Characteristics of primary school students' Scratch code

Dimitrios Nikolos, *dnikolos@upatras.gr*
Dept of Educational Sciences and Early Childhood Education, University of Patras, Greece
Vassilis Komis, *komis@upatras.gr*
Dept of Educational Sciences and Early Childhood Education, University of Patras, Greece

Abstract
In the present study we discuss characteristics of primary school students' Scratch code. Characteristics of code that appear in other Scratch code analyses include a) Dead code, b) Initialization problems, c) Extremely Fine-Grained Programming (EFGP), d) Duplicate code and e) Long scripts. Does student code reflect code that is found in large Scratch project repositories or other settings? Does student code have other characteristics? We taught two fifth-grade classes (23 students each) Scratch for 9 hours. Learning Scratch was part of a compulsory 1 hour per week ICT course. After each hour we gathered students' projects that comprised our data (code base). The analysis of the code base showed that many of the characteristics of code that were found elsewhere were also present in the classroom, namely, dead code, initialization problems and EFGP. Duplicate code and long scripts did not appear in the code base in a large percentage. We also discovered that students' projects featured race conditions and code that had no effect. Such codes provide learning opportunities. When students create these types of codes**,** they face a problematic situation and they will have to reflect on their code. Since Scratch makes tinkering easy by design, they may find a way to move forward, otherwise the teacher should be able to assist them.

Keywords
Scratch, code, primary school students, dead code, initialization, EFGP

Introduction

Scratch is a programming language that enables young students to create their own games and other types of interactive media (Resnick et al., 2009). Scratch originates from Logo and aims in enabling students to create meaningful artifacts. Doing so, they engage with a wide range of computational concepts that promote computational thinking (Brennan & Resnick, 2012; Meerbaum-Salant, 2013). These computational concepts can be found in the code of Scratch projects and include sequences, loops, parallelism, events, conditionals, operators, and data.

School teachers may use Scratch to extend student knowledge beyond basic office skills (Crook, 2010). Students face problems while programming with Scratch. Every problem that occurs when students create meaningful artifacts is a learning opportunity. When students overcome such problems, they construct their own knowledge. If they are unable to do so by themselves, teachers can look inside student code and help them. In most cases, teachers will find that some of the code works as intended and some other code has problems or doesn't work at all.

Code that can cause a Scratch project not to work as intended includes:

- Dead code: code that is not executed (Aivaloglou & Hermans, 2016). Dead code includes procedures that are not invoked, unmatched broadcast-receive messages, code that is not invoked and empty event scripts. Static analysis of code can locate dead code and it was found in more than 25% of the analyzed projects in (Aivaloglou & Hermans, 2016). Dr. Scratch, a tool that is used for evaluating computational thinking in Scratch projects also takes dead code into account (Moreno-León & Robles, 2015).
- Problematic initialization: changes in the state of the program that are not restored. An example of problematic initialization is a Scratch project that hides a sprite but never shows it again, not even when the program is restarted. The reason for this is that Scratch stores the state of the program between executions. By convention, a click on the green flag restarts the program (Franklin et al., 2016). Hairball can detect initialization problems using static code analysis (Boe et al., 2013).

Code that may hamper learning programming includes:

- Extremely Fine-Grained Programming (EFGP): Code that is fragmented in small logical blocks (Meerbaum-Salant, 2011).
- Long scripts: Large scripts that are not easily understood (Hermans & Aivaloglou, 2016).
- Duplicate code: Code that is repeated either in the same project or even in the same object (Hermans & Aivaloglou, 2016).

In the present study we determine a) which of the above code patterns can be found when primary school students learn Scratch in the classroom and b) what other problematic code, if any, emerges during the process. In the following section methodology is described, subsequently results are drawn, discussion and conclusions follow.

Methodology

The study was conducted with two fifth-grade primary school classes, during a compulsory ICT course. Each class had 23 students and they worked in groups of 2 or 3 students. The course took place in a primary school computer lab with 13 computers equipped with the Scratch 2.0 software.

The curriculum

We developed a curriculum 9 hours in duration. Students had to create a project that featured a basic Scratch concept in each hour. The Scratch concepts of each lesson were:

1. Introduction: Students navigate in the Scratch programming environment

2. Views: Students learn to change the appearance of Scratch objects and scene.

3. Interaction: Students learn to program interaction with the use of *"if <> then else"*.

4. Messages: Students learn to program broadcasting and receiving messages for synchronizing.

5. Variables: Students learn to add score in a game.

6. Recap: Students create a project with concepts they have already learned.

7. New blocks: Students create a project that features "New blocks" (procedures).

8. Clones: Students learn to program clones of their objects.

9. Students create their own games and finish the course.

After each lesson, students had to submit a project that featured the basic concept.

The projects

During the nine weeks of the course we gathered 152 projects that the students created (Table 1). We did not study the projects of the first week because they were very basic. The course took place in a regular school setting and we had common computer lab problems. Some students did not save their work and some computers malfunctioned during class. That is why there are different number of projects in each class and lesson.

Table 1. Number of projects

lesson	Number of projects	
	Class 1	Class 2
2. Views	9	8
3. Interaction	9	6
4. Messages	10	13
5. Variables	12	9
6. Recap	12	10
7. New blocks	11	12
8. Clones	6	9
9. Students create their own games	10	6
Total	152	

Some of the projects that the students created may not fit into the group of the projects of their peers. To find these outliers, we scored the projects using Dr. Scratch and created the boxplot of the scores grouped by lesson (Figure 1). The eleven outliers of the box plot were eliminated from the code analysis, leaving 141 projects that comprise the data (code base) of the present study.

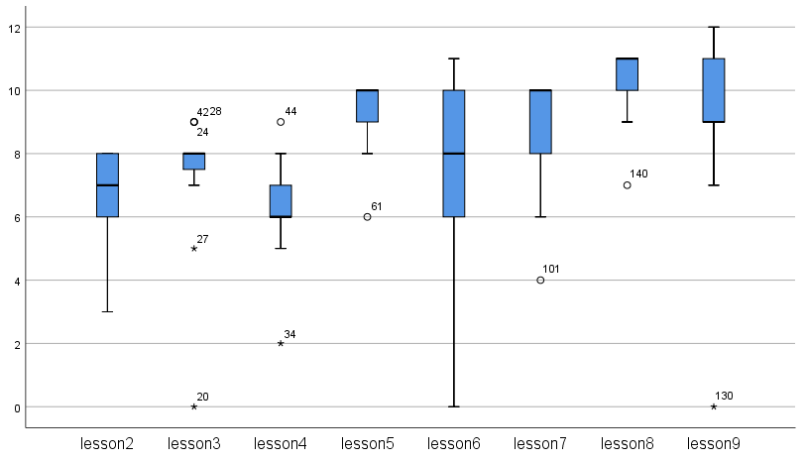

Figure 1. Dr. Scratch score of projects

The projects that the students created are small projects (Table 2). They feature 3.2 sprites on average, and 6.25 scripts per project, as opposed to projects downloaded from the Scratch

website that featured 5.68 sprites per project and 17.35 scripts per project on average (Aivaloglu & Hermans, 2016).

Table 2. Description of projects

Lesson	Average Number of sprites per project	Average number of scripts per project
2.Views	2.47	3.70
3.Interaction	3.2	4.90
4.Messages	3.6	3.76
5.Variables	3.05	8.30
6.Recap	4.05	5.91
7.New blocks	1.45	6.14
8.Clones	4.29	9.57
9.Own programs	3.53	8.33
Average	3.2	6.25

Each project was studied in respect to the following questions:

- How many blocks are never executed (dead code)?
- Does the project feature problematic initialization?
- Does the project feature Extremely Fine-Grained Programming?
- Does the project feature long scripts?
- Does the project feature duplicate code?
- Does the project feature other problematic code?

The results of this analysis are described in the next section.

Results

Dead code

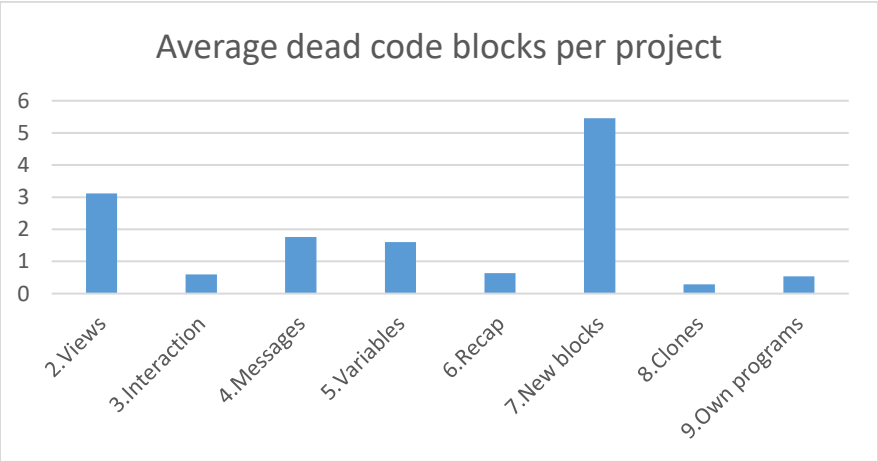

Figure 2. Average dead code blocks per project for every lesson

We counted the blocks that are not executed (dead code) in our code base. Projects contained 1.94 dead code blocks in average. Several projects contained no dead code. The maximum number of dead blocks that appeared in a project was 19. This project was submitted after lesson

7 (New Blocks) where we observed the maximum number of dead code blocks in general (Figure 2).

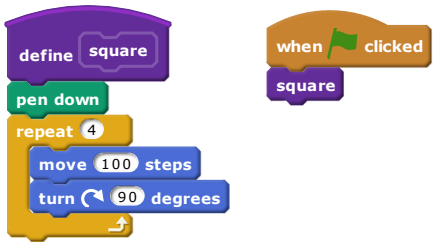

Figure 3. Complete example of "New blocks" usage

"New blocks" is the name of the way that Scratch implements procedures. A complete example of the usage of “New blocks” in a project is shown in figure 3. A new block is comprised of a definition and an implementation. It then needs to be called on an event in order to be executed.

Dead code was present in 18 out of the 22 submitted projects after the "New Blocks" lesson. We documented the reasons for dead code occurrence in relation to the concept of "New Blocks". We found four reasons for dead code occurrence: a) students did not attach the blocks of the implementation to the new block definition, or omitted the definition (4 projects), b) students did not call the new block at all (5 projects), c) students did not call the new block on an event (10 projects) and d) students called the new block in the wrong sprite (1 project).

Problematic initialization

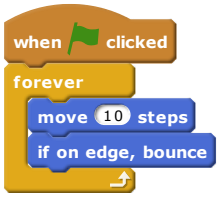

Figure 4. Example of code with no need for initialization

As far as initialization is concerned a project can have either:

a) correct initialization: the state of the program resets (usually on green flag click),
b) problematic initialization: the state of the program does not fully reset, or does not reset at all
c) no need for initialization: the state of the program does not need to reset. An example of a code that changes the state, i.e. the position, of a sprite without need for initialization is shown in figure 4. The sprite is continuously moving, and its starting position is unimportant.

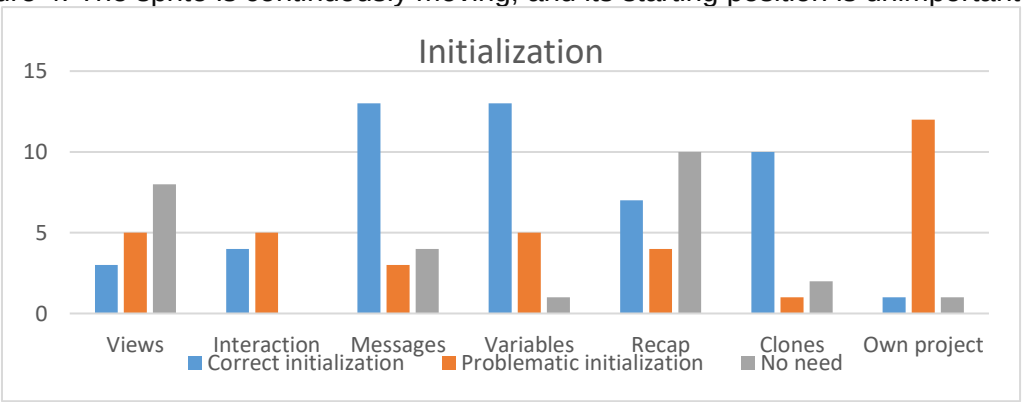

Figure5. Initialization of projects

We counted the number of projects that featured correct initialization, problematic initialization and no need for initialization (Figure 5). We noticed that when students created their own games, they did not implement correct initialization. The reason for the correct initialization in projects of lessons about messages and variables is that initialization was a vital part of these lessons.

Extremely Fine-Grained Programming

When a component of the program, e.g. a variable, is accessed or modified by two or more scripts in the same sprite, the script features Extremely Fine-Grained Programming (EFGP). We counted the occurrences of EFGP (Table 3) and found that around a quarter of the projects featured EFGP.

Table 3. Number of EFGP occurrences

	EFGP Projects	Total projects	Percentage
2. Views	7	17	41%
3. Interaction	3	10	30%
4. Messages	4	21	19%
5. Variables	4	20	20%
6. Recap	0	22	0
7. New blocks	16	22	72%
8.Clones	0	14	0
9. Own games	1	15	7%

While teaching new blocks (lesson 7) students created code like that of figure 6. The left script draws a square and the right script draws a triangle. However, if the sprite is clicked an irregular shape will be drawn because of the concurrent model of execution. Such codes account for the increased number of EFGP projects in lesson 7.

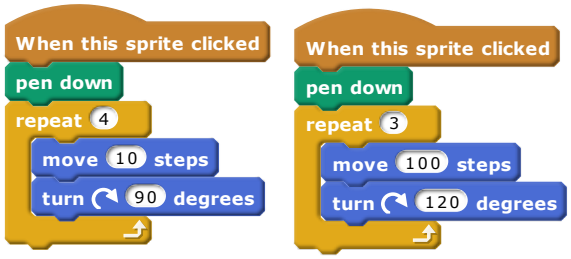

Figure 6. Extremely Fine-Grained Programming after lesson 7

We noticed that when students program their own games, EFGP does not occur (lessons 6 and 9). A possible explanation for this is that students created relatively small projects.

Long scripts

In figure 7 the distribution of projects per maximum script size is shown. Most projects feature maximum script size between 5 and 10 blocks. Only 1 out of the 142 projects featured a very long script with 32 blocks in one script. This script is shown in figure 8. The code checks if a sprite is touching one of the five bad apples that existed in the project. This code can be shortened using logical operators or clones.

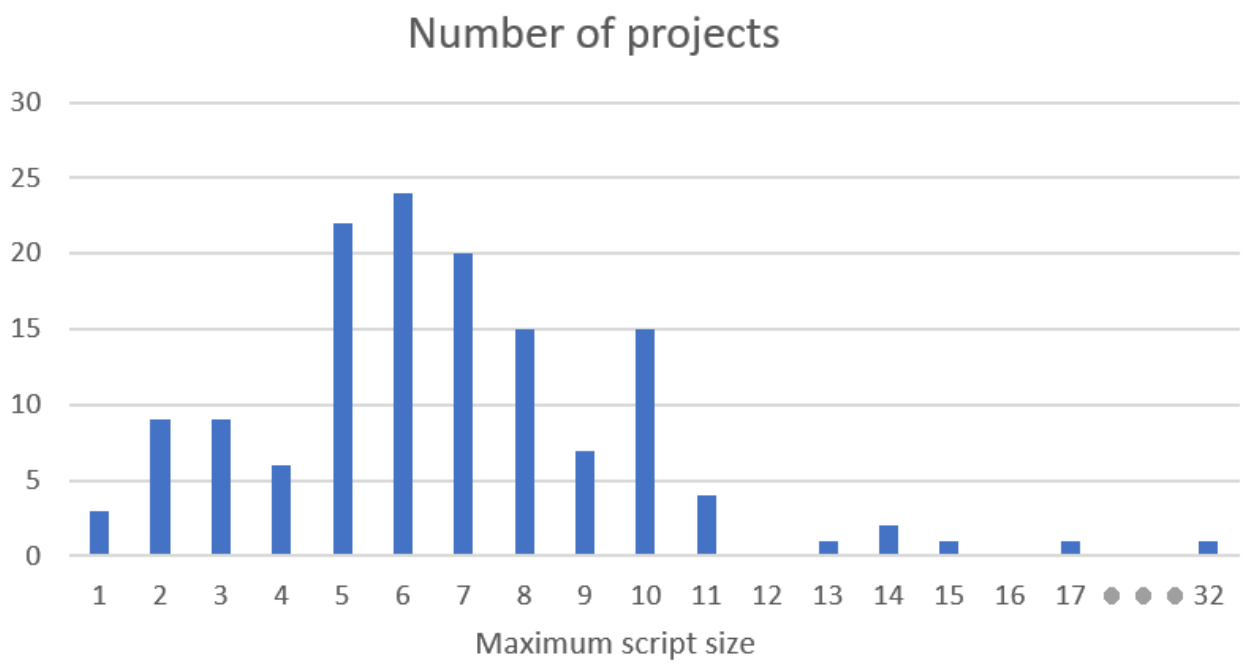

Figure 7. Distribution of number of projects per maximum script size

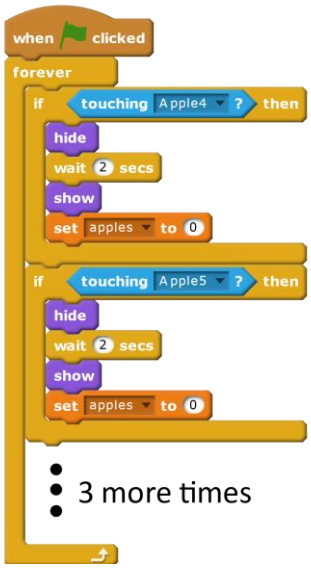

Figure 8. The long script

Duplicate code

Only 5 of the 142 featured projects featured duplicate code. As with long scripts the reason might be that students created small projects in this course. Interestingly, the three of the five projects that featured duplicate code were submitted after the students were familiarized with clones. When the course created the need for multiple sprites, student responded by creating duplicate code.

Other problematic code

We noticed that the already studied problematic code categories (dead code, problematic initialization, EFPG, long scripts, and duplication) were not the only ones that appeared in our code base. Others include a) race conditions and b) code without effect.

Race conditions

When students design games they want to program concurrent events (Kafai, 1995). Scratch provides a concurrent environment that makes these events possible (Maloney et al., 2010). Even though concurrency is intuitive, since we live in a concurrent world, problematic concurrent code emerges in Scratch programming (Meerbaum-Salant, et al., 2011). A case of problematic concurrent code occurs when the program modifies a component, e.g. a sprite attribute, in scripts that are executed at the same time. In that case, the code outcome depends on the way Scratch executes the program. Such codes create race conditions in concurrent programming languages (Netzer & Miller, 1992). In the example of figure 9 there is no way to determine the costume of the sprite by inspecting the code. The student cannot predict the outcome of the code, they can only observe it after execution.

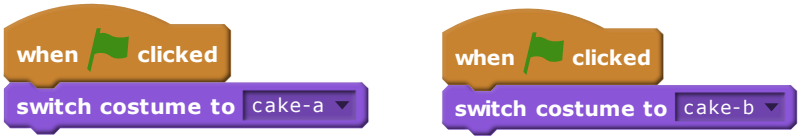

Figure 9. Race condition example

Race conditions occur if fragmented code (EFGP) exists. On the other hand, not all fragmented code leads to race conditions. An example of fragmented code that does not cause race conditions is shown in figure 10.

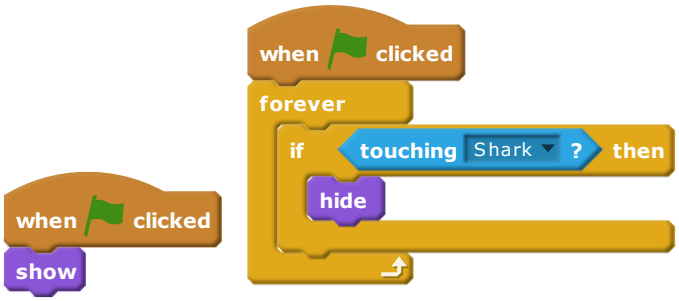

Figure 10. Fragmented code without race conditions

We counted the projects that created race conditions and we noticed that almost all the projects that featured EFGP created race conditions (Table 4).

Table 4. Number of projects with race conditions

	Projects with race conditions	EFGP Projects
2. Views	7	7
3. Interaction	1	3
4. Messages	4	4
5. Variables	0	4
6. Recap	1	0
7. New blocks	16	16
8.Clones	0	0
9. Own games	0	1

Code without effect

Students get immediate visual feedback when they program with Scratch (Maloney et al., 2010). They rapidly get used to anticipating changes in the code to reflect on the program outcome. However, they may create code that runs but has no effect. The *"hide"* block is code without effect in the example of figure 11. Since the *"show"* block is executed immediately after the *"hide"*, students do not observe the execution of the *"hide"* block. Notice that this kind of code does not fall into the dead code category. This code *is* executed but has no effect.

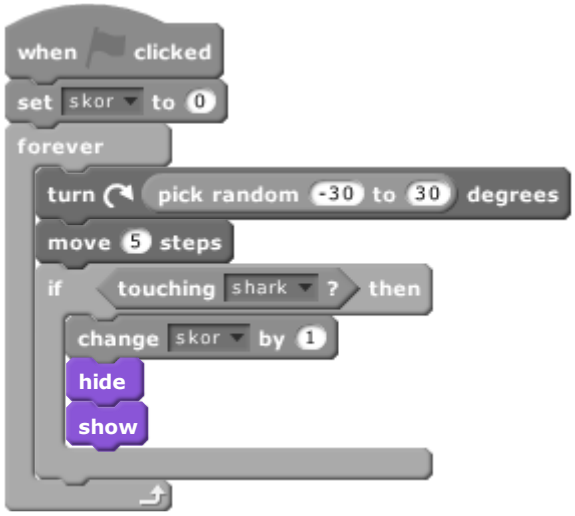

Figure 11. "Hide" block has no effect

We counted the projects that featured codes without effect and found 13 such codes. Other examples of code without effect are shown in figure 12. The *"change [ghosts] by (1)"* block has no effect. On the code on the right, the clone is deleted immediately after its creation.

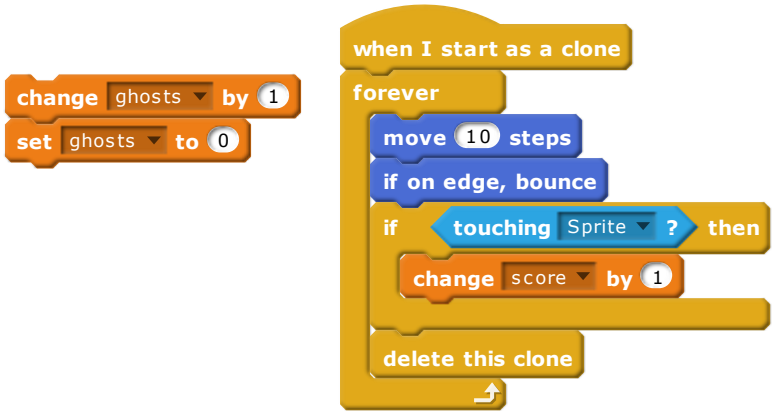

Figure 12. Code without effect

Primary school students created projects that feature dead code, problematic initialization and EFGP. Long scripts and duplicate code did not appear in a large percentage of the projects. During the study other problematic code such as race conditions and code without effect emerged.

Discussion

Following the tradition of Logo (Papert, 1980), Scratch programming language enables students to create artifacts they care about. Students use computer code to program the behaviour of their objects and when an unintended behaviour occurs, a learning opportunity emerges. In the present study, we identified such code.

We found that students leave dead code in their projects. Similar results are found in large Scratch project repositories (Hermans & Aivaloglou, 2016) and dead code is one of the bad smells detected by Dr. Scratch (Moreno-León & Robles, 2015). Dead code is not confusing in all cases. In some cases, students leave dead code for further usage and Scratch allows it by design (Resnick & Rosenbaum, 2013). But it is also possible for students to leave dead code because they have not constructed a mental model for a concept. In our study, we found that the "New blocks" concept led to increased dead code. We noticed that besides code that is not executed, i.e. dead code, students created code that was executed but had no effect. We feel that this type of code may puzzle students since it sabotages the immediate feedback that Scratch provides by design (Maloney et al., 2010).

Initialization problems are found in large Scratch project repositories (Aivaloglu & Hermans, 2016). Initialization is a point of concern in teaching programming either using professional languages, Logo (Lee & Lehrer, 1988) or Scratch (Franklin et al.,2016). In the present study, we found that initialization problems exist when students use Scratch in the classroom.

Most Scratch users make use of concurrent execution of stacks of code (Maloney et al., 2008). When this feature is overused, Extremely Fine-Grained Programming (EFGP) appears (Meerbaum-Salant et al., 2011). We found that EFGP occurs when students learn Scratch in the classroom. Furthermore, we noticed that race conditions appear along with EFGP. Race conditions can be confusing for students since they produce unpredictable results.

We did not find many projects with long scripts or with duplicate code in our code base, though students have been found to develop such code (Meerbaum-Salant et al., 2011). One possible explanation for this is the small size of students' projects. It is possible that both these types of code would appear if students created more complex projects. Code duplication was one of the code smells that was detected in higher numbers in large projects using Dr. Scratch (Vargas-Alba et al., 2019).

Conclusions

Discussing Scratch code is not about bad practices, professional habits, software maintenance or performance. Discussing Scratch code is about what students may find counterintuitive and what learning opportunities appear in the process of creating a project. While Scratch is taught in the classroom, students tinker and try out code in a constructionist approach. But, even with a programming language that is designed for novices, the produced code can be confusing.

Teachers may use the code that was described in this study as grounds for fruitful discussion with the student, ideally when the student stumbles upon it by themselves like our students did. It is possible that manifestations of such code are the result of the student's understanding of Scratch code, further research is needed to provide insights about this connection.

References

Aivaloglou, E., & Hermans, F. (2016). *How Kids Code and How We Know: An Exploratory Study on the Scratch Repository*. *Proceedings of the 2016 ACM Conference on International Computing Education Research* (ICER '16). pp. 53–61.

Boe, B., Dreschler, G., Barbara, U. C. S., Hill, C., Conrad, P., Barbara, U. C. S., & Franklin, D. (2013). *Hairball: Lint-inspired Static Analysis of Scratch Projects*. In *Proceeding of the 44th ACM technical symposium on Computer science education* (SIGCSE '13). pp. 215-220.

Crook, S. (2010). *Embedding Scratch in the Classroom*. International Journal of Learning and Media, 1(4). pp. 17–21.

Brennan, K., & Resnick, M. (2012). *New frameworks for studying and assessing the development of computational thinking*. Annual American Educational Research Association Meeting. pp. 1–25.

Franklin, D., Hill, C., Dwyer, H. A., Hansen, A. K., Iveland, A., & Harlow, D. B. (2016). Initialization in Scratch: Seeking Knowledge Transfer. SIGCSE '16. pp. 217–222.

Hermans, F., & Aivaloglou, E. (2016). *Do code smells hamper novice programming? A controlled experiment on Scratch programs*. IEEE International Conference on Program Comprehension, 2016. pp. 1–10.

Kafai, Y. (1995). *Minds in play*. New Jersey: Lawrence Erlbaum.

Lee, O., & Lehrer, R. (1988). *Conjectures concerning the origins of misconceptions in LOGO.* Journal of Educational Computing Research, 4(1), pp. 87-105.

Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). *Programming by choice: urban youth learning programming with Scratch*. ACM SIGCSE Bulletin, 40(1), pp. 367-371.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). *The Scratch Programming Language and Environment.* ACM Transactions on Computing Education, 10(4), pp. 1–15.

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

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2013). *Learning computer science concepts with Scratch*. Computer Science Education, 23, pp. 239–264.

Moreno-León, J., & Robles, G. (2015). *Analyze your Scratch projects with Dr. Scratch and assess your computational thinking skills*. In 2015 *Scratch conference*. pp. 12-15.

Netzer, R. H., & Miller, B. P. (1992). *What are race conditions? Some issues and formalizations*. *ACM Letters on Programming Languages and Systems (LOPLAS) 1*(1). pp. 74-88.

Papert, S. (1980). Mindstorms: Mindstorms: Children, Computers, and Powerful Ideas. Basic books: NY.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... & Kafai, Y. B. (2009). *Scratch: Programming for all*. *Commun. Acm*, *52*(11), pp. 60-67.

Resnick, M., & Rosenbaum, E. (2013). Designing for tinkerability. *Design, make, play: Growing the next generation of STEM innovators*, pp. 163-181.

Vargas-Alba, Á., Troiano, G.M., Chen, Q., Harteveld, C., & Robles, G. (2019). *Bad Smells in Scratch Projects: A Preliminary Analysis.* In Proceedings of the 2nd Systems of Assessments for Computational Thinking Learning Workshop (TACKLE@EC-TEL 2019).