Analysis of Scratch Projects of an Introductory Programming Course for Primary School Students

Alexandra Funke, Katharina Geldreich, Peter Hubwieser
Technical University of Munich
TUM School of Education
Munich, Germany
{alexandra.funke, katharina.geldreich, peter.hubwieser}@tum.de

***Abstract*—Computer Science (CS) is increasingly entering the early levels of early childhood education, like primary school or even kindergarten. Therefore, it becomes more and more important to gain insight into which teaching methods and content would be appropriate for young students of primary levels. To investigate this, we have designed a specific three-day introductory programming course for 4th grade students (ages 9 - 10), which was taught four times up to now. Fifty-eight children (26 girls and 32 boys) participated in the courses from May to August 2016. At the end of the courses, the children have developed 127 Scratch projects during the course. The methodology and the results of the qualitative analysis are described in this paper. We discovered that the students created three different types of programs in particular: Stories, Animations, and Games. The level of understanding of the students, who programmed a Game, was mostly found to be advanced. Stories, on the other hand, reached only the two basic levels. Most of the students met the requirements we had set for the projects.**

Keywords—computer science education; primary school; primary education; scratch; programming

I. INTRODUCTION

Computer Science has to overcome several challenges. Schools and universities are confronted with different misconceptions and prejudices towards CS [6] which manifest themselves from an early age [8]. In order to prevent students from developing negative attitudes, one approach is to introduce Computer Science concepts like programming already in primary school to provide opportunities for making experiences with technology and CS. Children should learn that they can use computers not only as users but also as creators [2]. This role change combined with a fun experience of programming could increase their self-confidence towards CS in particular and technology in general [9]. At the same time, the discussion about the necessity of computer science and programming in childhood education is growing steadily [9]. While several countries have already introduced CS in their primary school curricula (e.g. the UK [12] and Australia [13]), Germany has not yet developed mandatory guidelines for how to deal with the new topics.

To find out which teaching methods and content would be appropriate for German primary schools, we designed an introductory programming course for fourth graders. Although a predominant goal of the course was to change the students' attitudes towards CS, we also wanted to gain insight into how profoundly they can learn several programming concepts. According to this, two of our research questions were

- What are the learning outcomes of the programming course?
- Which programming concepts do children use in their programming projects?

This work is structured as follows: First, we discuss some background work regarding Computer Science courses for primary school students, especially programming courses. In addition, we provide a short overview of the design of our programming course. This is followed by a description of the methodology and analysis. In order to illustrate the qualitative analysis of the programming results, we present three project examples from our courses. Then, the results are presented with a discussion of the findings. The paper closes with a summary of the findings and an outlook for further work.

II. RELATED WORK AND THEORETICAL BACKGROUND

In recent years, a variety of courses have been developed to expose children to Computer Science concepts. Amongst others, various courses set their focus on programming. Tsan et al. [1] implemented an in-school computer science course for 5th grade students. They co-designed it with a primary school teacher who had prior knowledge in technology but no general CS background. The course was taught in 30-hours during a regular school year. In order to analyze the effectiveness of this collaboratively developed curriculum, the researchers collected data with interviews before and after the course, made videos of the students working, made screen recordings and collected student-created materials like short essays and storyboards. During the class, the students completed two programming projects with Scratch and almost all of them worked in pairs (18 pairs in total). The authors found that female-female pairs worked very well together and were creative. However, their products missed many requirements. For example, a male-male pair had a keen interest in and enthusiasm for programming. They tried out many things on their own and had a similar line of thinking. Nevertheless, the boys' products received low scores for usability and consistency. In their summary, the authors listed critical observations for future courses. One key finding from this research is the usefulness of supportive, collaborative work.

Programming is also an important part of the CS courses in the work of Duncan and Bell [20]. They described an example

https://doi.org/10.1109/EDUCON.2017.7943005

of a computer science course for primary school students. The authors implemented a CS class, which took place an hour a week during a school year. More than 600 students aged between 5 and 12 were taught during the study. The main goals of the course were that the students engage with the presented content and enjoy the classes. Furthermore, rather than learning to apply any specific programming language, the students were to become familiar with the basic principles of programming. During these courses, the students worked with the programming language Scratch and had to solve two main programming quizzes at the end. After analyzing the results, the researchers found no statistically significant difference between the average achievements of the participating girls and boys. However, girls were not as good as boys at predicting their capabilities. Despite doing as well as the boys, they did not seem to be aware of this. Girls needed more encouragement to try out new activities while the boys often followed a more experimental approach and tried out more on their own. In summary, most of the kids enjoyed using Scratch and wanted to continue learning about CS and programming.

To get insight into the learning outcomes of the students, the programming results have to be evaluated based on objective criteria. The authors of [19] carried out a general evaluation of programming results within a programming course for middle school students. The aim of this course was to help them develop computational thinking by programming computer games. In [19] they described the students' programming strategies and analyzed the games they developed. During one course with girls only, 108 games were developed using the software Stagecast Creator. Each game was coded within three main categories: *Programming*, *Documenting & Understanding Code*, and *Designing for Usability*, as well as within 24 subcategories. The authors described their coding system with the claim that it "can be adapted for use with any of the programming environments targeted at younger populations." Following this advice, several other research groups used this category system as a basis for coding programming results. An analysis of programming projects made by primary school students is presented in [11]. The authors developed a coding scheme to evaluate games made by children at the primary level using Scratch to show which programming skills they used. During the course with 60 participants, small groups of children developed 29 games. For coding the programming results, the authors refined the category system described in this section and extended it with the programming concepts that could be learned with Scratch [16]. Each game was then coded with the refined coding system, which consists of three main categories (*Programming Concepts*, *Code Organization* and *Designing for Usability*) and 22 subcategories.

Another approach to classify programming results is described in [9]. The authors of this work used SOLO (Structure of the Observed Learning Outcome) taxonomy [14] for developing a specific code system for programming projects. According to the five levels of increasing complexity of programs, Lister et al. defined the categories. Level 1 *Prestructural* is used when the programming code substantially lacks knowledge of programming constructs or is unrelated to the question. If the code represents a direct translation of the specification, it is coded as *Unistructural* (L2). For *Multistructural* (L3) the code has to represent a translation that is close to direct. In this level, the programming code may have been reordered to make a more integrated, valid solution. If the code provides a well-structured program that removes all redundancy and has a clear, logical structure, it is coded with *Relational* (L4). The highest level *Extended Abstract* (L5) can be reached if the code uses constructs beyond those required in the exercise to provide an improved solution.

In [10] this extended SOLO code system is used to classify the Scratch programs of primary school students. They provided specific examples for the first three levels adapted to an exercise during their courses. The authors found that the students' understanding varied between schools and depended, among other things, on the students' performance in school. Though some students reached level 4 of the system, they struggled to synchronize costume changes within conversations of their sprites.

III. DESIGN OF THE COURSE

We developed a three-day course for students of the fourth grade of primary school. As context, we chose "Circus" for all tasks and materials, out of three reasons. First, we regarded this as an attractive field of their personal experience. Second, a circus offers a variety of interesting tasks to simulate the actions of animals or human beings. Third, we hoped to attract both girls and boys equally by this metaphor. On each day, we spent four hours with the kids. Day by day, the students are exposed to a more and more detailed picture of programming. At the end, we expected that the children had learned the basic principles of programming, in particular to work with the programming environment as well as to apply and combine algorithmic control structures.

Day 1 Most of the students did not have any previous knowledge of programming or computer science. Therefore, the goal of the first day is to give them a basic idea of how a computer program works. They were to realize that programs execute a particular task by following precise and clear instructions. Because we did not want to overstrain the students, we decided to introduce the basic algorithmic concepts "unplugged" [5], before any programming. Hence we use social activities and group problem solving at day one, without actually working on computers. In order to learn how to split tasks into

Fig. 1. Programming with haptic blocks

Fig. 2. Simulating the execution of a program

smaller parts, we provided a variety of short exercises in which different activities had to be transformed according to unambiguous instructions. Afterward, the groups had to work together to solve a more complex task. To take up the circus theme, we let the students program each other, solving tasks like searching for missing items or animals in a circus tent. To represent the solutions, we use haptic (printed and shrink-wrapped) Scratch-like programming blocks to prepare "real" computer programming on day two (Fig. 1, 2).

Day 2 The goal of the second day is to enable the students to create simple Scratch programs that produce Multimedia output. To provide a child-friendly programming environment and to spare the students any unnecessary syntactical overhead, we decided to use the block-based language Scratch [7]. We created a learning circle with increasingly difficult stations, which introduces the core elements of Scratch one by one, leading from simple sequences to control structures as loops and conditional statements. In each part of the circle, the students have to solve tasks that are presented on out-handed instruction sheets. To support the students' expected variety in knowledge and learning pace, we prepared additional tasks as well as helpful tips.

Day 3 On the third day, we wanted to find out what the students had learned and if they could solve more open tasks. In addition, we wanted to stimulate the children to work creatively and in a self-directed fashion. Yet, to compare the outcomes, we set the following "mandatory" requirements for the students' projects. The programs should a) work on more than one sprite b) move the sprites during execution c) comprise at least one iteration and d) include at least one conditional statement. After meeting these requirements, the students should continue their programming work without any further guidelines. They were free to experiment with Scratch, to invent their own circus stories and implement these. At the end of day three, all programs were presented in front of the course, and the students had the opportunity to comment their project.

IV. METHODOLOGY

All courses were taught in German by a female, formally educated primary school teacher, being a CS researcher, too. Additionally, a male computer scientist assisted during the courses.

In May 2016, a pilot study was carried out at a student research center led by our university. After making improvements of the tasks, we conducted the course with fourth grade classes in June and July. The course was held in our department, where we could provide a consistent setting and stable technical equipment. Another course followed in August at the student research center. All sessions were videotaped, enabling us to analyze all actions and interactions of the students. At the beginning and at the end of each course day, we conducted group interviews with the classes. In order to get an idea of the students' prior knowledge, as well as their mental image of programming, we used a variety of interviewing and reflection methods. In addition, we captured the screens of the student computers during the programming exercises on days two and three, to get an image of the students' working methods. To collect the students' Scratch programs, we saved the projects after the course had ended.

One goal of the study was to examine the structure and quality of the programs the children created. We also wanted to find out which programming concepts they applied. In order to rate the quality of the projects, we developed a specific category system, by combining, adapting and extending different systems that were presented and explained in Section II. On the top-level, we distinguish between the four main categories: *A. Requirements*, *B. Programming Concepts*, *C. Code Organization* and *D. Operability* that are described and differentiated in the following sections.

A. Requirements

As described in Section III, the student programs should meet four "mandatory" requirements:

- Several Sprites: Counting all sprites to figure out if more than one sprite is used.
- Sprite Motion: Sprites should move during program execution. For this, all blocks of the type *Motion* were counted to match them with this category.
- Iteration: An iteration is included, when the students used *forever* or *repeat* blocks.
- Conditional Statement: If a program contains at least one *if _ then* or *if _ then else* block to check for conditions, we code this as a conditional statement.

B. Programming Concepts

The basic *Programming Concepts* of the Scratch programs were coded as follows:

- Sequence: A correct sequence is present when the program runs in a systematic order. For example, some students did not pay attention to how the program would be executed. When they created a conversation, sometimes all scripts ran at the same time by mistake.
- Variables: Variables can be created within Scratch and then be used within programs. To code this category, the variable blocks of *Data* are counted and matched.

- Lists: Similar to Variables, the list blocks of *Data* are counted and matched with this category.
- Event Handling: Responding to events triggered by either the user or another script. All blocks of the type *Event* were counted to match them with this category.
- Threads: If two or more scripts were going to execute at the same time, we coded this as existent threads.
- Coordination and Synchronization: Checking the program for all *wait _ secs*, *wait until*, *when I receive*, *broadcast* and *broadcast _ and wait* blocks, for coordinating the actions of multiple sprites.
- Keyboard Input: Includes the program an *ask _ and wait* block, and it provides a keyboard input for users.
- Random Numbers: A random number exists when a *pick random to* block could be found in the project.
- Boolean Logic: *And, or* and *not* were coded as boolean logic.
- Dynamic Interaction: The usage of *mouse x*, *mouse y* or *loudness* is used for dynamic interaction.

C. Code Organization

The organization of the code was categorized by three codes:

- Extraneous Blocks: A block is extraneous if it has no connection to a script with a starting event.
- Sprite Names: This category codes whether the original sprite name is overridden or not.
- Variable Names: This category codes whether the name of a variable is meaningful (e.g. "timer") or not.

D. Operability

Rating of interaction and functionality:

- Functionality: Whether or not the project performs correctly when it is started is coded as functionality.
- Sprite Customization: A sprite is customized, if a predefined sprite is adjusted (e.g. removing an arm) or if the student created their own sprite (e.g. drawing it).
- Stage Customization: A stage is customized, if a predefined stage is adjusted (e.g. add a drawing) or if the student created their own sprite (e.g. drawing it).
- Interactivity: A highly interactive program provides the user with opportunities to interact with it (e.g. key control). In contrast, a program with no interactivity is, for example, a sequence of actions.
- Usability: The project is intuitive, if the user understands how the program runs with little or no information. Sometimes programs use unorthodox keys for controlling the program, like "G". This is only partly intuitive for users.
- Project Type: We coded the project types Animation, Game, Interactive Art, Music and Dance, Story and Video Sensing according to the project designation on the Scratch web page[1]. For example, a Story is a sequence of actions without any user input or control possibilities.

E. Levels of Understanding

Finally, we analyzed the projects for the level of understanding with a code system inspired by the works described in Section II:

- Prestructural (L1): The script contains only a few blocks. The student does not understand how to extend the script to a meaningful program.
- Unistructural (L2): The script contains sequences of actions in a simple way. Control structures are not contained or they are unrelated.
- Multistructural (L3): The script fulfills all given requirements and includes a variety of different block types. The code may have been reorganized to make a more integrated solution.
- Relational (L4): The script provides a well-structured program that removes all redundancy and has a clear, logical structure.
- Extended Abstract (L5): The script uses concepts and blocks beyond those required in the exercise to provide an improved solution.

F. Qualitative Analysis

Applying these category systems, two researchers rated 27 (21%) of the primary school student projects from our courses to assess intercoder agreement and reliability. The two raters did not know which student created which project or the gender of the students. In order to get an idea of the accord, the Brennan coefficient [21] and raw agreement were calculated. Both resulted in an almost perfect agreement (> 0.81) according to Landis [23]. The common Cohen and Krippendorff factors are not applicable, as they require a normal distribution of the coded cases over the codes [22].

After the coding process, we clustered the projects by the courses. For each cluster, we studied the results and differences between them. Three examples of Scratch programs and their results are presented in the next sections.

V. Project Examples

In order to illustrate the programs, which were developed during the last course day, we describe three of them. Because the students used Scratch in German, the projects were translated by the authors.

A. Project 1

Figure 3 shows a story about two friends attending a circus show with seven sprites. Because the theme of the sessions was *Programming Circus*, this is a typical storyline in the projects.

[1]https://scratch.mit.edu/starter_projects/

Fig. 3. Stage in project 1

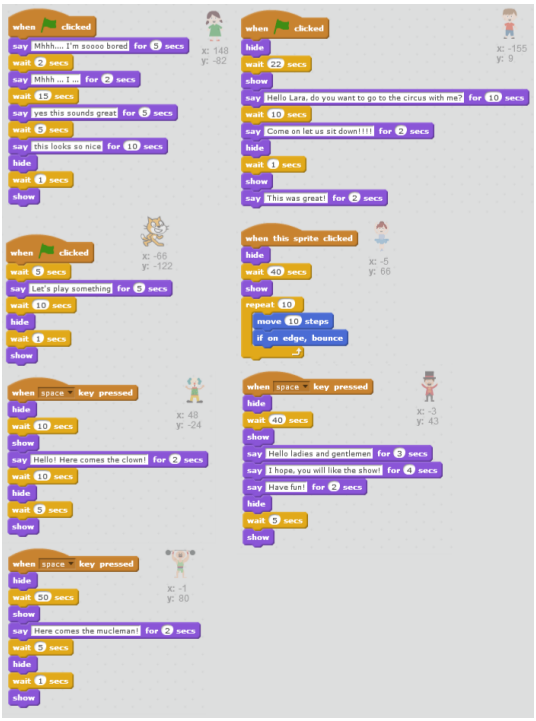

Fig. 4. Code for project 1

The student used ten different blocks, a total of 64 blocks. As shown in the code of Fig. 4, the most commonly used block type was *Look* (35 blocks). Thirteen of these blocks were *say _ for _ secs*. In order to start the scripts the student used *when green flag clicked* (3), *when _ key pressed* (3) and *when this sprite clicked* (1). Each sprite was assigned exactly one *Event* block. To structure the conversion of the story, the student used 19 *wait _ secs*.

B. *Project 2*

Screenshots of the underwater world and the related code created by a student are illustrated in Fig. 5 and 6. The child used twelve sprites to build the stage. After starting the project with 15 *when green flag clicked* blocks, the diver in the orange suit and the related bubbles move to the right until reaching the edge (*if on edge, bounce*). Using twelve *repeat* blocks, the program execution has no explicit end. Of the 60 blocks used out of four block types (*Event*, *Control*, *Motion*, *Sound*) ten *wait _ secs* were used to structure the motion. All other fish and bubble sprites,

Fig. 5. Stage in project 2

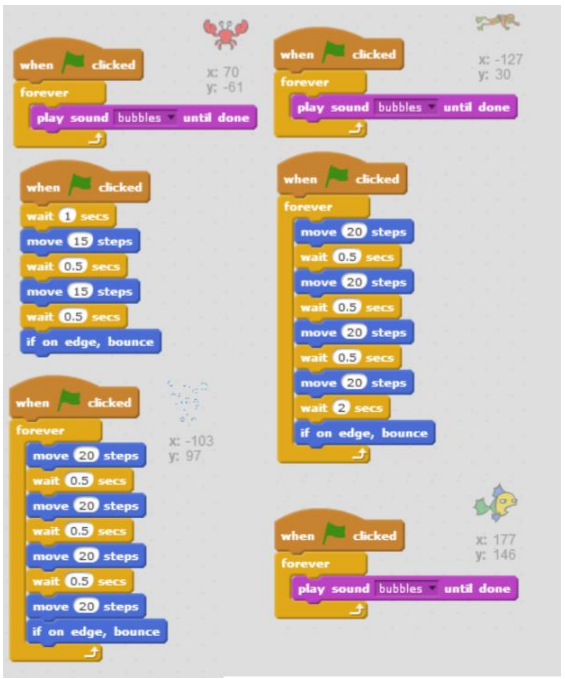

Fig. 6. Code for project 2

which are not shown in the code of Fig. 6, included the same scripts as the illustrated sprites and were left out of the figure for a better overview. Specific for this Scratch project is the heavy use of *play sound _ until done* (10) with the same sound.

C. *Project 3*

The programming result from a third student is shown in Fig. 7 and 8. A total of 41 blocks were used (13 different blocks). Out of them, 17 belong to *Motion* (eight times *move _ steps*). Even though the student only used two sprites to develop the small game, the program reacted to nine *Events* (seven times *when _ key pressed*). With six *if _ then* blocks, the student included conditional statements. The user can activate a new game round by pressing the key *m*. This broadcasts a message that causes that the sprite to move to the start.

VI. Results

Overall, 58 students took part in the sessions from May to August 2016 (26 female and 32 male students). Two boys deleted their programs before we could save them and two other boys worked together as a team. For this reason, we were only

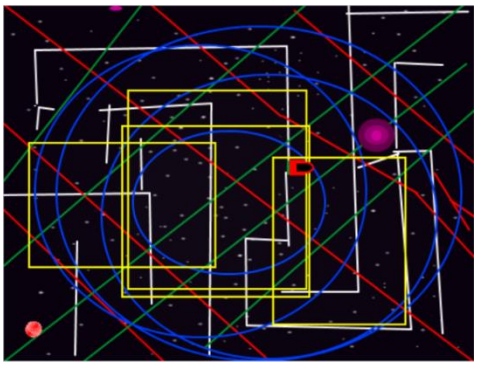

Fig. 7. Stage in project 3

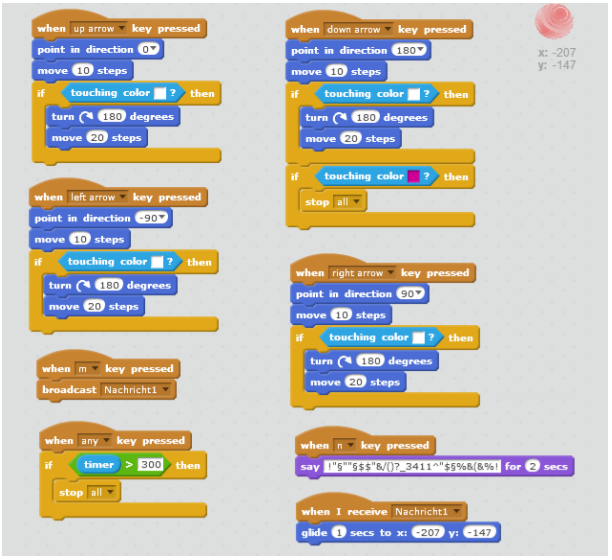

Fig. 8. Code for project 3

able to use the projects of 28 male students. Altogether, we have 127 individual Scratch projects from day 3 of the course, because some students created more than one project. The girls created 74 programs and the boys 53. Below, we grouped the sessions in *Summer School Courses*, which include the sessions in May and August during regular school holidays in the student research center, and *Class Courses*. This comprised the courses in June and July with whole 4th grade classes.

A. General Statistics

On the second course day, we introduced 28 different blocks to the students. Out of the 3,200 blocks used in Scratch projects, two third were part of the blocks introduced. The programs comprised 710 sprites all together. Overall, the students used an average of 25 blocks (median 16) and six sprites (median 4) per Scratch project. The ten most frequently applied blocks are shown in Table I. The mostly used blocks were *wait _ secs* and *when green flag clicked*, with about 11 % usage each. Another common *Event* for starting a script was *when _ key pressed* (6.2 %). For creating movements of sprites, the students often used *move _ steps* and *point in direction*. For changing the *Look*, they used *say for _ secs* and *hide* (the counterpart *show* was the eleventh most commonly used block).

TABLE I. TOP TEN BLOCKS APPLIED IN THE SCRATCH PROJECTS

Block	Image	Absolute usage	Percentage of all blocks used
wait _ secs	wait 1 secs	369	11.5 %
when green flag clicked	when clicked	368	11.5 %
move _ steps	move 10 steps	298	9.3 %
say for _ secs	say Hello! for 2 secs	273	8.5 %
when _ key pressed	when space key pressed	197	6.2 %
forever	forever	162	5.1 %
repeat	repeat 10	139	4.3 %
hide	hide	115	3.6 %
if _ then	if then	112	3.5 %
point in direction	point in direction 90	93	2.6 %

TABLE II. OVERVIEW OF THE PERCENTAGE OF STUDENTS WHO MET THE MANDATORY REQUIREMENTS

Requirement (#, term)	Total	Summer School Courses	Class Courses
Several Sprites	94 %	100 %	94 %
Sprite Motion	92 %	100 %	89 %
Iteration	81 %	82 %	78 %
Conditional Statement	63 %	76 %	56 %

B. Requirements

Almost all students met the first and second of the mandatory requirements (Table II). The most frequently missing element was the *Conditional Statement*. Twenty students did not use any appropriate block to solve this. However, iterations were only missing ten times.

C. Programming Concepts

The programs developed by the students differed strongly in complexity. Almost all students arranged the blocks in their projects in a systematic order (90 %) and used at least one event (93 %). One hundred and two projects (80 %) included more than two *Event* blocks. Parallel execution of two independent scripts was supported by 16 % of all projects. Moreover, 63 % supported the parallel launching of more than two scripts. Some subcategories of programming concepts were not included in any project. The unintegrated concepts were Lists, Boolean Logic and Dynamic Interaction. Furthermore, only four projects included variables.

D. Code Organization

No student overrode the names of the sprites. If variables were defined, the given names were meaningful. Out of all of the programs produced, only a small number (12 %) included extraneous blocks.

E. Operability

Eighty percent of the projects were categorized as being completely functional. Only 11 programs had no function at all. Most of the students did not customize the sprite (108 projects, 85 %) as well as the stage (112 projects, 88 %). Nearly two thirds of the projects did not offer any interactive elements to the user, such as user input or keyboard control to interact with the

TABLE III. PERCENTAGE OF PROJECTS CATEGORIZED BY PROJECT TYPE

Project type	Total	Summer School Courses	Class Courses
Animation	32 %	37 %	30 %
Game	17 %	54 %	2 %
Interactive Art	4 %	6 %	3 %
Music and Dance	2 %	-	3 %
Stories	45 %	1 %	61 %
Video Sensing	-	-	-

program. Over 80% of the projects were categorized as partly or completely intuitive, and only 23 projects (18 %) were not intuitive at all. The distribution of Project Types is presented in Table VII, which shows that the most frequent project type was *Story* with 45 % of all programs. It is followed by *Animation* (32 %) and *Game* (17 %). An important observation is that the class courses produced much more *Story*-programs. This differs from the students of the summer school courses who developed more programs, which were categorized as *Game*.

F. Levels of Understanding

Almost half of the students reached only the first level of the adjusted SOLO taxonomy, *Prestructural*, meaning that students did not understand how to animate the sprites in a meaningful and structured way. In some projects, no blocks were used at all. Level 2 *Unistructural* was reached in one fourth of the Scratch projects. First animations and logic were observable in them but at a very simple level. This distribution is similar to the next level, *Multistructural*. The highest level of understanding *Extended Abstract* was reached in only six projects. Table VIII shows the coding of the levels for the three most frequently created project types. We discovered that most *Story* programs could be classified as *Prestructural* or *Unistructural*. *Animation* projects were also often programmed at a *Prestructural* or *Unistructural* level. However, some were coded as *Multistructural* up to *Relational*. *Games* were mostly at Level 3 or 4. One fifth of them are even classified as *Extended Abstract*. Considering this, it seems that the sequence *Story – Animation – Game* represents an evolution line that was passed by the students step by step.

TABLE IV. PERCENTAGE OF PROJECTS IN DIFFERENT CLASSIFICATIONS AND THEIR CORRESPONDING LEVEL OF COMPREHENSION

Level of Understanding	Story	Animation	Game
L1: Prestructural	63 %	21 %	-
L2: Unistructural	23 %	51 %	9 %
L3: Multistructual	14 %	19 %	29 %
L4: Relational	-	9 %	38 %
L5: Extended Abstract	-	-	24 %

VII. DISCUSSION

Overall, our categorization turned out to be reasonably applicable, except the classification of levels of understanding, which was not entirely intuitive. Even if we reached an excellent intercoder agreement right from the beginning, the coders needed to come to an agreement for this particular classification. We argue that the ranking with five steps is not optimal for these kind of projects.

We found that there are three project types, which were used most frequently: *Animation*, *Story*, and *Game*. When connecting the results of block usage and project types, we found that the usage of *Look* blocks decreased from *Story* to *Game*. Equally, the percentage of *Motion* blocks increased from *Story* to *Game*. Based on this and the findings from Section VI. F., *Animations* seem to be a compromise between the two extremes.

We selected the projects presented in Section V based on the most commonly created project types. Program 1 is a *Story* project. Indications for this are the typical distribution of the blocks used: the most frequently used type is *Look* (especially *say for _ secs*), followed by *Control* (especially *wait for _ secs*). Furthermore, this program executes as a fixed flow of a conversation. The movement of elements is minimal. The student has accurately timed, when each sprite speaks. In order to activate the figures clown, muscle man and director the space key must be pressed exactly at the right time. The ballerina is to be activated with one click but is hidden from the very beginning. Thus, it is not clear what the user is supposed to do at this point. Therefore, usability can only be judged as "some parts are not intuitive". Considering the given requirements, this student fulfilled the first three. The program is missing a conditional statement.

The student of the second project created an *Animation*. This is easy to recognize, because there is no possible user interaction. All scripts are started with a click on the green flag and the program contains no *ask and wait* prompts. Furthermore, the program executes with a movement and a sound which does not end automatically. The heavy usage of *forever* blocks shows this.

In project 3, the student used quite a few blocks very well, and the mixture of the different blocks was balanced. One of the indications that this project belongs to the type *Game* is that a game uses fewer sprites as well as fewer blocks compared to other project types. Both characterizations are included in this case. A game, like this one, is mostly controlled by pressing keys and broadcasting messages. There are almost no *Look* blocks and many *Motion* blocks. Although there are only two sprites, the program includes nine event blocks. By using the arrow keys to control the game, it is clear to the user what to do: lead the small ball through a labyrinth, which is deliberately hard to recognize (white lines mark the way).

An important observation is that the class courses produced many more *Story* programs than the other courses. In contrast, the students of the summer class sessions developed more programs, which were categorized as *Game*.

There are many possible reasons for such differences. Firstly, the schools and teachers play a significant role in the development of their students. For example, the students of one class course had a man as class teacher. In contrast, a woman usually teaches the other class course. Furthermore, the students of three courses lived in a rural area and the others in an urban area.

Other reasons could be the educational background of the parents and the migration background of the students. Of all 18 students in one class course, only one girl had no migration

background. In addition, we as course teachers influence the students.

VIII. Conclusion

Overall, the results of our pilot courses demonstrated that at least every second child in grade 4 is able to learn basic programming with Scratch in three days. As the strongest argument for this, one might take the portion of programs that reached the second level of the SOLO taxonomy.

Yet, due to the low number of students, this result might still be caused by some specific luck circumstances of our courses. A lot of work will be needed to provide really convincing arguments that all children in Primary school are able to learn to program in several days. Of course we will repeat our programing course with many more classes in the following years.

To refine the evaluation, our next step will be to analyze the screen captures we recorded during the courses of all screens in order to find out in which order the students developed their programs. Furthermore, we will try to find connections between the video analysis and the analysis of the programming results.

We argue that the classification of the levels of understanding is not completely usable for the projects. It is important to revise this category system. In order to determine what the students are able to learn during the three-day introductory course, it is conceivable to analyze the learning outcome of the courses as was done in [18].

Acknowledgment

We thank Johannes Krugel, Ramona Olwitz und Marc Berges for supporting us during the course days.

References

[1] J. Tsan, K. E. Boyer, and C. F. Lynch. “How early does the CS gender gap emerge? A study of collaborative problem solving in 5th-grade computer science,” in Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16), ACM, pp. 388-393, 2016

[2] M. Armoni and J. Gal-Ezer. “Early computing education,” ACM Inroads vol. 5, issue 4, pp. 54-59, 2014

[3] T. Beaubouef and J. Mason. “Why the high attrition rate for computer science students,” ACM SIGCSE Bulletin, vol. 37, issue 2, 103-106, 2005

[4] K. Brennan. “Learning computing through creating and connecting. Computer,” vol. 46, issue 9, pp. 52-59, 2013

[5] T. Bell, I. H. Witten and M. Fellows. “CS unplugged - An enrichment and extension programme for primary-aged students,” 2015

[6] A. Funke, M. Berges, and P. Hubwieser. „Different perceptions of computer science,” in Proceedings of the 4th LaTiCE (LaTiCE '16), IEEE, 2016

[7] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. “The scratch programming language and environment,” ACM Transactions on Computing Education, vol. 10, issue 4, 15 pages, 2010

[8] K. Prottsman. “Computer science for the elementary classroom,” ACM Inroads, vol. 5, issue 4, pp. 60-63, 2014

[9] H. Topi. “Gender imbalance in computing,” ACM Inroads, vol. 6, issue 4, pp. 22-23, 2015

[10] L. Seiter. “Using SOLO to classify the programming responses of primary grade students,” in Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15), ACM, pp. 540-545, 2015

[11] A. Wilson, T. Hainey and T. Connolly, "Evaluation of computer games developed by primary school children to gauge understanding of programming concepts," International Journal of Games-based Learning, vol. 3, no. 1, pp. 93-109, 2013.

[12] N. C. C. Brown, S. Sentance, T. Crick, and S. Humphreys. “Restart: the resurgence of computer science in UK schools,” ACM Transactions on Computing Education, vol. 14, issue 2, 22 pages, 2014

[13] K. Falkner, R. Vivian, and N. Falkner. “The Australian digital technologies curriculum: challenge and opportunity,” in Proceedings of the Sixteenth Australasian Computing Education Conference (ACE2014), pp. 3-12, 2014

[14] John B. Biggs and Kevin F. Collis. “Evaluating the quality of learning: the SOLO taxonomy (structure of the observed learning outcome),” Academic Press, 1982

[15] G. Zaharija, S. Mladenović and I. Boljat. “Introducing basic programming concepts to elementary school children," Procedia - Social and Behavioral Sciences, vol. 106, pp. 1576-1584, 2013

[16] N. Rusk. “Scratch programming concepts,” retrieved 4th November 2016, http://scratched.gse.harvard.edu/sites/default/files/scratchprogrammingconcepts-v14.pdf, 2009

[17] A. Wilson and D. C. Moffat. “Evaluating scratch to introduce younger school children to programming,” in Proceedings of the 22nd Annual Psychology of Programming Interest Group Workshop (PPIG ’10), 2010

[18] O. Meerbaum-Salant, M. Armoni, and M. (M.) Ben-Ari. “Learning computer science concepts with scratch,” in Proceedings of the Sixth international workshop on Computing education research (ICER '10). ACM, pp. 69-76, 2010

[19] J. Denner, L. Werner and E. Ortiz. „Computer games created by middle school girls: can they be used to measure understanding of computer science concepts?,” Computers & Education, vol. 58, issue 1, pp. 240-249, 2012

[20] C. Duncan and T. Bell. “A pilot computer science and programming course for primary school students,” in the Workshop in Primary and Secondary Computing Education (WiPSCE ’15), pp. 39-48, 2015.

[21] R. L. Brennan and D. J. Prediger. “Coefficient κ: some uses, misuses, and alternatives,” Educational and Psychological Measurement, vol. 41, no. 3, pp. 687-699, 1981

[22] A. von Eye. “An alternative to Cohen’s κ,” European Psychologist, vol. 11, no.1, pp. 12-24, 2006

[23] J. R. Landis and G. G. Koch. “The measurement of observer agreement for categorical data,” Biometrics, vol. 33, no. 1, pp. 159-174, 1977