The Five Million Piece Puzzle: Finding Answers in 500,000 Snap*!*-Projects.

Sven Jatzlau, Stefan Seegerer
Computing Education Research Group
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
{sven.jatzlau, stefan.seegerer}@fau.de

Ralf Romeike
Computing Education Research Group
Freie Universität Berlin
Berlin, Germany
ralf.romeike@fu-berlin.de

***Abstract*—When learners use programming environments and languages or interact with them to construct their own knowledge, they create artifacts. These artifacts are a great source of insights into how learners use programming environments. Better knowledge of how learners use programming environments help us develop the theory of teaching programming languages. To this end, we analyzed a sample of 550,000 Snap*!*-projects, containing almost five million scripts, making this the first automated analysis of Snap*!*-projects of this magnitude. The majority of our initial results are in line with analyses of Scratch-projects of comparable scale: code smells such as standard names in sprites are found both in the Scratch-community and in these projects, and the complexity of projects in both languages is comparable. We also evaluated projects with regards to other factors, such as sprite naming, and the usage of blocks across the different categories. From our results, we can gain further insights into how learners use block-based programming environments, which ultimately assists us in developing teaching strategies for block-based languages.**

***Index Terms*—Snap!, education, block-based, programming, project analysis**

I. Introduction

In recent years, block-based programming languages have become more and more popular. Particularly introductory courses for learners are increasingly being taught using these languages, making them see widespread use in schools all over the world. Block-based languages embody several vital design principles that enable a constructionist learning theory: low floors (low entry barrier, no prior knowledge required), wide walls (wide range of possible applications and projects), and high ceilings (enabling the creation of complex programs) [9]. Snap*!*, in particular, was created to enable more complex projects than Scratch. To this end, it features many high-level concepts, and is described to have *no* ceiling rather than a *high* ceiling [3].

As our goal is the continuous development of the theory of teaching programming, we need to get a precise understanding of how learners use block-based languages, and what they produce with them; As pointed out by Aivaloglou [1], the artifacts users generate while programming - the saved and shared programs - present an opportunity to learn more about the way programming environments are used.

To reach this goal, we performed a static code analysis on a sample of 555,822 Snap!-projects that were rendered ownerless during a migration of the Snap!-cloud. There have been previous studies that performed analyses of similar scale on Scratch projects. For this reason, it seems appropriate to verify whether certain patterns and insights gained in these Scratch-related studies can be transferred onto Snap*!*. Specifically, we are interested in whether users of Snap*!* have outgrown the mistakes they made in Scratch. We also want to investigate whether they use the same concepts. Finally, since Snap*!* aims to enable more complex projects than Scratch, we want to see if this is reflected in projects created by the users. The research questions this preliminary study aims to address are, therefore, as follows:

- RQ1: How are code smells different in Snap*!*-projects, compared to Scratch-projects?
- RQ2.1: Do Snap*!*-projects differ from Scratch-projects with regards to concepts used?
- RQ2.2: Do users implement those concepts that differentiate Snap*!* from Scratch, such as nested objects [4]?
- RQ3: How does the complexity of Snap*!*-projects compare to that of Scratch projects?

II. Related Work

A. Tools

Several analyses on Scratch projects have been done using either Hairball [2], or Dr. Scratch [7] (based on Hairball).

Hairball is a static analysis tool for Scratch projects. Students can use it to find potential errors, while educators can use it to help assess learning processes. It therefore scans project files for initialization, synchronization between say-blocks and sound, synchronization between broadcasting and receiving, and use of timing and loops. Hairball has significantly lower mislabel and false positive rates than manual analysis [2].

Dr. Scratch represents an evolution of Hairball aimed at monitoring and assessing users' Computational Thinking skills and detecting bad programming habits. The program therefore relates usage of concepts to their understanding. The user receives feedback on their programming skills and is pointed to areas they can still improve in [6], [7].

B. Studies

Moreno and Robles conducted an analysis of 100 projects published on the Scratch community website. The aim of this study was to investigate whether certain issues can be found in Scratch projects, which would suggest that users acquire habits that go against basic programming recommendations. Specifically, these bad habits were objects with standard names (*Sprite1*, *Sprite2*, etc.) and duplicate code, two common "code smells". Their results show that 79% of the analyzed projects contain sprites with standard names, and 62% contain at least one repeated section of a script. These findings hint that Scratch users may find it difficult to use abstraction and modularization to structure their projects in a suitable way [5].

In order to find out "What do people do when they use Scratch?", Aivaloglou and Hermans expanded upon previous studies, not only analyzing projects for code smells, but also other factors such as complexity, abstractions and programming concepts. For their automated evaluation, the Scratch public repository was scraped, and around 250,000 projects were analyzed. Results of this study include the fact that most Scratch programs are of low complexity, and in addition to the code smells outlined in [5], Scratch projects also commonly feature mismatched broadcast signals [1].

Papavlasopoulou et al. evaluated the use of programming concepts in Scratch-projects. In contrast to other studies, however, this evaluation was done manually. The reason for this may be the smaller sample size compared to other studies (twenty-seven participants, nine projects in total). Most of the projects contain event handling, sequences and conditionals. The second most used concept was, surprisingly, threads (parallel execution) and operators. At the same time, however, synchronization is a much rarer occurrence: it was only found in two out of the nine games [8].

These studies of Scratch-projects can be used as a guide for the analysis of Snap!-projects. This allows for the generation of comparable results that can be aligned to existing research.

III. Methodology

Our evaluation was based on a static analysis of the program code generated when a project file is saved. In a two-step process, we first parsed the code of all project files in Python, and then performed quantitative analyses in R.

In total, the number of projects we analyzed is 555,822, excluding the 1571 projects (roughly 0.28%) that were unreadable due to a corrupted save state, or other factors. These projects were provided to us by the team of UC Berkeley, after they could not be attributed to an owner following a migration of the Snap!-cloud. They show a high level of heterogeneity: they are a mixture of games, animations, exercises, apps, presentations, turtle art, both finished and unfinished. Their origin also means that these projects are not published, or shared, which presents a stark contrast to similar analyses. What this difference implies will be outlined further in section 5. In our analysis, we evaluated sprite naming and standard names, like in [5]. We also check for usage of blocks and their respective categories, like in [1], and programming concepts, like in [8]. We investigate code smells and analyze project complexity according to McCabe, like in [1]. We also checked for several factors that comparable studies did not cover.

TABLE I
Percentages of projects that feature at least one Sprite with the respective names

Name	Percentage	Name	Percentage
Stage	99.2%	Ball	1.8%
Sprite	83.2%	Player	0.6%
Sprite(2)	21.9%	Platform	0.57%
Sprite(3)	12.4%	Coin	0.55%
Sprite(>3)	25.3%	Enemy	0.4%

TABLE II
Project characteristics and frequencies

Standard names	66.2%
Large number of sprites (> 20)	0.7%
Low number of sprites (< 3)	75.5%
Large sprites (> 51 blocks)	18.9%
Large scripts (> 13 blocks)	43.8%
Dead code	61.8%

Among these factors are variables, unique events, user-defined blocks, and object nesting.

IV. Results

a) RQ1: How are code smells different in Snap*!*-projects, compared to Scratch-projects?

To investigate this question, we initially evaluated the way sprites were named across the projects. After determining the most common names for sprites, we investigated relative percentages of projects that use these names. The results are shown in I (capitalization was ignored).

It is extremely apparent that most users do not rename the stage, or the initial sprite that is created on project creation. However, users do not seem to rename sprites in general. This phenomenon is a code smell called "standard names". More code smells are listed in II. We calculated the thresholds for large scripts and large sprites based on the top 10%. These thresholds are 13 blocks and 51 blocks, respectively. As for the most common sprite names, users typically seem to give names commonly found in games, such as *ball*, *player*, *platform*, *coin*, and *enemy*.

b) RQ2.1: Do Snap*!*-projects differ from Scratch-projects with regards to concepts used?

Table III shows the ten most used blocks across all 4,943,594 scripts, and the number of projects that feature at

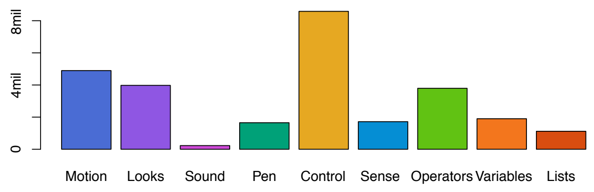

Fig. 1. Usage of blocks by category

TABLE III
THE 10 MOST USED BLOCKS AND THEIR FREQUENCY IN PROJECTS

Block	% of blocks	% of projects	Block	% of blocks	% of projects
if	5.6%	34.9%	say Hello! for 2 secs	3.8%	35.2%
set ▾ to 0	4.9%	39.6%	when ⚑ clicked	3.5%	59.3%
go to x: 0 y: 0	4.3%	48.5%	wait 1 secs	3.4%	28.1%
=	4.1%	27.9%	move 10 steps	3.4%	32.8%
when I receive ▾	3.9%	18.3%	when ▾ key pressed	3.0%	33.5%

TABLE IV
NUMBER OF UNIQUE EVENTS USED IN PERCENTAGE OF PROJECTS

Unique events	Percentage	Unique events	Percentage
1	58.2%	4	1.4%
2	31.8%	5	<0.1%
3	8.2%	6	<0.1%

least a single block of that type. Notably, the two most common blocks are not blocks one would typically expect novices to use - conditional statements and variables, respectively. On the other hand, Snap*!* is aimed at more advanced users who may have moved beyond Scratch. The blocks may also be needed for the projects: as indicated by the names of sprites in I, a number of Snap*!*-projects may be games, where these blocks would be expected. Lastly, control-related blocks seem to be very prominent across all projects. This dominance of control-related blocks is even more evident in figure 1, which shows the distribution of used blocks by category.

We analyzed the number of unique events users included in their projects. Snap*!* supports six types: (a) when green flag is clicked, (b) when a key is pressed, (c) when a message is received, (d) when an object is clicked, (e) when a condition becomes true, and (f) when a clone is created. Table IV shows the results of this analysis. We see that the vast majority of projects use three unique events or less. Based on the frequency of block types in table III, we can assume that these three events are (a), (b), and (c), and that event types (d), (e), and (f) are not nearly as common.

Lastly, we analyzed the usage of variables. Snap*!* supports three types of variables with varying degrees of visibility (scope): (1) global variables visible to every object, (2) object-/sprite-local variables visible only to that particular object, and (3) script-local variables, only visible within the script they are created in. The results are shown in table V.

We can see that variables are a very common concept that is used across a majority of projects, as evidenced by the set variable-block's popularity, and the usage of global variables. However, when it comes to scope and data encapsulation, users tend to opt for the highest possible scope (global). Notably, script-local variables are used more commonly than sprite-local variables.

TABLE V
PERCENTAGE OF PROJECTS USING DIFFERENT NUMBERS OF VARIABLES

At least one global variable	45.3%
More than two global variables	23.7%
More than four global variables	12.2%
At least one object-local variable	4.1%
At least one script-local variables	5.9%

TABLE VI
PERCENTAGE OF PROJECTS USING ADVANCED CONCEPTS OF SNAP*!*

At least one nested object	0.1%
At least one JavaScript-block	<0.1%
At least one global user-defined block	52.3%
More than two global user-defined blocks	35%
More than four global user-defined blocks	23%
At least one sprite-local user-defined block	1.5%

c) RQ2.2: : Do users implement those concepts that differentiate Snap*!* from Scratch, such as nested objects?

There are several concepts that differentiate Snap*!* from Scratch, such as object nesting [4], the ability to create user-defined blocks globally or object-locally (of course, Scratch supports this feature now, however, we deem it an advanced concept nonetheless), and the ability to incorporate JavaScript-blocks into the code. Table VI shows the results of our analysis.

Object nesting is rarely used. A significant number of those projects that feature nesting are copies or remixes of the Snap*!*-example used to illustrate the concept, called "swimmer". Out of the total number of scripts across all projects (nearly 5 million), less than 600 used JavaScript-blocks, indicating that users do not seem to make much use of JavaScript functions. User-defined blocks on the other hand, are very popular with the Snap*!*-community: 52,3% of all projects use at least one such block, 35% use more than two, and 23% of all projects even use more than four. The official tools-library was excluded from this number. It does not exist in newer version of Snap*!*, but including it in the results would have falsified the results. User-defined blocks limited to single objects/sprites are used sparsely, however, only being featured in 1.5% of all projects.

d) RQ3: : How does the complexity of Snap*!*-projects compare to that of Scratch projects?

Our third research question aims to compare the complexity of Snap*!*-projects when compared to Scratch-projects. Snap*!* is considered a more complex brother/sister of Scratch - is this reflected in the projects? The results are visualized in figure 2.

There are three noteworthy brackets: the overwhelming majority of scripts (86%) has a complexity of 1, containing no decision point (if, or if/else). 5.8% of scripts have a cyclomatic complexity of 2, with exactly one decision point. 5.2% of scripts have a complexity of 3 or 4. All higher complexities are spread out over only 3% of all scripts.

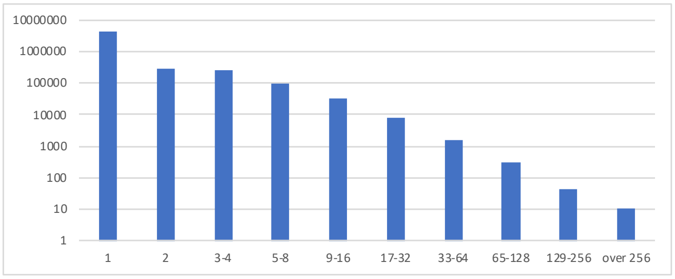

Fig. 2. Cyclomatic complexity of scripts (logarithmic scale)

V. Discussion

A. How do the results compare to results of similar studies?

a) Code smells: Regarding the code smell *standard names*, most users do not rename the initial sprite created by Snap*!* (85.3% of projects), and the majority of projects (upwards of 66%) features sprites with standard names. In a similar study, 79% of Scratch-projects were found to contain at least one non-renamed "SpriteX"-object [5]. For the two code smells *large scripts* and *large sprites*, they are featured at least once across all 43.8% and 18.9% of projects. These numbers are comparatively higher than those from a similar study on Scratch-projects [1], so visual clutter in these projects is higher, causing difficulties for users to understand program flow. For dead code, we found that 61.8% of projects had at least one script that was unconnected to an event block, or only consisted of an event-block. This number is significantly higher than the corresponding number in a comparable Scratch study [1].Overall, more Snap*!*-projects exhibit unconnected scripts, which could indicate that users program differently in Snap*!*, compared to Scratch.

b) Blocks and categories: Although a direct comparison to Scratch is difficult, as the blocks are arranged into slightly different categories, some similarities can be seen between our results and comparable studies [1]: the sound, pen and sensing categories are the least-used. Control blocks, such as events, conditionals and loops are the most commonly-used blocks. Notably, the looks-category seems to be more popular in Scratch than in Snap*!*.

c) Complexity of projects: Finally, we compare the McCabe cyclomatic complexities of projects. Our hypothesis is that Snap*!* allows for more complex programming tasks, which could suggest that Snap*!*-projects are indeed more complex than Scratch-projects. However, the graphs show few differences. In order to get a better understanding of these differences, we need to consider the limitations of this study.

B. Limitations

There are three connected factors that significantly influence the statements we can make based on our results.

Firstly, we do not know where the projects come from. Projects that came from curricula are typically more guided, i. e. what the user used may not be representative of what they would use without guidance, or what they understood.

This is reinforced by the second factor: we do not know the intended purpose of each project. If a project is meant to greet the user and show an animation, not using any variables does not mean the user has not understood variables. It simply means they may not be needed for the purpose of the project.

Finally, we do not know the status of the projects. As mentioned before, the projects are not published or shared (suggesting that they are intended to be seen by others, i. e. in a presentable state), as is the case with similar studies that scraped the Scratch *public repository*. Here, however, some projects may be finished, others may be in their initial stages, while others can be full of errors and bugs - possibly also intentionally.

C. Implications

As this study is only in its initial state, it is too early to draw generalized implications for educators. Nonetheless, we can outline several noticeable factors that could be built upon in the development of future studies and teaching approaches.

It may be beneficial to refine the definition of code smells for block-based programming languages. In related works, dead code has been frequently named as a common code smell, typically being identified by scripts that lack a hat-block. We argue that hat-less scripts, which are found in 56.3% of all projects, can also be a sign of users using direct manipulation, or direct drive, to manually execute scripts and blocks. This technique can be employed during project development to facilitate testing and debugging.

Overall, the results underline the necessity of finding didactic approaches to teaching block-based languages and the concepts we find in them – there is a lack of teaching methods for some of the most commonly-used blocks, such as events and the visual output on the stage.

VI. Conclusion

This paper documented the first automated large-scale study of Snap*!*-projects. We presented the insights we gained from an analysis of almost five million scripts spread out across over 550,000 projects. We analyzed the projects for sprite names, numbers of sprites and scripts, code smells, usage of blocks and categories, usage of programming concepts such as variables and user-defined blocks, and complexity.

Our results align with those presented in similar related studies, but show some interesting differences. We hypothesize that these differences originate from the nature of the projects themselves: unlike studies that were performed on the Scratch community website, these projects were never published or shared. For that reason, it seems likely that many of these projects are unfinished and incomplete.

With the new Snap! community website, it could be an interesting experiment to scrape a sufficiently large number of published/shared projects and perform a comparative analysis to ours. Such a study would allow for better contextualization of the results we found. As of the time of writing, however, the community website does not host enough projects to create a significant sample.

Acknowledgment

We would like to thank the team of UC Berkeley and the Snap*!*-team for providing access to these projects and making this study possible.

References

[1] Efthimia Aivaloglou and Felienne Hermans. How kids code and how we know: An exploratory study on the scratch repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pages 53–61. ACM, 2016.

[2] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. Hairball: Lint-inspired static analysis of scratch projects. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 215–220. ACM, 2013.

[3] Brian Harvey and Jens Mönig. Bringing no ceiling to scratch: Can one language serve kids and computer scientists. *Proc. Constructionism*, pages 1–10, 2010.

[4] Sven Jatzlau and Ralf Romeike. How High is the Ceiling? Applying Core Concepts of Block-based Languages to Extend Programming Environments. In Egl Jasut Valentina Dagien, editor, *Constructionism 2018: Constructionism, Computational Thinking and Educational Innovation: conference proceedings*, pages 286–294, 2018.

[5] Jesús Moreno and Gregorio Robles. Automatic detection of bad programming habits in scratch: A preliminary study. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–4. IEEE, 2014.

[6] Jesús Moreno-León, Gregorio Robles, et al. Analyze your scratch projects with dr. scratch and assess your computational thinking skills. In *Scratch conference*, pages 12–15, 2015.

[7] Jesús Moreno-León, Gregorio Robles, et al. Dr. scratch: a web tool to automatically evaluate scratch projects. In *WiPSCE*, pages 132–133, 2015.

[8] Sofia Papavlasopoulou, Michail N Giannakos, and Letizia Jaccheri. Discovering children's competences in coding through the analysis of scratch projects. In *2018 IEEE Global Engineering Education Conference (EDUCON)*, pages 1127–1133. IEEE, 2018.

[9] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S Silver, Brian Silverman, et al. Scratch: Programming for all. *Commun. Acm*, 52(11):60–67, 2009.