, 21] provides a larger explanation that encompasses many of the features of blocks-based environments that students cited as supporting their learning. The distribution cognition theory argues that a cognitive system is not limited to just the internal processes of the individual, but includes the larger environment in which the activity occurs. In this expanded view, physical objects can serve as memories devices and aid the individual in accomplishing the task at hand. In blocks-based environments, this means the blocks themselves "remember" much of what would otherwise need to be known a priori by the user, including what the blocks do (captured by the text labels and what color and category the reside in) and how and where they can be used (denoted by the shape of the blocks). Similarly, the browsable categories offload the need for the user to have to remember everything that is possible in the language, and instead can serve as a guide for what is possible and act as a source of inspiration for the user [35]. Through this lens, the affordances of the blocks-based environment that contribute to their ease-of-use can be understood as the aspects of the knowledge one needs to be a successful programmer that are designed into the environment and the representation itself.

That blocks can serve as memory aids, along with the other three characteristics of blocks-based programming tools discussed above, make up the four most salient features of blocks-based programming for the novice programmers we interviewed. It is important to reiterate that these features were identified at the outset of the learning process, not by the designers, researchers, or educators who bring specific goals to the use of such tools, or by learners who had not already mastered the use of either text-based or blocks-based programming. This analysis provides evidence that these tools are effective at scaffolding learners during the early stages of learning to programming and identifies specific features of blocks-based tools that the learners found useful.

4.3 What are the differences between blocks-based and text-based programming?

Having identified four reasons for the perception that blocks-based programming is easier than the text-based alternative, we now proceed to our second question, which asks what students see as the main differences between blocks-based and text-based programming. To answer this question we analyzed student responses to a pair of survey questions asking them to compare blocks-based programming in our custom versions of Snap! to text-based Java programming. The questions were asked at two points during the study. First, on the mid-study survey we asked: "*The thing that will be the most different about programming in Java compared to programming in Snap! is:*" Students answered this question after using Snap! for five weeks but before they had started working in Java. Five weeks later, after students had been working in Java, we asked the same question, just shifting from the future tense to the present tense. A total of 85 students took the mid-study survey with one fewer student taking the final survey, resulting in a total of 169 responses. We open-coded these two sets of responses and categorized them by what students chose to identify as the largest difference between the two environments. Figure 5 shows student responses to these questions grouped by difference identified, point-in-time, and the version of Snap! the students used.

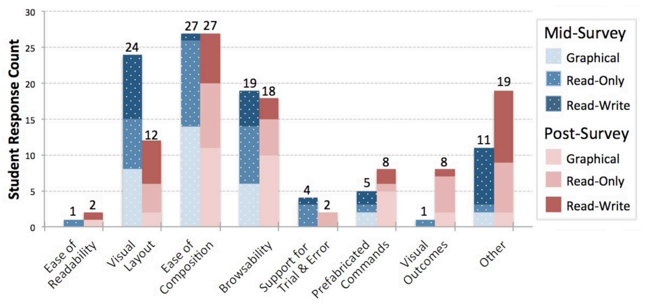


Figure 5. Student reported differences between Snap! and Java given at the mid-point and conclusion of the study.

This analysis revealed three new categories on top of the four themes that emerged during our interviews about what makes blocks-based programming easier. The new categories include the presence of prefabricated commands, the ease of trial-and-error programming in Snap!, and the different types of programs authored in Snap! versus Java. Table 2 provides examples of student responses for each category identified.

Despite the importance of the natural language labels on the blocks and the ease of readability that students emphasized at the outset of the study, students rarely cited this feature of blocks-based programming as being a major difference on the survey. The other three categories from the previous section were much more prominent. For the Visual Layout category, we included student responses that attended to shape or color of a block along with more general responses speaking to the graphical nature of the blocks. The Ease of Composition category included responses that directly referenced the drag-and-drop nature of the blocks or how the blocks can snap together. Our final category from the previous section, in which students highlighted the browsability of the blocks-based environment and how it alleviated the need to memorize syntax, was included in 37 student responses. Most of these responses highlighted how in Java, you have to know a command as well as it's syntax in order to use it in a program.

Beyond the four features discussed in the previous section, three other differences were repeatedly mentioned in students' survey responses. The first was how Java was not as conducive to the use of trial-and-error programming. This is particularly interesting as the trial-and-error approach is as valuable in text-based

Table 2. Sample responses to the question having students compare Snap! and Java

Category	Example Responses
Ease of Readability	*"The programming language will no longer be translated to English completely for a user to easily understand what is going on."* *"Snap! was easy to read."*
Visual Layout	*"There aren't going to be anymore colorful blocks."* *"I will have to code without having help from blocks."*
Ease to Composition	*"Actually having to type everything out instead of dragging and dropping."* *"Java is all hand typed while in Snap! you grab and drop blocks."*
Browsability	*"You will not have the blocks to aid you anymore and you will have to memorize and learn the Java script for everything you are trying to do."* *"Not feeling as restricted and having to think more because you don't have all the options in front of you."*
Support for Trial & Error	*"Java is not a trial-and-error program. If I make a mistake, then I must fix it on my own. There is no guessing involved, and I think I will have a really difficult time adapting to this process."* *"In Java, I will not be able to test out blocks and incorporate them and see if they work."*
Prefabricated Commands	*"There will be no set blocks that will provide you with pre made functions."* *"You do everything on your own without the help of preset blocks for the code, and you have to compile the file."*
Visual Outcomes	*"Java is more about having things such as text be displayed while Snap! was more about making sprites do things such as move or complete a goal etc."*

programming as in blocks-based, and nothing about text-based programming prevents the programmer from using the strategy. There are also potential consequences to thinking trial-and-error is not possible or not acceptable in text-based programming. Papert [28] addresses this in his discussion of the difference between learners perceiving errors as wrong versus errors as fixable and how the errors-as-fixable orientation is a much more productive learning strategy. If the shift from blocks-based to text-based programming also carries with it a shift from the trial-and-error strategy being supported to it being viewed as impractical or even not possible, it is important that we as designers and educators be aware of this misconception and try and address it.

The second new category to emerge was that of the lack of pre-fabricated commands in text-based programming. Whereas a single block can do something in Snap!, like move a sprite or ask a question, students thought that with text-based programming, the individual commands were more fine-grained, requiring more commands to be used to accomplish a comparable behavior. While this is not necessarily true when calling APIs or other pre-defined functions, this reported difference highlights the perceived contrast in the size of atomic block commands and text-based language primitives. The final new category captures students identifying the visual enactment of programs as being a major difference between Java and Snap! This difference speaks less to the blocks versus textual nature of the languages themselves, and more to the larger environments in which the programming is occurring. Interestingly, this was only identified by one student as a difference before the Java portion of the course, but was highlighted by eight students at the end of the study. It is worth mentioning this last difference need not always be the case as numerous text-based development environments (Java and otherwise) make it possible to have visual outcomes from the outset, with Logo being one prominent example.

Between the two sets of survey responses, there were an additional 35 differences identified that did not occur often enough to warrant their own category. These responses including Java requiring more planning upfront, Snap! running scripts in parallel, the compilation step required in Java programming, and the ease of debugging in the blocks-based environment. Additionally, 19 responses across the two sets of survey responses did not articulate a specific difference, instead giving broad, vague responses like "[Java] *won't be as fun*" or "*Java is much easier for me than Snap!*"

Looking at the differences between responses given at the midpoint of the study when students had only used the blocks-based tools and the end of the study after students had been exposed to Java, a number of shifts are visible. First, there was a significant drop in the number of students who identified the visual nature of the blocks (referring to their shape, color, and nested structure) as being the most significant difference between the modalities. This suggests that after working in Java, the visual representation loses significance relative to other differences that exist. A second difference was the growth in students attending to what is possible with Snap! and how the language supports accomplishing that objective. This can be seen in the rise of students identifying the visual outcomes of programs as being the most salient different as well as the loss of prefabricated blocks and the need to use more commands to achieve a specific outcome. Taking a step back, these shifts suggest that as experience with text-based programming language grows, what students attend to shifts from the visual presentation and layout of the program to differences in what can be done with the different tools and how one goes about accomplishing it.

Up to this point in our discussion, we have grouped the three conditions of the blocks-based environment together. As a reminder, the three versions of Snap! that students used were: an all-graphical version (the lightest colors in Figure 5), a read-only version of Snap! where students could read JavaScript versions of the programs they authored (the middle shade of blue/red in Figure 5), and a read-write version that added the ability for learners to define the behaviors of new blocks by writing short JavaScript programs (the darkest shade of the colors in Figure 5). For the most part, there was little difference between the conditions in students' responses. One notable exception is in the Ease of Composition category, which was rarely cited as a difference between the blocks-based and text-based tools in the read-write condition. This is unsurprising given that in this version of Snap! it was possible for students to write small snippets of text-based code, thus making the ease of composition a less prominent difference. A second major difference among responses across the three conditions was the number of differences cited by students in the read-write condition that fell outside of the larger categories. After recoding the Other responses, we were unable to identify a pattern to explain this difference and leave it as an open question we hope to return to in future analyses of the data we collected in this study.

4.4 Drawbacks to blocks-based programming

While most of the differences we have presented thus far have generally showed students holding a positive impression of the blocks-based programming approach, stemming from its perceived ease-of-use, the use of blocks-based tools for introducing high school students to programming was not entirely unproblematic. Over the course of our ten-week study, students identified a number of drawbacks to blocks-based programming. We present these drawbacks to shed light on reservations students have regarding the use of this strategy in formal introductory programming contexts at the high school level. The data we present below were drawn from both the introductory interviews we used to answer our first research question as well as the survey responses given at the midpoint and conclusion of our study asking students to compare the Snap! and Java programming environments. Across this dataset, three drawbacks to programming in a block-based environment were raised.

4.4.1 Less Powerful

The first drawback to blocks-based programming students cited was that block-based programming was viewed as a less powerful programming technique compared to the text-based alternative. By power, we are referring to the set of things that are possible with the language. As one student said, with text-based programming "*you can do a lot more.*" A second student reiterated this point, saying: "*blocks are limiting, like you can't do everything you can with Java, I guess. There is not a block for everything*." This comment is interesting as one could rebut that there is not a command for everything in Java either. The student who made this comment did not know how to program in Java, but nonetheless held the belief that the two representations were not equally powerful or expressive. Another student made these same points saying: "*In Java you can make it more complex than something you make in Snap! or Scratch."* She then continued: "*I'm pretty sure there are going to be some things that are too big to put in blocks...too complex*." This student viewed the blocks-based interface as a simplified version of Java, saying: "*I think what Snap! does it just takes the simpler things in Java and then turns them into blocks.*" This last statement is particularly interesting given that the available set of primitives provided by Snap! is largely a superset of the keywords reserved in Java, not the other way around. When asked why we chose to start the course with Snap! before moving to Java, a grade ten student responded: "*to increase understanding of programming. I mean like, Snap! is an awesome program, but there is only so much you can learn in it. But in Java, you can like figure out how to do like, all the other stuff.*" When pressed, the student was unable to articulate what "*other stuff*" consisted of, but still, this reveals a perceived limitation of what can be accomplished with blocks-based programming environments. In our post survey, one student summed up the difference between Java and Snap! succinctly by saying of Java: "*there are more possibilities*."

4.4.2 Slower Authoring and More Verbose

The second drawback brought up by a number of students was the time and number of blocks it takes to compose a program in the blocks-based interface compared to the text-based alternative. For example, when comparing Snap! to her previous experience making web pages, a 9th grade interviewee said: "*I know you have the variables* [in Snap!] *that you can edit and mess around with but sometimes that takes a lot of time, but HTML and CSS you can kind of get creative and quickly just type something in to do something different*". This was reiterated by a second student who said: "*if you want a specific block and it's not there, you're going to have to put a lot of blocks together to make it do what you want it to do, and I think with JavaScript, it's just, like, one sentence I guess.*" While it is unclear what is mean by a "*sentence"* in JavaScript, this comment does give us insight into how the student perceived text-based programming to be advantageous. Text being more concise was identified as not only useful for composing programs, but students also thought that the resulting shorter text-based programs could be easier to understand. "*It seems like when there is more blocks it's more confusing…when we did the games, we did a lot of, like a whole bunch of blocks, it was really hard to find where mistakes were.* [Text-based programming] *seems easier when there is like a lot.*" During our five-week study, programs rarely exceeded the size of the screen the students were working on, but in this case, the students experience with longer blocks-based programs lead to the recognition that longer blocks-based programs can be difficult to manage.

4.4.3 Inauthentic

The third and final drawback identified about the use of blocks-based tools is potentially the most damaging with respect to the effectiveness of their being used in introductory programming courses for older learners. Some of the students we interviewed expressed concerns over the authenticity of blocks-based programming. By authenticity, we mean how closely the programming tool and practices adhere to conventional, non-educational programming contexts. As one student said: "*Java is actual code, while Snap! is something nobody will let you code in.*" This same point was made by another student who said: "*if we actually want to program something, we wouldn't have blocks.*" It is important to note that this view was not universally held. As part of our interview protocol, students were asked if they thought what they were doing in Snap! constituted programming, to which every student answered in the affirmative. A number of students recognized blocks-based programming as being an introductory tool, giving responses like "*I think* [blocks-based programming] *is the same thing, just easier*" and "*I would say* [blocks-based programming] *is like beginners programming*". This suggests that even when perceived as potentially inauthentic, students still recognize the pedagogical usefulness of blocks-based tools. This drawback in particular seems like it is more likely to affect older learners who are eager to develop skills that can be used beyond the classroom, be it for a job or further computer science coursework.

These three perceived issues with blocks-based programming expressed by our participants gives us insight into potential drawbacks of the use of this approach at the high school level. We do not see this finding as undermining the use of blocks-based programming in formal, high school contexts, but instead, see these data as providing a fuller picture of how students perceive the tools we use for instruction, which in turn can be used to better inform educators on how to best utilize them in their classrooms. Further, identifying these perceived drawbacks can provide a roadmap for the improvement of these tools moving forward.

5. DISCUSSION

This study looks at the use of block-based programming tools in high-school introductory programming contexts. This means older students learning in a formal setting, which is quite distinct from the younger audience and informal settings that Scratch, and many of the tools inspired by Scratch, were initially designed for [26]. Much of the empirical work that has been done on these tools focuses on the younger audiences and informal spaces that match the audience they were initially designed for [e.g. 9, 20, 21]. Given the role these tools are increasingly playing in introductory computer science classes at the high school level, it is important to understand how high-school aged learners are making sense of them and if they are effective in their role of introducing learner to the programming component of the field of computer science. As the data in this study show, high school students generally find blocks-based programming tools to be easier to use than conventional text-based alternatives. They attribute this ease-of-use largely to visual features of the environment including the graphical presentation of the blocks, the drag-and-drop mechanism for authoring programs, and the ease of browsing the available set of blocks to figure out what commands to include in the program. This suggests that such blocks-based tools are effective for making it easier for high school aged students to get started programming, but that is not the whole story. Our study also found that some high school students see drawbacks to the use of such tools. Students saw the blocks-based tools as less powerful, potentially more cumbersome to use for larger projects, and inauthentic relative to conventional text-based programming. These findings are similar to what DiSalvo found in her analysis of high school students' preferences after work with both Alice and Jython (a text-based language) [6]. That study had the additional finding that student motivation and interest further influences student perceptions.

One obvious take-away from this research is to make teachers aware of these findings. Teachers can highlight the useful features of the environments while also addressing and downplaying concerns students have, like the perceived inauthenticity or lack of expressive power that some students associate with blocks-based interfaces. The presence of the teacher is a feature of formal education spaces that we can leverage to alleviate some of these drawbacks. Additionally, as designers, we can use the findings reported above to potentially improve both the graphical introductory tools as well as the text-based programming environments the students use as they move on to Java, Python, or whatever languages await them.

Knowing the strengths and drawbacks of blocks-based programming environments as perceived by the high school learners that are using them, helps inform us as educators and designers about what is working, what aspects of their design we might want to modify for the high school audience, and what features of these tools we might want to introduce to text-based environments for novices. For example, to address authenticity, the blocks-based environments could support not just controlling on-screen sprites, but also make it possible to have programs read from and output to a conventional terminal, akin to what many early Java programs do. The idea is not to replace the stage or introductory activities with less engaging text-only exercises, but instead to reinforce the similarities between programming in blocks-based tools and text-based languages; to provide a concrete way to more directly show the isomorphism between the two types of programming by making it possible to write (and run) the exact same program in blocks-based tools and Java to see how the two tools are similar. Such an addition to a blocks-based environment could start to break the perception that blocks-based tools can only be used to control graphical sprites and show how it can be used to accomplish what students might perceive as more authentic programming tasks.

Just as our findings can inform the design of blocks-based tools, so to can they be used to improve introductory text-based environments. For example, students frequently cited the browsability of the blocks-based environment as a feature that made it easy to use. Why not add a similar browsability to introductory text-based environments? The auto-complete feature of modern programming environments is similar to this, but is not curated or displayed in the same way the blocks environments are, where the commands are persistent and grouped by function. Adding an easily browsed, well organized library of valid commands that lives inside the Java or Python programming environment is one example of how we can use what we learn from novices about what makes blocks-based tools easy to improve and better prepare them for the transition to the text-based tools that await them in more advanced courses.

The final point we want to make is on the importance of recognizing the gap between how novices and experts approach a program or, in our case, a programming environment. When a seasoned programmer looks at a blocks-based language, the meaning of the shapes and colors of blocks are immediately apparent, with the most common response being "*how clever,*" as they can see how the blocks convey syntactic information and obviate some of the less obvious features of programming languages (like semi-colons and curly braces). It cannot be assumed that novices see the same thing. They have no prior knowledge of the syntax that is being represented and do not know what difficulties the graphical representation is alleviating from the act of programming. This is apparent as we have seen students using Snap! and other blocks-based tools write their first program left to right, completely ignoring the shape of the blocks. Upon telling students that programs are written top-to-bottom and showing the how blocks fit together, the shape of the blocks start to take on meaning, but it is important to state that we cannot take this knowledge for granted. We bring this point up as a reminder of the expert blindness that a designer who already knows how to program brings to the design of introductory programming tools. The solution to this is to remember that learners are the experts when it comes to understanding how novices make sense of and build an understanding of the practice of programming. As such, it is critical that we conduct studies like the one presented here, analyzing tools not from the perspective of those who have already mastered the content, but instead from the perspective of the learners who the tools is designed for.

5.1 Limitations

There are a few limitations to the study we conducted that may affect the generalizability of the findings. First, the school we conducted our study in is a selective enrollment school with an exceptional computer science department. This issue is partially mitigated by the fact that the school is public and the selectivity criteria for enrollment are designed to ensure the student body reflects the racial and socio-economic diversity of the district, but we do recognize that the students we worked with were exceptionally bright and motivated. Second, in these interviews, while we were trying to ask questions about the nature of blocks-based programming, we often got responses that were specific to the Snap! and Java or JavaScript tools we were using. Students struggled to disentangle the specific instance from the larger class of languages that we were using them to represent. While we recognize this drawback, we still find student responses to be insightful as to differences between the two modalities and, as we move forward in our work, intend on adding new languages and tools to address this. Finally, the study design followed students for five weeks in Snap! and five weeks in Java, but the amount of material that can be covered in five weeks is vastly different. Wherein after five weeks of Snap! students were using conditional logic, variables, and loops, in Java, at the end of five weeks, we had only covered basic I/O and started calling methods, and thus, had not discussed many of the other topics we covered in Snap! This difference in coverage speaks to the ease of teaching with Snap! but also means students had not been exposed to the same set of material in the two modalities.

6. CONCLUSION

Blocks-based programming is becoming the standard way to introduce learners to programming both inside classrooms and beyond. Educators and designers advocate for this approach arguing that it is easier to get started and more engaging for the learner. In this paper, we sought to understand how high school students enrolled in an introductory programming course perceived the blocks-based programming approach. Through cognitive interviews and surveys, we found that students generally found blocks-based programming to be easier than the text-based alternative, citing reasons including the natural language labels on the blocks, the shapes and colors of the blocks, the drag-and-drop composition mechanism, and the ease of browsing the blocks library. Students also identified drawbacks to the blocks-based programming approach, including issues of authenticity, expressive power, and challenges in authoring larger, more sophisticated programs. We also found that the differences high school students see between blocks-based and text-based programming span the visual interface, the types of programs that can be authored, as well a different programming practices that each representation supports. By analyzing student responses, we can better understand how the learners themselves are making sense of these introductory tools, isolate what they are identifying as useful about the environment in advancing their developing understanding, and use these insights to improve the tools we are currently using in classrooms and inform the design of the next generation of introductory programming environments. Our hope is that by doing so, we can better prepare today's students for the computational challenges of tomorrow.

7. REFERENCES

[1] Astrachan, O. and Briggs, A. 2012. The CS principles project. *ACM Inroads*. 3, 2 (2012), 38–42.

[2] Begel, A. and Klopfer, E. 2007. Starlogo TNG: An introduction to game development. *Journal of E-Learning*.

[3] Bontá, P. et al. 2010. Turtle, Art, TurtleArt. *Proc. of Constructionism 2010 Conference* (Paris, Fr., 2010).

[4] Bruckman, A. and Edwards, E. 1999. Should we leverage natural-language knowledge? *Proc. of the SIGCHI conference 1999*, 207–214.

[5] Cooper, S. et al. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*. 15, 5, 107–116.

[6] DiSalvo, B. 2014. Graphical Qualities of Educational Technology: Using Drag-and-Drop and Text-Based Programs for Introductory Computer Science. *IEEE computer graphics and applications*. 6, 12–15.

[7] Donzeau-Gouge, V. et al. 1984. Programming environments based on structured editors: The MENTOR experience. *Interactive Programming Environments*. McGraw Hill.

[8] Duncan, C. et al. 2014. Should Your 8-year-old Learn Coding? *Proc. of WiPSCE* 2014 (New York, USA), 60–69.

[9] Esper, S. et al. 2013. CodeSpells: embodying the metaphor of wizardry for programming. *Proc. of ITiCSE*, 249–254.

[10] Feurzeig, W. et al. 1970. Programming-languages as a conceptual framework for teaching mathematics. *SIGCUE Outlook*. 4, 2, 13–17.

[11] Fields, D.A. et al. 2014. Programming in the wild: trends in youth computational participation in the online scratch community. *Proc. of WiPSCE* 2014, (New York, USA) 2–11.

[12] Fraser, N. 2013. *Blockly*. Google.

[13] Goode, J. et al. 2012. Beyond curriculum: the exploring computer science program. *ACM Inroads*. 3, 2, 47–53.

[14] Guzdial, M. 2004. Programming environments for novices. *Computer Science Education Research*. 2004, 127–154.

[15] Harel and Papert. 1991. *Constructionism*. Ablex Publishing.

[16] Harvey, B. and Mönig, J. 2010. Bringing "no ceiling" to Scratch: Can one language serve kids and computer scientists? *Proc. of Constructionism 2010* (Paris, Fr.), 1–10.

[17] Hollan, J. et al. 2000. Distributed cognition: toward a new foundation for human-computer interaction research. *ACM Trans. on Computer-Human Interaction*. 7, 2, 174–196.

[18] Horn, M.S. et al. 2014. Frog pond: a codefirst learning environment on evolution and natural selection. *Proc. of the 2014 IDC conference*, 357–360.

[19] Horn, M.S. and Wilensky, U. 2012. NetTango: A mash-up of NetLogo and Tern. *Paper presented at AERA 2012.*

[20] Horstmann, C.S. 2012. *Java Concepts: Early Objects*. Wiley

[21] Hutchins, E. 1995. How a cockpit remembers its speeds. *Cognitive science*. 19, 3, 265–288.

[22] Kelleher, C. and Pausch, R. 2005. Lowering the barriers to programming: A taxonomy of programming environments for novice programmers. *ACM Computing Surveys*. 37, 2.

[23] Lego Systems Inc 2008. *Lego Mindstorms NXT-G System*.

[24] Lewis, C.M. 2010. How programming environment shapes perception, learning and goals: Logo vs. Scratch. *Proc. of the 41st ACM Technical Symposium on CSE*, 346–350.

[25] Maloney, J.H. et al. 2008. Programming by choice: Urban youth learning programming with Scratch. *ACM SIGCSE Bulletin*. 40, 1, 367–371.

[26] Maloney, J.H. et al. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education*. 10, 4, 16.

[27] Miller, P. et al. 1994. Evolution of novice programming environments: the structure editors of Carnegie Mellon University. *Interactive Learning Envs*. 4, 2, 140–158.

[28] Papert, S. 1980. *Mindstorms: Children, computers, and powerful ideas*. Basic books.

[29] Resnick, M. et al. 2009. Scratch: Programming for all. *Comm. of the ACM*. 52, 11, 60.

[30] Roque, R.V. 2007. *OpenBlocks: An extendable framework for graphical block programming systems*. MIT.

[31] Da Silva Gillig, J. 2014. *miniBloq*.

[32] Slany, W. 2014. Tinkering with Pocket Code. *Proc. of Constructionism 2014* (Vienna, Au.).

[33] Turkle, S. and Papert, S. 1990. Epistemological pluralism: Styles and voices within the computer culture. *SIGNS: Journal of Women in Culture and Society*. 16, 1, 128–157.

[34] Weintrop, D. and Wilensky, U. 2012. RoboBuilder: A program-to-play constructionist video game. *Proc. of Constructionism 2012* (Athens, Gr.).

[35] Weintrop, D. and Wilensky, U. 2013. Supporting computational expression: How novices use programming primitives in achieving a computational goal. Paper presented at AERA 2013.

[36] Wilkerson-Jerde, M.H. and Wilensky, U. 2010. Restructuring Change, Interpreting Changes: The DeltaTick Modeling and Analysis Toolkit. *Proc. of Constructionism 2010* (Paris, Fr.).