How to Transform Programming Processes in Scratch to Graphical Visualizations

Alexandra Simon, Katharina Geldreich, Peter Hubwieser
Technical University of Munich
TUM School of Education
{a.simon,katharina.geldreich,peter.hubwieser}@tum.de

ABSTRACT

Currently in many countries efforts are undertaken to bring programming education into the early levels of childhood education, like primary school or even kindergarten. Therefore, it is becoming more and more important to gain insight into which teaching methods and content would be appropriate for young students of primary levels. For this, we have designed a specific three-day introductory programming course for 4th-grade students (ages 9 - 10), which was held four times up to now. Fifty-eight children (26 girls and 32 boys) participated in the courses from May to August 2016. Besides the analysis of the course results, it is particularly interesting, in which way the programming processes of the children take place and if there are distinguishable types of young programming learners. During the courses, we captured the screens of the students' laptops and also filmed all the events in the classroom. In a case study, we used these recordings and videos to investigate the programming processes of two students (one girl and one boy from the same class). After the coding process of the videos, we developed a new visualization technique to illustrate the processes and to explore differences and special features of the individual approaches.

CCS CONCEPTS

• **Social and professional topics → Computer science education**; **Children**.

KEYWORDS

Computer science education; primary school; primary education; scratch; programming; screen capturing; mixed methods; qualitative analysis

ACM Reference Format:
Alexandra Simon, Katharina Geldreich, Peter Hubwieser. 2019. How to Transform Programming Processes in Scratch to Graphical Visualizations. In *14th Workshop in Primary and Secondary Computing Education (WiPSCE'19), October 23–25, 2019, Glasgow, Scotland Uk.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3361721.3361723

1 INTRODUCTION

Computer Science (CS) has to overcome several challenges. Schools and universities are confronted with different misconceptions and prejudices towards CS [8] which manifest themselves from an early age [17]. In order to prevent students from developing negative attitudes, one approach is to introduce Computer Science concepts like programming as early as in primary school to provide opportunities for children to experience technology and CS. Children should learn that they can use computers not only as users but also as creators [1]. This role change combined with the fun experience of programming could increase their self-confidence towards CS in particular and technology in general [20]. At the same time, the discussion about the necessity of computer science and programming in childhood education is growing steadily [20]. While several countries have already introduced CS in their primary school curricula (e.g. the UK [4] and Australia [7]), Germany has not yet developed mandatory guidelines on 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. Within this, we used Scratch as a programming environment, because it is appropriate for our target audience of 9 to 10-year old children. [6].

Although a predominant goal of the course was to change the students' attitudes towards CS, we also wanted to gain insight into how they create programs and how they apply the learned programming concepts. For this, we recorded the screens of the students' laptops and filmed all events in the classroom during the course days. Afterwards, we analyzed these recordings with a self-developed category system and visualized the results of this step in order to better understand the process. Our research questions in this work were:

- How can programming processes be analyzed by using screen recordings?
- Which differences can be detected between the programming processes of the students?
- How can programming processes be visualized so that differences are observable?

This may be the first step to investigate if there are different programming types even in primary levels and how they can be distinguished.

This work is structured as follows: First, we discuss some background work regarding computer science courses for primary school students, programming habits, and methodological approaches. 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 processes, we present the analysis of two students 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.

2 RELATED WORK AND THEORETICAL BACKGROUND

2.1 CS courses for primary school students

In recent years, a variety of courses has been developed to expose children to computer science concepts. Amongst others, various courses set their focus on programming. Tsan et al. [21] 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 materials created by the students, such as 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). 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 [5]. They described an example 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.

2.2 Usage and analysis of screen recordings

In the light of the ongoing digitization, new research methods are emerging. One of these is the capturing of computer screens which enables researchers to digitally record the interaction between humans and computers. Although other methods like videography are more common, the main advantage of screen captures is that they obtain a clear and consistent picture of the screen. Furthermore, it is possible to track the audio from the microphone synchronous with the video [19]. Another benefit is that the researcher does not have to be present as an observer next to the participant [11]. There is a variety of software to capture screens. Some of the popular applications are Camtasia[1] and Snagit[2] of the company Techsmith, Adobe Captivate[3] and Snapz Pro[4]. All of these applications offer different recording modes, e.g. deciding the size of the recorded region, recording mouse movement and the size of the resulting files [11]. Screen recordings are used within research projects of various scientific fields. The authors of [19] want to get insights into how teams use their computers to coordinate work. For this, they installed the Techsmith Camtasia software on the computers of eight participants. By starting and stopping the software on their own, they were able to decide which part of their work they want to share with the researchers. Tang et al. [19] mention the following advantages of screen recordings: no researcher has to be physically present, all computer interactions/conversations are recorded, and no physical video equipment in participants' work environments are needed. Further, the recordings capture a very detailed record of interactions.

Tsan et al. [21] created screen recordings of students programming during an in-school computer science course for 5th-grade students. The screen captures are used to discover how the students interact with the Scratch software and how they develop their programs. Because almost all children worked in pairs, it is interesting to see which student programmed what part of their project.

2.3 Visualization of (programming) processes

An important step to answer the given research questions was to visualize the coded data - the development of a perfect visualization is often a big challenge [12].

There are different classifications of visualization. In [12] the authors named three categories of visualization: a) Scientific visualization to understand physical phenomena or mathematical models; b) Software visualization, which helps people to learn the use of e.g. software; c) Information visualization which visualizes information with the use of spatial or graphical representations. A more detailed classification can be found in the periodic table of visualization [14]. For the six main categories *Data Visualization*, *Information Visualization*, *Concept Visualization*, *Strategy Visualization*, *Metaphor Visualization* and *Compound Visualization*, the authors described suitable methods. Further, they divided the methods in *Process Visualization* and *Structure Visualization*. This work focuses on *Process Visualization* methods of the *Information Visualization* class. Examples for this in the periodic system are Cycle Diagram, Petri Net, System Dynamics / Simulation, Timeline, Flow Chart and Data Flow Diagram.

Very popular diagrams in computer science are behavior and structure diagrams of the Unified Modeling Language (UML)[18]. To visualize processes and workflows in software, there are, for example, Activity Diagrams, Communication Diagrams, and Sequence Diagrams. For this work, the most interesting UML diagram type is the Sequence Diagram. It shows object interactions during a program execution arranged in a time sequence [18].

2.4 Habits of programming

A habit, in general, can be defined as "a settled tendency or usual manner of behavior" or as "an acquired mode of behavior that has become nearly or completely involuntary"[5]. By this definition, Meerbaum-Salant et al. [16] defined two characteristics to identify a habit: 1) a behavior must be settled or usual and 2) the behavior must be involuntary. They found that Scratch engenders different programming habits. The first one is a bottom-up programming process. The students try to solve problems by dragging all blocks

[1]https://www.techsmith.com/camtasia.html
[2]https://www.techsmith.com/screen-capture.html
[3]http://www.adobe.com/de/products/captivate.html
[4]http://www.ambrosiasw.com/utilities/snapzprox/

[5]https://www.merriam-webster.com/dictionary/habit

they think are appropriate to the script. Afterwards, they combine them to one script. Another found habit is called Extremely Fine-Grained Programming and complements the habit described above. Students used a top-down approach where tasks are decomposed into smaller ones. But they took them to its extreme. With the decomposition, the units became extremely small and usually lacked logical coherency. Both programming habits are contrary to common practices in computer science: to solve a problem the common process starts with the development of an algorithm. Further programming constructs are used to structure programs in a clear way.

3 DESIGN OF THE COURSE

We developed a three-day course for primary school students of the fourth grade (ages 9 - 10), see also [10]. 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 were exposed to a more and more detailed picture of programming. In 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" [2] before any programming. Hence we use social activities and group problem-solving on day one without actually working on computers. In order to learn how to split tasks into 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 the second day.

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 [15]. 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 like 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. 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 circus stories and implement these. At the end of the third day, all programs were presented in front of the course, and the students had the opportunity to comment their project.

4 METHODOLOGY

To answer the research questions (see Section 1), we apply a qualitative analysis. The described code system and visualization were developed with the first recorded data from the pilot course in May 2016. Both methods were refined in an iterative process.

4.1 Category system and coding process

To examine the process of programming of the students, we categorized all of their actions. Also we wanted to find out if and how these processes differ from child to child. In order to rate the actions which were made during the programming process, we developed a specific category system. For this, we consider which actions are possible in the Scratch environment and which actions are interesting for our research question. At the top-level, we distinguish between the four main categories: *A. Sprites*, *B. Backdrops*, *C. Blocks* and *D. Program* that are described and differentiated in the following.

A. Sprites

Every sprite has a name (*Sprite name*) and gets an identifier (*Sprite ID*) when added to the project. Possible actions which were codable for sprites are *Add sprite*, *Remove sprite* and *Edit sprite*. The code *Edit sprite* is used when the student opens the costume view and edits/changes the costume/sprite. It is common that a student opens the costume view by mistake. Then this was not coded as an edit. For every action which was coded, name and identifier of the sprite were noted.

B. Backdrops

Like *Sprites*, every backdrop has a name (*Backdrop name*) and an identifier (*Backdrop ID*). There are codes for adding a backdrop (*Add backdrop*), editing a backdrop (*Edit backdrop*) and removing a backdrop (*Remove backdrop*). Students edited a backdrop when they opened the backdrop view in Scratch. If the student chose another backdrop from a stack of added backdrops, it is coded as *Change backdrop*. For every activity which was coded, name and identifier of the backdrop were captured.

C. Blocks

Scratch blocks have a predefined name (*Block name*). When adding a block to the project (*Add block*) it gets an identifier. A block can be removed from the script (*Remove block*) and moved to another position (*Move block*). The code *Edit block* is used when the student changes e.g. the text in a *say_for_secs*. Every block has a current position within the Scratch script which was coded as *Block position*. When moving a block the *Block position* is the position after the movement. For every action which was coded the name and identifier of the block were noted.

D. Program
Because it is important for the programming process, program actions were also categorized. The student can do a *Run program* e.g. when clicking on the green flag. Also the program can be stopped (*Stop program*). Most of the students created more than one project. To track if a new project is created, we added the category *Create new file*. To capture the change between projects that are the codes *Open file* and *Close file*.

By using this code system, we coded the screen captures of two children of the third course day. Two researchers rated these screen captures to assess inter-coder agreement and reliability. First, we imported the screen captures and the video of the camera of this day in Adobe Premiere Pro. It was shown that this software is well suited for this kind of analysis because the videos can be viewed in parallel and they can be easily re-sized. After the synchronization and cropping of the video tracks, we coded the videos with markers which do not alter the video itself.

To analyze the programming processes, we used screen captures in combination with videotaped material because this procedure increases the understanding of the working methods and the coding becomes easier. Sometimes the children swap their seats with their friends for some minutes. This is very difficult to detect only with the screen capture. Also because the voices of the students are similar at this young age.

Further, the children say important things for us from time to time, like "Huh? Why doesn't this work?". This is a marker for us to take a closer look at this position.

4.2 Visualization

As shown in Section 2.3, there are currently many visualizations for software and hierarchical data. The problem with most diagrams and images is that they show parallel or hierarchical data. In contrast, a programming process is sequential; there are no hierarchical structures or alternatively branches. But still, it is important to see different sprites in the same diagram. For this purpose, we found no appropriate visualization.

Even the described UML diagrams which are common in software engineering are not usable in this case. While it is possible to model interactions and communication within a program or between program and user, this is not the use case in this work.

As result of the coding of the videos, we obtained a huge table with all coded positions, which includes e.g. time-stamp, ID, action, and sprite. With this table, we developed two different diagram types to show the results of each student in a clear and comprehensible way. They are described and illustrated in the following sections.

4.3 Development diagram

We called the first created diagram type *Development Diagram*. It is based on UML sequence diagrams which show how objects operate with one another and in what order. Instead of this original diagram, *Development Diagrams* describe the programming process for all Sprites of one Scratch project, but no communication between them. It contains eight elements which are displayed in Table 1. Fig. 1 shows an example *Development Diagram*. It starts with the first Sprite which is added to the project. Because the time-line runs from top to bottom, every new step in the programming process is a new line in the diagram. After adding Sprite A, the user added Sprite B. The editing of Sprite B is shown with the three points in line 3. To describe if a block was added, edited, deleted or moved to another position within the script, the image of the block is displayed with the associated icon in front of it (Table 1). A dashed horizontal line stand for a run of the script. In this example, the project is started with a click on the green flag. Sprite C is deleted without editing the script or the sprite itself.

Table 1: Elements of *Development Diagrams*

Element	Description
▭	A sprite is added
+	A block is added
...	A block / sprite is edited
>	A block is moved
X	A block / sprite is removed
move ● steps	Image of the block
--- ---	Program is started
——	Lifeline of a sprite

Table 2: Elements of *Sprite Diagrams*

Element	Description
▭	A sprite is added
+	A block is added
...	A block / sprite is edited
>	A block is moved
X	A block / sprite is removed
move ● steps	Image of the block
→	Block is contained in another block

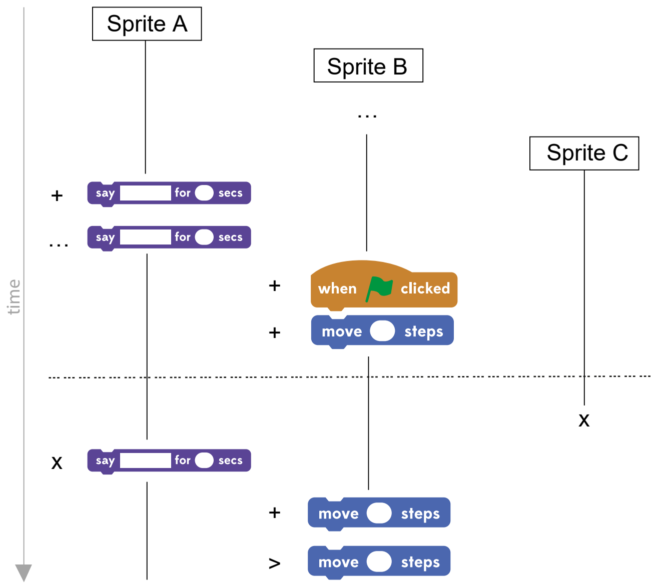


Figure 1: Example of a *Development Diagram*

4.4 Sprite diagram

The second diagram type we called *Sprite Diagram*. It is in the broadest sense a mix between an object diagram and an activity diagram. *Sprite Diagrams* describe the programming process for one Sprite of the Scratch project. It contains seven elements which are displayed in Table 2. Fig. 2 shows an example *Sprite Diagram*. It starts with the name of the Sprite.

Because the time-line runs from top to bottom, every new step in the programming process is a new line in the diagram. The columns of the diagram are blocks which were used in the project. Because it is interesting, which exact block was added or deleted, the changes of the same block are written among each other. Some block types, like *forever* or *if_then* can contain other blocks. To visualize which blocks were added within *forever* or *if_then*, arrows are used.

5 APPLICATION

All courses were taught in German by a female, formally educated primary school teacher, who was also a CS researcher. 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. To collect the important data from the course, we used a mixed method approach. All sessions were videotaped, enabling us to analyze all actions and interactions of the students. At the beginning and 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

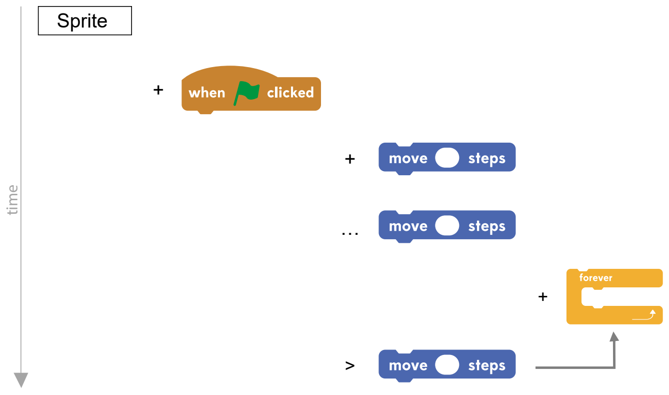


Figure 2: Example of a *Sprite Diagram*

student computers during the programming exercises on days 2 and 3, to get an image of the students' working methods. To collect the students' Scratch programs, we saved the projects after the course had ended. In order to obtain further insight into the quality of the projects and to make a first step towards the whole analysis, we analyzed the Scratch results of the participants [9].

Further, the students described at the beginning of day 3, which program they want to create, which sprites they want to use and how the sprites are supposed to act during execution. We compared these statements with the codings and the results of the Scratch projects.

Applying this category system, two researchers rated the two cases which we describe in the following sections to assess inter-coder agreement and reliability. In order to get an idea of the accord, the Brennan coefficient [3] and raw agreement were calculated. Both resulted in an almost perfect agreement according to Landis [13]. The common Cohen and Krippendorff factors are not applicable, as they require a normal distribution of the coded cases over the codes [22].

6 RESULTS

In order to illustrate the described data analysis process, we show two example cases in the following sections. Both children participated in the same course in July 2016 and were selected randomly for the analysis. We intentionally chose a female and a male student. We anonymized their identities.

6.1 Student 1 - Oliver

The first student, Oliver, is a 10-year old boy. At the beginning of day 3, he explained that he wanted to create a program with five sprites. They are intended to play soccer or do other sports activities. After the coding process, we examined if Oliver reached his set goal from the beginning of the course day: his project is exactly like he wanted it to be (see Fig. 3 and 4). He integrates five characters which are acting and three sprites just for decoration (the letters "T", "O" and "R" which stands for the word "goal" in German).

The program is a part of a soccer game. As soon as the figure "Adrian" touches the ball, it glides into the goal.

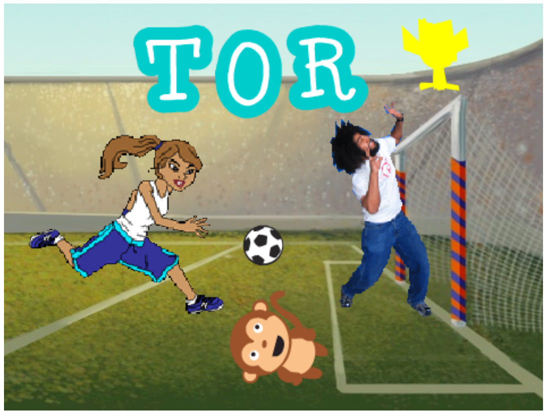
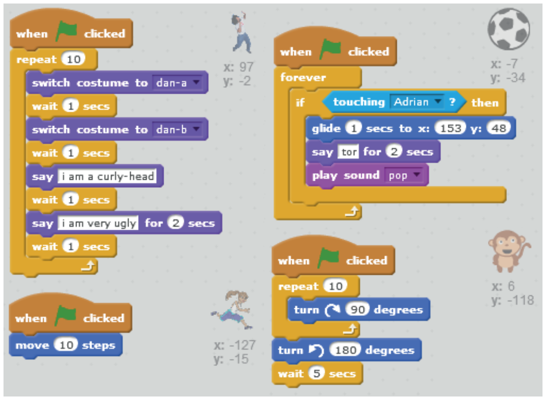


Figure 3: Stage for the project of Oliver (Student 1)



```
when [flag] clicked
repeat (10)
    switch costume to [dan-a]
    wait (1) secs
    switch costume to [dan-b]
    wait (1) secs
    say [i am a curly-head]
    wait (1) secs
    say [i am very ugly] for (2) secs
    wait (1) secs

when [flag] clicked
move (10) steps

when [flag] clicked
forever
    if <touching [Adrian] ?> then
        glide (1) secs to x: (153) y: (48)
        say [tor] for (2) secs
        play sound [pop]

when [flag] clicked
repeat (10)
    turn ↻ (90) degrees
turn ↺ (180) degrees
wait (5) secs
```

Figure 4: Code for the project of Oliver (Student 1)

Even during the coding process, the researchers discovered some interesting positions in the programming process. This was confirmed by the diagrams which were prepared afterward. Altogether, the student worked in a very straight-forward way. He added blocks (and sometimes edited it directly after adding), and then he went to the next step. He rarely edited a block again after many other steps. Regarding this, we can easily find special points in the diagrams. For example, Fig. 9 shows such a point. Oliver added the same block *touching* to the sprite "Football player" many times, edited it, moved it and deleted it. This shows an irregularity in his behavior. When taking a look at exact this position in the screen capture, the boy said: "Huh? Why doesn't this work?" This shows that the diagram was a good indicator to find this position. The problem for him here was that he first chose the current sprite for touching. In Scratch, nothing happened at this moment.

Editing the script of a wrong sprite was a typical mistake of his. During coding, we noticed that at some point round about 15 starts (click on green flag) in turn happened. At a closer look, we found that he wanted the sprite Adrian to move towards the soccer ball. But instead of using a loop to solve this, he only added a *move_steps* block and clicked the green flag as often as needed to move Adrian to the soccer ball. As shown in Fig. 7, it seems that Oliver deleted some sprites without any obvious reason. For example, the sprite "Ball" The removal of blocks, on the other hand, is rare. What was special about his programming was, that he sometimes added the same sprites again after removing them. An example in the figure is "Soccer ball". He edited the script of this sprite and removed it suddenly. A few steps later he added the sprite again to the project.

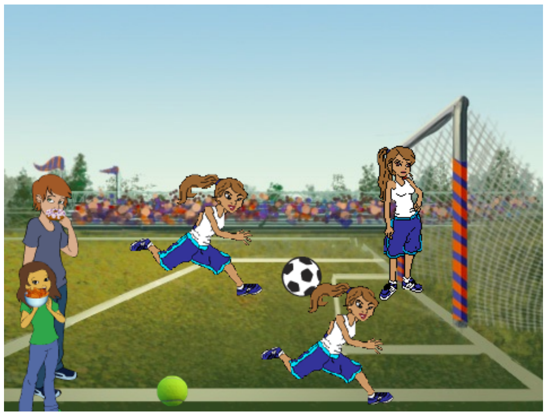
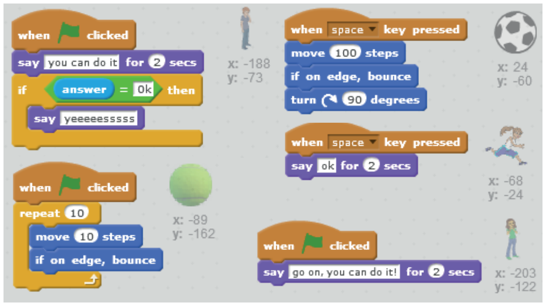


Figure 5: Stage for the project of Charlotte (Student 2)



```
when [flag] clicked
say [you can do it] for (2) secs
if <(answer) = [Ok]> then
    say [yeeeeesssss]

when [flag] clicked
repeat (10)
    move (10) steps
    if on edge, bounce

when [space] key pressed
move (100) steps
if on edge, bounce
turn ↻ (90) degrees

when [space] key pressed
say [ok] for (2) secs

when [flag] clicked
say [go on, you can do it!] for (2) secs
```

Figure 6: Code for the project of Charlotte (Student 2)

6.2 Student 2 - Charlotte

The second student was a 10-year old girl in the same class. We called her Charlotte. At the beginning of day 3, she explained, that she wanted to include four sprites in her project: an elephant, children, a monkey and a circus director. They should be part of a circus show.

Already at the beginning of the coding, we determined, that her first Scratch program differs greatly from her planning. Altogether Charlotte created three different projects and fullfilled all given requirements with her first project (see Fig. 5 and 6). Similar to Oliver, the researchers encountered interesting points in the programming process, which are confirmed by the diagrams. She worked a bit different than her classmate. She used a lot of time at the beginning of each program to create the scene and she rearranged elements often. It seems like Charlotte paid great attention to the overall picture of her project. She edited sprites almost every time after adding it to the scene (Fig. 8). After adding a new block, she looked up all other blocks, whether they still fit in the project. In contrast to Oliver, the sprite diagrams of Charlotte are less wide but much deeper (see Fig. 10). Thus she used less different blocks, but moved

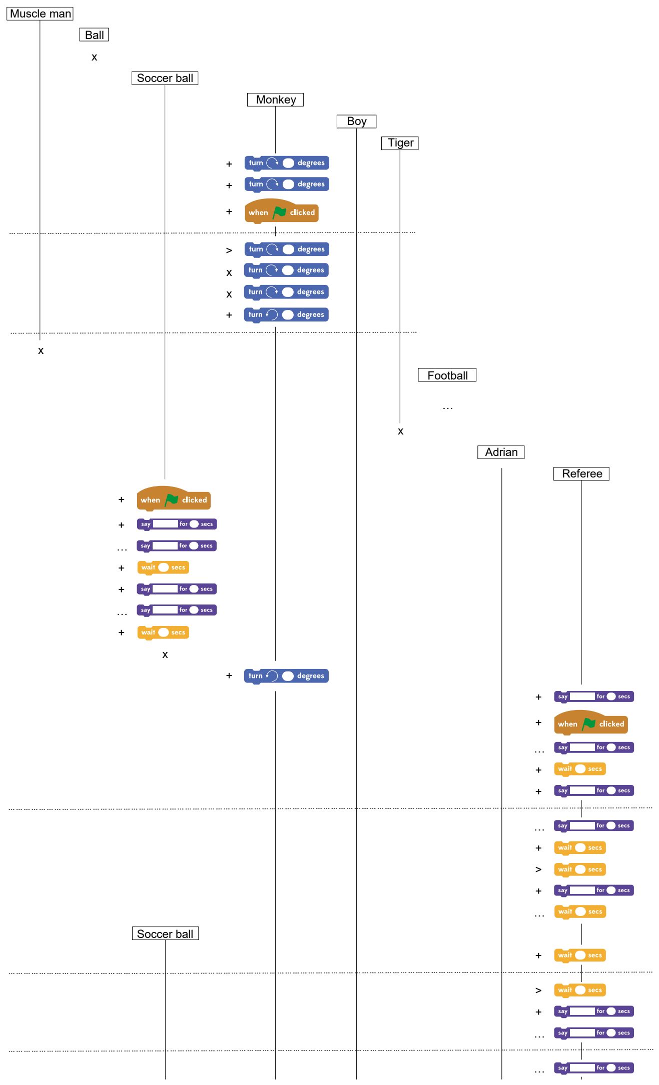


Figure 7: *Development Diagram* for step 1 to 51 of Oliver (Student 1)

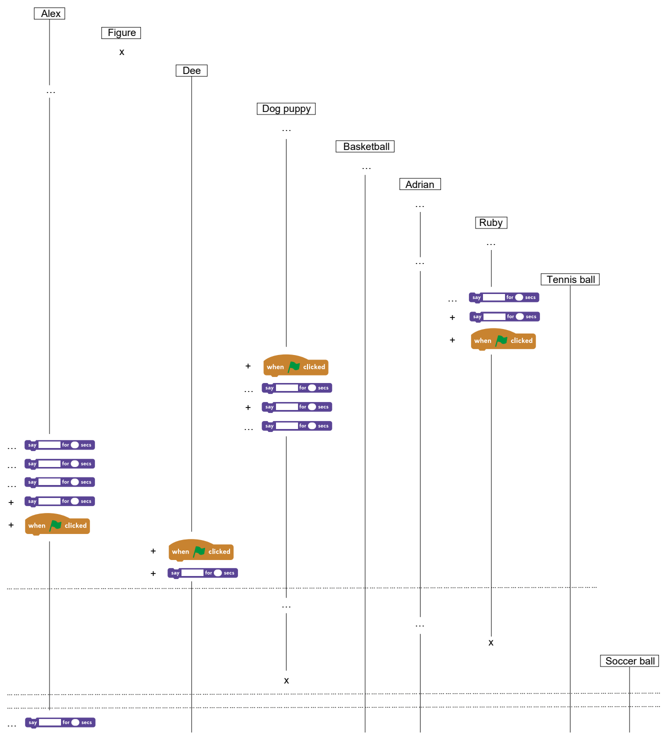


Figure 8: *Development Diagram* for step 1 to 39 of Charlotte (Student 2)

and edited them very often. Special points in her programming process can be found in moments when she worked very straightforward. This was often when she replicated a script to another sprite.

Further as shown in Fig. 10, she sometimes seemed to be a bit confused about structures. With this code, she wanted to ask the user something and use the answer for the further program sequence. But she did not succeed with this. In the resulting project there is this construct with *if_then*, *answer* and _=_ included, but it does not work. After Charlotte couldn't solve this problem, she ignored it. Another interesting feature is that she edited blocks (especially *say_for_secs*) often before she added them (Fig. 8).

7 DISCUSSION

After analyzing the results, we found that the working processes between the two students differ at some points (Section 6). Oliver (Student 1) realized exact the program, what he described to do at the beginning of course day 3. Instead, Charlotte, Student 2, programmed a completely different project than she intended to do. The first student in our example worked in a very straightforward way. He added a block and went on with the next step. He rarely moved back to a block and edited a block again after many other steps. Charlotte used a lot of time at the beginning of each program to create the scene and rearranged elements often.

With the use of the diagrams, we found important points in the programming processes of the children. Fig. 9 shows such a point for Oliver. He added the same block *touching* to the sprite "Football player" many times, edited it, moved it and deleted it. This shows an irregularity in his behavior. By analyzing the development diagram (Fig. 7) of the boy, we found that he deleted some sprites without any obvious reason. Further, he sometimes added the same sprites again after removing them. Charlotte paid great attention to the overall picture of her project. As one can see in Fig. 8, she edited sprites almost every time after adding them to the project. After adding a new block, she looked up all other blocks, whether they still fit in the project. This can be observed in the diagrams as often changes of the current edited sprites. After she couldn't solve a problem in her program, she ignored it. There are many possible reasons for such differences. Firstly, this could be influenced by personal characteristics of the students and maybe our results show

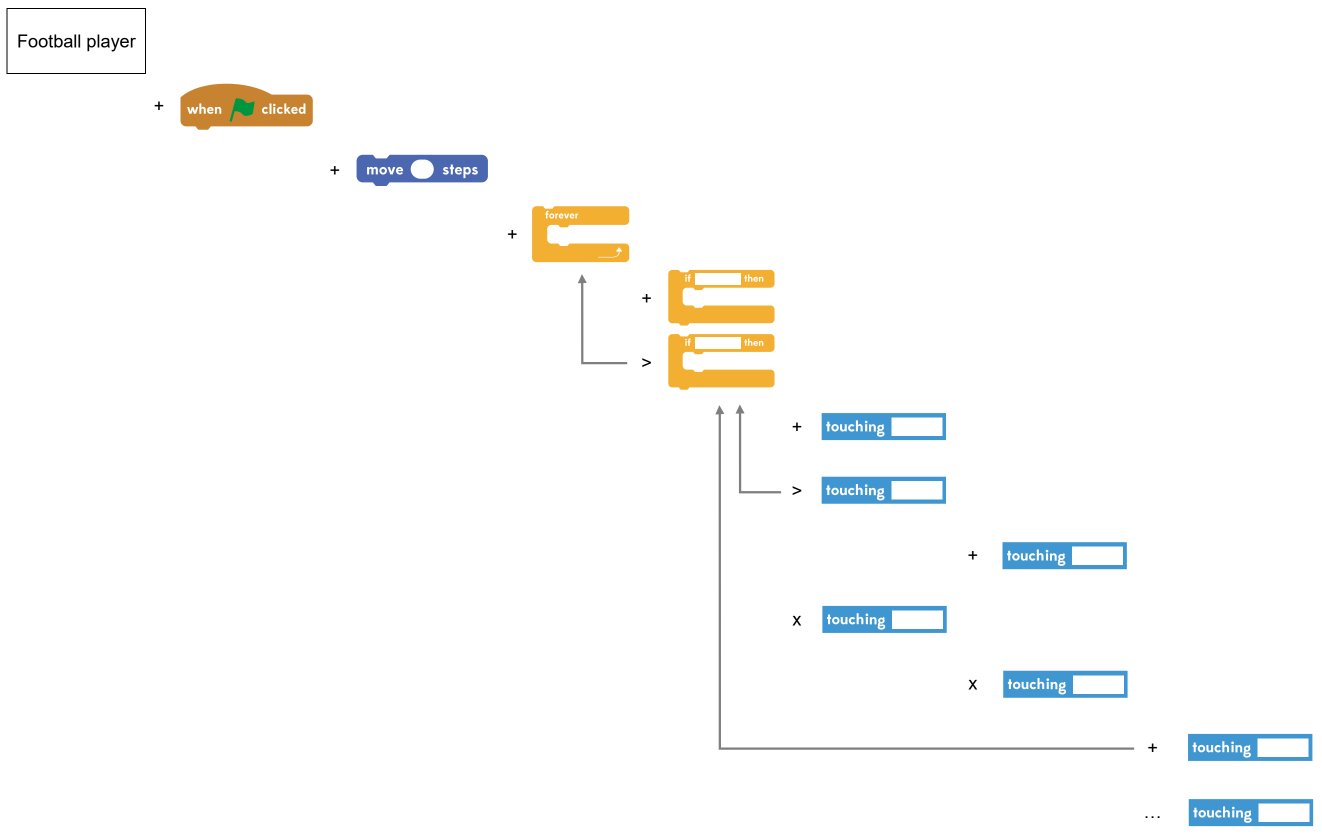


Figure 9: *Sprite Diagram* for sprite *Football player* of Oliver (Student 1)

the typical working method of the two children. Further, we have to take into account that the students could be affected by their neighbors, for example. To determine this, we have to include the video recordings of the course day in our analysis.

Due to the small number of students, we presented in this work, it is not possible to make general statements and assumptions for all participants of our courses. But this use-case analysis is a first step towards a better understanding of the programming processes of primary school children.

Our analysis showed, that the developed category system (see Section 4.1) is sufficient for this kind of process analysis and our research questions. Currently, there is no need for further adaptions of the system. The visualization combined with the screen recordings can confirm the initial assumption about an interesting position in the programming process. Further, this combination can help to understand - the students' challenges at this point. With the usage of both methods, we find for example the deviation in Oliver's programming process as shown in Fig. 9. It is true that the visualizations always show only a certain section of the processes and never contain all the information. But this is an intentional reduction of information. While the development diagram shows the chronological development process of all sprites in a project, the sprite diagram describes the programming of exactly one sprite. Both diagrams have their advantages and it is appropriate in an analysis process to use both types in parallel.

8 CONCLUSION

With the developed diagram types, it is easily possible to find important points in the programming processes of the students. Although we show only the results of two children in detail. From four courses in 2016, we obtained screen captures of 37 children. Sometimes we faced technical challenges so that it was not possible to capture the screens of all students. These screen captures are currently coded. Based on this work and the created visualizations, we will develop a system for the automated generation of the visualizations from a list of codes. Afterwards, the visualizations will be automatically generated and analyzed. This can be applied to the other screen recordings for a quantitative evaluation. One goal of this is to find out if we can distinguish different programming types between the students, the classes or specific groups of children. We think that the visualizations together with screen captures and video recordings will play an important role in this step. The findings will be useful for teacher education and as a tool for teachers in class to understand the programming process of their students. In a previous work of ours [9] we found that the children created three project types in particular: Story, Game, and Animation. We

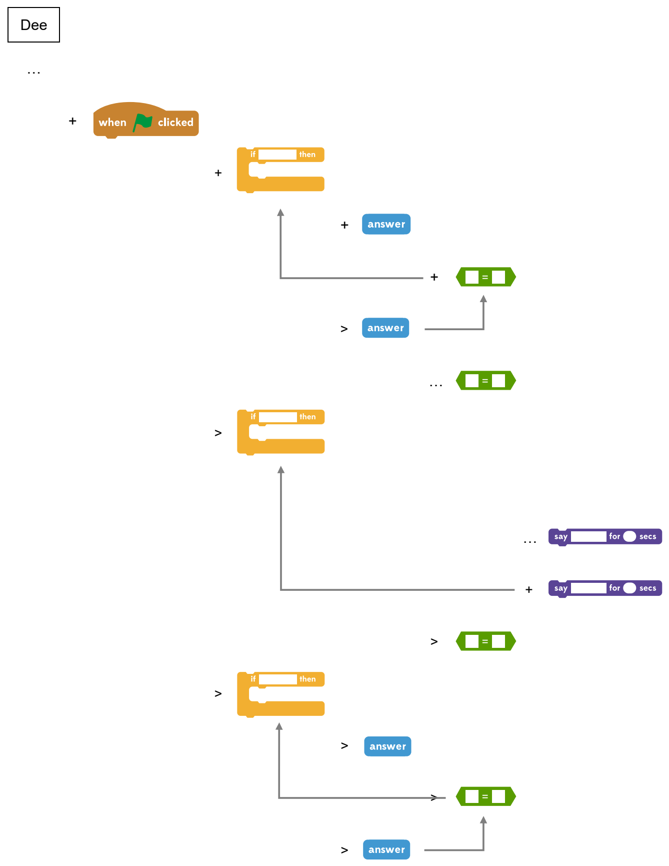


Figure 10: *Sprite Diagram* for sprite *Dee* of Charlotte (Student 2)

wonder if the project type relates to the programming process and the other way around. Furthermore, we will try to find connections between each part of the analysis: video, screen captures and the analysis of the Scratch projects.

REFERENCES

[1] M. Armoni and J. Gal-Ezer. Early computing education. *ACM Inroads*, 5(4):54–59, 2014.
[2] T. Bell, I. H. Witten, and M. Fellows. *CS Unplugged: An enrichment and extension programme for primary-aged students.* 3rd edition, 2015.
[3] R. L. Brennan and D. J. Prediger. Coefficient Kappa: Some Uses, Misuses, and Alternatives. *Educational and Psychological Measurement*, 41(3):687–699, 1981.
[4] 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*, 14(2):1–22, 2014.
[5] 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*, pages 39–48, 2016.
[6] C. Duncan, T. Bell, and S. Tanimoto. Should your 8-year-old learn coding? In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, WiPSCE '14, pages 60–69, New York, NY, USA, 2014. ACM.
[7] K. Falkner, R. Vivian, and N. Falkner. The Australian Digital Technologies Curriculum: Challenge and Opportunity. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, ACE '14, pages 3–12, Darlinghurst, Australia, Australia, 2014. Australian Computer Society, Inc.
[8] A. Funke, M. Berges, and P. Hubwieser. Different Perceptions of Computer Science. In *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, pages 14–18. IEEE, 2016.
[9] A. Funke, K. Geldreich, and P. Hubwieser. Analysis of Scratch Projects of an Introductory Programming Course for Primary School Students. In *Proceedings of the 2017 IEEE Global Engineering Education Conference (EDUCON)*, pages xx – xx. IEEE, 2017.
[10] K. Geldreich, A. Funke, and P. Hubwieser. A Programming Circus for Primary Schools. In *Proceedings of the 9th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, pages 46–47. 2016.
[11] B. Imler and M. Eichelberger. Using screen capture to study user research behavior. *Library Hi Tech*, 29(3):446–454, 2011.
[12] M. Khan and S. S. Khan. Article: Data and information visualization methods, and interactive mechanisms: A survey. *International Journal of Computer Applications*, 34(1):1–14, November 2011.
[13] J. R. Landis and G. G. Koch. The Measurement of Observer Agreement for Categorical Data. *Biometrics*, 33(1):159–174, 1977.
[14] R. Lengler and M. J. Eppler. Towards a periodic table of visualization methods of management. In *Proceedings of the IASTED International Conference on Graphics and Visualization in Engineering*, GVE '07, pages 83–88, Anaheim, CA, USA, 2007. ACTA Press.
[15] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, 10(4):1–15, 2010.
[16] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Habits of programming in scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 168–172, New York, NY, USA, 2011. ACM.
[17] K. Prottsman. Computer science for the elementary classroom. *ACM Inroads*, 5(4):60–63, 2014.
[18] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual.* Pearson Higher Education, 2004.
[19] J. C. Tang, S. B. Liu, M. Muller, J. Lin, and C. Drews. Unobtrusive but invasive: Using screen recording to collect field data on computer-mediated interaction. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, pages 479–482, New York, NY, USA, 2006. ACM.
[20] H. Topi. Gender imbalance in computing. *ACM Inroads*, 6(4):22–23, 2015.
[21] J. Tsan, K. E. Boyer, and C. F. Lynch. How Early Does the CS Gender Gap Emerge? In *the 47th ACM Technical Symposium*, pages 388–393, 2016.
[22] A. von Eye. An Alternative to Cohen's . *European Psychologist*, 11(1):12–24, 2006.