



Dipl. Ing. Roxane Koitz, BSc

# **Formula Composition and Manipulation in Educational Programming Languages for Children and Teenagers**

## **MASTER'S THESIS**

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

**Graz University of Technology**

Supervisor

Univ.-Prof. Dipl.-Ing. Dr. techn. Wolfgang Slany

Institute for Software Technology

Graz, March 2016

## **AFFIDAVIT**

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

---

Date

---

Signature

## Abstract

Computer science education has emerged as a recurring topic within the last years due to the growing relevance of skills within the context of information technology. In particular, the discussions and initiatives focus on primary and secondary education, thereby stressing the significance of imparting the basic concepts of computer science and programming to children and adolescents early on. Programming itself is a demanding activity. Therefore, numerous languages and environments have been devised to facilitate the development of programs for non professionals. These tools try to stimulate the users' interests in becoming creators of digital content, while conveying general ideas such as abstraction, logical thinking, as well as mathematical theories. In order to simplify programming, many applications for beginners adopt a visual representation and creation of the software. Visual programming languages (VPL) use graphical components representing programmatic constructs, which are connected or placed in relation to each other to form programs. By focusing on semantic aspects of programming, while reducing the possibility of syntactical errors, visual approaches have been applied to compute science education at the secondary level. In this regard Scratch poses the state of the art educational programming environment enabling children and novices to create their own animations and programs. Scratch features a purely visual language based on color-coded Lego-like blocks representing operators, functions, or object attributes and has been intended for the use on traditional computers.

While usually desktop computers and laptops have been employed to develop software for decades, the smartphone penetration among citizens of industrial countries provides new opportunities for education proposes in this regard. Owing to their pervasive nature, Internet capabilities and sensors, portable devices can support learning anytime and anywhere. Mobile phones have become an essential part of teenagers' and children's everyday life, thus expanding the efforts of novice programming environments to smartphones and tablets represents the next logical step to foster computer science education.

Pocket Code is a mobile integrated development environment (IDE) and interpreter developed for the VPL Catrobat targeting teenage users. Inspired by Scratch programs in the Pocket Code app are constructed via colored bricks while taking full advantage of the device sensors available. Achieving an appealing programming experience on mobile devices, however, is impeded by the small screen sizes and error-prone data entry methods. Thus, solely relying on graphical blocks can be impractical. Especially considering formula composition or manipulation, employing a visual representation can become cumbersome and confusing with the size of the formula. Although being a crucial part of programming itself, this issue has

gained little attention in research so far. To avoid this issue, Pocket Code utilizes a hybrid textual/visual editor for constructing and adapting formulas.

In order to evaluate the mechanics of the hybrid approach we carried out two independent usability assessments, namely a heuristic evaluation and a usability test. Regarding the former, a review of the formula editor interface of Pocket Code was conducted considering specific guidelines for mobile applications. Several reviewers with usability experience were employed to identify possible issues violating the provided principles. Due to the heuristics used, the identified problems give a general picture of the usability of the formula editor besides the pure mechanics of creating or editing a formula. The empirical assessment was performed as a formal usability study comparing the purely visual formula composition and manipulation in Scratch to our textual/visual method in regard to efficiency, effectiveness, and perceived user satisfaction. While the test results indicate that the hybrid approach is more efficient and effective than a purely visual technique, we still were able to create an aggregated list of usability concerns based on both evaluations for the formula editor. These issues have been the basis for our recommendations, which include a revalidation of the terminology used, the inclusion of an expert mode as well as a redesign of tap target sizes and the addition of visual clues within the interface.

**Keywords:** Usability, Pocket Code, Visual Programming Language, Novice Programming Environment, Computer Science Education, Scratch, Android, Mobile Phone, Heuristic Evaluation, Empirical Study

## Acknowledgements

First, I would like to acknowledge my better half, Krasimir, who encouraged and reassured me during the entire process of working on this thesis.

I would like to express my gratitude to my supervisor, Wolfgang Slany, for enabling me to create a thesis I am passionate about, for his expertise as well as guidance, and most of all for his patience.

I am grateful for my parents and brother, who have always motivated me to challenge myself and supported me during my entire studies.

My special thanks are extended to the Catrobat team<sup>1</sup> as well as the test participants as without them this thesis would not have been possible.

Graz, March 2016

Roxane Koitz

---

<sup>1</sup> <http://catrob.at/credits>

# Contents

---

<b>List of Tables</b>	<b>iv</b>
<b>List of Figures</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Smartphone Penetration . . . . .	2
1.2 Problem Statement . . . . .	3
1.2.1 Pocket Code . . . . .	4
1.2.2 Formula Manipulation . . . . .	6
1.3 Contribution of the Thesis . . . . .	9
1.4 Thesis Outline . . . . .	9
<b>2 Related Work</b>	<b>11</b>
2.1 Research on Visual Programming Languages . . . . .	11
2.1.1 Algebraic Specification . . . . .	14
2.1.2 Language Comparison and Evaluation . . . . .	18
2.2 Programming Education . . . . .	19
2.2.1 Teaching Systems . . . . .	19
2.2.2 Constructivism . . . . .	21
2.2.3 Metaphor . . . . .	22
2.2.4 Educational Programming Languages and Environments . . . . .	23
2.2.4.1 Logo and Its Descendants . . . . .	23
2.2.4.2 Rule-based Systems . . . . .	28
2.2.4.3 Mobile App Programming Environments . . . . .	33
<b>3 Theoretical Background</b>	<b>37</b>
3.1 Attributes . . . . .	38
3.2 Evaluation Methods . . . . .	40
3.3 Heuristic Evaluation . . . . .	41
3.3.1 Evaluators . . . . .	42
3.3.2 Heuristics . . . . .	43
3.3.3 Usability Issues and Ratings . . . . .	45
3.3.4 Advantages and Disadvantages . . . . .	46
3.4 Usability Testing . . . . .	47
3.4.1 Test Plan . . . . .	47
3.4.2 Test types . . . . .	48
3.4.3 Sampling . . . . .	49
3.4.4 Tasks . . . . .	50

3.4.5	Quality of Testing . . . . .	50
3.4.6	Measuring the User Experience . . . . .	50
3.4.6.1	Quantitative Data . . . . .	51
3.4.6.2	Self-Reported Data . . . . .	53
3.4.6.3	Behavioral and Physiological Metrics . . . . .	56
3.4.6.4	Single Usability Metric . . . . .	59
3.5	Method Triangulation . . . . .	59
3.6	Children and Teenagers . . . . .	60
3.6.1	Designing for Children . . . . .	60
3.6.2	Usability Evaluations . . . . .	63
3.6.2.1	Age Ranges . . . . .	64
3.6.2.2	Guidelines . . . . .	65
3.6.3	Teenagers . . . . .	66
3.7	Mobile Devices . . . . .	67
3.7.1	People At the Centre of Mobile Application Development . . . . .	68
3.7.2	Interaction Mode . . . . .	69
3.7.3	Operating System Guidelines . . . . .	70
3.7.4	Children and Mobile Devices . . . . .	71
3.7.5	Mobile Usability Testing . . . . .	72
<b>4</b>	<b>Usability Evaluation</b>	<b>75</b>
4.1	Heuristic Evaluation of Pocket Code . . . . .	75
4.1.1	Evaluators . . . . .	75
4.1.2	Heuristics . . . . .	76
4.1.3	Results . . . . .	78
4.2	Summative Usability Study . . . . .	89
4.2.1	Formula Manipulation in Scratch . . . . .	89
4.2.2	Procedure and Data Collection . . . . .	91
4.2.2.1	Participants . . . . .	94
4.2.2.2	Pilot . . . . .	97
4.2.2.3	Training Sessions . . . . .	97
4.2.2.4	Tasks . . . . .	98
4.2.2.5	Questionnaires . . . . .	107
4.2.3	Results . . . . .	108
4.2.3.1	Effectiveness . . . . .	108
4.2.3.2	Efficiency . . . . .	109
4.2.3.3	Questionnaires . . . . .	111
4.2.3.4	Single Usability Score and Preference Data . . . . .	111
4.2.3.5	Eye Tracking . . . . .	115
4.2.3.6	Issues and Interpretation . . . . .	121
4.2.3.7	Validity . . . . .	129
4.3	Recommendations . . . . .	130
<b>5</b>	<b>Conclusion and Future Work</b>	<b>138</b>
	<b>Bibliography</b>	<b>141</b>

<b>Appendix</b>	<b>151</b>
7.1 Pre Test Questionnaire . . . . .	152
7.2 Tasks German . . . . .	157
7.3 Post Test Questionnaire . . . . .	159



# List of Tables

---

3.1	Nielsen's usability heuristics. . . . .	44
3.2	Issue rating according to the problem's severity and frequency of occurrence. . . . .	46
3.3	Statistics for different data types and usability metrics. . . . .	52
3.4	Curved grading scale interpretation of SUS scores. . . . .	55
3.5	Design recommendations for childrens' software. . . . .	62
4.1	Heuristics for the usability evaluation of mobile interfaces. . . . .	77
4.2	Example of a heuristic evaluation report row. . . . .	78
4.3	Results of the heuristic evaluation. . . . .	86
4.4	Sample size iteration procedure for t-tests. . . . .	96
4.5	Geometric mean of time on task (in seconds). . . . .	110
4.6	Post-test interview answers. . . . .	114
7.1	Pre-test questionnaire answers (1). . . . .	154
7.2	Pre-test questionnaire answers (2). . . . .	156

# List of Figures

---

1.1	Smartphone and tablet penetration in Germany (2014).	3
1.2	Mobile programming app Pocket Code.	5
1.3	Pocket Code formula editor view.	7
1.4	Error highlighting and computation results in Pocket Code.	8
2.1	Four problem groups for Boolean expressions.	15
2.2	Textual OBJ specification.	16
2.3	Visual OBJ specifications	17
2.4	Frame cursor depicted by the blue line.	17
2.5	Logo Turtle.	23
2.6	Etoys.	24
2.7	Scratch's multi-pane view.	25
2.8	Visual differences between brick types in Scratch.	25
2.9	Parts of a Scratch program.	26
2.10	Snap!	27
2.11	Scratch Jr.	27
2.12	Coding apps for younger children.	28
2.13	AgentSheets.	29
2.14	HANDS.	30
2.15	ToonTalk.	31
2.16	Alice 2.	32
2.17	Kodu Game Lab.	32
2.18	TouchDevelop's editors.	33
2.19	App Inventor Component Designer.	34
2.20	App Inventor Block Editor.	35
2.21	Stencyl.	35
2.22	Tickle.	36
3.1	Three dimensions of user experience.	38
3.2	Learning curves for two hypothetical systems.	39
3.3	Ratio of usability problems found as more evaluators are added.	43
3.4	Usability test set-up.	48
3.5	Rating scales in surveys.	53
3.6	Single Ease Question	54
3.7	Visualizations of eye tracking data.	58
3.8	PACMAD	69
3.9	Summary of how people hold and interact with mobile phones.	70
3.10	Comparison of touch screen interaction between adults and children.	71

3.11	Usability testing set-ups for mobile devices. . . . .	73
4.1	Issues uncovered in the heuristic evaluation (1). . . . .	87
4.2	Issues uncovered in the heuristic evaluation (2). . . . .	88
4.3	Nested formula in Scratch. . . . .	89
4.4	<i>Operator</i> category in Scratch. . . . .	90
4.5	Formula construction in Scratch. . . . .	91
4.6	Replacing the <i>and</i> operator in the second nesting level. . . . .	92
4.7	Illumination of the drop location of a brick in Scratch. . . . .	92
4.8	Test set-up for Pocket Code. . . . .	93
4.9	Test set-up for Scratch. . . . .	94
4.10	Recordings used for data collection. . . . .	95
4.11	Age and gender of participants. . . . .	97
4.12	Training session for Pocket Code. . . . .	98
4.13	Programs used for performing the tasks. . . . .	99
4.14	Task 2 Pocket Code. . . . .	100
4.15	Task 2 Scratch. . . . .	101
4.16	Task 3 Pocket Code. . . . .	103
4.17	Task 3 Scratch. . . . .	104
4.18	Task 4 Pocket Code. . . . .	105
4.19	Task 4 Scratch. . . . .	106
4.20	SEQ . . . . .	107
4.21	Task completion rate by application and task. . . . .	108
4.22	Efficiency results (Error bars represent the 95% confidence interval). . . . .	112
4.23	SEQ Scores and Single Usability Score (Error bars represent the 95% confidence interval). . . . .	113
4.24	AOI for the formula editor for Task 3. . . . .	116
4.25	Eye tracking analysis of Task 2. . . . .	118
4.26	Heat map of Task 3 in the time frame after inserting $\times$ till inserting the random number generator. . . . .	119
4.27	Heat map of Task 4 when users are scanning the screen for $>$ . . . . .	119
4.28	Eye tracking analysis of Task 4 when users are scanning the screen for <i>AND</i> . . . . .	120
4.29	Adding a minus within a field. . . . .	121
4.30	Illumination indicating brick drop location. . . . .	123
4.31	Nested formula of Task 4. . . . .	125
4.32	Delete/Undo button. . . . .	126
4.33	Scratch brick overview. . . . .	127
4.34	Pocket Code categories. . . . .	128
4.35	Nested formula in Scratch. . . . .	129
4.36	Expression choices for Boolean operators. . . . .	131
4.37	Scrape user analysis visualization . . . . .	132
4.38	Tydlig interface featuring several tabs. . . . .	133
4.39	Current formula editor with adapted text highlighting. . . . .	136
4.40	Current formula editor. . . . .	137

# 1 Introduction

---

“Tell me about the programming you and your friends do.”

“We don’t do programming.”

“What about that game you showed me with the running man?”

“Mum, that wasn’t programming, that was simple.” [106]

“Computational Thinking” refers to the thought process of formulating and solving problems that involve abstraction, algorithmic thinking, the application of mathematical concepts, and the comprehension of problems of scale. Those fundamental skills are, however, not only relevant for computer scientists, but should be part of every child’s analytical ability [141]. Within the last years due to the ubiquity of computing a discussion on Science, Technology, Engineering, and Mathematics<sup>1</sup> (STEM) skills acquisition at the K-12<sup>2</sup> level has taken place not only in the United States, but also within the European Union. As reported by the Industriellenvereinigung [42] Austria ranks low in regard to the amount of teenagers choosing STEM careers in a pan-European comparison [62].

Several initiatives and organizations have been launched such as *code.org*<sup>3</sup> or *made with code*<sup>4</sup> with the specific goal to expose more children and adolescents to programming. The idea is to spark interest in computer science early on and make them not only consumers of digital content, but rather creators. However, on the one hand programming is a challenging intellectual task, which involves creative thinking and critical analysis. Programming requires developing applications in little stages, where each new part depends on the source code written so far. Thus, creating programs is a cycle between reading and understanding the program and producing new source code. On the other hand to impart computational knowledge onto children and teenagers, the process of programming should be both accessible and exciting. As noted by Resnick et al. [114], children face difficulties in understanding the syntax and semantic of programming [62].

Visual Programming Languages (VPL) have been considered to ease end-user<sup>5</sup> programming, as they hold the potential to empower users with little to no programming experience to write

---

<sup>1</sup> MINT (Mathematik, Informatik, Naturwissenschaft und Technik) is a comparable term within the German speaking countries [42]

<sup>2</sup> K12 describes education delivered to students primarily between the age group of 6 to 18 years.

<sup>3</sup> code.org (accessed 2014-01-30)

<sup>4</sup> madewithcode.com (accessed 2014-01-30)

<sup>5</sup> The term *end user* has been coined by Nardi [87] during her research on spreadsheets usage in offices and refers to anybody who is using a computer.

source code themselves [105]. VPL use a predefined graphical representation of language components such as commands, control structures, or variables, instead of a textual one. Programs are developed by placing and connecting these visual objects in different ways. Resulting from the encoding of the syntax within the components' shapes, syntactically incorrect statements are omitted [137]. However, utilizing visual elements alone does not yield an easy to learn programming environment as it merely removes some of the frustrations related to syntactical challenges of textual programming languages [112]. In particular, user studies have shown that the superiority of visual programming languages subsides on larger undertakings [36].

Since VPL can reduce the programming burdens for novice programmers they have been used in the context of computational education at the K-12 level. Scratch is an example of a successful visual programming language as well as an environment designed for traditional computers. It is aimed at children and builds on a Lego-style block metaphor [114]. By removing the syntactical obstacles a focus on semantic solutions and the underlying programming principles is possible [62].

## 1.1 Smartphone Penetration

Although desktop computers and laptops have been utilized as a convenient programming set-up for visual as well as textual programming languages for decades, the advent of mobile devices, such as smartphones and tablets, has created new possibilities. Increasing computational power and memory comparable to smaller traditional desktop computers or laptops make mobile devices compelling pervasive computers. Most mobile devices further provide some sort of Internet connection, advanced touchscreen, and several sensors offering auxiliary information, for example, on the device's inclination or acceleration [135, 62]. As the cost of the devices decreases, they are becoming available for low-income households as well.

Children and teenagers adopt new technology rather fast as they are digital natives<sup>6</sup>. According to PewInternet [74] 37% of all teenagers in the United States owned smartphones in 2012 and one in four teens had a tablet computer, a number comparable to the general adult population. Higher numbers of smartphone penetration have been reported in the European Union, where 53% of children between the ages of nine to sixteen years, who use the Internet, possess their own smartphone, and 18% own tablets [78].

Due to their ubiquitous computation power, portable devices support learning anytime and anywhere, creating an increasing interest in the field of mobile education, with mobile programming environments as one possibility of computer science education within this context [40]. In 2012 approximately 80,000 educational apps were available for mobile devices including iOS, Android, BlackBerry and Windows Phone apps [25]. As depicted in Figure 1.1 the smartphone dissemination among teenagers and children in Germany grows with age.

---

<sup>6</sup> Prensky [108] coined the terms *digital native* and *digital immigrant*. The former refers to the generation growing up with technology such as computers and the Internet and thus are "native speakers" of this digital language. The later are the individuals born before the digital world.

Important here is that for children and teenagers over the age of ten years, we can observe that more than 50 % among this population own a smartphone. The increasing availability of smartphones and tablets leads to a phenomenon where touch screen interactions are becoming commonplace while other input methods such as keyboards or mouse interactions are eclipsing.

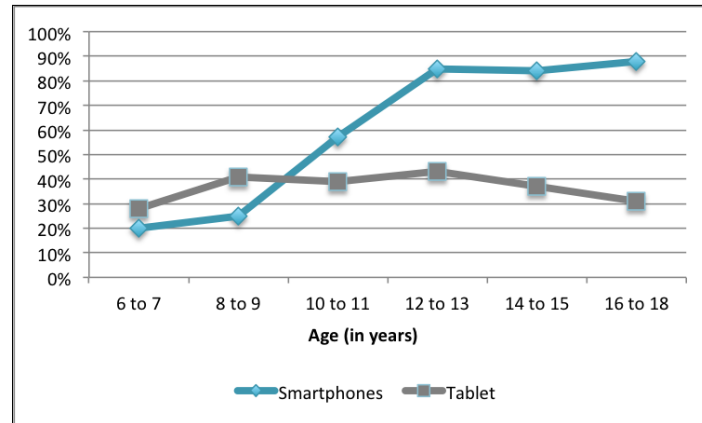


Figure 1.1: Smartphone and tablet penetration in Germany (2014).<sup>7</sup>

## 1.2 Problem Statement

Even though smartphones and tablets are becoming increasingly popular among children and teenagers, creating a positive and engaging programming experience on mobile devices is a challenging task, as screen size impedes interface design. In addition, data entry methods on touch screens are more prone to errors, hence typing large amounts of source code on the mobile device's virtual keyboard is not an appealing option from a user experience point of view [44]. Microsoft's TouchDevelop<sup>8</sup> demonstrates one solution to this problem. It is a textual programming language with a specialized structural editor, making each editing step possible with a tap on a touchscreen [135]. When inserting a statement, a calculator view is displayed. This calculator view comprises a virtual on-screen keypad, with several keypad modes containing, for example, operators or literals. Even though the interaction is in a graphical style, the actual visualization is purely textual [62].

While TouchDevlop's approach is optimized for mobile devices, creating a purely textual programming language for children and adolescents is just one approach. Pocket Code is a mobile programming environment for the visual programming language Catrobat<sup>9</sup>, which has been inspired by Scratch and is intended for novice teenage users. Due to the limited display size of smartphones, a purely visual approach as in Scratch is impractical. Especially formula composition can become cumbersome on small displays. Therefore, the Catrobat

<sup>7</sup> [www.emarketer.com](http://www.emarketer.com) (accessed 2014-12-28)

<sup>8</sup> <https://www.touchdevelop.com/> (accessed 2014-01-30)

<sup>9</sup> <http://www.catrobat.org/> (accessed 2014-01-30)

team implemented a hybrid formula manipulation approach, where formulas are created visually and displayed textually through an electronic pocket calculator metaphor [62].

Harzl et al. [45] present a classification of formula manipulation in the context of end-user programming. According to the authors, three categories can be distinguished [62]:

- Purely visual: Visual formulas are constructed connecting predefined visual components. Scratch is an example of a purely visual formula manipulation environment.
- Purely textual: Formulas are displayed and created textually, for example in spreadsheet applications as Microsoft's Excel.
- Hybrid textual/visual: In the hybrid approach formulas are created using a mixture of visual and textual representation.

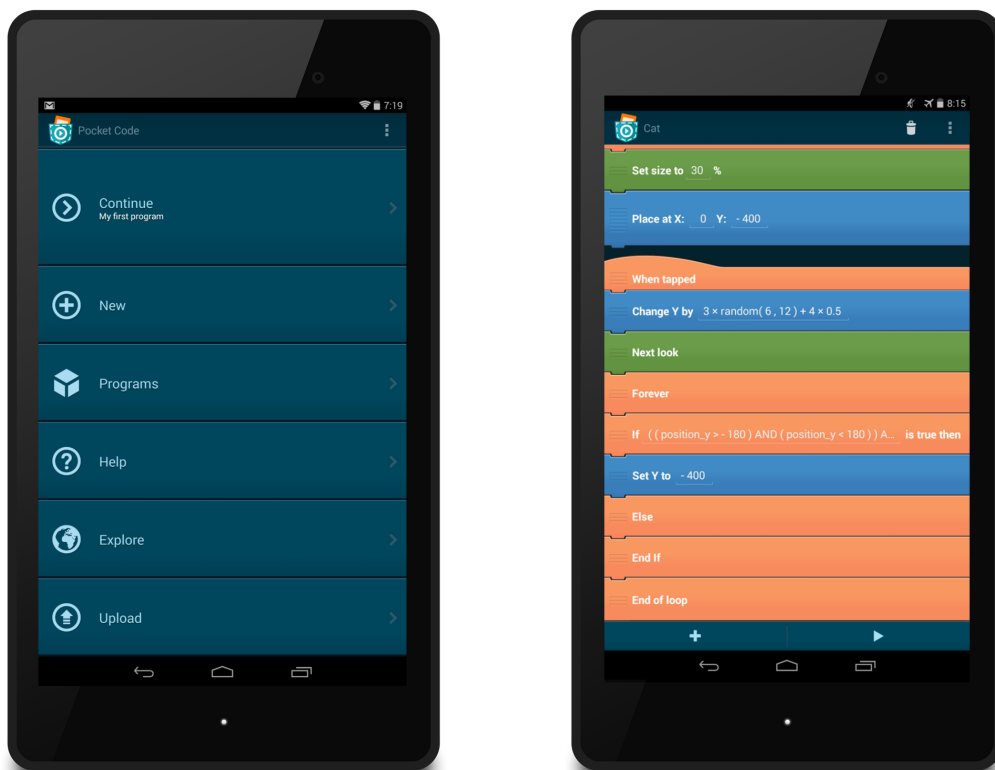
Additionally, the authors propose a formal experiment design in order to compare the effectiveness, efficiency, and subjective user satisfaction of the different approaches. Their design focuses in particular on programming environments available on mobile devices [62].

Despite the rationale behind the hybrid formula manipulation approach, an evaluation is yet to be conducted. In this thesis, we investigate the advantages and disadvantages of hybrid formula manipulation in programming environments for teenagers.

### 1.2.1 Pocket Code

Catrobat is a visual programming language and a set of creativity tools for mobile devices especially designed for teenagers between the ages thirteen to eighteen. An interpreter for Catrobat and a mobile integrated development environment (IDE) are combined in the Pocket Code app (see Figure 1.2). The Pocket Code Android app is available on Google Play and versions for iOS and Windows Phone are in development currently [62]. Pocket Code enables teenagers to create animations and develop programs directly on mobile devices in a fun and engaging way without the need for traditional desktop computers. Pocket Code has been inspired by Scratch, adopting its block metaphor and community aspect. However, it differs from Scratch, as it has been designed and developed for multi-touch mobile devices with small screens starting from three inches. Due to its focus on smartphones and tablets it allows the usage of the device sensors (e.g., compass, acceleration, and inclination sensors) [127]. Further, there are several sub projects extending the basic functionality, such as a 2D physics engine, Arduino, or Lego Mindstorms blocks [62].

As in Scratch, programs are developed by composition of color-coded Lego-style blocks, with a particular functionality embedded such as conditionals, loops, or other statements. These blocks are organized in categories and can be connected via drag and drop. Figure 1.2b depicts the scripting area, where blocks can be interlocked to form programs [62].



(a) Home screen.

(b) Script view.

Figure 1.2: Mobile programming app Pocket Code.



### 1.2.2 Formula Manipulation

Since Pocket Code is a mobile app, the amount of information that can be displayed at once is limited. Furthermore, the input methods differ from those of desktop computers and are more error-prone. For example, tap gestures are not as precise as mouse-clicks [145]. Therefore, a purely visual representation of formulas is not a practical approach, especially as formulas can become quite complex resulting from nesting and long variable names [62].

Pocket Code uses a visual formula composition by tapping on the appropriate buttons in the formula editor, but displays them textually within text field and the script view as shown in Figure 1.3. By exploiting buttons symbolizing attributes or functions, the formula editor input method is less prone to errors than typing entire formula using a virtual keyboard. The textual depiction, however, requires less screen space than interlocked visual blocks would. Therefore, the hybrid approach should combine the ease of VPL with the effectiveness and clarity of textual representations. The formula editor uses an electronic pocket calculator metaphor, which was chosen since it maximizes the transfer of knowledge as the target audience would likely be familiar with calculators from math classes at school or calculator apps. Further, a textual representation is feasible on narrow screens by using text wrapping. In addition to the common digits and the arithmetic operators available on standard pocket calculators, statements are organized in categories within the formula editor, displayed in Figure 1.3 at the bottom right. Those categories comprise object attributes, mathematical functions, logic and relational operators, as well as sensors and variables. Figure 2.3a depicts a nested formula containing a variable (“*car position*”), an object attribute (*position\_y*) as well as logical and relational operators [62].

Note that it is initially possible to create syntax errors on an expression level, since there is no automatic type checking done during the creation of the formula. In case the user tries to save an unfinished or incorrect formula, the formula editor highlights the corresponding sub formula as depicted in Figure 1.4a and thereby eliminates some of the possible syntax errors preemptively. Depending on the operator type coercions can occur in case the operands do not match the operator. For example, relational operators expect operands of type Real, however, whenever Boolean values are supplied they are automatically converted to Reals.

Additionally, parentheses provide a common visual clue to denote grouping and order of operation within the calculator metaphor. Indicating grouping can be a crucial point when creating deeply nested or large formulas. “Undo” and “redo” functionality have been implemented to facilitate the composition of formulas. Moreover, the *Compute* button depicted in Figure 1.4b, which evaluates the formula and displays the current result, is present, and also allows to interactively take into account sensor values [62].

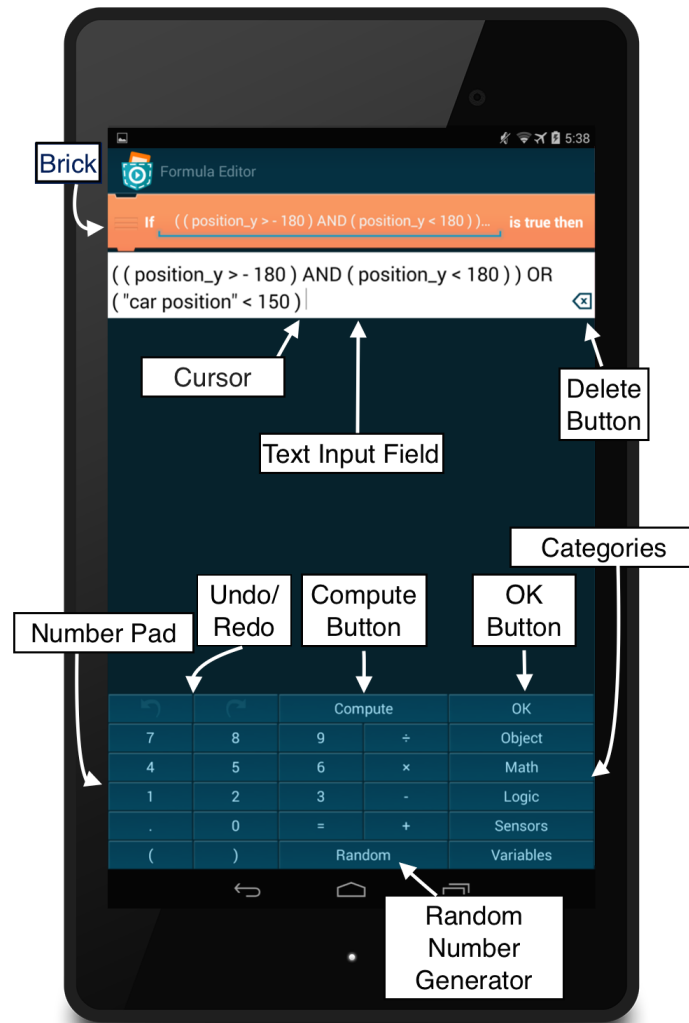
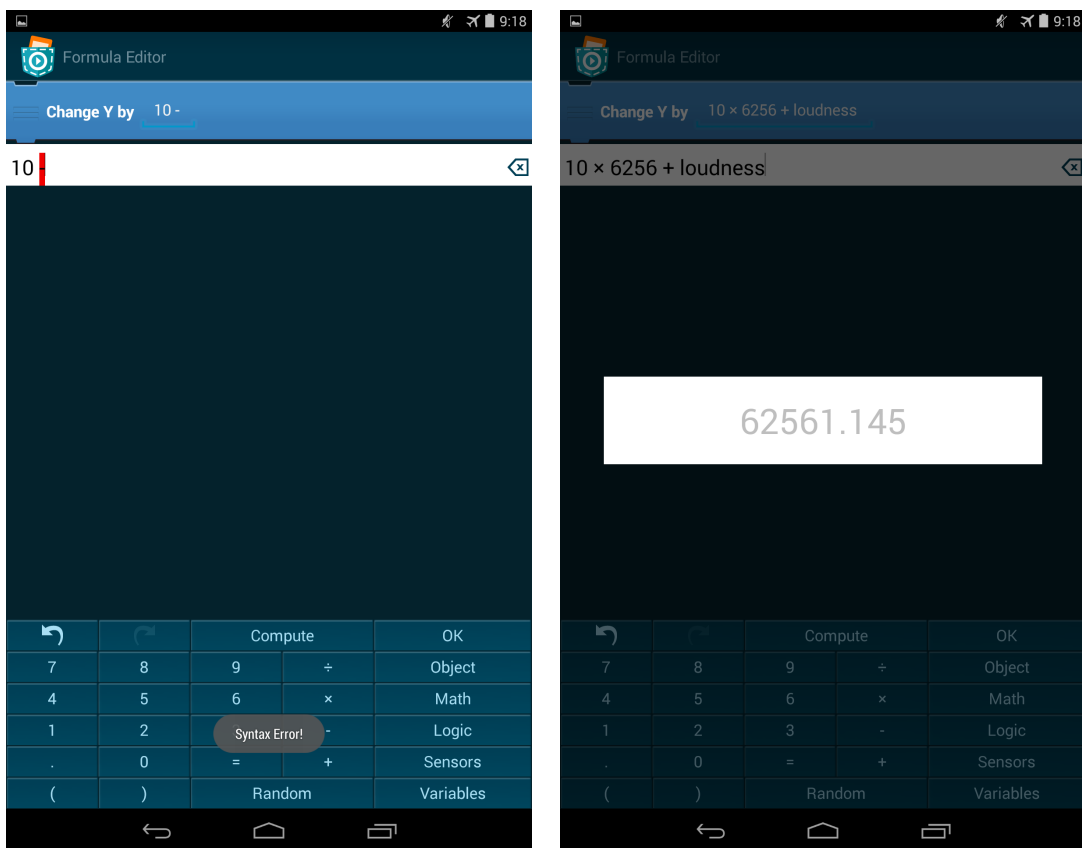


Figure 1.3: Pocket Code formula editor view.



(a) Syntax error highlighting in Pocket Code.

(b) Computation result.

Figure 1.4: Error highlighting and computation results in Pocket Code.

### 1.3 Contribution of the Thesis

Within this thesis, we have on the one hand accumulated a broad overview of the state of the art research on educational programming languages and environments as well as their inherent issues and aspects concerning formula composition. On the other hand, we have administered two evaluations to assess the usability of formula manipulation. Subsequently, we aggregated the uncovered issues and developed a set of suggested actions to improve the usability of the formula editor in Pocket Code. This thesis presents the following contributions:

- We give an extensive overview of research in educational programming environments and in particular on the question whether VPL are superior in this regard to their textual counterparts.
- We performed a heuristic evaluation of the formula editor in Pocket Code based on mobile application usability guidelines.
- We conducted a formal experiment, comparing the efficiency, effectiveness, and perceived satisfaction of formula development in Pocket Code to Scratch.
- Based on the results of the two usability assessments we created a set of recommendations for the formula editor in Pocket Code.
- Lastly, we provide some notions of possible extensions of the research described in this thesis as well as additional aspects worth investigating.

### 1.4 Thesis Outline

The thesis is structured as follows. In Chapter 2 we give an overview of the body of research on comparing visual to textual programming languages and education programming environments. In particular, we discuss aspects proven useful in minimizing the burden of learning computer science concepts. Numerous languages and environments for computer science purposes have been developed since the 1960s and we list the most important ones in regard to our research within the last section of this chapter.

Chapter 3 comprises essential theoretical background on usability research, i.e. its attributes and assessment methods. Subsequently, we focus on heuristic evaluation as an inspection technique and formative usability testing as an evaluation approach involving participants from the target user group. As Pocket Code is a mobile application intended for children and teenagers, we further point out essential usability adaptations necessary for children and teenage users as well as mobile devices.

In Chapter 4 we present two assessments, which we have conducted on the formula editor of Pocket Code. In the first section, we describe a heuristic evaluation we performed with members of the user experience team of the Catrobat project and its results. Subsequently, we discuss a formative usability study where we empirically compared the hybrid textual/visual formula composition paradigm implemented in Pocket Code to the purely visual approach of Scratch. Section 4.2 is based on Koitz and Slany [62]. We further aggregate the results of

the heuristic evaluation and the usability test into a set of recommendations for the formula editor of Pocket Code as well as present ideas for suitable future research.

Finally in Chapter 5, we conclude the thesis and provide additional suggestions for future work within the context of formula manipulation in Pocket Code and generally in regard to novice programming environments for teenagers and children.

## 2 Related Work

---

*“One might say the computer is being used to program the child. In my vision, the child programs the computer, and in doing so, both acquires a sense of mastery over a piece of the most modern and powerful technology and establishes an intense contact with some of the deepest ideas from science, from mathematics, and from the art of intellectual model building.”*  
[103]

Nowadays, computer users encompass a diverse and significant population, since information technology has become a ubiquitous part of everyday life. Thus, the term *Computer Literacy* has been defined as the skills to use tools such as word processing, spreadsheets, basic computer operations and Internet tools [72]. Teaching layman, i.e. non-professional software developers, to create their own programs has been promoted within the last years as the next logical step to *Computer Literacy*. The intention to create programming languages and environments which allow a broader public to be able to access programming has already been pursued since the 1960s [58]. End user programming (EUP) can be defined as “*programming to achieve the result of a program primarily for personal, rather [than] public use*” [61]. End users write code in order to achieve their goals within their domain, thus the end product is not necessarily intended for a large user base with changing needs [58]. It has been shown that the amount of research on EUP has increased within the last years with the larger number of lab-based studies [134].

Within the context of computer science education, research on programming languages for children and teenagers has received attention since the 1980s with Seymour Papert as one of the key pioneers [103]. Since then various learning environments for desktop computers and other devices have been promoted. In this chapter, we discuss on the one hand educational programming languages and environments for various target groups with a focus on children and teenagers and on the other review literature in the context of VPL.

### 2.1 Research on Visual Programming Languages

Research on VPL has started as early as the 1970s, when Smith [129] introduced an icon-based programming language called Pygmalion. Pygmalion establishes the nowadays common drag and drop functionality, which is prevalent in educational programming environments for children [68]. In visual languages, graphics replace some or all of the textual program source code and were thought to facilitate end-user programming and thereby empower novices not

to merely consume content but to create programs themselves. Contrasting their textual counterparts they have the “*potential to minimize the conceptual distance between the cognitive and computational model*” [105]. Furthermore, VPL make implicit relations and information explicit which in turn simplifies programming and problem solving.

Boshernitsan and Downes [16] provide a classification for VPL environments:

- *Purely visual languages*: The programmer only interacts with graphical representations.
- *Hybrid text and visual systems*: A combination of textual and visual elements is embedded.
- *Programming-by-example systems*: The user manipulates or creates visual objects and thereby instructs the system how to behave.
- *Constraint-oriented systems*: Primarily utilized in simulation design, the programmer models objects, which are behaving according to certain restrictions such as natural laws, in a graphical way.
- *Form-based systems*: In form-based systems, the users create cells and defines formulas for these cells. Spreadsheet programming environments fall under this category. Spreadsheets are one of the most successful end-user programming systems as of today, however, are mostly textual.

These categories are not mutually exclusive, thus a language can fit into several categories at the same time. In contrast Asamoah [7] classify VPL based on the visuals they are incorporating:

- *Diagram-based*: The features are represented visually, often as diagrams.
- *Form-based*: Forms or templates are used to retrieve all necessary information from the user.
- *Icon-based*: These types of VPL require the user to directly manipulate visual representations of processes with a pointing device (mouse or finger).

There have been several studies to verify the advantages of VPL over their traditional textual counterparts. Proponents of visual programming languages state that reducing or eliminating text in programming will improve usability as visual representations are easier to grasp than textual ones and syntax issues are deemphasized [130, 62]. Comparing visual perception to reading text, the former is more efficient and natural. Particularly, program structure can be easier analyzed in visual systems than textual ones. Further, the relationships between components can be visualized quite easily and the spatial layout of the components allows to further convey information [37].

Yet, much of the rationale behind the claims of VPL-benefits encountered skepticism [13]. User studies have shown mixed results on the advantages of visual languages, which are relative to the particular application task [36]. In general, the superlativist theory [37]—claiming VPL to be inherently superior to textual representations, in all situations—cannot be supported [62].

Green and Petre [36] examined the comprehensibility of textual and visual programs through comparison of LabVIEW, a circuit-diagram-like language, to a purely text-based one for conditional logic. In particular, the authors conducted the study under the match-mismatch hypothesis. In the context of VPL, this hypothesis states that the ease of retracting information or problem solving depends on how well the notation structure matches the problem structure [89]. The authors concluded that the data flow paradigm was, in fact, harder to comprehend than the textual one and that the benefits of VPL may lie in the creation of programs rather than understanding already existing source code [62]. Graphical source representations inherently involve a greater level of abstraction than traditional text-based languages. Contrasting the superlativist theory, the authors argued the VPL was more difficult to comprehend due to the fact that the structure of graphical representations is more difficult to scan in comparison to textual ones. Moher et al. [84] constructed the same experiment as Green and Petre [36], but utilized Petri Nets as a visual representation to examine whether the previous results were dependent on the LabView forms. The authors concluded the same result as Petre and Green under every condition.

Pandey and Burnett [99] argued in favor of VPL and focused on program constructability in the context of matrix manipulation. The study compared textual languages, namely Pascal and OSU-APL, to Forms/3 [22], a spreadsheet-based visual programming language. Based on their results the authors concluded that source construction in VPL is less error-prone. This has been especially of interest as the participants had all prior experience in Pascal [62].

In contrast, Booth and Stumpf [15] recently reported greater difficulties in the creation of visual programs than their modification. The study analyzed Modkit<sup>1</sup>, a visual programming environment based on Scratch, and text-based programming languages for Arduino<sup>2</sup>. Their investigation supported the belief that visual programming environments provide a greater user experience and perceived success as well as decrease the subjective workload for adult end-user programmers [62].

A well-known study by Shneiderman [126] tested comprehension, composition, debugging, and modification of flow chart documentation on novice users. There was no significant difference between the task times of the flow chart and non-flow chart group. Ramsey et al. [110] compared flow charts to a program design language and showed that there was a significant advantage of the latter, due to the compression of information associated with flow charts, triggering space saving habits which reduced the quality of the designs. Scanlan et al. [122] investigated comprehension of conditional logic in structured pseudo code and structured flow charts. The results indicate a significant advantage of the flow chart variant in regard to time and error rate.

In a recent study, Price and Barnes [109] compared textual programming language to a brick-based language while isolating the language effects from the programming environment. In particular, they created two environments, a brick-based and a textual one. They found that while the perceived effort was independent of the testing language, their results convey that the VPL lead to a greater number of tasks performed. Price and Barnes [109] further identified the three characteristics of many block-based programming environments; First,

---

<sup>1</sup> <http://www.modkit.com/> (accessed 2014-01-28)

<sup>2</sup> Arduino are microcontroller boards.



novice users are the target group, in particular, children and teenagers. Second, VPLs mimic the structure as well as syntax of existing languages and third they usually provide the possibility to include multi-media content.

As can be seen from the literature, there is no consensus on the superiority of VPL to their textual counterparts. Generally, an effective visual approach can outweigh their counterparts. VPL have been shown to be more effective and efficient in the cases of rapid-prototyping tools or shell scripting for novices in comparison to purely textual approaches. However, the superior position of VPL diminishes with the complexity and magnitude of tasks. In addition, it has been shown that for program understanding situations visual languages are less suited than their textual ones [36, 100]. Visual languages have a high viscosity, i.e. implementing changes requires effort for rearranging components and planning a layout ahead of time [38]. Furthermore, the screen space is not efficiently used in visual languages and a restricting inherent feature of VPL environments is the necessity of a display large enough to see as much of the program code as possible in order to be able to reorder the programming blocks [88].

### 2.1.1 Algebraic Specification

Accurately specifying Boolean expressions is a common problem among programming languages as well as among user search queries and database retrieval tasks [100]. Even though constructs such as *AND*, *OR*, and *NOT* are difficult for novices, there are still no generally better replacements. The difficulties of Boolean expressions are amplified when several operators are to be used in the same conditional. Pane et al. [101] conducted a study with participants from various age groups, genders, and programming experiences. In their investigation of Boolean expressions they provided four problem sections (see Figure 2.1):

- Writing text (WT)
- Writing forms (WF)
- Reading text (RT)
- Reading forms (RF)

Their study showed that, for example, users rate the precedence of *NOT* higher than of *AND* and lower than of *OR*. Often *AND* is interpreted as the union by users, while it is evaluated as intersection in most programming languages. Parentheses seem to improve nested Boolean expressions, however, some studies have shown that beginners still face difficulties with the usage of parentheses [39]. While other terms from natural language have not proven to be better suited than *AND*, *OR*, and *NOT*, some graphical representations have been shown promising results in regard to effectiveness such as truth tables, Venn diagrams, or visual metaphors [101].

Neary and Woodward [89] investigated the effects of visualization in the domain of algebraic specifications. In particular, an experiment was conducted on the programming language OBJ. Figure 2.2 shows a textual OBJ specification. This textual form was compared to two visual representations with different notations: Nassi-Shneiderman charts and Vertical

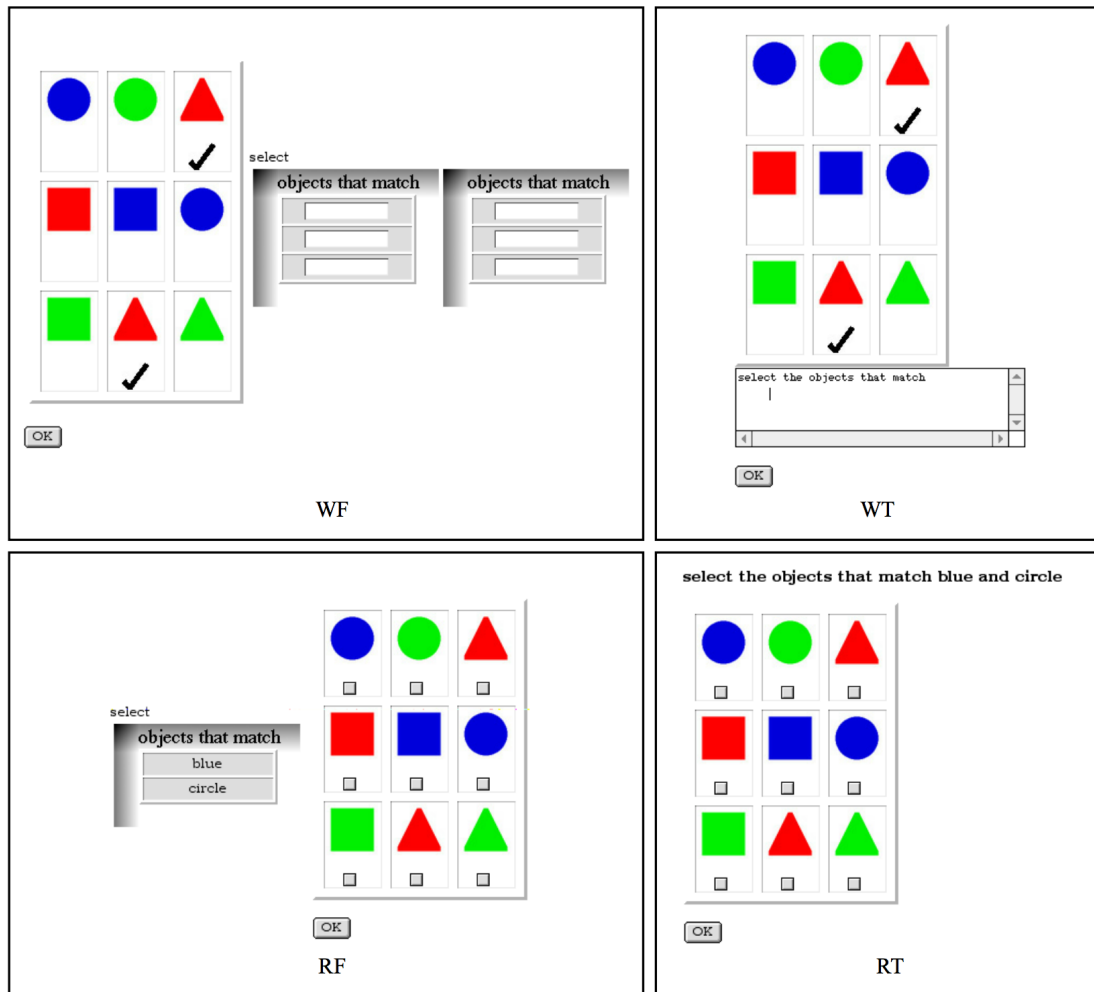


Figure 2.1: Four problem groups for Boolean expressions as described in the study by Pane et al. [101].

```

MonitorTank
OBJ Tank
SORTS tank
OPS
  empty :          -> tank  *** create empty tank ***
  add   : nat tank -> tank  *** add to tank          ***
  use   : nat tank -> tank  *** use from tank        ***
  level? : tank    -> nat   *** how much in tank? ***
  max   :          -> nat   *** tank capacity       ***
  full? : tank     -> BOOL  *** is tank full?       ***
VARS
  t : tank
  q : nat
EQNS
  ( max = 50 ) *** EQN 1 ***
  ( full?( t ) = level?( t ) == max ) *** EQN 2 ***
  ( level?( empty ) = 0 ) *** EQN 3 ***
  ( level?( add( q, t ) ) = level?( t ) + q
    IF level?( t ) + q <= max ) *** EQN 4 ***
  ( level?( add( q, t ) ) = max
    IF level?( t ) + q > max ) *** EQN 5 ***
  ( level?( use( q, t ) ) = level?( t ) - q
    IF level?( t ) >= q ) *** EQN 6 ***
  ( level?( use( q, t ) ) = 0
    IF level?( t ) < q ) *** EQN 7 ***
JBO.

```

Figure 2.2: Textual OBJ specification [89].

Nested Box. Both graphical forms are depicted in Figure 2.3. To evaluate the approaches, the authors generated a multiple choice questionnaire to determine basic syntax and semantics skills, equation comprehension, and term rewriting. Overall there was no statistical difference in task time and error rate between the notations, even though a slight favoring of the visual representation was observed. All the participants had previous experience in textual OBJ.

In block-based programming languages the users build programs by arranging blocks of code. The composition then resembles the structure of the program. Often bricks cannot only be connected to each other, but several nesting levels can be created. For example, in a conditional block the condition itself could be an arithmetic operation represented by another brick. Brown et al. [19] argue in their paper that the lack of keyboard support is restricting the usage of block-based languages to more proficient use. In particular, they show that while the development of programs utilizing block-based language is simple, the entry and manipulation is time-consuming and cumbersome. For example, creating a complex formula such as  $\sqrt{x * x + y * y}$  requires the usage of eight blocks, where the action to position each brick comprises finding the particular palette, selecting the brick, dragging it to the correct position and dropping it there. In a textual language this might require around fourteen key strokes. Thus, the authors propose frame-based programming. Frame-based programming is a hybrid approach incorporating blocks as well as text-based elements, whereby frames are components similar to bricks in block-based languages. To support keyboard entries, a frame cursor indicates where the next frame will be positioned. Frames contain expressions, which can be inserted textually. In order to diminish the occurrence of syntax errors, frames are structurally edited.

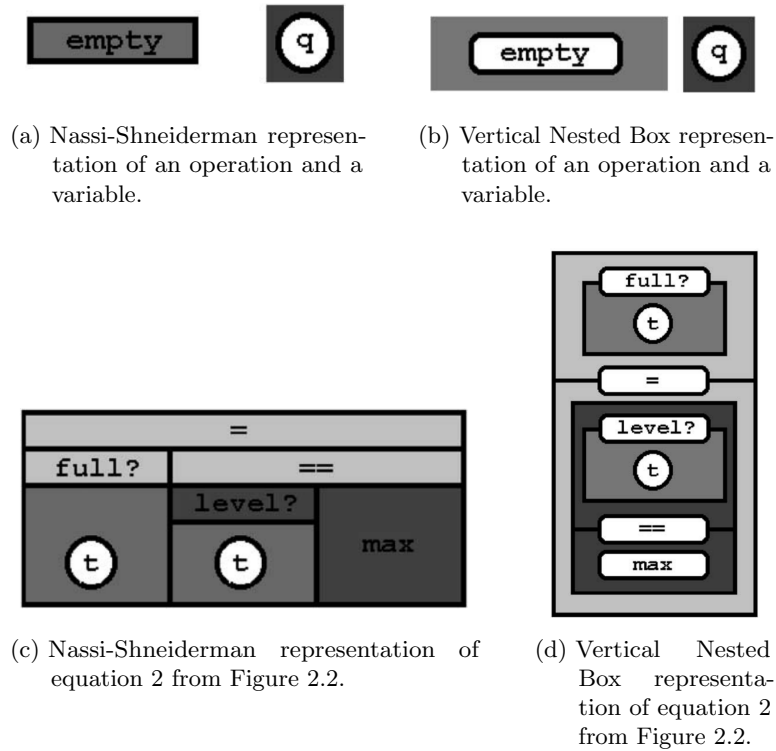


Figure 2.3: Visual OBJ specifications [89].

```

    Check whether we have hit the left edge.
    public void checkLeft()
    {
        if (getX() < 0)
        {
            turn(180)
            numBounces = numBounces + 1
            if (numBounces >= 4)
            {
                health = health - 1
            }
        }
    }
  
```

Figure 2.4: Frame cursor depicted by the blue line [19].

### 2.1.2 Language Comparison and Evaluation

Green and Petre [38] proposed the Cognitive Dimensions Framework to evaluate visual programming environments as well as languages. Their framework encompasses thirteen items:

- *Abstraction gradient*: How deep is the maximum and minimum level of abstraction?
- *Closeness of mapping*: In case there is a mapping between real life entities to parts of the environment, how much does the user have to learn in order to be able to use this mapping.
- *Consistency*: Is the user able to infer the functionality, input types, etc. of new items from previously seen ones.
- *Diffuseness*: How many visual objects are necessary to achieve a certain behavior?
- *Error-proneness*: Does the paradigm prevent “careless mistakes”?
- *Hard mental operations*: Are there scenarios, where the user has to use additional tools (e.g. pen and paper) to be able to comprehend the execution?
- *Hidden dependencies*: Are interconnections explicit?
- *Premature commitment*: Do programmers have to make decisions before acquiring all information necessary?
- *Progressive evaluation*: Does the environment allow to execute unfinished programs to assess their current functionality.
- *Role-expressiveness*: Is the relation between the entire program and its parts explicit?
- *Secondary notation*: Do other visual programming decisions, such as program layout or color, carry additional information.
- *Viscosity* : Can single changes be incorporated without having to adapt other parts of the program?
- *Visibility*: Is it possible to show every part of the program simultaneously?

When designing a language the Cognitive Dimension Framework does provide a good overview of possible trade-offs. However, the framework does not offer help in high level decisions, for example, whether a data flow or block metaphor is more suitable.

McIver [79] has investigated approaches for comparing and evaluating programming languages for novice users. Their proposed approach focuses on removing any bias due to the programming environment, thus the testing environment is a simple text editor with an additional run-button to execute the program and return any error messages. Based on the results the number and types of errors can be analyzed and performance, learning, and understanding measures can be applied.

## 2.2 Programming Education

The motivation to learn to program can have several driving factors, such as the end user wishes to pursue a career as a programmer, to improve problem solving skills and logical thinking, to be able to create personalized software, or to explore ideas in other areas by the means of coding [58]. Learning to program encompasses various activities such as learning the language's syntax, designing software and understanding already available program.

In a study Pane et al. [102] gave non-programmers various programming tasks, which included essential concepts such as control structures or Boolean logic. The participants, including adults as well as children, were asked to solve these tasks with pen and paper. The recorded solutions were analyzed and the majority of proposed approaches were event- or rule-based systems, featuring mostly natural language for arithmetic operations. In their examination the authors found that the usage of Boolean operations were not in accordance with most programming languages, which is in line with the previously discussed work on arithmetic operations [101]. Interesting enough most test users drew sketches to depict the program's layout and described it's behavior with additional text.

The majority of teaching environments focus on the one hand on enabling the students to express their intentions in a programmatic way and on the other hand teach them to understand how computers execute instructions [58]. Novice programmers are often expected to progress from a beginner's languages to commercially used ones after a while. Therefore, most systems support languages similar to general-purpose ones, i.e. an if-structure will behave similar in a teaching systems as it does, for example, in Java [58].

Sheehan [125] recommended for programming environments for children the incorporation of multimedia content, high level instructions, and an easy switch between viewing the source and seeing how the program is running. For example, initializing variables seems to be a more complicated concept than updating or testing them. Further, control structures such as loop or conditionals are more difficult than simple statements. Even though drag-and-drop has become the established input method for educational VPL, Inkpen [50] reports that children between the age of nine and thirteen prefer point-and-click over drag-and-drop.

### 2.2.1 Teaching Systems

Kelleher and Pausch [58] purpose a taxonomy of programming languages and development environments with the specific focus on systems enabling novices to program. In particular, they categorize the tools in teaching systems and systems which aim at empowering the end-user. In this section, we focus on the former.

Studies on novice programming environments have identified the mechanics of programming as the main hindering factor in successfully acquiring these skills. The mechanics comprise the following fragments [58]:

- *Language syntax*
- *Structuring programs*

- *Comprehension of program execution*

## Language Syntax

Novice programmers face difficulties in expressing themselves in syntactically correct code. There are two possible remedies to assist in overcoming this barrier: simplifying the input or creating alternative entry methods. Most commercially used programming languages are expected to be used for problems from various domains, thus, provide a vast number of functionalities which in turn leads to being more difficult for novices to master. Applications aiming at simplifying the entering of code share two common approaches [58]:

- *Simplifying the language*: General-purpose languages often include syntactical elements difficult for novices to understand, since the names are uncommon or the terms' purpose differs from their meaning in natural language. To ensure clarity the programming language GRAIL [80], for example, follows three guiding principles:
  1. Have consistent syntactical requirements.
  2. Use terminology novices understand and refrain from standard programming terms having a different intention than in natural language.
  3. Restrict the language to feature simple programmatic constructs.

For example, in GRAIL the multiplication operator is represented by an '×' rather than a '\*', since this representation is familiar to novices from mathematic classes. Further, instead of denoting an assignment by  $a = b$  which is ambiguous, an arrow is utilized e.g.  $b \rightarrow a$ .

Pane et al. [102] created a programming environment named Human-centered Advances for the Novice Development of Software (HANDS) taking into account usability principles. For example the heuristic to "speak the users' language" known in Human-computer Interaction (HCI), also applies to programming environments. In particular, the use of terms differing in their meaning in natural language such as *void* or *static* should be avoided. Users tend to apply knowledge of other areas when they do not know how to achieve a goal. Thus, when the programming language's syntax and semantics are in conflict with natural language or mathematics confusion arises (e.g.  $x = x + 10$ ).

- *Preventing syntax errors*: There are tools focusing on creating an environment which prevents common syntax errors for already existing languages. By limiting the set of possible commands to the ones syntactically correct at the current program position, eliminates the possibility for faults. For example, the programming environment GNOME [83] relied on displaying programs hierarchically and enforced users to correct syntactical errors before continuing to program. GNOME environments were available for Pascal, Fortran, and Lisp.

While simplifying the language is beneficial to novices, this user group still faces difficulties remembering commands, formal parameters or the right usage of parentheses. Thus, some systems have removed syntactical burdens altogether. According to Kelleher and Pausch [58]

we can categorize these in VPL, programming-by-example systems, and hybrid environments. In VPL conditionals, commands or operators are represented as graphical entities, where the syntax is encoded in the shape and/or color of the objects. In Section 2.2.4 we discuss some well known examples of this type of environment such as LogoBlocks, Alice, or Scratch. Others environments create programs by using the interaction between the user and the interface, e.g. by interpreting button presses and their sequence. An instance of programming-by-example environments is LegoSheets [35], where the user starts by manually controlling a small computer, called a Brick. The program is then created on basis of this interaction. The third type provides various input methods, such as Leogo [23]. Leogo generates simple graphics and provides three methods for generating code: typed syntax, interface manipulation as well as a language which includes templates for common commands.

### Structuring Programs

Instead of focusing on the syntax on statement level, these types of systems attempt to simply structuring code. Pascal for example was introduced in the early 1970s to provide a language allowing to teach structured programming. While Pascal was developed for programming classes, the language Smalltalk was designed for children as a simple programming language with few commands and little syntax [57].

### Comprehension of Program Execution

Even when a program is syntactically correct, it does not necessarily perform as the author intended. Debugging can be a difficult challenge for novices and requires to understand how the computer executes the statements provided in the source code. Systems in this category either provide feedback on how the code is executed, how variables change, or which part of the source code is executed. In Scratch, for example, the currently executed part of the program is highlighted. Another type of environment within this category provides metaphors to assist users in understanding the execution of programs. For example, in Toontalk trained robots share information by talking to each other to convey communication between objects.

#### 2.2.2 Constructivism

In 1980 Papert [103] proposed constructivism. Papert's constructivism, derived from constructivist theories, claims that people in general and children in particular have more enjoyable learning experiences and a higher effectiveness by creating. In particular, the activities of designing, personalizing, sharing, and reflecting play an essential role as they enable children to be innovative and apply knowledge rather than acquiring facts [11]. Based on constructivism educational construction kits aim at providing learning through the process of creation [113]. There are several principles [116] and connections [21], which should be considered in order to create effective construction kits, such as:



- providing a “*low floor, wide walls and a high ceiling*” [116], i.e. the system should be easy to use for novices, while it allows to explore and is capable of performing advanced tasks,
- using simple commands which symbolize sophisticated functionalities, to allow users to create programs with little code,
- trading off extensive functionality and elegant design for simplicity,
- allowing various ways to reach the same solution, and
- giving the user not the features they want but the functionality they need.

Petre and Blackwell [106] conducted an informal observation of their own children and determined the following:

- Children do not intentionally learn programming, but use it as a means for playing. They often are not aware that they are programming.
- They learn by tinkering: analyzing existing artifacts and then adapting them in order to fit their needs.
- Children learn within a social network and not an educational context.

Independent research by Lye and Koh [71] investigated empirical articles on programming in K-12 and higher education where students employed VPL. They found that the predominant strategy is based on constructivism, i.e. students create a program to strengthen the learned aspects. In most of the studies they have covered positive results were observed. This is well in line with the research of Resnick [114] on Scratch. A study by Meerbaum-Salant et al. [82] has shown that computer science concepts can be conveyed with the help of Scratch in the context of middle-school classrooms. However, the authors could show that certain ideas require a more in depth teaching approach than others. In particular, variables, initialization or concurrency are programming constructs posing difficulties for novices.

### 2.2.3 Metaphor

A metaphor is a familiar analogy which implicitly describes how the programming environment or language operates. The quality of the metaphor depends on whether the user can infer how to use the programming system by relying on already available knowledge [100]. To achieve a close mapping, the metaphor should be conceptually close to a real entity widely known by the target users. Metaphors are, however, not a panacea. Users tend to attempt to extract too much information from metaphors, i.e. confusion arises when the metaphor and the system behavior do not match entirely.

An example of such a discrepancy of the metaphor and the physical object is the turtle graphic in the novice programming language Logo. The turtle graphic draws lines according to the program code. When the turtle is facing south its right side become the user’s left side and vice versa. Thus, it can be unintuitive for beginners to direct the turtle in a new position in relation to the turtle.

## 2.2.4 Educational Programming Languages and Environments

In this section we discuss various programming environments and languages utilized for computer science education. We have divided these into three categories:

- *Logo and its descendants*
- *Rule-based systems*
- *Mobile application programming environments*

These classes are not mutually exclusive, since e.g. some mobile development environments are inspired by Logo and its successors. Another type of programming environments for novices, which we will not discuss, are environments supporting traditional programming languages, which are mainly used in higher education [41].

### 2.2.4.1 Logo and Its Descendants

Designed in the 1960's, the Logo [103] programming language is a re-engineered Lisp, where much of the punctuation is neglected in order to provide a child-friendly syntax. Besides numerical and textual computation in the context of mathematics, science, language and music, Logo is best known for its turtle graphics. Originally derived from a small robot, the turtle later became a simulation in a two dimensional space drawing line graphics according to coded movement commands. Figure 2.5 shows the turtle robot as well as the on-screen cursor turtle. The programs are object-centric, as for example the forward command would move the turtle in its own forward direction. Logo is generally an interpreted language using immediate feedback and descriptive error messages allowing users to create procedures and recursion. There are several derivatives of Logo, as for example LEGO/Logo merged Logo

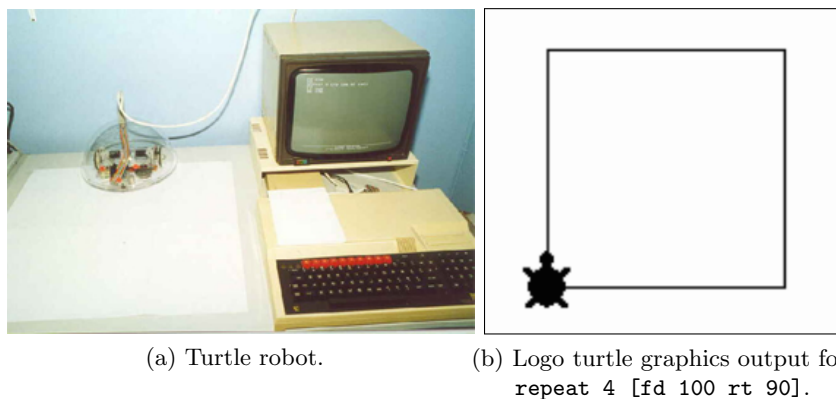


Figure 2.5: Logo Turtle.  
[Logo Turtle [41].]

construction sets with Logo and enabled the animation of Lego constructions via sensors and motors [115]. While these systems were still connected to a traditional computer through a

wire, the later released Programmable Brick incorporated its own computer [77]. Programs were developed on a desktop computer and then transferred to the Brick, which would execute them. LogoBlocks is a VPL designed for the Programmable Brick and a predecessor of Lego Mindstorms [11].

### Etoys

Etoys, previously known as Squeak, is an educational programming environment, which has been mainly influenced by Logo [56]. It features a variety of objects, which are programmable via tiles. These tiles can be dragged and dropped into place to compose a program. The only control structure available are simple if-statements and besides mouse events there is a predefined set of buttons to stop and start the behavior of objects. Figure 2.6 shows a script of a car object with a tile to move the car along the x-axis with a given speed.

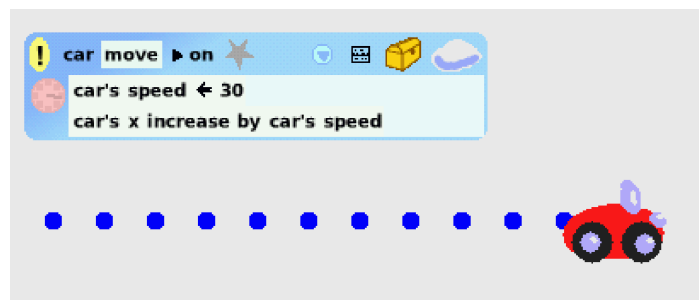


Figure 2.6: Etoys<sup>3</sup>.

### Scratch

Scratch<sup>4</sup> is a graphical programming language and environment developed by the Lifelong Kindergarten research group of the MIT Media Lab and has been developed especially for children and novices [114]. The core audience of Scratch ranges from children of eight years of age to teenagers of sixteen years of age [62]. In November 2015 Scratch ranked 30th on the TIOBE index<sup>5</sup>, which measures programming languages according to their popularity based on search engine results.

Scratch is intended for the use on traditional desktop computers with mouse as well as keyboard and its main goal is to support creative and systematic thinking using programming. Programs are composed by piecing together graphical Lego-style blocks that represent operators, data types, or functions etc. These blocks are organized in different command categories and color coded to highlight their membership. In addition to the blocks' color, their form further distinguishes them, as only synthetically compatible tiles can be connected with each other. Hence, incorrect statement combinations are impossible and syntactical load is being reduced [62]. Statements are connected vertically, while expression blocks are interlocked vertically.

In Figure 2.8 the various shape differences between inputs are shown. A visual clue whether a block can be dropped into a specific field, is done through an illuminating border around

<sup>3</sup> <http://www.squeakland.org/> (accessed 2015-06-26)

<sup>4</sup> <http://scratch.mit.edu/> (accessed 2014-01-30)

<sup>5</sup> <http://www.tiobe.com/> (accessed 2015-12-26)

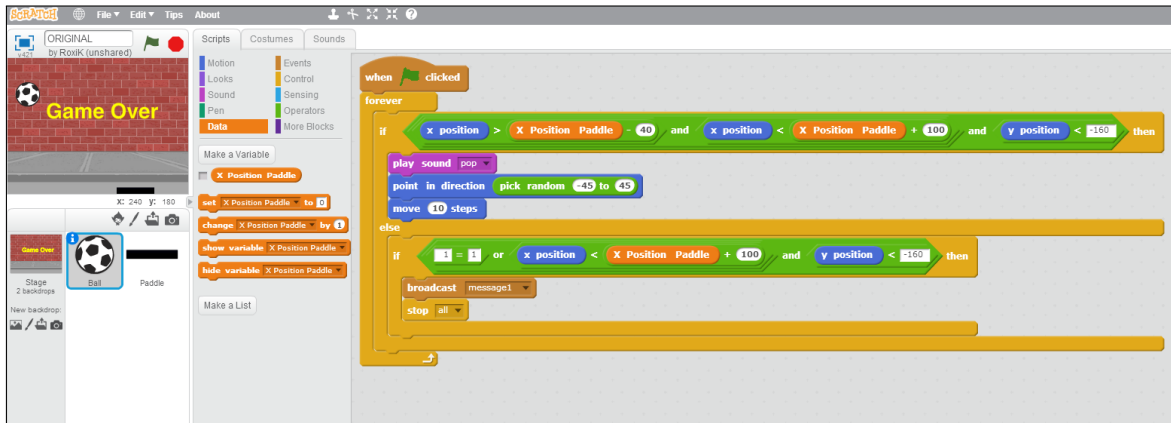


Figure 2.7: Scratch's multi-pane view.

said field. Figure 2.9 depicts parts of a program written in Scratch consisting of interlocked bricks of various categories.

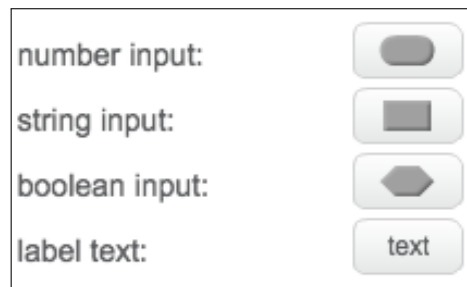


Figure 2.8: Visual differences between brick types in Scratch.

For the purpose of a facilitated navigation a single-window multi-pane interface is employed as detailed in Figure 2.7. In the browser-based version, the left upper pane contains the stage, where the program code is executed. Underneath that the sprite grid, showing the project's available objects, is located. The content of the pane on the right hand side depends on the tab selection in the middle. If the *Scripts* tab is selected, the command palette as well as the scripting area is shown. The command palette is organized in colored categories containing the programming blocks [62].

Lewis [66] compared teenagers using Scratch and Logo and showed that students perceived the tasks in both environments equally difficult. In addition, the study uncovered that Logo is better suited for the comprehension of loops, while Scratch is more useful in teaching conditionals.

Dwyer et al. [30] studied how student use visual cues when reading block-based programs; in particular, they utilized a variant of Scratch. They found that users take advantage of visual block attributes, such as color or shape, to predict the block's functionality. McKay and Kölling [81] utilized a prototyping tool to compare the time on task for various problems

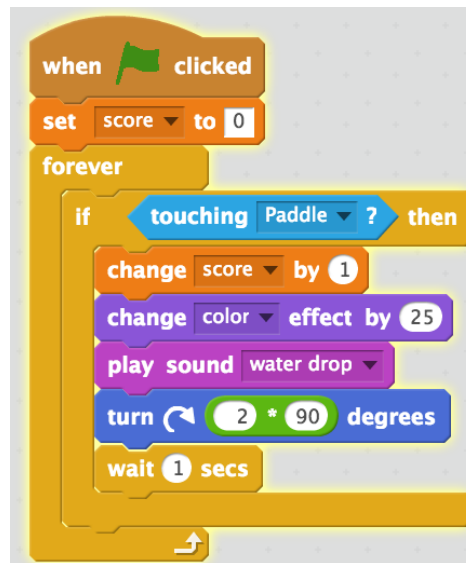


Figure 2.9: Parts of a Scratch program.

including, for example, Scratch, NetBeans Java Editor, LEGO Mindstorms NXT and Python. Their study showed that textual languages are better suited for insertion and replacement, while block languages are better for deletion and movement. Furthermore, the results indicate that there is a large variance between block languages. In Scratch, for instance, viscosity stems from how blocks stick together, meaning that when dragging a block from the middle of a composite brick there are blocks connected to it, which in turn have to be detached and reassembled within the block structure. Furthermore, their study showed that block languages handle text and numeric literals differently, i.e. some use individual blocks while other use text fields. As restriction to their evaluation is the fact that their experiment did not take into account the time necessary to plan the construction of the solution.

### Build Your Own Blocks/Snap!

Snap!, formally known as Build Your Own Blocks<sup>6</sup>, is a Scratch derivative. The JavaScript browser-based programming environment features graphical bricks to compose animations, games and other applications (see Figure 2.10). However, while Scratch is intended for a younger audience, the target user group of Snap! is not limited to children, as they expand Scratch's functionality for example with lambdas and recursion.

### Applications for younger children

Recently, ScratchJr was released, which is an iOS app based on Scratch to allow young children (ages five to seven) to learn programming. Other applications for younger children include Hopscotch or Daisy the dinosaur, which are all drag-and-drop VPL as shown in Figure 2.11 and Figure 2.12.

<sup>6</sup> <http://snap.berkeley.edu/> (accessed 2014-01-30)

<sup>7</sup> <https://itunes.apple.com/> (accessed 2015-04-23)

<sup>8</sup> <https://itunes.apple.com/> (accessed 2015-04-23)

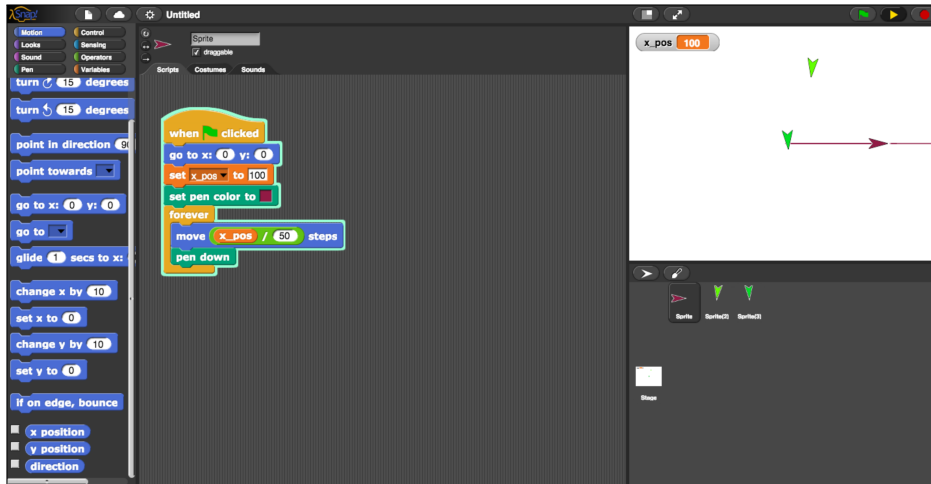
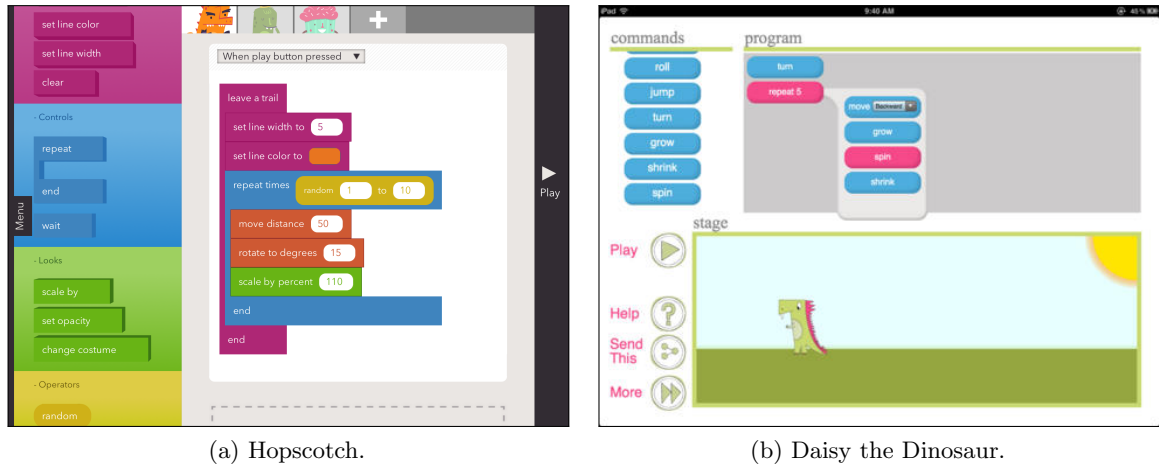


Figure 2.10: Snap!



Figure 2.11: Scratch Jr.<sup>7</sup>



(a) Hopscotch.

(b) Daisy the Dinosaur.

Figure 2.12: Coding apps for younger children<sup>8</sup>.

### 2.2.4.2 Rule-based Systems

Rule-based systems have been inspired by Prolog and do not state how to solve a problem as imperative languages do, but define states of the world and how objects should change in case conditions are met.

#### AgentSheets

AgentSheets<sup>9</sup> is a rule-based VPL and environment based on the spreadsheet metaphor with the purpose to teach programming. Users can create simulations by programming the behavior of sprites in a 2-dimensional grid-based world. Programs are developed via graphical rewrite rules which can be seen at the bottom right of Figure 2.13. For each rule the user selects conditions and determines the action to be taken in case these conditions are met. To define the consequences of the rule the user moves the agent to the position it should be in if the rule gets triggered [111].

#### KidSim

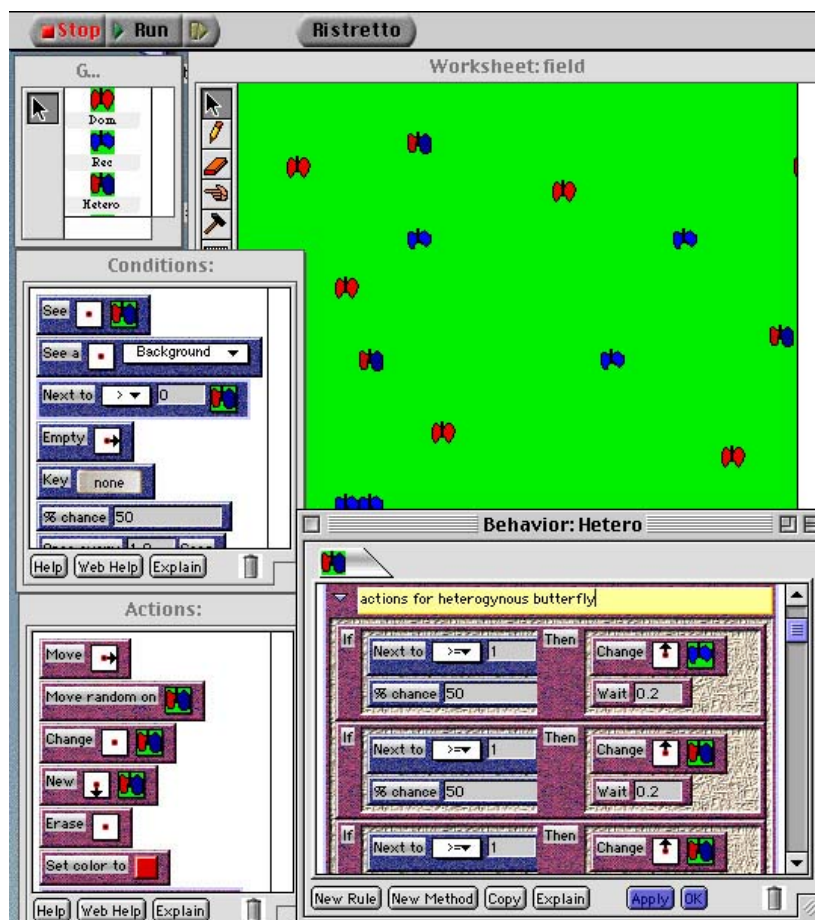
KidSim and its commercial version Stagecast provide a grid-based world, where children can create simulations and their own actors. Simulations are specified rules where the user defines the conditions to be fulfilled for a rule to apply and its consequences. Whenever the world matches the condition the rule is applied [130].

#### HANDS

In the design of HANDS the authors incorporated HCI knowledge and various principles to develop a programming environment for children. Besides common HCI heuristics, such as consistency, simplicity, or speaking the users language, HANDS incorporates further criteria mentioned in the Cognitive Dimensions Framework by Green and Petre [38]. These criteria

<sup>9</sup> <http://www.agentsheets.com/>(accessed 2014-01-30)

<sup>10</sup> <http://www.agentsheets.com/>(accessed 2014-01-30)

Figure 2.13: AgentSheets<sup>10</sup>.



encompass closeness of mapping, viscosity, or visibility. In particular, the first aspect has been of rather importance in the design of HANDS. The system provides a close mapping between the mental model of a problem solution and its programmatic solution. HANDS supports a hybrid approach, which allows visual as well as textual programming and incorporates context-sensitive menus showing the user the options available [102]. In HANDS objects are represented as playing cards, where the back of the card can depict the object's visual, while its front contains its properties (see Figure 2.14). To program the user inserts code into the dogs though bubble in the upper left corner. Whenever the play button in the middle is clicked and conditions are met, the rules are executed and the board in the middle of the screen is changed [100].

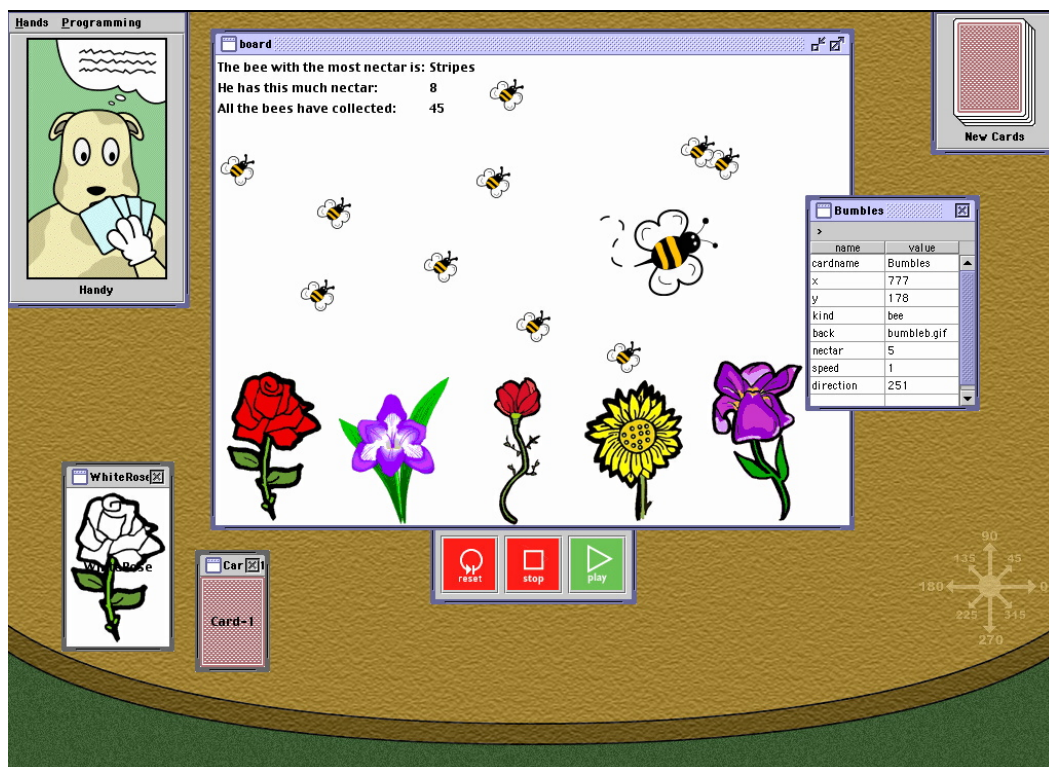


Figure 2.14: HANDS [100].

## ToonTalk

While visual programming languages are an improvement in children's computer education, Khan [54] argues that animated programs are better suited for representing dynamic behavior in programming languages. The programming environment ToonTalk is inspired by video games as according to the author they are easier to learn and children are likely to be familiar with them. Abstract computational concepts are mapped to concrete metaphors, e.g. a computation is a city, an agent or object is represented by a house and birds carry messages between houses. Methods are depicted as robots trained by the user to perform certain tasks. ToonTalk uses programming-by-example, with additionally requiring the user to remove details to achieve generality. Arithmetic operations are performed by a mouse as illustrated

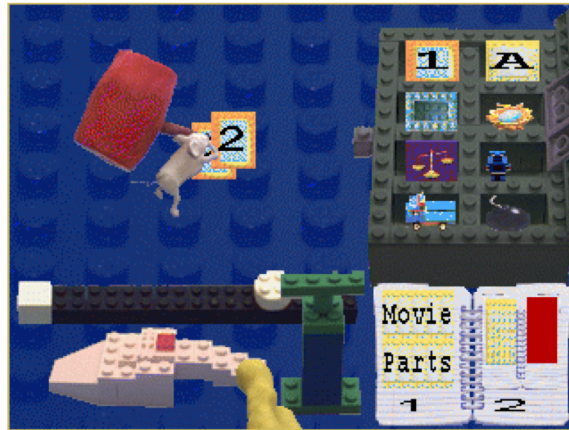


Figure 2.15: ToonTalk [54].

in Figure 2.15. If two number pads are put over top of each other, the mouse smashes those pads together with a big hammer and the result is a single pad containing the sum of the numbers. By placing  $\times 3$  on top of a pad, the number underneath is multiplied by 3. Text concatenation, for example, is performed in a similar manner, as text pads are stacked on top of each other [54].

### Alice

Alice is an object-based programming environment and language utilizing drag-and-drop to create 3D models. Figure 2.16 shows the programming environment incorporating the drag-and-drop programming style. The first version of Alice, Alice98 [26], was designed for non-science major college students. Alice provides a scene layout editor and a scripting tab, where the users can specify the behavior of the world. Generally, the programming language is Python with a few modifications e.g. it is case insensitive. Additionally, a set of operations for manipulating the 3D objects is available, such as *forward* or *up* to describe directions rather than XYZ coordinates. In order to speak the users language instead of “scale” for example “resize” is utilized. Further, commands can have varying levels of details. For instance, a move command can not only contain the direction, it can also specify the speed and duration of the operation or even on a very detailed level specify the interpolation style. Thus, novices can start from a rather simple level and once more proficient can employ more advanced features. To help users understand the source code, Alice animates the changes of the world [27].

### Kodu Game Lab

Kodu Game Lab is a visual programming environment for desktop computers and Xbox 360 and intended to be used by novices shown Figure 2.17. While other visual environments are executed in a two dimensional world, Kodu uses a 3D simulation environment as Alice does.

<sup>11</sup> <http://programacionyvideojuegos.blogspot.co.at/> (accessed 2015-12-26)

<sup>12</sup> [research.microsoft.com](http://research.microsoft.com) (accessed 2014-09-30)

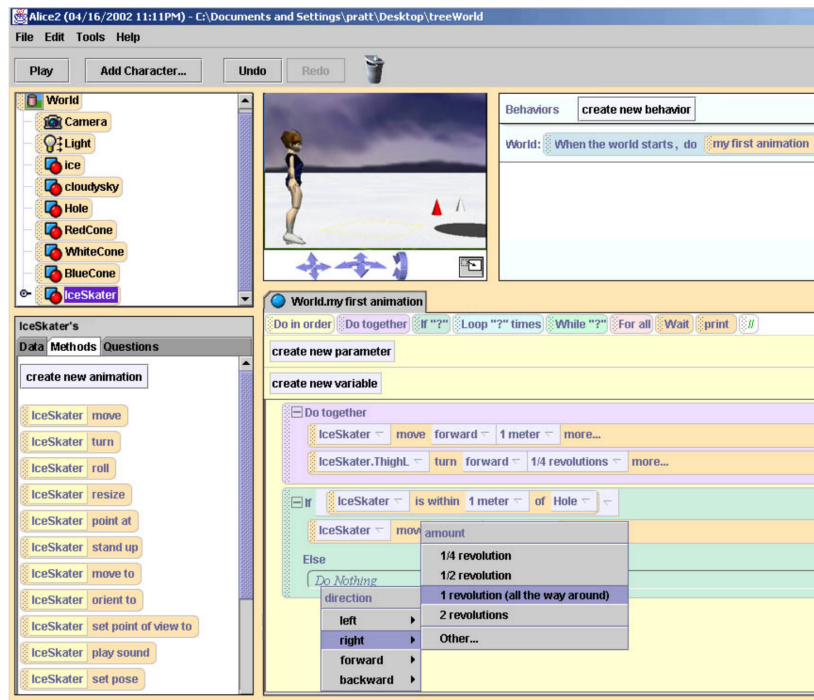


Figure 2.16: Alice 2<sup>11</sup>.

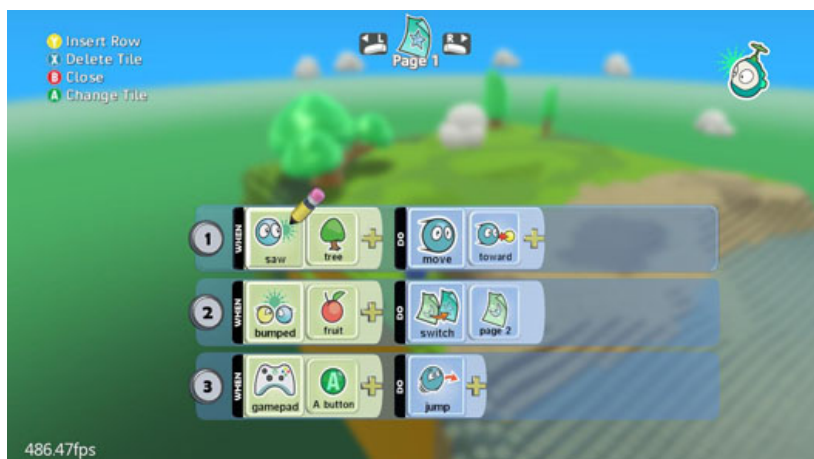


Figure 2.17: Kodu Game Lab<sup>12</sup>.

### 2.2.4.3 Mobile App Programming Environments

Recently, several projects have launched facilitating the creation of applications for mobile devices. Some perform the actual programming on traditional desktop computers while others have been designed to be used on hand held devices themselves.

#### TouchDevelop

TouchDevelop<sup>13</sup> is a programming language and environment designed for mobile devices, which allows users to develop mobile applications utilizing the device's sensors, stored photos, or music [135]. The textual programming languages is built upon imperative and object oriented features. An application comprises global variables, the user interface elements, and a set of methods.

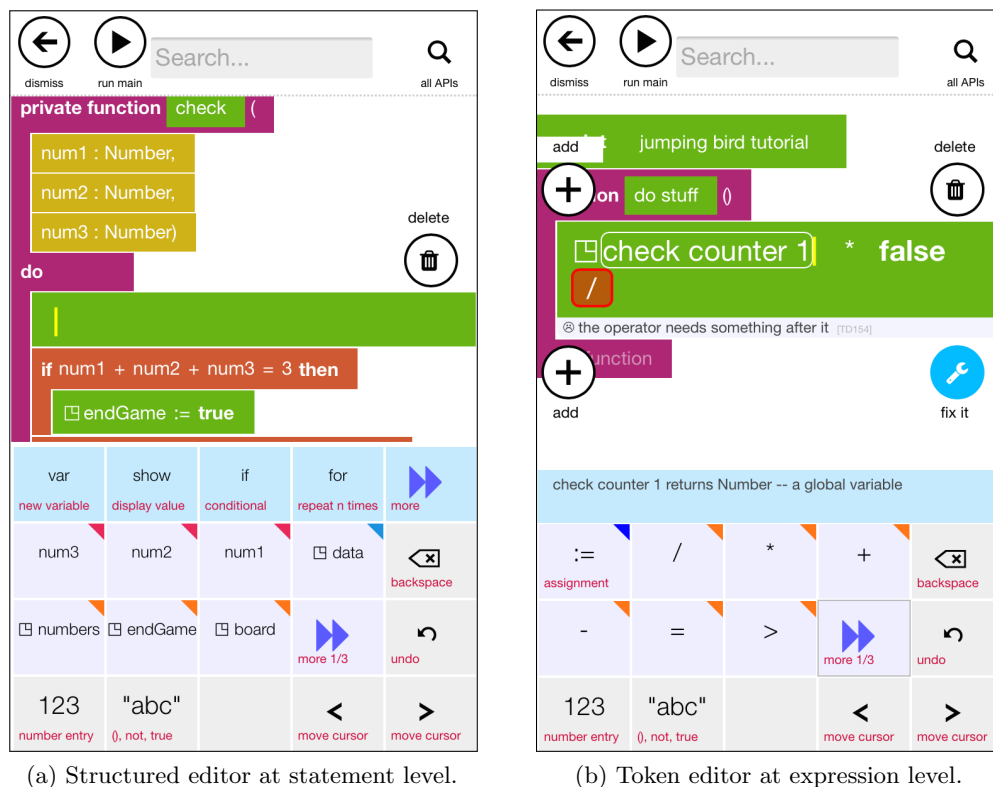


Figure 2.18: TouchDevelop's editors [135].

TouchDevelop has two types of editor styles depicted in Figure 2.18, since mobile devices lack a precise keyboard, which increases the difficulty to code programs with a strict syntax. First, source code on the statement level is programmed within an editor enforcing a certain structure and thus eliminating the possibility to create syntactically invalid statements. Second, on the expression level a token editor with auto completion is used. As can be seen

<sup>13</sup><https://www.touchdevelop.com/> (accessed 2014-01-30)

in Figure 2.18b in case there is a syntactical error, TouchDevelop provides an error message at the bottom of the text field and a button to resolve the issue. Furthermore, each editor button has additional information in a textual form at the bottom.

## App Inventor

MIT's App Inventor<sup>14</sup> for Android is a visual programming platform that targets novice programmers and enables the creation of applications for the Android platform [86]. However, for program development, App Inventor requires a regular computer with a full Java runtime environment, i.e. applications are created on a personal computer and deployed to mobile devices. App Inventor utilizes a VPL like Scratch, where programs are created by snapping code blocks together. Users generate the user interface with the Component Designer, which allows to drag interface objects into a view and change their properties as can be seen in Figure 2.19. The app's functionality is added with the Block Editor. The Block Editor as shown in Figure 2.20 offers two palettes, one containing the blocks suitable for the components already added to the app and the other one comprising of standard blocks (e.g. control flow statements, logical operators, or lists). More experienced users can even create and share their own blocks, augment their apps with textual source code and even program custom classes. Since the destination is a mobile device, App Inventor provides an emulator to test the applications [143].

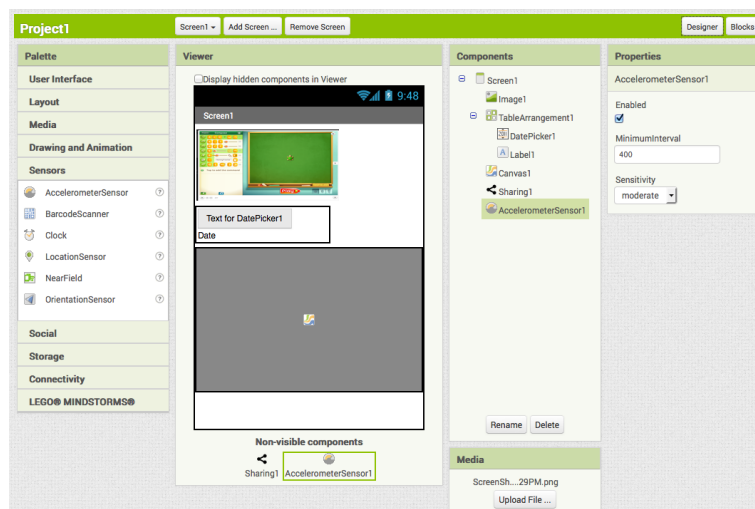


Figure 2.19: App Inventor Component Designer.

## Stencyl

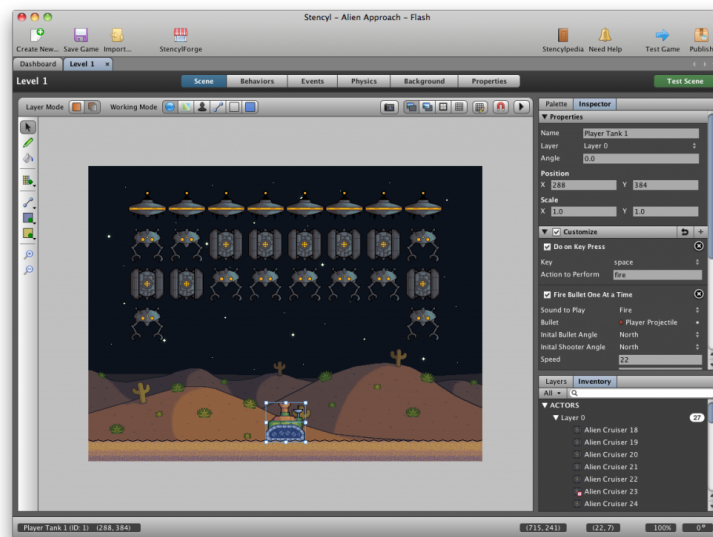
Stencyl is block-based environment incorporating a similar idea as App Inventor, i.e. games are developed on a desktop computer but executed on mobile devices. In contrast to App Inventor, the target system is not limited to the Android operating system. Figure 2.21 shows a scene layout in Stencyl.

<sup>14</sup> <http://ai2.appinventor.mit.edu/> (accessed 2015-12-28)

<sup>15</sup> <http://www.stencyl.com/> (accessed 2015-12-28)



Figure 2.20: App Inventor Block Editor.

Figure 2.21: Stencyl.<sup>15</sup>

## Tickle

Recently<sup>16</sup>, the mobile programming environment Tickle<sup>17</sup> has been released for iOS devices. As can be seen from Figure 2.22a it features similar colored bricks as Scratch does in addition to being able to access the device's sensors. In contrast to Pocket Code, Tickle creates formulas visually by interconnecting blocks. As can be seen in Figure 2.22b on small devices

<sup>16</sup>Note that this mobile app has been released after the usability study has conducted.

<sup>17</sup><https://tickleapp.com/> (accessed 2016-02-26)

the formulas can quickly exceed the screen, i.e. the user has to scroll in order to see certain parts of the formula. Depending on the size and complexity of the formula this can become an issue, in particular, since there is no zoom capability implemented.

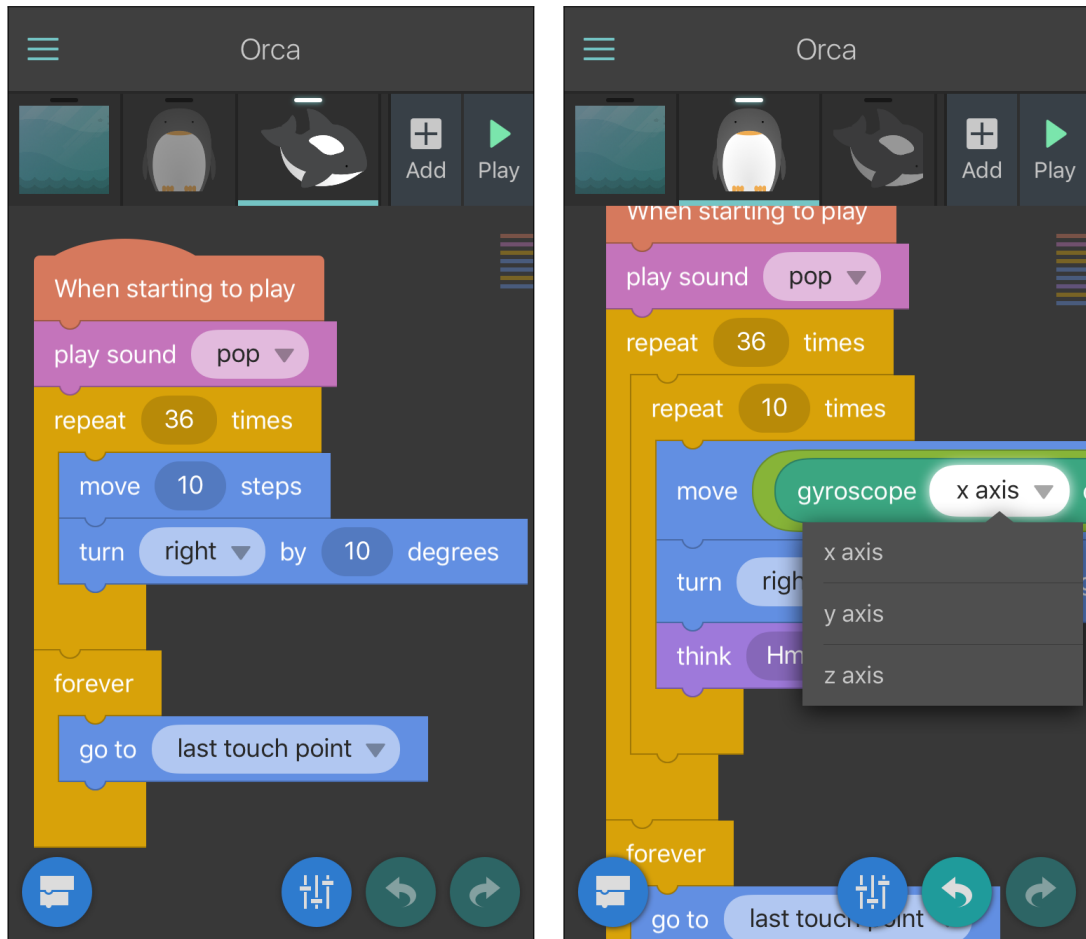


Figure 2.22: Tickle.

## 3 Theoretical Background

---

*“When a product or service is truly usable, the user can do what he or she wants to do the way he or she expects to be able to do it, without hindrance, hesitation, or questions.”* [117]

Usability and User Experience (UX) have emerged as buzz words within the last two decades, especially within the context of websites and other software products. Even though usability and UX are essential to the success of any product or system, they differentiate in their aim. Usability focuses on the individual’s ability to successfully and efficiently perform tasks, while UX addresses the user’s interaction with the system as a whole and comprises the user’s perception, beliefs, and feelings [65]. Since UX is interested in the overall picture, the efficiency and ease of use as defined in usability are an integral part. A last term often used within the context of creating user-friendly products, is user-centered design (UCD). UCD is a broad term describing a process which emphasizes the user’s role as the focal point of the design and evaluation process [97]. There exist various methods ranging from field studies, to usability inspection and evaluation methods, surveys, iterative and participatory design within the context of UCD [3].

The usability of any product or system comprises of a set of different characteristics, which all influence the ease of use. The ISO standard 9241 part 11 [1] defines the concept of usability as follows:

*“Extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.”* [1]

This definition puts emphasis on the user, the goal a user is a trying to achieve with his actions, and the context the product is used in. The users’ characteristics as well as the task variations have the greatest influence on usability. Therefore, the identification of the target audience is a fundamental step towards the creation of an easy to use system. The term user comprises all different groups of people who are going to install, maintain, or work with the product. By definition knowing the users is difficult, since the system might aim at a large and therefore versatile group of people. To maintain the complexity at an appropriate level the age, education, computer background, and other characteristics of the users are essential criteria. One way of classifying the target users is the user cube as described by Nielsen [93], which is depicted in Figure 3.1. According to this, users can be categorized in regard to their experience with the system, with computers in general and with the task domain. On the system experience scale the users can range from *novices* to *experienced users*. Novices have their initial encounter with the system, while skilled or expert users already have a high proficiency concerning the product. Even expert users, however, normally do not acquire



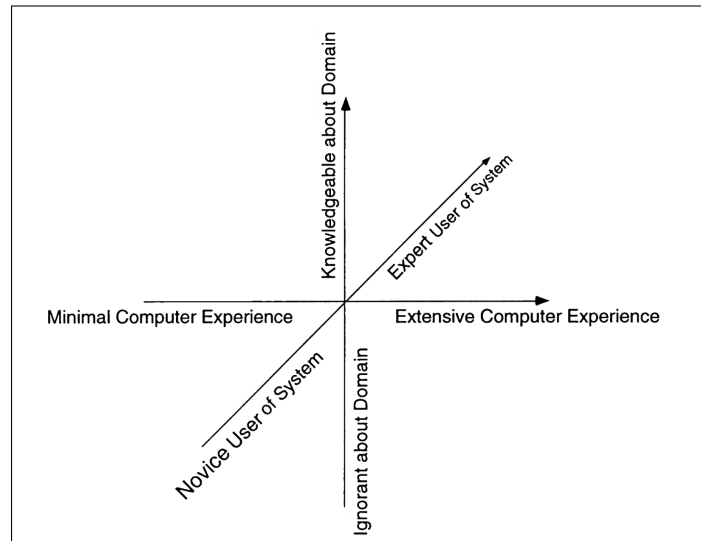


Figure 3.1: Three dimensions of user experience [93].

mastery in every portion of the system and might be novices in regard to certain aspects. In contrast, *power users* or *super users* have a detailed knowledge of the product, even though they might only require certain parts of the system. Lastly, we can define *casual users* as having experience with the system, yet using it less frequently. In regard to the user cube, the interface design further depends on the users' general computer knowledge, e.g. input entry methods, as well as knowledge of the target domain, e.g. specialized terminology.

The ISO definition explicitly states that the users' goals are an aspect to consider, i.e. allowing the user to perform relevant tasks is essential. Specified context of use generally covers the users, tasks, equipment, and environment the product is used in. Furthermore, the ISO standard states three attributes—effectiveness, efficiency and satisfaction—determining the usability, which we discuss in the upcoming subsection.

### 3.1 Attributes

In order to measure the usability of a product, there are three attributes which tied into the definition of usability as given by the ISO standard 9241 part 11 [1, 9, 44]:

- *Effectiveness*: How completely and accurately can a user achieve specified goals.
- *Efficiency*: How quickly can a user achieve specified goals.
- *Satisfaction*: The user's perception of content, absence of discomfort, and positive attitudes towards the system.

Nielsen [93] identifies five distinct attributes of usability :

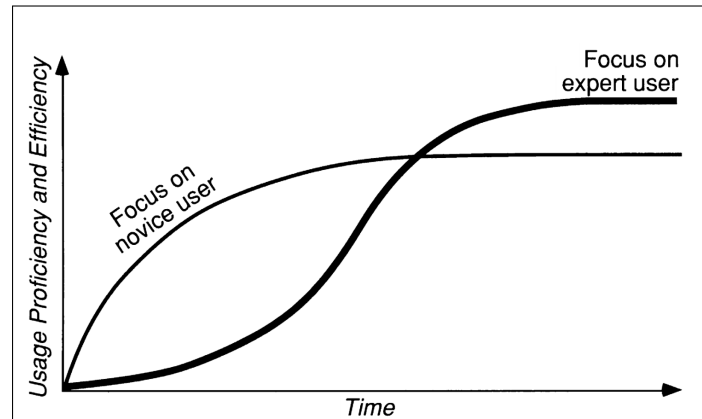


Figure 3.2: Learning curves for two hypothetical systems. One focuses on novice users, thus is easy to learn but less efficient to use. In contrast, it is more difficult to reach proficiency in the second system, but once the user has become an expert the system it is highly efficient [93].

- *Learnability*: Allowing novices to quickly acquire an expertise in using the system is an essential part to enable an efficient performance of tasks. The initial learnability of any product can be easily determined by taking novice users and measuring the time it takes them to reach a certain proficiency level. The proficiency can hereby be defined by a certain task the users should be able to complete.

Figure 3.2 shows two hypothetical systems, suggesting that a system is either easy to learn but in turn less efficient or hard to master, but eventually allowing users to be more productive with the system. A steep incline for the first part of the learning curve is an indicator for a highly learnable system. One way to create an easy to learn yet efficient system is to provide multiple interaction styles, one easy to learn and another one which is more efficient. In order to restrict the user interface's complexity novice users should be able to use the system without being confronted with expert modes [138]. Another possibility is to include accelerators, which are elements to perform certain tasks quicker, although there is a more general and possibly slower way to reach the same goal.

- *Efficiency*: Once the learning curve has flattened out the user has become experienced. The efficiency of the system then refers to level of performance of expert users [138].
- *Memorability*: Memorability refers to the characteristic that casual users do not need to learn a system from scratch, but remember how to use it based on their previous learning.
- *Errors*: Wilson [138] defines an error as any action that does not accomplish the desired goal. Ideally, the system has a low error rate, no catastrophic errors and in case an error occurs the user can recover from it quickly.
- *Satisfaction*: The users should feel subjectively satisfied when using the system. This is of utter importance in non-work related systems, since the enjoyment of using the

system is the main goal users pursue.

While the ISO standard does not regard *Learnability*, *Memorability* and *Errors* as usability attributes, it can be argued that they are implicitly included within efficiency, effectiveness, and satisfaction [44].

Rubin, Chisnell and Spool [117] provide another attribute list comprising of *Learnability*, *Efficiency*, *Usefulness*, *Effectiveness*, *Satisfaction* and *Accessibility*. Four out of the six attributes have already been discussed. *Usefulness* focuses on the possibility to achieve a certain goal with the product. Hence, this ties in with the ISO definition in regard to fulfill a certain intend. *Accessibility* refers to the possibility of accessing a product for users with special needs.

It is not possible to achieve all usability attributes to a full extend simultaneously, hence trade-offs are inherent; to avoid catastrophic failures the efficiency of the system might drop or the complexity of the overall design increases in order to ensure *Learnability* and *Efficiency* at the same time. Decisions on trade-offs should be made in regard to the usability goals and aspects of the project [138].

## 3.2 Evaluation Methods

In order to determine the usability of a product, one can measure the conformance of the system with various usability attributes. On the one hand there are inspection methods where the user interface is evaluated in accordance to a set of guidelines or heuristics. On the other hand empirical assessments involve actual participants from the target user group examining the system.

Usability inspection methods involve an evaluator assessing an interface and attempting to uncover and rate usability issues occurring in a design. The advantages of inspection methods include the possibility to apply these techniques early in the usability engineering life cycle, as they do not necessarily require a working prototype. Further, by employing usability experts as evaluators no actual users have to be acquired. Nielsen [94] lists seven inspection methods:

- *Heuristic evaluation*: A small set of usability experts examine each screen element in compliance to a list of predefined usability heuristics.
- *Cognitive walkthrough*: In a detailed process the user's interaction with the interface is simulated to determine whether it is possible to reach the correct next stage.
- *Formal usability inspections*: Combines heuristic evaluation and cognitive walkthrough.
- *Pluralistic walkthrough*: A group of representative users, product developers, and usability specialists walk through a scenario step by step together.
- *Feature inspection*: Typical tasks are broken down into sequences in order to find difficult or long series.

- *Consistency inspection*: Checks consistency among designs, to ensure that the similar tasks can be performed in similar ways.
- *Standards inspection*: Inspects whether an interface fulfills a set of predefined standards.

While heuristic evaluation, cognitive walkthrough as well as feature and standard inspection are performed by a single evaluator at a time, pluralistic walkthroughs and consistency inspections are performed in a group setting. Formal usability inspections are a combination of individual and group assessments. In Section 3.3 we focus on a more detailed description of heuristic evaluation, as for the practical solution a heuristic evaluation of the formula editor in Pocket Code was conducted.

In contrast to inspection methods, empirical evaluations require participants from the target audience, which on the one hand makes them more laborious but on the other hand give insight into how real users might operate the system [93]. There are various usability testing methods, such as:

- *Thinking-Aloud*: The participant articulates his thought process while performing tasks with the system under test.
- *Co-discovery*: Two users work together with the system to fulfill the tasks.
- *Performance Measurement*: The user is asked to achieve various goals with the product and quantitative measurements are taken.
- *Wizard-of-Oz Method*: Complex functionalities are performed by a person rather than the system itself without letting the user know.

In particular, we will explain evaluations measuring performance in Section 3.4 as in the second part of the practical we conducted a formal experiment comparing Pocket Code to Scratch in regard to formula manipulation.

### 3.3 Heuristic Evaluation

Heuristic evaluation is an usability inspection method, depending on the judgment of usability specialists. Each evaluator assesses the system in regard to a set of heuristics, which are simplified principles or commonsense rules. In order to get the most out of this technique ideally multiple reviewers examine the interface, as a single person would likely not uncover all issues. Additionally, by using several evaluators different sets of usability problems can be obtained and it facilitates finding major issues [91].

An individual expert performs the inspection of the interface alone in order to avoid bias. During the session the evaluator inspects all screens several times and determines whether they comply with predefined usability principles. There are several possible foci of a heuristic evaluation [139]:

- An *object-based* evaluation concentrates on particular interface objects such as mobile screens, web pages, menus, controls, or error messages.

- In a *task-based* heuristic evaluation the examiners are equipped with a set of tasks to perform in order to identify problems.
- The *object-task hybrid* approach combines the two latter by letting the evaluators work through a set of tasks first and then have them examine certain user interface (UI) objects in regard to the heuristics.

While the reviewers are often permitted to decide how they examine the system, it is advised to have them inspect the entire interface twice; once to become familiar with it and a second time to inspect the product against the set of heuristics. Once each of the evaluators has inspected the interface and reported the found issues, an aggregated list of issues is being created, including all obstacles found. Difficulty arises when evaluators report a problem at different levels of granularity as both descriptions have to be combined into a single issue. In a subsequent step each reviewer ranks the aggregated issues according to their severity without being aware of the ratings of the other evaluators in order to avoid bias. The individual severity ratings are averaged to obtain the final score for each problem.

### 3.3.1 Evaluators

As reported by Nielsen [91] there are major differences among distinct evaluators due to their individual experiences. In his case study the author had a single interface examined by three evaluator groups:

- *Novices* with general computer experience but no usability expertise.
- *Single specialists*, i.e. general usability experts.
- *Double specialists* with usability expertise in general as well as with the particular interface type.

Nielsen [91] was able to show that usability specialists perform better than non specialists. Ideally, evaluators are double specialists, i.e. they are usability as well as specific interface experts. Wilson [139] for example states that evaluators who have more than ten years of experience in the usability domain produce similar results in a heuristic evaluation to a usability test.

Cockton et al. [24] even provide further detail on the recommended evaluator knowledge:

- *User knowledge* (e.g., knowledge of expertise and skills)
- *Task knowledge* (e.g., knowledge of the tasks users want to perform)
- *Domain knowledge* (e.g., knowledge of the target domain)
- *Design knowledge* (e.g., knowledge on usability and UX)
- *Interaction knowledge* (e.g., knowledge on how the users work with the system)
- *Technical knowledge* (e.g., knowledge on hardware or software)
- *Product knowledge* (e.g., knowledge of the features).

In a study on six projects Nielsen and Molich [96] determined that on average single evaluators found only 35% of the usability problems in the interfaces. Thus, a general recommendation is to have three to five reviewers in order to identify the majority of issues, i.e. between 60 % to 75 %. Figure 3.3 depicts the percentage of issues found by various numbers of evaluators.

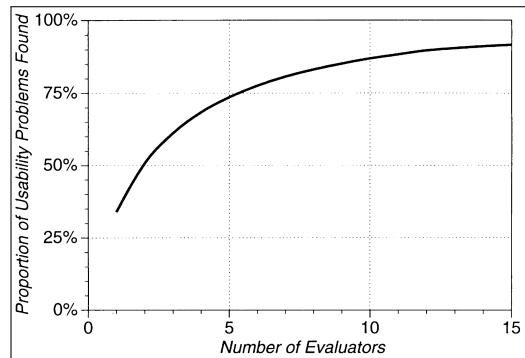


Figure 3.3: Ratio of usability problems found as more evaluators are added [91].

### 3.3.2 Heuristics

The usability heuristics do not only provide the basis for evaluating a system, but following these guidelines during the design phase ensures that the system incorporates proven principles and features a recurrent interface design which increases recognizability.

There are different kinds of guidelines [93]:

- *General* guidelines contain essential rules relevant for various domains, user groups, and tasks, e.g. “Provide feedback to the user”.
- *Category-specific* principles depend on the system type, i.e. some heuristic guidelines might be inherent to the application. In the context of mobile usability and Android applications for example, a guideline to follow would be “*Use illumination and dimming to respond to touches, reinforce the resulting behaviors of gestures, and indicate what actions are enabled and disabled*”<sup>1</sup>.
- *Product-specific* guidelines apply to the product itself. For example, e-learning software for children might include a heuristic “*The screen design appears simple, i.e., uncluttered, readable, and memorable*” [5].

Nielsen and Molich [96] developed the original set of heuristics, which Nielsen revised to include the following ten guidelines as shown in Table 3.1.

<sup>1</sup> <http://developer.android.com/design/index.html>

	<b>Heuristic</b>	<b>Description</b>
1.	<i>“Visibility of system status ”</i>	<i>“The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.”</i>
2.	<i>“Match between system and the real world”</i>	<i>“ The system should speak the users’ language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.”</i>
3.	<i>“User control and freedom”</i>	<i>“ Users often choose system functions by mistake and will need a clearly marked “emergency exit” to leave the unwanted state without having to go through an extended dialog. Support undo and redo.”</i>
4.	<i>“Consistency and standards”</i>	<i>“ Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.”</i>
5.	<i>“Error prevention”</i>	<i>“ Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.”</i>
6.	<i>“Recognition rather than recall”</i>	<i>“Minimize the user’s memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialog to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.”</i>
7.	<i>“Flexibility and efficiency of use ”</i>	<i>Accelerators—unseen by the novice user—may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions. ”</i>
8.	<i>“Aesthetic and minimalist design”</i>	<i>“Dialogs should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.”</i>
9.	<i>“Help users recognize, diagnose, and recover from errors.”</i>	<i>“ Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.”</i>
10.	<i>“Help and documentation”</i>	<i>“ Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user’s task, list concrete steps to be carried out, and not be too large. ”</i>

Table 3.1: Nielsen’s [92] usability heuristics.

In addition, the reviewers can point out issues which do not necessarily relate to the predefined heuristics but pose threats to the usability of the design. Besides reporting interface issues the evaluators should point out positive aspects of the interface as well. The findings of an individual evaluator are either recorded in a report by the evaluators themselves or by an observer. After all evaluators have conducted the evaluation the findings are aggregated. The results is a list of usability issues of the interface in respect to the heuristics. Since the issues are reported as well as the heuristic they violate, it is fairly easy to some point to create recommendations in order to fulfill the guidelines. Another way is to have a debriefing session after the last evaluation has been conducted in order to brainstorm on possible solutions [9].

### 3.3.3 Usability Issues and Ratings

Whenever a user is unable to perform a task and the interface can be determined as the hindering factor, then we can categorize this as a user interface issue, i.e. UI issue. Generally, we distinguish between a usability issue and a problem, where the issue determines the underlying cause of the problem. Subsequently, the problem is a manifestation of the issue, which prevents the user to achieve or effectively complete a certain task. A usability issue for example irritates or confuses the user, causes mental overload, poor user performance, or an error [138]. An error is an incorrect action, which may lead to task failure [4]. For example, in case the user cannot register on a website, then the problem is the inability to register, while the issue might be inexplicit password characteristics. An error occurs, when the user tries to register with a password not fulfilling all necessary attributes.

Often resolving all found usability issues is not feasible due to time or monetary constraints. In order to repel severe or even catastrophic design and implementation mistakes, the issues have to be prioritized by usability experts. Each usability issue is rated by each professional based on the impact on the performance of the user. To counterbalance any bias it is advisable to have the experts note their severity ratings separately and then average the score for every issue over all the evaluator ratings.

In order to create a prioritization of issues, severity codes can be assigned representing their impact on the user experience. A typical code set might be [9, 93]:

- *Catastrophe* prevents task completion.
- *Major problem* has significant potential impact on the usability.
- *Minor problem* has a low priority, but should be considered.
- *Cosmetic problem* should be corrected in case there is time.

Another way to rank issues is according to their criticality as described by Rubin and Chisnell [117]. The criticality of a problem is an aggregated value composed of the problem severity and the frequency of occurrence. The problem severity is similar to the one introduced before, while the frequency of occurrence is an estimation of the percentage of total users affected



and the probability that a user from that affected group will experience the problem. Table 3.2 depicts the rating of the severity and frequency of occurrence. The criticality can then be computed as a sum over the frequency of occurrence and the severity. For example, an issue is categorized as a serious problem, thus rated with 3, but only occurs less than 10 % of the time, i.e. rated 1, the overall criticality of this problem is 4.

Ranking	Severity	Frequency of Occurrence
4	Task failure – prevents this user going further	Will occur $\geq 90\%$ of the time the product is used
3	Serious problem – may hinder this user	Will occur 51–89% of the time
2	Minor hindrance – possible issue, but probably will not hinder this user	Will occur 11–50% of the time
1	No problem – satisfies the benchmark	Will occur $\leq 10\%$ of the time

Table 3.2: Issue rating according to the problem’s severity and frequency of occurrence [117].

A further refinement would involve a third severity dimension besides the frequency of occurrence and the impact of the problem, namely the user’s experience with the system. Hence problems, which only occur with novice users, are judged less severe than ones, which affect also experts [117].

### 3.3.4 Advantages and Disadvantages

Heuristic evaluations can be a simple and relatively fast way to review an interface in comparison to a usability study involving extensive prior planning. Due to not relying on additional resources such as usability labs they are cost effective [139].

As mentioned one of the factors influencing the quality of an heuristic evaluation is the expertise of the evaluators. In regard to this it might be helpful to train people involved in a heuristic evaluation before hand. Depending on the experience of the evaluators, the inspections might focus on surface issues such as misalignment of controls or poor message texts while disregarding some substantial work flow issues. Further, due to the individual evaluator’s experience it can occur that every reviewer uncovers a different set of issues, thus, in a final meeting all the found problems should be discussed to determine their relative severity [139].

Especially in complex systems, heuristic evaluations alone cannot find the majority of issues. Furthermore, heuristic evaluations often produce false positives, i.e. identify issues which in fact are not an issue in actual use. This arises as the evaluators differ from the actual user group. Heuristic evaluators, even when working with task scenarios, are not immersed in the task in the same way as end users.

## 3.4 Usability Testing

Even though a variety of inspection methods are available, observing and measuring actual usage of a system with representatives of the target population provides valuable information and insights. In traditional lab-based usability evaluations, a facilitator leads the test and provides the tasks one at a time to the participants. Usually, tests are recorded in some form and additionally observers take notes to capture the users reaction in its entirety. Nielsen [93] identifies four stages of a formal usability test:

- *Preparation:* In order to provide the same starting point for all participants, the system has to be set back to its initial state. Further, questionnaires and task description should be prepared.
- *Introduction:* The facilitator describes the test objective and further explains that the system is under test and not the participant.
- *Test:* A usability test can be a stressful situation for participants, with mistakes and difficulties during the test adding to the discomfort. It is the facilitators responsibility to reduce the stress as much as possible. One strategy to create a comfortable environment for the user is to start with an easy task. This way an early success is likely. Ideally, the user performs the tasks without interaction with the facilitator. Whether or not the facilitator is allowed to help the participant in case they are having difficulties with a task, depends on the participants characteristics as well as the facilitators perception of the situation. On the one hand in order to achieve valuable insight on the real use of a system, providing help is not ideal as it is interesting to see how users recover from mistakes. On the other hand when the user is struggling providing a hint is appropriate in order to avoid the participant getting frustrated [93, 9, 117].
- *Debriefing:* After the test, the user provides answers to perceived satisfaction questionnaires and additional comments to the system.

### 3.4.1 Test Plan

The test plan is the main design part of a usability study and reflects on several aspects of the study [138]:

1. *What* is the objective of the study, hence what should be tested?
2. *Where* will the study be conducted? There are different possible locations: usability studies can be conducted in a usability lab, the field, or remote. Figure 3.4 shows a possible usability test set-up in a laboratory. Each location has advantages and disadvantages, for example, in lab tests the parameters are easier to control, however, they are less natural. Field tests might provide a more genuine setting, yet, distractions from the environment can influence the study. Recently, moderated and unmoderated remote usability studies have increased in popularity. In remote studies the test participants are geographically separated from the study conductor.

3. *How to test?* Depending on resources available and time constraints the overall test procedure has to be determined.
4. *Who to test?* In order to have a valid study, selecting the participants in accordance to the target group is essential. Thus, deciding on a sampling strategy and user characteristics such as age, gender, education, and experience is essential to conduct the test with the right participants.

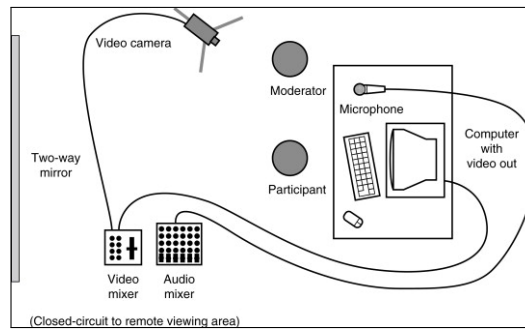


Figure 3.4: Usability test set-up [138].

Before conducting usability studies on a larger scale it is advisable to test the study design. Pilot tests fulfill exactly this purpose and should identify confusing task descriptions, incorrect time estimates as well as hardware or software issues. The pilot user is selected from the target group of the study and completes the entire test scenario. Ideally, the study design is evaluated well in advance to adapt the test plan if necessary [93].

### 3.4.2 Test types

Depending on the progress in the product life-cycle different usability studies are being conducted. Generally two main types of usability tests can be distinguished [93]:

- Formative (or exploratory) studies
- Summative (or assessment) studies

The former relates to smaller experiments which try to detect usability problems and their solutions. They are used throughout the iterative process of developing a system and results are used to improve the product. Formative studies examine the effectiveness of high-level aspects and can be conducted in informal settings. Thinking-Aloud tests are a very common formative test, where the users are asked to narrate their thought process during the exploration of the system. This verbalization allows the test conductors to get information on parts of the system which cause problems or are unclear. Conveniently, this sort of tests do not require a fully developed system but can be conducted using paper mock-ups or prototypes. However, since the users are asked to multi task, i.e. explain and actually perform the assignment, task times can be misleading [138].

While the latter still identifies usability issues the tests are usually conducted later in the life cycle and use metrics in order to quantitatively measure the usability. Summative studies

can be categorized in benchmark and comparison tests. Benchmark usability tests describe a system in regard to a set of benchmark goals and are often also referred to as validation or verification tests [117]. In comparison studies more than one application is being evaluated. This can either be different versions of the same applications or competing products. Within this section we mainly focus on the theoretical background of comparison studies, since we conducted a summative study comparing Pocket Code to Scratch in Section 4.2.

### 3.4.3 Sampling

A crucial aspect within the design of a usability study is the set of test participants, as its composition determines the prediction value for the target population. In most cases we do not know every individual user group present in the population, in these cases sampling is referred to as non-probability sampling. In contrast in probability sampling we know the number and characteristics of the population and are able to randomly pick participants with the same probability. The most common non-probability sampling method is convenience sampling, i.e. the test users are selected due to some factors such as close proximity, affiliation relations, etc. Note that this type of participant selection introduces bias [121].

Furthermore, variation of the sample can affect the study outcome. Variation in this case refers to the degree of how different the test users are expected to perform the tasks. In a homogeneous user group participants are more likely to perform in a similar manner, which in turn facilitates observing a statistical difference in case one exists.

Determining the right sample size is essential in order to create a successful study. However, there is no consensus in the literature on a recommended number of participants. Nielsen and Landauer [95] state that the first five participants will uncover about 80% of the usability issues. Albert and Tullis [4] restricted these recommendation to studies with a limited evaluation scope where the target users are well represented within the sample and Molich et al. [85] showed that six participants reveal far less than 80 %.

Comparison studies allow for two different ways to employ participants:

- *Between-subject design*: In between-subject studies every test user is assigned to a single application or version. Due to the individual variability of the participants and the fact that each user only tests a single system, a larger number of testers is required.
- *Within-subject design*: Within-subject user studies eliminate the individual variability as every user examines all systems. These repeated or paired measures provide stronger evidence to compare designs and thus require a smaller sample size to observe a difference. In order to reduce the effect of a learning bias between applications, it is recommended to interchange the application test order. Since every participant evaluates each product the data of a system is computed relative to the other.

#### 3.4.4 Tasks

A task is a small assignment, which can be achieved in a relatively short period of time, since a test session should last approximately around an hour. It is important to choose tasks representative for the user group under test in order to achieve study validity. In general, we can state that the tasks should be reasonable, specific, doable, in a logical sequence, and moderately long [138]. When designing the test, the goal for every assignment has to be determined as well as a minimum number of steps necessary to achieve this goal. Ideally, the user receives each task on a different piece of paper, one at a time. This eliminates the possibility of having the moderator using a different wording when describing tasks and allowing the participants to refer to the task description again. When planning the task sequence it is advisable to select a simple task for the beginning, as completing a task allows testers to gain confidence in their ability to use the system [93].

#### 3.4.5 Quality of Testing

There are several biases introduced in studies such as participants, tasks, the environment, facilitators, or the methodology [4]. Reliability and validity determine the quality of a usability study. Reliability refers to the repeatability of the results. Confidence intervals and hypotheses testing give us information on the soundness of the study. Validity refers to the power of the study to find actual usability issues, thus the validity depends on the sample and the set of tasks [121]. Internal validity focuses on the study design itself and whether the data has been analyzed correctly, while external validity deals with whether the study can make claims for the population based on the sample [31]. Generally, utilizing skilled users reduces the performance variability and it is preferable to have a homogeneous user group, however, one is not to reduce the external validity if the population is less homogeneous [121]. Confusion among the participants will add measurement variance, thus, providing clear task description is advisable.

Due to the fact that tests are artificial situations, the results of a study are not necessarily a guarantee that a system is usable or the opposite of that. Further, participants are rarely a representation of the entire population and it is well known that participants rate products in post-test or post-task questionnaire as easier to use than they have truly experienced it [117, 9].

Molich et al. [85] published several Comparative Usability Evaluations. These studies showed that in general only a fragment of issues is uncovered, usability experts do not generate better results in inspections than participants in usability studies and even professionals make study design errors.

#### 3.4.6 Measuring the User Experience

Usability is an essential success factor in any product or system. Thus, measuring it reliable and effectively is critical. In this section, we discuss various metrics, which can be utilized to assess usability.

There are various performance metrics to determine usability as well as issues and their magnitude. Ideally, we can associate the measurements with the attributes of usability, which we defined in Section 3.1 [138]. For example, task success measures the *Effectiveness* to which users are capable of achieving a given goal. Furthermore, the number of *Errors* reflects the mistakes the user made due to confusing or misleading interface parts. Time-on-task is the most common performance metric, measuring *Efficiency* as the time elapsed between the start and end of a specific task. Another *Efficiency* quantity includes the effort associated with completing the task, e.g. number of clicks on a website.

In order to determine the user's perception of the system, a number of standardized tests have been developed assessing usability at a system as well as task level. *Learnability* and *Memorability* can be investigated by conducting several studies across different points in time on the same test users. By comparing the results from more recent tests, one can determine whether the performance changed over time. Evaluating several metrics allows us to obtain an overall sense of the usability of a system [138].

After a usability test traditionally the following data is available:

- *Background information*: A pre-test questionnaire provides a way to get additional relevant information about each participant.
- *Notes*: The test observers take notes during the test which might already contain possible usability issues.
- *Quantitative data*: Collected data on times, task completion, errors, etc.
- *Self-reported data*: From post-task and post-test questionnaires and interviews.

#### 3.4.6.1 Quantitative Data

Quantitative data is a type of performance measurement and can include the following variables [93]:

- Time on task.
- Number of tasks completed within a given time span.
- Time to recover from errors.
- Number of user errors.
- Number of functions utilized by the user.
- Frequency of use of the help or manual.
- Frequency of positive and critical statements.
- Preference statements.
- Time the user is not interacting with the system.

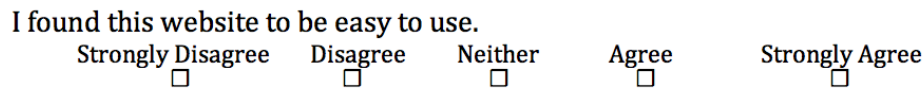
Statistics provide a way to generalize some of the behaviors observed or measured during a test and can be categorized into descriptive and inferential statistics. Descriptive statistics describe the sample data, such as mean, median, or mode. While the latter allows us to draw conclusions about the population beyond the data obtained during the test, such as statistical significance [4]. Depending on the data different statistical methods are suitable. Table 3.3 provides an overview of data types and some appropriate statistics.

Data Type	Metric	Statistical Procedure
Nominal (categories)	Task success (binary), errors (binary)	Frequency, Chi-square
Ordinal (ranks)	Severity ratings	Frequencies, Chi-square, Wilcoxon rank sum tests
Interval	Likert scale data, SUS scores	All descriptive statistics, t-tests, ANOVAs, correlation, regression analysis
Ratio	Completion time, time (visual attention), average task success (aggregated)	All descriptive statistics (including geometric means), t-tests, ANOVAs, correlation, regression analysis

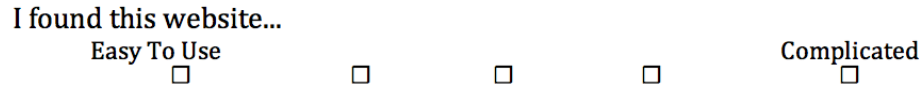
Table 3.3: Statistics for different data types and usability metrics [4].

**Task completion** provides a measure of how effectively the users can complete a given task with the system. There are two possibilities to categorize completion: either task completion is measured in a binary way as “success” and “failure”, or on a more granulated scale, where we define various levels. For example, we can generate three categories “Pass”, “Pass with assistance” and “Fail”. Of course the granularity is determined by the researcher and may include a finer differentiate, e.g. [4]:

- *Complete success*
  - *With assistance*
  - *Without assistance*
- *Partial success*
  - *With assistance*
  - *Without assistance*
- *Failure*
  - *Participant thought the task was complete, but it was not*
  - *Participant gave up*



(a) Likert Scale.



(b) Semantic Differential.

Figure 3.5: Rating scales in surveys.

A common way to analyze task completion rates is by task, as it gives a general overview of the system's effectiveness for each goal represented as a task.

**Time-on-task** is used to measure the efficiency of a product and is of special interest for actions which have to be performed repeatedly by the user. Time-on-task is the time elapsed between the start of a task and its end, usually expressed in minutes and seconds. Typically, for each task the average amount of time across all users is calculated. Due to the different abilities of the test users there might be a high variability leading to a skewed distribution and in turn can drag the average towards outliers. Therefore, reporting the confidence interval to show the variability in the data or providing the median or geometric mean give a more accurate representation of the data [4].

### 3.4.6.2 Self-Reported Data

The recording of self-reported data gives us the possibility to gain information on the user's perception of the system and how they evaluate their interaction. Self-reported data can be collected on a task-based level or at the end of the testing session. Usually, a Likert scale is used, which records the level of agreement between a statement and the user. Typically, a five or seven item interval is used, with the neutrality being one important aspect. Likert scales are often confused with semantic differentials. In the semantic differential technique the user is presented opposite adjectives and a series of points in between these bipolar pairs. The difficulties are twofold with semantic differentials: determining opposite terms can be difficult and it has been shown that the responses from users are more positive when asked in person than over an anonymous survey for semantic differentials [121]. Figure 3.5 depicts both methods side by side.

Questionnaires unfold their true potential when the task or test in question is followed immediately by the questionnaire completion. These user perceived ratings might change over the course of the study, thus, measuring in between each task, at the end of the test and averaging those results allow to capture a more wholesome picture. Otherwise, in case only a single questionnaire is completed at the end of the study, merely the last impression is recorded. Standardized measurements allow an objective interpretation of results and enable



**Overall, how difficult or easy did you find this task?**

Very difficult	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Very Easy
	1	2	3	4	5	6	7

Figure 3.6: Single Ease Question [118]

usability professionals to independently confirm other user studies. Further, the usage of standardized quantifications permit usability experts to focus on other aspects of the study rather than to find a suitable source of information. Again, reliability and validity are key quality attributes of standardized forms [121].

There are various questionnaires that have proven useful in usability studies. We have taken four relevant ones, namely *Single Ease Question*, *After-scenario Questionnaire* as well as *Subjective Mental Effort Question* for post-task ratings and the well known *System Usability Scale* as an example of a post-test questionnaire.

**Single Ease Question** (SEQ) is a 7-point post-task question, which assesses the perceived task difficulty [118]. Ideally, the user is asked immediately after each task, in order to ensure the maximal recall of their experience. As can be seen in Figure 3.6, the user should evaluate the general ease of completing the task. Even though some researchers utilize a 5-point scale, Sauro and Lewis [121] suggests a 7-point scale due to research on the relative reliability of these types of scales.

**After-scenario Questionnaire** (ASQ) comprises three questions, each on a 7-point scale from “Strongly agree” to “Strongly disagree”. The items focus on three important usability attributes, namely ease of task completion, time necessary to complete the task and quality of supporting material:

1. Overall, I am satisfied with the ease of completing this task.
2. Overall, I am satisfied with the amount of time it took to complete this task.
3. Overall, I am satisfied with the support information (on-line help, messages, documentation) when completing this task.

The score is determined by the average of the three questions [67].

**Subjective Mental Effort Question** (SMEQ) is a single item questionnaire with a rating scale from zero (“Not at all hard to do”) to 150 (“Tremendously hard to do”). In between there are seven more verbal labels. The questionnaire has been calibrated psychometrically against tasks [146].

**System Usability Scale** (SUS) is probably the most commonly used standardized questionnaire [18], which provides a composite measure of the overall usability of the system being studied. The SUS comprises ten statements in regards to the system’s usability on a 5-point Likert-scale between “Strongly disagree” to “Strongly agree” [18]:

1. I think that I would like to use this system frequently.

2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

The statement results can be easily combined to an overall score ranging from zero to 100 points, thus the SUS score can be utilized to compare products using confidence intervals. Some adjustments of the original SUS have been suggested. For example, item eight utilizes the term “cumbersome”, which has been shown to lead to confusion or misinterpretation, thus it is recommended to replace it with “awkward”. In addition, there exists a positive version of the SUS, i.e. a variant where all statements are phrased in a positive manner. The results for the this version are not significantly different to the original one, but lead to less mistakes among participants [121].

Albert and Tullis [4] suggest that an average SUS score under about 60% as relatively poor, while one over about 80% could be considered rather good. Sauro and Lewis [121] found that by analyzing close the 500 studies, the overall mean is 68% with a standard deviation of 12.5. Based on these results the author developed the grading scheme shown in Table 3.4.

SUS Score Range	Grade	Percentile Range
84.1-100	A+	96-100
80.8-84	A	90-95
78.9-80.7	A-	85-89
77.2-78.8	B+	80-84
74.1-77.1	B	70-79
72.6-74	B-	65-69
71.1-72.5	C+	60-64
65-71	C	41-59
62.7-64.9	C-	35-40
51.7-62.6	D	15-34
0-51.7	F	0-14

Table 3.4: Curved grading scale interpretation of SUS scores [121].

Another source of qualitative data are open ended responses, which are, however, more difficult to analyze. In particular, letting the user describe their experiences allows researchers to gain

insights in the aspects of usability important to the test user. Further, letting a participant articulate their opinion when comparing products is helpful to determine preferences [4].

### 3.4.6.3 Behavioral and Physiological Metrics

There are various metrics beyond task completion or perceived satisfaction. A wide range of subjective user opinions and preferences can be inferred from verbal (e.g. unprompted comments), non-verbal (e.g. facial expressions) or physiological measures (e.g. stress via heart rate or skin conductance) [12].

**Verbal Expressions** give information of participants' emotional and mental state, such as "This is easy" or "I do not know what to do". In particular, when comparing designs or products, computing the ratio of positive to negative comments allows to infer preferences, problem areas, and effective design decisions. Just as with task success, it is possible to further granulate the categorization, e.g. "questions", "frustration", "suggestions", etc.

**Eye Tracking** devices allow researchers to detect eye movement and hence determine where the users are looking at a specific moment, the path their eyes follow, and the time they spend looking at certain UI elements. It can provide information of where a person's attention is being directed to on a user interface and can be attributed to actual cognitive processing [107]. Many tools nowadays rely on corneal reflection; the eye is illuminated by an infrared light source and the reflection on the cornea and the pupil caused by this illumination is then detected by a high-resolution camera. Image processing algorithms compute the gaze point as the relative location of the reflection of the cornea to the reflection on the pupil [4, 12]. Besides computers equipped with eye trackers or movable constructs, there are also more lightweight and fully portable solutions like eye tracking glasses. They allow a complete free head movement, which is not always the case with stationary eye trackers. However, the data analysis is especially tedious since the eye tracker is mounted on the head of the participant and his frame of reference is changing constantly. Further, automatically video segmentation is not possible [12]. Even though eye tracking systems are becoming increasingly better, there are still participants (between ten % to 20%) where eye-tracking is not possible due to eye wear which interrupts the path of the reflection such as contact lenses or prescription glasses. Further, participants with large pupils or lazy eyes can be difficult to track [107].

There are two portions relevant when analyzing eye tracking data [12]:

- *Fixations* are pauses in eye movement, thus, a fixation denotes a time frame when the eyes are relatively stationary.
- *Saccades* are performed continually and are the eye movements between fixations.

Interpretation of eye tracking data can generally be performed in two ways, either top-down or bottom-up [52]. The top-down approach is based either on a cognitive theory suggesting that, for example, longer fixations on a control element are due to an unclear control structure or rely on a design hypothesis, e.g. infer that advertise placement in the center of the page leads to longer observations by the users. The bottom-up method is solely based on data observations without a defined theory. For example, the data shows that users take longer in

completing a certain step in a task, the question arises where they are looking at during this step.

There are different metrics usually analyzed in eye tracking studies for example [52]:

- *Fixations* are measured as their total number or as their average duration.
- *Areas of Interest (AOI)* is a part of the interface under evaluation containing relevant UI elements. Ideally, the white spaces between AOI are as small as possible leading to a limited number fixations between these regions.
- *Gaze Duration* is a series of successive fixations within an AOI. A fixation outside the AOI ends the gaze.
- *Scan Path* describes a complete sequence of a saccade to a fixation to a saccade again.
- *Dwell Time* comprises the overall time spent looking at an AOI, i.e. all fixations and saccades over all visits.
- *AOI Sequence* represents the order in which the AOIs were first fixated.

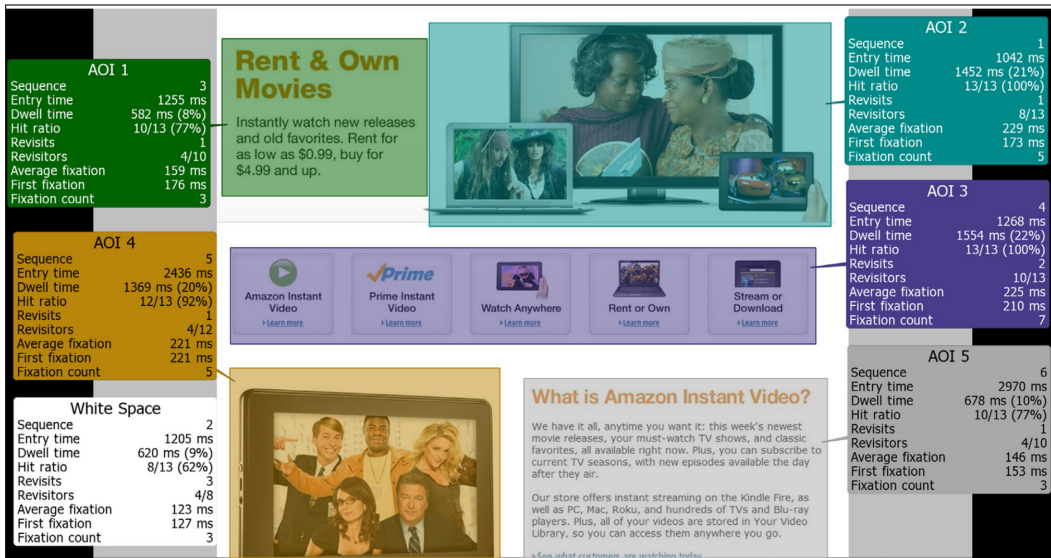
There are different interpretations of eye tracking data. For example the total number of fixations could be used to determine the search efficiency within a user interface. An inefficient search, for example due to a poor UI element placement, can be reflected by a greater total of fixation points. When evaluating the count of fixations, the task time has to be taken into account as longer tasks would clearly lead more fixation points. In contrast, more fixations on a particular part of the interface can reflect importance or notability of that area. Faster times to the first fixation of an element or AOI can indicate that it draws the users attention. Hence, if an important area is not fixated enough it might has to be highlighted. A larger average fixation duration can indicate problems of participants extracting information from an interface or missing meaningfulness of icons. The proportion of time looking at each AOI can reflect the importance of an element, however, at the same time this can indicate difficulties in understanding the elements [52, 107]. Evidently, interpreting eye tracking data cannot be done separately, but the data has to be analyzed in combination with video material, notes etc. to be able to grasp the context and in turn understand the meaning of data.

Visualizations are especially helpful in conveying eye tracking results as they usually highlight the areas of the interface which were examined the most by the participants. Either the visual depicts the results of single individual or aggregates the data of several participants. In case of the former a scan path of the interface might give insights into how the user searched for a particular element. Figure 3.7a illustrates an example of such a scan path. Displaying the scan paths of numerous users in a single visualization is also possible, however, the most common eye movement visualization for multiple participants is through a heat map as depicted in Figure 3.7b. In a heat map usually the brightest areas show the greatest density of fixations and hence where the most attention was drawn to. Another visualization within this context would be a focus map, where the more fixations an area has the more transparent the area is. Areas with less attention are darkened and hence less or not at all visible [4].

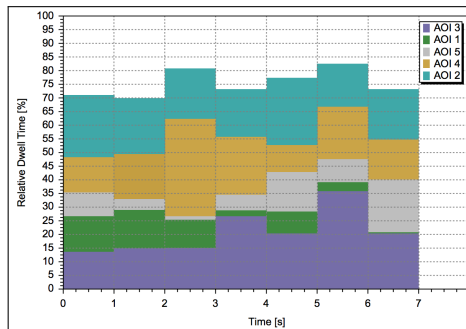
Usually, the eye tracking data visualization includes a summary of the statistics of each AOI, as depicted in Figure 3.7c. This resume comprises for example the number of fixations,



(a) Example of a test user's scan path on the Amazon Video website [4]. (b) Example of a heat map for all participants. Areas with more attention are shown in red, orange and yellow [4].



(c) Example of a summary statistics for each AOI [4].



(d) Example of a binning chart using 1-second intervals [4].

Figure 3.7: Visualizations of eye tracking data.

average fixation time, the sequence of the first visitation of the AOI, or the hit ratio. The hit ratio is the percentage of users who had at least one fixation in the AOI. Binning charts illustrate the percentage of dwell time for each AOI in sequence and as can be seen in Figure 3.7d the bins are color coded corresponding to the AOI.

#### 3.4.6.4 Single Usability Metric

Summative usability evaluations provide the researcher with various metrics to assess the usability of a system. While all of these metrics provide information on certain usability aspects, i.e. efficiency, effectiveness, or satisfaction, an overall assessment of the system is needed. Sauro and Kindlund [119] propose a single usability metric (SUM), which attempts to represent the usability of a system within a single metric. Their quantitative model combines four integral summative usability measures at a task level: task completion rates, error counts, task times and, satisfaction results into a single score. Albert and Tullis [4] propose another possible combined metric with only three measures, namely task times, task completion rates and the subjective ratings. While a single measure cannot replace all information, it provides a general overview of the usability, which is especially interesting when comparing different designs or products.

### 3.5 Method Triangulation

The most powerful method for determining problems is triangulation [140]. In the context of HCI triangulation is the approach to utilize multiple data collection and analysis methods to demonstrate a convergence on issues. On the one hand we can combine different research methods such as heuristic evaluation, questionnaires, or formal usability tests in order to obtain various usability indicators. On the other hand qualitative-quantitative triangulation consolidates quantitative measures such as time on task and qualitative data such as user comments. Despite the fact that usability metrics correlate, they cannot replace each other, e.g. even though the user is not able to complete the tasks, the subjective rating might still be positive.

Recently, Schmettow, Bach and Scapin [123] conducted a study in which the authors tested different evaluation and testing techniques. In particular, they examined expert inspection, usability testing, and document inspection. The authors determined that a single method could not find a substantial subset of usability issues alone, since no inspection method predicts end-user problems. Vice versa, due to the limited set of test participants a formal study might not be able to identify all usability concerns.

Barnum [9] suggests to conduct a heuristic evaluation and then use the results in a usability test. One can eliminate the issues found in the evaluation before testing the product to allow the participants to find additional usability problems. However, often due to time constraints the issues cannot be fixed before the tests. Then the result of the evaluation can be used to identify the goals of the usability tests, since there is already some notion of ways the users might experience problems. If the issues determined in the evaluation and the test correspond

to each other, then the combined results provide a stronger confirmation of the problems. In case they do not match then the evaluation still gives some valuable insight into usability problems which do not affect the user.

## 3.6 Children and Teenagers

When considering children and teenagers as users, special modifications are necessary in the design process, since guidelines mainly focus on adult users. Children and adolescents have other cognitive as well as physical capabilities, which should be reflected in the design of products specifically tailored for this user group [21]. In this section we focus on the one hand on principles for designing child-friendly software and on the other hand on special issues and considerations to be taken into account when evaluating the usability of products with young test participants.

### 3.6.1 Designing for Children

Within the process of designing and developing new technologies, children and teenagers can be assigned different roles, such as users, testers, informers, or design partners [29]. In many cases technologies designed for adults are not suitable for younger users, since children have a different set of needs and capabilities. For example, Hourcade [48] proposed a mouse which adapts its speed depending on the individual and the difficulty of the pointing task.

Observation is a powerful technique to gain insights into how users actually interact with products. Thus, analyzing how children utilize existing technologies, allows developers to study the effects of technology on the learning process and discover new technology concepts. Once a prototype has been implemented, evaluating the products with the help of children, provides information on the usability. Often a simple conversation can yield important statements on the user experience and usability from a child's perspective, which can be during any stage of the design process [29].

Idler [49] defines five key success factors for good UX of digital media for children:

- *Entertainment*: Digital media should be fun in order to have children motivated to use them.
- *Visual appeal*: Attractive designs can activate children's interest in a system and motive to use it.
- *Usability*: Children are impatient and since their motivation is to have fun, their tolerance for systems not working as they expect is low. Further, the system should match the cognitive and physical abilities of the target age group, e.g. the language should be understandable for children, font sizes should be large enough, color contrast should be high, and the input entry method should be appropriate.

Depending on the application the three main usability attributes *Efficiency*, *Effectiveness* and *Satisfaction* have to be present in various quantities. For example, in entertainment

software the child's satisfaction is essential as it is the motivation to use the system. In contrast in educational applications it is more important to have efficient and effective software to achieve goals [2].

- *Age appropriate content*: The content should reflect the mental models and interest of the target age group.
- *Encourage learning*: An important aspect is to allow children to learn in a fun and engaging way through play.

Liebal and Exner [2] provide various recommendations for software for children; we have taken some essential guidelines and listed them in Table 3.5.

### Screen Design

Color	<ul style="list-style-type: none"> <li>• Use various colors to create contrast</li> </ul>
Font and Text	<ul style="list-style-type: none"> <li>• Use a proper font and font size, e.g. simple and relatively large</li> <li>• High contrast between text and background</li> <li>• Avoid text on background images</li> <li>• Do not animate text</li> </ul>
Layout	<ul style="list-style-type: none"> <li>• Use standard layouts</li> <li>• Central information should be displayed visibly</li> <li>• Main content area should have an appropriate size</li> </ul>

### Visual Elements

Images and Graphics	<ul style="list-style-type: none"> <li>• Only provide images useful in assisting comprehension</li> <li>• Use meaningful graphics</li> </ul>
Animation	<ul style="list-style-type: none"> <li>• Create meaningful, short and interesting animations</li> </ul>

### Interaction

Mouse	<ul style="list-style-type: none"> <li>• All mouse buttons have their usual functionality</li> <li>• Avoid double click</li> </ul>
Touchscreen	<ul style="list-style-type: none"> <li>• Touchable fields should be large enough</li> </ul>
Interaction Technique	<ul style="list-style-type: none"> <li>• Consistent interaction technique through application</li> <li>• Point-and-Click are simpler and more effective</li> <li>• Use drag-and-drop only on short mouse paths</li> <li>• Avoid scrolling</li> </ul>

### Navigation and Menus

Navigation	<ul style="list-style-type: none"> <li>• Use standardized navigation techniques and stay consistent</li> <li>• Child should always know where it is</li> <li>• Simple way to move back</li> <li>• Choose meaningful names for menu items</li> </ul>
------------	---



Interface Metaphors	<ul style="list-style-type: none"> <li>• Use simple and consistent metaphors</li> <li>• Choose metaphors appropriate for mental model of the child</li> <li>• Choose metaphors from children's very day life</li> </ul>
Icons	<ul style="list-style-type: none"> <li>• Use standardized icons</li> <li>• Keep icons colorful</li> </ul>
Buttons	<ul style="list-style-type: none"> <li>• Visually highlight buttons</li> </ul>
<b>User support</b>	
Tutorials	<ul style="list-style-type: none"> <li>• Provide tutorials and training</li> </ul>
Feedback	<ul style="list-style-type: none"> <li>• Provide instant feedback</li> </ul>
Help	<ul style="list-style-type: none"> <li>• Design interfaces simple enough to avoid help texts</li> </ul>
<b>Content</b>	
Age appropriate	<ul style="list-style-type: none"> <li>• Provide age appropriate content to motivate use</li> </ul>
Text Passages	<ul style="list-style-type: none"> <li>• Short and easy to understand text</li> <li>• Avoid technical language</li> </ul>
Headings	<ul style="list-style-type: none"> <li>• Short and easy to understand</li> <li>• Larger than text</li> </ul>

Table 3.5: Design recommendations for childrens' software [2].

Although a study found that point-and-click interfaces are more effective, accurate and motivating than drag-and-drop interfaces, children's software still often features a drag-and-drop interaction style [50]. Barendregt and Bekker [8] conducted a study, where they encountered that children between six to twelve years expect to find a drag-and-drop interface rather than point-and-click. Children prefer drag-and-drop in cases where they already have experience with this type of input method. In those cases they appreciate the tactile feedback of letting the mouse button go and dropping the object [53].

Metaphors are conceptual models similar in certain facets to a physical objects and are widely used to explain abstract notions in familiar terms. These analogies have been used in interaction design to make complex computer concepts accessible, such as the desktop metaphor [76]. The best conceptual models are obvious and intuitive. Since a metaphor is only loosely coupled to a specific object, the abstract entity can have its own behaviors and properties not ascribed to the original artifact. Thus, an analogy may lead to unpredicted user interpretations outside their intended meaning [14]. Further, when working on products for children, metaphors from the adult world are not necessarily suitable for children [21].

UCD states that when designing software environments three main issues are to be addressed, namely *Tasks*, i.e. which goals have to be achieved with this software, *Tools*, i.e. which tools are provided to fulfill the tasks, and *Interfaces*, i.e. how do the tools' interfaces look like.

UCD proposes that the user is in the center of those three parts. Learner-centered design (LCD) is a development of UCD shifting the focus to the learner's needs [131]. The learner's needs are:

- *Understanding is the goal*: Helping the learner to understand by coaching them.
- *Motivation is the basis*: Learners are not necessarily motivated to learn. Low-overhead and immediate success should help to create and sustain motivation.
- *Diversity is the norm*: Learners are diverse individuals, with distinct needs, thus including various learning techniques is necessary to support different learners.
- *Growth is the challenge*: Software should be adaptable to learners once they have reached a higher proficiency level.

To accommodate these Soloway, Guzdial, and Hay [131] propose the Tools-Interfaces-Learner's Needs-Tasks (TILT) model to guide the design of educational software, where *Tasks*, *Tools* and *Interfaces* are connected to the Learner's Needs via scaffolding. Scaffolding in the educational context, is to provide assistance and support learners while they are becoming familiar with a new assignment. In order for the learner to acquire skills in the task domain, scaffolding techniques include for example coaching. Growth of the learner can be achieved by having adapting tools and in order to have learners express themselves different media and modes of expression should be integrable through the interface.

### 3.6.2 Usability Evaluations

Further, when assessing the usability of a product, the evaluation methods involving young participants have to be tailored to their developmental level [43]. Markopoulos and Bekker [75] record several characteristics of children affecting the process of usability testing as well as its outcome:

- *Verbalize*: Children are still developing their capacity to verbalize thoughts. Thus, Thinking-Aloud protocols can be problematic, since children might not be able to verbally express their thoughts depending on their age.
- *Extroversion*: Some children might have difficulties to speak up to adults and consequently report less issues.
- *Capability to concentrate*: The ability to concentrate on one activity is still developing in children. Depending on the age, the session length should be adapted to accommodate these issues.
- *Motivation*: Facilitator intervention and the children's motivation to please adults could impact the test results.
- *Trustworthiness of self-report*: Children might be influenced by the need to please adults and their reports are questionable.

- *Ability for abstract and logical thinking*: Complex reasoning like cause and effect relations are not entirely evolved and they might not understand abstract task description. Furthermore, the number of items they can keep in mind at once is limited.
- *Progress towards a goal*: The ability to monitor goal-directed performance develops throughout childhood and adolescence.
- *Gender differences*: Some ages may have more noticeable gender differences than others.
- *Motor skills* : Especially younger children do not necessarily have the needed motor skills to use standard input devices like the mouse.

### 3.6.2.1 Age Ranges

Depending on the age, we can identify different cognitive abilities as well as behaviors in children [43]. Note that the categorization is somewhat arbitrary as child development is a continuous process with behaviors overlapping and there exist classifications slightly differentiating from the one we present [21]. Generally, children below the age of two years are not suitable for conducting usability studies as their cognitive as well as physical development is not advanced enough. Children over the age of fourteen years, should be treated as adults as they will likely behave similar during the test.

#### Preschool-Aged Children (2 to 5 years)

Usability tests with children under the age of six require several modifications, since they have a shorter attention span than adults (around 30 minutes). Children cannot image abstraction or another person's view and often concentrate on a single aspect of a given task, while ignoring others. Furthermore, preschoolers are more likely to focus on the current state of the task and do not keep in mind previous or future events [47].

While the test situation can be stressful for children of all ages, especially, young children might not adapt as easily to the testing environment and the unknown individuals conducting the study. Therefore, it is advised to have the parents present during the session. Preschoolers are often not capable of clearly expressing themselves and their preferences or dislikes, hence, observing their non-verbal behavior is especially crucial in order to get insights. Generally, allowing children to freely explore the system, is more suitable than a set of predefined tasks. Furthermore, we can expect children in this age range to be preliterate and likely to be trying to please the adult, thus expressions of appeal should be analyzed critically.

#### Elementary-School-Aged Children (6 to 10 years)

Due to their experience in school, children within this age group can follow instructions and concentrate for a certain period of time. Further, they are used to being around adults other than family and are more proficient in expressing their opinion and answering questions. Elementary school children can anticipate the future and take into account past events when solving problems [47]. Depending on the age and character of the child, a Thinking-Aloud test might generate insightful usability data. Donker and Markopoulos [28] for example conducted a study with children ranging from eight to fourteen years and concluded that Thinking-Aloud

produced the most usability issues in comparison to interviews and questionnaires. Even younger children, aged six to seven, were capable of thinking aloud in a usability study of an interactive toy [136]. The same study by Donker and Markopoulos [28] pointed out that due to the limited number of items in the memory of small children, in post test interviews children might have difficulties recalling actions and decisions taken. Co-discovery produced the least number of comments from the children within this study, as the children were trying to individually solve the problems. However, conducting a study within a usability lab, only connected to the test facilitator via speaker, might be too stressful for children. Depending on their exposure to technology, they will have experience with computers and other devices.

### Middle-School-Aged Children (11 to 14 years)

Nowadays, it is very likely that the children in this age range are proficient with a variety of devices and input methods, such as smartphones and touch screens. These children can further perform specific tasks, after becoming familiar with the product. However, children have difficulties concentrating more than an hour, thus the test time should be planned accordingly.

While these age ranges provide us with some notion of the capabilities to expect from children within a certain range, research has indicated that a child's development will only produce a likelihood that the child will behave in a particular way according to their development stage, since there is a large individual variation [34].

#### 3.6.2.2 Guidelines

Hanna, Ridsen and Alexander [43] provide guidelines for usability testing with children, addressing a wide set of subjects, e.g. the room decoration, utilized equipment, and facilitator behavior suitable for children based on their experience. Subsequently, we briefly discuss some of the additionally mentioned aspects to remember when testing with children:

- When conducting the study in a usability lab, it is advantageous to explain the system to the children in order to gain their trust.
- Further, building a connection to the child via smalltalk is important, to make the child feel comfortable.
- Decorating the lab in a child-friendly manner, also helps to put children at ease.
- Children should be selected in regards to their level of experience with computers, since a base proficiency is necessary.
- Before the test session, the facilitator should explain the goal for the study and what the child is to expect from the test and the product under test. It is of importance to explain that the product is being tested and not the child itself and that their help is essential in order to improve the product.
- Preschoolers might require a training phase with the device being used for testing.

- Ideally, the task order is interchanged between the child participants, in order to remove some results being affected by the children getting tired towards the end of the test and it is advisable to offering a break after 45 minutes of testing.
- Children often are used to asking questions. However, in the test situation it is important to ask the children to solve the problem themselves and help them once they get frustrated.
- Providing positive feedback, when they are doing the tasks is important in order to keep them motivated.
- Depending on the age, a reliable post-test rating might not be expectable. However, older children can provide insightful information, based on a smiley rating scale, from a smiley face to a frown.
- Further, when testing minors a consent form signed by the parents is important and that a little token of appreciation is shared with them at the end of the test, such as a test certificate.

### 3.6.3 Teenagers

Teenagers are defined as individuals between the ages of thirteen and nineteen. As digital natives ubiquitous technology and information has changed their thinking patterns [108]. Therefore, it is beneficial to take advantage of these user group to gain insights in their usage of new technology. Teenagers are usually still living with their parents, but differentiate from children, as they are seeking increasing independence and often have a larger variety of technology at their disposal [32]. It has been shown that teenagers process information differently from children as well as adults [133]. They are early technology adopters combining child-like features with a more advanced ability of articulation. Media and technology play an important role in the of teenagers [32].

Bruckman, Bandlow, and Forte [21] state that adolescents over the age of fourteen can be treated equally to adults in usability studies. Markopoulos and Bekker [76] propose a similar view and state that in early adolescence teenagers become more goal-oriented and social. Further, they are capable of understanding abstraction and solve more complex problems. They are able to interpret situation from various points of view and can integrate these views, concepts and ideas into their problem solving. However, teenagers are still emotionally and cognitively different from adults and still posses childlike tendencies.

Teenagers are not often studied in the context of HCI, mainly due to their categorization as adult like participants in usability studies [33]. Thus, there is a gap between the Child-Computer Interaction research community and mainstream HCI. Fitton et al. [33] define several traits of adolescents differentiating them from children and adults such as the desire for independence and autonomy from authority figures, the development of an individual identity, and the search for peer connection and acceptance. Teenagers are likely to have developed different personas depending on their surroundings, thus, the testing environment might change their behavior. For instance, conducting studies within the school might be

convenient, however, the facilitator might be seen as an authority figure similar to a teacher [133].

Teenager development can be categorized across three components [32]:

- *Fundamental changes*: Changes are taking place at the cognitive, biological, and social level. Memory capabilities as well as processing speed increases. Further, the ability to abstract reasoning improves. The reproductive functions are transforming a child gradually into an adult, with changes in physical appearances and increasing sexual interest. Teenagers are given rights and responsibilities as they slowly change to adults.
- *Context*: The context corresponds to the environment where the changes happen during adolescents. Four distinct surroundings can be distinguished, namely family, peer groups, school, and leisure environments, where peer-related influences have the most impact on development [128].
- *Psychosocial development*: Autonomy, achievement, identity, intimacy, and sexuality are the five key psychosocial challenges during adolescents.

Wodike, Sim, and Horton [142] conducted a study to investigate the possibilities to have adolescents, aged twelve and thirteen years, facilitate a heuristic evaluation with their peers acting as the expert evaluators of a game. As heuristics the set proposed by Nielsen and Molich [96] was used without adaptation. Their results indicated the evaluators struggled to find problems and got distracted from the evaluation process. In contrast, Pasiali's [104] study has shown that teenagers between thirteen to fourteen years where engaged in the heuristic evaluation technique and overall promising results were obtained. To avoid possible misunderstanding, Pasiali [104] simplified Nielsen and Molich [96] heuristics.

The Nielsen Norman Group [70] conducted various studies in order to determine web site usage by teenagers and found out that while their success rate has been improving it is still well below the adult completion rate due to three key factors, namely insufficient reading skills, less sophisticated research strategies, and dramatically lower levels of patience.

### 3.7 Mobile Devices

Smart mobile devices are becoming increasingly widespread and their ubiquitous computation power constitutes them a suitable platform for interactive engagement. Simultaneously to the mobile pervasiveness, various interaction modes for touch screens are becoming increasingly common [6]. Since mobile interfaces in general are limited to small screens as well as a restricted control space, applications have to be designed in regard to those characteristics [44].

Due to the advent of smart mobile devices new usability challenges have been introduced and the traditional usability models are not entirely applicable within this context. Mobile devices have a unique set of features which have to be taken into account when designing applications. In particular, the user's environment is potentially constantly changing and

requires the user's attention while performing tasks on the mobile device. Thus, there is an additional cognitive load involved [44].

Zhang and Adipat [145] list a number of problems stemming from mobile devices and their distinctive attributes:

- *Mobile Context*: Due to the ubiquitous nature of mobile devices, their usage is not limited to a certain location. Further, the user might be interacting with other individuals or objects or trying to accomplish other tasks while on the device.
- *Data Entry Methods*: Input methods on mobile devices differentiate from the traditional input techniques on traditional computers and require a greater proficiency, are more error prone, and decrease the data entry rate.
- *Connectivity*: Depending on the user's location the network connectivity might be restricted which can lead to performance issues for applications relying on said features.
- *Small Screen Size*: Even though the screen sizes are continuing to grow in the recent years, the screen on mobile devices is still far smaller than on desktop computers, restricting the amount of information displayable at once.
- *Different Display Resolutions*: The display resolution on mobile devices is inferior to desktop computers.
- *Limited Processing Capability and Power*: In order to have portable devices the processing power and memory is limited on mobile devices.

Some of the issues, which were apparent in 2005 when Zhang and Adipat [145] wrote the paper, are now less evident or have decreased in impact. Mobile screens are getting larger, especially within the last years, and they feature higher resolutions. The processing power is improving and network connectivity is increasing. However, depending on the device and the user's location all points are still relevant to a certain extend.

Within the next two sections we focus on *Mobile Context* and present a model for usability of mobile devices and subsequently, discuss some points in regard to *Data Entry Methods*.

### 3.7.1 People At the Centre of Mobile Application Development

Recently, Harrison, Flood, and Duce [44] proposed the People At the Centre of Mobile Application Development (PACMAD) usability model, which integrates the usability attributes developed by Nielsen [93] and the ISO Standard [1] in the context of mobile devices. Figure 3.8 depicts the three critical factors and seven attributes which can affect the overall usability of a mobile application.

The factors and six of the seven attributes have already been discussed in Section 3.1 as part of the ISO 9241:11 standard [1] and Nielsen's usability attributes [93]. The main contribution of the PACMAD is the acknowledgment of the *Cognitive Load* in addition to the already known attributes. In contrast to traditional desktop programs, the user can perform other tasks while utilizing the device, such as walking. Hence, more cognitive processing is necessary



Figure 3.8: PACMAD [44].

to multi task. There have been studies investigating the impact of using mobile devices while walking, which indicate an accuracy drop to 36% when carrying a bag in the dominant hand. Accuracy reduced to 34% when holding a box under the dominant arm [90]. When handling mobile devices, another study found for everyday situations the interaction with mobile devices was broken into periods of four to eight seconds due to constant interruptions of the user's attention [98].

### 3.7.2 Interaction Mode

Smart mobile devices rely on touch for interaction. The easily touchable and reachable areas of the smartphone screen, however, vary across users, devices, and holding gestures. There have been studies examining the usability of touch and surface gesture interaction on various platforms such as PDAs or mobile devices. Pointing accuracy of a finger is smaller in comparison to other pointing devices, such as a computer mouse [12]. Therefore, buttons of smartphone applications have to be larger in size and the space between buttons has to increase, to reduce the chance of incorrect touch events. In general, a size of seven to ten millimeters is recommended for touchscreen targets in order for the user's to accurately tap them. For example, the Google's Metrics and keylines Guidelines<sup>2</sup> recommend that all touchable user interface components follow the 48 density-independent pixel rhythm. On average, 48 density-independent pixels translates to a physical size of about 9 millimeters.

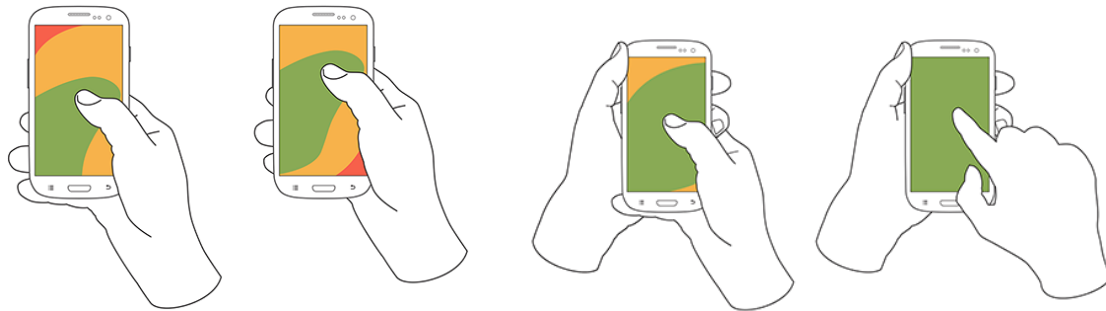
Virtual keyboards are very different across devices. Mode switches are of special interest, as they can effectively have an unlimited number of modes and thus an unlimited number of input keys on a key board. When you switch modes labels, for each key, as well as the position and shape of keys can change.

Recently, Hooper [46] conducted an informal observation on how users naturally hold their touchscreen mobile phones in case they are not engaged in a passive activity or on a call. Figure 3.9 depicts the tree three basic ways of holding the phone: one handed (Figure 3.9a),

<sup>2</sup> <https://www.google.com/design/spec/layout/metrics-keylines.html>

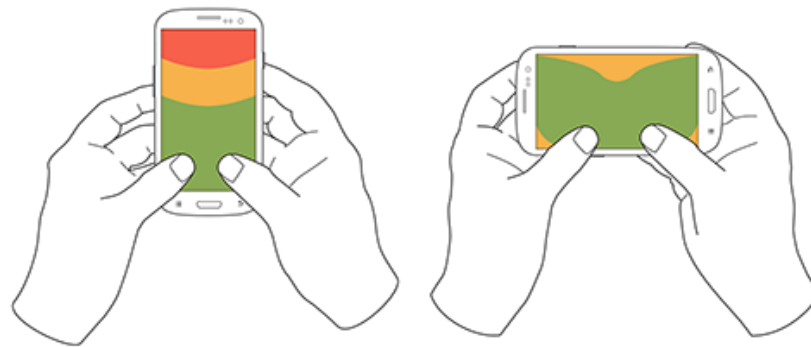


cradled (Figure 3.9b), and two handed (Figure 3.9c). Further, the graphic shows which areas of the screen can be easily accessed in green, and which are more difficult to reach (orange) or not accessible at all (red). The observations additionally indicated that this is not static, i.e. users change the way they hold their phone often.



(a) One-Handed Use (around 49 %): right thumb on the screen (67%) or left thumb on the screen (33%) .

(b) Cradling in Two Hands (around 36 %): thumb on the screen (72%) or finger on the screen (28%).



(c) Two-Handed Use (around 15 %): vertically, i.e. in portrait mode (90%) or horizontally, i.e. in landscape mode (10%).

Figure 3.9: Summary of how people hold and interact with mobile phones [46].

### 3.7.3 Operating System Guidelines

In order to ensure continuity among mobile applications for various operating systems, the providers have developed design guidelines. Google, for instance, has published a set of interface recommendations for Android devices designated to the components and interaction modes available. For example, the recommendations describe the ideal icon and button size as well as location, menu responsiveness, text formats, and notification features<sup>3</sup>. Some of these guidelines are very similar to the heuristics we have discussed in Section 3.3.2, e.g. *“If it looks the same, it should act the same”* meaning that functional differences should be visually distinguishable. Apple’s iOS Human Interface Guidelines<sup>4</sup> describe the iOS app characteristics that are necessary in order to publish an application in the AppStore.

<sup>3</sup> <http://developer.android.com/design/index.html>

<sup>4</sup> <https://developer.apple.com>

### 3.7.4 Children and Mobile Devices

Anthony et al. [6] examined interaction and recognition issues of touch and surface gestures for adult and child users. Generally, there has been research identifying difficulties of drag-and-drop interactions for children and consistent evidence that the performance of pointing tasks increases with the childrens' age. Anthony et al. [6] identified several recommendations when developing touch interfaces for children:

- *Account for holdover touches*<sup>5</sup>
- *Use consistent, platform-recommended target sizes*
- *Increase active area for interface widgets*
- *Align targets to edge of screen*

Children experience different challenges depending on the interaction mode. Intuitive gestures for preschool children include tap, draw/move finger, swipe, drag, and slide [124]. However, children face difficulties with executing a continuous stroke, making draw and drag demanding, thus partial completion might be useful. When using swipe gestures it is essential not to have other functionality spots in the swiping area and for sliders it is recommended to have a strong visual indication. Pinch, tilt/shake, multi-touch or double tap are less intuitive interaction styles for children.

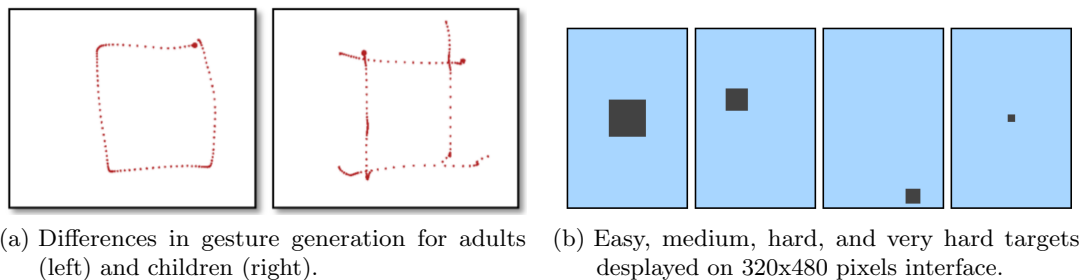


Figure 3.10: Comparison of touch screen interaction between adults and children [20].

Brown and Anthony [20] investigated differences between children (seven to eleven years of age) and adults in gestures and target hit rates on touch screens. Figure 3.10a depicts a square gesture by an adult and a child in comparison. While the adult's gesture is a single move across the interface, the child utilized several strokes in order to generate a square. Touch tasks involved a 320x480 pixels interface, where the participants were asked to touch targets of various sizes and positions. The difficulty is determined by the target size ranging from 100x100 to 20x20 pixels and its position on the UI as depicted in Figure 3.10b. The results showed that children missed the target 50 % more often than adults and by a greater distance. Thus, there is a need to increase the target size which in turn, however, reduces the amount of on screen information which can be displayed.

<sup>5</sup> Holdover touches are intentional touches in a location of a previous target. This phenomenon can occur, when the user does not notice that the target has been activated already. This is especially the case for children, who cannot cognitively process information as quickly.

### 3.7.5 Mobile Usability Testing

Usability testing of mobile devices poses challenges and differences in comparison to evaluations of traditional desktop computer applications. Mainly, since these devices are handheld using an external video camera to record the screen might be infeasible. Keeping the screen in focus, dealing with various lighting situations due to changing angles to the light source and even keeping the device within the range of the camera is demanding. Therefore, specialized labs have additional equipment to test hand held devices. A possible set-up has a camera mounted on the ceiling above the participant and the device static on the table or a structure cradling the mobile device. A document camera as shown in Figure 3.11a can also be utilized for testing mobile devices [9].

According to Krannich [63] there are three common set-ups for mobile usability studies:

- Mounting the device to a stand (Figure 3.11b).
- Mounting a mini camera to the device (Figure 3.11c).
- Mounting a camera to the participant<sup>6</sup> (Figure 3.11d).

For many mobile operating systems screen recording applications are available, which can be utilized to capture the screen in an unobtrusive way. However, as stated by Lang [64] screen recorders typically do not capture gestures, have a limited recording time, do not integrate with common usability test recording software and do not record facial expressions or comments. Another factor arguing against screen recorders is that users cannot use their own device. Note that some testing methods are more difficult to conduct on mobile devices. For example in co-discovery both participants have to interact with the application, thus mounting a camera on the device possibly disables at least one of the users to look at the screen.

Especially the context of mobile devices has been investigated within usability testing. There have been studies which evaluate the application in the field, e.g. the user performs the task while walking to a certain destination, in the laboratory simulating a realistic environment or on a simulator on the desktop computer. A study by Kjeldskov and Graham [60] revealed that more than 70% of all mobile usability studies are conducted in laboratories rather than the field, since the data collection is rather laborious in an open setting.

Kallio et al. [55] compared usability testing of mobile applications in the lab to field studies and found that studies in a natural environment are more than twice as time consuming as lab-based studies and the number of unplanned events and interruptions increases significantly. To have a more realistic experience some researchers are using new techniques within laboratory testings such as Beck et al. [10] who conducted an experiment where participants were asked to perform tasks on a mobile phone while sitting or while walking on a treadmill with various speed settings. The authors compared the results to a field test on a pedestrian street as a

---

<sup>6</sup> Note that this a more invasive method and therefore not applicable in every situation.

<sup>6</sup> <http://www.tobiipro.com/>

<sup>7</sup> <http://www.mrtappy.com/>

<sup>8</sup> <https://gopro.com/>



(a) Document Camera for mobile devices usability tests [9].



(b) Device mounted to a stand<sup>7</sup>.



(c) Camera mounted to the device<sup>8</sup>.



(d) GoPro mounted to the head<sup>9</sup>.

Figure 3.11: Usability testing set-ups for mobile devices.

reference. The results show that in the set-up where the participants were sitting the most usability issues were uncovered.

## 4 Usability Evaluation

---

*“If you create a long formula and then realize you should put the square root over the entire thing, then there is not way back; you have to do it all over again. Game over, man! Game over!”*<sup>1</sup>

Formula manipulation is an essential part in Pocket Code, as the formula editor itself is a powerful tool necessary to create meaningful programs. Based on a pocket calculator metaphor, the editor was designed to have users be able to work instantly by relying on familiar concepts known from mathematics in school. The objective of this thesis is to determine whether the formula editor in Pocket Code in fact is usable, i.e. effective, efficient and yields subjective user satisfaction. In order to achieve a more wholesome picture, we applied two techniques; first, we performed a heuristic evaluation on the formula editor, where a team of reviewers examined the interface in accordance to a set of guidelines. Second, we conducted a formal experiment, comparing Pocket Code to the state of the art novice programming environment for children, namely Scratch. In the upcoming section we describe the interface inspection, i.e. the evaluators, the guidelines we utilized as well as the issues we have uncovered. In Section 4.2, we discuss our summative usability study design in more detail including the results of the data analysis as well. Subsequently, Section 4.3 comprises a short summary on the issues of both usability studies and presents a set of recommendations.

### 4.1 Heuristic Evaluation of Pocket Code

In order to maximize the number of found usability issues of the formula editor, we conducted a heuristic evaluation prior to a formal experiment. We utilized object-based heuristics specifically developed for mobile applications.

#### 4.1.1 Evaluators

Recall from Section 3.3 that five evaluators will on average identify 75 % of interface problems. Thus, we involved five usability reviewers who examined the formula editor at least twice and recorded all possible usability issues they could identify. Generally, all evaluators were members of the UX team of the Catrobat project, thus were usability experts with varying

---

<sup>1</sup> Comment by one of the evaluators of the heuristic evaluation.

levels of experience and all had been part of the project for some time and thus were familiar with Pocket Code and mobile app usability. Further, most of them have been part of previous usability studies conducted with teenagers on Pocket Code, thus already had some notion of possible problems within the formula editor design.

#### 4.1.2 Heuristics

Since mobile interfaces differ from traditional UIs of desktop applications and websites, we decided to use a variation of the principles proposed by Nielsen [91]. Machado Neto and Pimentel [73] derived new heuristics especially designed for the usability evaluation of mobile user interfaces based on Nielsen’s traditional ten guidelines. Table 4.1 contains the eleven heuristics they have developed, which in fact are similar to the originals but feature a description specific to the mobile context. To evaluate them they conducted an inspection of an Android application which was examined by two evaluator groups, one used Nielsen’s heuristics and the other one the newly developed guidelines. The results show that the experts identified more usability issues with the specific mobile centered heuristics. We decided to use these heuristics instead of a child-centered set of heuristics, as the main target group of Pocket Code are teenagers. In addition most interface guidelines for children focus on a rather young age group, thus are not suited for teenagers.

	<b>Heuristic</b>	<b>Description</b>
1.	“ <i>Use of screen space</i> ”	“ <i>The interface should be designed so that the items are neither too distant, nor too stuck. Margin spaces may not be large in small screens to improve information visibility. The more related the components are, the closer they must appear on the screen. Interfaces must not be overwhelmed with a large number of items.</i> ”
2.	“ <i>Consistency and standards</i> ”	“ <i>The application must maintain the components in the same place and look throughout the interaction, to facilitate learning and to stimulate the user’s short-term memory. Similar functionalities must be performed by similar interactions. The metaphor of each component or feature must be unique throughout the application, to avoid misunderstanding.</i> ”
3.	“ <i>Visibility and easy access to all information</i> ”	“ <i>All information must be visible and legible, both in portrait and in landscape. This also applies to media, which must be fully exhibited, unless the user opts to hide them. The elements on the screen must be adequately aligned and contrasted.</i> ”
4.	“ <i>Adequacy of the component to its functionality</i> ”	“ <i>The user should know exactly which information to input in a component, without any ambiguities or doubts. Metaphors of features must be understood without difficulty.</i> ”

	<b>Heuristic</b>	<b>Description</b>
5.	<i>“ Adequacy of the message to the functionality and to the user”</i>	<i>“The application must speak the user’s language in a natural and non-invasive manner, so that the user does not feel under pressure. Instructions for performing the functionalities must be clear and objective.”</i>
6.	<i>“ Error prevention and rapid recovery to the last stable state”</i>	<i>“The system must be able to anticipate a situation that leads to an error by the user based on some activity already performed by the user [8]. When an error occurs, the application should quickly warn the user and return to the last stable state of the application. In cases in which a return to the last stable state is difficult, the system must transfer the control to the user, so that he decides what to do or where to go.”</i>
7.	<i>“ Ease of input”</i>	<i>“The way the user provides the data can be based on assistive technologies, but the application should always display the input data with readability, so that the user has full control of the situation. The user should be able to provide the required data in a practical way.”</i>
8.	<i>“ Ease of access to all functionalities”</i>	<i>“The main features of the application must be easily found by the user, preferably in a single interaction. Most-frequently-used functionalities may be performed by using shortcuts or alternative interactions. No functionality should be hard to find in the application interface. All input components should be easily assimilated.”</i>
9.	<i>“ Immediate and observable feedback”</i>	<i>“Feedback must be easily identified and understood, so that the user is aware of the system status. Local refreshments on the screen must be preferred over global ones, because those ones maintain the status of the interaction. The interface must give the user the choice to hide messages that appear repeatedly. Long tasks must provide the user a way to do other tasks concurrently to the task being processed. The feedback must have good tone and be positive and may not be redundant or obvious.”</i>
10.	<i>“ Help and documentation”</i>	<i>“The application must have a help option where common problems and ways to solve them are specified. The issues considered in this option should be easy to find.”</i>
11.	<i>“ Reduction of the user’s memory load”</i>	<i>“The user must not have to remember information from one screen to another to complete a task. The information of the interface must be clear and sufficient for the user to complete the current task. ”</i>

Table 4.1: Heuristics for the usability evaluation of mobile interfaces [73].



### 4.1.3 Results

The five evaluators were asked to perform a heuristic evaluation with the focus on the formula editor and the guidelines given in the last section. Each inspection was conducted by each evaluator alone and they were asked to review the editor at least two times and report their findings in an on-line spreadsheet in the format as can be seen in 4.2:

Heuristic	Name	Description
4,3	Error messages	If I enter a formula incorrectly, I am not told, how to correct it.

Table 4.2: Example of a heuristic evaluation report row.

After all evaluations have been completed, we analyzed the issues reported and summarized similar descriptions into one usability problem. Afterwards, this aggregated list incorporating all found issues was distributed to all the reviewers who then were asked to individually rate each issue, i.e. the evaluators could not see the rating of the other reviewers. We utilized a simple severity code system based on a 5-point scale as described by Nielsen [93]:

- 5 *Catastrophe* leads to a task failure.
- 4 *Major problem* has significant potential impact on the usability.
- 3 *Minor problem* has a low priority, but should be considered.
- 2 *Cosmetic problem* should be corrected if there is time.
- 1 *No problem*

Subsequently, we averaged the rating results for each usability problem and generated a prioritization of issues. In a meeting, we then discussed each issue, talked about possible solutions and re-prioritized the list. Table 4.3 provides all issues found by the evaluators. The heuristics are ordered according to their average issue rank indicated in the fifth column, "R.". The first column indicates the issue's priority and the second column, "H.", contains the heuristics violated .

No.	H.	Name	Description	R.	Proposed Solution
1	2	Square root	When computing the value of $\text{sqrt}(-1)$ using the <i>Compute</i> button the result is 1 (see Figure 4.1a).	4.4	Since, it can happen that when using a variable in a program, that it becomes $-1$ and then using the square root would lead to a <i>NaN</i> . Thus, in order to avoid a <i>NaN</i> , it is approximated to 1. During the program execution an approximation is useful, however, when utilizing <i>Compute</i> the correct mathematical solution should be shown. In particular, since we have this calculator metaphor and an actual calculator does show <i>NaN</i> or <i>Error</i> .
2	2	Sinus	When computing the value of $\text{sin}(180)$ using the <i>Compute</i> the result is not 0 but $1.22E-16$ , which is not mathematically correct.	4.4	During the program execution an approximation is useful, however, when utilizing <i>Compute</i> the correct mathematical solution should be shown. In particular, since we have this calculator metaphor and an actual calculator does show the correct value.
3	2	Division by zero	When computing the value of $0/0$ using the <i>Compute</i> the result is not <i>NaN</i> but 0, which is not mathematically correct.	4.4	During the program execution an approximation is useful, however, when utilizing <i>Compute</i> the correct mathematical solution should be shown. In particular, since we have this calculator metaphor and an actual calculator does show the correct value.

No.	H.	Name	Description	R.	Proposed Solution
4	2	Infinity divided by infinity	When computing the value of <i>infinity/infinity</i> using the <i>Compute</i> the result is $1(1/0)/(1/0)$ , which is not mathematically correct.	4.4	During the program execution an approximation is useful, however, when utilizing <i>Compute</i> the correct mathematical solution should be shown. In particular, since we have this calculator metaphor and an actual calculator does show the correct value.
5	5	Syntax error	When creating a syntactically incorrect formula, e.g. parentheses mismatch, several operators without operands, etc. the error message displayed reads <i>Syntax error</i> (see Figure 4.1b). There is no further granularity in the error message, thus does not aid the user in determining how to edit the formula to correct it.	4.2	The type of syntax error should be displayed in more detail and <i>Syntax error</i> should be renamed.
6	2	Rounding	When computing the value of <i>round(1,4999999)</i> using the <i>Compute</i> button the result is 2.	4.2	During the program execution an approximation is useful, however, when utilizing <i>Compute</i> the correct mathematical solution should be shown.

No.	H.	Name	Description	R.	Proposed Solution
7	2	Natural Logarithm	When computing the value of $\ln(-1)$ using the <i>Compute</i> button the result is 1.	4.2	Since, it can happen that when using a variable in a program, that it becomes $-1$ and then using the square root would lead to a <i>NaN</i> . Thus, in order to avoid a <i>NaN</i> , it is approximated to 1. During the program execution an approximation is useful, however, when utilizing <i>Compute</i> the correct mathematical solution should be shown. In particular, since we have this calculator metaphor and an actual calculator does show <i>NaN</i> or <i>Error</i> .
8	5	Representation of large numbers	When computing the value of a large number using the <i>Compute</i> button the result is shown in exponential notation (see Figure 4.1c), which is not necessarily known to teenagers.	3.6	Showing the entire number would be a better solution.
9	9	No feedback during large number computation	When computing the value of a very large number using the <i>Compute</i> button the result is not shown immediately, but there is also no feedback that the computation is taking place.	3.6	Add feedback, i.e. typical Android “Loading” circle for this case.

No.	H.	Name	Description	R.	Proposed Solution
10	6	Two sided inequalities	When computing the value of $1 < 0 \leq 6$ using the <i>Compute</i> button the result is <i>True</i> , since the formula is evaluated left to right and 0 represents <i>False</i> and everything else represents <i>True</i> (see Figure 4.1d).	3.4	No solution yet, but the discussion of coercions will probably be necessary.
11	7	Random number generator	The usage of the two parameters of the random function is difficult to understand for people who have never programmed.	3.4	Adding some type of additional help text would be useful for some mathematical functions.
12	4	Meaning of <i>position_x</i> , <i>position_y</i>	The object attributes <i>position_x</i> and <i>position_y</i> are the center point of the object, but that is not necessarily clear when starting to use Pocket Code, could also be the left lower edge.	3.2	Adding some type of additional help text would be useful for some attributes.
13	6	No way to add something in the beginning	In case we create a formula and then want to put the square root over the entire formula, there is no way of doing this. The user has to delete everything, select the square root function and type the formula again.	3.2	Implement the possibility to add a function around already existing parts of a formula or add some type of copy/paste capability for sub formulas.
14	7	Difficult to set the position of the cursor	It is difficult to set the position of the cursor.	3.2	Either reimplement the cursor or add arrow buttons to move the cursor left and right.

No.	H.	Name	Description	R.	Proposed Solution
15	7	Wait -1 seconds	It is possible to set the waiting time of the <i>Wait</i> brick to a negative value or a logical value.	3.2	There should be type checking done. Maybe in this case the <i>Logic</i> Category button is inactive.
16	4,5,10	Category/function names	The category names and mathematical names are not necessary intuitive, e.g. <i>Logic, sqrt, mod, abs</i> .	3.2	Some of them are as on a calculator thus would match the metaphor, however, we argue that since there is enough room a more representative name or an additional information on the function would be helpful. For example, using <i>square root</i> or $\sqrt{\quad}$ . For the category names there is no solution yet.
17	1,7	Button size	Buttons are rather small, thus, it is easy to miss the target, especially on the number pad.	3	Increase the button size.
18	2	No rename/copy functionality for variables	Throughout Pocket Code it is possible to rename and copy Objects or Scripts, but it is not possible for variables.	2.8	Implement Rename/-Copy functionality for variables
19	1	Input fields to small on big screens	The bricks text field does not use the entire width of the screen. In particular, for larger devices this is a waste of space.	2.6	Use a different layout for bigger screen resolutions.
20	4	Underscores in sensor values look “too technical”	<i>position_x</i> seems very close to variable names in textual programming languages.	2.6	In the list of object attributes <i>x position</i> would be teenager friendly.

No.	H.	Name	Description	R.	Proposed Solution
21	2	Random number generator in formula editor view	The random number generator should not be outside the mathematical function category as there is no reason for it to have a special position within the formula editor.	2.6	Remove it from the formula editor view.
22	2	Delete button	To delete a long formula one has to keep tapping the delete button, when keeping it pressed it still only removes a single character or function.	2.6	Add continuous delete.
23	2	Cursor design	The cursor in the formula editor does not look like the standard Android cursor, also it does not flash and its position is not centered.	2.4	The cursor should centered, Grey and flashing.
24	7	Cursor behavior on click in text field	When clicking in the text field after entering some text the cursor jumps to the right, it looks like there is a space character inserted, but there is not.	2.2	The cursor should not move when selecting the text field.
25	2	No $x^n$	There is no function for calculating the exponentiation.	2.2	Introduce function to calculate the exponentiation.
26	2	Compute dialog does not close on tap	Tapping on the compute dialog does not have any effect. The user has to tap next to the dialog which is not intuitive.	2.2	The compute dialog should be closed on tap.
27	2	Inconsistent error messages	Syntax errors can cause different behaviors. For some errors a toast message is shown, for others there is not.	2.2	Implement consistent error handling.

No.	H.	Name	Description	R.	Proposed Solution
28	1	Unnecessary '=' operator	The '=' operator is not necessary in the formula editor view, as it is a logical operator. Further, its current position is confusing as it could be interpreted to having a similar functionality as <i>Compute</i> (see Figure 4.2c).	2.2	Remove the '=' operator from the formula editor view.
29	2	Inconsistent text highlighting	When highlighting text in the formula editor the text background changes to yellow (see Figure 4.2a), which does not follow the Android standards.	2.2	For text highlighting the default Android style should be used, which is a light blue color.
30	2	Long variable names	When choosing a (very) long variable name the value of the variable is not visible anymore within the variable list view (see Figure 4.2b).	2.2	If the variable name is too long it should be shortened by using '...', but the value should still be visible, i.e. thisisaverylongvar...: 0.0 .
31	8	Function order	The functions have no specific order.	1.6	The functions should be sorted, e.g. trigonometric functions one after the other etc.
32	2	The delete button is badly positioned	The button moves along with the text field in is somewhat difficult to find in the text field.	1.6	Move the delete button to the top of the number pad.
33	2	Brick highlighting	After inserting a variable in the formula editor, the light glow on the brick disappears.	1.6	The part of the brick should still glow.
34	3	Landscape mode	Landscape mode is not supported in the formula editor.	1.6	-



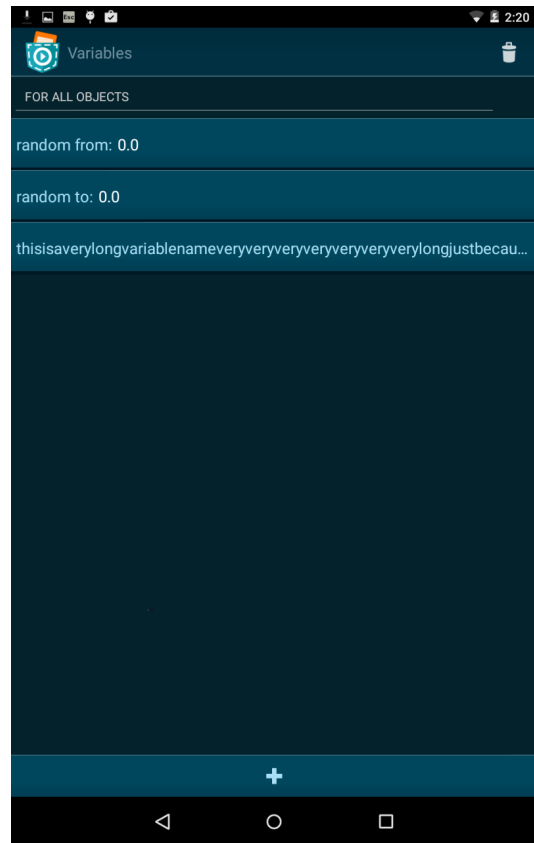
No.	H.	Name	Description	R.	Proposed Solution
35	1	Space key	It would be helpful to have a space key to be able to create some visual differentiation between sub formulas.	1.4	-
36	1	OK button	The OK Button is redundant, since you can return to the scripting view using the Android back button.	1.6	-

Table 4.3: Results of the heuristic evaluation.





(a) Inconsistent text highlighting.



(b) Very long variable name .

		Compute		OK
7	8	9	÷	Object
4	5	6	×	Math
1	2	3	-	Logic
.	0	=	+	Sensors
(	)	Random		Variables

(c) Unnecessary '=' operator.

Figure 4.2: Issues uncovered in the heuristic evaluation (2).

## 4.2 Summative Usability Study

The study presented has been designed to compare the usability of the formula composition method in Scratch to Pocket Code. We chose Scratch as the alternative application since first, Pocket Code has been inspired by Scratch and second, it is the state of the art educational programming environment. The controlled experiment was conducted with teenage participants as a repeated-measures design with a counterbalanced interface order, and included four tasks per application. As we attempt to measure the usability, we focus on three measurable attributes according to the ISO 9241-210:2010: (1) effectiveness, measuring the accuracy and completeness with which users achieved specified goals, (2) efficiency, the relation between time spend in order to accomplish a particular task, and (3) satisfaction, the user's positive attitudes towards the system [62].

Our goal in this study was to test the following null hypothesis:

*The electronic pocket calculator metaphor (hybrid textual/visual approach) is more effective, efficient, and yields a greater perceived user satisfaction than a purely visual approach in the context of formula manipulation.*

It should be noted that we solely investigated the formula manipulation and not the comprehension of already existing formulas. Even though it might be argued that a certain level of understanding of the formula is necessary in order to edit it, we did not put a specific focus on this within our research [62].

### 4.2.1 Formula Manipulation in Scratch

Formula manipulation in Scratch is achieved purely visually through the Lego-style-block metaphor. Operators and variables are predefined blocks, which are interlocked in order to compose formulas. The *Operators* category in Scratch contains arithmetic, relational, and logical operators as well as mathematical and string functions necessary for formula manipulation. In Figure 4.3 a nested formula is depicted comprising a variable (*X Position Paddle*), object attributes (*x position* and *y position*), as well as logical, arithmetic, and relational operators. The sign operator is not part of the *Operator* category in Scratch, depicted in Figure 4.4, but is inserted as part of the numeric value [62].

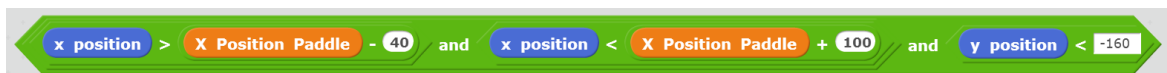


Figure 4.3: Nested formula in Scratch.

When creating such a nested formula in Scratch it is necessary to be aware that groupings are implicit in the connection of the bricks, e.g. a textual representation of the formula in Figure 4.3 would be:

$$\left( \left[ x \text{ position} > (X \text{ Position Paddle} - 40) \right] \text{and} \left[ x \text{ position} < (X \text{ Position Paddle} + 100) \right] \right) \\ \text{and} (y \text{ position} < -160)$$

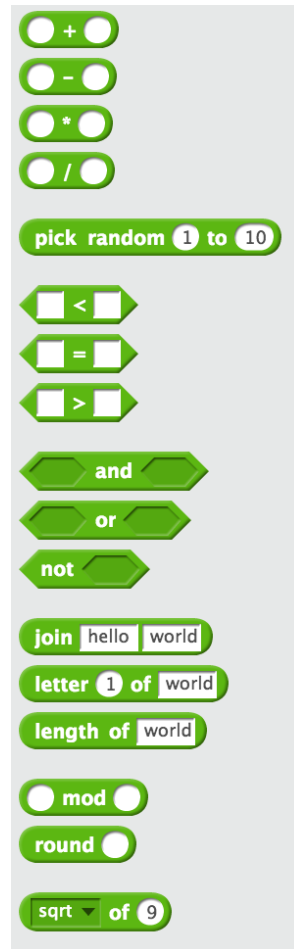


Figure 4.4: Operator category in Scratch.

A tree representation of this formula is given in Figure 4.5. As can be seen Scratch has a prefix notation, thus, in order to achieve a correct formula one has to apply the operator before the operands and construct the formula starting from the most outer operator to the most inner one. Looking at the tree formula construction would translate to traversing the tree top-down, i.e. from the root node to the leaves.

Therefore, when replacing an operator within the deeper levels of a formula it is necessary to remove any statement placed within the operator. For example, in case we want to change the *and* operator in the second nesting level, we would need to execute the following steps:

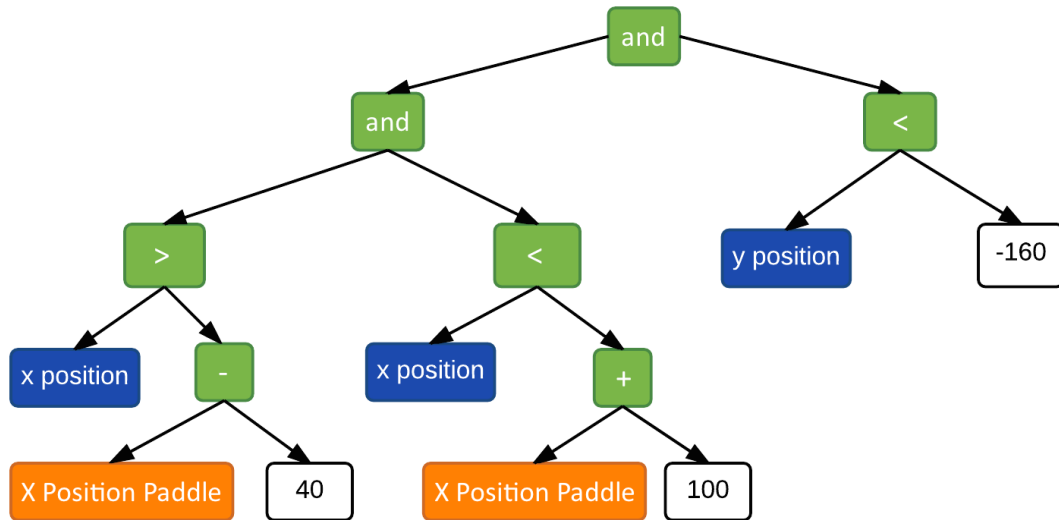


Figure 4.5: Formula construction in Scratch.

1. Extract the *and* from the whole formula.
2. Remove each inner formula from the *and*, i.e. the  $>$  and  $<$  formulas.
3. Remove the empty *and* operator and add the *or* operator.
4. Place the  $>$  and  $<$  formulas as operands to the *or* operator.
5. Add the *or* operator to the entire formula.

Figure 4.6 shows graphically the steps necessary. In order to visually signal at which position a block will be entered, Scratch provides an illumination of the field in which the current brick will be dropped (see Figure 4.7).

#### 4.2.2 Procedure and Data Collection

In order to compare subjective preferences, we intended for every participant to test both applications. Therefore, within each test session, a single participant consecutively evaluated both programming environments after completing a training session. Using a within-subject design reduces some of the variation within the sample, facilitating the detection of a difference between the products if one exists. To counterbalance any learning bias we alternated the sequence of the applications [59, 62]. Each study session lasted between 45 minutes to one and a half hours in total and included:

- an introduction of the facilitator on the test objective, the used equipment and the role the participants play within this setting,



Figure 4.6: Replacing the *and* operator in the second nesting level.

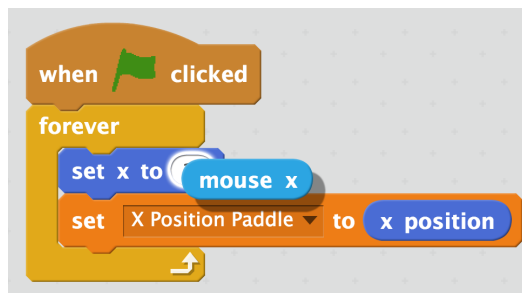


Figure 4.7: Illumination of the drop location of a brick in Scratch.

- a pre-test questionnaire on basic demographic information as well as prior experiences with computers and smartphones (see Appendix 7.1),
- a tutorial for each application,
- a post-task question on the ease of the task,
- a post-application questionnaire on the application in general,
- a post-test questionnaire (see Appendix 7.3).

As both applications are specialized for different environments, we devised a test where each program is investigated in its intended set-up. Pocket Code was examined on a second generation Nexus 7, a seven inch tablet. In the case of Scratch we tested the browser-based version on a fifteen inch laptop computer with an attached optical mouse. Morae<sup>2</sup> screen capture software recorded the laptop screen as well as the participants' facial expressions via the integrated web cam. Further, the post-test questionnaires was taped with Morae. A screen recorder app was used to record the Pocket Code test sessions on the device itself.

<sup>2</sup> <http://www.techsmith.com/morae.html>

Therefore, the visual touch feedback of the Nexus' was enabled. Moreover, eye-tracking glasses<sup>3</sup> were utilized in both set-ups [62]. When conducting an eye tracking study, the first step within each test session is to calibrate the system by asking the participant to look at predefined points [12]. Further, it is necessary to have an observer continuously examining whether the device is recording and the entire screen is clearly visible from the eye tracking glasses. In order to analyze the data, we eventually had to sync the recordings from Morae, the eye-tracking glasses, and the screen recordings from the device.

Figure 4.8 depicts the test set-up, where a participant wears the eye tracking glasses and performs a task in Pocket Code using a tablet with a screen recorder application running. The integrated camera of the laptop records the facial expressions with Morae. The test-set up for Scratch, shown in Figure 4.9 is similar, except that Morae is not only used to capture the participant's facial expressions but also to record the laptop screen. Figure 4.10 depicts a screen shot of a recording of the eye tracking glasses during an evaluation of Scratch as well as a screen shot of the video the integrated web cam records of the participants face. In addition to the automatic data collection, for every test session there was at least one observer taking notes.



Figure 4.8: Test set-up for Pocket Code.

Some of the test sessions were conducted in schools, while others took place at the university. No matter the location, the set-up was identical, as the set-up only requires a desk, chairs, an Internet connection as well as the tablet, laptop and the eye tracking glasses.

<sup>3</sup> <http://www.smivision.com/en.html>



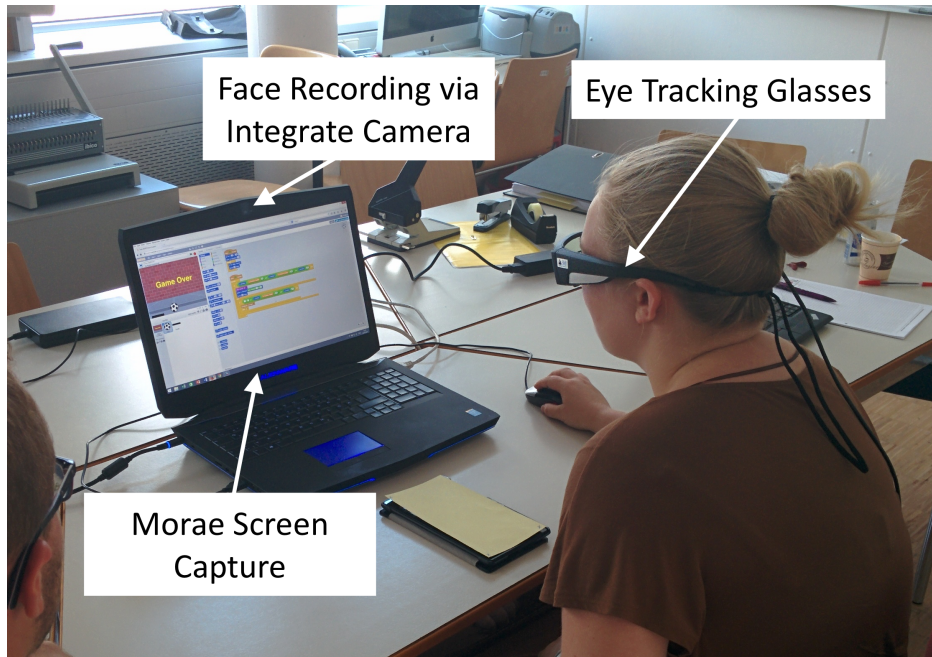


Figure 4.9: Test set-up for Scratch.

#### 4.2.2.1 Participants

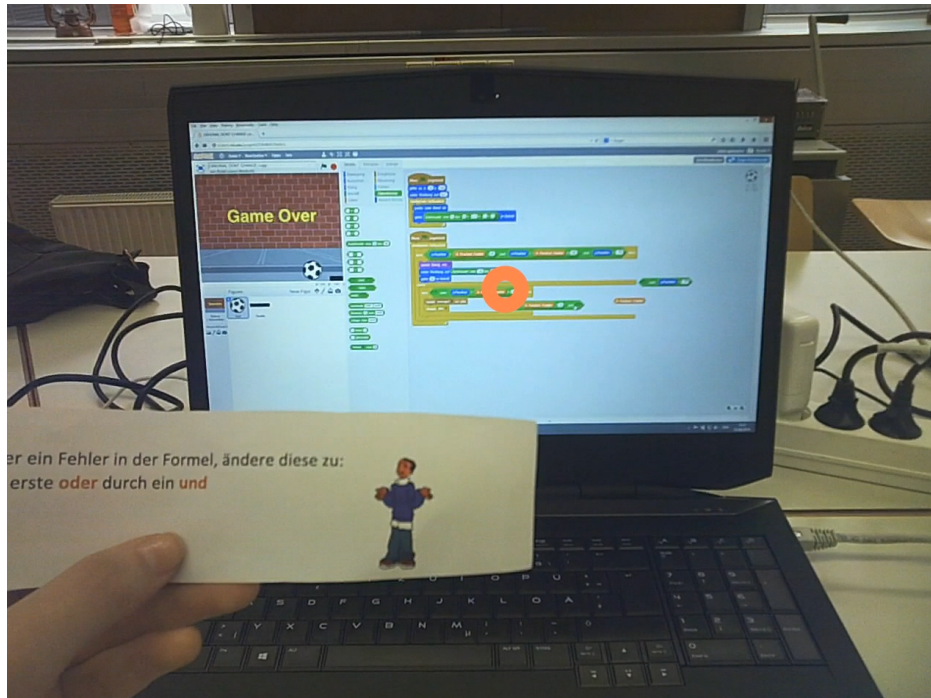
For our usability study our target users were teenagers between the age of twelve and eighteen years<sup>4</sup>, with basic computer skills. In the context of the mobile device proficiency we required the test users to have some experience with Android smart phones or tablets. Moreover, we demanded that the participants were novices or at most very inexperienced users of Scratch as well as Pocket Code and additionally, our test users should have little to no prior programming experience in general. To recruit the users, we utilized convenience sampling, i.e. mobilizing teenagers from schools close to the university. Appendix ?? comprises the demographic and technology experience information of the test users [62].

#### Sample Size Computation

To estimate a suitable sample size Sauro and Lewis [121] propose to use the following equation

$$n = \frac{(t^2 s^2)}{d^2} \quad (4.1)$$

<sup>4</sup> Since most of our test participants were under aged, we required a consent form from their parent or legal guardian prior to the test.



(a) Eye Tracking recording of the laptop screen.



(b) Facial expression recording using the integrated laptop camera.

Figure 4.10: Recordings used for data collection.

where  $t$  is the critical value,  $d$  is an observed difference (i.e. smallest difference we should be able to detect) and  $s^2$  is an estimate of the variance (ideally of a similar study). Since  $t$  depends on the degrees of freedom and therefore on the sample size, there is an iterative way to compute  $n$ , which removes this issue. The iterative steps are [121]:

1. Initially use the z-score instead of the t-score for the desired confidence level.
2. Compute the initial sample size  $n = \frac{(z^2 s^2)}{d^2}$ . Round up to the next whole integer.
3. Compute the t-score with the computed sample size estimate, i.e. the degrees of freedom are  $n - 1$ .
4. Recalculate  $n$  with the computed t value.
5. Repeat Step 3 and 4 until the sample estimation does not change for two iterations.

If there is no previous estimate of the variance available, it is possible to handle this by defining  $d$  as an effect size. The effect size measures the magnitude of a results:

$$e = \frac{d}{s} \quad (4.2)$$

According to Sauro and Lewis [121] as a rule of thumb the value of  $e$  should be set to 0.33 to uncover small effects, 0.5 to find medium effects and a value of 0.8 for large effects. The equation for  $n$  then changes to:

$$n = \frac{t^2 s^2}{d^2} = \frac{t^2 s^2}{(es)^2} = \frac{t^2 s^2}{e^2 s^2} = \frac{t^2}{e^2} \quad (4.3)$$

Since there is no estimate for the variances available for our experiment as no comparable study has been conducted within the project, we utilized Sauro and Lewis' method to estimate a sample size. We decided on a 90% confidence interval and an effect size  $e = 0.33$  which should enable us to detect a rather small effect. Table 4.4 shows the iterative sample size computation as described by Sauro and Lewis [121].

	<b>Initial</b>	<b>1</b>	<b>2</b>	<b>3</b>
$t$	1.645 <sup>5</sup>	1.318	1.341	1.337
$e$	0.33	0.33	0.33	0.33
$df$	-	24	15	16
<b>Unrounded</b>	24.84	15.95	16.51	16.4147
<b>Rounded up</b>	25	16	17	17

Table 4.4: Sample size iteration procedure for t-tests [121].

<sup>5</sup> In the initial iteration we use the  $z$  value instead of the  $t$  value.

### Test users

We have recruited 23 participants. Of these individuals three participants already had object-oriented programming experience and one was younger than the Pocket Code target age and was not familiar with the use of electronic pocket calculators. Another participant was the designated pilot test user. Therefore, their data have been subsequently removed from the quantitative analysis. The remaining 19 participants' ages covered a range from twelve to eighteen years of age, with an average age of fifteen years ( $SD=1.89$ ), with ten male and nine female participants. All participants were established users of traditional desktop computers as well as mobile devices, such as smartphones, tablets, or both [62].

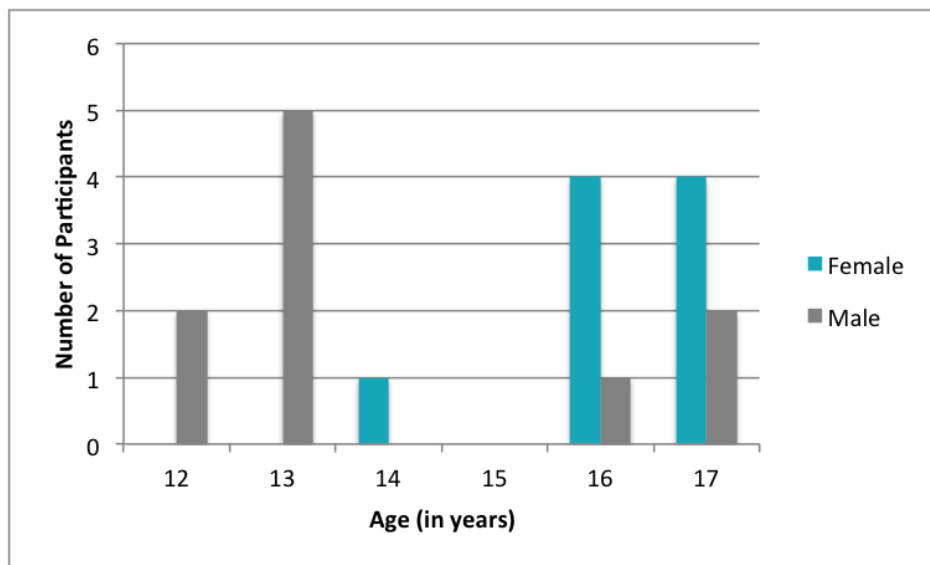


Figure 4.11: Age and gender of participants.

#### 4.2.2.2 Pilot

We conducted a pilot study with a fifteen year old male participant in order to discover initial design flaws. In consideration of the feedback from the pilot session, the tasks' descriptions were rephrased and the overall test was shortened [62].

#### 4.2.2.3 Training Sessions

Due to the requirement of having novice Scratch and Pocket Code users, we presented a tutorial video on Scratch and Pocket Code respectively to each participant before testing the applications. The users were asked to participate in the tutorials and simultaneously follow the steps on the devices. In Figure 4.12 a male participant is shown following the video training on the laptop while at the same time creating the program on the tablet device. In case of Scratch, a second laptop was utilized to present the instructions. The training

sessions for each application focused on the same exercises and introduced some of the features evaluated in the test. This measure was introduced to allow participants to acquire the skills necessary for formula composition. Further, it introduces some of the terminology utilized within the usability test, such as “brick” [62].

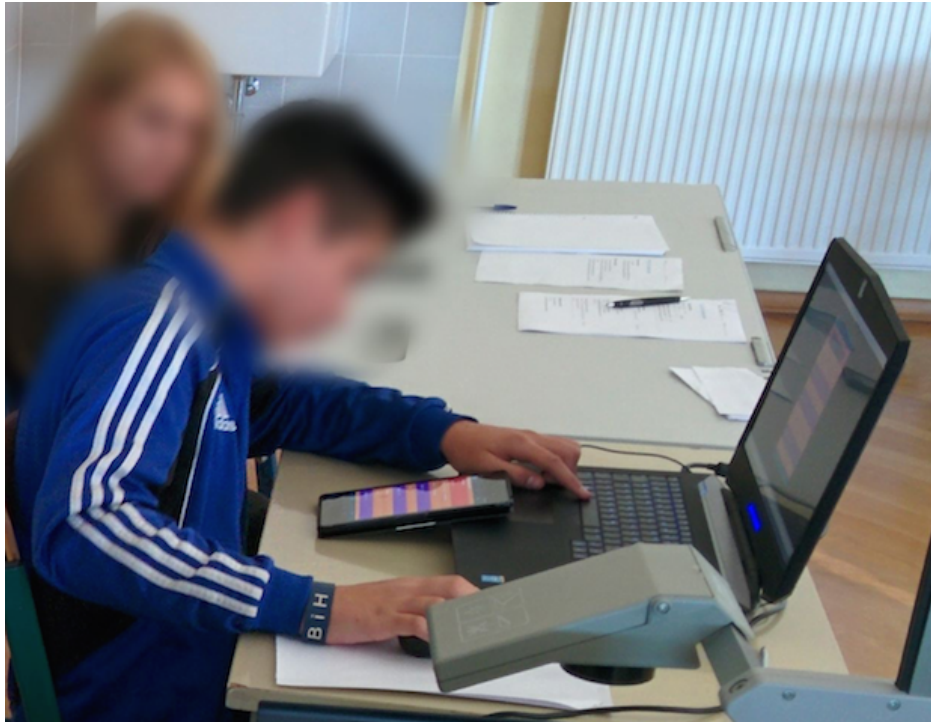


Figure 4.12: Training session for Pocket Code.

#### 4.2.2.4 Tasks

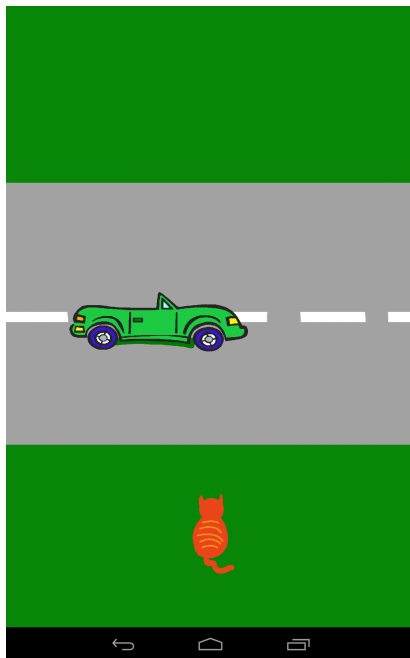
Each test covered four tasks per application, testing identical features in each programming environment. The tasks were sorted according to their difficulty, making the last task the most challenging one. The participants were asked in written language to compose formulas as well as change already existing formulas, ranging from simple insertions of object attributes to more complicated nested formulas containing logical operators, mathematical functions, and numerical constants. In order to make the test more engaging, we incorporated our tasks within the setting of an already created game. Each task was presented to the participant one at a time. In Sections 7.2 of the Appendix the German tasks are illustrated as presented to the participants. The initial task, Task 1, does not concern formula manipulation but has been listed to describe the complete experiment [62].

### Task 1

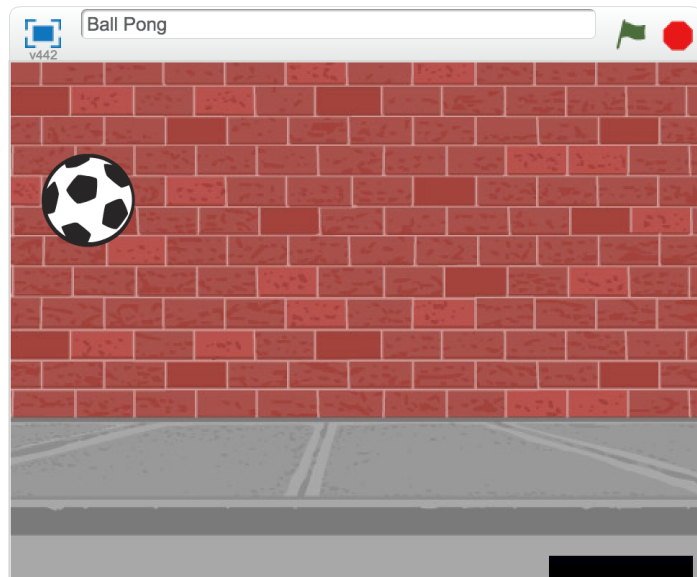
As mentioned before Task 1 was an introductory task aiming at acquainting the participants with the test situation and the games they have to adapt during the sessions.

**Pocket Code:** *Try out the game Kitty Cross.*

We programmed a simple game “Kitty Cross”, inspired by Frogger, where the goal is to get the cat across the street to the top of the screen. The car drives continuously on the road from one side of the screen to the other. In Figure 4.13a the starting situation is shown. When tapping the cat, it moves forward towards the top of the display. However, in this initial program, we have not programmed collision detection between the cat and the car, for example.



(a) Kitty Cross.



(b) Ball Pong.

Figure 4.13: Programs used for performing the tasks.

**Scratch:** *Try out the game Ball Pong.*

The idea of the remixed game “Ball Pong” is to use the paddle to keep the ball from touching the ground. The black paddle can be move with the mouse. However, instead of setting the paddles location to the x position of the mouse it uses the y position in the initial set-up. In Figure 4.13b we see the start situation. The game is over when the ball touches the bottom of the screen.

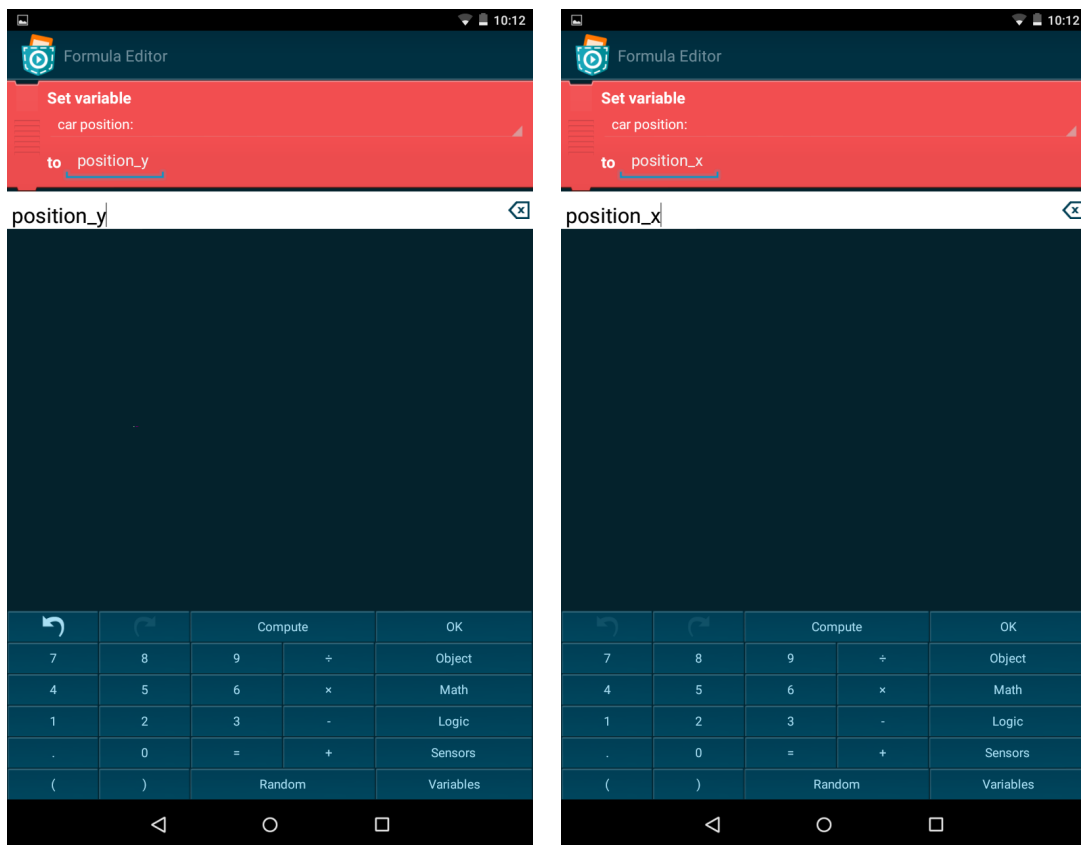
## Task 2

In this task we had the participants replace an object's attribute with another one for a specified brick.

**Pocket Code:** *We want the game to be over when the cat gets hit by the car. But there is an error in the Car scripts.*

- Go to the Car scripts.
- Look for the brick Set variable 'car position'.
- Replace the `position_y` with `position_x`.

Once the user has located the correct brick and entered the formula editor view, the task consists of removing `position_y` via the delete button, selecting the *Object* category, tapping the attribute `position_x`, which returns the user to the formula editor view. Lastly, confirming the formula with the *OK* button completes the task. Figures 4.14a and 4.14b show the start state as well as the goal formula. A minimum of four taps are necessary to manipulate the formula as intended.



(a) Start screen.

(b) Goal screen.

Figure 4.14: Task 2 Pocket Code.

**Scratch:** *The paddle does not seem to work correctly, since it does not follow the mouse. Please repair it:*

- Go to the scripts of the Paddle.
- Look for the `set x to brick`.
- Replace the `mouse_y` with the `mouse_x` brick.

In order to complete this task, the user has to find the correct brick within the program code of the Paddle. Figure 4.15a shows the corresponding bricks before the user attempts to solve the task. To complete this task, the user has to remove the light blue `mouse_y` brick, go to the *Sensing* category, select the `mouse_x` brick and drag it from the category to the `set x to` brick and drop it. It would require a minimum of three clicks once the script view of the Paddle has been located to achieve the goal state as depicted in Figure 4.15b.

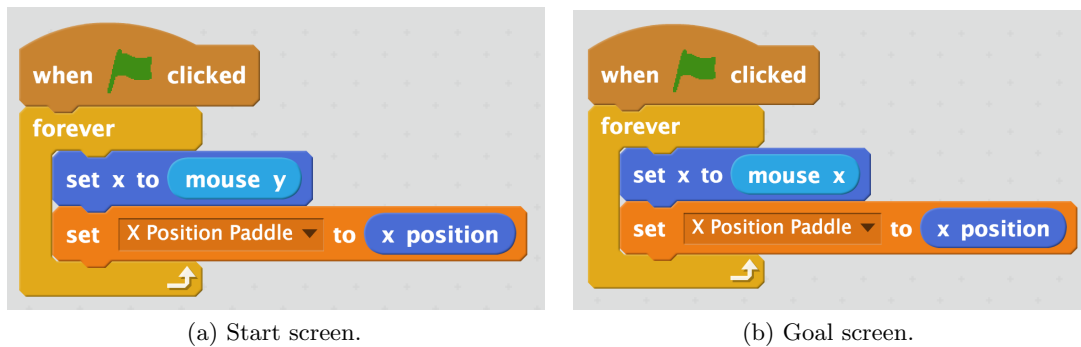


Figure 4.15: Task 2 Scratch.

### Task 3

In this task we had the users find a specific brick of an object and enter a formula anew. In particular, the formulas contain numerical values and the random number generator function. We assume common operator precedence. As we wanted the participants to be made aware of the planning necessary involved in constructing a formula, i.e. especially in Scratch it is essential to place the blocks with lower precedence first. Therefore, we did not provide a formula grouping via parenthesis. The last instruction then requires the users to change one operator and two numeric values.

**Pocket Code:** *Our cat is too slow, help it go faster! Do the following:*

- Go to the Cat scripts.
- Look for the brick `Change Y by _____`.
- Enter the following formula into that brick:  $3 * \text{random}(5,10) + 4 / 0.5$ .
- Change the formula to:  $3 * \text{random}(6,12) + 4 * 0.5$ .



In Pocket Code the user can create the formula easily from left to right, by adding the numerical values as well as random number generator. Note that within this task, it is possible to complete this without entering a category, since the random number generator as well as arithmetic operators are all located within the formula editor view. To construct the formula from the start screen (see Figure 4.16a) to the first goal screen (see Figure 4.16b) requires at least 16 taps. Changing the formula then is merely exchanging the parameter values of the random number generator, removing the division operator and adding the multiplication operator. Of course in this task it is necessary to position the cursor at the correct location within the formula to be able to replace the numeric values and operators. Completing the formula as represented in Figure 4.16c takes an additional nine taps.

**Scratch:** *The game is too easy like that, we want the ball to go faster.*

- *Go to the scripts of the Ball .*
- *Look at the first block of bricks, there is a move \_\_\_\_\_ steps brick.*
- *Enter the following formula into that brick: pick random(0 to 8) / 0.5 - 3 \* 2.*
- *Change the formula to: pick random(0 to 8) / 0.5 + 1 \* 1.*

Once the user has found the correct scripting view, the next step is to pick the division operator from the *Operator* category to the start screen (see Figure 4.17a). Note that all bricks necessary to solve this task are located within the *Operator* category. Only then the random number generator `pick random( _____ to _____ )` should be entered. Afterwards the multiplication and then the subtraction operators should be added. Additionally, the numerical values can be inserted at the end or during the formula composition. In order to reach step three of the task (see Figure 4.17b) at least four clicks plus adding the numerical values through the keyboard are necessary. To exchange the subtraction with an addition within the second half of the formula, the user has to remove the minus operator, select and add the plus operator as well as the multiplication. The last step is to add the numeric values. This adaption of the formula requires ideally three clicks as well as typing the operands. The goal state of this task is depicted in Figure 4.17c.

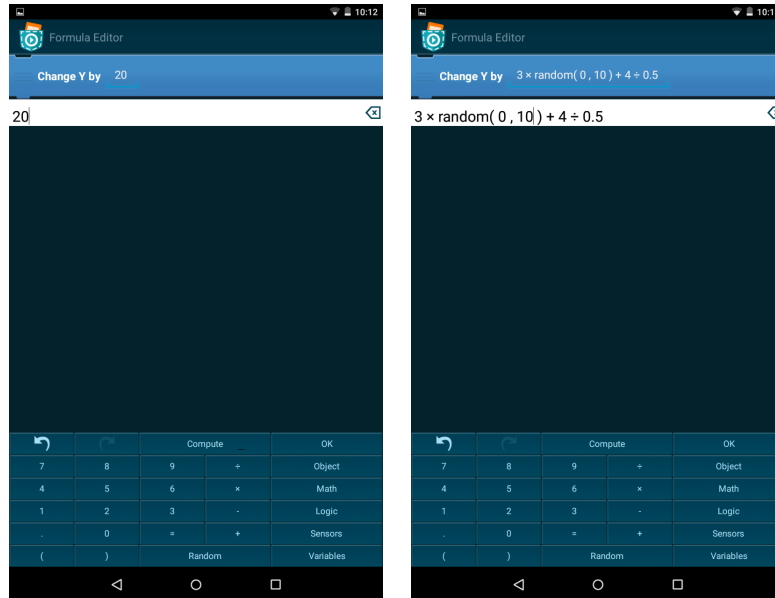
#### Task 4

In this task we asked the participants to add a more difficult and nested sub formula to an existing formula.

**Pocket Code:** *We still have to program the collision. Perform the following steps:*

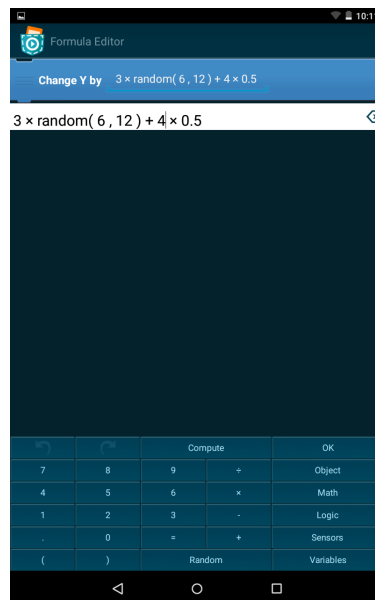
- *Go to the Cat scripts.*
- *Look for the If brick.*
- *Change the formula*

*From:* `((position_y > -180) AND (position_y < 180)) OR 0`



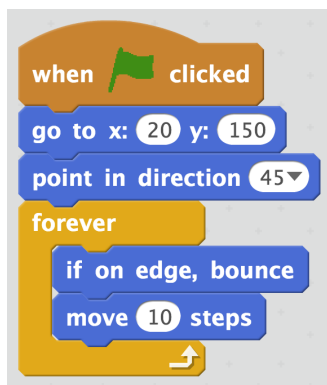
(a) Start screen.

(b) After entering the first part of the formula (i.e. without step 4).

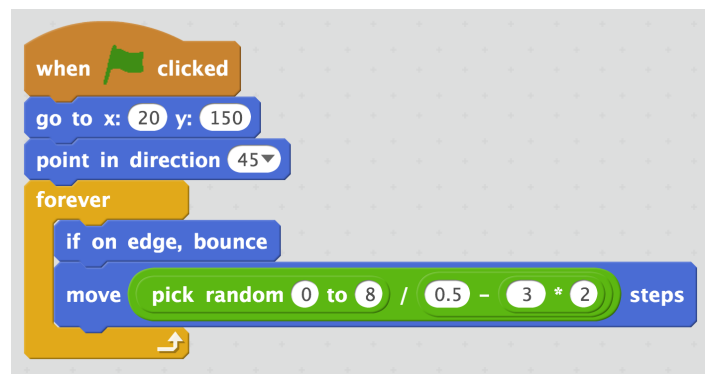


(c) Goal screen.

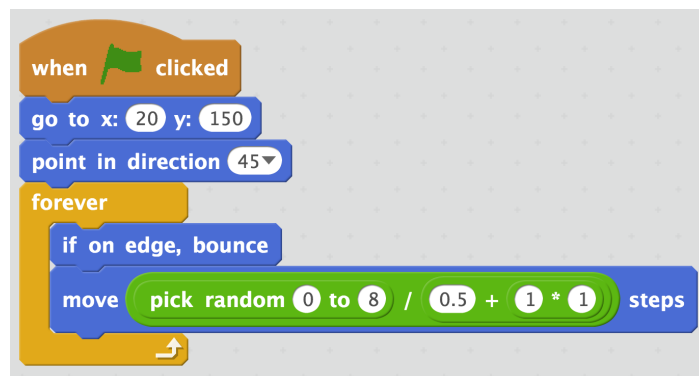
Figure 4.16: Task 3 Pocket Code.



(a) Start screen.



(b) After entering the first part of the formula (i.e. without step 4).



(c) Goal screen.

Figure 4.17: Task 3 Scratch.

To:  $((\text{position\_y} > -180) \text{ AND } (\text{position\_y} < 180)) \text{ OR } (("\text{car position}" > -150) \text{ AND } (("\text{car position}" < 150))$

*Hint: car position is a variable.*

The first step in creating the desired formula depicted in 4.18b from the formula in Figure 4.18a, is to remove the right part of the disjunction, i.e. the zero. To obtain the variable "car position" the user has to choose the *Variable* category. The logical and relational operators are located within the category named *Logic*. To construct the formula at least 25 taps are necessary.



(a) Start screen.

(b) Goal screen.

Figure 4.18: Task 4 Pocket Code.

**Scratch:** *We want the game to end when the ball hits the floor. Perform the following steps:*

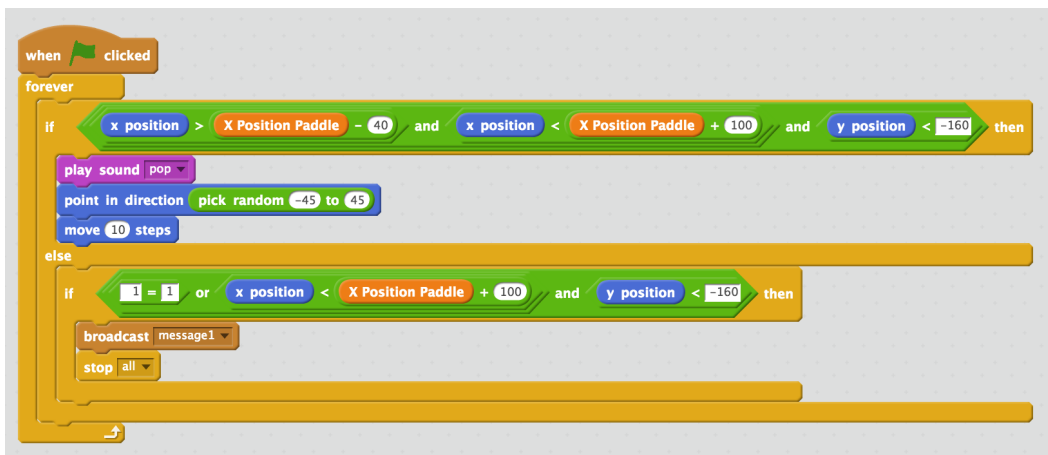
- *Go to the scripts of the Ball.*
- *In the 2nd brick block look for the second if \_\_\_\_\_ then brick.*
- *Change the formula*

*From:*  $(1=1)$

To:  $x \text{ position} > (\text{X Position Paddle} - 25)$

Hint: X Position Paddle is a variable.

To create the required formula, the user has to delete the comparison operator brick. Then the relational  $>$  has to be added from the *Operator* category.  $x \text{ position}$  is the Ball's position on the x axis and the corresponding brick is located within the *Motion* category. To complete the right side of the formula, first the subtraction has to be entered and then the variable X Position Paddle has to be dragged from the *Data* category to the position. Lastly, the numeric operand has to be added. To reach from the start state, depicted in Figure 4.19a, the goal state, shown in Figure 4.19b, at least five clicks and adding the numeric values are necessary.



(a) Start screen.



(b) Goal screen.

Figure 4.19: Task 4 Scratch.

#### 4.2.2.5 Questionnaires

After each task, the participants were asked to judge its overall difficulty. The goal was to evaluate each task according to the users' attitudes, in addition to quantitative measurements. We used a derivation of the SEQ [118], allowing an answer on a five-point Likert scale from *very easy* to *very difficult*. Sauro and Dumas [118] compared three one-question rating types and determined that SEQ was easy and the users' responses on the SEQ correlate to task-time and task-completion rates [120]. We decided to adapt the SEQ, to from seven to five points in order to be consistent with our other questionnaires. Note that we used in addition to the terms smiley faces on the Likert scales, ranging from a smile to a frown, as can be seen in Figure 4.20 for the SEQ [62].

Task: \_\_\_\_\_

Overall, this task was:






Very easy 	Easy 	Okay 	Difficult 	Very difficult 
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 4.20: SEQ

Each application test concluded with an adapted SUS [18]. First, we decided to use an all positive-version of the SUS, as the scores are similar to the standard SUS and the participants are less likely to confuse the points on the scale by mistake. Second, we switched the position of the terms, i.e. usually *Strongly disagree* is located at the left side of the scale, yet, we decided to move it to the end of the scale on the right side in order to comply to our other questionnaires [62]. Third, we again added smiley and frown faces to the point in addition to the labels. Fourth, since we conducted the usability tests in German we utilized a German version [69]:

1. Ich denke, dass ich dieses System gerne regelmäßig nutzen würde.
2. Ich fand das System unnötig komplex.
3. Ich denke, das System war leicht zu benutzen.
4. Ich denke, ich würde die Unterstützung einer fachkundigen Person benötigen, um das System benutzen zu können.
5. Ich fand, die verschiedenen Funktionen des Systems waren gut integriert.
6. Ich halte das System für zu inkonsistent.
7. Ich glaube, dass die meisten Menschen sehr schnell lernen würden, mit dem System umzugehen.
8. Ich fand das System sehr umständlich zu benutzen.

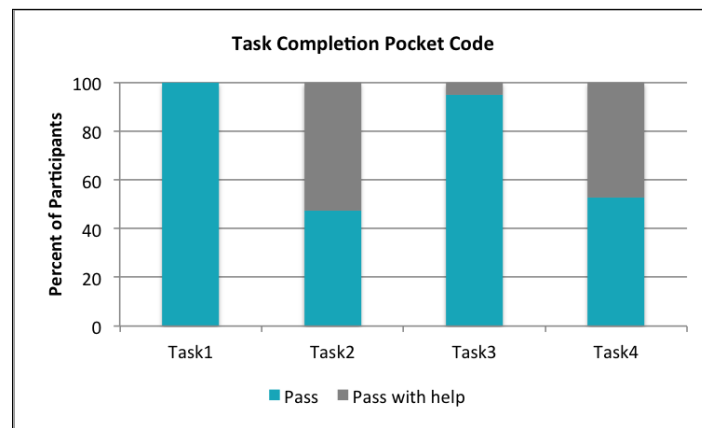
9. Ich fühlte mich bei der Nutzung des Systems sehr sicher.

10. Ich musste viele Dinge lernen, bevor ich mit dem System arbeiten konnte.

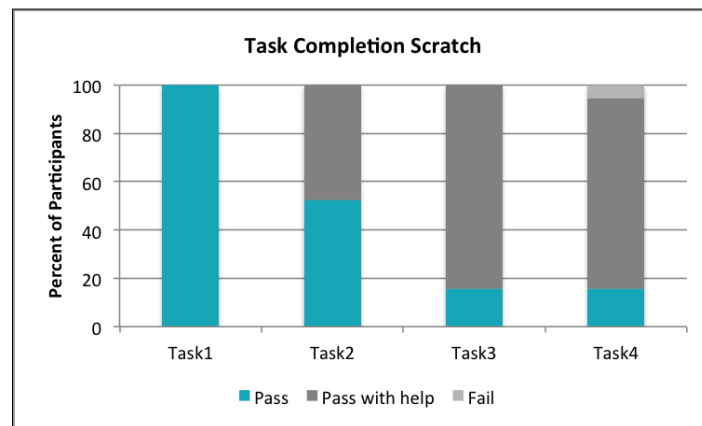
At the end of the test session the participants were further interviewed on difficulties faced during the entire usability test as well as subjective preferences.

### 4.2.3 Results

The evaluation is based on the quantitative and qualitative data collected during the experiment. As we examined Pocket Code and Scratch in the context of usability, we focused on three main attributes, namely effectiveness, efficiency, and user satisfaction [62].



(a) Pocket Code task completion



(b) Scratch task completion

Figure 4.21: Task completion rate by application and task.

#### 4.2.3.1 Effectiveness

When evaluating the results we categorized three levels of task completion [62]:

- *Pass*: The correct and complete solution was produced by the participant.
- *Pass with help*: The solution was obtained with additional comments from the facilitator towards the solution.
- *Fail*: The participant was not able to complete the task in a timely manner or was not making any progress.

When participants asked about the task description itself, e.g., the block nesting or grouping was unclear, this would not be considered as help. Further, if the participant initially produced an incorrect formula, but then was able to correct it, the task was still reported as passed without help [62].

In Figure 4.21 the task completion rate for both applications is presented. As can be seen Pocket Code ( $M=73.68\%$ ,  $SD=27.52\%$ ) has a higher overall task completion without help rate than Scratch ( $M=46.05\%$ ,  $SD=39.94\%$ ). The results from the introductory Task 1 are plotted solely for reasons of completion, as the task did not address formula manipulation. In Task 2, participants were asked to interchange a single not nested attribute. As can be seen from Figure 4.21 Pocket Code (47.37%) and Scratch (52.63%) feature around the same task completion rate, since the attribute to be changed has not been nested. The difficulties stem from the users having problems in finding the corresponding blocks in both applications [62].

As illustrated in Figure 4.21, Task 3 and 4 respectively have been especially problematic in Scratch, as less than 1/6 of the users were able to complete these tasks on their own. In the third task the users were asked to create a formula and then edit it by changing a single operator. In Pocket Code, replacing an operator can be done efficiently as only the single operator has to be deleted and the new one introduced. In Scratch—due to the nesting of the blocks—there are dependencies between the statements. By removing bricks, which have other blocks nested within, the deeper levels are deleted as well. Consequently, participants accidentally removed necessary formula parts and had difficulties recovering from those errors [62].

The last task was especially challenging as on the one hand it required the participants to find the according category of the variables and on the other hand create a nested formula and embed it into an already complicated nested structure. Even though variables are color coded in Scratch, and occurrences of the same variable were visible in other parts of the script, the participants were frequently unable to find the suitable category containing the variable. One of the test users could not complete the task in Scratch, hence the facilitator stopped the test [62]. On average each test participant completed 2.95 ( $SD=0.7$ ) tasks successfully in Pocket Code and 1.84 tasks ( $SD=0.83$ ) in Scratch.

#### 4.2.3.2 Efficiency

Due to the dependencies of nesting levels in Scratch, participants were forced to reenter some of the blocks they accidentally removed previously. The effects are observable in the time spent on Task 3 and 4 according to Figure 4.22a showing the average task times and 95%



confidence interval<sup>6</sup>. In the case of Pocket Code participants faced obstacles in positioning the cursor [62].

Usability task times, are often skewed distributions, meaning that the population mean will be larger than the population median. In general there is a factor between four to ten between the performance of the best and the worst user. In programming this number rises to 20, which means that the best programmer is about 20 times more productive than the worst programmer [93]. Therefore, to estimate the center of the distribution, the geometric mean has proven useful as it is less vulnerable to outliers and in particular useful for sample sizes under 25. To compute the geometric mean, the raw task times are converted using a log transformation. Then we determine the mean of the transformed values and convert it back its original scale, which then is the geometric mean [121]. We computed the geometric mean of the time on task for each application as depicted in Table 4.5.

	Scratch	Pocket Code
<b>Task 1</b>	88.8	25.6
<b>Task 2</b>	126.59	95.89
<b>Task 3</b>	162.25	305.21
<b>Task 4</b>	433.36	228.65

Table 4.5: Geometric mean of time on task (in seconds).

To identify whether there is a significant difference between the task completion times in the two applications we test the following hypotheses:

- $H_0$  : There is no significant difference in task times between the formula manipulation in Pocket Code and Scratch.
- $H_A$  : There is a significant difference in task times between the formula manipulation in Pocket Code and Scratch.

To compute whether the null hypothesis  $H_0$  is true or formulated differently there is a difference between means of continuous or rating scales in within-subject studies, we can utilize a paired  $t$ -test:

$$t = \frac{\check{D}}{\frac{s_D}{\sqrt{n}}} \quad (4.4)$$

where  $\check{D}$  is the mean of the difference scores,  $s_D$  is the standard deviation of the difference scores,  $n$  is the sample size, i.e. total number of pairs and  $t$  is the test statistic using the  $t$ -distribution on the sample size for the two-sided area.

<sup>6</sup> A confidence interval is a measure for location and precision and the common confidence level of 95 % express that when replicating the study 100 times, 95 times the interval will contain the true value, for example the actual mean. The more variation is present in the population, the larger the confidence interval will be, thus to decrease the confidence interval, one has to increase the sample size.

Once we have obtained the test statistic  $t$ , we check whether this is significant by looking up the p-value for the significance  $\alpha = 0.05$ <sup>7</sup> using the t-table. Since task time data is not normally distributed, applying a two-sided t-test is advised as it is more robust and generate accurate p-value with smaller sample sizes with less than 30 participants [121]. Generally, a p-value  $\leq \alpha$  indicates evidence that we should reject  $H_0$ .

Generally Pocket Code seems to be more efficient than Scratch with a smaller time on task. A paired t-test indicates a statistically significant difference ( $t=4.86$ ,  $p=.0000032$ ) between the task times recorded for Pocket Code ( $M=113.3$ ,  $SD=71.05$ ) compared to Scratch ( $M=182.80$ ,  $SD=159.0$ ).

In order to compare the efficiency of both approaches, we combined task success and time on task for each application and calculated the average number of tasks successfully completed (without help) per minute. Figure 4.22b illustrates this average efficiency. Our data reveals strong evidence ( $t=5.91$ ,  $p=.0000134$ ) that the formula manipulation is more efficient in Pocket Code ( $M=0.44$ ,  $SD=0.21$ ) than Scratch ( $M=0.18$ ,  $SD=0.11$ ) [62].

#### 4.2.3.3 Questionnaires

In addition to the performance metrics, we collected self-reported data. Figure 4.23a gives an overview of the average results from the SEQ asked at the end of each task. These results correlate with the collected task completion rates (*Pearson correlation coefficient*  $=-0.96$ <sup>8</sup>,  $p=.0001$ ). Generally, the tasks in Pocket Code ( $M=2.05$ ,  $SD=0.88$ ) were perceived as easier in comparison to Scratch ( $M=2.56$ ,  $SD=1.15$ ) [62].

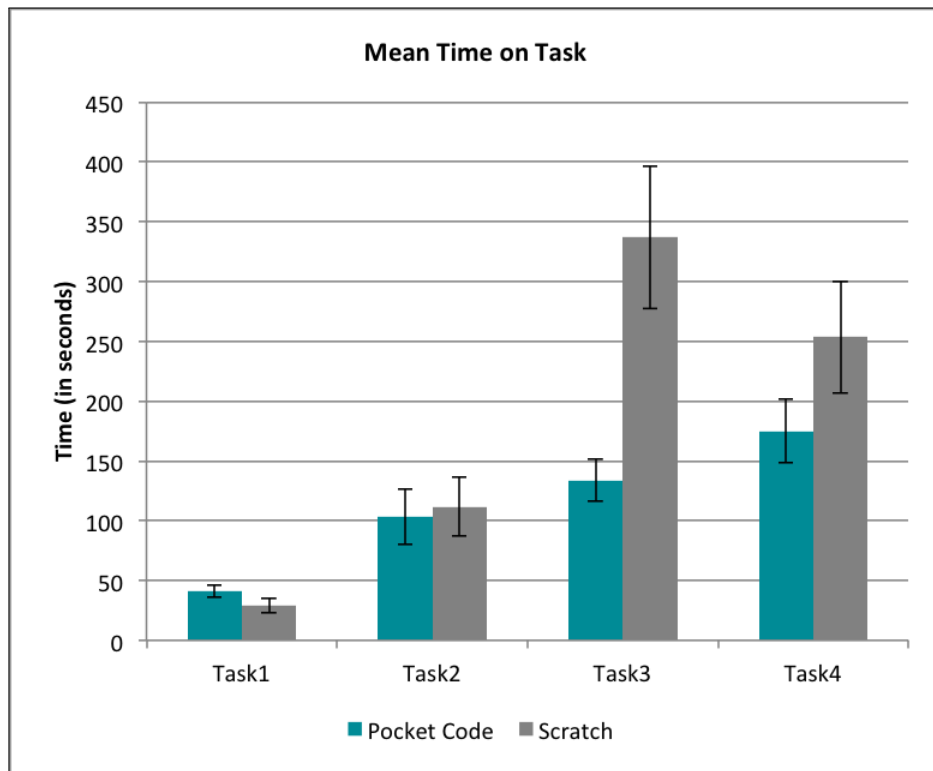
The SUS score of both applications has been below average; we recorded a score of 68.16 for Pocket Code ( $M=68.2$ ,  $SD=9.99$ ) and 60.66 for Scratch ( $M=59.3$ ,  $SD=11.56$ ). The significance test ( $t=3.02$ ,  $p=.007$ ) provides a strong evidence that Pocket Code receives a higher SUS Score. Interestingly, during the post-test questionnaire only a few participants were able to recall specific difficulties they had during the test itself [62].

#### 4.2.3.4 Single Usability Score and Preference Data

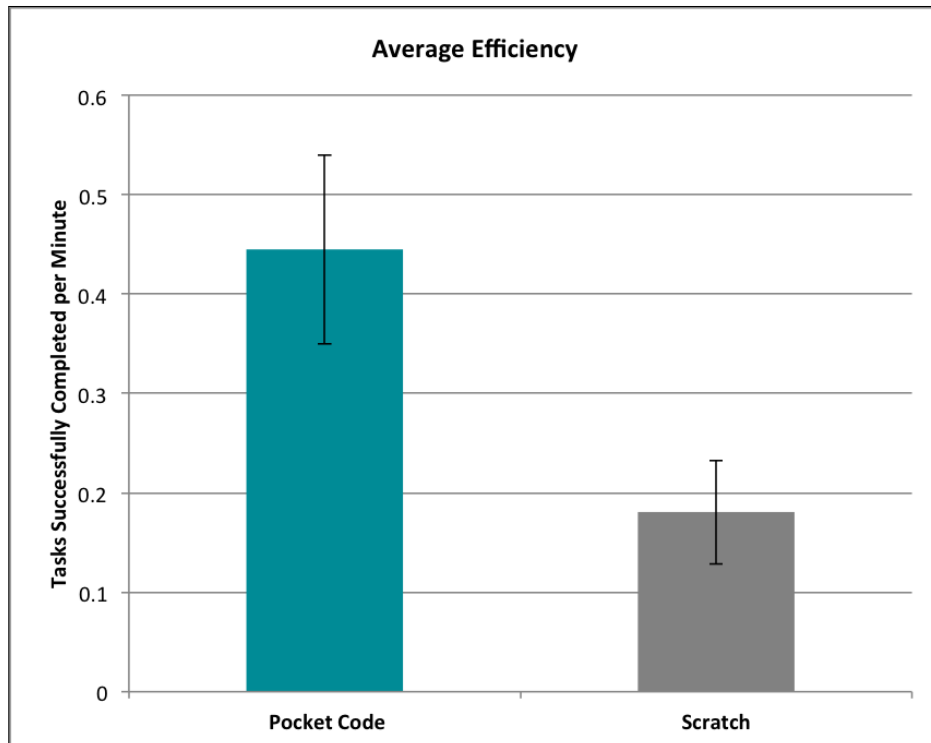
We computed a single usability score [4] for both applications, based on every participant's average time on task, average task completion rate and SUS score. Averaging over all test users we arrived a score of 68.13 % for Pocket Code and 54.62% for Scratch ( $t=5.24$ ,  $p=.00006$ ). When combining the overall results of the collected data into a single score, significant evidence of a higher usability in Pocket Code formula manipulation is apparent in Figure 4.23b. After each test session we conducted a short interview asking the participants about their subjective preferences and about the applications. Table 4.6 lists some of the questions

<sup>7</sup> Indicates the probability we reject  $H_0$  even though it is true.

<sup>8</sup> Note that we observe a negative correlation since task completion is a ratio of completed tasks to all tasks, i.e. making 1.0 the optimal and every value smaller worse, while in the rating for SEQ a higher value denotes more difficulty, i.e. 1.0 again is the optimum, however every value higher is a worse score.

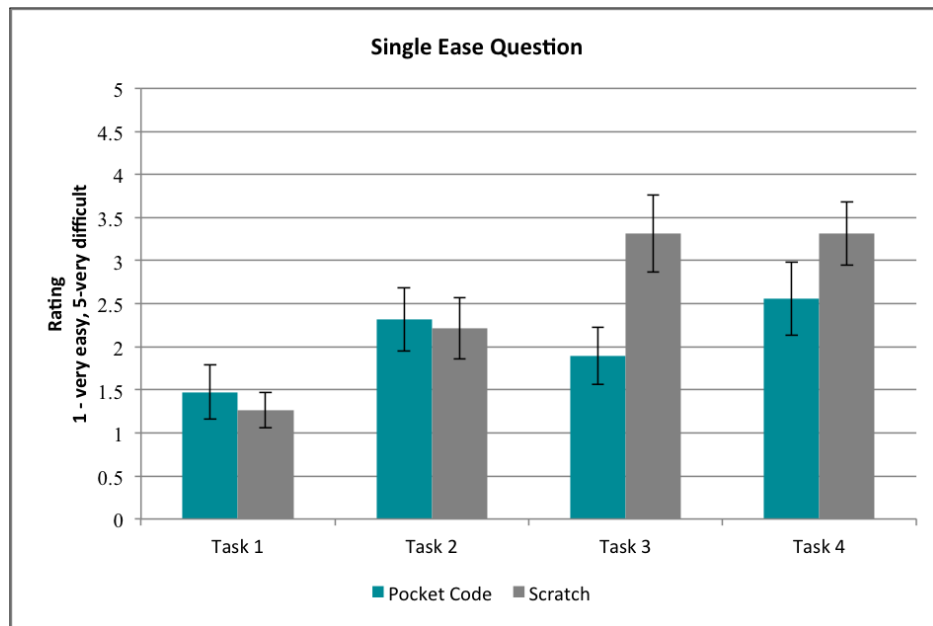


(a) Mean time on task.

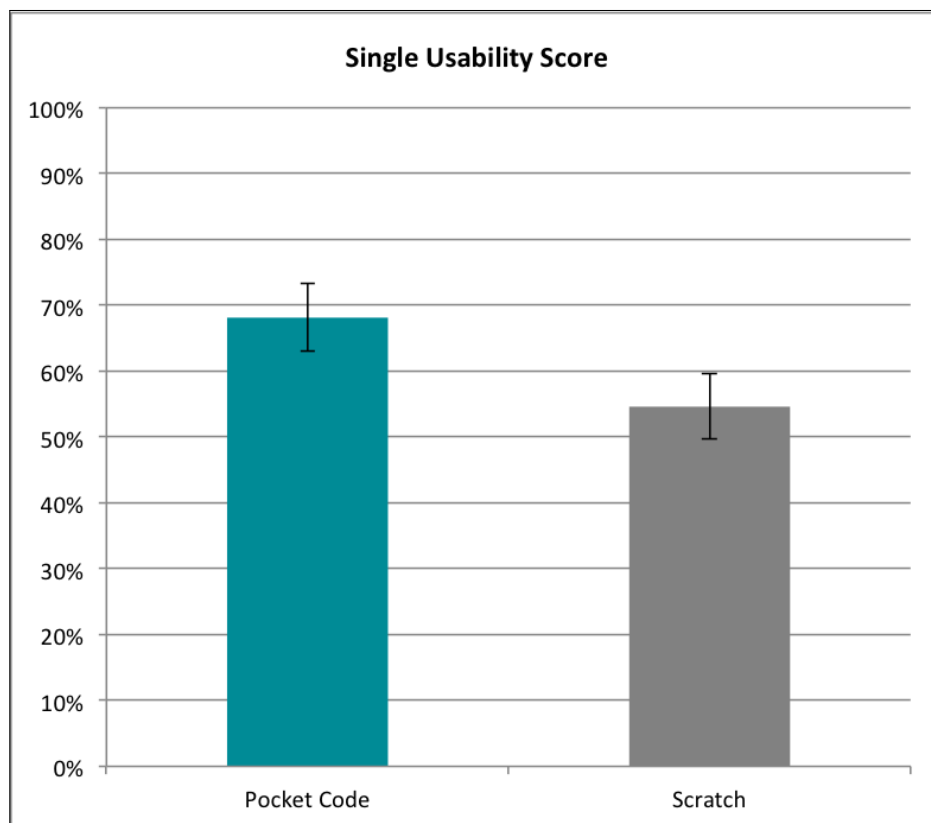


(b) Average efficiency.

Figure 4.22: Efficiency results  
(Error bars represent the 95% confidence interval).



(a) Average SEQ rating (1-very easy to 5-very difficult).



(b) Single Usability Score as an average percentage of time on task, task completion, and SUS-rating .

Figure 4.23: SEQ Scores and Single Usability Score (Error bars represent the 95% confidence interval).

and the percentage ratio of answers. As can be seen Pocket Code was preferred, however, in the context of the formula manipulation lost several percent.

	<b>Pocket Code</b>	<b>Scratch</b>	<b>Undecided</b>
Which application do you prefer?	73.68%	15.79 %	10.52 %
Which application is easier to use?	78.95%	15.79 %	5.26 %
With which application is it easier to work with formulas?	57.89%	26.31 %	15.79 %
If you could choose, which application would you want to learn in school?	63.158%	26.31 %	10.52 %

Table 4.6: Post-test interview answers.

Further, we have collected some of the comments the users made while answering the questions <sup>9</sup>:

- *Which task did you perceive as the most difficult one in Pocket Code?*
  - Formula manipulation in general (1)
  - Where to find the needed operators (3)
  - To know which variables are available and variables in general (1)
  - Task 4, where to find y\_Position (1)
- *Which task did you perceive as the most easy one in Pocket Code?*
  - Inserting formulas in case you know where to find the correct category (1)
  - Task 1 (i.e. Executing a program) (1)
- *Which task did you perceive as the most difficult one in Scratch?*
  - Where to find the needed operators and this takes time (3)
  - Creating and editing formulas (6)
  - To know which variables are available and variables in general (1)
  - Bricks are too small (1)
  - Positioning of the bricks (3)
  - Changing between objects (1)
- *Which task did you perceive as the most easy one in Scratch?*

<sup>9</sup> Note we only provide key phrases, as many comments were of similar content. Further, some participants were not able to recall particularly difficult or easy portions of the test.

- Task 1 (i.e. Executing a program) (2)
- *What do you prefer about Pocket Code over Scratch?*
  - Easier to use (11)
  - Better design, less childish (2)
  - Well arranged (1)
  - Prefer using a tablet/smart phone (2)
  - Can add formula without bricks (3)
- *What do you prefer about Scratch over Pocket Code?*
  - Prefer using a desktop computer (2)
  - Easier to use blocks (3)
  - Is more logical (1)
  - Less prone to mistyping (1)
  - Better layout (2)
  - Formulas easier since bricks are colored and easier to find (1)
  - Can do more with Scratch (2)

#### 4.2.3.5 Eye Tracking

We experienced difficulties in calibrating the eye-tracking glasses for some test participants and overall a decreased quality of calibration during most test sessions. On the one hand, we could not reach an optimal calibration for contact lens users and for female participants wearing mascara. On the other hand, we believe that especially for our younger participants the glasses were too large, i.e. when they were moving the glasses would slide on their nose, thus requiring a new calibration. However, since this would happen various times throughout the test, it would not have been possible to recalibrate every time. Further, we have to note that spectacle wearers are problematic as well. First, it is impossible or rather uncomfortable to wear optical eye wear underneath the eye tracking glasses. Second, not every user is capable of performing tasks without their prescription glasses. Even though, we believe our set-up is a rather portable alternative to various common usability kits, we definitely see possibilities to improve in respect to the eye-tracking recordings.

Note that in this section the graphics showing the visual summary of the eye tracking data are mapped onto the screens of Pocket Code in German. In this section, we only discuss eye tracking data from the test sessions with Pocket Code, as we have a particular interest in evaluating its formula editor. Further, we did not analyze the data for Task 1 since it did not involve the formula editor.

We identified four AOI for the formula editor screen, namely (1) the brick itself on top of the screen, (2) the text field where the user enters the formula, (2) the numbers at the bottom left side and (4) the column containing the categories (see Figure 4.24). For each task a corresponding formula editor screens reference image was used to map the fixations to. On basis of this the heat maps were created aggregated over all participants. Whenever a user navigated into a category, we also mapped those fixations to the category button, since we were not particularly interested in how the user scans the category content itself but rather how the user examines the formula editor screen. This however, also lets us infer from the heat map categories in which the user spend a long time searching for the various formula elements.

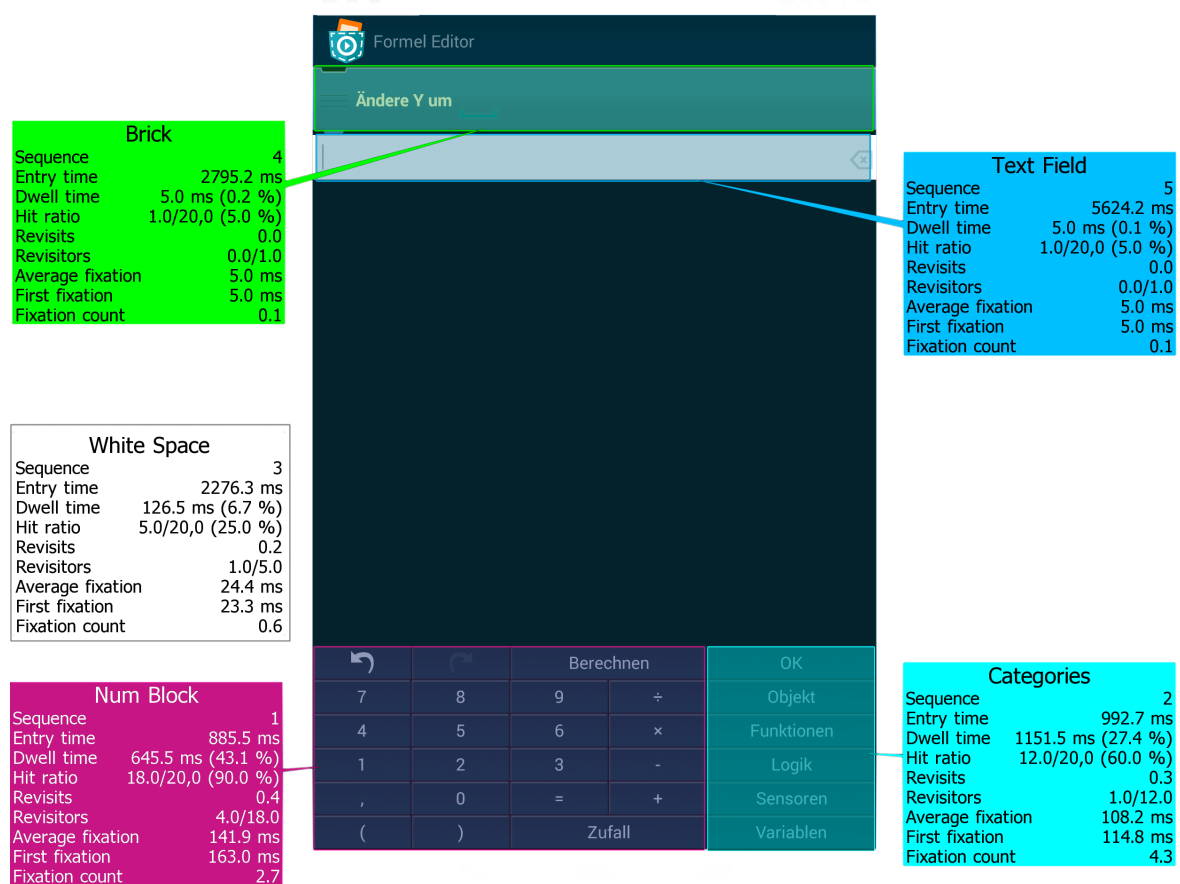


Figure 4.24: AOI for the formula editor for Task 3.

## Task 2

In this task the users were asked to replace the object attribute *position\_y* with *position\_x* in the script of the *Car* object. Thus, we wanted to evaluate how the test users scan the editor containing the `Set variable` brick, in order to change the assignment to another variable.

In this context the user first sees the formula editor within the test scenario, with exception of the tutorial. Figure 4.25a shows the heat map of this task, i.e. the average number of fixations for all participants mapped to a reference image. The red spots indicate a high number of fixations. As can be seen the text field itself received a majority of fixations as well as the *Object* category. However, we can observe that, further the other categories receive attention, which can be interpreted that the users were not sure where to find *position\_x*. The binning chart in Figure 4.25b shows that the attention on the text field decreases with time.

### Task 3

Task 3 was concerned with creating a formula using numerical operators as well as a random number generator. In this context we analyzed the eye tracking data to determine how the users examine the interface when asked to insert a random number generator. The task was to insert  $3 \times \text{random}(5,10)$ , thus we analyzed the data from the time the user has entered the multiplication operator to the time the user has actually found the `random` function. The random number generator has been positioned within the interface as a special function on the formula editor screen, but furthermore is represented within the *Math* category.

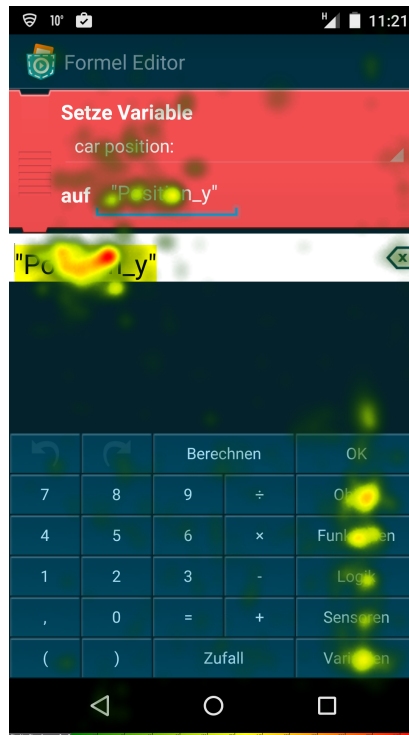
As Figure 4.26 shows the main fixation at the random number generator button in the middle of the bottom of the screen. As can be seen the second area with a large number of fixations is the *Variable* category. We believe that this is due to the fact that the users were not able to distinguish the random function from a variable.

### Task 4

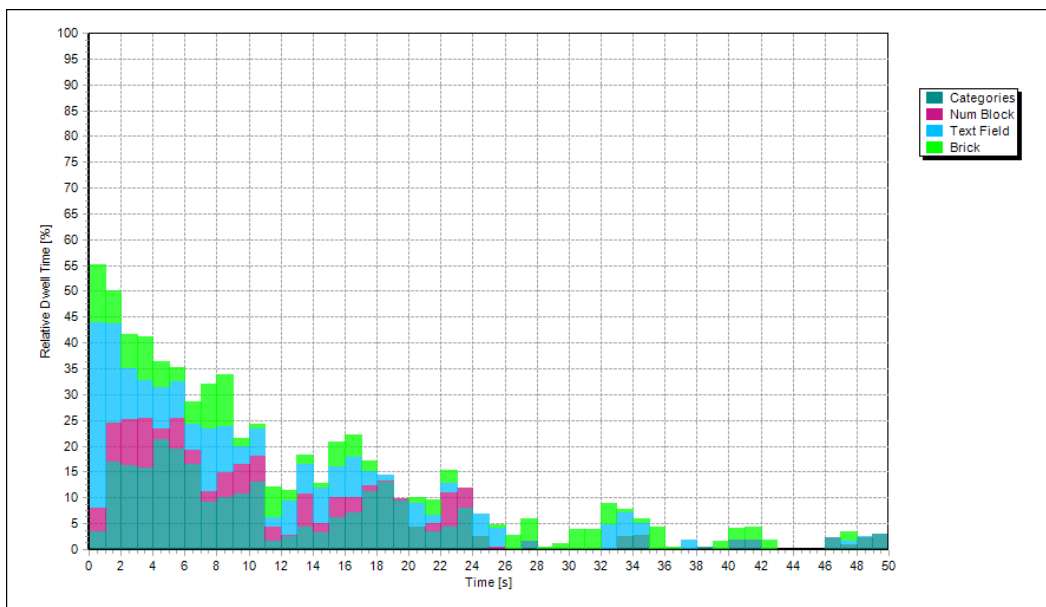
In the forth task, we asked the users to change parts of a formula by inserting a new sub formula consisting of a variable, numeric values as well as relational and logical operators. In particular, one step is to add a variable and afterwards the relational operator `>`, which is located within the *Logic* category. From Figure 4.27 we can see that the two main fixation points are on the one hand the text field and on the other hand the button for the *Logic* category.

We also examined how the users scan the screen for the logical *AND* operator. Interestingly, we can see from Figure 4.28a that even though *AND* is located in the same category as `>` there was no learning effect, as we can see that the heat map is more dispersed than for the `>`. From the binning chart in Figure 4.28b it is apparent that the participants spend most of the time within the categories searching for the logical operator.





(a) Heat map.



(b) Binning chart (1 second interval).

Figure 4.25: Eye tracking analysis of Task 2.

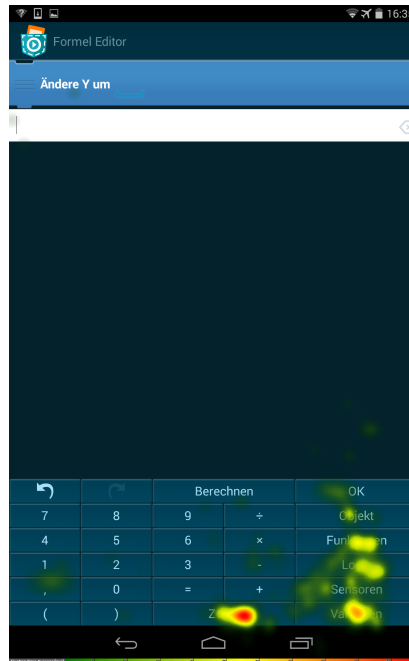


Figure 4.26: Heat map of Task 3 in the time frame after inserting  $\times$  till inserting the random number generator.

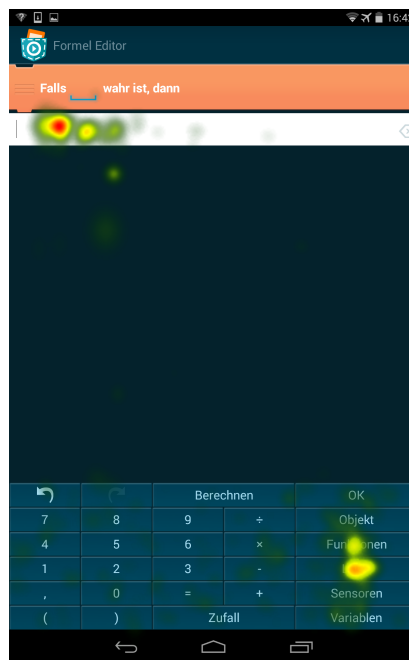
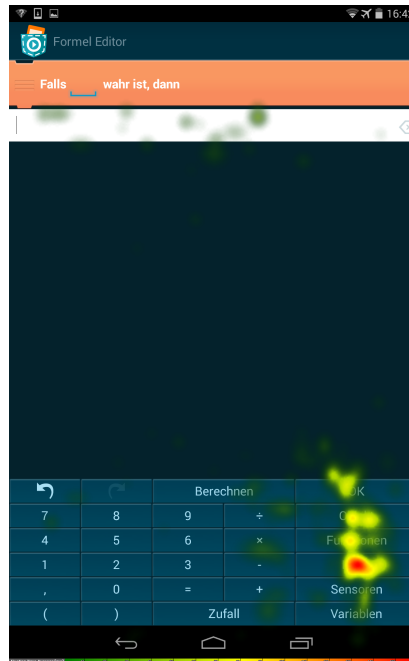
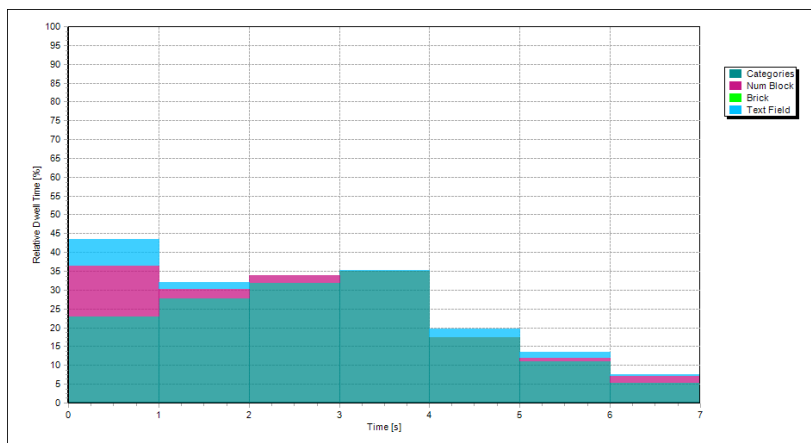


Figure 4.27: Heat map of Task 4 when users are scanning the screen for  $>$ .



(a) Heat map



(b) Binning chart (1 second interval)

Figure 4.28: Eye tracking analysis of Task 4 when users are scanning the screen for *AND*.

### 4.2.3.6 Issues and Interpretation

Overall, our results show strong evidence towards a greater usability in the context of formula manipulation of the hybrid approach as implemented in Pocket Code in comparison to a purely visual one used in Scratch. Figure 4.23b depicts the average single usability score (considering single task completion, time on task and perceived satisfaction ratings) as a percentage value of both applications (Pocket Code  $M=68.13\%$ ,  $SD=11.37\%$ ; Scratch  $M=54.62\%$ ,  $SD=11.05\%$ ;  $t=5.24, p=.00006$ ). Nevertheless, we observed several issues within either programming environment [62].

#### Scratch

##### 1. Input Error Minus/Random Number:

Description: Test users try to type the minus operator or “pick random(0 to 8)” using the keyboard rather than the subtraction brick or the random number operator brick (see Figure 4.29). Since it is possible to enter text within the fields in which bricks can be interconnected, the test users tried to enter the formulas this way.

No. Occurrences/No. Test Users: 26/14

Severity: 4

Proposed Solution: Staying consistent within the usage of operators would be useful. Thus, we propose to have the binary sign operator be a brick within the *Operators* category such as the logical **not**. Further, we think it would be better to have some sort syntax check there to see whether a textual input would make sense in this particular brick. Then it could be avoided to have users type entire formulas or operators.

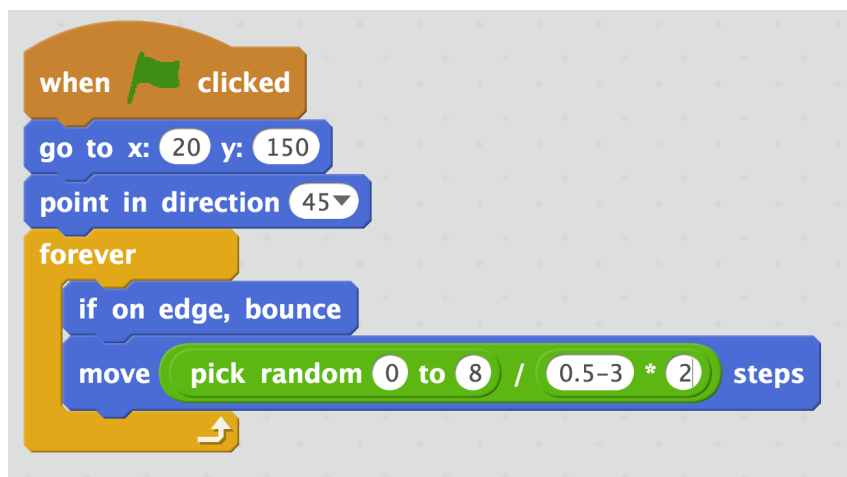


Figure 4.29: Adding a minus within a field.

**2. Operator Search:**

Description: Test users cannot find a specific operator or other brick. The usual strategy of the test users then is to go through all categories (even several times) until they locate the brick. The entire brick collection is rather large (see Figure 4.33).

No. Occurrences/No. Test Users: 22/18

Severity: 3

Proposed Solution: The coloring of the bricks should already provide a visual clue to where to find a particular type of brick. However, it seems to be not enough for beginners. A solution, which might increase the efficiency, would be to have bricks, which have been already used within the program several times, at a type of *recent bricks* category to have an easy access to frequently used bricks. Another possibility would be to hide some of the more advanced bricks in an expert mode to diminish the total number of bricks visible.

**3. Operator Position:**

Description: Test users place bricks at the wrong position within a formula by accident. This happens when the brick is dropped not at the right location above the brick. As can be seen in Figure 4.30 moving the brick slightly triggers different drop locations as indicated by the illumination.

No. Occurrences/No. Test Users: 12/10

Severity: 3

Proposed Solution: Even though there is an illumination of the field a brick will be dropped into, some of these areas are so close together that by moving the brick slightly an unintended drop location is selected.

**3. Automatic Formula Decomposition:**

Description: Test users place bricks at the wrong position within a formula by accident. In this case it can happen that another part of the formula is replaced, i.e. the new brick is placed at the position of the original brick, which in turn is positioned somewhere within the scripting area outside any brick block. The user then is not able to recreate the state before since the user forgot the formula and there is no undo.

No. Occurrences/No. Test Users: 7/6

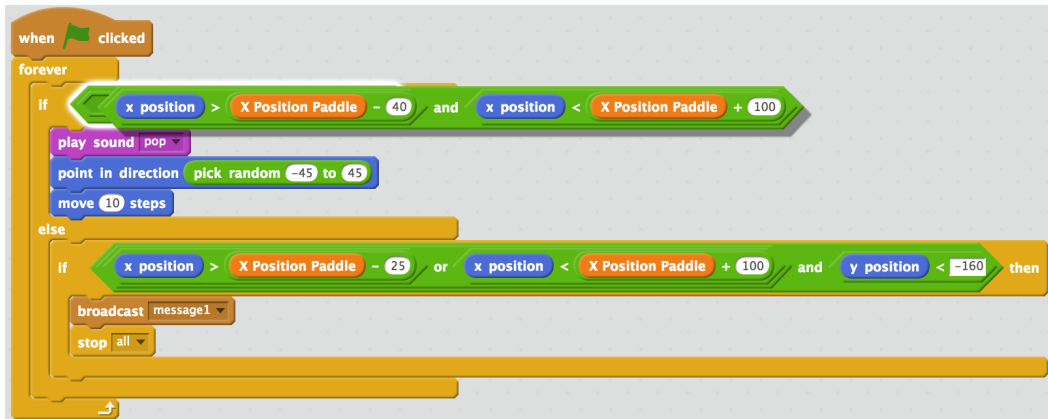
Severity: 3

Proposed Solution: Adding a simple undo/redo would be sufficient.

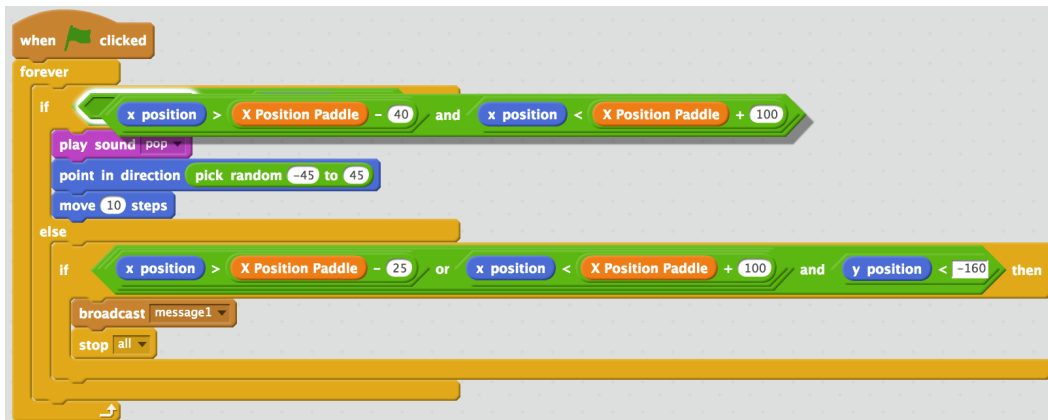
**4. Deleting formulas by accident:**

Description: Test users by accident delete a subformula or formula and are not able to recreate it from memory and there is no undo.

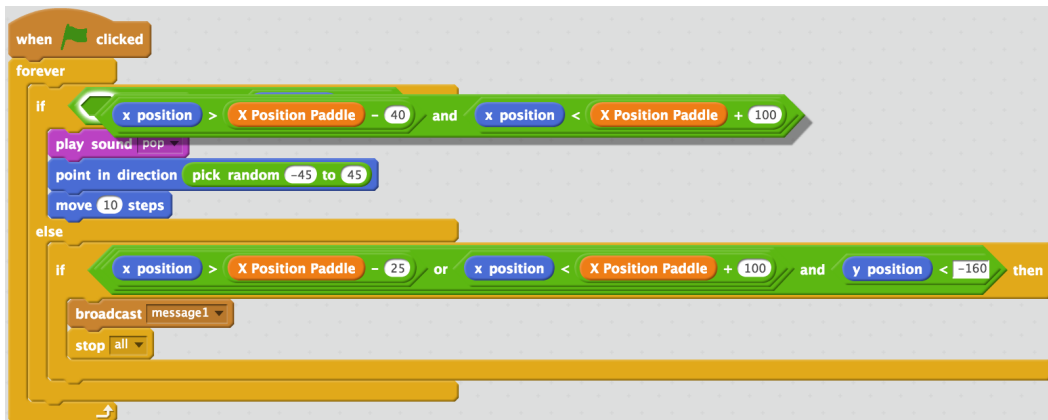
No. Occurrences/No. Test Users: 3/3



(a) Illumination of the outer formula field.



(b) Illumination of the middle formula field.



(c) Illumination of the most inner formula field.

Figure 4.30: Illumination indicating brick drop location.

Severity: 2

Proposed Solution: Adding a simple undo/redo would be sufficient.

#### 5. Misinterpretation Attribute:

Description: Test users interprets the object attribute *x-position* as a formula *x-position* and types it into the brick field.

No. Occurrences/No. Test Users: 1/1

Severity: 0

Proposed Solution: -

### Pocket Code

#### 1. Operator Search:

Description: Test users cannot find a specific operator/function/attribute. The usual strategy of the test users then is to go through all categories (even several times) until they locate the brick. The entire brick collection is rather larger (see Figure 4.34).

No. Occurrences/No. Test Users: 22/14

Severity: 4

Proposed Solution: Since there is no visual aid in where to find a certain type of function it would be helpful to maybe add a color scheme as featured in Scratch. Even though, we can see from the issues list of Scratch this is not necessarily the entire solution, we think that for the small screen it would be helpful to have some type visual distinction. Further, more appropriate category names could aid the user to locate the right category.

#### 2. Cursor Positioning:

Description: Test users had difficulties positioning the cursor at a specific location within the formula.

No. Occurrences/No. Test Users: 8/7

Severity: 4

Proposed Solution: Reimplementing a more precise cursor or adding arrow buttons to navigate through the formula would be sufficient.

#### 3. Button Size:

Description: Test users typed erroneous formula due to button size and closeness of the buttons.

No. Occurrences/No. Test Users: 3/3

Severity: 3

Proposed Solution: It will probably be necessary to redesign the formula editor button field at the bottom to feature larger buttons.

#### 4. Formula representation:

Description: Test users edited incorrect formula parts, as they were confused by the nested formula in Task 4. Due to the many parentheses and no visual indication of which parts form sub formulas, it can be difficult to distinguish depended formula fragments (see Figure 4.31).

No. Occurrences/No. Test Users: 3/3

Severity: 3

Proposed Solution: In many programming environments there is the possibility to select a parenthesis and the environment visually emphasizes the corresponding opening or closing bracket. In an additional step to have a more structured view of the inserted formula would allow the users to have a better overview of the formula.

```
(( position_y > - 180 ) AND ( position_y < 180 ) ) OR  
( ( "car position" > - 150 ) AND ( "car position" <  
150 ) )
```




Figure 4.31: Nested formula of Task 4.

#### 5. Editing formulas outside formula editor:

Description: Test user tries to insert the formula within the brick inside the script view.

No. Occurrences/No. Test Users: 5/5

Severity: 1

Proposed Solution: -

#### 6. Delete/Undo button:

Description: Test user confuses the *Delete* button with the *Undo* button (see Figure 4.32).

No. Occurrences/No. Test Users: 1/1

Severity: 1

Proposed Solution: -



## Interpretation

In both applications the participants encountered obstacles locating operators necessary for composing the given formulas. In the case of Scratch, even though the blocks are color coded, many users did not know in which category to look for certain blocks. The participants tended to search through every category—even several times—until they discovered the block they were looking for. In Pocket Code a similar problem could be observed. We believe that the naming of categories in Pocket Code is not suitable, as adolescent users often seem not to be familiar with terms such as *Logic*. We believe that a revised naming in terms suitable for the target audience is required. However, due to the fact that in Pocket Code the categories of the formula editor are separated from the general block categories, the search space is diminished in comparison to Scratch. It may be reasonable to assume that some of the time on task differences originate from this. In Figure 4.34 we show the various operators of the formula editor and the collection of bricks available in Scratch (Figure 4.33) to depict the search space the participants had to cover [62].

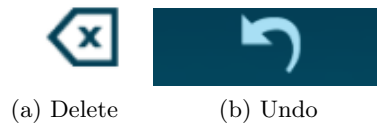


Figure 4.32: Delete/Undo button.

However, Scratch’s most prominent weakness within this study, has been the block construction of formulas. Building nested formulas in Scratch is especially difficult, as it requires the user to plan the composition steps a-priori, since the nesting level implies the groupings of operators. Most of the teenagers we tested tended to not read through the entire formula before working on the solution. Hence, they tried to compose the formula from left to right and eventually had to edit, rearrange, delete, and add blocks to create a formula with the required grouping. Generally, the participants faced obstacles with the blocks in Scratch, as they frequently dropped them at the incorrect position, destroying already existing formulas. The viscosity of Scratch accounts for considerable correction as well as manipulation time and frustrated the participants. Further, in order to pick up a brick nested within a structure it is necessary to exactly click the operators name. Often the test participants were imprecise in targeting the brick and by accident picked up an incorrect part of the formula. An easily accessible “Undo/Redo” feature could diminish the time used for corrections [62].

Some participants found that it is possible to textually add the minus symbol into certain fields, which however is not interpreted as the arithmetic operator but instead as the binary operator to denote the sign. This created confusion as the participants did not seem to understand its purpose. These might be some of the reasons why Scratch was perceived as more difficult by the participants. A solution would be to create a block for the sign, which would be more in line with the general methodology used within Scratch [62].

Due to the fact that Pocket Code uses parentheses to denoted grouping, the participants were able to construct the formulas from left to right. It seemed that using a virtual pad was easier as the teenage participants were more familiar with this way of editing and creating



Figure 4.33: Scratch brick overview.

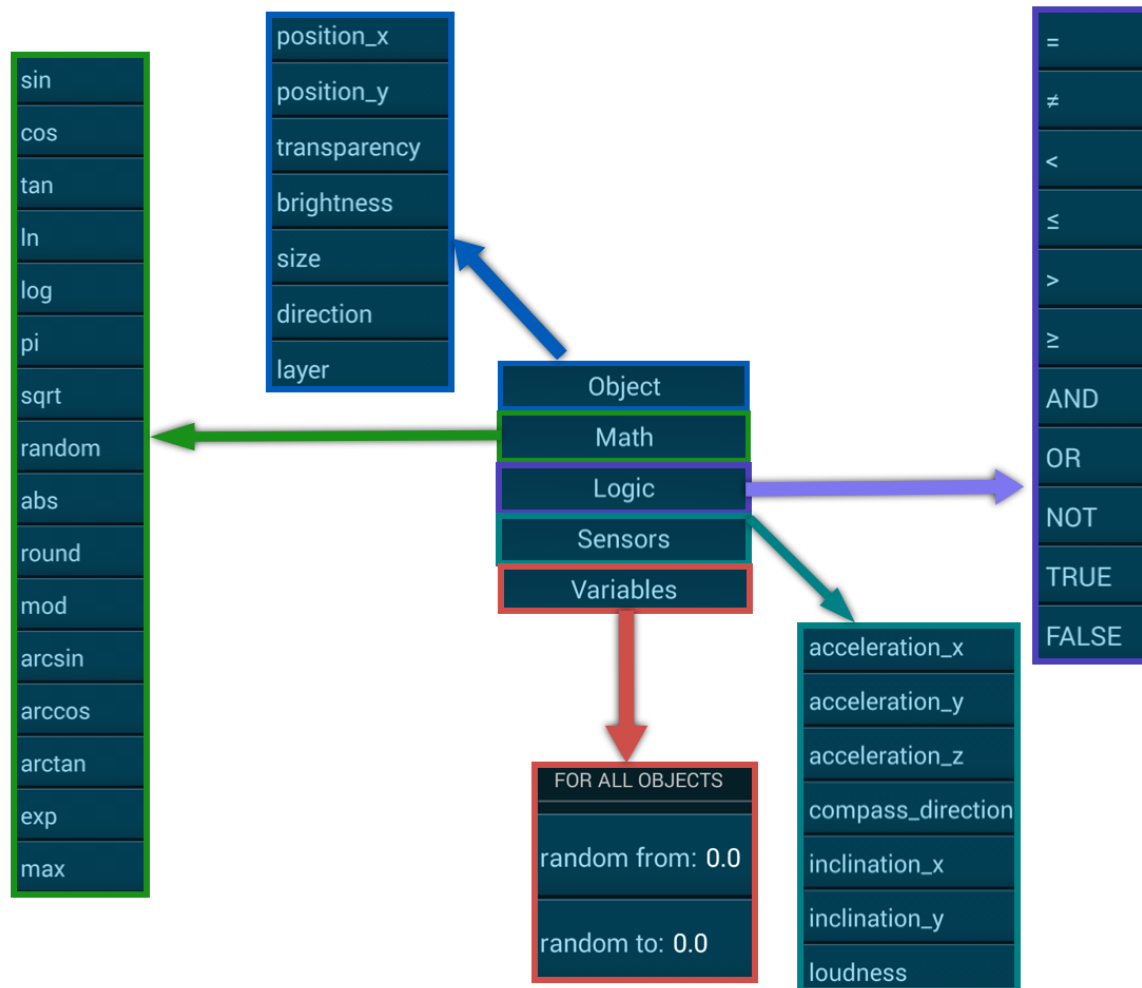


Figure 4.34: Pocket Code categories.

formulas. In Pocket Code participants had difficulties using the cursor in the formula editor, resulting from its imprecise positioning. A reimplementaion of the cursor is essential, as its impreciseness impedes its general purpose, which is to be able to easily edit any position within the formula. Additionally, it appears that the general interface layout should be improved upon, since button sizes and proximity yielded typing errors [62].

Regarding the pocket calculator metaphor, we would also like to mention the participant who has been disregarded from our results due to not being familiar with pocket calculators. She had several difficulties in working with the formula editor, especially as she was not familiar with the vocabulary used (e.g., Variables). We cannot generalize on this; however, we assume that our metaphor would not necessarily facilitate formula composition among younger users who have not used pocket calculators before [62].

In addition, the perceived satisfaction reported by the users should be viewed with skepticism. It is well known that test participants tend to describe their experiences more pleasant and products more usable than the recorded data actually implies. In particular, this effect is increased with younger participants such as children and teenagers.

#### 4.2.3.7 Validity

The validity of this study can be questioned based on the relatively small sample size [51]. Moreover, the way we presented the tasks to the users most likely influenced their performance in favor of Pocket Code. Due to the nesting of formulas we were in need of a representation which shows the grouping of the statements. While it would have been possible to change the formula representation for Scratch by displaying the formulas in their block representations, we believe that the blocks are difficult to visually separate from each another (see Figure 4.35). Hence, we chose a textual formula representation to convey the tasks to the test users, which likely primed them for a textual representation of formulas. A possible solution could be, instead of specifying explicit formulas, to indicate a general problem statement in natural language, and to let the test users come up with the needed mathematical expression on their own. However, due to the way mathematics and natural sciences are taught in middle and high schools, it seems plausible that users would nevertheless have formulated the mathematical expression in textual form [62]. Further, we have disregarded the attribute of *Cognitive Load* as suggested by the PACMAD [44] usability model, which however is common in mobile usability testing [55].

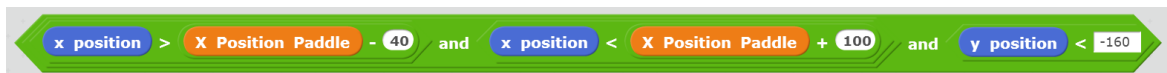


Figure 4.35: Nested formula in Scratch.

### 4.3 Recommendations

We have conducted a heuristic evaluation of the formula editor in Pocket Code as well as a summative usability study comparing formula manipulation in Pocket Code to Scratch. While the former yielded a greater number of issues, we could only confirm some of them in our empirical experiment. There are several reasons for this mismatch; first, within the heuristic evaluation reviewers are asked to examine the entire interface several times, whereas the tasks of the test covered only parts of the formula editor. Second, the usability study focused on the mechanics of formula manipulation and composition, i.e. did not incorporate tasks to evaluate the comprehensibility. An inspection on the other hand takes into account general guidelines and a broader view of usability, therefore, the evaluators reported issues they would anticipate in various aspects of using the formula editor.

#### Naming Conventions

Both methods agreed that identifying the corresponding category to a single operator or function is difficult in the current interface design. In the heuristic evaluation we referred to this in an issue stating that the nomenclature is problematic as it does not truly reflect the language used by children or teenagers. For example, *Logic* has a different meaning in natural language in comparison to the programmatic concept, thus, we assume that the target audience has difficulties in understanding these terms. Recalling the usability test, the main issue causing both applications to be inefficient is the fact that the participants spent considerable amounts of time searching for operators or functions. Even though the participants were asked to perform several tasks, we could not observe an improvement in their ability to locate expression parts efficiently over the course of the test. Thus, as Pocket Code's category naming conventions are inspired by Scratch, it seems that the heuristic of speaking the users language is violated in both.

Pocket Code's formula editor has been devised with common pocket calculators in mind. Yet, Markopoulos and Bekker [76] utilize the example of a virtual calculator as inspired by an already badly designed physical artifact. Pocket calculators often feature an excessive amount of modes, functions and inappropriate labels. Function names such as *sqrt*, *mod* or *abs* are in line with the calculator metaphor, yet, are not ideal within the context of Pocket Code as a programming environment for children and teenagers. Thus, strictly enforcing the pocket calculator metaphor might not be the correct approach for usable formula manipulation.

Due to disregarding formula comprehension within our study design, but rather evaluating the mechanics, we could not observe a problem with Boolean expressions. However, studies have shown that they do pose a difficult programming construct, as *AND* and *OR* are used differently in linguistic usage than in programming languages [101]. There might be the need to find more suitable expressions to describe logical operators in terms familiar to the target audience. Stefik and Siebert [132] investigated how intuitive terminology in general purpose programming languages is by providing descriptions of concepts to users and asking them to rate how well common terms relate to these descriptions. For example for the relational operator the description states: "Suppose you wanted to check whether something named *x* had the same value as something named *y*. Rate each expression according to how well you

think it represents checking whether  $x$  and  $y$  have the same value.” Figure 4.36 shows the results for various concepts.

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Boolean Equals	Non-programmer	$x = y$ (8.15, 3.10), $x$ is $y$ (7.87, 2.84), $x == y$ (7.04, 3.29)	$x ? y$ (3.37, 2.88), $x ? : (y 3.15, 2.75)$ , $x > < y$ (2.90, 2.72)
	Programmer	$x == y$ (8.64, 2.20), $x$ isEqual: $y$ (7.17, 2.66), $x$ is $y$ (6.81, 3.17)	$x < - * y$ (3.14, 2.84), $x < > y$ (2.12, 2.52), $x > < y$ (2.02, 2.36)
Not Equal To	Non-programmer	$x$ unequal $y$ (6.32, 2.96), $x$ not= $y$ (5.84, 3.16), $x = \backslash = y$ (4.16, 3.32)	$x / y$ (2.74, 2.91), $x == y$ (2.57, 3.11), $x = y$ (2.43, 3.10)
	Programmer	$x != y$ (7.61, 2.77), $x$ not= $y$ (6.54, 2.87), $x$ unequal $y$ (6.45, 3.12)	$x / y$ (1.92, 2.64), $x = y$ (1.90, 2.78), $x == y$ (1.85, 2.67)
And	Non-programmer	$x$ and $y$ (8.29, 3.09), $x \& y$ (8.15, 2.89), $x \&\& y$ (6.61, 3.03)	only $x$ or $y$ (1.98, 2.64), either $x$ or $y$ (1.89, 2.84), $x$ nor $y$ (1.56, 2.55)
	Programmer	$x$ and $y$ (8.85, 2.42), $x \& y$ (8.65, 2.38), $x \&\& y$ (7.99, 2.80)	either $x$ or $y$ (1.65, 2.48), only $x$ or $y$ (1.62, 2.45), $x$ nor $y$ (1.46, 2.37)
Or	Non-programmer	$x$ or $y$ (6.28, 3.53), either $x$ or $y$ (5.95, 3.37), $x$ and $y$ (5.41, 3.45)	$x v y$ (3.94, 3.17), $x ^ y$ (3.54, 2.79), $x$ nor $y$ (2.13, 2.86)
	Programmer	$x$ or $y$ (7.60, 3.17), either $x$ or $y$ (6.75, 3.40), $x    y$ (5.93, 3.73)	$x ^ y$ (3.49, 3.47), $x$ exclusive and $y$ (3.38, 3.54), $x$ nor $y$ (1.67, 2.44)
Xor	Non-programmer	either $x$ or $y$ (6.29, 3.42), $x$ or $y$ (6.24, 3.41), only $x$ or only $y$ (6.20, 3.53)	$x$ and $y$ (3.15, 3.53), $x$ exclusive and $y$ (2.99, 3.00), $x$ nor $y$ (2.91, 3.06)
	Programmer	only $x$ or only $y$ (7.86, 3.21), either $x$ or $y$ (7.20, 3.24), $x$ xor $y$ (6.33, 3.42)	$x \& y$ (1.82, 2.70), $x \&\& y$ (1.58, 2.54), $x$ and $y$ (1.46, 2.69)

Figure 4.36: Expression choices for Boolean operators [132].

Conducting a similar study within the context of the Catrobat project, asking teenagers and children to rate operators and even category names, might give a better understanding of possibilities to rename certain parts of the formula editor.

### Expert Mode

Furthermore, we believe that an expert mode would be beneficial in order to hide some of the functionality from novice users in order to have an easy to learn yet highly efficient system [138]. This would allow experts to still be able to incorporate advanced calculation in their programs, while at the same time beginners are not overwhelmed.

Brennan and Resnick [17] exploited Scrape [144], a tool analyzing programming blocks in Scratch, to assess computational thinking among Scratch users. For example, in Figure 4.37 the projects of an experienced and a novice Scratch user are visualized by projects (columns) and bricks (rows). The darker the color of the cell, the more frequently the brick has been used within the particular program, while the last column represents unused blocks. From the illustration it is apparent that the novice user only takes advantage of a small subset of bricks.

In a similar manner the functions or operators essential for novices in Pocket Code could be determined by analyzing uploaded projects. Based on the experience level of the user and their usage of formula elements a suitable subset for a novice user mode could be computed.

Another question in this regard would be whether the provided bricks in Pocket Code should be extended to contain more blocks from Scratch. In Scratch, for example, collisions can be

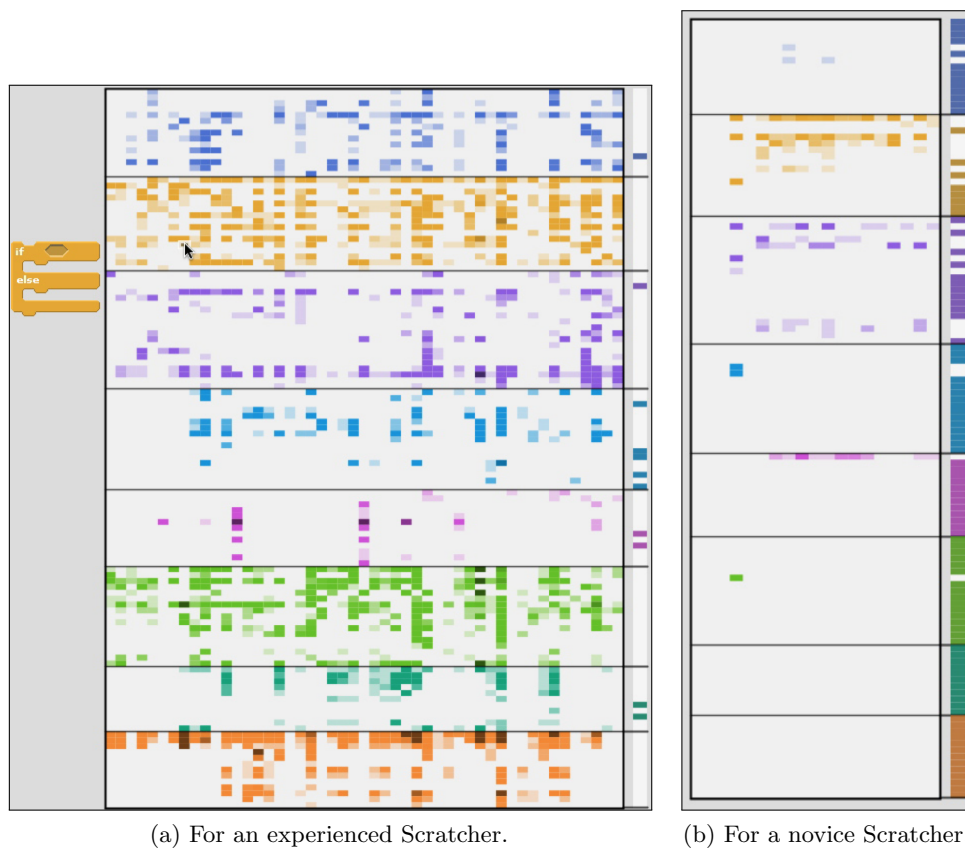


Figure 4.37: Scrape user analysis visualization [17].

detected using a designated brick. This brick is not available in Pocket Code, however, it would facilitate programming for beginners.

### Button Size

The pointing accuracy of a finger cannot compete with the precision of a computer mouse, thus targets need to have at least a certain width and height. Increasing the button sizes is probably impossible since smaller devices from up to three inches should be supported. In those cases the entire formula editor would not fit on the screen. However, for larger devices such as tablets the number pad and category buttons can be increased as there is enough blank space to the text field at the top. In particular, when analyzing the footage we can observe that the buttons height is the main issue on the seven inch devices rather than its width. For smaller devices, we would suggest to provide an editor similar to TouchDevelop, i.e. have more nesting levels to reach an operator or operand. Instead of having all categories on the initial formula editor screen, a button *Categories* could be added leading to an intermediate screen listing all category buttons, e.g. *Logic*, *Object*, etc. Another possibility would be to have designated tabs for the number pad and categories, such as in the the calculator application Tydlig<sup>10</sup> (see Figure 4.38).

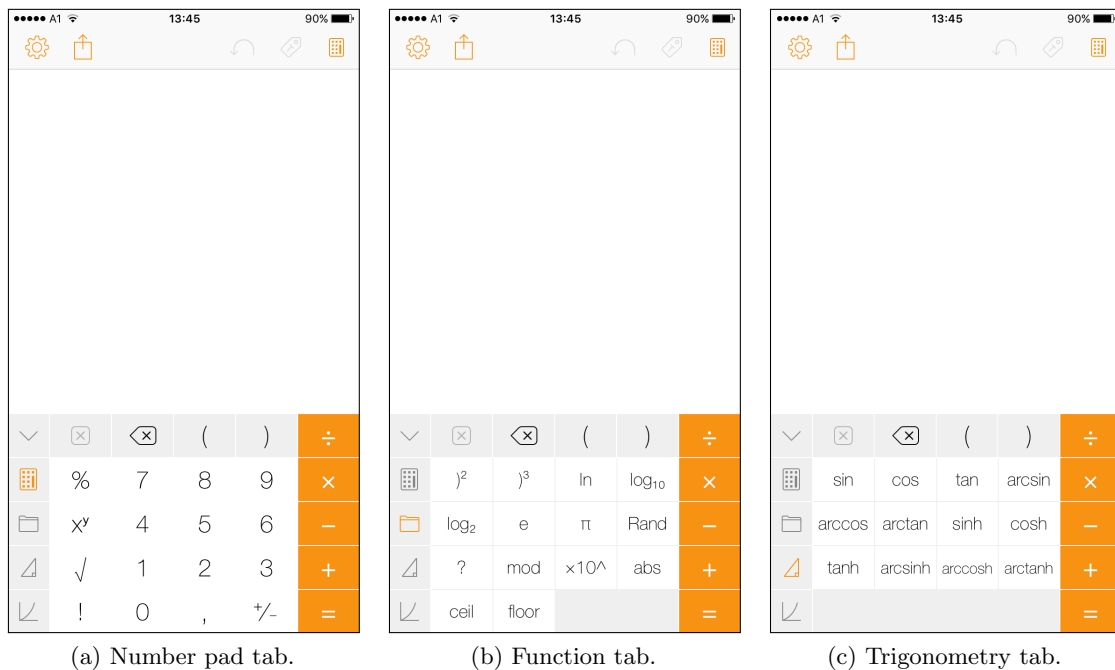


Figure 4.38: Tydlig interface featuring several tabs.

### Cursor

The cursor implementation of the formula editor unfortunately does not meet its objective, which is to provide an effortless way to pinpoint an exact formula position to facilitate

<sup>10</sup><http://tydligapp.com/>



manipulation and thus reduce viscosity. Viscosity in this context refers to the ease of changing parts of a formula while maintaining the rest of it. There are generally two possibilities; ideally the cursor is reimplemented to be more accurate. Otherwise designated buttons to move the cursor left and right could be integrated.

### Copy Sub Formulas

Another issue in regard to viscosity, is the absence of a copy function for sub formulas, which would ease the creation of formulas with similar parts, such as collision detection. Furthermore, the implementation of the editor requires premature commitment in the sense that it is not possible to span, for example, a mathematical function over an entire already existing formula. However, this is necessary, as formula development is often an interleaved process of creation and debugging. Thus, allowing to adapt formulas at various positions is essential.

### Visual Clues

There is no visual separation between various building blocks of a formula, i.e the text field does not provide an additional indication besides spaces of which Strings form an expression or a formula element, e.g. *Object* attribute or *Sensor* value. Even though a grouping through parentheses is available, there is no visual automatic matching to allow the user to determine missing open and close parentheses or highlight the sub formula encompassed by a pair of bracket. This is, however, very common in programming environments for textual languages. Since the formula editor provides a textual representation, adding well proven techniques of textual programming environments would be useful.

Another common procedure in programming environments of textual languages is to have parts of the code, e.g. keywords, identifiers, constants etc., highlighted in various colors to allow to determine functionality quickly. Thus, similar visual clues might be practical for the text field. Further, we believe that Pocket Code in general might be missing a more structured UI, where the use of addition color might function as a visual separator and connector of functionality, i.e. the number pad features a different color than the *Compute* and *Ok* button.

### Basic Usability Issues

As mentioned the heuristic evaluation covers a vaster range of usability concerns, thus we aggregated the results not discussed yet:

- Provide feedback for large number computation
- Consistent error handling
- Basic functionality in accordance to the Android guidelines (e.g. cursor design and text selection).
- Computing meaningful values using the *Compute* button, i.e. *Compute* of 0.54 should not lead to 0.5400000214576721. While an approximation is useful for internal representation, the user should not be made aware of this.

- Discussion of useful coercions (e.g. inequalities or `Wait ___ seconds brick`) and possible type checking warnings.
- Remove “=” from the formula editor view, as it is not an assignment, but a logical operator. Within the context it is presented, it rather suggests to provide the same functionality as *Compute*.

### New Formula Editor

Since the usability study, the formula editor has been redesigned, considering a few of the findings, e.g. Android guideline conform text highlighting. As can be seen in Figure 4.39 the ordering of the buttons at the bottom of the screen has changed and it is possible to create Strings (*Abc*) as well as Lists (contained in the *Data* category).

While some of the issues have been solved so far, e.g. error messages have improved, i.e. *Syntax error* has changed to *Formula is not valid* (see Figure 4.40a), there are still issues needing attendance, such as the internal and external representation of numbers. In case of the error handling, there are some inconsistencies. For example, if the user creates a formula using as operands reals as well as Strings, when pressing compute an error is shown as depicted in Figure 4.40b. However, by leaving the editor and saving the formula, the user can continue programming (see Figure 4.40c). In case we assign this value to a variable which we display during the execution of the game, we can observe that during execution the variable has the value *NaN*. Ideally, the user cannot save such an expression in the first place but receives a type checking error. Further, if the internal representation should be hidden from the user in the formula editor as for example in Figure 4.40d, the same holds for the execution area.

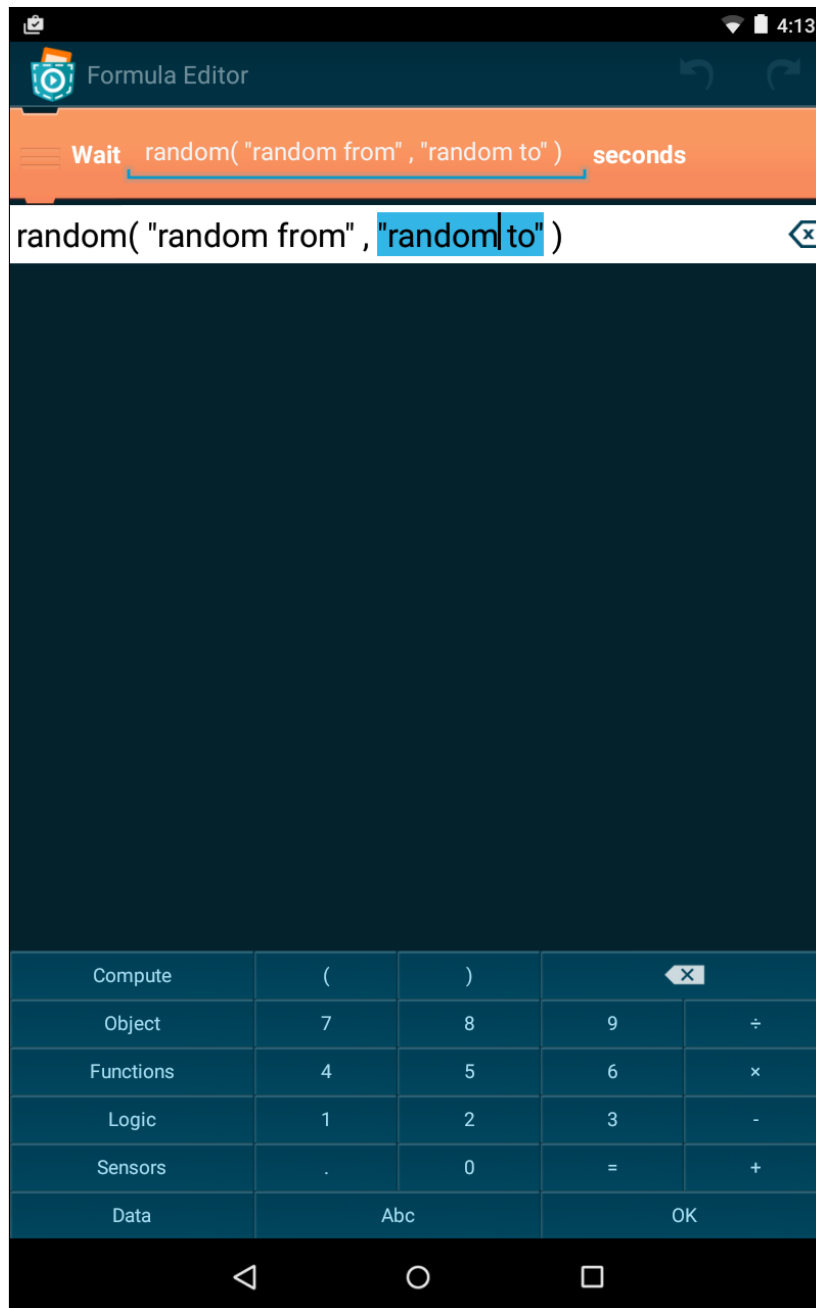
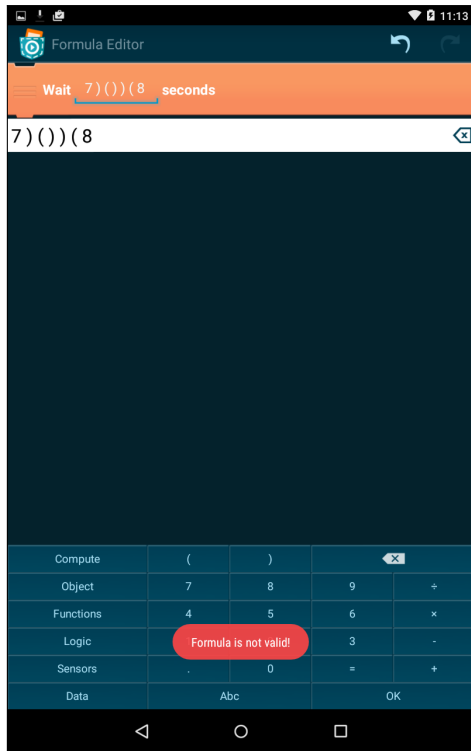
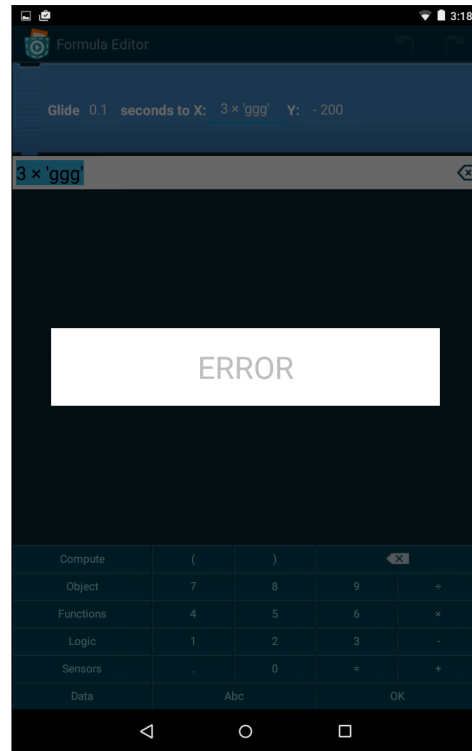


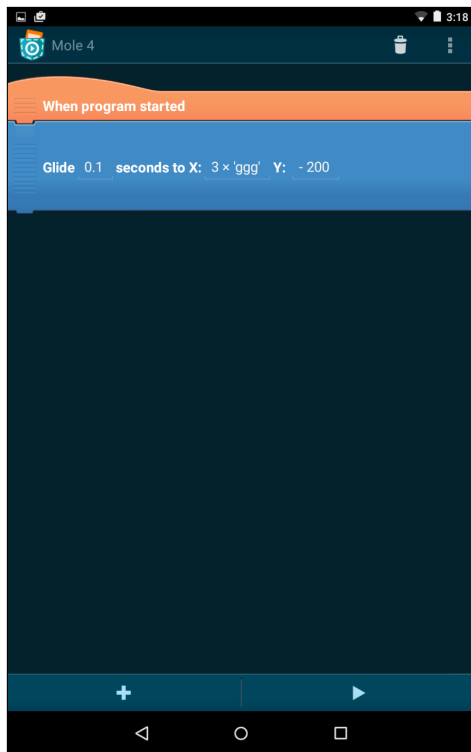
Figure 4.39: Current formula editor with adapted text highlighting.



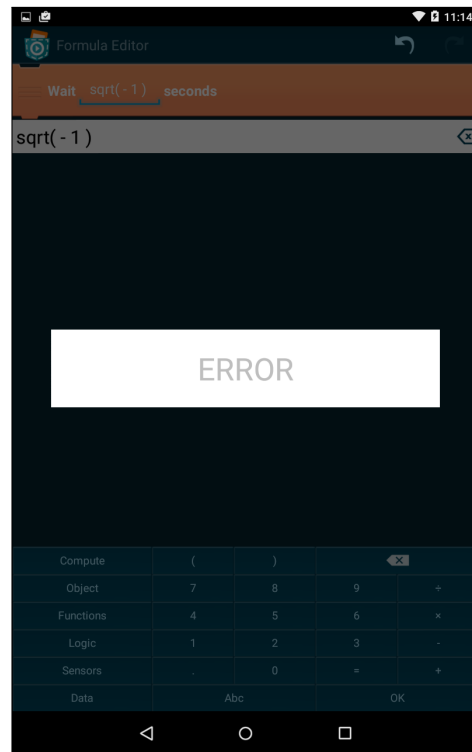
(a) Syntax error.



(b) Type checking error: multiplication of integer and String using *Compute*.



(c) Type checking error: script view.



(d) Error for Not a Number

Figure 4.40: Current formula editor.

## 5 Conclusion and Future Work

---

*“Learning to code is the single best thing anyone can do to get the most out of the amazing future in front of us.”*<sup>1</sup>

In this thesis, we have studied formula composition and manipulation methods in programming languages of educational environments. In particular, we have investigated Pocket Code, a mobile programming applications for children and teenagers, which has been inspired by the visual block-based programming environment Scratch. Due to the restricted screen size of hand held devices, it is infeasible to represent formulas, particularly nested ones, in a brick structure. Thus, the designers and usability experts of Pocket Code have developed a formula editor inspired by traditional pocket calculators familiar to the target audience from mathematic classes. Comprising object attributes, arithmetic, relational and logical operators as well as variables and sensor measurements, the formula editor represents a powerful tool within Pocket Code and an essential part in order to develop meaningful programs. The editor uses a hybrid formula composition scheme, i.e. formulas are created visually by tapping buttons, while they are represented in a textual manner. This approach has several advantages; on the one hand, it facilitates formula input by not relying entirely on a virtual keyboard and at the same time diminishes the possibility to create syntactically incorrect formulas. On the other hand, displaying the formulas textually is a space saving option necessary for mobile devices.

To assess its usability we exploited method triangulation in two ways; first, we utilized several evaluation techniques, i.e. heuristic inspection and an empirical usability study; second, within the summative test we collected quantitative as well as qualitative data to create a holistic overview of usability by examining various attributes. For the heuristic evaluation we employed five evaluators to review the formula editor interface in regard to a set of heuristics designed for mobile UIs. Subsequently, our empirical experiment compared formula composition in Pocket Code to Scratch, as Scratch is the state of the art educational programming environment for a younger audience. We collected a set of usability issues from both assessments and besides some easy to fix issues, such as consistent error handling, we found that even though in comparison to Scratch the formula manipulation in Pocket Code seems easier, there are still some problems inherent to the user interface. Particularly, participants had difficulties knowing where to locate functions, operators, or operands. It seems that the terminology within the editor is not ideal and visual clues are missing. Thus, we see the main area to focus on in the future is to explore remedies for this specific concern.

---

<sup>1</sup> <https://code.org/quotes>

Whether the recommendations we proposed in the previous section should be followed, generally boils down to the decision whether Pocket Code should be a novice programming language—empowering the user to create animations and games, or an education system, teaching beginners programming concepts and preparing them for more advanced textual languages. The aim of the former suggests to create an easy to use interface and a small set of operators and functions which are essential to develop programs. Further, this might include the introduction of additional bricks, e.g. a collision brick, to trade off the formula editor’s power for more easily accessible functionality in the form of blocks. The latter, however, implies that many programming concepts, such as Boolean expressions, should remain in Pocket Code as in other general purpose programming languages. Which in turn entails that certain difficult aspects have to be simplified by supplying additional support in the form of tutorials or help texts.

This thesis concentrates on the mechanics of formula development, which constitutes the basis for efficient and effective programming within Pocket Code. The aspect of formula understanding has only been raised to a limited extent within the heuristic evaluation. Nevertheless, the comprehension of formulas and the process of formulating a goal and achieving it in a programmatic way via the formula editor has yet to be investigated, constituting an obvious continuation of the research we have presented. We believe that the research in this area has to incorporate empirical results to truly uncover usability concerns. To examine whether users understand formulas we would propose two parts. First, a set of tasks in which the test participant is asked to extend parts of a program, i.e. there are already existing formulas which have to be interpreted and adapted as well as the need to compose new formulas. The description of these assignments, in contrast to the presented study, would only describe the goal to reach and not how to accomplish it. For example, the user could program a simple collision between two objects without any additional help on how to perform this task. In the second part of the test, the users would be asked to solve, for example, a mathematical problem within the context of a Pocket Code program, where they are allowed to use the Internet to search for suitable formulas. This type of study could be conducted as a formal experiment with more experienced Pocket Code users, or even within a more informal setting organized as workshop collecting observations.

An additional validation concern of the summative study is the rather small sample size and the study design incorporating a textual formula within the task description. A possible solution would be to give a general task description and ask the users to develop the necessary formulas on their own. This type of task requires a certain level of programming experience to be able to construct a solution, hence it is not entirely applicable to studies involving novices. Furthermore, we compared a desktop application to a mobile programming environment, meaning that depending on the experiences the participants had so far they could be biased towards a device. Developing a testing environment along the lines of McIver [79], hiding the environments completely and solely providing the basic functionality to compose and edit formulas, would increase the internal validity. Otherwise a comparison of Pocket Code to the iOS application Tickle<sup>2</sup> would be an interesting experiment, since both are mobile programming applications and Tickle uses a purely visual formula manipulation.

---

<sup>2</sup> <https://tickleapp.com/> (accessed 2016-02-26)

- AOI** Area of Interest
- ASQ** After-scenario Questionnaire
- EUP** End user programming
- HANDS** Human-centered Advances for the Novice Development of Software
- HCI** Human-computer Interaction
- IDE** Integrated Development Environment
- LCD** Learner-centered Design
- PACMAD** People At the Centre of Mobile Application Development
- SEQ** Single Ease Question
- SMEQ** Subjective Mental Effort Question
- STEM** Science, Technology, Engineering, and Mathematics
- SUM** Single Usability Metric
- SUS** System Usability Scale
- TILT** Tools-Interfaces-Learner's Needs-Tasks Model
- UCD** User-centered Design
- UI** User Interface
- UX** User Experience
- VPL** Visual Programming Languages

# Bibliography

---

- [1] ISO 9241-210:2010 ergonomics of human-system interaction – part 210: Human-centred design for interactive systems.
- [2] Usability für Kids: Ein Handbuch zur ergonomischen Gestaltung von Software und Websites für Kinder, author=Liebal, J and Exner, M, journal=Wiesbaden: Vieweg+Teubner Verlag, year=2011.
- [3] C. Abras, D. Maloney-Krichmar, and J. Preece. User-centered design. *Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications*, 37(4):445–456, 2004.
- [4] W. Albert and T. Tullis. *Measuring the user experience: collecting, analyzing, and presenting usability metrics*. Newnes, 2013.
- [5] A. Alsumait and A. Al-Osaimi. Usability heuristics evaluation for child e-learning applications. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, pages 425–430. ACM, 2009.
- [6] L. Anthony, Q. Brown, J. Nias, B. Tate, and S. Mohan. Interaction and recognition challenges in interpreting children’s touch and gesture input on mobile devices. In *Proceedings of the 2012 ACM international conference on Interactive tabletops and surfaces*, pages 225–234. ACM, 2012.
- [7] A. Asamoah. Should we be using visual programming languages like Alice to teach programming? 2006.
- [8] W. Barendregt and M. M. Bekker. Children may expect drag-and-drop instead of point-and-click. In *CHI’11 Extended Abstracts on Human Factors in Computing Systems*, pages 1297–1302. ACM, 2011.
- [9] C. M. Barnum. *Usability testing essentials: ready, set... test!* Elsevier, 2010.
- [10] E. Beck, M. Christiansen, J. Kjeldskov, N. Kolbe, and J. Stage. Experimental evaluation of techniques for usability testing of mobile systems in a laboratory setting. 2003.
- [11] A. Begel. Logoblocks: A graphical programming language for interacting with the world. *Electrical Engineering and Computer Science Department, MIT, Boston, MA*, 1996.
- [12] J. R. Bergstrom and A. Schall. *Eye tracking in user experience design*. Elsevier, 2014.
- [13] A. F. Blackwell. Metacognitive theories of visual programming: What do we think we are doing? In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 240–246. IEEE, 1996.



- 
- [14] A. F. Blackwell. The reification of metaphor as a design tool. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 13(4):490–530, 2006.
- [15] T. Booth and S. Stumpf. End-user experiences of visual and textual programming environments for Arduino. In *End-User Development*, pages 25–39. Springer, 2013.
- [16] M. Boshernitsan and M. S. Downes. *Visual programming languages: A survey*. Citeseer, 2004.
- [17] K. Brennan and M. Resnick. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*, 2012.
- [18] J. Brooke. SUS—a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.
- [19] N. C. Brown, M. Kölling, and A. Altadmri. Position paper: Lack of keyboard support cripples block-based programming. *To appear in Blocks and Beyond*, 100, 2015.
- [20] Q. Brown and L. Anthony. Toward comparing the touchscreen interaction patterns of kids and adults. In *ACM SIGCHI EIST Workshop 2012*, 2012.
- [21] A. Bruckman, A. Bandlow, and A. Forte. HCI for kids, 2002.
- [22] M. Burnett, J. Atwood, R. Walpole Djang, J. Reichwein, H. Gottfried, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of functional programming*, 11(02):155–206, 2001.
- [23] A. Cockburn and A. Bryant. Leogo: An equal opportunity user interface for programming. *Journal of Visual Languages & Computing*, 8(5):601–619, 1997.
- [24] G. Cockton, A. Woolrych, D. Lavery, A. Sears, J. Jacko, I. Tsuchiya, and G. Grandy. Inspection based evaluations. 2008.
- [25] M. . Company. Transforming learning through medication, 2012.
- [26] M. J. Conway. Alice: easy-to-learn 3D scripting for novices. 1997.
- [27] S. Cooper, W. Dann, and R. Pausch. Alice: a 3-D tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116. Consortium for Computing Sciences in Colleges, 2000.
- [28] A. Donker and P. Markopoulos. A comparison of think-aloud, questionnaires and interviews for testing usability with children. In *People and Computers XVI-Memorable Yet Invisible*, pages 305–316. Springer, 2002.
- [29] A. Druin. The role of children in the design of new technology. *Behaviour and information technology*, 21(1):1–25, 2002.
- [30] H. Dwyer, C. Hill, A. Hansen, A. Iveland, D. Franklin, and D. Harlow. Fourth grade students reading block-based programs: Predictions, visual cues, and affordances. In *Proceedings of the eleventh annual International Conference on International Computing Education Research*, pages 111–119. ACM, 2015.

- [31] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008.
- [32] D. Fitton and B. Bell. Working with teenagers within HCI research: Understanding teen-computer interaction. In *Proceedings of the 28th International BCS Human Computer Interaction Conference on HCI 2014-Sand, Sea and Sky-Holiday HCI*, pages 201–206. BCS, 2014.
- [33] D. Fitton, J. C. C. Read, and M. Horton. The challenge of working with teens as participants in interaction design. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pages 205–210. ACM, 2013.
- [34] J. H. Flavell, P. H. Miller, and S. A. Miller. *Cognitive development*. Prentice-Hall Englewood Cliffs, NJ, 1985.
- [35] J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo, and A. Repenning. LEGOsheets: a rule-based programming, simulation and manipulation environment for the LEGO programmable brick. In *Visual Languages, Proceedings., 11th IEEE International Symposium on*, pages 172–179. IEEE, 1995.
- [36] T. R. Green and M. Petre. When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics)*. GC van der Veer, MJ Tauber, S. Bagnarola and M. Antavolits. Rome, CUD. Citeseer, 1992.
- [37] T. R. Green, M. Petre, and R. Bellamy. Comprehensibility of visual and textual programs: A test of superlativism against the 'match-mismatch' conjecture. *ESP*, 91(743):121–146, 1991.
- [38] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.
- [39] S. L. Greene, S. J. Devlin, P. E. Cannata, and L. M. Gomez. No IFs, ANDs, or ORs: A study of database querying. *International Journal of Man-Machine Studies*, 32(3):303–326, 1990.
- [40] GSMA. Mobile education in the united states, 2012.
- [41] M. Guzdial. Programming environments for novices. *Computer science education research*, 2004:127–154, 2004.
- [42] W. Haidinger. MINT2020 zahlen, daten und fakten, 2013.
- [43] L. Hanna, K. Ridsden, and K. Alexander. Guidelines for usability testing with children. *interactions*, 4(5):9–14, 1997.
- [44] R. Harrison, D. Flood, and D. Duce. Usability of mobile applications: literature review and rationale for a new usability model. *Journal of Interaction Science*, 1(1):1–16, 2013.

- [45] A. Harzl, V. Krnjic, F. Schreiner, and W. Slany. Comparing purely visual with hybrid visual/textual manipulation of complex formula on smartphones. In *DMS*, pages 198–201, 2013.
- [46] S. Hooper. How do users really hold mobile devices, 2013.
- [47] J. P. Hourcade. Interaction design and children. *Foundations and Trends in Human-Computer Interaction*, 1(4):277–392, 2008.
- [48] J. P. Hourcade, K. B. Perry, and A. Sharma. Pointassist: helping four year olds point with ease. In *Proceedings of the 7th international conference on Interaction design and children*, pages 202–209. ACM, 2008.
- [49] S. Idler. 5 key criteria of a good user experience for children, 2013.
- [50] K. M. Inkpen. Drag-and-drop versus point-and-click mouse interaction styles for children. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(1):1–33, 2001.
- [51] J. P. Ioannidis. Why most published research findings are false. *Chance*, 18(4):40–47, 2005.
- [52] R. Jacob and K. S. Karn. Eye tracking in human-computer interaction and usability research: Ready to deliver the promises. *Mind*, 2(3):4, 2003.
- [53] R. Joiner, D. Messer, P. Light, and K. Littleton. It is best to point for young children: a comparison of children’s pointing and dragging. *Computers in Human Behavior*, 14(3):513–529, 1998.
- [54] K. Kahn. Toontalk TM—an animated programming environment for children. *Journal of Visual Languages & Computing*, 7(2):197–217, 1996.
- [55] T. Kallio, A. Kaikkonen, et al. Usability testing of mobile applications: A comparison between laboratory and field testing. *Journal of Usability studies*, 1(4-16):23–28, 2005.
- [56] A. Kay. Squeak etoys, children & learning. *online article*, 2006, 2005.
- [57] A. C. Kay. The early history of smalltalk. In *History of programming languages—II*, pages 511–598. ACM, 1996.
- [58] C. Kelleher and R. Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, 37(2):83–137, 2005.
- [59] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on*, 28(8):721–734, 2002.
- [60] J. Kjeldskov and C. Graham. A review of mobile HCI research methods. In *Human-computer interaction with mobile devices and services*, pages 317–335. Springer, 2003.
- [61] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, et al. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, 43(3):21, 2011.

- [62] R. Koitz and W. Slany. Empirical comparison of visual to hybrid formula manipulation in educational programming languages for teenagers. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 21–30. ACM, 2014.
- [63] D. Krannich and J. Friedrich. Mobile usability-testing, 2010.
- [64] T. Lang. Eight lessons in mobile usability testing.
- [65] E. L.-C. Law, V. Roto, M. Hassenzahl, A. P. Vermeeren, and J. Kort. Understanding, scoping and defining user experience: a survey approach. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 719–728. ACM, 2009.
- [66] C. M. Lewis. How programming environment shapes perception, learning and goals: logo vs. scratch. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 346–350. ACM, 2010.
- [67] J. R. Lewis. Ibm computer usability satisfaction questionnaires: psychometric evaluation and instructions for use. *International Journal of Human-Computer Interaction*, 7(1):57–78, 1995.
- [68] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.
- [69] Lohmann and Schaeffer. System Usability Scale (SUS) – an improved german translation of the questionnaire, 2013.
- [70] H. LORANGER and J. NIELSEN. Teenage usability: Designing teen-targeted websites. 2013.
- [71] S. Y. Lye and J. H. L. Koh. Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior*, 41:51–61, 2014.
- [72] C. Lynch. Information literacy and information technology literacy: new components in the curriculum for a digital culture. *Committee on Information Technology Literacy. Retrieved October*, 8:2004, 1998.
- [73] O. Machado Neto and M. d. G. Pimentel. Heuristics for the assessment of interfaces of mobile devices. In *Proceedings of the 19th Brazilian symposium on Multimedia and the web*, pages 93–96. ACM, 2013.
- [74] M. Madden, A. Lenhart, M. Duggan, S. Cortesi, and U. Gasser. Teens and technology 2013, 2013.
- [75] P. Markopoulos and M. Bekker. How to compare usability testing methods with children participants. In *Interaction Design and Children*, volume 2. Citeseer, 2002.
- [76] P. Markopoulos and M. Bekker. Interaction design and children. *Interacting with computers*, 15(2):141–149, 2003.

- [77] F. Martin. Kids learning engineering science using LEGO and the programmable brick. *Proc of AERA*, 96, 1996.
- [78] G. Mascheroni and A. Cuman. Net children go mobile - final report. 2014.
- [79] L. McIver. Evaluating languages and environments for novice programmers. In *Fourteenth Annual Workshop of the Psychology of Programming Interest Group (PPIG 2002)*, Brunel University, Middlesex, UK, 2002.
- [80] L. McIver, L. M., and D. Conway. GRAIL: A zeroth programming language, 1999.
- [81] F. McKay and M. Kölling. Predictive modelling for HCI problems in novice program editors. In *Proceedings of the 27th International BCS Human Computer Interaction Conference*, page 35. British Computer Society, 2013.
- [82] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Learning computer science concepts with Scratch. *Computer Science Education*, 23(3):239–264, 2013.
- [83] G. Meter and P. Miller. Engaging students and teaching modern concepts: Literate, situated, object-oriented programming. In *ACM SIGCSE Bulletin*, volume 26, pages 329–333. ACM, 1994.
- [84] T. G. Moher, D. Mak, B. Blumenthal, and L. Levanthal. Comparing the comprehensibility of textual and graphical programs. In *Empirical Studies of Programmers: Fifth Workshop*, pages 137–161. Ablex, Norwood, NJ, 1993.
- [85] R. Molich, M. R. Ede, K. Kaasgaard, and B. Karyukin. Comparative usability evaluation. *Behaviour & Information Technology*, 23(1):65–74, 2004.
- [86] R. Morelli, T. de Lanerolle, P. Lake, N. Limardo, E. Tamotsu, and C. Uche. Can android app inventor bring computational thinking to K-12. In *Proc. 42nd ACM technical symposium on Computer science education (SIGCSE'11)*, 2011.
- [87] B. A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT press, 1993.
- [88] B. A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT press, 1993.
- [89] D. Neary and M. Woodward. An experiment to compare the comprehensibility of textual and visual forms of algebraic specifications. *Journal of Visual Languages & Computing*, 13(2):149–175, 2002.
- [90] A. Ng, S. A. Brewster, and J. H. Williamson. Investigating the effects of encumbrance on one-and two-handed interactions with mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1981–1990. ACM, 2014.
- [91] J. Nielsen. Finding usability problems through heuristic evaluation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 373–380. ACM, 1992.

- 
- [92] J. Nielsen. Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 152–158. ACM, 1994.
- [93] J. Nielsen. *Usability engineering*. Elsevier, 1994.
- [94] J. Nielsen. Usability inspection methods. In *Conference companion on Human factors in computing systems*, pages 413–414. ACM, 1994.
- [95] J. Nielsen and T. K. Landauer. A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 206–213. ACM, 1993.
- [96] J. Nielsen and R. Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 249–256. ACM, 1990.
- [97] D. A. Norman and S. W. Draper. User centered system design. *Hillsdale, NJ*, 1986.
- [98] A. Oulasvirta, S. Tamminen, V. Roto, and J. Kuorelahti. Interaction in 4-second bursts: the fragmented nature of attentional resources in mobile HCI. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 919–928. ACM, 2005.
- [99] R. K. Pandey and M. M. Burnett. Is it easier to write matrix manipulation programs visually or textually? an empirical study. In *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, pages 344–351. IEEE, 1993.
- [100] J. Pane and B. Myers. Usability issues in the design of novice programming systems. 1996.
- [101] J. F. Pane, B. Myers, et al. Tabular and textual methods for selecting objects from a group. In *Visual Languages, 2000. Proceedings. 2000 IEEE International Symposium on*, pages 157–164. IEEE, 2000.
- [102] J. F. Pane, B. Myers, L. B. Miller, et al. Using HCI techniques to design a more usable programming system. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, pages 198–206. IEEE, 2002.
- [103] S. Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc., 1980.
- [104] S. Pasiali and A. MacFarlane. Adapting the heuristic evaluation method for use with children. In *Proceedings of the Workshop on child computer interaction: methodological research*, 2005.
- [105] M. Petre. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995.
- [106] M. Petre and A. F. Blackwell. Children as unwitting end-user programmers. In *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*, pages 239–242. IEEE, 2007.

- [107] A. Poole and L. J. Ball. Eye tracking in HCI and usability research. *Encyclopedia of human computer interaction*, 1:211–219, 2006.
- [108] M. Prensky. Digital natives, digital immigrants part 1. *On the horizon*, 9(5):1–6, 2001.
- [109] T. W. Price and T. Barnes. Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the eleventh annual International Conference on International Computing Education Research*, pages 91–99. ACM, 2015.
- [110] H. R. Ramsey, M. E. Atwood, and J. R. Van Doren. Flowcharts versus program design languages: an experimental comparison. *Communications of the ACM*, 26(6):445–449, 1983.
- [111] A. Repenning. Agentsheets: a tool for building domain-oriented visual programming environments. In *Proceedings of the INTERACT’93 and CHI’93 conference on Human factors in computing systems*, pages 142–143. ACM, 1993.
- [112] A. Repenning. Making programming accessible and exciting. *Computer*, 46(6):78–81, 2013.
- [113] M. Resnick, A. Bruckman, and F. Martin. Planos not stereos: Creating computational construction kits. *interactions*, 3(5):40–50, 1996.
- [114] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.
- [115] M. Resnick, S. Ocko, et al. *LEGO/logo—learning through and about design*. Epistemology and Learning Group, MIT Media Laboratory, 1990.
- [116] M. Resnick and B. Silverman. Some reflections on designing construction kits for kids. In *Proceedings of the 2005 conference on Interaction design and children*, pages 117–122. ACM, 2005.
- [117] J. Rubin and D. Chisnell. *Handbook of usability testing: how to plan, design and conduct effective tests*. John Wiley & Sons, 2008.
- [118] J. Sauro and J. S. Dumas. Comparison of three one-question, post-task usability questionnaires. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1599–1608. ACM, 2009.
- [119] J. Sauro and E. Kindlund. A method to standardize usability metrics into a single score. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 401–409. ACM, 2005.
- [120] J. Sauro and J. R. Lewis. Correlations among prototypical usability metrics: evidence for the construct of usability. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1609–1618. ACM, 2009.
- [121] J. Sauro and J. R. Lewis. *Quantifying the user experience: Practical statistics for user research*. Elsevier, 2012.

- [122] D. Scanlan et al. Structured flowcharts outperform pseudocode: An experimental comparison. *Software, IEEE*, 6(5):28–36, 1989.
- [123] M. Schmettow, C. Bach, and D. Scapin. Optimizing usability studies by complementary evaluation methods. In *Proceedings of the 28th International BCS Human Computer Interaction Conference on HCI 2014-Sand, Sea and Sky-Holiday HCI*, pages 110–119. BCS, 2014.
- [124] sesameworkshop. Best practices: Designing touch tablet experiences for preschoolers, 2013.
- [125] R. Sheehan. Children’s perception of computer programming as an aid to designing programming environments. In *Proceedings of the 2003 conference on Interaction design and children*, pages 75–83. ACM, 2003.
- [126] B. Shneiderman, R. Mayer, D. McKay, and P. Heller. Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM*, 20(6):373–381, 1977.
- [127] W. Slany. A mobile visual programming system for Android smartphones and tablets. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, pages 265–266. IEEE, 2012.
- [128] J. G. Smetana, N. Campione-Barr, and A. Metzger. Adolescent development in interpersonal and societal contexts. *Annu. Rev. Psychol.*, 57:255–284, 2006.
- [129] D. C. Smith. Pygmalion: a creative programming environment. Technical report, DTIC Document, 1975.
- [130] D. C. Smith, A. Cypher, and J. Spohrer. KidSim: programming agents without a programming language. *Communications of the ACM*, 37(7):54–67, 1994.
- [131] E. Soloway, M. Guzdial, and K. E. Hay. Learner-centered design: The challenge for HCI in the 21st century. *interactions*, 1(2):36–48, 1994.
- [132] A. Stefik and S. Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4):19, 2013.
- [133] L. Steinberg. Cognitive and affective development in adolescence. *Trends in cognitive sciences*, 9(2):69–74, 2005.
- [134] D. Tetteroo and P. Markopoulos. A review of research methods in end user development. In *End-User Development*, pages 58–75. Springer, 2015.
- [135] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. TouchDevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 49–60. ACM, 2011.



- 
- [136] I. E. van Kesteren, M. M. Bekker, A. P. Vermeeren, and P. A. Lloyd. Assessing usability evaluation methods on their effectiveness to elicit verbal comments from children subjects. In *Proceedings of the 2003 conference on Interaction design and children*, pages 41–49. ACM, 2003.
- [137] K. N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, 8(1):109–142, 1997.
- [138] C. Wilson. *User Experience Re-Mastered: your guide to getting the right design*. Morgan Kaufmann, 2009.
- [139] C. Wilson. *User Interface Inspection Methods: A User-Centered Design Method*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2014.
- [140] C. E. Wilson. Triangulation: the explicit use of multiple methods, measures, and approaches for determining core issues in product development. *interactions*, 13(6):46–ff, 2006.
- [141] J. M. Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006.
- [142] O. A. Wodike, G. Sim, and M. Horton. Empowering teenagers to perform a heuristic evaluation of a game. In *Proceedings of the 28th International BCS Human Computer Interaction Conference on HCI 2014-Sand, Sea and Sky-Holiday HCI*, pages 353–358. BCS, 2014.
- [143] D. Wolber, H. Abelson, E. Spertus, and L. Looney. *App Inventor*. ” O’Reilly Media, Inc.”, 2011.
- [144] U. Wolz, C. Hallberg, and B. Taylor. Scrape: A tool for visualizing the code of Scratch programs. In *Poster presented at the 42nd ACM Technical Symposium on Computer Science Education, Dallas, TX*, 2011.
- [145] D. Zhang and B. Adipat. Challenges, methodologies, and issues in the usability testing of mobile applications. *International Journal of Human-Computer Interaction*, 18(3):293–308, 2005.
- [146] F. Zijlstra and L. Van Doorn. *The construction of a scale to measure perceived effort*. University of Technology, 1985.

# Appendix

---

## 7.1 Pre Test Questionnaire

Before the test we asked the participants demographic as well as questions on their computer experience. "-" denotes that the participant was not able to answer the question.

- General
  - Q1 Are you allowed to use a calculator in school?
  - Q2 Do you have experience with using a calculator?
  - Q3 What are the brand and model name of the calculator you are using?
- Computer Experience
  - Q4 Do you own a computer?
  - Q5 Do you daily use the computer?
  - Q6 Do you have Internet access at home?
  - Q7 Are you allowed to go on the Internet by yourself?
  - Q8 What do you usually use the computer for?
- Smart Phone Experience
  - Q9 Do you own a smart phone?
  - Q10 For how long have you had the smart phone?
  - Q11 Do you know the brand and model name of the smart phone?
  - Q12 Do you own a tablet?
  - Q13 For how long have you had the tablet?
  - Q14 Do you know the brand and model name of the tablet?
  - Q15 What is the name of your favorite app?
  - Q16 What do you usually use the smart phone for?
  - Q17 What do you usually use the tablet for?

Participant	Gender	Age	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
1	female	16	yes	yes	-	yes	no	yes	yes	YouTube
2	male	16	yes	yes	with graphical display	yes	no	yes	yes	Games, Study/ Homework, YouTube, Facebook, Internet, Emails
3	male	17	yes	yes	-	yes	yes	yes	yes	Games, Study/ Homework, YouTube, Facebook, Internet, Emails
4	male	17	yes	yes	Ti Nspire Cx	yes	yes	yes	yes	Games, Study/ Homework, Youtube, Facebook, Internet, Emails
5	male	13	yes	yes	-	yes	no	yes	yes	Internet
6	male	13	yes	yes	-	no (sharing with siblings)	no	yes	yes	YouTube
7	male	13	yes	yes	-	no (sharing with siblings)	no	yes	yes	Games, Facebook
8	female	14	yes	yes	-	yes	no	yes	yes	YouTube, Email
9	female	16	yes	yes	-	yes	yes	yes	yes	Internet
10	female	16	yes	yes	-	yes	no	yes	yes	Games
11	female	16	yes	yes	-	yes	yes	yes	yes	YouTube
12	female	17	yes	no	TI Inspire XT	yes	yes	yes	yes	Games, YouTube, Study/ Homework
13	female	17	yes	yes	TI Inspire XT	yes	yes	yes	yes	YouTube, Study/ Homework
14	female	17	yes	yes	-	no (shared with parents)	no	yes	yes	Study/ Homework
15	female	17	yes	yes	TI Inspire TX	yes	yes	yes	yes	Facebook, YouTube

16	male	12	no	-	-	no (shared with siblings/parents)	yes	yes	yes	Games, Study/ Homework, Internet, Facebook, YouTube
17	male	13	no	-	-	no (shared with parents)	no	yes	no	Games, Study/ Homework
18	male	13	no	-	-	yes	no	yes	yes	Games, Study/ Homework, YouTube
19	male	12	no	-	-	yes	no	yes	yes	Games, Study/ Homework, YouTube, Facebook, Internet

Table 7.1: Pre-test questionnaire answers (1).

Participant	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17
1	yes	3 yrs	iPhone	yes	1.5yrs	CMX	Instagram	Drawing, Texting, Calling, WhatsApp	Internet
2	yes	3-4 yrs	Galaxy S4	yes	1 yrs	iPad	Facebook, WhatsApp, Internet	Facebook, WhatsApp, Internet	Games, Internet
3	yes	4 yrs	iPhone 5s	yes	1 yrs	mini iPad	Snapchat	Taking pictures, Texting, Calling, Internet, Facebook, Games	Taking pictures, Texting, Calling, Internet, Facebook, Study/Homework
4	yes	6 yrs	iPhone 5s	yes	2-3 yrs	iPad	Facebook	Internet, Facebook, Study/ Homework	Looking at Pictures, Internet
5	yes	2 yrs	iPhone	no	-	-	Clash of Clans	Games, Internet	-
6	yes	1 yrs	iPhone 4	no	-	-	YouTube	Taking pictures, Facebook, Internet	-
7	yes	1.5 yrs	Sony Xperia T	no	-	-	Fifa14	Facebook, WhatsApp	-
8	yes	0.5 yrs	Samsung Galaxy Core Plus	no	-	-	WhatsApp	Texting, Calling	-
9	yes	3 yrs	iPhone 4	no	-	-	WhatsApp	Internet	-
10	yes	1 yrs	Sony	no	-	-	Chrome	Texting	-
11	yes	3 yrs	Galaxy S2	yes	0.5 yrs	-	YouTube	Texting	YouTube
12	yes	4 yrs	iPhone 5C	no	-	-	WhatsApp, Snapchat	Texting, Calling	-
13	yes	5 yrs	Motorola	no	-	-	Twitter, Emails, Facebook	Emails, Twitter	-
14	yes	2 yrs	HTC One mini	yes	2 yrs	iPad	Facebook	Texting, Calling, Facebook	Games, Facebook
15	no	-	-	no	-	-	-	-	-

Participant	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17
16	yes	1.5 yrs	LG Optimus L5	no	-	-	Playstore Games	Texting, Calling	-
17	no	-	-	-	-	-	-	-	-
18	yes	1.5	Galaxy S3	no	-	-	Facebook	Facebook, Calling, WhatsApp	-
19	yes	2	Note 2	no	-	-	Games	Games, Taking Pictures, Internet, Facebook	-

Table 7.2: Pre-test questionnaire answers (2).

## 7.2 Tasks German



### Aufgabe 1

Probiere das Spiel Kitty Cross aus.



### Aufgabe 2

Wir möchten, dass das Spiel beendet wird sobald die Katze vom Auto erwischt wird. Aber leider ist ein Fehler im Skript vom Auto.

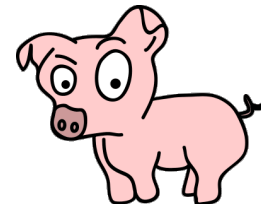
- Gehe zu den Skripts vom Auto.
- Suche das Skript „Setze Variable 'car position:' auf“
- Ersetze „position\_y“ mit „position\_x“



### Aufgabe 3

Unsere Katze ist zu langsam, hilf ihr schneller zu laufen! Dazu mach folgendes:

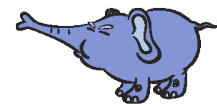
- Gehe zu den Skripts von der Katze.
- Suche das Skript „Ändere Y um“.
- Gib die folgende Formel ein:  $3 * \text{Zufall}(5,10) + 4 / 0.5$
- Bitte ändere nun die, gerade eingegebene, Formel in:  
 $3 * \text{Zufall}(6,12) + 4 * 0.5$



### Aufgabe 4

Wir brauchen noch immer ein Skript für die Kollision. Führe dazu die folgenden Schritte durch:

- Gehe zu den Skripts der Katze
- Suche nach dem „Falls“ Skript
- Ändere die Formel  
von:  $((\text{position}_y > -180) \text{ UND } (\text{position}_y < 180)) \text{ ODER } 0$



zu:  $((\text{position}_y > -180) \text{ UND } (\text{position}_y < 180)) \text{ ODER } ((\text{"car position"} > -150) \text{ UND } (\text{"car position"} < 150))$

Tipp: "car position" ist eine Variable.



# SCRATCH



## Aufgabe 1

Probiere das Spiel Ball Pong aus.



## Aufgabe 2

Der Balken (Paddle) scheint nicht richtig zu funktionieren, da er nicht der Mausbewegung folgt. Bitte stelle dies richtig:

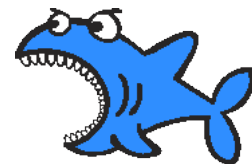
- Gehe zu den Skripts vom Balken (Paddle)
- Suche nach dem „setze x auf“ Skript
- Ersetze „Maus y-Position“ mit dem „Maus x-Position“ Skript.



## Aufgabe 3

Das Spiel ist so zu einfach, wir wollen, dass der Ball sich schneller bewegt.

- Gehe zu den Skripts vom Ball.
- Im ersten Block der Skripts ist ein „gehe \_er-Schritte“ Skript.
- Füge die folgenden Formel in dieses Skript ein:  
Zufallszahl von (0 bis 8) / 0.5 – 3 \* 2
- Ändere die Formel zu: Zufallszahl von (0 bis 8) / 0.5 **+ 1 \* 1**



## Aufgabe 4

Wir möchten, dass das Spiel beendet wird sobald der Ball den Boden berührt. Mache dazu folgendes:

- Gehe zu den Skripts vom Ball.
- Im 2. Skript Block suche nach dem zweiten „Falls \_ dann“ Skript.
- 



Ersetzte: (1 = 1)



durch: **x-Position > (X Position Paddle – 25)**

Tipp: "X Position Paddle" ist eine Variable.

### 7.3 Post Test Questionnaire

- General Questions
  1. How was it?
  2. Which task do you think was especially easy/difficult using Pocket Code?
  3. Which task do you think was especially easy/difficult using Scratch?
- Preference Questions
  1. Which system did you like better? Why?
  2. Which system do you think is easier to use?
  3. Was it easier to work with the formulas in Pocket Code or Scratch? (Explain your answer)
  4. Would you prefer to learn Scratch or Pocket Code in school?
  5. Could you imagine using Pocket Code or Scratch by yourself in your free time?