Tackling the Transition from Block-based to Text-based Programming Languages

Luke Moors
Computer Science Department
University of Auckland
New Zealand
lmoo228@aucklanduni.ac.nz

ABSTRACT

Block-based programming environments are becoming increasingly popular as introductory tools for teaching programming to children. These environments differ significantly to their text-based counterparts and have proven to be successful in motivating children and making it easy to start programming. However, several studies have recognised drawbacks to these tools where they could potentially be detrimental when transitioning to text-based languages. In this paper, the distinguishing differences between block-based environments and text-based languages are discussed and the effectiveness of these differences are explained. In considering the transition to text-based languages, this paper identifies two significant weaknesses to block-based programming and suggestions for improvement are discussed.

Keywords

Children; Computer Science Education; Block-based programming

1. INTRODUCTION

It is widely acknowledged that programming is difficult to learn [5, 9, 11, 21, 26]. Novice programmers have great difficulty in typical programming tasks such as predicting the output of a program, identifying the correct order of commands, or writing a simple program to solve a task [21]. Many of these tasks are challenging to novices as they are required to think about solutions in a way that is not natural [18]. To write a simple program they are required to consider both the syntax and semantics of a programming language where programming terms and commands do not translate well into standard English [26]. Moreover, introductory programming courses fail to connect with the interests of students [26]. It is these obstacles that may deter students from pursuing a career in programming, and can potentially create a lack of motivation to even begin [19].

To alleviate the frustration of understanding both syntax and semantics and to improve interest among children, a myriad of educational programming environments have been developed [9, 15], many placing a large emphasis on the creation of multimedia content [19, 28]. These programming environments typically remove the ability to make syntax errors and simplify the programming language, allowing children to focus on understanding the core computer science concepts and the structure of a program.

In recent years there has been a large increase in users of block-based programming. In *Scratch* - a block-based environment launched in 2007, there has been an increase from 78,000 to 174,000 monthly active users in the past two years alone [27]. This style of programming implements a block-like structure where blocks of code fit together like jig-saw pieces. These blocks differ in shape and colour to provide cues about how instructions can be assembled and to differentiate between concepts [26]. In addition, the environments typically encourage novice programmers by allowing them to create media-rich content in relation to their own interests. *Storytelling Alice*, in particular, implements a Storytelling approach found to be appealing to female students [10, 24]. On the other hand, *App Inventor* encourages novices using a block-based style of programming to provide the ability to create mobile applications in a simplified manner[19]. Many of the environments which use block-based programming (e.g. Scratch, Alice, App Inventor) aim to lower the barriers to programming, making it easier for beginners to start programming [7].

Much of the research related to the subject has reported the benefits of such environments including increased motivation and improved grades [3, 10, 13], however other studies have suggested potential drawbacks to block-based environments when transitioning to text-based languages [20]. After switching to text-based languages students are found to describe block-based environments as not "real programming" due to its simplicity. Furthermore, students are found to acquire poor programming habits while using block-based environments which can make it particularly difficult when switching to textual languages.

To understand what makes block-based programming environments motivating and simple, this paper will provide an overview of the distinguishing features of block-based programming environments and the existing research on the effectiveness of these features will be explored. Furthermore, to address the research on the drawbacks to block-based programming, this paper will discuss the literature on transitioning to text-based languages to give a detailed overview of their pedagogical value as educational programming tools.

This paper seeks to address the following research questions:

1. What are the key features that distinguish block-based programming environments from text-based languages and how effective are these features?

2. How effective are block-based programming environments for transitioning to text-based languages?

2. DISTINGUISHING FEATURES

Block-based programming environments share a number of characteristics. The environments use a multimedia context, where programmers can create art, games and stories [9] and they tend to target children [25]. In reviewing some of the most popular educational programming environments, a number of distinguishing features have been identified.

Many of the existing block-based programming environments have a similar set of features. Scratch, Alice, Lego Mindstorms and Blockly implement a block structure where blocks of code can be snapped together like Lego bricks [27]. As blocks are used instead of text, these environments remove the need for syntax, and having code instructions in blocks allows these environments to use a natural language that closely resembles a human conversation. Additionally, each of these environments use different colours and shapes to distinguish between concepts and indicate where blocks can be placed.

Although there are a number of features common among novice programming environments, there are also features which are specific to certain environments. For example, Alice differs from other environments by using a 3D virtual world which holds a collection of 3D objects(e.g. people, animals, props, vehicles) that can be manipulated [29]. On the other hand, Blockly differs to other environments as it allows code to be translated into different text-based languages like Python or JavaScript. In Scratch, concurrency is made possible as blocks can be placed in different areas of the screen and run in parallel. Personalisation is encouraged where users can operate with their own images, music and voice recordings. Moreover, Scratch encourages users to be social by allowing users to share their projects with the community and build on the projects of others. Users are also able to provide feedback on their work from other programmers.

As there are a number of features that separate block-based languages from one another, this paper focuses on the key features shared by majority of block-based programming environments: No syntax, Block-like structure, colour and shape, and simplified language.

2.1 No Syntax

Learning to program with a text-based language requires a lot of effort and the level of focus required for syntax is commonly viewed as the biggest obstacle in introductory programming courses [4]. As Malan and Leitner [13] explain: "students must become masters of syntax before solvers of problems". In Java - one of the most popular programming languages [2], the simple program (Figure 1) that prints "Hello World!" to the terminal involves an overwhelming amount of syntax to consider: opening/closing brackets, upper/lower case letters, semicolons, and also spelling. Even with other less syntactically demanding languages like Python, users are still required to consider indentation and spelling which can be challenging for novices.

For this reason, a major feature of educational programming environments is the elimination of syntax errors. Novice programmers are able to focus on the core concepts in computer science such as flow of control, conditionals and loops, without the frustration of ensuring a program is syntactically correct.

Syntax removal is evidently beneficial in making programming easier for novice programmers [23]. In student reported

```
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }

}
```

Figure 1: A simple Java program.

differences between Java and *Snap!* - an extended version of block-based tool Scratch, students highlighted how both commands and syntax are required in Java, whereas Snap! removed the need to memorise syntax [31]. When programming in Alice, the elimination of syntax was considered to be a contributing factor to the success children had when using the environment [23]. However, the removal of syntax simplified programming to the extent that when students moved to Java, there were reports of "syntax overload". The consequence of this is discussed in Section 3.

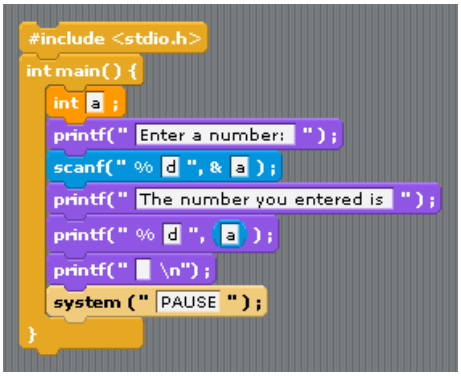

Figure 2: A simple program made with BlockC.

Other research has focused on designing block-based versions of specific languages to ease the transition into text-based languages [4]. In particular, Federici developed *BlockC* - a block implementation of the C programming language. In this environment, a number of new blocks were created where each block included C-like syntax (see Figure 2). With syntax similar to what one would expect in text-based languages, it creates the opportunity for novices to recognise correct syntactic structure without worrying about its implementation. This type of environment could be promising if used as an intermediate step between existing block-based environments and textual languages.

2.2 Block-like Structure

Block-based environments use a drag-and-drop interface where blocks of code can be placed at different locations on the screen to form syntactically correct programs. Children recognise the benefits in using blocks where they alleviate the need for memorization as programming blocks are made easily accessible and are organised into distinct categories [31].

A number of studies have evaluated the positive effect from using block-like structures to convey instructions [25, 31]. Price and Barnes [25] assessed the differences between textual and block interfaces using Tiled Grace, a program-

ming environment that allows for both textual and block interfaces. Using this environment, a program could be written with either interface while using the same text, the only difference being that in the block interface text was contained within blocks. The students in each condition were provided with a programming exercise which consisted of 9 independent goals that tested different concepts. It was found that students in the Block group outperformed the Text group for all goals. In addition, of all students that completed the goals, the block group spent significantly less time to complete them, suggesting that the block-based interface was easier to use.

There have been attempts to increase the similarities between block-based environments and text-languages by adding the ability to define new blocks and higher-order functions to existing environments [7]. By adding new blocks it allows expert block-based programmers to learn higher programming constructs before switching to text-based languages. However, as Monig et al. [16] explain, experienced users of block environments may find the drag and drop interface cumbersome. The large amount of time it takes to find an appropriate block can interrupt the user's flow of thought and this problem can worsen as the number of available blocks increases.

To diminish issues surrounding drag and drop interfaces like the large amount of space blocks occupy and the amount of time spent searching for blocks, Monig et al. [16] suggest techniques which could act as an intermediate step between block programming and textual programming. To lessen the amount of space occupied by blocks in the script area, the authors suggest using blocks that look like text while maintaining the same structure of blocks. This could also improve novice attitudes when transitioning by having a closer resemblance to textual languages. A second technique explained was to use keyboard-based block editing. Expert users could begin typing the name of a block they want to use and a drop down menu would appear with possible matches. Using this technique, experienced users are not burdened with the need to drag blocks across the screen, while novices are able to progress at a standard rate.

2.3 Colour and Shape

Educational programming environments often implement blocks of different shapes to distinguish concepts from one another. This also provides cues as to which blocks can be connected to create a syntactically correct program. In Scratch, blocks that accept Booleans are hexagonal shaped and numbers are oval shaped. Likewise, Boolean and number values are shaped accordingly (see Figure 3). Furthermore, control structures are C-shaped to indicate that blocks can be placed inside [26].

Novice programmers identify the shape of blocks as a key feature that makes block-based programming easy. In an interview with students that had programmed in both Snap! and Java, four out of nine students said the shape of blocks was a key reason for why block-based programming is easy [31]. One student commented that the visual layout of blocks "teaches you that order is important", showing an understanding of the importance of order in a sequence of commands.

Similar to the way text-based environments use colour to distinguish between different types and keywords (see Figure 1.), block-based environments also use colour to separate

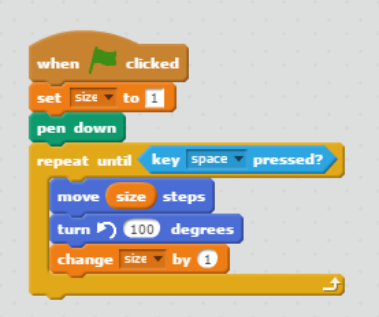

Figure 3: A Scratch script highlighting the different block colours and shapes.

categories of blocks. However, it is not clear whether this feature is useful in demonstrating the different categories. It is possible that the different colours of block could become confusing when tracing through a large program. Weintrop and Wilensky [31] suggest that novices may not immediately recognise the significance of block shapes and colours. Novice users may not have any experience of syntax in programming to understand the purpose of using shape and colour.

2.4 Simplified Language

The languages used in text-based languages can be difficult to interpret for children. Pea [21] argues that novices of all ages, tend to hold the notion that "there is a hidden mind somewhere in the programming language that has intelligent, interpretive powers". The notion stems from the tendency for novices to resort to an analogy of conversing with a human when writing programs. *Psychology of Programming* [8] supports this notion where it is identified that there is a mismatch between the way humans and computers think. Nielsen [17] explains how user interfaces should "speak the user's language" and how there should be a good mapping between the user's conceptual model and the computer's interface. By designing a programming environment with natural language, it can make the interface easier to understand and creates a stronger mapping between the user's existing knowledge and the problem at hand [18].

To accommodate for the analogy on a conversation, educational programming environments implement languages in a way which resemble having a conversation and are more closely aligned with students existing mental models of problem solving [18]. In a study involving young student perceptions of block-based environments [31], students identified the easy to read labels on blocks as the most helpful. The labels were described as being a closer representation of English where students pointed out their resemblance to having a conversation.

Storytelling Alice implements a number of methods that resemble a human conversation more closely than text-based languages (see Figure 4). For example, a user can select an object, such as a Magician, and can then call the method "when mouse is clicked on Magician, do levitate". In a comparison between Storytelling Alice and Generic Alice - a version of Alice without storytelling support, participants who used Storytelling Alice were found to spend 42% more time editing their program and were three times more likely to spend extra time programming after the session had ended

Generic Alice methods:
move, turn, roll, resize, play sound, move to, move toward, move away from, orient to, point at, set point of view to, set pose, move at speed, turn at speed, roll at speed
Storytelling Alice methods:
say, think, play sound, walk to, walk offscreen, walk, move, sit on, lie on, kneel, fall down, stand up, straighten, look at, look, turn to face, turn away from, turn, touch, keep touching

Figure 4: A comparison of the language used in Generic Alice and Storytelling Alice.

[10]. They were also found to express greater interest in future use of the environment.

Weintrop and Wilensky [32] also provide support for the benefits of using a natural language in block-based environments. In their research, they looked at the impact of block-based versus text-based languages on novice programmers' understanding of computer science concepts. Students were asked questions about different programming concepts in both text and block modality. Results of the experiment found students to be significantly better at conditional logic, procedures, and iterative logic (see Figure 5). The authors concluded that the use of text versus blocks does have an affect on novices' understanding of concepts. Further, the authors identified several possible explanations for their finding. The structure of blocks was said to be helpful in describing scope. The natural language of block labels was more descriptive than their text counterparts. Finally, the shape of blocks were helpful in questions involving functions as the shape of blocks that return values are different to those that perform actions, which may have helped the students when answering the related questions.

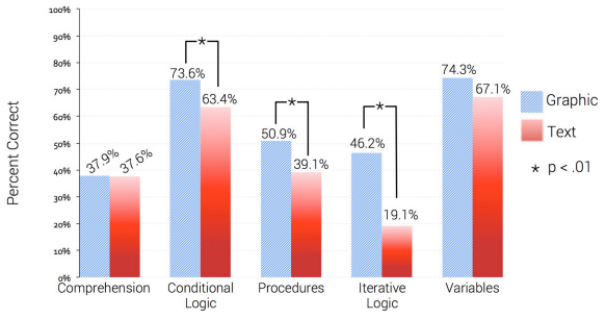

Figure 5: Performance for different concepts and grouped by block (graphic) and text.

3. TRANSITION TO TEXT LANGUAGES

Many of the programming environments mentioned above have aimed to lower the barriers to programming and make it easy for children to start programming. Although this goal has been achieved in most cases, some studies have suggested a detrimental effect when transitioning to text-based languages. From reviewing the literature on these environments, two important weaknesses were identified and are discussed below.

3.1 Lost Confidence

While many educational programming environments offer ease of use through the removal of syntax, it does not aid the learner in transitioning to a more traditional language that involves strict syntax. Instead, these environments postpone the inevitable requirement until the core programming concepts have been understood. This is done in hope of encouraging children to start programming and build up confidence so that they will be more willing to continue programming at a higher level [30]. However, this approach can have the opposite effect.

Several studies have reported students' loss of confidence after switching to text-based languages. Powers et al. [23] studied the transition to high-level programming languages where the first half of a course was taught with Alice and the second half with Java (and C++ the following year). The authors note that the transition was poor. Both strong and weak students were overwhelmed by the syntactical requirements and students quickly became discouraged when their programs would not compile. Most importantly, after using a High-level language, students lost confidence in their programming ability and concluded that Alice was not "real programming".

Similarly, in a study involving the knowledge transfer from text-based languages to Alice, Parsons et al. [20] concluded that "Students do not seem to naturally make a strong connection between the formal coding process and what they are doing with Alice.", where several students agreed that Alice was simplistic with little resemblance to "real programming". In these cases, the students failed to construct mental models of programming that are suitable for a text-based context.

In a study comparing the learning outcomes of students programming in Logo and Scratch, Lewis' [12] findings support this claim where students programming in *Logo* - a text-based language, identified themselves as having higher confidence in their programming ability compared to programmers using Scratch. This suggests that children may not identify their program writing in block environments as true programming.

Bau [1] has attempted to close the learning gap between block programming and text programming. In *Droplet* - a programming editor, users can load existing code written in Javascript or CoffeeScript and translate it into blocks (and vice versa) for easy editing. Droplet allows experienced block users to transition to commercial text-based languages easily with the ability to switch back if problems arise. This could prove useful in raising confidence levels for students that find block-based programming as not "real programming".

3.2 Incorrect Ideologies

In the process of designing a block-based environment that is easy to use, it is possible for children to develop ideas about programming which are not true. Weintrop and Wilensky [31] support this claim as they suggest blocks can lead children to form incorrect beliefs about programming. During an interview on programming in Scratch, when talking about the blocks, one student stated that "everything has its place". The student found it difficult to use the blocks as he believed there was a certain combination required to make his program. As every block in Scratch has a particular shape and can be snapped into place with another block,

children may develop the idea that every block has its own place. In text-based programming, this can be inhibiting to performance as instructions are not restricted to specific places but can be used in a variety of ways.

Other literature has identified bad habits that novice programmers acquire when using Scratch [14]. Bottom-up programming was a technique found to be taken to the extreme where students would start with the most basic components of Scratch and would link these elements to form a greater subsystem. Students would drag a number of instructions to the script area and would then try to piece them together appropriately.

This type of behaviour conflicts with the accepted practice in computer science that encourages programmers to start by designing algorithms to solve a problem. Environments like Scratch enforce this behaviour as programming elements are made visible at all times so programmers do not need to remember the instructions. This habit is further fostered by the ability to leave individual blocks of instructions in the script area without affecting the execution of the main program.

A second bad habit discovered by researchers was the use of extremely fine-grained programming [14]. Students took a top-down approach where tasks were broken down into smaller sub tasks that became incoherent. A common example of this behaviour was identified when a finite loop structure was required - a task repeated until a specific condition was reached (see Figure 6). However, with extreme fine-grained programming students would split this concept into multiple sub tasks: an infinite loop that performed the repetition and another which checked if the condition was met (see Figure 7).

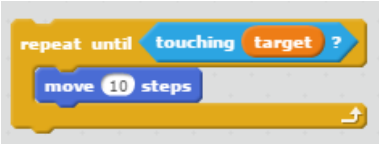

Figure 6: A simple implementation of a finite loop.

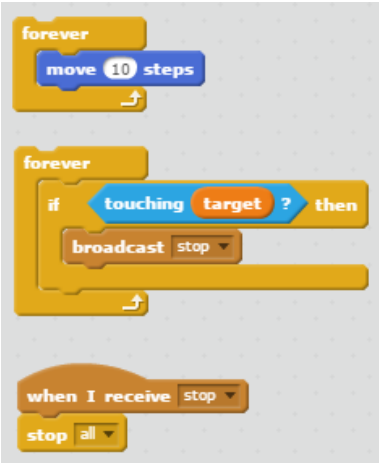

Figure 7: An EFGP implementation of a finite loop.

This style of programming is made simple by having the ability to run multiple scripts concurrently by placing each script in a different area of the screen. However, although the ability to perform concurrent behaviour can be advantageous [6], it can also be viewed as damaging when students are faced with problems surrounding concurrency in Scratch.

Using extreme fine-grained programming in their programs, students avoided the fundamental control structures: conditionals, and bounded loops. This behaviour is especially detrimental when transitioning to text-based languages as these concepts are known to be particularly difficult to learn and the opportunity is missed when students use environments which encourage this kind of behaviour [14].

4. DISCUSSION

It is evident that block-based programming environments are effective at lowering the barriers to programming and motivating for children. The elimination of syntax and block-like structures simplify the process and the focus on media creation is beneficial in connecting with the interests of children. However, there are mixed results on the effectiveness of these features and environments when transitioning to text-based languages.

Existing block-based programming environments have the possibility of reducing confidence levels in children from the unanticipated change toward syntax heavy languages. Furthermore, these environments can have damaging effects by instilling incorrect ideologies of programming behavior. Recent studies [1, 4, 7, 16] have attempted to improve popular existing environments but there is still plenty of room for improvement.

From the studies reviewed, two significant weaknesses were identified which may be harmful to students when transitioning to textual languages, but there may be other factors which have not been discussed. The focus on multimedia creation can be encouraging in the early stages of programming, however it may also bring students to refrain from progressing ahead if they become attached to this style of programming. Creating a game or animation in a textual language requires much more effort than in Scratch or Alice. Students may not see the value in switching to other languages if they can use block-based environments to create similar programs in less time, with fewer lines of code. If this is true then schools which use these environments in their curriculum should be emphasising the value of textual languages.

By understanding the strengths and weaknesses of block-based environments, educators can work towards implementing programming tools that are most beneficial for novices and for the transition to text-based languages. For example, if children are losing confidence after using textual languages due to their syntactical requirement, future block-based environments could incorporate the automatic placement of syntax so the structure of a program is a closer match to one written in text. By learning about the incorrect habits children acquire, educators could focus on teaching the appropriate skills about programming before using the environments. Early research in computer science explains that programming has a mathematical foundation with a strong application of problem solving [22]. If this material is taught with a core focus on programming, it has the ability to reduce the problems that children pick up while programming. With an understanding of how to improve block-based environments, educators can minimize the gap between block-based and text-based languages allowing for a smooth tran-

sition and an interest in computer science.

5. FUTURE WORK

The mixed results on the effectiveness of block-based environments indicate the need for future research in the area. While current programming environments prove to be beneficial in encouraging children to start programming, it would be useful to investigate ways to achieve this without focusing solely on media creation. Gamification elements such as levels and leader-boards could be applied to existing block-based languages with the focus on solving mathematical or logical problems to progress ahead. This way novices can remain motivated to progress while creating programs similar to those made in introductory courses that use textual languages.

Future research should also consider investigating the impact of using helpful syntax errors rather than no syntax at all. If blocks are placed incorrectly, the system could revert back to the last valid state while providing useful information about what went wrong. This may soften the transition into text-based programming if children are made aware of the kinds of problems that can occur and steps required to solve them.

6. REFERENCES

[1] Bau, D. Droplet, a blocks-based editor for text code. *Journal of Computing Sciences in Colleges 30*, 6 (2015), 138–144.

[2] Carbonnelle, P. Pypl popularity of programming language. http://pypl.github.io/PYPL.html, 2016. Accessed: 2016-8-29.

[3] Cooper, S., Dann, W., and Pausch, R. Teaching objects-first in introductory computer science. In *ACM SIGCSE Bulletin* (2003), vol. 35, ACM, pp. 191–195.

[4] Federici, S. A minimal, extensible, drag-and-drop implementation of the c programming language. In *Proceedings of the 2011 conference on Information technology education* (2011), ACM, pp. 191–196.

[5] Gross, P., and Powers, K. Evaluating assessments of novice programming environments. In *Proceedings of the first international workshop on Computing education research* (2005), ACM, pp. 99–110.

[6] Guzdial, M. Programming environments for novices. *Computer science education research 2004* (2004), 127–154.

[7] Harvey, B., and Mönig, J. Bringing "no ceiling" to scratch: Can one language serve kids and computer scientists. *Proc. Constructionism* (2010).

[8] Hoc, J.-M. *Psychology of programming*. Academic Press, 2014.

[9] Kelleher, C., and Pausch, R. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR) 37*, 2 (2005), 83–137.

[10] Kelleher, C., Pausch, R., and Kiesler, S. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (2007), ACM, pp. 1455–1464.

[11] Lahtinen, E., Ala-Mutka, K., and Järvinen, H.-M. A study of the difficulties of novice programmers. In *ACM SIGCSE Bulletin* (2005), vol. 37, ACM, pp. 14–18.

[12] Lewis, C. M. How programming environment shapes perception, learning and goals: logo vs. scratch. In *Proceedings of the 41st ACM technical symposium on Computer science education* (2010), ACM, pp. 346–350.

[13] Malan, D. J., and Leitner, H. H. Scratch for budding computer scientists. In *ACM SIGCSE Bulletin* (2007), vol. 39, ACM, pp. 223–227.

[14] Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education* (2011), ACM, pp. 168–172.

[15] Miller, P., Pane, J., Meter, G., and Vorthmann, S. Evolution of novice programming environments: The structure editors of carnegie mellon university. *Interactive Learning Environments 4*, 2 (1994), 140–158.

[16] Monig, J., Ohshima, Y., and Maloney, J. Blocks at your fingertips: Blurring the line between blocks and text in gp. In *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE* (2015), IEEE, pp. 51–53.

[17] Nielsen, J. *Usability engineering*. Elsevier, 1994.

[18] Pane, J. F., Myers, B. A., et al. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies 54*, 2 (2001), 237–264.

[19] Papadakis, S., Kalogiannakis, M., Orfanakis, V., and Zaranis, N. Novice programming environments. scratch & app inventor: a first comparison. In *Proceedings of the 2014 Workshop on Interaction Design in Educational Environments* (2014), ACM, p. 1.

[20] Parsons, D., and Haden, P. Programming osmosis: Knowledge transfer from imperative to visual programming environments. In *Conference of the National Advisory Committee on Computing Qualifications. Citeseer* (2007).

[21] Pea, R. D. Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research 2*, 1 (1986), 25–36.

[22] Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., and Paterson, J. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin 39*, 4 (2007), 204–223.

[23] Powers, K., Ecott, S., and Hirshfield, L. M. Through the looking glass: teaching cs0 with alice. In *ACM SIGCSE Bulletin* (2007), vol. 39, ACM, pp. 213–217.

[24] Powers, K., Gross, P., Cooper, S., McNally, M., Goldman, K. J., Proulx, V., and Carlisle, M. Tools for teaching introductory programming: what works? In *ACM SIGCSE Bulletin* (2006), vol. 38, ACM, pp. 560–561.

[25] Price, T. W., and Barnes, T. Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the eleventh annual International Conference on International Computing*

Education Research (2015), ACM, pp. 91–99.

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

[27] Scratch. Scratch statistics - imagine, program, share. https://scratch.mit.edu/statistics/, 2016. Accessed: 2016-8-28.

[28] Stolee, K. T., and Fristoe, T. Expressing computer science concepts through kodu game lab. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (2011), ACM, pp. 99–104.

[29] University, C. M. What is alice? http://www.alice.org/index.php?page=what_is_alice/what_is_alice, 2016. Accessed: 2016-8-29.

[30] Wagner, A., Gray, J., Corley, J., and Wolber, D. Using app inventor in a k-12 summer camp. In *Proceeding of the 44th ACM technical symposium on Computer science education* (2013), ACM, pp. 621–626.

[31] Weintrop, D., and Wilensky, U. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children* (2015), ACM, pp. 199–208.

[32] Weintrop, D., and Wilensky, U. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *Proceedings of the eleventh annual International Conference on International Computing Education Research* (2015), ACM, pp. 101–110.