Yang, J. C. et al. (Eds.) (2018). Proceedings of the 26th International Conference on Computers in Education. Philippines: Asia-Pacific Society for Computers in Education

Improving on Guidance in a Gaming Environment to Foster Computational Thinking

Sören WERNEBURG[a*], Sven MANSKE[a], Jessica FELDKAMP[a] & H. Ulrich HOPPE[a]
[a]*COLLIDE Research Group, University Duisburg-Essen, Germany*
*werneburg@collide.info

Abstract: The application of newly introduced computational constructs is challenging for learners when they start programming from scratch. Basic code examples can be used and modified to help learners exploring these constructs in order to enable a seamless transition to create own programming code. This paper presents a redesign of the game-based learning environment *ctGameStudio* to address this issue. The existing level structure is enhanced using implicit and explicit scaffolds to support the learning process with respect to computational thinking competences. The redesign of *ctGameStudio* has been evaluated in a study with eleven students in a pre-posttest scenario by measuring the learning gain and the learning progression. In comparison to a previous study with 40 students, we could show that students apply newly introduced computational constructs better if they were supported by the proposed scaffolds.

Keywords: Computational Thinking, Abstractions, Scaffolding, Guidance, Programming

1. Introduction

"Thinking at multiple levels of abstraction" (Wing, 2006) is an important part of Computational Thinking (CT). For this reason, learners need an environment that guides them through these levels and supports them in their learning process. *ctGameStudio* is a game-based learning environment and contains a microworld with a programmable virtual robot (Werneburg, Manske, & Hoppe, 2018). The programming interface of the virtual robot is a visual block-based programming tool based on Blockly[1]. In this environment, two modes are available – a story mode for novices in programming and an open stage that can be used subsequently. In the story mode, the users learn basic computer science (CS) concepts with a focus on fostering CT competences. In the open stage the students have the open-ended task to build strategies to fight against other robots programmed by other users to train the learned competences.

However, a previous evaluation of the story mode revealed weaknesses in introducing basic CS concepts such as loops and other abstractions (Werneburg et al., 2018). Based on the findings of the associated study, we present conclusions and a redesign of the learning environment in this paper, which supports the learning process with scaffolds.

Also, Grover and Basu (2017) discovered similar problems in other learning environments related to CS concepts. To solve this problem, they developed a learning scenario with digital and unplugged activities to "draw on dynamic math representations" (Grover, Jackiw, Lundh, & Basu, 2018) with different interfacing. The provision of "representational flexibility" to make choices between different representations like different programming tools but also between abstractions and computational models to learn CT is necessary (Hoppe & Werneburg, 2018). The familiarization of different kinds of abstractions has to be structured, because CS concepts are hard to learn through open exploration (Mayer, 2004).

Our approach for the redesign of the story mode of *ctGameStudio*, for learning CS concepts and fostering CT competences through a guided learning process is with implicit and explicit scaffolds. We point out that abstractions can be learned more easily through structured introduction

[1] https://developers.google.com/blockly/

with these features. Finally, we show the results of a new study we conducted with eleven people using *ctGameStudio* and compare these results with the outcomes of the exploratory pre-study with 40 people. We present our findings on analyzing the grades of the pre-posttest and the learning progression of the students.

2. Background and Related Work

The definition of CT has evolved over the past few years. Wing (2006) described CT as "solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science." However, Aho (2012) defined CT as "thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms" with an appropriate model of computation. Building abstractions based on these models are active actions of creating logical artifacts in term of data and process structures (Hoppe & Werneburg, 2018) and there is a need to invent "appropriate new models of computation" (Aho, 2012).

Learning environments based on models with a low threshold are needed to give the opportunity to build abstractions. Visual block-based programming tools like Blockly (Fraser, 2015) let students start early solving CT tasks (Weintrop & Wilensky, 2015) and therefore also building abstractions in the related learning environment. They "are relatively easy to use and allow early experiences to focus on designing and creating, avoiding issues of programming syntax" (Grover & Pea, 2013).

However, in the game-based learning environment *ctGameStudio* where students can use such visual block-based programming tool based on Blockly, learners had problems applying presented abstractions to given programming tasks (Werneburg et al., 2018). Students had to explore these concepts from scratch and struggled with the creation of computational artifacts based on abstractions. According to Leutner (1993), "students without any support learn how to play the game, but that they only acquire a minimum of verbal knowledge about domain-specific concepts, facts, rules, and principles." Mayer (2004) emphasized, that it is a "mistake to interpret the current constructivist view of learning as a rationale for reviving pure discovery as a method of instruction." Studies in the context of LOGO show that learners with "guided discovery methods performed better on generating and debugging new programs than did students who learned by pure discovery" (Lee, 1991).

Effective guidance can support student exploration (Mayer, 2004). Scaffolds can restrict the comprehensiveness of tasks, give hints towards possible solutions as well as provide affordances to perform actions (Podolefsky, Moore, & Perkins, 2013). Lazonder and Harmsen (2016) extend the typology of scaffolding – they define guidance as a more generic term. *Process constraints* reduce the number of erroneous paths towards a student's solution. *Prompts* are timed reminders to perform specific actions. *Heuristics* provide basic structures how to accomplish a task. *Explanations* support the learners by describing the steps towards a solution or specific action. Podolefsky et al. (2013) subdivided mechanisms of guidance into implicit and explicit scaffolds. While prompts, heuristics, and explanations are explicit scaffolds, process constraints are more implicit.

Lee et al. (2011) promoted the Use-Modify-Create-Progression, that we will use as one basic scaffold for our approach of the redesign. At the beginning, a minimal example is given – described as "not mine" – which can be tried out in the *Use* phase to become familiar with the learning task. Then, the students can modify this programming code and observe changes in its output. As a result, "scaffolding increasingly deep interactions will promote the acquisition and development of CT" (Lee, Martin, & Apone, 2014). Working with given abstractions in a combined *Use-and-Modify* phase supports the "exploration" of these abstractions. By slightly modifying and running code the students discover the functionality of given constructs which reduces the threshold to enter the upcoming *Creation* phase. This phase of creation is the main part for CT tasks and gives the opportunity to be creative as „learning the language of creative coding is essential to expression in a digital medium" (Peppler & Kafai, 2005).

With this concept, it is possible to target specific strategies and guide the students to transfer their elaborated strategies to other learning tasks, but it is necessary to "provide appropriate levels of constraints and guidance" (Grover et al., 2018). Knowledge-based instructions help to structure

the development process, because "linking declarative and procedural knowledge is recommended as a means to this end" (Swan & Black, 1993).

To evaluate CT skills, it is important to understand how the students came to their solutions. Many studies have been carried out in the past: Interviews were analyzed, in which students were asked how their learning progressed (Baker & Yacef, 2009). In other studies, the solutions of the students were analyzed to create appropriate assessments (Grover & Basu, 2017; Werner, Denner, Campe, & Kawamoto, 2012). There are also studies in which specific questions about pre-generated programming artifacts have been analyzed. The learning process, however, can rather be captured with logging of the activities, the analysis of which can provide information on how the final solution was reached (Werner, McDowell, & Denner, 2013). For the game-based learning environment *ctGameStudio* such logging agents are implemented to observe the behavior of creating, but also of using and modifying of given programming code. Various metrics support this observation (Werneburg et al., 2018).

3. Redesigning *ctGameStudio*: Scaffolding and Guidance

ctGameStudio[2] is a web-based learning environment to foster CT competences by students with a game-based approach. The students control a virtual robot with a visual block-based programming tool. The interactions of the robot within the microworld provide direct feedback to the learners. By solving pre-defined programming tasks, certain CT competencies are fostered through the learning environment. The targeted CT competences are – in line with Selby & Woollard (2013) – algorithmic thinking, generalization, evaluation, decomposition, and abstraction. Additionally, *ctGameStudio* provides a seamless transition from guided tasks towards open exploration. According to Bauer, Butler, and Popović (2017), learning environments "need to combine open-ended exploration with sufficient structured guidance." The story mode of *ctGameStudio* will be the guided part with scaffolds and the open stage will be a mode for open-ended exploration.

The level system of the story mode is the basic layer for the redesign. In the previous version of *ctGameStudio*, each level focused on a specific CS concept and was related to a suitable CT competence. In combination with a task, a specific CS construct or abstraction type was introduced with descriptions. The learners had to start from scratch to solve these tasks. One observation from the previous study was that the students struggled with applying newly introduced abstractions. They started to solve the tasks with previous known components with a trial-and-error behavior and avoided the use of new mechanisms (Werneburg et al., 2018).

For the redesign, we apply the concept of the Use-Modify-Create Progression. Each level contains several sublevels. Now, the first sublevel of each level addresses using and modifying given programming code. This provides affordances in the sense of an implicit scaffold. Learners are encouraged to use and modify this given code without the possibility of taking detours. The transition from the *Use* phase to the *Modify* phase is smooth and both phases are combined in the respective first sublevel after a new concept has been introduced. The given programming code is executable but does not solve the sublevel – it encourages learners to modify this programming code. Through this implicit scaffold, the learners get familiar with newly introduced abstractions starting from a basic configuration.

The second sublevel of each level addresses the *Create* phase of the Use-Modify-Create progression. Students have to build programming code based on the abstraction type introduced in the previous sublevel. In the sense of explicit scaffolds, students can request a prompt with heuristics for the desired abstraction type. Additionally, only a limited set of commands is available in each level. Therefore, the learners are gradually introduced to the functionality of the programming tool and the interaction with the microworld. In each level, however, new command blocks are added to expand the range of possibilities steadily until the last level, where all command blocks are available. The "block library" allows access to descriptions of the commands at any time. These mechanisms feature two types of guidance: First, the restriction to command blocks relevant to each task is a *process constraint*. Second, the *explanations* in the block library specify how to perform actions with the given code blocks. In addition to the general concept of scaffolding in *ctGameStudio*, our

[2] http://ct.collide.info/ctgamestudio/

framework contains mechanisms of guidance that are specific to particular CT concepts and levels (cf. table 1). Visual feedback of the interaction with the microworld supports the provision of affordances. For example, in level 5, when an enemy robot attacks the virtual robot, the learner implicitly gets requested to dodge or defend.

The level system is redesigned to address both, a specific microworld construct as well as a specific computational construct (cf. table 1). The microworld constructs are always – primarily or secondarily – geometric constructs. Structured moving of the virtual robot is in each level an important part to act in the microworld. But the interaction with other objects is also targeted. The computational constructs are connected to different abstraction types like loops, events, procedures and functions.

Table 1
Framework for scaffolds of ctGameStudio.

Microworld Constructs	**Computational Constructs**	**Targeted Level**	**Guidance**	
			Implicit	**Explicit**
Geometric concepts like squares and equilateral triangles	Sequences and loops	1.1, 1.2 and 2.1 - 2.3	Constraint (restricted sequence length)	Prompt (Overlay of the targeted path)
Interaction with static objects like avoiding bombs	Event mechanism	3.1 - 3.4	Visual feedback (scanning of the robot)	Explanation (written guide how to use scanning)
Interaction with moving objects in the shape of an enemy	Procedures	4.1 - 4.4	Visual feedback (enemy reaction)	Heuristic (attack strategy templates)
Interaction with attacking objects	Functions with parameters	5.1 and open stage	Visual feedback (enemy reaction)	Heuristic (defend-attack strategy template)

The first two levels (with their respective sublevels) involve geometric concepts on the layer of the microworld. The learners are introduced to simple movements, distances and directions in order to be able to let the robot walk on the shape of squares and equilateral triangles in the sense of LOGO. At the layer of computation, the possibilities are provided to form sequences and to abstract repetitions into loops. Implicitly the learning environment provide the learner with the required programming blocks (*forward*, *repeat n times*, ...); students can explicitly request a visualization of what the movement pattern should look like. Level three sets the focus on the event mechanic as a computational mechanism. In the microworld, the student interacts with objects like bombs and ammunition. For programming, scanning blocks are introduced. Implicit, the act of scanning is visualized with a beam. Likewise, invisible objects become visible when they are detected. In the computational constructs layer, flags and callbacks are introduced to realize events. Level four and five relate to the idea to interact with enemies. The strategies of the opponents are becoming in each sublevel more sophisticated. While the enemy robot is resting at first, he will dodge later, then defend himself and then attack the robot of the user.

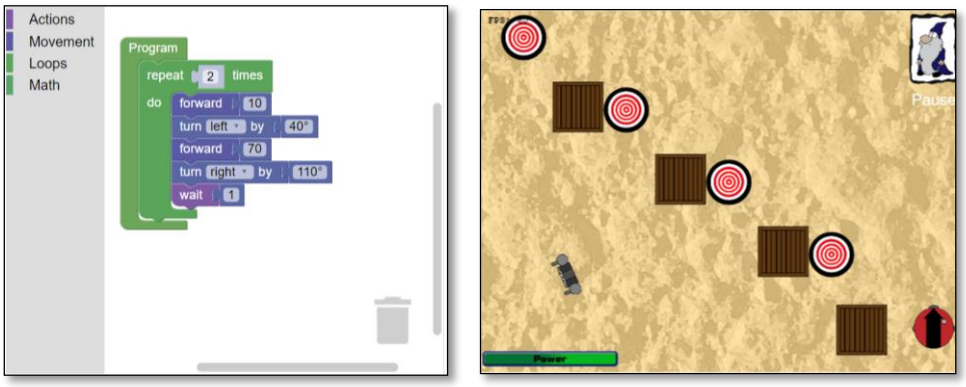

Figure 1. Given programming code (left) for the microworld of sublevel 2.1 (right).

Figure 1 shows the first sublevel for the *loop* abstraction where the students vary parameters in the head of the loop as well as parameters and the sequence of statements in the loop's body. In a first step, the students can run the given programming code and can observe the behavior of the virtual robot in the microworld. Varying parameters and running the programming code helps to understand the behavior of the virtual robot using this abstraction. If the students need additional help, they can click on the mentor in the right upper corner (Figure 2).

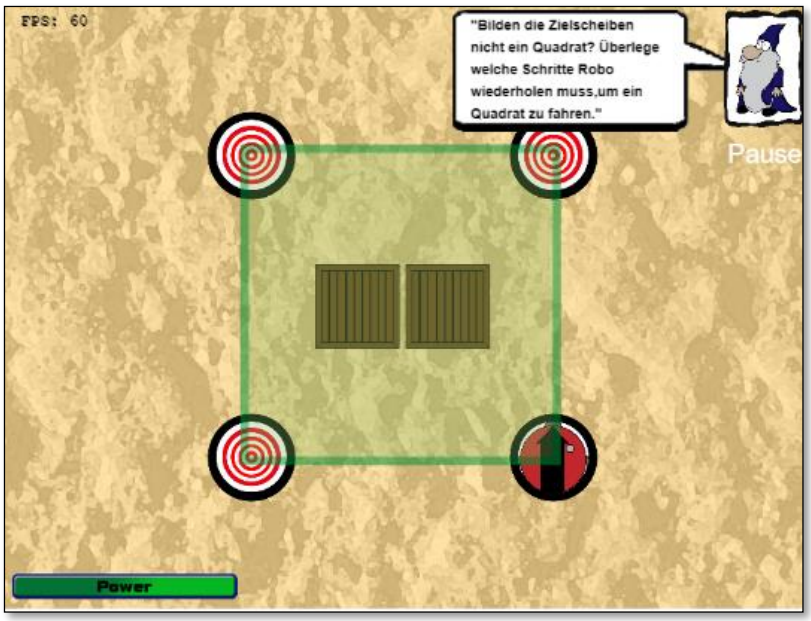

Figure 2. Visualization of hints available in the sublevel 2.2 related to the *loop* abstraction.

For each level, the second sublevel is a *Creation* level. The students create their own solutions using the introduced abstraction. In the third (and fourth) sublevels, the students must use the created programming code of the previous level to generalize their ideas to solve these level. In the case of level 2 (loop abstraction) the students have to generalize drawing from a square (figure 2) to other equilateral polygons. This progression is intended from LOGO, the main task for this generalization process is the rethinking of the outer angles and the related number of loop repeats (Simmons & Cope, 1990) – and this goal can be achieved with an equilateral triangle or an approximated circle.

4. Evaluation

The aim of this study is to find out whether *ctGameStudio* leads through guided learning to a learning gain for the students. To do this we use a pre-post testing, analyze the programming behavior of the learners, and compare the data with the results of the previous study.

4.1 Experimental setting

The participants (female: 10; male: 1; mean age: M = 24.36, SD = 3.57) had a time limit of 45 minutes to complete the levels related to movement, loops, and events (eight sublevels) of the learning environment *ctGameStudio*. The participants have been introduced to the *ctGameStudio* environment and have been instructed to solve the tasks. In addition to demographic data, a self-rating regarding to programming experience and was requested (figure 3). The field of study of four persons is Applied Computer Science and of seven persons Applied Cognitive and Media Science.

The previous study was with 40 students (24 men; 16 women; mean age: M = 22.23, SD = 3.98). They had a time limit of 45 minutes, too (Werneburg et al., 2018).

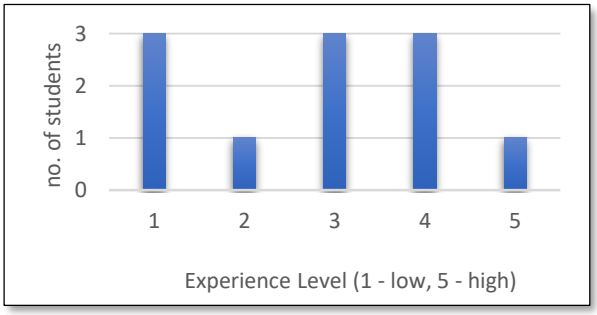

Figure 3. Self-rating of the students regarding to programming experience.

4.2 Learning gain

A pre-posttest with tasks related to basic CS concepts (sequences, loops, branching) was used for the assessment of learning gain. The questionnaire is based on the CT test of Román-González, Pérez-González, and Jiménez-Fernández (2017). The test contains seven multiple choice questions and two tasks where the learners had to write programming code on paper. In addition, these tasks have a focus on completion, debugging, and sequencing programming code related to computational practices (Brennan & Resnick, 2012). We defined the null hypothesis "Subjects score higher or equal in the pretest than in the post test" and the alternative hypothesis *"Subjects score higher in the posttest than in the pretest"* with the results in table 2.

Table 2

Student pre-post assessment results.

Descriptive Statistics		**Shapiro-Wilk-Test**		**Hypothesis test**	
Pretest	**Posttest**	**Pretest**	**Posttest**	**Wilcoxon-Test**	**Effect size**
M = 18.18 SD = 5.47	M= 21.00 SD = 4.15	p = 0.56	p = 0.57	p = 0.016	r = 0.47

As can be seen in table 2, the Shapiro-Wilk-Test (Shapiro & Wilk, 1965) allows the assumption of a normal distribution for both, the pre-test and the post-test since $p > 0.05$ holds in each case. Thus, the Wilcoxon-Test (Wilcoxon, 1945) can be used, and this test shows that the null hypothesis must be rejected because of $p < 0.05$ and the hypothesis "Subjects score higher in the posttest than in the pretest" must be accepted, where the effect size with $0.3 < r < 0.5$ only suggests that it is a medium effect (Cohen, 1992) and seven positive, two neutral, and only two negative ranks supports the correctness of the hypothesis.

These results may depend on the field of study of the students. This results in the null hypothesis "*Students of Applied CS achieve an average lower or equal score than students in the field of Applied Cognitive and Media Science.*"

Another possible dependency can be related to the self-rated experience in programming. To analyze this, we clustered the students depending on their self-rating (cf. figure 4) in the group with low experience (level 1 or 2; 4 students) and high experience (level 4 or 5; 4 students). This results in the null hypothesis "*Subjects with higher programming experience score on average lower or equal than subjects with low programming experience*" and the alternative hypothesis "*Subjects with higher programming experience score on average higher than subjects with low programming experience.*"

For both cases, the Mann-Whitney-U-Test (Mann & Whitney, 1947; Wilcoxon, 1945) is applicable. Table 3 shows for the field of study that in each case is $p > 0.05$ and the null hypothesis must be accepted. There is no significant difference in the results of the pretest and the posttest. As a consequence, the results do not depend on the field of study. In case of the self-rating of the programming experience the results for the pretest show that the null hypothesis must be rejected because of $p < 0.05$ with a high effect size with $r > 0.5$. For the posttest the null hypothesis must be accepted because of $p > 0.05$. A higher self-reported programming experience influenced the results

of the pre-test positively. After using the *ctGameStudio* environment, this effect could not be observed anymore – the results in the post-test to not differ across the clusters (high/low experience).

Table 3
Student pre-post assessment results depending on the field of study and self-rating of the programming experience.

Clustering	Descriptive Statistics		Mann-Whitney-U-Test	
	Pretest	Posttest	Pretest	Posttest
Applied CS	M = 18.50 SD = 4.20	M= 21.00 SD = 3.16	p = 0.35	p = 0.51
Cognitive and Media Studies	M = 18.00 SD = 6.40	M = 21.00 SD = 4.87		
High experience	M = 19.75 SD = 5.68	M = 22.75 SD = 3.59	p = 0.03 (r = 0.77)	p = 0.13
Low experience	M = 13.00 SD = 0.82	M = 18.75 SD = 4.86		

4.3 Learning progression

To analyze programming code, we defined several features which are related to CT competences (Werneburg et al., 2018). The feature "# runs" describes the testing and evaluating behavior of the created programming code. One observation of the previous study was, that students need less runs if the level was easy to solve.

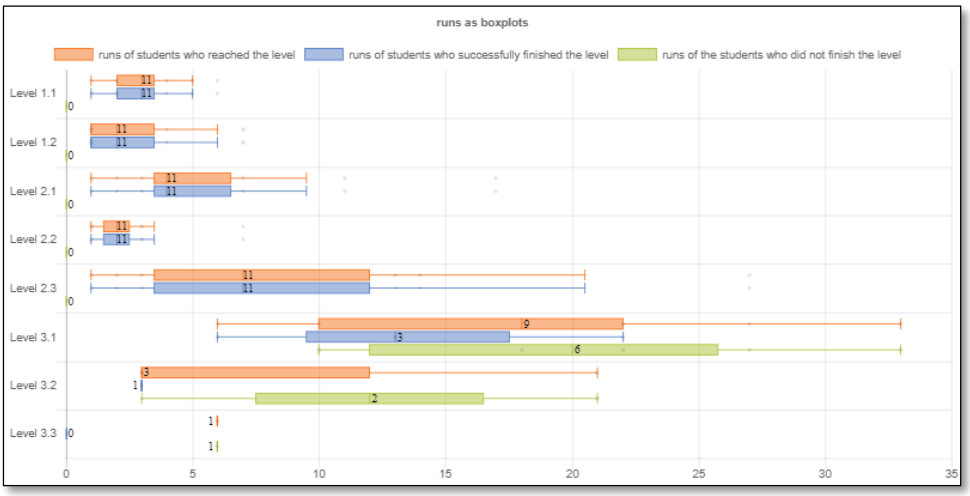

Figure 4. Distribution of # runs per level with the number of users.

The boxplots in figure 4 show the distribution of the runs per sublevel. It is observable that each student finished the first five levels (1.1-2.3), but not everyone started programming in level 3.1. Level 3.1 and 3.2 were only solved by two of three participants. The last sublevel 3.3 was only reached by one student and this student did six runs but did not finished this level.

The first sublevels (1.1, 2.1, 3.1) contain the Use-Modify approach. A predefined programming code is given, and learners should try it out and then modify it to get a correct solution. In these sublevels the boxplots show that much more runs were done than in the associated second sublevels, which contain the creation part of the Use-Modify-Create progression according to Lee (2011). For example, the median in level 2 falls from four runs in sublevel 2.1 to two runs in sublevel 2.2. This indicates that the respective first sublevel was used extensively to get to know the corresponding abstraction type and that the creation process in the second sublevel was easier to carry out.

With the feature "# Changes per run" the advanced planning behavior can be analyzed. It gives an insight into how far the users have kept to the Use-Modify-Create progression. Additionally, one observation of the last study was that most changes were done in the first run in a sublevel, when the concepts were better understood by the learners. Levels with "abstraction gaps" showed an unstructured pattern. Figure 5 shows the first two sublevels for introducing loops of this study.

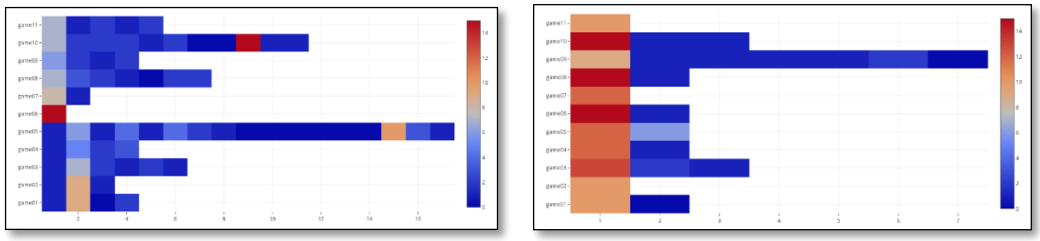

Figure 5. # Changes per run for level 2.1 (Use-Modify, left) and level 2.2 (Create, right).

Sublevel 2.1 was the level to use and modify given programming code related to loops. While the learners game01 to game05 used the given programming code without any changes (dark blue tiles), the learners game06 to game12 made direct first and major modifications for the first run (light blue tiles – skipped using part of the Use-Modify-Create progression). Nevertheless, the first five users made major changes for the second run. The following changes per run are minor for all users. In the next sublevel, where users should create a solution from scratch with this abstraction type, the same behavior – most changes were done before the first run – is observable.

In figure 6, the behavior of the user game10 in sublevel 2.2 of the just-in-time observation is presented. A change (per run) can be moving a block in the block structure, creating new blocks, deleting existing blocks and varying parameters. While all blocks used were created before the first run, the user only varied parameters for the second and third run, so the last run led to the level's completion.

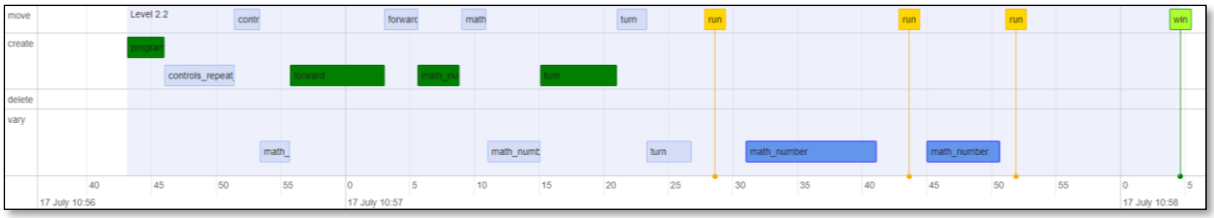

Figure 6. Timeline of the programming code changing behavior of user game10 in sublevel 2.2.

That indicates that the use and modify phase are helpful to get a basic understanding of the targeted abstraction type. The light blue events in the timelines of figure 6 indicate that they are all related to the associated loop block (in this case "controls_repeat"). Green blocks indicate block creations that have been placed only on the canvas. For example, after the creation of the "turn" block, this block was moved into the loop block and then the parameter of this block was changed. Dark blue blocks indicate that parameters have been changed that are either independent of the loop or if they are related to the loop, as in this case, after the run for which the loop was created.

As in the previous study, level 3.1 with the introduction of events is a big gap for the students, too. Although each learner reached the level, only three students completed it. Feedback of the students after the study showed that primarily the temporal restriction meant that they could not begin/complete the level. On average, each user who started the sublevel spent *15.58* minutes in this level, while those who completed, it took an average of *1.62* minutes longer.

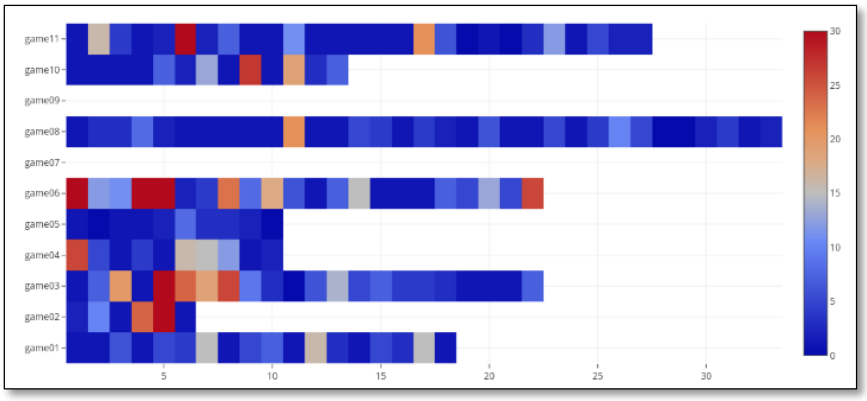

Figure 7. # Changes per run for level 3.1.

As can be seen in figure 7, an unstructured pattern of changes is for each student observable. While two students began directly to modify the given code, the others used the code before modifying it. At the end, user game2, game3, and game10 have completed the level. Each of these students applied in the propagated Use-Modify approach: initially using the given programming code, then inserting minor and then major changes. In the end minor changes followed to optimize the behavior of the virtual robot and to complete the sublevel.

5. Conclusion

In this paper, we presented a redesign for the game-based learning environment *ctGameStudio*. The new version provides implicit and explicit scaffolds, in which students get just-in-time feedback and on-demand hints. The Use-Modify-Create progression, which is integrated in the level structure, guides the students through different tasks with a focus on specific microworld constructs and computational constructs. The comparison of two studies (with and without scaffolds) showed that the students performed better with scaffolds. Also, learners who consistently followed the Use-Modify-Create progression completed more levels in the same time than learners who, for example, skipped the *Use* phase and started directly with modifications. Of course, it is not desirable to limit the learners in their *Creation* phase by forcing them to work through the *Use* phase. However, this could lead to better results as learners would be able to gradually build understanding of the given computational construct.

We are aware that our proposed framework for guidance only covers static mechanisms which are following best practices for learning CT and introductory programming. In our future work, we will extend *ctGameStudio* by using dynamic and adaptive scaffolds which are built on process and content analysis. Code metrics are a common approach to measure code quality in the field of software engineering. However, such methods are only partially applicable for "small-scale" education-oriented programming. The use of dynamic and static code metrics for a similar use case has been proposed in the context of creative problem solving with programming, which demands CT competences (Manske & Hoppe, 2014). Implemented features like # runs, # changes per run, # creates, # consecutive changes per create, and time spent in minutes will be the basis for this approach towards automated assessment of CT competences and for providing dynamic feedback to the learners in game-based environments.

References

Aho, A. V. (2012). Computation and computational thinking. *The Computer Journal, 55*(7), 832-835.

Baker, R. S., & Yacef, K. (2009). The state of educational data mining in 2009: A review and future visions. *JEDM-Journal of Educational Data Mining, 1*(1), 3-17.

Bauer, A., Butler, E., & Popović, Z. (2017). *Dragon architect: open design problems for guided learning in a creative computational thinking sandbox game.* Paper presented at the Proceedings of the 12th International Conference on the Foundations of Digital Games.

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. *Paper presented at the Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*.

Cohen, J. (1992). A power primer. *Psychological bulletin, 112*(1), 155.

Fraser, N. (2015). Ten things we've learned from Blockly. *Paper presented at the Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*.

Grover, S., & Basu, S. (2017). Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. *Paper presented at the Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*.

Grover, S., Jackiw, N., Lundh, P., & Basu, S. (2018). Combining Non-Programming Activities with Programming for Introducing Foundational Computing Concepts.

Grover, S., & Pea, R. (2013). Computational Thinking in K–12. *Educational Researcher, 42*(1), 38-43. doi:10.3102/0013189X12463051

Hoppe, H. U., & Werneburg, S. (2018). Computational Thinking - more than a Variant of Scientific Inquiry! In S.-c. KONG (Ed.), *Computational Thinking Education*. *Hong Kong. Springer. (to appear)*

Lazonder, A. W., & Harmsen, R. (2016). Meta-analysis of inquiry-based learning: Effects of guidance. *Review of Educational Research, 86*(3), 681-718.

Lee, I., Martin, F., & Apone, K. (2014). Integrating computational thinking across the K--8 curriculum. *ACM Inroads, 5*(4), 64-71.

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., . . . Werner, L. (2011). Computational thinking for youth in practice. *ACM Inroads, 2*(1), 32-37.

Lee, M. O. C. (1991). Guided instruction with Logo programming and the development of cognitive monitoring strategies among college students.

Leutner, D. (1993). Guided discovery learning with computer-based simulation games: Effects of adaptive and non-adaptive instructional support. *Learning and Instruction, 3*(2), 113-132.

Mann, H. B., & Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 50-60.

Manske, S., & Hoppe, H. U. (2014). Automated indicators to assess the creativity of solutions to programming exercises. *In Proceedings of the 2014 IEEE 14th International Conference on Advanced Learning Technologies, 497-501*.

Mayer, R. E. (2004). Should there be a three-strikes rule against pure discovery learning? *American Psychologist, 59*(1), 14.

Peppler, K., & Kafai, Y. (2005). Creative coding: Programming for personal expression. *Retrieved August, 30*(2008), 314.

Podolefsky, N. S., Moore, E. B., & Perkins, K. K. (2013). Implicit scaffolding in interactive simulations: Design strategies to support multiple educational goals. *arXiv preprint arXiv:1306.6544*.

Román-González, M., Pérez-González, J.-C., & Jiménez-Fernández, C. (2017). Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test. *Computers in Human Behavior, 72*, 678-691.

Selby, C., & Woollard, J. (2013). Computational thinking: the developing definition.

Shapiro, S. S., & Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika, 52*(3/4), 591-611.

Simmons, M., & Cope, P. (1990). Fragile knowledge of angle in turtle geometry. *Educational Studies in Mathematics, 21*(4), 375-382.

Swan, K., & Black, J. B. (1993). Knowledge-Based Instruction: Teaching Problem Solving In a Logo Learning Environment. *Interactive Learning Environments, 3*(1), 17-53.

Weintrop, D., & Wilensky, U. (2015). To block or not to block, that is the question: students' perceptions of blocks-based programming. *Paper presented at the Proceedings of the 14th International Conference on Interaction Design and Children.*

Werneburg, S., Manske, S., & Hoppe, H. U. (2018). ctGameStudio - A Game-Based Learning Environment to Foster Computational Thinking. *Paper presented at the 26th International Conference on Computers in Education, Philippines.*

Werner, L., Denner, J., Campe, S., & Kawamoto, D. C. (2012). The fairy performance assessment: measuring computational thinking in middle school. *Paper presented at the Proceedings of the 43rd ACM technical symposium on Computer Science Education.*

Werner, L., McDowell, C., & Denner, J. (2013). Middle school students using Alice: what can we learn from logging data? *Paper presented at the Proceeding of the 44th ACM technical symposium on Computer science education.*

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics Bulletin, 1*(6), 80-83.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33-35.