

Sondre Kvisli

Investigating how debugging tools can be used to improve programming teaching systems and ease the transition to traditional programming

Master's thesis in Informatikk

Supervisor: Trond Aalberg

August 2020

Sondre Kvisli

Investigating how debugging tools can be used to improve programming teaching systems and ease the transition to traditional programming

Master's thesis in Informatikk
Supervisor: Trond Aalberg
August 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science



Norwegian University of
Science and Technology

Abstract

This master's thesis investigates the role of debugging when teaching programming to novices. It explores if traditional debugging tools can be implemented in modern programming teaching systems to improve their effectiveness and ease the transition to traditional programming.

A block-based programming teaching system was developed from the ground up, specifically designed to conduct a quasi-experiment with a static group comparison. The results showed that traditional debugging tools does not affect the amount of tinkering, code understanding or general performance of the programmers. They also showed that the purpose of both the debugger and game graphics is to aid the programmer in understanding the execution of the program, but when available, the game graphics is preferred. This led to the conclusion that implementing a gradual transition from game graphics to a traditional debugger in teaching systems might be just as important for easing the transition to traditional programming as a gradual transition from block-based to text-based programming, as is the main focus of most research today.

Sammendrag

Denne masteroppgaven undersøker rollen til debugging i læring av programmering til nybegynnere. Den utforsker om tradisjonelle debuggingsverktøy kan implementeres i moderne programmerings-læringssystemer for å forbedre deres effekt og lettere gjøre overgangen til tradisjonell programmering.

Et blokkbasert programmerings-læringssystem ble utviklet fra grunnen, spesifikt designet for å gjennomføre et kvasieksperiment med statisk gruppesammenligning. Resultatene viste at tradisjonelle debuggingsverktøy ikke påvirket graden av «tinkering», kodeforståelsen eller den generelle ytelsen til programmererne. De viste også at hensikten med både debuggeren og spillgrafikken er å hjelpe programmereren å forstå kjøringen av programmet, men når tilgjengelig, var spillgrafikken foretrukket. Dette førte til konklusjonen om at å implementere en gradvis overgang fra spillgrafikk til en tradisjonell debugger i læringssystemer kan være like viktig for å lettere gjøre overgangen til tradisjonell programmering som en gradvis overgang fra blokkbasert til tekstbasert programmering, som er hovedfokuset til forskningen i dag

Preface

The following thesis is the resulting work of a master's thesis conducted at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway, in the period of 12.08.2019 – 03.08.2020.

I would like to thank my supervisor Trond Aalberg for inspiration and guidance throughout the project. I would also like to thank my father Harald Kvisli for valuable dialogues, perspectives and proof-reading.

Sondre Kvisli
Notodden, July 31, 2020

Table of Contents

1	INTRODUCTION	1
1.1	BACKGROUND AND MOTIVATION	1
1.2	GOALS AND RESEARCH QUESTIONS	2
1.3	METHODOLOGY	2
1.4	RESEARCH PROCESS	3
1.5	THESIS STRUCTURE	3
2	BACKGROUND THEORY AND STATE OF THE ART	4
2.1	TAXONOMY OF TEACHING SYSTEMS	4
2.1.1	<i>Expressing programs</i>	5
2.1.2	<i>Structuring Programs</i>	6
2.1.3	<i>Understanding Program Execution</i>	7
2.2	STATE OF THE ART OF TEACHING SYSTEMS	8
2.3	TRANSITIONING FROM BLOCK-BASED TEACHING SYSTEMS TO TRADITIONAL PROGRAMMING...	9
3	THE GAME	11
3.1	WHY DEVELOP FROM SCRATCH?	11
3.2	LIMITING THE SCOPE	11
3.3	DIFFICULTY AND PEDAGOGIC DESIGN	15
3.4	INSTRUCTING THE PARTICIPANTS	17
3.5	DEBUGGING TOOLS	17
3.5.1	<i>Stepping</i>	18
3.5.2	<i>Variables</i>	19
3.5.3	<i>Log</i>	19
3.5.4	<i>Grid</i>	20
3.5.5	<i>Loop iteration counter</i>	20
4	EXPERIMENT DESIGN	22
4.1	EXPERIMENT DESIGN	22
4.2	OBSERVATION	22
4.3	INTERVIEW	24
4.4	PILOT EXPERIMENT	25
4.5	EXPERIMENT EXECUTION	25
5	RESULTS	26
5.1	OBSERVATIONS	26
5.1.1	<i>Debugger tools usage</i>	26
5.1.2	<i>Comparing the performance</i>	27
5.2	INTERVIEWS	33
6	GOAL EVALUATION AND DISCUSSION	36
6.1	GOAL EVALUATION	36
6.2	FURTHER DISCUSSION	37
6.3	WEAKNESSES	37

7	CONCLUSION AND FUTURE WORK	39
7.1	FUTURE WORK.....	39
8	BIBLIOGRAPHY	40
	APPENDIX A: OBSERVATION TABLES.....	42
	DEBUGGER GROUP.....	42
	CONTROL GROUP.....	50

List of Figures

FIGURE 2.1: TAXONOMY OF TEACHING SYSTEMS (ADAPTED FROM [15])	5
FIGURE 2.2: A TYPICAL GAME LEVEL IN 7 BILLION HUMANS [20]	8
FIGURE 2.3: CODE.ORG COURSE ON LOOPS [12]	9
FIGURE 3.1: LEVEL 2 FOCUSED ON PUTTING BLOCKS INTO SEQUENCES TO AVOID OBSTACLES AND REACH THE END FLAG. THE BLOCKS AVAILABLE ARE "STEP RIGHT" AND "JUMP"	12
FIGURE 3.2: LEVEL 6 FOCUSED ON USING A WHILE LOOP TO REUSE CODE TO SOLVE LEVELS WITH A REPEATING PATTERN. AVAILABLE BLOCKS ARE "STEP RIGHT", "JUMP" AND "REPEAT"	13
FIGURE 3.3: LEVEL 11 FOCUSED ON USING NESTED LOOPS TO PICK ALL THE COINS FROM THE CHESTS BEFORE REACHING THE END FLAG WITH AS FEW BLOCKS OF CODE AS POSSIBLE. THE AVAILABLE BLOCKS ARE "STEP RIGHT", "PICK COIN" AND "REPEAT X AMOUNT OF TIMES"	14
FIGURE 3.4: LEVEL 13 INTRODUCED NO PV. NOTE THAT THE PROGRAM IS RUNNING (INDICATED BY THE GREEN MARKER IN THE CODE), BUT THE GAME CHARACTER IS INVISIBLE AND THE CURRENT NUMBER OF COINS IN THE CHESTS ARE HIDDEN	15
FIGURE 3.5: RUN AND DEBUG BUTTONS	17
FIGURE 3.6: AN OVERVIEW OF THE DIFFERENT TOOLS THE DEBUGGER OFFERED	18
FIGURE 3.7: THE STEP FORWARD AND BACKWARD BUTTONS	18
FIGURE 3.8: THE VARIABLES VIEW.....	19
FIGURE 3.9: THE LOG VIEW.....	19
FIGURE 3.10: THE GRID	20
FIGURE 3.11: LOOP ITERATION COUNTER.....	20
FIGURE 5.1: DEBUGGING TOOLS' USAGE	27
FIGURE 5.2: PERFORMANCE COMPARISON TOTAL	29
FIGURE 5.3: PERFORMANCE COMPARISON WITH PROGRAM VISUALIZATION.....	30
FIGURE 5.4: PERFORMANCE COMPARISON WITHOUT PROGRAM VISUALIZATION	31
FIGURE 5.5: PERFORMANCE COMPARISON PRE-BUILT PROGRAMS.....	32
FIGURE 5.6 EXAMPLE OF DARK LINES IN GRASS SEPARATING GRID CELLS	34
FIGURE 5.7 THE SAME AREA WITH GRID ENABLED	34

List of Tables

TABLE 3.1: OVERVIEW OF THE GAME LEVELS	16
TABLE 4.1: OBSERVATION EVENTS	23
TABLE 4.2: TABLE EXPLAINING HOW THE CODING CONCEPT PROFICIENCY WAS GRADED	24
TABLE 5.1: PERFORMANCE RESULTS TOTAL	28
TABLE 5.2: PERFORMANCE RESULTS WITH PROGRAM VISUALIZATION	29
TABLE 5.3: PERFORMANCE RESULTS WITHOUT PROGRAM VISUALIZATION.....	31
TABLE 5.4: PERFORMANCE RESULTS PRE-BUILT PROGRAMS	32

List of Abbreviations

BBP Block Based Programming

TBP Text Based Programming

VP Visual Programming

PV Program Visualization

GPPL General Purpose Programming Language

DSPL Domain Specific Programming Language

1 Introduction

This chapter presents an overview of this master's thesis. First the subject of the thesis and the motivation behind it is presented, followed by the overarching goal and research questions. Then the research method and process are described and finally the structure of the rest of the thesis is presented.

1.1 Background and Motivation

Programming courses are making their way into the primary and secondary education systems of many countries [1]. From 2014-2020, 50 countries and all 50 US states have set policies or announced efforts to offer computer science classes and the EU have put programming on its *Digital agenda for Europe*, encouraging its members to promote programming in their schools [2], [3].

The introduction of programming at such an early stage leads to challenges for both teachers, students and software [1, p. 94]. How do you most effectively teach programming to students in this age group? Much research has been done on programming teaching systems and how they should be designed to teach programming to novices in an effective and motivating manner. Popular modern teaching systems like Code.org [4] and Scratch [5] are the results of this research. They combine block-based programming (BBP) with intuitive and colorful game graphics that are controlled by the program. This combination has been proven to increase motivation and make programming more accessible for novices [6, p. 22], [7], [8]. The increasing relevance of these systems motivated this thesis' explorative work on how these systems can be improved further.

A review of popular teaching systems revealed that despite directly targeting debugging, there were a distinct lack of more advanced debugging tools like those found in traditional programming environments. By not offering debugging tools these teaching systems gave off the impression that they encouraged tinkering, that is mindlessly remixing the code and running it over and over until bugs are corrected, rather than creating a deeper understanding of the code and underlying concepts. This observation was supported by C. Kim and J. Yan in their article "Debugging in block-based programming" [9]. These observations motivated an experiment to see if implementing traditional debugging tools into teaching systems could decrease the amount of tinkering.

BBP teaching systems lowers the bar of entry for programming, but this way of programming does not scale well to more complex systems [10]. Moving from a novice to intermediate programming skill-level therefore requires transitioning to a traditional programming environment. This transition has shown to be challenging, often leaving the programmers feeling overwhelmed and with a loss of confidence in their programming

skills [11]. This motivated the research of strategies for easing this transition and this thesis will address the debugging aspect of the transition.

1.2 Goals and Research Questions

GOAL: *How can traditional debugging tools be implemented in modern teaching systems to improve them?*

The overarching goal of this thesis was to explore if and how traditional debugging tools can be implemented into modern BBP teaching systems to improve them. Two areas to improve were recognized. The first was the teaching systems' effectiveness in teaching programming and the second was aiding in the transition to traditional programming.

Research question 1: *How does the inclusion of debugger tools affect the amount of tinkering, the code understanding and the general performance of novice kids in BBP teaching systems?*

The first research question is directed at the teaching system's effectiveness. More specifically it asks if the inclusion of traditional debugging tools can decrease the amount of tinkering, increase the code understanding, and increase the performance of the programmers in general.

Research question 2: *How can traditional debugging tools be used to ease the transition from teaching systems to traditional programming?*

The second research question explores how traditional debugging tools can be used to bridge the gap between modern teaching systems and traditional programming.

1.3 Methodology

This is a quick overview of what was done in this master's thesis:

- A quasi-experiment with static group comparison was designed
- Observation tables and interview guides for data gathering were designed
- A new BBP teaching system with traditional debugging tools was developed
- Pilot testing was done
- The research questions and BBP teaching system was revised based on pilot test results
- Final experiment
- Data analysis of quantitative and qualitative data

To answer the research questions a quasi-experiment with a static group comparison was chosen to compare the results of a debugger group, with access to traditional debugging tools, and a control group. As this thesis explored untested ideas, a BBP teaching system fitting for this experiment did not yet exist. A big part of this master's thesis therefore consisted of developing a new BBP teaching system from the ground up, inspired by the design and curriculum of Code.org's K-5 computer science courses [12]. An observation

table of pre-defined events was developed to observe any differences between the groups that could help answer the research questions. A guide for semi-structured interviews was developed to try to capture qualitative data of the programmers' experience and thought processes during the experiment. Both the quantitative and qualitative data was then analyzed and compared between the groups to answer the research questions.

1.4 Research Process

This thesis started out with an initial review of the most popular BBP teaching systems to identify possible areas to improve. Personal, previous knowledge with Scratch and Code.org made these a good starting point. Code.org's catalogue of other resources combined with google searches for variants of the keyword "learn programming" was used to explore other alternative teaching systems.

The issue of tinkering was found, and the literature was consulted for previous research on this topic. The research was also reviewed for background theory on teaching systems and block-based programming in general. During this review the challenges of transitioning from BBP teaching systems to traditional programming occurred in several articles. This subject was also brought up by my supervisor as a relevant subject of research.

After the initial subjects of this thesis was set, a structured literature review was done. The search engines Google Scholar [13] and Scopus [14] was used to find relevant publications. The literature was found on a wide range of keyword, the main ones being "block-based programming", "debugging in block-based programming", "novice programming" and "visual programming". The papers were filtered on relevance by reading their abstract and number of citations to ensure it was well regarded among the community. For highly relevant papers a review of their reference list was done to discover additional relevant literature.

1.5 Thesis Structure

The remainder of this thesis is structured as following. Chapter two presents background theory of teaching systems, the state of the art of teaching systems and the current research on the transitioning from BBP teaching systems to traditional programming. Chapter three presents the BBP teaching system developed to support the quasi-experiment and explains its design process. Chapter four explains the design of the experiment, observation table and interview guide. Chapter five presents the results from the experiment followed by chapter six that discusses these results in the light of the research questions. Finally, chapter seven concludes this thesis findings and contributions and suggests areas for future work.

2 Background Theory and State of the Art

This chapter gives insight into the previous research done in the field of teaching systems and introduces terminology useful for discussing them. It first presents a taxonomy of teaching systems that highlight different aspects of the mechanics of programming. The state of the art of teaching systems are then presented followed by the current research on the challenges of transitioning from block-based programming teaching systems to traditional programming.

2.1 Taxonomy of Teaching Systems

In "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers" [15] Kelleher & Pausch identify different aspects of programming and strategies that teaching systems can implement for making them more approachable for beginners. They then present a taxonomy that categorize these strategies and gives examples of how they can be implemented. According to Kelleher and Pausch most of the existing teaching systems focuses on the mechanics of programming, that is expressing intentions to the computer through writing programs and understanding the outgoing actions of the computer [16].

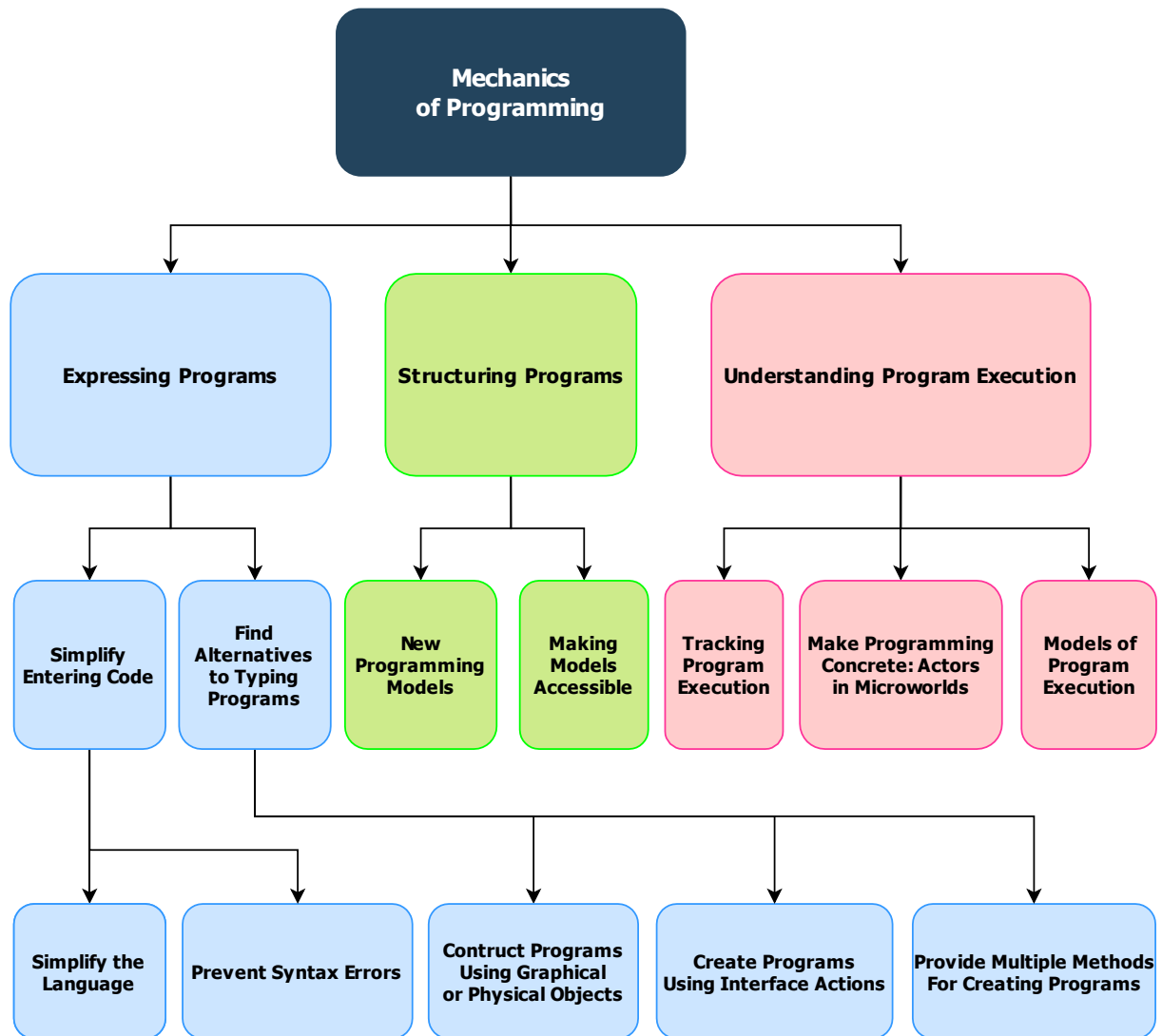


Figure 2.1: Taxonomy of Teaching Systems (adapted from [15])

They state that the mechanics of programming can be divided into three sub-categories: expressing the program, structuring the program and understanding the execution of the program. The next sections will address these categories and strategies that teaching systems have implemented to lower the barrier to programming.

2.1.1 Expressing programs

Novice programmers often find both the complexity of general purpose programming languages (GPL) and the strict syntax of traditional programming to be challenging when trying to express their intentions to the computer [15]. The taxonomy has two categories of strategies that solve this. The first is to simplify entering code. The second is to find alternatives to typing programs.

2.1.1.1 Simplify entering code

This category has two sub-categories.

Simplify the Language

With the power and flexibility of a GPPL, also comes complexity. This complexity can make novices feel overwhelmed and the expressive power it gives is unnecessary when learning basic programming concepts. This motivates the creation of simpler domain specific programming languages (DSPL) in teaching systems. J.M. Hoc identified that novices tend to look at programming like a conversation with a human, motivating a programming language more closely resembling natural language [17]. The programmers are expected to transition to GPPLs at some point and it is therefore beneficial that the DSPLs resembles the GPPLs. The goal of teaching systems that take this approach therefore is to create a DSPL that best balances simplifying the language while keeping it as close to GPPLs as possible.

Prevent Syntax Errors

Remembering intricate syntax rules and getting syntax error messages can be very frustrating for novices when trying to express their intentions. Eliminating syntax errors is therefore a major feature of teaching systems [11]. The interaction design principle of feedback [18, p. 23] states that giving feedback to the user as early as possible is important to prevent user frustration. This category offers strategies that gives immediate feedback in the form of graphical elements like shape and color or sound, as signifiers as to which commands can be combined and in what order. They can also offer templates that can be filled with parameters further abstracting the syntax.

2.1.1.2 Alternatives to typing programs

The design of this category is based on the hypothesis that writing plain text is not the optimal method for novices. There are three main alternatives. The first is using graphical objects that represent code that can be dragged and combined with other objects to create programs. The most significant advantage of this is that the programmers can recognize objects that represent what they want to express instead of having to recall how to build the statement with text. This is significant because the human brain is much better at recognizing than recalling. The second method creates a program using an interface consisting of a combination of switches and dials to manipulate the program. The third alternative combines multiple input methods. A common idea is to combine graphical elements with text-based programming thus harnessing the power of both. The programmer can then pick the alternative best suited for their needs at any time. All three alternatives to typing a program can be considered Visual Programming (VP). This style of programming is defined by Myrre as any program language that specify a program in a two- (or more) dimensional fashion [19]. This has shown to better utilize the human visual information processing systems and allows for processing data in a way that is closer to the real world, making it more intuitive for novices.

2.1.2 Structuring Programs

Teaching systems focusing on this aspect of programming try to create new programming paradigms by organizing the code in a different, more intuitive way. These techniques are more disruptive, and the major pitfall is that the eventual transition to

traditional programming can be much harder. This might be the reason why this is a less popular approach and it will not be given much attention in this thesis.

2.1.3 Understanding Program Execution

Teaching systems focusing on this aspect of programming offer tools that help the programmer understand the computer's actions when executing the program. It is common to use some form of graphical elements to make the program execution more understandable and this strategy is called Program Visualization (VP) [19]. The distinction between Visual Programming and Program Visualization is important and defined as VP is used for creating programs while PV is used to illustrate programs at runtime. Myrrs argues that programs created with VP obviously should be illustrated with graphics, thus it is more correct to use the term PV for programs created with text and graphics are used for execution visualization only. However, I find this distinction artificial and find it very useful to have two terms to separate the VP and PV elements of the same teaching systems. This thesis will therefore not use this distinction.

2.1.3.1 Tracking Program Execution

Teaching systems using this strategy offer tools that help the programmers follow the execution of programs. The scope of these tools often resembles that of a modern debugger, which is worth noting as it will be a point of discussion for later in this thesis. The systems often include some representation of the state of the program, either graphical or textual or both. When done graphically, Myrrs classifies this as data visualization. The representation will change to reflect the outcome of the instructions being executed. Myrrs separates between dynamic and static visualization. Dynamic can show animations and transitions between states while static only shows snapshots [19]. It is also common to indicate what line of code is up next for execution. The strategy of adding graphical marks to the code is called code visualization and can offer much more advanced features than just indicating a line of code.

2.1.3.2 Actors in Microworld

Teaching systems using this strategy aim to make programming more concrete by allowing the programmer to control an actor in a microworld. The actors often have a limited set of actions they can perform, making it easy to create a simplified and intuitive DSPL. A graphical simulator is often included, and it can show the actor's actions in the microworld as the program executes, often in a dynamic way.

2.1.3.3 Models of Program Execution

Teaching systems using this strategy uses physical metaphors to explain code in a more concrete and intuitive way. This is best explained with an example. In the game "7 billion people" [20] data registers are replaced with green boxes called "data cubes" as seen in Figure 2.2 below.

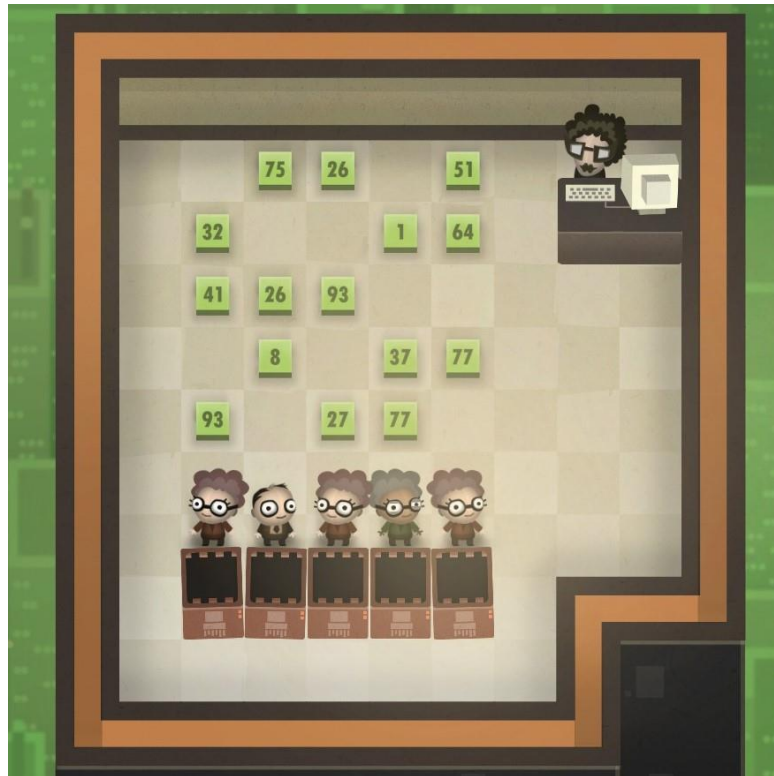


Figure 2.2: A typical game level in 7 billion Humans [20]

The data cubes can store values between zero and hundred and the programmer controls a set of workers that can act upon these cubes. The actions include picking them up, moving them, reading, and writing to them and performing arithmetic computations with their values. If the programmer wants to delete a register he instructs one of the workers to bring the data cube over to a shredding machine to destroy the block. The game replaces abstract concepts of data storage with physical objects that can be manipulated in physical space.

2.2 State of the Art of Teaching Systems

The most popular modern teaching systems incorporate multiple strategies to lower the barriers for novice programmers discussed in the taxonomy. They use a Visual Programming paradigm called Block-Based Programming (BBP). BBP uses draggable blocks representing code that can be picked up from a palette, placed in the workspace and pieced together to create programs. The blocks use color and shape to categorize them and interlocks when compatible with each other, preventing syntax errors [11]. Some of the blocks also act as templates, allowing the programmer to change parameters via drop-down menus or input fields, increasing the expressive power while maintaining the simplified syntax. The programming languages are simplified DSPLs that describe actions an actor can do in a microworld [11], [15]. The languages often contain flow control blocks that resemble those found in GPPLs like loops, conditionals and event handlers. This allows for learning basic flow control in a beginner friendly environment. Both the actions of the actor and the state of the microworld are dynamically simulated

and some code visualization is included in the form of highlighting the current line of code being executed.



Figure 2.3: Code.org course on loops [12]

One could say that these systems are the pinnacle of teaching systems based on, and evolved from, knowledge gathered over the last decades. They have been proven successful in motivating and inspiring novices to program. But in the pursuit of lowering the barrier of entry, the gap from BBP games to traditional programming has become wide, and the transition has proven to be difficult. This has created a need for new strategies to bridge the gap.

2.3 Transitioning from Block-Based Teaching Systems to Traditional Programming

BBP teaching systems have mostly been shown to be more effective at teaching novices the basic concepts of programming than text-based systems [21], however an important part of teaching systems is that the knowledge gained will prepare the programmers for more advanced programming [8]. Research have proven this not to be the case, documenting programmers feeling overwhelmed by "syntax overload", loss of confidence and development of misconceptions and bad habits [11]. In a quasi-experiment, D. Wintrop found that students that learned programming in a BBP environment showed greater learning gains, but that this difference quickly faded when transitioning to traditional programming, arguing that better strategies and tools for making the transition are needed [8].

A good amount of attention has already been given to easing the transition from BBP to TBP component of the transition. Bi-directional systems like the Droplet editor [22] found in Code.org Labs [23] and the BlockPy editor [24] found in EduBlocks [25] offer block based versions of GPPLs with seamless switching between a block- and text-based representation of the program. These types of systems are a promising intermediate step between existing block-based environments and textual languages according to L. Moors [11].

Another approach is dual-modality, which instead of separating the block- and textual-representations of the program, tries to combine the benefits of both with the use of frames that separates each code-statement into a indivisible unit, helping with syntax errors and keeping track of the scope, but keeping much of the freedom of TBP [26]. Despite much research on the area there are plenty of room for improvement according to Moors [11].

A common trait of the aforementioned attempts at easing the transition is that they focus on the challenges of expressing the program. This thesis will therefore explore challenges of understanding the program execution and how novices can transition from program visualization to a traditional debugger.

3 The Game

This chapter will first explain the reasoning behind developing a game from scratch to conduct the experiment. It will then explain how the game was designed including deciding on a scope, level design, difficulty curve and how instructions were communicated to the participants. Finally, the different debugging tools implemented are discussed in detail. A huge amount of time and effort was put into creating the game to make the experiment possible. The full game is available at <https://master.d33fy60y53bq5n.amplifyapp.com/> .

3.1 Why develop from scratch?

To conduct the experiment for this thesis a game was needed that fulfilled the following criteria:

1. Feature Virtual Programming in the form of Block Based Programming
2. Feature Program Visualization in the form of Actors in a Microworld
3. Feature a simple to learn DSPL
4. Feature Debugging Tools
5. Be highly flexible. Feature ability to toggle functionality like PV and debugging tools on and off for different testing groups.
6. Runs on low end hardware with no time-consuming installation processes.

Several existing solutions like Code.org, Scratch, CodeCombat [27], 7 Billion Humans [20] were evaluated and attempted adapted to the experiment to save time and resources. However, the conclusion was that none of these solutions were satisfactory and it was decided to create a custom game from the ground up.

Using a library for the BBP aspect of the game, like Google's Blockly [28], was considered to save development time. However there were two main reasons for not adding it.

1. The cost of limited flexibility of the block's design and function was assessed as too high compared to the amount of time it would save.
2. The limited scope of the game meant only a small part of the library was going to be used, making it hard to justify spending time familiarizing with a library.

Based on these decisions the game was made from the ground up in JavaScript with the game engine Phaser 3 [29].

3.2 Limiting the Scope

Inspired by Code.org's computer science curriculum for Grades K-5, five fundamental programming concepts were identified:

- Sequences
- Events
- Loops
- Conditionals
- Debugging

Given that the test subjects had no prior programming experience and given a desire to keep the duration of a single test limited to less than one hour, the scope of the game had to be limited. The game was therefore limited to the three concepts of sequences, debugging and loops. As stringing code together in sequences is the most fundamental concept of programming this was a natural starting point for the earliest game levels.

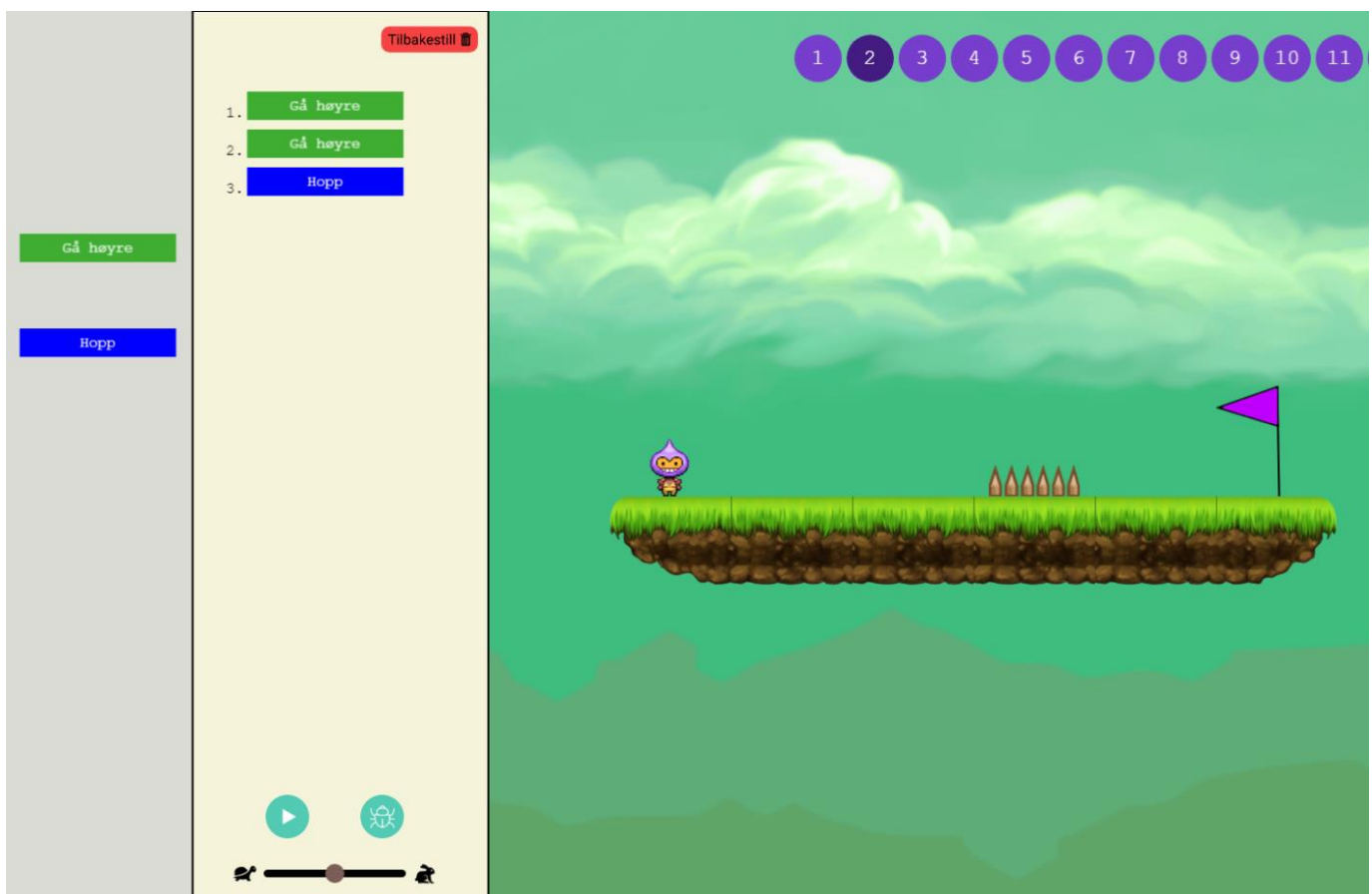


Figure 3.1: Level 2 focused on putting blocks into sequences to avoid obstacles and reach the end flag. The blocks available are "step right" and "jump"

Debugging would hopefully give useful insight into if there were any difference between the debugger- and control group regarding debugging existing code. Finally, the concept of loops was chosen as it gives great opportunities for increasing the difficulty of the levels from basic infinite loops, to for-loops to nested for-loops. This would also give flexibility in scaling the difficulty on a per level basis as the more difficult levels could be solved either optimally with nested loops or in a more verbose way with several single loops for lower performing students.

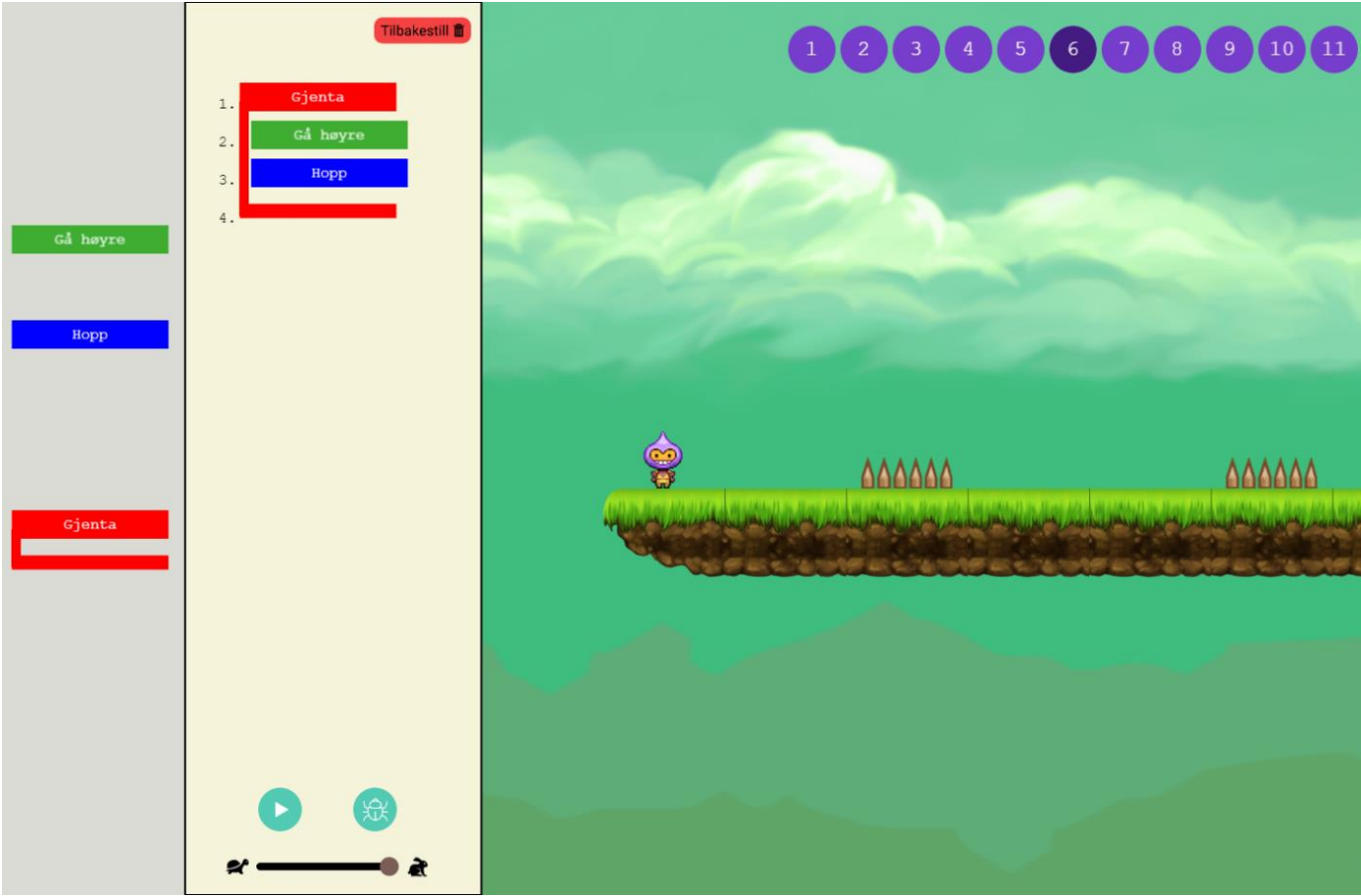


Figure 3.2: level 6 focused on using a while loop to reuse code to solve levels with a repeating pattern. Available blocks are "step right", "jump" and "repeat"

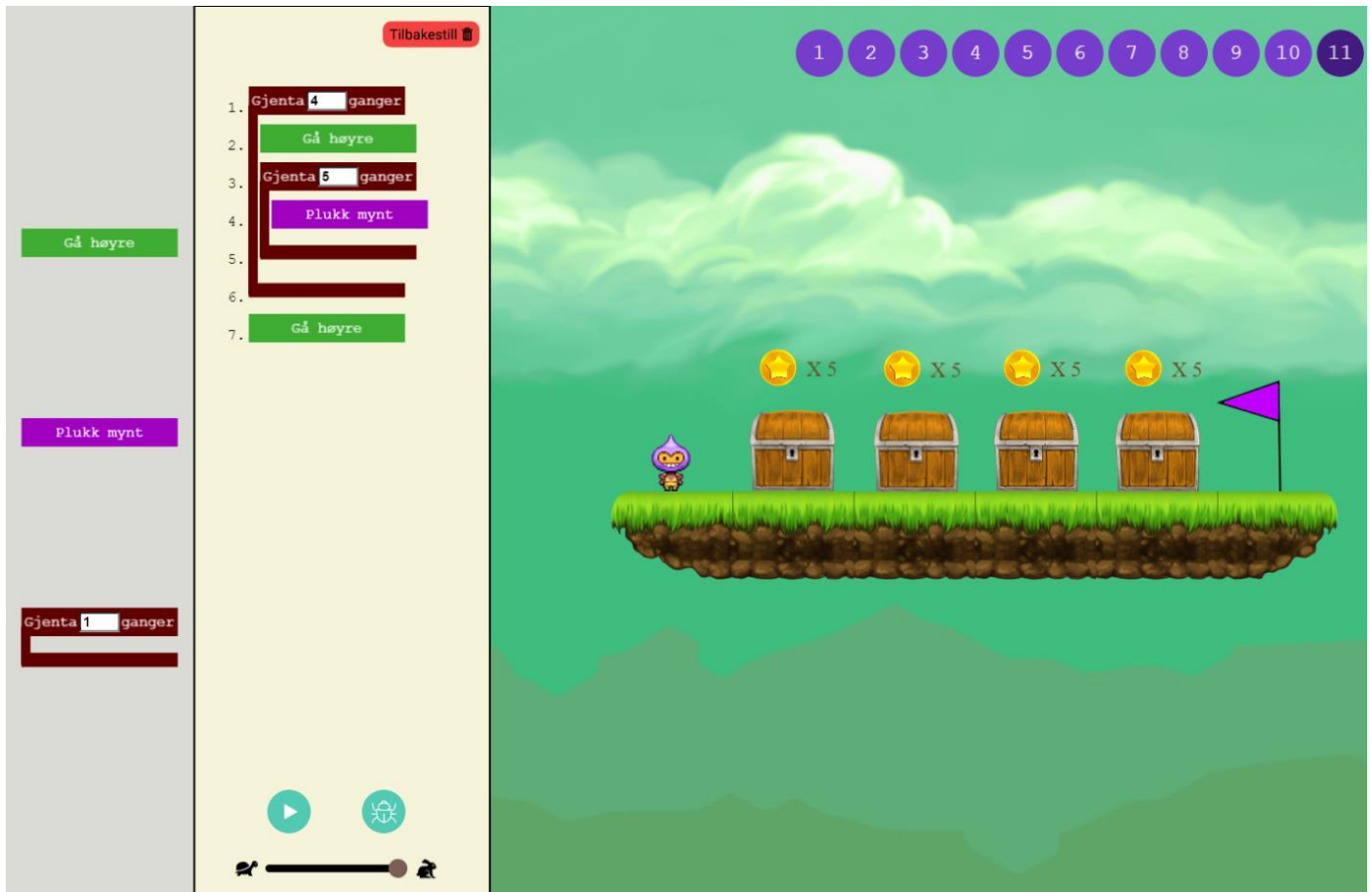


Figure 3.3: level 11 focused on using nested loops to pick all the coins from the chests before reaching the end flag with as few blocks of code as possible. The available blocks are "step right", "pick coin" and "repeat X amount of times"

3.3 Difficulty and Pedagogic Design

To teach programming in a pedagogic and effective manner the courses C, D and E of Code.org Computer Science Fundamentals for Elementary School [12] was used to get inspiration for exercises and to calibrate the difficulty curve. The fifteen game levels designed can be divided into five topics, each topic roughly following the same structure. First a new concept is introduced, and the programmer must use this in an elementary way to solve a level. Then the same concept must be used to solve a more difficult level. Finally, a level with a pre-built program containing errors and/or incompleteness is presented to the programmer and it must be debugged and corrected to complete the level.

Four of the topics were picked based on the three fundamental coding concepts discussed in the previous chapter. The fifth topic of no program visualization was added as a result of pilot testing of the game.

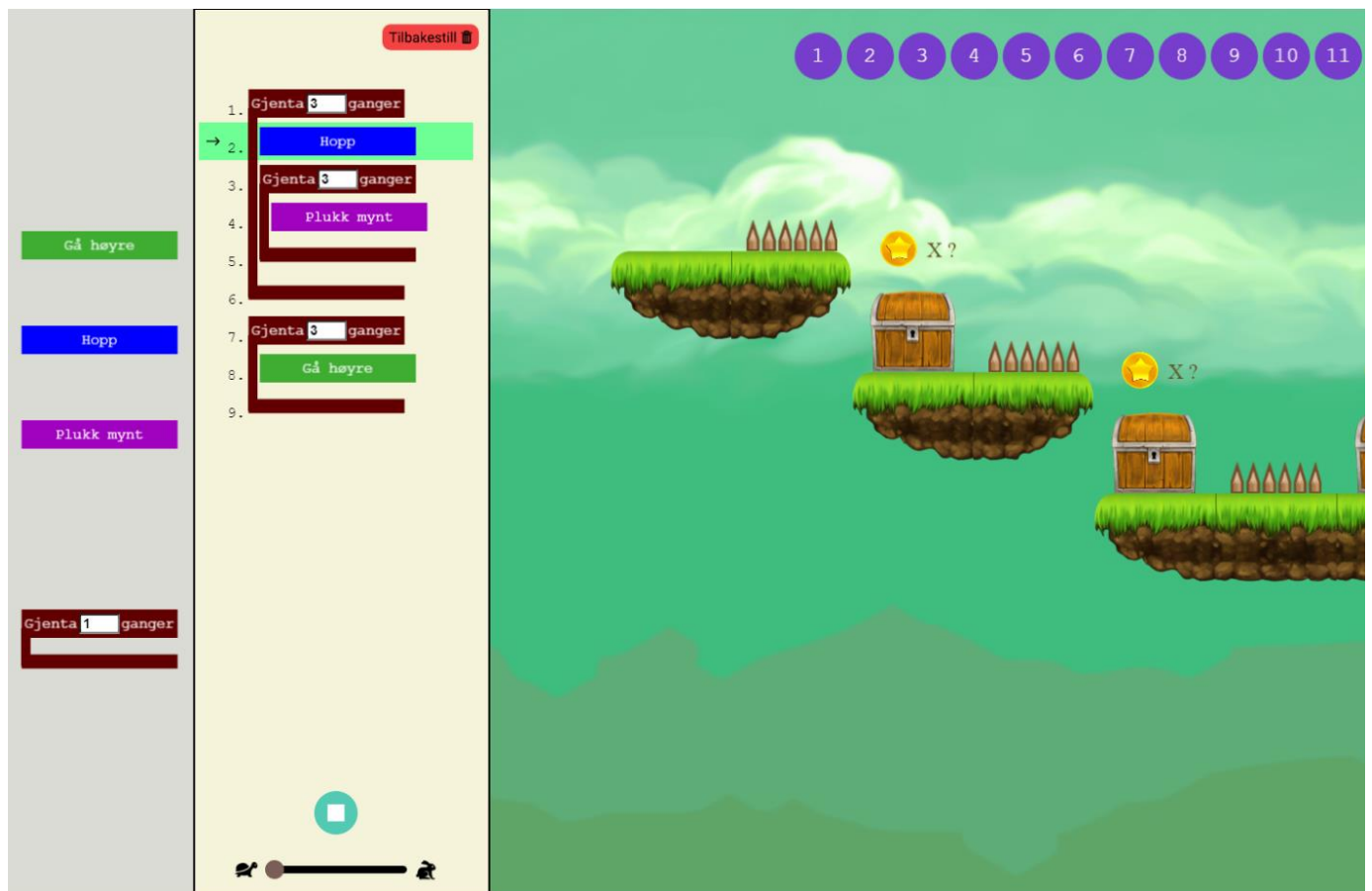


Figure 3.4: level 13 introduced no PV. Note that the program is running (indicated by the green marker in the code), but the game character is invisible and the current number of coins in the chests are hidden

The levels built upon each other, so the programmer had to combine previous knowledge with the new concepts to complete the levels. This was especially true for the more

difficult levels of each topic. An overview of all the game levels, with the five topics and what the programmer was learning is presented in the table below.

Lvl	Pre-built program	PV	Topic	What the programmer is learning
1	No	Yes	sequences	a) Add a block to the program and run it
2	No	Yes		b) Add multiple blocks into a program to create sequences
3	Yes	Yes		c) Move the game character with code d) Avoid world obstacles like spikes e) Complete levels by reaching the end flag
4	No	Yes	While-loops	f) Repeat parts of the code with loops
5	No	Yes		
6	No	Yes		
7	Yes	Yes		
8	No	Yes	For-loops	a) Set custom number of iterations for a loop
9	Yes	Yes		b) Use multiple loops in one program (non-nested)
10	No	Yes	Nested for-loops	a) Pick up coins from chests
11	No	Yes		b) Use nested loops to repeat code to efficiently pick up coins from multiple chests
12	Yes	Yes		
13	No	No	No Program Visualization	a) Use all previous knowledge but with no PV
14	No	No		
15	Yes	No		

Table 3.1: Overview of the game levels

3.4 Instructing the Participants

The game was designed to be as intuitive as possible, however for a person who have never seen a piece of computer code in their lives, some basics were necessary to explain. The instructions were standardized and spoken orally to the participants for each level. Using text was considered and would be the best option if the participants were programming on their own. However, some participants expressed anxiousness of programming in front of another person for the first time, in fear of being judged. The choice of giving instructions orally was done to create a dialog and try to create a more relaxed and comfortable testing environment for the participant to gather the most representative data.

Before the participants started programming on their own, a brief introduction to BBP was given. This included how to create programs by dragging and dropping blocks, how to start the program, how a computer program executes each line of code after the other and how the program represents instructions to the game character on how to act in the mini world. Finally, the overarching goal of moving the game character to the end flag with as few lines of code as possible was explained. The debugger group also got an introduction to the debugging tools, how to use stepping, read variables and use the other tools explained in chapter 3.5 Debugging Tools.

On each new level the participants also got a brief introduction. If the level introduced new blocks their basic usage was explained. If the level had a pre-built program the participants were instructed to try to find errors and fix the program. When reaching the levels with no PV, the participants were told that they would not be able to watch the character and number of coins when running the program and that they should try to cope with this as best as possible. The instructions were carefully designed not to lead the participants to use the debugging tools in any way.

3.5 Debugging Tools

The debugging tools implemented in the game were similar to those found in traditional programming environments, but with a few modifications. The debugging group had access to a debugging button placed adjacent to the run button that would make the game enter debugging mode. Having a separate mode made it easier to recognize when participants were actively using its features.



Figure 3.5: Run and debug buttons

The debugging tools implemented were forward and backward stepping in code execution, a variable view, a log, a grid and a loop iteration counter. A screenshot of

these tools is shown in **Error! Reference source not found.** below. The tools were available to the debugging group throughout all of the fifteen game levels.

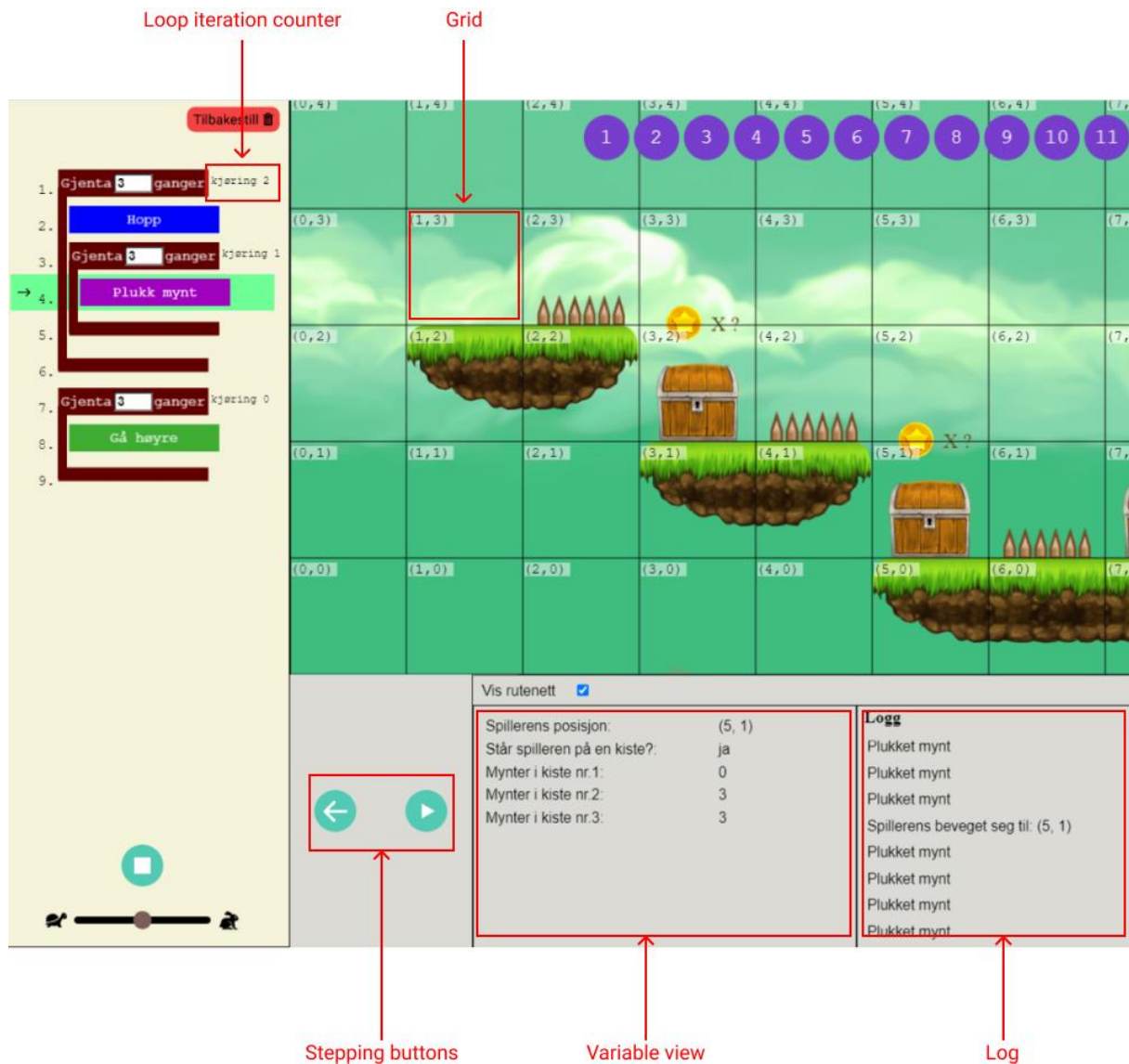


Figure 3.6: An overview of the different tools the debugger offered

3.5.1 Stepping

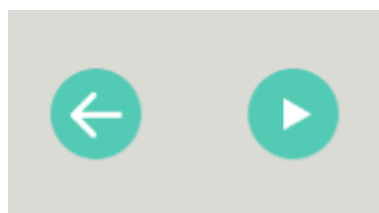
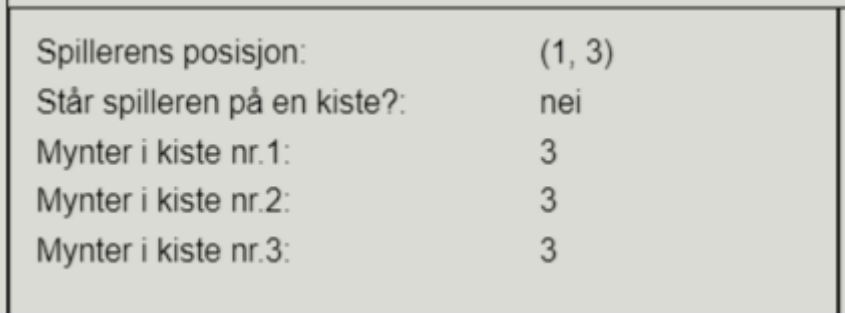


Figure 3.7: The step forward and backward buttons

This tool allows the programmer to step through the execution of the program one code line at the time. While more traditional debuggers only allow for stepping forward from a specified point, the simplified nature of the micro world makes it feasible to store a copy of the state for each instruction and move between them. The rationale behind implementing this tool was to help the programmers easier identify what exactly went wrong by stepping back and forth around areas of interest. The buttons' icons were designed differently to communicate that forward button was playing the next line while back acted more like an undo button. Implementing break-points was considered, but was omitted to save on development time. The programs created never got very long, so although break-points could have been convenient, stepping from the start each time was a satisfactory user experience.

3.5.2 Variables



Spillerens posisjon:	(1, 3)
Står spilleren på en kiste?:	nei
Mynter i kiste nr.1:	3
Mynter i kiste nr.2:	3
Mynter i kiste nr.3:	3

Figure 3.8: The variables view

The variable view shows the current value of variables at a specific time in the execution of the program. The view included the players x , y position, if a player was standing on a chest or not and the current number of coins in each chest in the level.

3.5.3 Log



Logg	Tom
Spillerens beveget seg til: (3, 2)	
Plukket mynt	
Plukket mynt	
Plukket mynt	
Spillerens beveget seg til: (5, 1)	

Figure 3.9: The log view

The log prints out every major action the game character does, like moving to a new location and picking up coins. The actions are listed in a chronological scrollable list. The log works similarly to manually printing events to the console in more traditional programming. Although it is not strictly a part of the debugger it is a valid debug strategy deemed worthy of being included in the game. A separate print block was

considered, giving the programmer freedom of what and when to log things, but it was decided that this would unnecessary complicate the process.

3.5.4 Grid



Figure 3.10: The grid

The grid can be activated by toggling a checkbox in the debugger menu. It adds a grid overlay to the game world with x, y coordinates associated with each cell. Together with the player's x, y coordinates this can be used to locate the game character's position in the game world.

3.5.5 Loop iteration counter



Figure 3.11: Loop iteration counter

The loop iteration counter is a form of code visualization that keeps track of how many iterations a loop has done at a specific time during the program execution. The counter was placed next to the associated loop block to easily identify the associated block. In Figure 3.11 the counters indicate that the outer loop is on its 2nd iteration and the inner loop on its 1st.

4 Experiment design

This chapter describes the design of the experiment and how observations and interviews were used to gather data to best answer the research questions. It also describes the pilot tests that were done and how the experiment was revised based on the results of these. Finally, this chapter describes how the final experiment was conducted.

4.1 Experiment design

Originally the participants of the experiment were going to be students from a fifth grade in elementary school from Notodden, Norway, randomly assigned to the two test groups. This would ensure a representative selection of the population and the randomness would ensure an approximately equal skill between the groups. Due to the COVID-19 pandemic and closed schools, this was not feasible. The participants were therefore chosen at random from adults with minimal to no previous experience with programming. The effects of this is further discussed in **Error! Reference source not found. Error! Reference source not found.** The pilot tests showed that age and previous experience with computer games dramatically impacted the results. This combined with the relatively small number of participants made it clear that a random selection was not the optimal way to create two groups of equal skill. The criteria of an age between 20-30 years, some experience with games and an academic background of more than three years was therefore added to keep the factors constant. The participants were distributed among the two test groups with an even distribution among sex and academic background.

4.2 Observation

The observation process can be divided into two parts. The first part was a highly systematic observation of pre-defined events that gathered quantitative data. The events were chosen by their ability to measure the participants' performance in light of the research questions. The specific events and the reasoning behind them are shown in the table below.

	Event	What it measures	Data type
<i>Both groups</i>	The program is ran	A high number can indicate that the participant is tinkering. A low number can indicate that the participant understands the underlying concepts well enough to predict the outcome	Number
	A block is deleted	Same as above	Number
	Deaths	Same as above	Number
	Level completed	The participant showed enough coding proficiency to solve the level	Boolean
	Level completed with optimal solution	The participant showed enough coding proficiency to solve the level AND can use the concepts in an optimal way to create short and concise programs	Boolean
	Time used	A performance metric. Can also indicate which levels were more difficult than others	Time (mm:ss)
	Coding concept proficiency grade	A grade given to the participant based on their understanding of the code and the concepts being taught (for more details see grading table below)	Grade from 1-5
<i>Debugger group only</i>	Debug mode is entered	How useful the combination of all the debugging tools was	Number
	The grid is used	How useful this specific tool was	Number
	The variables are read	Same as above	Number
	The log is read	Same as above	Number
	Step forward is used	Same as above	Number
	Step backwards is used	Same as above	Number
	Loop's times ran indicator is read	Same as above	Number

Table 4.1: Observation events

Coding concept proficiency grading

1. No proficiency	The test subject showed no understanding of the code and made it look like they tinkered their way to a working program. No understanding of the underlying coding concepts.
2. Low level of proficiency	The test subject showed some understanding of the code, but many parts were unclear. Low understanding of underlying concepts.
3. Medium level of proficiency	The test subject showed understanding of large parts of the code, but some parts were unclear. The participants could use the basic concepts.
4. High level of proficiency	The participant could explain most of the code and use the underlying concepts. Some advanced concepts like nested loops were unclear.
5. Full level of proficiency	The test subject could explain all of the code and created optimal solutions based on the underlying concepts.

Table 4.2: table explaining how the coding concept proficiency was graded

Originally the number of times a participant asked for help and number of times they got completely stuck on a level was going to be observed. The first metric was dropped because it was discovered to be more dependent on personality traits than performance. The second metric was dropped because it never occurred. An approach where the game was programmed to automatically log all quantitative data was considered, but some of the metrics had some nuances to them that needed human evaluation, like evaluating if a program was optimal or not and grading the coding concept proficiency.

The second part of the observation was a less systematic one. It consisted of reviewing the recordings and mapping patterns and behavior of the participants, creating qualitative data. This strategy was implemented to capture results that was difficult to predict and emerged during testing.

4.3 Interview

After completing the game, the participants were interviewed in a semi-structured manner to get further insight into their experience. Both groups were asked the following questions:

- How was your general experience with the game?
- How did the removal of the game graphics affect you?

In addition, the debugging group was asked the following questions:

- What was your general experience with the debugging tools?
- When and how did you use the grid?
- When and how did you use the variable view?
- When and how did you use the log?

- When and how did you use the step forward and backward tools?
- When and how did you use the indicator for how many times a loop had ran?

4.4 Pilot experiment

Before the initial pilot tests the main focus on this thesis was research question 1: How does the inclusion of debugger tools affect the amount of tinkering, the code understanding and the general performance of novice kids in BBP teaching systems? After a prototype of the game was developed there were conducted two pilot tests where both participants had access to the debugging tools. The results showed that the tools were minimally or not used at all. The literature was consulted, and a hypothesis that the results could be explained by an overlap in functionality of the debugger and game graphics was formed. Both the debugger and game graphics serve the purpose of helping the programmer understand the program execution, only that the graphics does so in a more intuitive manner. This makes the debugging tools redundant. To test out this hypothesis three levels with no program visualization was added to the game (level 13-15). If this resulted in an increase in the debugging tool usage it would prove that there was indeed an overlap between the two.

4.5 Experiment execution

The experiments were executed by videocall. The test-subjects accessed a website hosting the game and shared their screen so that the researcher could observe their interaction with the game. The videocalls were recorded, giving a record of both audio and video of the experiments. These were later used as the main input for the data analysis. The test subjects were encouraged to vocalize their thought processes to give the researcher insight. They were also instructed to create the shortest programs possible (i.e., fewest lines of code). This was to encourage the use of loops and clean code. Examples of unclean code are the use of loops with only one iteration and using more lines of code than necessary to guarantee that the character moves at least as far as the end flag.

5 Results

This chapter presents the results from the experiment. First the debugging tools' usage, how often the different tools were used and on what type of levels, are presented. Then the results from the observation of pre-defined events are used to compare the performance between the debugging group and control group. After this the results from the interviews are presented, shedding light on the experience of the debugging group

5.1 Observations

The records of the videocalls were used as the source for data analysis. The records were played back and occurrences of actions of interest were logged in an individual observation table for each participant. The data consisted mostly of discrete ratio data, with a few sets of ordinal data and was organized and analyzed in an Excel spreadsheet.

The experiment originally had seven participants in each test group. The debugger group had one extreme outlier that in some cases scored values four times the median and were therefore excluded from further statistical analyses.

5.1.1 Debugger tools usage

As the pilot tests showed minimal usage of the debugging tools, this was an aspect of interest to analyze after the final experiment. These results are presented first to give context to the rest of the results.

To be able to compare the relative usage between different types of levels, their average usage per level was used. The average was calculated by dividing the number of uses by the number of levels of a specific type. For example the debug mode was entered seven times across eight normal levels with PV, which results in:

$$\frac{7 \text{ uses}}{8 \text{ levels}} = 0,875 \text{ uses per level}$$

The chart below displays the usage results for the different types of levels.

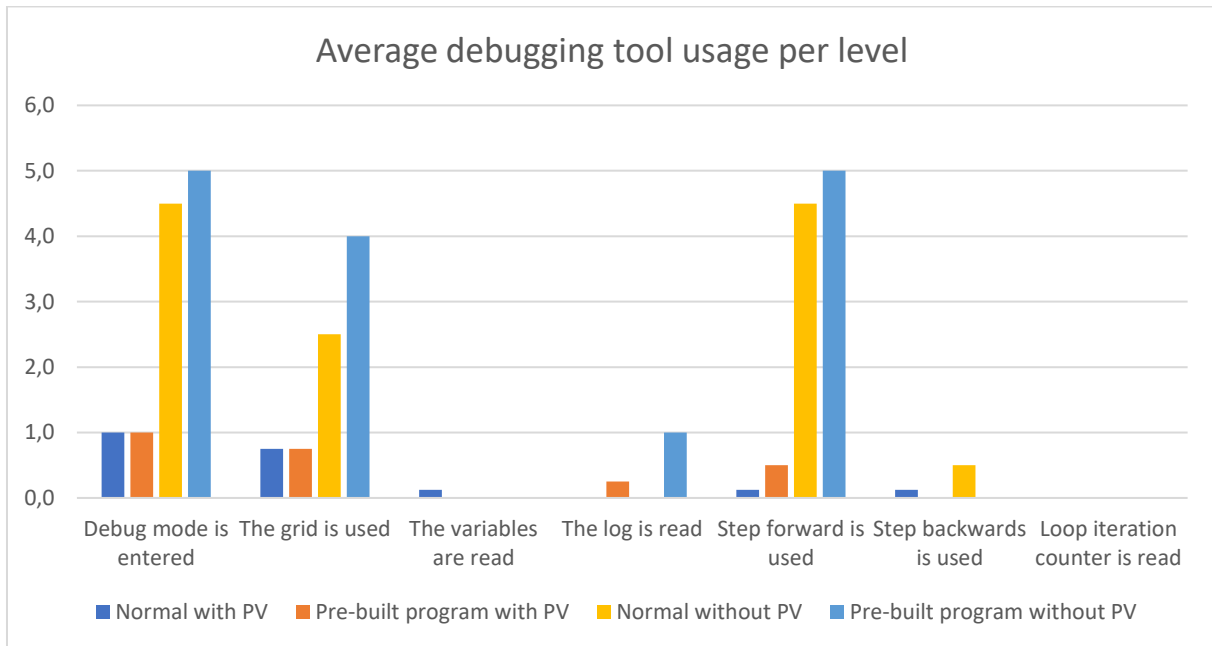


Figure 5.1: Debugging tools' usage

The results show that the debugging tools were used very sparingly during levels with PV, both in normal and pre-built program levels. The tools most used on these levels was the grid. When PV was removed there was an increase in the debugging tools' usage by 4,5 times for normal levels and 5 times for pre-built program levels. The most used tools were step forward followed by the grid.

5.1.2 Comparing the performance

The performance metrics were compared between the debugger and control group for different types of levels. As a result of the removal of an outlier the two groups had a different number of participants and the results are therefore given as an average for the group. The differences were evaluated with the Student's T-Test for statistical significance in Excel, using two-tail distribution and two-sample equal variance. Significant scores of $p < 0,05$ are highlighted in green in the tables. Also recall that the coding concept proficiency grade is a scale from one to five, where five is the highest score. The grade is therefore shown as an average across the levels to be easier to understand.

5.1.2.1 Total

The results for all the levels combined are presented in the table below.

	Avg. use per person		T-test
	<i>Debugger</i>	<i>Control</i>	
The program was ran	29,0	21,4	0,116
A block was deleted	15,0	9,4	0,027
Deaths	9,3	3,3	0,020
Lvls completed	15,0	15,0	
Lvls completed with optimal solution	11,2	13,4	0,095
Time used	20:30	16:04	0,036
Coding concept proficiency grade	4,4 (65,8)	4,8 (72,0)	0,089

Table 5.1: Performance results total

There was a noticeable difference between the two groups. Firstly, the debugger group scored worse in every metric except levels completed, which was the same. A block is deleted, number of deaths and time used showed a statistically significant difference with $p < 0,05$.

The diagram below shows the relative performance of the debugger group compared to the control group, where the control group's score is 100%. This makes it possible to compare the performance of different metrics, despite having different types of values, like number of occurrences, time and proficiency grade. It also accounts for a difference in metrics if a higher or lower score is better.

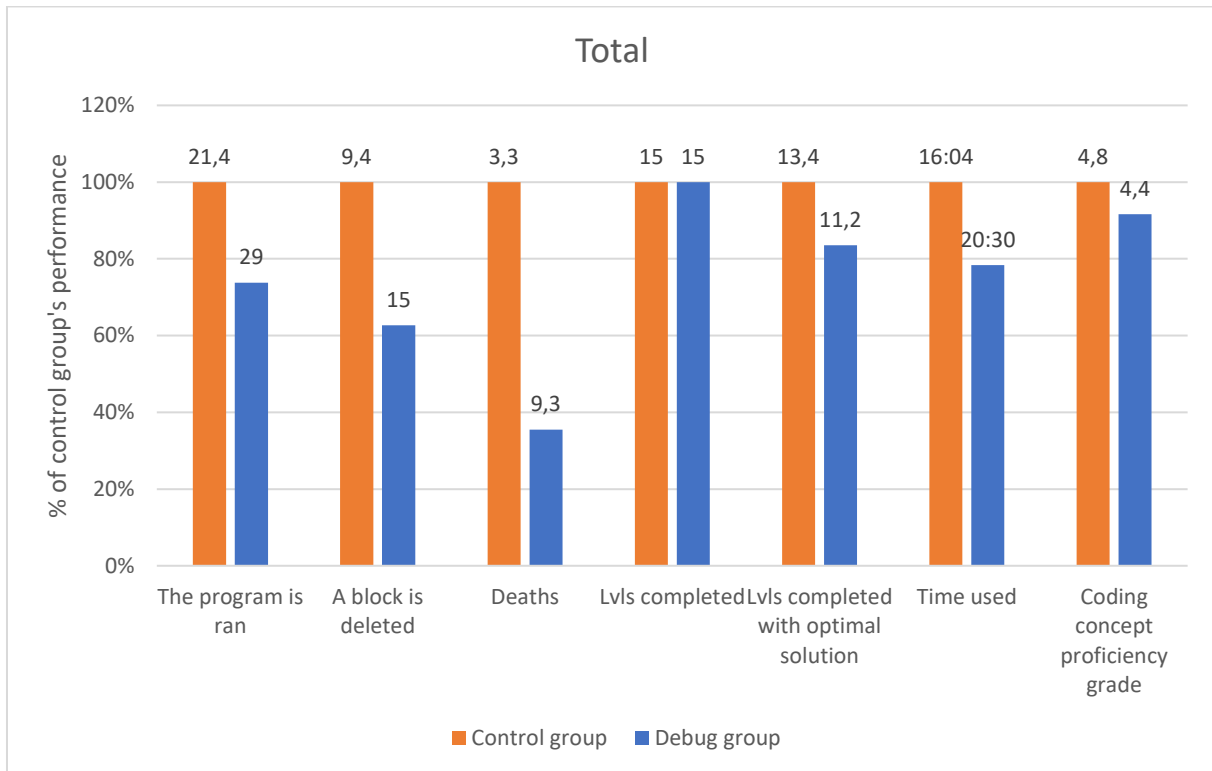


Figure 5.2: Performance comparison total

5.1.2.2 With Program Visualization

The table below shows the results from all the levels with PV, both normal and debugging.

	Avg. use per person		T-test
	Debugger	Control	
The program was ran	23	16,4	0,097
A block was deleted	11	6,7	0,046
Deaths	7	2	0,018
Lvls completed	12	12	
Lvls completed with optimal solution	8,8	10,7	0,095
Time used	12:39:10	9:53:26	0,170
Coding concept proficiency grade	4,4	4,8	0,117

Table 5.2: Performance results with Program Visualization

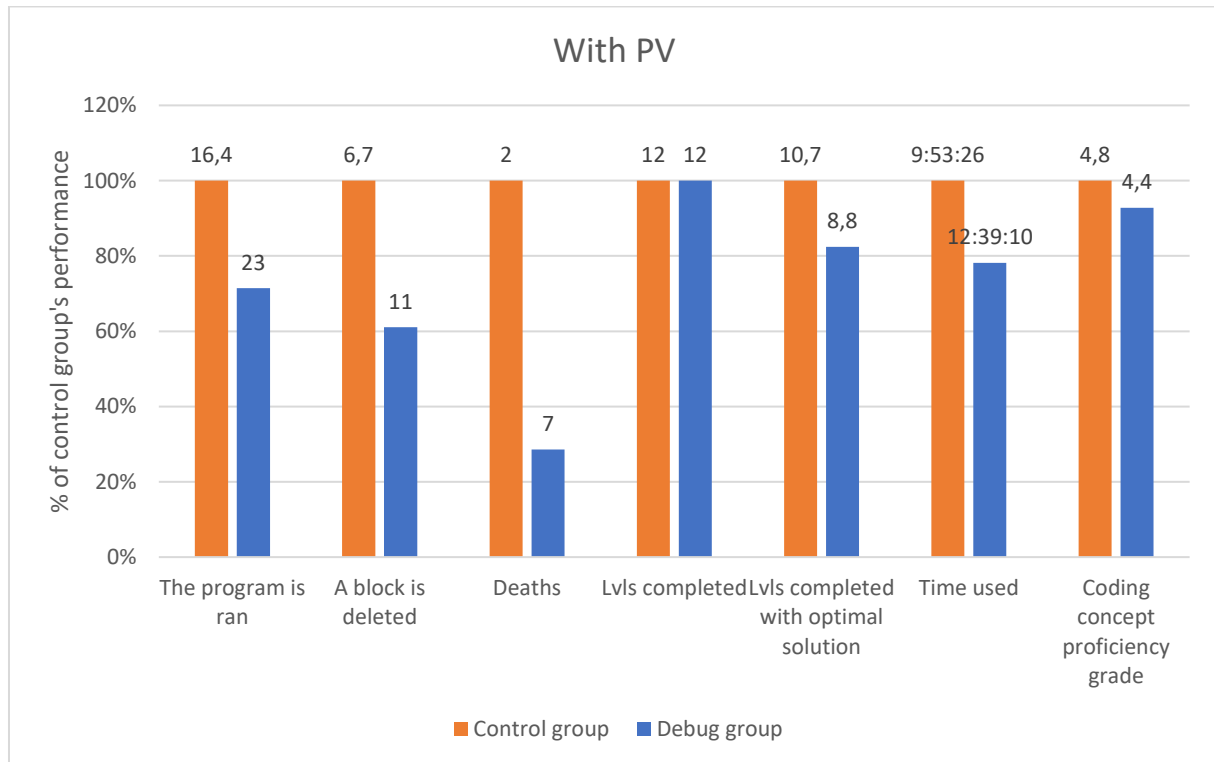


Figure 5.3: Performance comparison with Program Visualization

The results are very similar to the results from the total. Given the knowledge that the debugging tools were barely used during these levels, we can quite confidently conclude that the differences seen between the groups are not because of the debugging tools, but another factor. This is more discussed in chapter **Error! Reference source not found.. Error! Reference source not found..** The T-test shows that the difference in time used no longer is statistically significant, even though the relative performance is similar to the results from the total at 80% of the control group's. This is because of a higher standard deviation in the results.

5.1.2.3 Without Program Visualization

The table below shows the results from all the levels without PV, both normal and with pre-built programs.

	Avg. use per person		T-test
	Debugger	Control	
The program was ran	6	5	0,398
A block was deleted	4	2,7	0,280
Deaths	2,3	1,3	0,198
Lvls completed	3	3	
Lvls completed with optimal solution	2,3	2,7	0,320
Time used	7:50:50	6:10:17	0,037
Coding concept proficiency grade	4,2	4,9	0,082

Table 5.3: Performance results without Program Visualization

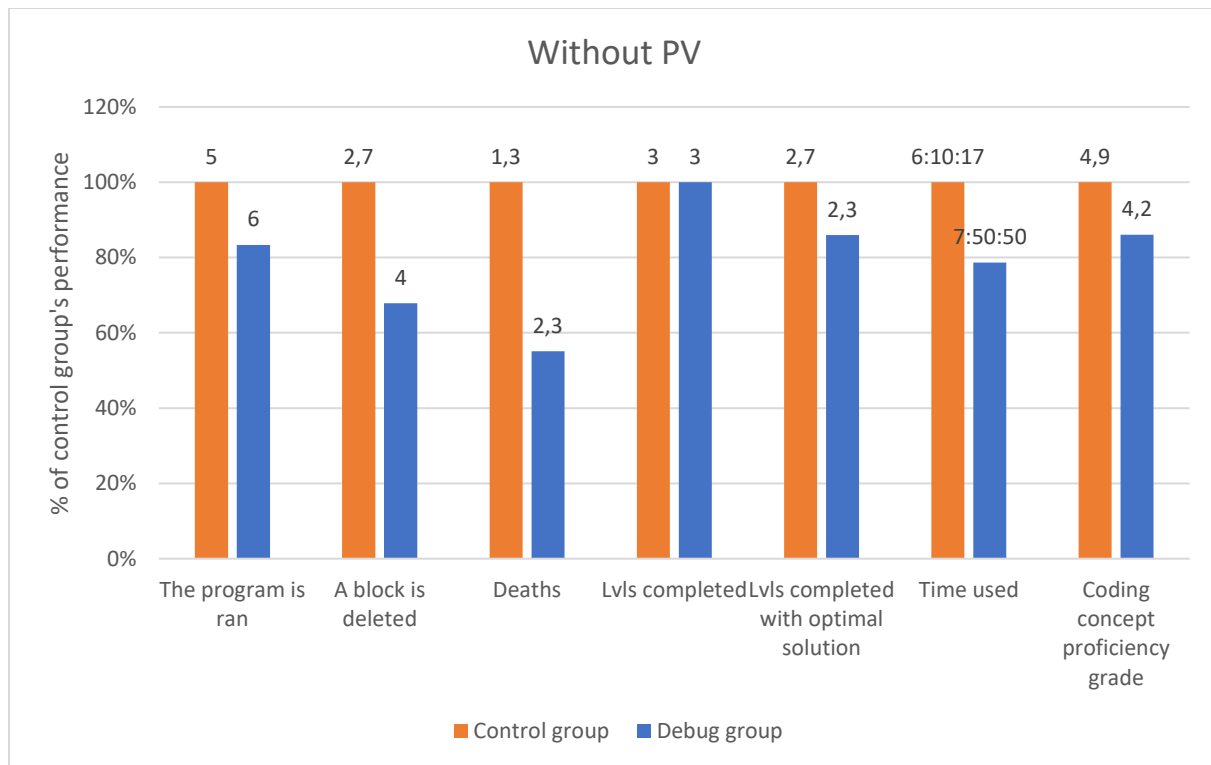


Figure 5.4: Performance comparison without Program Visualization

The results very much follow the same pattern as both the total and with PV. These levels had a good amount of debugger tool usage, further supporting the conclusion that the observed differences in performance are not due to access to debugging tools.

5.1.2.4 Pre-built program levels

As shown in the previous results there seemed to be no relative difference in performance between the levels with and without PV. The results of all the pre-built program levels, both with and without PV, are therefore combined and compared between the two group which can be seen in the table below.

	Avg. use per person		T-test
	Debugger	Control	
The program is ran	12,17	6,71	0,215
A block is deleted	7,83	4,71	0,378
Deaths	5,83	1,57	0,009
Lvls completed	5,00	5,00	
Lvls completed with optimal solution	3,83	4,57	0,037
Time used	8:52:00	5:02:00	0,407

Coding concept proficiency grade	21,00	24,14	0,0002
-----------------------------------------	-------	-------	--------

Table 5.4: Performance results pre-built programs

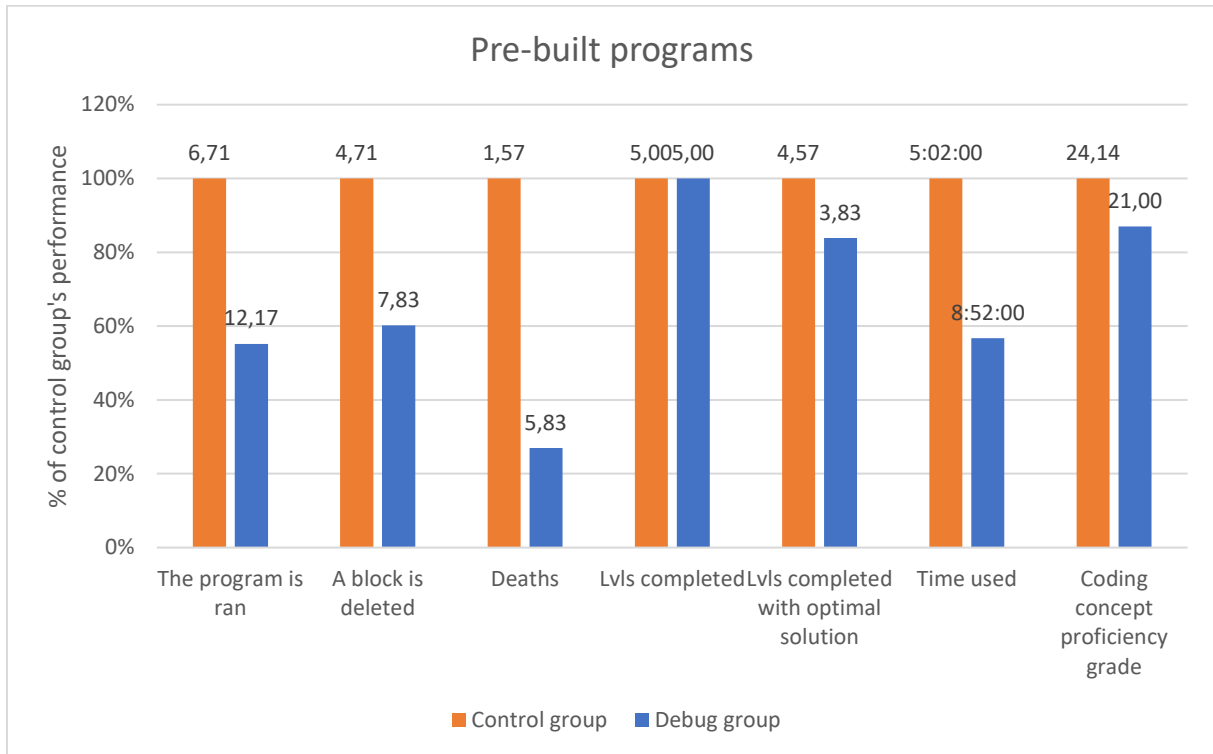


Figure 5.5: Performance comparison pre-built programs

In the pre-built programs levels, the patterns are the same as previously but the debugger group falls even further behind the control group.

5.2 Interviews

The interviews were transcribed from the video recordings and organized in Word. Each question was then analyzed for emerging themes and patterns in the answers. The participant excluded from the quantitative analysis was included to give valuable insight into how a person who struggled used the debugging tools. The most common themes from each question are presented, with some additional comments of particular interest.

General Experience

The answers from the interviews match the results from the observations to a large degree. Five of seven participants answered that they used the debugger more when the program visualization was removed. Especially the tools that directly replaced the game graphics, like the grid and the variables players x,y-coordinates and number of coins in the chests. Several of the participants stated that they forgot that the debugging tools existed, and only when the PV was removed did they recall them. One participant never used the debugger and argued that "it is easier to understand what is happening watching the graphics than reading it in text". This person performed well above average on all levels and did not struggle when the PV was removed.

One participant tried to use the debugging tools but found them confusing and quickly discarded them. The person said the tools gave him the feeling of information overload which only derailed his train of thought and it was easier just to ignore them. This participant scored lower than the average and struggled on the levels with no PV.

The participant that was the outlier in the performance analysis used the debugging tools extensively and stated that "I would not stand a chance on the invisible levels (no PV) without the grid, variable view and the arrow showing which line of code the program was on". This participant seemed more interested in investing time into the game and learning to use the debugger. The person also created programs by adding a few number of blocks and running them often to investigate their effect. The person said this was a conscious strategy to not get confused. This explains the low scores on both time and other performance metrics.

Grid

There were two use cases for the grid. The most frequently occurring was to use the characters x, y coordinates to locate the character in the world on the last three levels with no PV. The other more unexpected use case was to easier visualize the movement grid. Some of the participants struggled to recognize the dark lines in the grass indicating one cell or unit in the grid. By enabling the grid, the cell borders were easier to identify.



Figure 5.6 Example of dark lines in grass separating grid cells

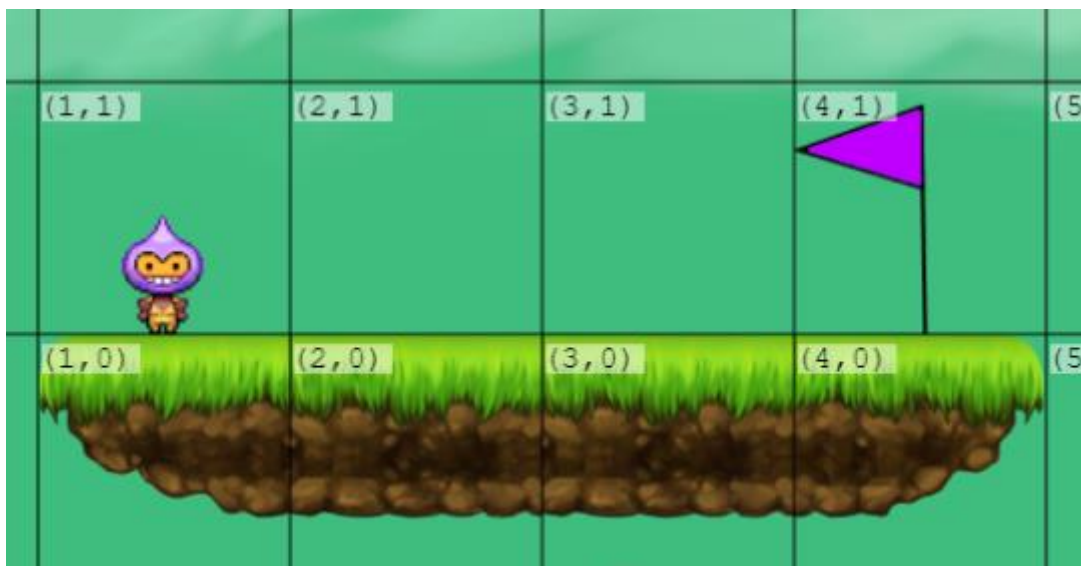


Figure 5.7 The same area with grid enabled

This should be considered a design flaw of the program where aesthetics was prioritized over function.

Variable view

The participants that used the variable view used it on levels with no PV to get the coordinates of the player and the amount of coins left in the chests to try to imagine where the player was in the program execution.

Log

The log had overlapping use-cases as the variable view.

Step forward and backward

Two participants answered that they used the step forward and backward when they did not immediately understand the reason for the programs failing. One participant mentioned a particularly tricky part in level 14 where a new movement pattern is introduced, but the execution is invisible, making the results difficult to predict. One participant answered that he used the step forward to move slowly towards the problem area of the program, but never saw the need to step back when it was located. One participant tried the stepping but said it was confusing and preferred to run the program normally.

Loop iteration counter

Six of seven participants stated that they never noticed this tool. One participant used the tool and the use case was that the person was stepping forward and did not understand why the character did not move all the way to the goal flag. The participant then discovered the iteration counter and could conclude that the program was not finished, and she continued to step forward.

6 Goal Evaluation and Discussion

This chapter answers the research questions in light of the results from the experiment and discusses other relevant topics that emerged. It then discusses weaknesses recognized with the experiment.

6.1 Goal Evaluation

Research question 1: *How does the inclusion of debugger tools affect the amount of tinkering, the code understanding and the general performance of novice kids in BBP teaching systems?*

The results from the experiment show that implementing traditional debugging tools into BBP teaching systems is not a good way of neither decreasing the amount of tinkering, increasing code understanding or increasing the general performance of novice. The main reason for this is that BBP teaching systems already offer superior tools for understanding the program execution. The program visualization offered by the dynamic simulation of the actor and the microworld is shown by both the observations and the interviews to be preferred over the traditional debugging tools, making the debugging tools redundant.

Research question 2: *How can traditional debugging tools be used to ease the transition from teaching systems to traditional programming?*

The removal of program visualization dramatically increased the usage of the debugging tools, indicating that there is an overlap of the purpose of the debugging tools and the PV. They are both tools that aid the programmer in understanding the execution of the program and the transitioning from PV to debugger tools in the context of transitioning to a traditional programming environment is therefore interesting. This thesis argues that this transition is just as important as the transition from BBP to TBP and should therefore be given a similar focus.

The transition from PV to debugging tools could be approached in a similar way to the transition from BBP to TBP, either with a bi-directional or dual-modality approach. In practice this would mean offering the programmer to switch between PV and a traditional debugger to describe the execution of the program for a bi-directional approach. For a dual-modality some elements could be described with PV, like a game character and other more specific elements could be described by a debugger, like a characters statistic, inventory etc. A third option were the PV is gradually replaced by debugging tools are also possible.

6.2 Further discussion

To look at the bigger picture of the transition to traditional programming, the feeling of being overwhelmed described by novices is most likely a result of being introduced to too many new concepts at once. In addition to the transition from BBP to TBP, and PV to debugging tools, the programmers also transition from a domain specific programming language to a general purpose programming language. A strategy to decrease the feeling of being overwhelmed would be to isolate each of these aspects, and tackle one at a time. For example one programming environment with TBP and PV, like the existing Code Monkey [30] and CodeBattle [27], another programming environment with BBP and debugging tools and one with BBP-based GPPL with PV. After the novices have gained experience with each of these new aspect of programming on their own, they can be combined as a final step towards traditional programming.

6.3 Weaknesses

Internal validity

In chapter 5.1.2 Comparing the performance, the conclusion was made that because the difference in performance between the groups was seen even when the debugging tools were barely used, the difference must be caused by another factor. This poor internal validity is most likely caused by an uneven distribution of the participants, making the control group better performing from the start. This was attempted accounted for by using the differences seen on these levels as a baseline to see if the difference increased or decreased on other types of levels.

Too old participants

As a result of the COVID-19 pandemic, the experiment was not conducted on a 5th grade in primary schools as it was designed for, but older people between 20-30 years old. The consequence was that the difficulty of the levels was generally easy for the participants. This is believed to be one of the reasons for why the debugging group scored lower than the control group in the levels with no PV. None of the participant in the control group found these levels especially challenging and a common characteristic of the best performing participants was their ability to run the programs in their heads before executing them. This allowed them to not rely on the PV and as a consequence they were not very affected by its removal. The generally weaker performing participants in the debugging group was overall worse at running the programs in their heads and were more affected by the removal of PV, despite the access to debugger tools. If harder game levels were implemented, requiring longer more complex programs to solve them, having the debugger tools might have been a bigger advantage.

Inaccurate data

Observing every move the test subject made was difficult, even with video recordings of the experiments available. Values that needed the user's interaction like mouse clicks were easy to catch, but others were the subject were glancing at information were hard to recognize. A possible improvement to the accuracy of this would be to implement eye tracking. The subjects were encouraged to speak their mind and vocalize their thought process and every move. In practice this was unreliable because when the levels got

harder, the participant needed to focus their attention at solving the level and forgot to describe their actions.

The interviews were not perfectly accurate either. Due to the length and mental demand of the experiments, the process became kind of a blur for the participants. There were a few instances of participants claiming not to use some debugging tools, when the videotape clearly showed otherwise. Combining the result from both the observations and the interviews was therefore critical to obtain a best effort representation of what actually happened.

7 Conclusion and future work

The experiment done in this thesis has shown that implementing traditional debugging tools into modern block-based programming teaching systems is not a good way to decrease the tinkering, increase code understanding or increase general performance. In fact, the debugging tools were minimally used because these teaching systems already offer program visualization, in the form of an actor in a microworld, that explain the program execution in a more intuitive and concrete way than the debugging tools.

The removal of program visualization dramatically increased the usage of the debugging tools, indicating that they serve a similar role of aiding in understanding the execution of the program. The transitioning from PV to debugger tools in the context of transitioning to a traditional programming environment is therefore believed to contribute to the feelings of being overwhelmed, expressed by novice programmers making the transition to traditional programming. This is argued to be a just as important reason for the challenges as the transition from block-based programming to text-based programming and should therefore be given a similar amount of focus in further research.

7.1 Future work

Firstly, the hypothesis that a more gradual transition from PV to debugging tools will help mitigate the challenges and negative effects seen with programmers making the transition to traditional programming should be investigated. A possible research strategy for this is a case study following two intro classes in programming where one of them are using teaching system that gradually exposes them to debugging tools. The two groups should then do exercises in traditional programming, measuring any differences in motivation and frustration, code understanding and performance due to the different teaching approaches.

If this study supports the hypothesis, research and experimentation on how the debugging tools best can be implemented and introduced in teaching systems should be done.

Investigating if isolating different aspects of the transition to traditional programming can be beneficial could also be done.

8 Bibliography

- [1] European Commission/EACEA/Eurydice, "Digital education at School in Europe Eurydice Report," 2019.
- [2] "Coding - the 21st century skill | Shaping Europe's digital future," *European Commission*, 2018. [Online]. Available: <https://ec.europa.eu/digital-single-market/en/coding-21st-century-skill>. [Accessed: 31-Jul-2020].
- [3] "Code.org 2019 Annual Report | Code.org." [Online]. Available: <https://code.org/about/2019>. [Accessed: 01-Aug-2020].
- [4] "Code.org," 2020. [Online]. Available: <https://code.org/>. [Accessed: 22-Jun-2020].
- [5] "Scratch," 2020. [Online]. Available: <https://scratch.mit.edu/>. [Accessed: 20-Jul-2020].
- [6] F. García-Peñalvo, "TACCLE 3, O5: An overview of the most relevant literature on coding and computational thinking with emphasis on the relevant issues for teachers KA2 project " TACCLE 3 – Coding " (2015-1-BE02-KA201-012307)," p. 2016, 2016.
- [7] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari, "From scratch to 'Real' programming," *ACM Trans. Comput. Educ.*, vol. 14, no. 4, pp. 1–15, Dec. 2015.
- [8] D. Weintrop and U. Wilensky, "Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms," *Comput. Educ.*, vol. 142, p. 103646, Dec. 2019.
- [9] C. M. Kim, J. Yuan, L. Vasconcelos, M. Shin, and R. B. Hill, "Debugging during block-based programming," *Instr. Sci.*, vol. 46, no. 5, pp. 767–787, 2018.
- [10] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, and S. Yang, "Scaling Up Visual Programming Languages," *Computer*, vol. 28, no. 3. pp. 45–54, 1995.
- [11] L. Moors, A. Luxton-Reilly, and P. Denny, "Transitioning from Block-Based to Text-Based Programming Languages," in *Proceedings - 2018 6th International Conference on Learning and Teaching in Computing and Engineering, LaTICE 2018*, 2018, pp. 57–64.
- [12] "Computer Science Curriculum for Grades K-5 | Code.org," 2020. [Online]. Available: <https://code.org/student/elementary>. [Accessed: 07-May-2020].
- [13] "Google Scholar," 2020. [Online]. Available: <https://scholar.google.com/>. [Accessed: 17-Jul-2020].
- [14] "Scopus," 2020. [Online]. Available: <https://www.scopus.com>. [Accessed: 17-Jul-2020].
- [15] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys*, vol. 37, no. 2. pp. 83–137, Jun-2005.
- [16] N. S. Anderson, D. A. Norman, and S. W. Draper, "User Centered System Design: New Perspectives on Human-Computer Interaction," *Am. J. Psychol.*, vol. 101, no.

- 1, p. 148, 1988.
- [17] J. Hoc, "psychology of programming thinking." 2014.
- [18] D. Norman, *The Design of Everyday Things*. 2016.
- [19] B. A. Myrre-T", "Taxonomies of Visual Programming and Program Visualization*," 1990.
- [20] "Tomorrow Corporation : 7 Billion Humans," 2020. [Online]. Available: <https://tomorrowcorporation.com/7billionhumans>. [Accessed: 08-May-2020].
- [21] D. Weintrop and U. Wilensky, "Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs," in *ICER 2015 - Proceedings of the 2015 ACM Conference on International Computing Education Research*, 2015, pp. 101–110.
- [22] "Droplet." [Online]. Available: <http://droplet-editor.github.io/>. [Accessed: 31-Jul-2020].
- [23] "App Lab | Code.org," 2020. [Online]. Available: <https://code.org/educate/applab>. [Accessed: 30-Jun-2020].
- [24] "BlockPy." [Online]. Available: <https://think.cs.vt.edu/blockpy/>. [Accessed: 31-Jul-2020].
- [25] "EduBlocks," 2020. [Online]. Available: <https://edublocks.org/index.html>. [Accessed: 30-Jun-2020].
- [26] N. C. C. Brown, A. Altadmri, and M. Kolling, "Frame-based editing: Combining the best of blocks and text programming," in *Proceedings - 2016 International Conference on Learning and Teaching in Computing and Engineering, LaTICE 2016*, 2016, pp. 47–53.
- [27] "CodeCombat," 2020. [Online]. Available: <https://codecombat.com/>. [Accessed: 08-May-2020].
- [28] "Blockly | Google Developers," 2020. [Online]. Available: <https://developers.google.com/blockly>. [Accessed: 08-May-2020].
- [29] "Phaser 3." [Online]. Available: <https://phaser.io/phaser3>. [Accessed: 08-May-2020].
- [30] "CodeMonkey." [Online]. Available: <https://www.codemonkey.com/partner-no-tell-forlag/>. [Accessed: 01-Aug-2020].

Appendix A: Observation tables

Debugger group

	Event	lv1	lv2	lv3	lv4	lv5	lv6	lv7	lv8	lv9	lv10	lv11	lv12	lv13	lv14	lv15	
Both groups	The program is ran	1	2	3	3	2	3	2	4	3	2	7	4	6	2	1	
	A block is deleted			6	2		1	1		3				2	1	1	
	Deaths			1			1			1		2	3	1	1		
	Participant asked for help	1		3				1									
	Participant is completely stuck																
	Completed	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
	Completed with optimal solution	T	T	T	T	T	T	T	T	F	T	F	F	F	F	F	T
	Time used	0:50	1:00	2:44	1:23	0:36	1:26	0:41	1:59	2:19	0:43	5:20	6:11	17:49	9:11	9:09	
	Coding concept proficiency grade	4	4	5	5	5	5	5	5	5	5	5	4	3	5	4	5
Debugger group only	The program is debugged													3	6	2	
	The grid is used													2	1	2	
	The variables are read													2			
	The log is read													4	4	2	
	Step forward is used													4	5	1	
	Step backwards is used																
	Loop's times ran indicator is read													1			

Control group

	Event	lvl1	lvl2	lvl3	lvl4	lvl5	lvl6	lvl7	lvl8	lvl9	lvl10	lvl11	lvl12	lvl13	lvl14	lvl15	
Both groups	The program is ran	1	1	2	2	4	1	1	1	1	1	3	2	1	2	2	
	A block is deleted			1				1		1		3					
	Deaths			1								1	1		1		
	Participant asked for help																
	Participant is completely stuck																
	Completed	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
	Completed with optimal solution	T	T	T	T	F	T	T	F	F	T	T	T	T	T	F	T
	Time used	0:15	0:20	0:25	0:22	1:49	0:53	0:20	0:26	0:52	0:20	1:56	0:29	1:20	1:05	0:38	
	Coding concept proficiency grade	5	5	5	5	1	5	5	4	3	5	4	4	5	4	4	

	Event	lvl1	lvl2	lvl3	lvl4	lvl5	lvl6	lvl7	lvl8	lvl9	lvl10	lvl11	lvl12	lvl13	lvl14	lvl15	
Both groups	The program is ran	2	1	1	1	1	1	1	2	1	1	1	1	2	3	1	
	A block is deleted			1				1		1			2		3	2	
	Deaths														1		
	Participant asked for help																
	Participant is completely stuck																
	Completed	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
	Completed with optimal solution	T	T	T	T	T	T	T	T	T	T	F	T	T	T	T	
	Time used	0:49	0:18	0:32	0:28	0:25	0:22	0:22	0:22	0:44	0:12	0:45	1:02	2:01	4:27	1:21	
	Coding concept proficiency grade	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	

