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

# *ctGameStudio* – A Game-Based Learning Environment to Foster Computational Thinking

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

**Abstract:** This paper presents a game-based learning environment supporting the acquisition of computational thinking skills. The game environment offers two types of usage: a guided progression through levels and an open stage. Seven levels of guided learning were evaluated in a study with 40 students to measure their learning progress. We introduce five features to describe the students' learning behavior and map the performance parameters to computational thinking competences. In our analysis, we distinguish different basic actions and compare the usage and distribution of these actions over time and between groups of learners with different achievement levels. This allows for identifying specific challenges related the interactive acquisition of CT skills.

**Keywords:** Computational Thinking, Game-Based Learning, Abstractions, Learning Analytics

## 1. Introduction

During the last decade, the term Computational Thinking (CT) emerged from the field of computer science education. *"The essence of Computational Thinking lies in the creation of 'logical artifacts' that externalize and reify human ideas in a form that can be interpreted and 'run' on computers"* (Hoppe & Werneburg, 2018). These logical artifacts can be programming solutions that learners create in order to be executed on a computer and to produce solutions to a specific problem. Wing (2014) emphasized the importance of abstraction in this context: "the most important and high-level thought process in computational thinking is the abstraction process. Abstraction is used in defining patterns, generalizing from specific instances, and parameterization."

Although the term Computational Thinking has gained popularity more recently, Papert (1996) described the idea already in conjunction with turtle geometry in Logo. In the 1970s, Logo has been designed as an easy-to-learn educational programming language fostering self-directed and explorative learning in various fields of mathematics, science, language and arts. One of the key elements in Logo was turtle graphics that enables learners to steer a virtual turtle to draw shapes and explore properties of geometry. Logo included functional programming elements (inherited from LISP), provided natural and easy access to recursion but also offered simple iterative constructs.

The current practice of CT education is dominated by block-based visually oriented programming languages. Accordingly, much effort has been invested into the development of innovative approaches and tools for introductory programming following this principle. Such tools support the learners in avoiding syntactical errors when writing their programming code. Scratch (Resnick et al., 2009) is one of the most prominent examples of this approach. It is rooted in the Logo history and philosophy following the principles of "low threshold and no ceiling". Scratch contains basic elements of game engines allowing learners to create their own interactive microworlds. However, such activities primarily involve aspects of game design and do not necessarily foster CT skills in a systematic way. While Dr. Scratch (Moreno-León, Robles, & Román-Gonzáles, 2015) facilitates the assessment of CT skills for programming solutions in the Scratch environment, it does not support guidance for an incremental learning progression towards becoming a "Computational Thinker".

In this paper, we present an approach to introductory game-oriented programming by using a virtual robot, which can be programmed by the learners with basic operations taken from turtle geometry. *ctGameStudio* provides learners a visual block-based programming editor that is connected to a microworld hosting the virtual robot. We support the learner in acquiring the relevant CT skills and competences, such as abstractions and decomposition, through a guided learning process. We present the results of an exploratory study with 40 participants using the *ctGameStudio* environment. In this context, we particularly elaborate on findings from analyzing the learning progressions.

## 2. Background and Related Work

Selby (2013) identifies five main dimensions of CT skills: algorithmic thinking, decomposition, generalization, abstraction, and evaluation. Following an understanding based on general common sense, the concept of abstraction is often seen quite similar to the one generalization. However, as pointed out by Wing (2008) and further explicated by Hoppe & Werneburg (2018), abstraction in computer science is based on specific constructive operations used in the formal specification and creation of logical artifacts including concepts such as recursion, abstract data types, higher-order functions and lambda expressions. When we address abstraction in CT we should be aware of this specific meaning.

In the history of computer science education, Logo is an important reference and milestone as an educational programming language particularly designed for mathematical problem-solving (Papert, 1996). The reduced set of syntactic rules and predefined commands helps to introduce programming without the typical barriers for novices (McNerney, 2004). On the other hand, Logo was designed to give access to deeper mathematical constructs, including recursion and functional abstraction as well as turtle geometry as an alternative to using global Cartesian coordinates (Abelson & DiSessa, 1986). The design of the Logo turtle followed the concept of "body syntonic" reasoning, where the learners take a first person perspective and imagine they would be in the place of the turtle. This enables the learners to understand geometric concepts through their "sense and knowledge about their own bodies" (Papert, 1980). Papert and Turkle (1990) understand "computation as the ultimate embodiment of the abstract and formal".

Using games with virtual agents in a microworld is another way to support a first-person perspective. Games also create challenges at various levels and thus come with a high degree of problem orientation in the sense of Wing (2006). The strategies formulated as solutions to the challenges are of algorithmic nature. With Robocode and RoboBuilder, Weintrop and Wilensky (2012) provide a game-based environment with a text-based and a visual block-based programming tool to develop strategies for robot fights. The attempt to win grants learners a strong incentive to improve the algorithms developed. Robocode and RoboBuilder are open environments with a high degree of freedom to develop and refine strategies. Although a well-structured wiki is available for the API, the entry threshold is very high. Students start with the whole set of commands and have no introducing levels to become familiar with the learning environment or to learn basic CT skills. Kazimoglu, Kiernan, Bacon, and MacKinnon (2012) defined in their game-based environment "Program Your Robot" consecutive tasks starting with a low entry threshold. With increasing demands on the design-debug-run stages, key aspects of CT are fostered.

Following the tradition of Logo's turtle geometry, a low threshold and smooth progression facilitate the development of abstractions. However, if students have only this opportunity at their disposal, they cannot discover certain abstraction types on their own. So, it is inevitable to provide a flexible progression for both forms of learning (Bauer, Butler, & Popović, 2017) – guided learning and open exploration – like on-demand or just-in-time direct guidance for new tools.

Another kind of learning support and guidance is provided by intelligent tutoring system. ITS are based on a knowledge model of the domain and solution strategies for a certain set of problems (Corbett, Koedinger, & Anderson, 1997). Although working under this type of system requires a high level of knowledge organization, it has been applied to teaching programming tasks.

Blikstein (2011) collected in an open programming course all actions, including programming events and non-coding events like button-clicking and parameter variation in NetLogo. He proposed an approach directed to identify behavior from these log files and uses the

error rate per compilation as an outcome measure. His approach is the base to develop metrics to provide an assessment tool and pattern-finding lenses for programming tasks.

Moreno-León et al. (2015) went one step further with the assessment tool "Dr. Scratch" based on the Scratch learning environment. Dr. Scratch provides students with feedback about their programming activities. The assumption is that feedback on the errors and successes in the programming process will help to improve the learners' CT skills. The underlying analysis checks the solutions for concepts like parallelism, synchronization, etc. Then, the authors map the observed competence to a scale from 0 to 3. For example, the usage of an "if"-clauses is scored with one point, whereas an "if-else" receives two points, logic operations add another point etc. These scores are the basis for feedback to the learners, which is easy and fast to understand. But the question is, how to map theses CT concepts to an overall CT competence assessment; also, it is questionable whether a binary attribute (used the block/does not use the block) is sufficient to assess a competence.

Brennan and Resnick (2012) conducted artifact-based interviews to assess the computational concepts. They point out, that it is difficult to conceive CT skills as a binary state at a single point in time based on a given program solution. They put a focus on process features such as the development of ideas and the role of debugging. Grover and Basu (2017) used questionnaires to detect typical errors. Their conclusion is that the students (from $6^{th}$ to $8^{th}$ grade) have problems to comprehend how, e.g., loops and Boolean operators work. And these problems with fundamental abstractions lead to difficulties in developing executable programming solutions. Using these abstractions needs to be analyzed more closely in order to be able to give meaningful interventions on a pedagogical level.

Among the discussed approaches, the study of Blikstein is the only one that does not use a block-based visual programming environment. Instead, it uses the error rate per compilation as a feature for the analysis. Sure, the subjects are students from higher education. But the question is, whether errors in syntax and semantic are indicators of CT skills. Visual block-based programming tools are often used, since they are easy to use for students and have a low threshold and avoid issues of programming syntax (Grover & Pea, 2013).

Taking into account the grown practice of visual programming environments in education we want to design an open arena for the acquisition of CT skills that exploits a gaming approach but also scaffolds a systematic learning progression through computational abstractions such as loops, procedures and functions. In the following section, we will discuss how the use of these abstractions – and not only concepts – can be defined and measured. The approach is evaluated in a study with 40 participants.

## 3. Conceptual Approach

In conjunction with a student software project at the University of Duisburg-Essen entitled "Computational Thinking through Gaming", we designed and developed the web-based learning environment *ctGameStudio*[1]. This project aimed at providing a game-based environment to learn programming and CT skills together with a guidance component. Guidance is personalized through a mentor as a virtual companion.

Figure 1 shows how students control a virtual robot in a microworld via the visual programming tool. Running the program code results in a direct visual feedback in the microworld. The mentor in the upper right corner gives just-in-time hints to support the programming process. The choice fell on a block-based visual programming tool designed with Blockly[2] to stay as close as possible to natural language wording and descriptions for strategies. Developed by Google, Blockly is a library to build an editor that facilitates the building of visual block-based programming tools, especially editors, and provides a framework for generating code in text-based languages like JavaScript (Pasternak, Fenichel, & Marshall, 2017).

In addition to the perceptual benefits of a visual block-based programming tool, syntax errors are excluded, and the learning and gaming flow is not interrupted. Combining blocks always

[1] http://ct.collide.info/ctGameStudio/
[2] https://developers.google.com/blockly/

produces executable code (Resnick et al., 2009). Also, highlighting of the actual executed programming block supports the evaluation of the code.

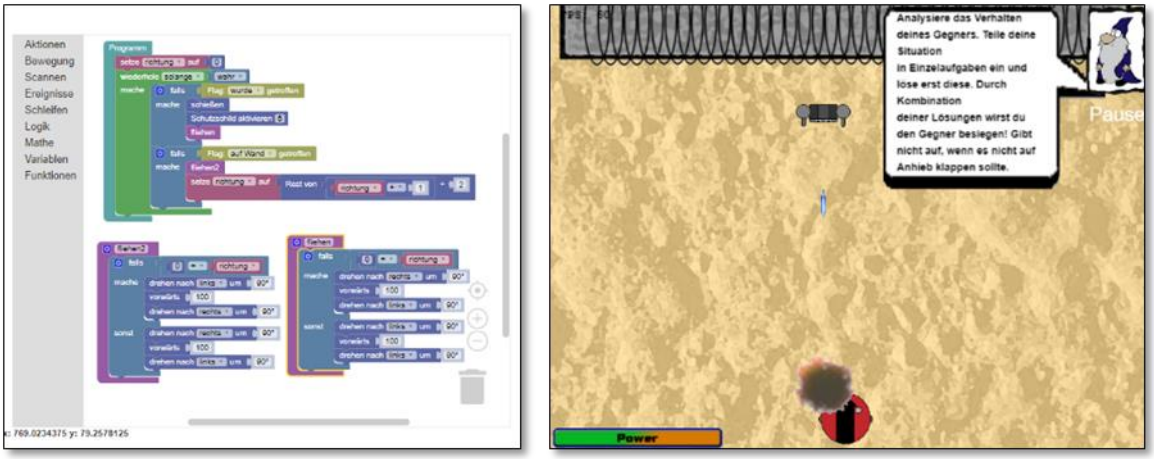


*Figure 1*. *ctGameStudio* with a visual block-based programming tool (left) and the microwold with a red controlable virtual robot (right)

The students have access to a guided learning environment and an open stage. The goal of the open stage is to defeat robots programmed by other users in a tournament. The goal of the guided learning environment is the preparation for this tournament mode, but also to develop computational competences, abstractions and to foster CT. The guided learning environment leads students through the exploration of new abstractions. According to Grover and Basu (2017), we focus on the basic abstractions. In the open stage, the learners can apply these new concepts.

According to the study of Blikstein (2011) and others, the evaluation of the programming solutions is important for providing appropriate feedback and support. For this reason, an analytics component is connected to *ctGameStudio*, in which learner data are collected to assess the students' behavior.

Table 1 shows the main features of the guided learning environment of *ctGameStudio*. It is divided in four main levels with a subdivision to focus on specific Computational Thinking competences. Instructions at the beginning and a just-in-time guidance by the mentor help to complete the levels. At each sub-level, new blocks become available to slowly introduce the learner to the tool. The functionality of the available blocks can be read at any time in a "block lexicon".

Table 1
*Level structure of the guided learning environment of ctGameStudio.*

| **Level** | **Sub-level** | **Content** | **Focus of the Computational Thinking competence** |
|---|---|---|---|
| 1 Movement | 1.1 | Move forward | Evaluation |
| | 1.2 | Sequences | Algorithmic Thinking |
| | 1.3 | Approximated circle Movement | Generalization |
| 2 Loops | 2.1 | Structured repetition | Decomposition |
| | 2.2 | Transformation in a loop | Abstractions |
| 3 Events | 3.1 | Scanning of a static object | Abstractions |
| | 3.2 | Tracking of the mentor depending on his random movement | Algorithmic Thinking |
| 4 Interaction | 4.1 | Attacking strategies against a passive opponent | Combination of all Competencies |
| | 4.2 | Defending strategies against an attacking opponent | |

Level 1.1 focuses on evaluation of system feedback. Here, the learner uses only the basic "move forward"-block. But the student can vary the parameter for the distance to reach the target. The highlighting gives also feedback about the activity and is active until the move is executed. In the following sub-levels, the set of commands was extended by "turn left/right". The students create sequences of these commands and create – in analogy to turtle geometry with LOGO – polygons (e.g. hexagons).

Level 2 sets the focus on introducing and creating iterative loops. In the first part, the students compose a sequence of move commands in the shape of stairs. Such repetitive subsequences imply the use of loop abstractions, which motivates the use of a count-controlled loop in the next sub-level.

The required abstractions for level 3 are on different layers, working with attributes of objects such as object type, and an event mechanism for scanning these objects: The learners have to distinguish object types in order to implement type-specific behavior of the virtual agent. For example, the virtual agent scans an object and dodges it if it is a bomb or collects it if it is ammunition. The last sub-level combines the latter with loop abstractions and the virtual robot tracks an object "on scan".

Level 4 helps to develop a strategy against other robots. In the first sub-level, the user has to implement an attack strategy and in the second sub-level a defense strategy. This given decomposition gives the student a structure for improving in the next steps their programming. (Parametrized) procedures and functions are introduced as another abstraction, to reduce code and brush up the structure and quality of the programming.

The open stage is a mode to develop strategies against other players. This approach creates a seamless transition from guided introductory programming to an open sandbox to foster and practice CT competences without limiting the solution space.

## 4. Evaluation

The aim of this exploratory evaluation is to better understand the learning progression with guided learning environments and to draw conclusions from the programming behavior based on action logs.

### *4.1 Experimental setting*

The experiment has been conducted in a supervised setting using the web-based learning environment (*ctGameStudio*) for capturing data of the programming tasks and learning progressions. An online survey tool has been used to assess the usability and user experience of the learning environment. In this experiment, 40 people participated (24 men; 16 women; mean age: $M = 22.23$, $SD = 3.98$). A requirement to participate was having no or a low experience in programming.

The participants had a time limit of 45 minutes to accomplish the programming tasks (seven levels of *ctGameStudio*). Each participant has been supervised, but without providing additional hints, to solve the tasks. However, we had to exclude seven cases from the data set of the exploratory analysis due to time limit violations (e.g. early quitting) and technical issues that led to data loss.

The programming tasks were situated in the first seven levels of the *ctGameStudio* environment and cover classical programming-related tasks that are known from turtle geometry (e.g. simple movement, tracing a polygon), and from event-based programming (scanning of objects, event-handling "on scan"). The final level combines both aspects in a 'follow task' that incorporates the use of scanning of a moving object in order to track it 'on scan'. Every level starts with a short story and short instructions about the new concepts to be introduced. The participants had the possibility to get additional information about the specific programming blocks and predefined methods in a "block lexicon".

### *4.2 Analysis of the learning progression*

In our experimentation, the learner artefacts, particularly the programming solutions to the specified tasks of each level, have been collected through the *ctGameStudio* environment. To characterize the

learning progression of each learner, we extracted the following features from each solution: #Runs, #Changes per run, #Creates, #Consecutive changes per create, Time spent in minutes. These features are mapped to CT competences to describe the learning progression of each user. Additionally, we subdivide the users into groups of different achievement levels and compare the performance characteristics and learning progression for these groups.

Table 2 shows the interpretation of the basic indicators in terms of CT-related skills. During the study each action was logged: changing the programming code, running the programming code and the event of completing a level. The changing events are divided in creating a block, changing the block structure by moving a block in or out of its target scope, deleting a block, and varying a parameter.

Table 2
*Interpretation of basic measures for the collected data.*

| **Indicator** | **Interpretation** |
|---|---|
| # Runs | Testing and evaluating behavior of the created programming code |
| # Changes per run | Trial and error behavior or advanced planning |
| # Creates | Active extensions of the programming solution |
| # Consecutive changes per create | Structured editing |
| Time spent in minutes | Measure for efficiency |

Figure 2 presents the distribution of runs per level for all users per sub-level as a measure for their evaluation behavior.

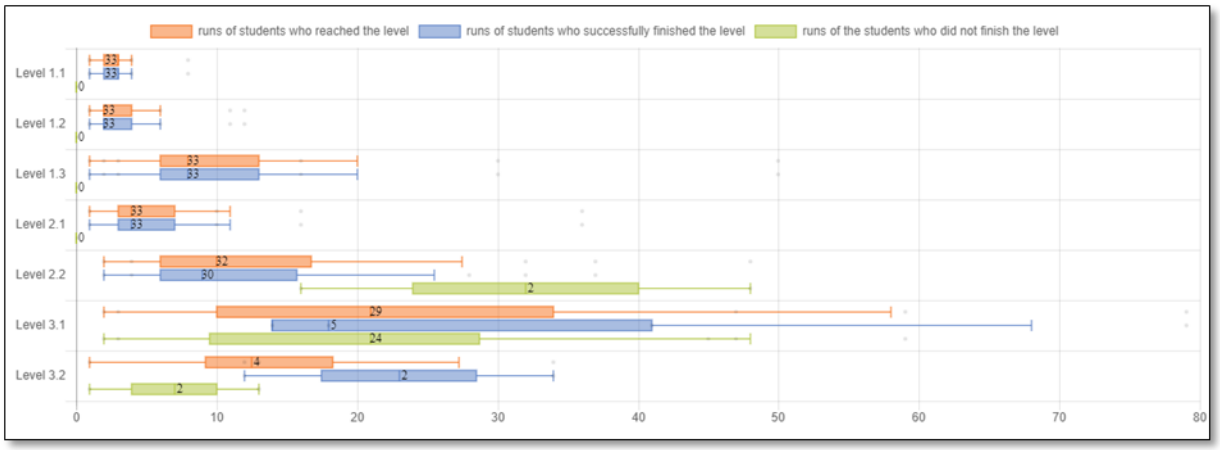


*Figure 2.* Distribution of the runs per level with the number of users.

The boxplots in Figure 2 show the distribution of runs per level for all users who reached the level (regardless of whether or not they completed the level), all users who successfully completed the level, and for all users who did reached but did not complete the level. Starting with level 2.2, not every learner has successfully completed the level, so first differences in the boxplots can be seen. Two students did not pass this level and dropped out. Of these, one user executed the code 16 times and the other one 48 times. These values are much higher than the median (nine runs) of the users who successfully finished the level. Since they could not skip the level, they used the time left to modify the programming code but did not develop any ideas to solve this level. Supportive functions are needed at this point to help these learners to find a solution.

Level 3.2 was only reached by four students and solved by two students. The figure shows that those students who completed the level needed more runs than those who did not finish the level – an indicator that those students who did not complete the level had not enough time to explore, to try out, and to solve the level. A similar structure can be seen in level 3.1.

Additionally, outliers are not included in the boxplot and shown as dots outside of the box. In the first two levels the distribution is small and symmetrical around a low median. In level 1.3 the median of executed runs increases abruptly. Also, a skewness to the right side shows that there are

some participants who need many more runs to complete the level. The preparation phase for the loop was solved, on average, in less runs. Beginning with level 2.2, different abstractions (loops, event-callback associations, structured object types) were introduced. In these cases, the median and deviation increase extremely, primarily with a skew to the right.

To further analyze the "abstraction gaps" that we started to observe at level 2.2, we want to dig more deeply into the advanced planning behavior of the students indicated by the feature #Changes per run. Figure 3 presents the values for each user and each run.

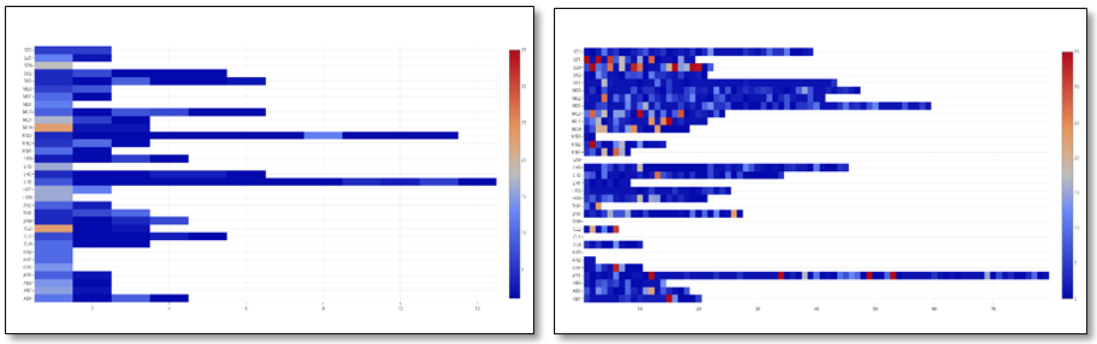

*Figure 3.* Changes per run and user for level 1.2 (left) and 3.1 (right)

As can be seen in Figure 3 and level 1.2, most changes were made in the first run and only small changes in the following runs, expect for those who needed more than six runs. These students also did not pass level 3.1. In level 3.1, the first run was done with a low number of changes and then, an unstructured pattern of changes can be observed. In addition, not everyone who needed just a few runs completed the level. This is due to the time limit of 45 minutes. It was only related to working on the whole learning environment. In each sublevel, the learners could take as much time as they needed. Consequently, a subdivision is needed that analyzes the behavior in the learning environment depending on the learning outcome.

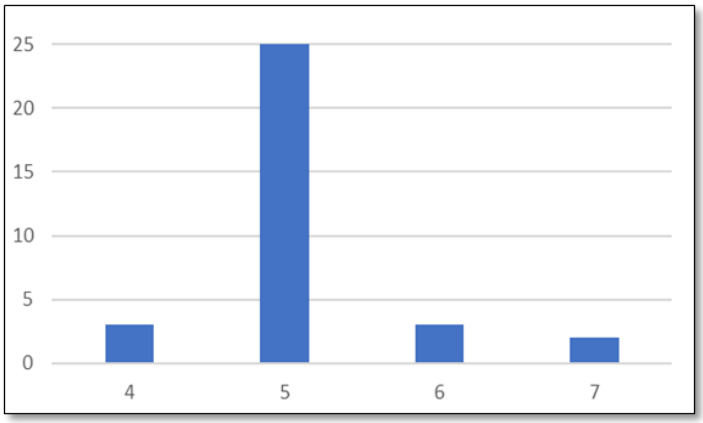


*Figure 4.* Number of students who finished *n* levels

As shown in Figure 4, most of the students finished five levels (average success), five students finished more levels (high success), and three students finished only four levels (low success). Table 3 presents all features for these three groups in level 1.2 and 2.2. In level 2.2, the loop was introduced as the first abstraction and the group with low success did not finish this level.

Table 3 shows that the high achievement group has lower values in each category (e.g., no. of runs, time spent) in comparison to the groups with average or low achievement. This indicates, that they followed a strategic approach and did not need as much exploration as the other groups. The differences are relatively small for level 1.2 but quite distinctive for level 2.2. The low achievement group needed much more runs to evaluate their changes than the other two groups. This group also created more blocks and needed almost half of the available time. I.e., time was not used efficiently, also on average, many changes were made per run and many blocks were created to try out different strategies.

Table 3
*Median and standard deviation of each feature and learner group in level 1.2 and 2.2.*

| **Items** | **Level 1.2** | ***Success level*** | | |
|---|---|---|---|---|
| | | **low** | ***average*** | ***high*** |
| # Runs | *M* | **3.3** | **3.5** | **2.2** |
| | *SD* | 1.7 | 2.8 | 0.4 |
| # Changes per run | *M* | 5.1 | 7.1 | 5.7 |
| | *SD* | 2.8 | 4.4 | 1.3 |
| # Creates | *M* | 3.0 | 4.0 | 3.2 |
| | *SD* | 0 | 1.3 | 0.4 |
| # Consecu-tive Changes per create | *M* | 2.2 | 2.3 | 1.8 |
| | *SD* | 0.6 | 0.8 | 0.6 |
| Time spent in minutes | *M* | **2.6** | **1.6** | **1.1** |
| | *SD* | 0.5 | 0.9 | 0.6 |

| **Items** | **Level 2.2** | ***Success level*** | | |
|---|---|---|---|---|
| | | **Low** | ***average*** | ***high*** |
| # Runs | *M* | **32.0** | **13.0** | **8.6** |
| | *SD* | 16 | 9.3 | 3.8 |
| # Changes per run | *M* | 5.1 | 2.8 | 2.7 |
| | *SD* | 0.9 | 2.1 | 1.0 |
| # Creates | *M* | **14.5** | **8.8** | **7.8** |
| | *SD* | 2.5 | 5.9 | 5.6 |
| # Consecutive Changes per create | *M* | 2.3 | 2.1 | 2.1 |
| | *SD* | 0.05 | 0.6 | 0.3 |
| Time spent in minutes | *M* | **19.2** | **6.6** | **4.0** |
| | *SD* | 6.4 | 5.1 | 1.9 |

The task of level 2.2 was to create a loop through which the code would be condensed and abstracted. Taking a look at the solutions of the low achieving group shows that parameters were varied in an unstructured way and superfluous code was added in/before/after the loop, which explains the high value of the feature #changes per run.

### *4.3 Usability*

The user experience has been tested using a shortened version of the UEQ questionnaire (Laugwitz, Schrepp, & Held, 2006). Some items in the test have been modified in order to be more suitable for younger participants in terms of the vocabulary used (Hinderks, Schrepp, Rauschenberger, Olschner, & Thomaschewski, 2012). The questionnaire used a 5-point Likert scale (1 = “no agreement”, 5 = “total agreement”). Table 4 shows the results of the survey. Most of these average results were above the neutral point on the scale. The participants valued the creative and novel user interface (4.10) concept as an alternative approach to programming. The dimension of stimulation was well received (3.80), which might have the reason that the learning environment contains a game setting with a visually attractive game engine so that stimulates the learners in a way that is similar to computer games.

Table 4
*User Experience of ctGameStudio regarding the results of the UEQ. (-) indicates an inverted item.*

| **Scale** | **Item** | ***M*** | ***SD*** |
|---|---|---|---|
| Stimulation | Interesting | 3.80 | 1.018 |
| Dependability | Supportive | 3.15 | 1.189 |
| Efficiency | Organized | 3.45 | 0.904 |
| Novelty | Creative | 4.10 | 0.841 |
| Perspicuity | Easy to learn | 3.10 | 1.128 |
| Perspicuity | Complicated (-) | 3.43 | 1.059 |
| Dependability | Unpredictable (-) | 2.47 | 1.176 |
| Dependability | Easy to operate | 3.68 | 0.997 |
| Dependability | Meets expectations | 3.38 | 1.170 |

### *4.4 Discussion*

As a negative aspect of the usability evaluation, participants found the environment complicated (3.43). This might be caused by the low detail in the guidance by the tutor, which is on a too general level, but also caused by a jump in the difficulty between level 2.2 and 3.1. The analysis of the learning progressions uncovered such a gap.

As can be seen on the left-hand side of Figure 5, in the first four levels the number of changes per run increases constantly. The learners become more secure in the implementation of their strategies and have more focus on the goal that they want to achieve. The runs themselves behave similarly, but in level 1.3 there is a strong upward bias. This is due to the structure of the level. Learners are supposed to create an approximate circular motion that requires many block elements. Frequent testing of the strategy to see if the parameters have been chosen correctly is essential. Level 2.1 was much easier after this level, so they needed fewer runs and the construction of the programming was easier.

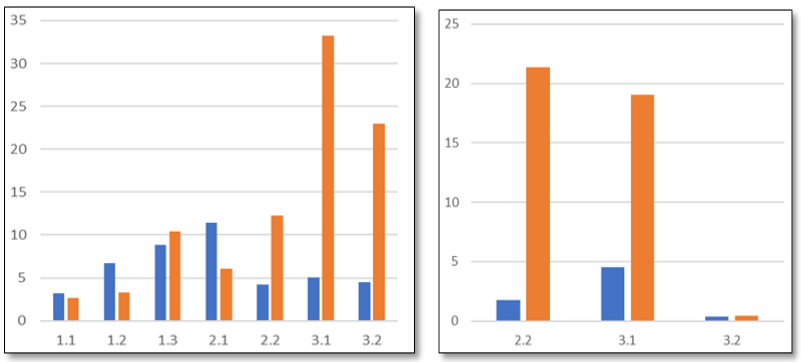


*Figure 5.* Average of the runs per level (orange) and the changes per run and level (blue) for the users who finished the level (left) and did not finished the level (right)

In contrast to the previous levels, not all of the students finished level 2.2. This level marks a break as everyone needed a lot of runs with relatively fewer changes per run. In addition, the students who could not finish the level, needed almost twice as many runs and made almost half as many changes per run (cf. Figure 5, right diagram). This suggests that the level of difficulty in introducing the abstraction "loop" is very high for students. A similar behavior can be seen in level 3.1, where the aspects "data abstraction and events" were introduced. The same applied for level 3.2, which indicates that learners need more support when abstractions are introduced.

## 5. Conclusion

In this paper, we presented a game-based learning environment designed to help and stimulate students in developing CT competences. On the one hand, we provide a guided learning mode based on levels, in which a student can get just-in-time feedback and on-demand hints with the help of a tutor (or "mentor"). On the other hand, the learner can perform "free" robot fights against other users with similar skill levels in the open stage as suggested by Bauer et al. (2017). The learners can choose an environment that corresponds to their skill level and needs. The possibility of the competitive open stage allows for open-ended challenges.

Our analysis showed that the main distinctive difficulty in the learning progression in guided mode was the introduction of loops and other abstractions in level 2.2. In particular, the low achieving students struggled with this hurdle. Grover and Basu (2017) also discovered a similar obstacle related to basic abstractions such as loops. As a consequence, we see a need for providing more worked out examples before students are asked to use the constructs actively. We expect similar issues to occur with higher level abstractions. We have been able to confirm this finding with code analysis techniques.

For the evaluation, we introduced features to measure the behavior of the students when they are programming. Each feature relates to specific CT competences. While Moreno-Leon et al. (2015) proposed measurements for specific computational concepts depending on their occurrence on different layers, the results of the features presented in this paper are based on the learner behavior in the event of difficulties with specific computational concepts. Identifying difficulty in the programming process is, according to Blikstein (2011), important "to design and allocate support resources." Therefore, we provided a set of features and metrics that identified these difficulties in

our exploratory study. These features will be a basis for further learning analytic metrics and support mechanisms to provide feedback and help especially for lower achieving students.

## Acknowledgements

We would like to thank all the people who prepared and revised previous versions of this document, especially the students of the project course and the participants of the study.

## References

Abelson, H., & DiSessa, A. A. (1986). Turtle geometry: The computer as a medium for exploring mathematics. *MIT Press.*

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.*

Blikstein, P. (2011). Using learning analytics to assess students' behavior in open-ended programming tasks. *Paper presented at the Proceedings of the 1st international conference on learning analytics and knowledge.*

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.

Corbett, A. T., Koedinger, K. R., & Anderson, J. R. (1997). Intelligent tutoring systems. *In Handbook of Human-Computer Interaction (Second Edition)*, 849-874.

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., & Pea, R. (2013). Computational Thinking in K–12. *Educational Researcher, 42*(1), 38-43.

Hinderks, A., Schrepp, M., Rauschenberger, M., Olschner, S., & Thomaschewski, J. (2012). Konstruktion eines Fragebogens für jugendliche Personen zur Messung der User Experience (Construction of a questionnaire for adolescents to measure the user experience). *Usability Professionals*, 2012, 78-83.

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

Kazimoglu, C., Kiernan, M., Bacon, L., & MacKinnon, L. (2012). Learning programming at the computational thinking level via digital game-play. *Procedia Computer Science, 9*, 522-531.

Laugwitz, B., Schrepp, M. & Held, T., (2006). Konstruktion eines Fragebogens zur Messung der User Experience von Softwareprodukten (Construction of a questionnaire to measure the user experience of software products). *In: Heinecke, A. M. & Paul, H. (Hrsg.), Mensch und Computer 2006: Mensch und Computer im Strukturwandel. München: Oldenbourg Verlag.* (S. 125-134).

McNerney, T. S. (2004). From turtles to Tangible Programming Bricks: explorations in physical language design. *Personal and Ubiquitous Computing, 8*(5), 326-337.

Moreno-León, J., Robles, G., & Román-Gonzáles, M. (2015). Dr. Scratch: automatic analysis of scratch projects to assess and foster computational thinking. *RED. Revista de Educación a Distancia, 15*(46).

Papert, S. (1980). Mindstorms: Children, computers, and powerful ideas. *Basic Books Inc.*

Papert, S. (1996). An exploration in the space of mathematics educations. *International Journal of Computers for Mathematical Learning, 1*(1), 95-123.

Pasternak, E., Fenichel, R., & Marshall, A. N. (2017). Tips for creating a block language with blockly. *Paper presented at the Blocks and Beyond Workshop (B&B)*, 2017 IEEE.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Silverman, B. (2009). Scratch: programming for all. *Communications of the ACM, 52*(11), 60-67.

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

Weintrop, D., & Wilensky, U. (2012). RoboBuilder: A program-to-play constructionist video game. *Paper presented at the Proceedings of the Constructionism 2012 Conference*. Athens, Greece.

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

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society of London A: mathematical, physical and engineering sciences, 366*(1881), 3717-3725.

Wing, J. (2014). Computational thinking benefits society. *40th Anniversary Blog of Social Issues in Computing*, 2014.