Chair for Computer Science 10 (Media Computing and Human-Computer Interaction)

RWTH AACHEN UNIVERSITY

Tangible Robotics: Developing a Tangible Programming Interface for Lego Mindstorms

Bachelor's Thesis
submitted to the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Till Bußmann

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Ulrik Schroeder

Registration date: 23.11.2019
Submission date: 19.05.2020

Eidesstattliche Versicherung

______________________________	______________________________
Name, Vorname	Matrikelnummer

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

____

____

____

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

______________________________	______________________________
Ort, Datum	Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

______________________________	______________________________
Ort, Datum	Unterschrift

Contents

List of Figures

Abstract

Programming is both an incredibly useful skill to have in any fields related to STEM and one that is notorious for being difficult to learn. This seems to scare off and discourage pupils and students alike. Since this problem has been well-known since the very beginnings of widespread programming, many approaches have been developed that try to both entice novices and make their entry into programming as smooth as possible. The approaches that we present in this thesis revolve around using robotics as a very palpable discipline of programming, visual programming languages, tangible user interfaces and pair programming. With regard to pair programming, we present a current interface design approach called 'behavior-centered game design' and the concepts of 'distinct actions' and 'distributed resources' in the context of pair programming.

Based on our review of previous approaches, we present Tangible Robotics, a tangible programming interface for Lego Mindstorms robots. It is a visual programming language environment on a tabletop multi-touch display that utilizes tangibles as representations of distinct types of actions that can be performed by the robot.

To evaluate our interface, we further present the findings of a study into its intuitiveness as well as possible improvements to the interface. Subsequently, we outline a study that is able to investigate the effects of tangibles as a 'distributed resource' in the context of pair programming but that we were unfortunately unable to perform due to the COVID-19 pandemic.

Überblick

Programmieren zu können, ist sowohl eine in allen MINT-Disziplinen sehr nützliche Fähigkeit als auch berüchtigt dafür, dass sie schwer zu erlernen ist. Dies scheint sowohl Schüler als auch Studenten zu verunsichern und abzuschrecken.Da dieses Problem schon seit Anbeginn der Programmierung bekannt ist, gibt es auch viele Ansätze, um in Neulingen Interesse zu wecken und ihnen den Eintritt in die Programmierung so reibungslos wie möglich zu gestalten. Die in dieser Arbeit vorgestellten Ansätze drehen sich um Robotik als sehr greifbaren Anwendungsbereich, visuelle Programmiersprachen, Tangible User Interfaces und Paarprogrammierung. Hinsichtlich der Paarprogrammierung stellen wir einen aktuellen Interfacedesign-Ansatz namens 'behavior-centered game design' und die Konzepte 'distinctive actions' und 'distributed resources' im Kontext der Paarprogrammierung vor.

Ausgehend von unserer Betrachtung vorausgegangener Ansätze, stellen wir Tangible Robotics vor, eine Tangible-gestützte Programmierschnittstelle für Lego Mindstorms Roboter, die eine visuelle Programmierschnittstelle auf einem waagerechten Multi-Touch-Display ist und Tangibles als Repräsentation von verschiedenen Arten von Roboteraktionen nutzt.

Um unser Interface zu evaluieren, stellen wir die Ergebnisse einer Studie über seine Intuitivität sowie mögliche Verbesserungen des Interfaces vor. Anschließend stellen wir noch eine Studie dar, die die Auswirkungen von Tangibles als 'distributed resource' im Zusammenhang der Paarprogrammierung hätte erörtern sollen, die wir aber leider wegen Einschränkungen durch die Corona-Pandemie nicht durchführen konnten.

Acknowledgements

First off, I would like to thank my thesis supervisor Christian Cherek for his time and guidance in the process of writing this thesis.

I also want to thank Prof. Dr. Jan Borchers and Prof. Dr. Ulrik Schroeder for supervising and examining the thesis.

Furthermore, I want to thank all those that took time out of their day to participate in my user study and provide me with valuable feedback.

Last but not least, thank you to both my family and friends for supporting me in many ways, may it be offering their insight, proof reading or just non-CS-related conversations during lunch breaks.

Oh, and COVID-19. Thanks for nothing.

Conventions

Throughout this thesis we use the following conventions.

Text conventions

Definitions of technical terms or short excursus are set off in colored boxes.

Excursus:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition: *Excursus*

Source code and implementation symbols are written in typewriter-style text.

`myClass`

The whole thesis is written in American English.

Chapter 1

Introduction

Programming is both a useful skill in many fields and difficult to learn.

The ability to program has become more and more relevant today than it ever has been. It has transitioned from a skill only possessed by expert professionals and enthusiasts to one that proves useful in almost all STEM-related higher education. 'Construction Informatics', 'Computer Science in Mechanical Engineering' and 'Geo Informatics' are only three examples of interdisciplinary modules[1] by the RWTH Aachen university that teach programming and basic concepts of computer science to students in fields outside computer science.
It is not surprising then that more and more voices call for a stronger focus on computer science (and programming in particular) in educational institutions at even younger ages and more widespread than now. And even though many schools offer computer science as an elective subject, this offer appears to not be used as much as it could be. For example, according to the Schulministerium NRW [2019], only about 1.500 of over 75.000 pupils taking the Abitur in North Rhine Westphalia in 2019 elected to take computer science as one of their final exams. That makes it the least popular of the main STEM fields.
This is often attributed to the difficulty that comes with programming. And looking at the world of higher education where introductory programming courses see a worldwide average failure rate of about a third (Watson

[1]https://online.rwth-aachen.de/RWTHonline/ee/ui/ca2/app/desktop/#/slc.cm.reg/student/modules

and Li [2014]) appears to prove that right.
Learning to program is said to not just result in a practical skill but also hone students' critical thinking and problem solving skills (Saeli et al. [2011]). And yet, the slow adoption and failure rates quite clearly suggest that learning and just as importantly teaching to program is not easy and neither is motivating students to give it a try.
Thus, many concepts and tools have been developed to both encourage potential programmers to try it and to ease the earliest steps. We will showcase some of the most common ones in the following paragraphs.

Programming education often starts with textual languages.

Part of the problem may be the way programming is traditionally taught in schools and universities. Quite often, students' first programming language is one of the most popular textual ones, such as Java or Python. This is especially the case in university introductory classes. However, a very popular and well researched way of easing the learning process in programming is using visual programming languages instead.

Definition: *Visual programming languages*

VISUAL PROGRAMMING LANGUAGES:
Visual programming languages (VPLs) are all those languages in which programs are not created through typing key phrases and words but through a graphical interface that can utilize spatial arrangement, graphical metaphors or visual cues.
A common concept in VPLs is using blocks to represent distinct actions.
Figure 1.1 shows an example of a VPL.

Visual programming languages have key advantages over textual languages that makes them more suitable for education.

Rather quickly, visual programming languages have shown to provide some key advantages over textual programming languages.
VPLs have shown to

- improve user experience (Booth and Stumpf [2013])
- give an easier entry point for beginners and a tighter development and feedback cycle (Powers et al.

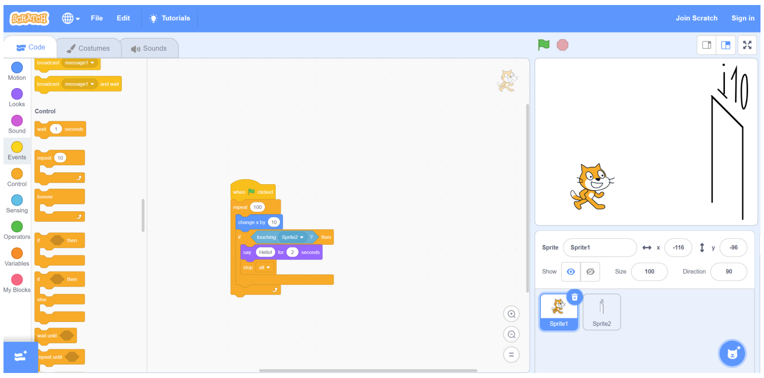


Figure 1.1: The visual programming language Scratch[2] is one of the most popular. It was developed by the MIT Media Lab

[2006]) as programs are always ready to run

- and be more engaging than textual ones in certain environments (Papert and Watt [1977]).

Early programming experience, especially with visual languages, has shown to spark interest and motivation to pursue programming even if knowledge transfer can be limited due to visual languages obscuring some vital aspects of textual languages (Franklin et al. [2016]).

When it comes to the nature of programming tasks that will be tackled by novices, educational robots are a popular target. They are used due to their close ties to real world tasks, their feedback-driven nature and affinity towards collaborative learning (Anwar et al. [2019]).

Robotics is a preferred field of application due to its physicality.

In formal, educational environments such as a schools' computer science courses, it is rarely the case that students are learning to program alone in front of a computer or in teacher-centered lectures. Instead, the majority of programming education follows a pair programming approach, meaning that two students team up in a practical programming task. As such, pair programming clearly falls into the category of collaborative learning.

Collaborative learning in the form of pair programming is the predominant form of programming education.

To gain insight into why pair programming is the predominant form of teaching, we can look at the criteria for environments in which collaborative learning makes the most sense. According to a review of multiple studies by Preston [2005], the criteria are:

- a complex or conceptual task
- problem solving is desired
- divergent thinking or creativity is desired
- mastery and retention are important
- quality of performance is expected
- higher-level reasoning strategies and critical thinking are needed.

Pair programming leads to better performance, increased understanding and more enjoyment.

According to Preston's review, all these criteria are met by pair programming. As a result, it appears that pair programming leads to better resulting programs, decreased task completion time and increased understanding of the coding process. Furthermore, students appear to enjoy programming more and exam results and course completion rates increase. All these effects are backed by the findings of the published review of multiple studies concerning pair programming.

Pair programming can break down due to a difference in skill level or the nature of the task.

However, there is also evidence that suggests that pair programming is not always an effective method of teaching programming. Chaparro et al. [2005] accompanied an Object Oriented Programming postgraduate course and came to the conclusion that differences in skill level strongly affected participants' collaboration, more precisely in that the more experienced user would take the 'pilot' role of programming more often. They further found that pair programming was even detrimental for some tasks such as debugging as participants felt it was tiring and less enjoyable.

Pyrus offers a game-like programming environment that gives pair programming more structure.

The Pyrus[2019] programming environment, which we will properly introduce in 2.1 "Collaborative Programming and Behavior-Centered Game Design", is an attempt

at giving pair programming more structure through a game-like development process. In it, researches identify desirable behavior in participants and methods of promoting such behavior. The methods the authors identified for programming education were 'distributed resources', the concept of distributing elements that are needed for task completion between users, 'enforced turn-taking' and programs consisting of 'distinct actions'.
They found that users collaborated more equally and spent more time planning.

Tabletop multi-touch displays lend themselves to the use of tangibles.

While Pyrus had two participants sitting next to each other in front of separate computers using mouse and keyboard, our work will focus on tabletop multi-touch displays instead.
If turned on their side, large multi-touch displays afford not just interaction with virtual elements on the screen but also gain the functionality of a tabletop. Provided a relatively large display, multiple users can then gather around the tabletop and all interact with it at once.
Another advantage of a tabletop setup is that its very nature affords placing objects on it. This makes tabletop multi-touch displays a very popular environment for tangible user interfaces.

Definition: *Tangible user interfaces (TUIs)*

TANGIBLE USER INTERFACES (TUIS):
Tangible user interfaces are interfaces that translate physical interactions with real world objects into virtual actions seamlessly. The real world objects that are used as controls are referred to as *tangibles*.
Figure 1.2 shows three exemplary tangible user interfaces.

PUCs are a form of tangible that is used on tabletop multi-touch displays and that can be detected by the table.

Probably the most common variety of tangibles in connection with tabletop displays are objects that interact digitally with the tabletop's displayed content. In the case of multi-touch displays, this can happen through touch input. An example of this kind are passive untouched capacitive widgets (PUCs) by Voelker et al. [2013]. PUCs are widgets with multiple conductive pads on their underside that are con-

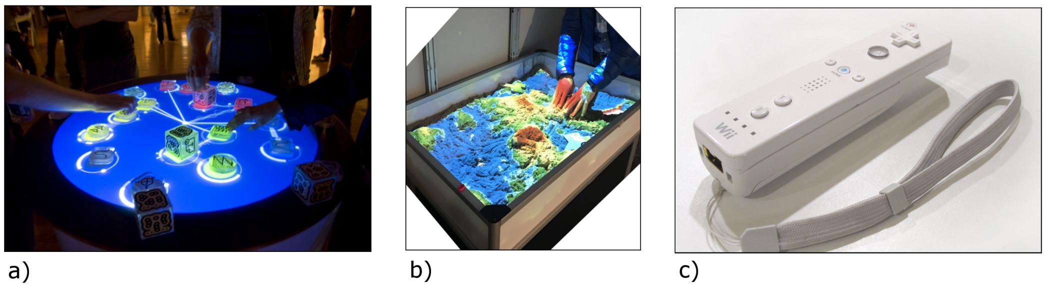


Figure 1.2: Three tangible user interfaces of different design. a) The reacTable* by Jordà [2006], a collaborative music instrument with a tabletop tangible user interface. Changing an object's position and orientation changed the instrument's sound[a]. b) SandScape by Ishii et al. [2004] uses sand as an input method as users alter a sand model with their hands. The model is simultaneously scanned and terrain analysis data is projected back onto it[b]. c) The Nintendo Wii Remote can be seen as a commercial tangible user interface (e.g., when used by games as a golf club)[c]

[a]https://www.flickr.com/photos/84466661@N00/539568298/
[b]https://commons.wikimedia.org/wiki/File:SandScape.jpg
[c]https://commons.wikimedia.org/wiki/File:Wii_Remote_Image.jpg

nected to each other through conductive material. Their electrical connection allows them to be picked up as touch points by any multi-touch display using mutual capacitance. As the pads are able to ground themselves, PUCs do not require external grounding through a user or wire. If PUCs have at least three conductive pads in a unique constellation, they can be identified through the touch points they create and their orientation can be extrapolated from them.

Outline of our work and contribution

While the behavior-centered game design approach by Shi et al. [2019] and Pyrus, the result of applying this mentality to programming education, offer an interesting and valuable approach to programming, Pyrus has also shown to be too restrictive for some users who complained about the system being inefficient as a result.
A question that arises out of this is whether the positive effects of Pyrus can be kept even if its restrictions are lowered somewhat.

In this paper, we propose a robotics programming environment that aims to be as novice-friendly as possible. In our approach at doing so, it is supposed to inherit the positive effects of robotics and tangible user interfaces while also providing a structured pair programming environment that benefits from collaborative learning effects. We will combine the aspects of 'distinct actions' and 'distributed resources' to achieve said structure.
In the evaluation of our environment, we are particularly interested in user collaboration but also in how much time they spend planning compared to testing and debugging.

Chapter 2

Related work

Overview

In this chapter, we will present other research projects that revolve around the intersection of tangible interaction, robotics, programming education and collaborative learning.
We will begin by investigating when collaboration in pair programming breaks down. Additionally, we look at an approach that aims to prevent that from happening.
Next, we will present research that indicates that programming education also benefits from the use of robotics as it is a field of programming that is very connected to the real world. Moreover, we will also look at research into tangible user interfaces, both in general collaborative learning environments and in the field of robotics.
Lastly, we outline the impact of all these findings on our own work and where it fits into the existing research.

2.1 Collaborative Programming and Behavior-Centered Game Design

As mentioned, pair programming can sometimes break down due to discrepancies between partners' skill level and the nature of the task at hand. While pairing individuals with equal programming knowledge would address some of these shortcomings, it would not address the issue that debugging poses. On top of that, this can only be done in formal environments with an instructor. In informal environments where pairs would form naturally, this is not possible.

Instead,Behavior-centered game design identifies obstacles, desirable behaviors when encountering obstacles and methods to encourage said behavior. current research intent on improving pair programming focuses on building game-like programming interfaces that facilitate active collaboration and encourage a longer planning phase before programming in order to reduce the time spent debugging.

h

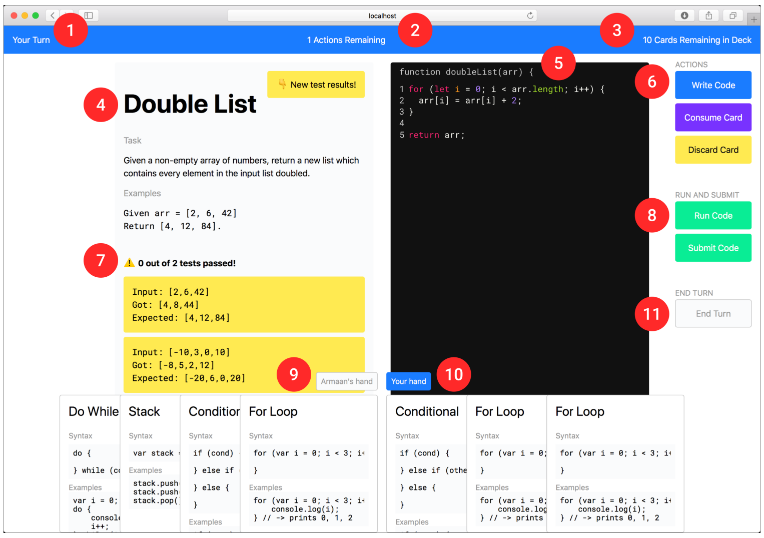
Shi et al. [2019]ave developed a behavior-centered game design approach for this. In this approach, game designers identify obstacles in the learning process and the low-level behaviors that they want to encourage in players in order to overcome these obstacles and reach high-level learning objectives.

In the case of collaborative programming education, Shi et al. identified programming newcomers omitting a planning stage and therefore having a poorer understanding of a problem as one of the obstacles. Another identified obstacle was that novices did not collaborate in an effective way due to a lack of structure in the pair programming approach.

Based on this, Shi et al. aimed to encourage users to plan more and participate equally in problem solving. The methods they employed for this purpose were giving participants discrete actions to choose from, a failure condition to encourage planning and enforced turn-taking as well as distributed resources in the form of programming constructs to encourage more equal participation.

Pyrus presents restrictions that lead to more planning and more equal collaboration but also result in frustration.

They designed the Pyrus programming interface, displayed in figure 2.1, alongside an equivalent interface that did not enforce these restrictions and compared them in a user study. Even though they did find that Pyrus encouraged novices to plan more and participate more equally in the problem solving, many participants also complained that the introduced constraints were frustrating and made Pyrus less efficient than pair programming.



Double List

Task

Given a non-empty array of numbers, return a new list which contains every element in the input list doubled.

Examples

```
Given arr = [2, 6, 42]
Return [4, 12, 84].
```

⚠ 0 out of 2 tests passed!

```
Input: [2,6,42]
Got: [4,8,44]
Expected: [4,12,84]
```

```
Input: [-10,3,0,10]
Got: [-8,5,2,12]
Expected: [-20,6,0,20]
```

```
function doubleList(arr) {
for (let i = 0; i < arr.length; i++) {
  arr[i] = arr[i] + 2;
}

return arr;
```

Figure 2.1: The Pyrus interface, which displays §1) the current player's turn, §2) the pilot's number of remaining actions, §3) the number of cards in the deck, §4) the problem prompt, §5) the editor, §6) available actions (i.e., write, consume, and discard), §7) test cases, §8) buttons to run or submit code, §9) the partner's hand, §10) the player's hand, and §11) a button to end the turn. §6 and §11 are omitted in the co-pilot's interface, since the co-pilot cannot perform actions.

2.2 Using Robotics in (Collaborative) Programming Education

Another common sight in early programming education is robotics. All around the world, robots are a popular means of teaching the basics of programming to novices and children and the demand for educational robots appears to still be increasing. The international non-profit STEM organization FIRST that focuses on hosting robotics competitions registered 660 000 students aged 4 to 18 for its 2019-2020 season alone[2020].

Robots lend themselves well to programming novices due to their feedback-driven nature and their recourse to real world tasks.

The popularity of robotics in schools can be attributed to educational robots having a host of benefits for school children as reported by Anwar et al. [2019]. In a systematical review of 147 studies on educational robots that took part between 2000 and 2018, they found that education robots helped students understand abstract concepts, created a learning environment that was both feedback-driven and of a collaborative nature and gave them the opportunity to work on real-world problems. As a result, using educational robots lead to students demonstrating improved knowledge. In addition, their findings indicated that educational robots improved students' attitude towards science, engineering and robots.

Robotics also positively affect learning in collaborative settings.

As it is also a discipline of programming, robotics education also appears to be affected positively by applying the concept of collaborative learning, which a study by Denis and Hubert [2001] showed. The study took place in a primary school and analyzed the behavior of groups of 2 to 4 pupils who were given the task of building and programming a robot. They observed that during the activity, pupils always worked together and communicated about their task. They also observed that if provided with a prebuilt robot, pupils did not collaborate less and that conflicts about the task influenced the outcome positively.

2.3 Tangibility in a Collaborative Learning Environment

A reason why robotics education works well in collaborative learning environments may be the inherent tangibility of robotics. Groups of novices that learn about robotics are bound to interact with not just the intangible programming process but also the very tangible robot.

Tangible user interfaces improve task performance and boost learning effects if used in a perceptual mapping as well as in a more abstract mapping.

Tangible user interfaces have been shown to enhance collaborative learning environments in a multitude of ways.
A study by Do-Lenh et al. [2010] had groups of participants work on the design of a warehouse with either a tangible interface or with pen and paper. Figure 2.2 shows participants using both interaction methods. In the tangible condition, a scaled down version of the warehouse is constructed out of plastic shelves, loading docks and the like. Each element has a unique visual tag that is recognized by a camera mounted above the model. Visual feedback such as whether a forklift can fit between two shelves is projected onto the model.
When comparing the two interaction methods, the researchers found that participants using the tangible user interface were able to, on average, find more solutions and a better final solution. While they did not find significant differences in participants' understanding of the subject and problem-solving ability, some of the same researchers lead by Schneider et al. [2011] later repeated the study with a multi-touch interface instead of the pen and paper design process. They found that groups using the tangible user interface outperformed the groups that used an interactive multi-touch interface in the same performance metrics as before. Further, they did find a significantly higher learning effect for the tangible interface compared to the multi-touch interface.

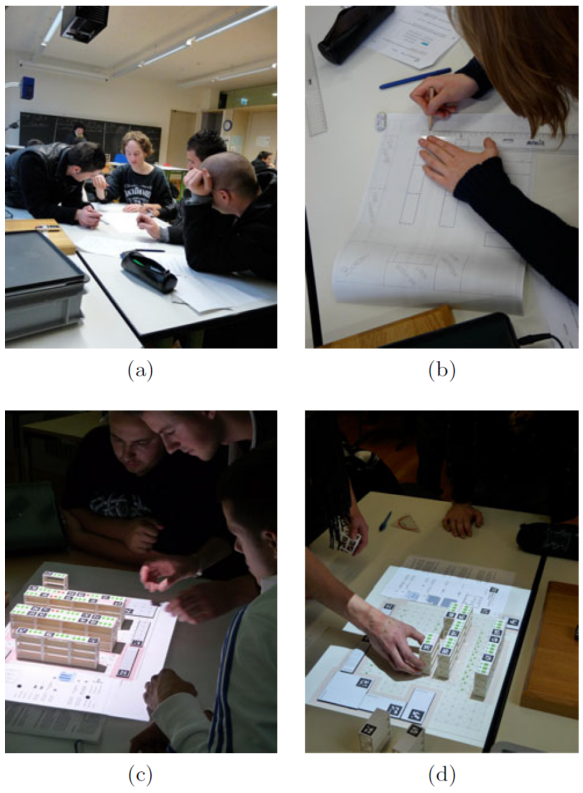

(a) (b) (c) (d)

Figure 2.2: a, b): Participants using pen and paper to design a warehouse layout. c, d): Participants using tangible 'shelves' with an interactive warehouse simulation for the same task.

Definition: *Mapping*

MAPPING:
In the context of interfaces, mappings describe the relationship of control elements, performed actions and their intended result.
Well-designed, natural mappings are immediately understood by users and can be achieved through

- spatial analogies: the control layout mimics the spatial attributes of the intended action or result
- perceptual analogies: the control elements mimic the appearance of the controlled object. Perceptual mappings are always spatial mappings.
- biological or cultural analogies: the control layout leverages biological or cultural norms (e.g., order from top to bottom)

Another study that is worth mentioning in this context is Combinatorix, also by Schneider et al. [2012]. Based on the findings of their previously mentioned work, they explored whether the same effects can be observed if tangibles are not used in a spatial or perceptual mapping but for a complex concept. The developed Combinatorix system taught students the abstract concepts of combinatorics by associating them with tangible objects. In a small, informal study, users enjoyed using Combinatorix and found it helpful.

TUIs also work with non-spatial, abstract mappings.

The success of Schneider et al. in mapping tangibles to abstract concepts leads to the question of whether the complex topic of programming could also be taught well through tangible user interfaces.

2.4 Tangibility in Robotics Education

Approaches of bringing programming into the physical world of learners with the help of tangible objects have been around for decades. According to a historical overview by McNerney [2004], the Logo programming language of the 1970s and the robots it controlled were among the first of these approaches. The idea of tangible objects as blocks of a programming language first showed up in the late 1990s.

Robotics education benefits not just from the tangibility of robots themselves but also from tangible user interfaces.

Compared to other programming situations, robotics already affords an additional tangibility due to programmers having a clear, physical representation of the result of their program's commands and their goal due to them being anchored in the real world. It is therefore not surprising that a lot of interest in designing just as tangible ways of programming robots has existed for quite some time now.

One approach to make the robot programming task more tangible is using the robot as part of the programming process itself.

Using the robot itself as a tangible is a natural way of interaction.

RoboTable2 [2011] is a tangible programming environment of this category. Sugimoto et al. used a multi-touch table and a robot that, if placed on the table, could be tracked. Robot movements could be programmed directly by moving the robot by hand. If a robot was moved so that an obstacle fell into its virtual 'cone of sight', the multi-touch interface gave users the option to program behavior for when the robot sees the obstacle again. An exemplary task can be seen in figure 2.3.

When comparing RoboTable2 to an equivalent interface that only used the multi-touch table, the researchers found that participants found it easier to both tell the robot when to do something and what. Moreover, users were more satisfied with the capabilities of their programs.

While RoboTable2 looked at the effect of tangible interaction on programming learning for single individuals, there is also the question of whether tangible user interfaces also hold up in collaborative programming environments. The two studies described in the following paragraphs are aiming to shed light on this question.

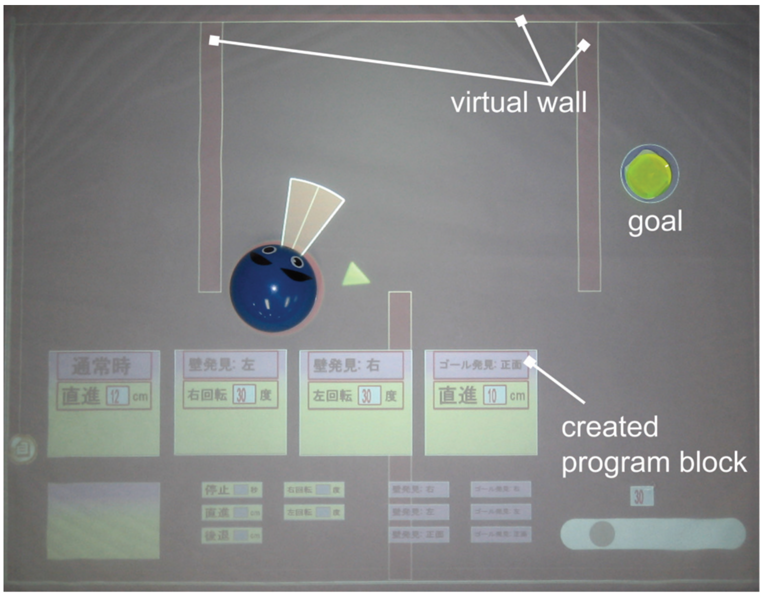


Figure 2.3: The interface of RoboTable2. Users were asked to program the robot to navigate a maze.

Tangible program blocks are more inviting than digital ones and lead to more collaboration.

In a series of studies by Horn et al. [2009], researchers compared Tern, a tangible programming interface, to a graphical one using mouse and keyboard input in the informal environment of a robotics exhibition. Figure 2.4 shows the programming blocks that were used by Horn et al. for Tern. Each block represents one robot action such as movement in a direction or playing a whistling sound.

Their findings suggest that the tangible interface was more inviting and more suitable for child use, both two aspects with great importance in informal environments that require potential programming learners to engage out of their own interest. They further found that while Tern was not more apprehensible or engaging, groups of visitors collaborated more actively, which suggests that the positive effects of collaboration in programming education might be enhanced through the use of tangible user interfaces.

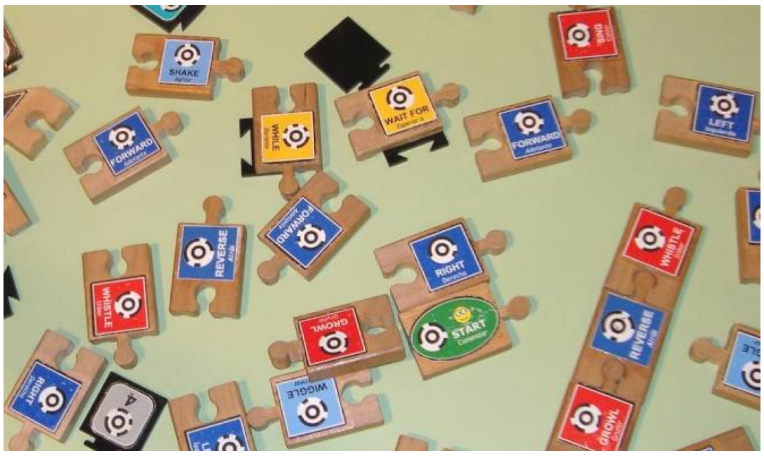


Figure 2.4: Tangible programming blocks as used in Tern. They are shaped like puzzle pieces and connect to each other to form a chain.

Tangible program blocks also improves novices' self-beliefs.

The concept of physical blocks representing singular programming commands was also used in a study conducted by Melcer and Isbister [2018]. They created Bots & (Main)Frames, a programming game that used physical blocks to dictate a robot's movements as well as more complicated blocks such as a loop or a call to a self-defined function. While no actual robot was used, users navigated a virtual robot avatar through a series of puzzles. It is further noteworthy that users were limited in the number of commands they could use in their program and additional function. Figure 2.5 shows the tangible user interface of Bots & (Main)Frames. The main program and the additional function are declared by placing a block on the respective position on the table and by hooking further programming blocks onto them.

In a 2x2 study, they compared the effects of individual vs. collaborative work and mouse vs. tangible interface by comparing Bots & (Main)Frames to an equivalent graphical programming interface that was operated by mouse input. In accordance with Horn et al., they found that using the tangible interface resulted in users having improved programming self-beliefs (such as debugging self-efficacy or programming interest) and higher situational interest and feelings of enjoyment. While they did find that collaboration reduced users' programming anxiety, they

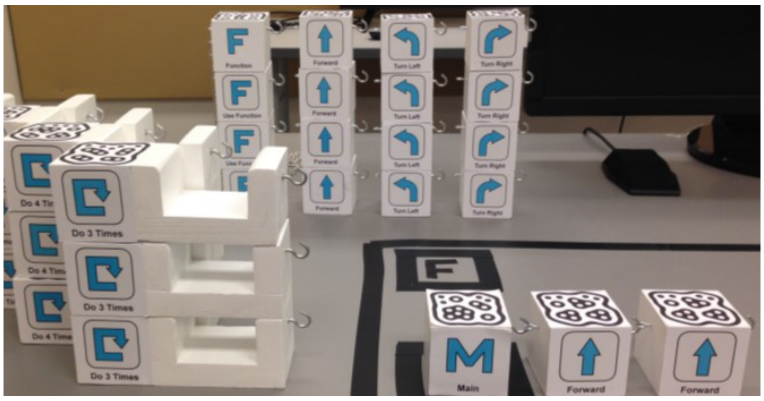


Figure 2.5: Bots & (Main)Frames used tangible programming blocks for movement as well as function calls and loops. They were chained together through hooks.

also noted that the use of a single access point in the form of one mouse resulted in notably worse performance in all of the above mentioned metrics. They attribute this to a tendency of one user dominating control of the single input device.

Chapter 3

Implementation

Overview over the interface

Tangible Robotics, the robotics programming environment that we present in the following chapter, was built with a very specific use case in mind. We aim to investigate the effects of using tangibles as distributed resources in a pair programming robotics learning environment. This imposes a set of specific requirements on the system.

The interface needs to be simple to use

Firstly, the interface needs to be very easy to understand due to it being employed in a learning environment with users who have not interacted with it before and may have very little or no prior programming knowledge. For this reason, we decided to implement a visual, block-based programming environment in which blocks mirror distinct actions. As mentioned in 1 "Introduction", this is common among programming environments tailored to novices as it engages novices more and leads to users achieving their programming goals quicker and more frequently. These effects were also investigated and confirmed by Price and Barnes [2015] in a study directly comparing both methods.

It needs to support tangibles and multi-touch

Secondly, it must support both a tangible-based input in which the tangibles govern a set of resources in a way that makes it unavoidable for pairs to work together and a more traditional but directly comparable input method. As a consequence, the programming interface was implemented in Swift with the MultiTouchKit framework by Linden [2015] which supports PUCs as introduced by Voelker et al. [2013]. It runs on a horizontal, large, multi-touch display. Using MultiTouchKit enables the software to uniquely identify tangibles and their positions when placed on the display. The first attempt at linking tangibles to a distributable resource was to link each tangible to a specific program block and distributing the tangibles among users. This idea was in the end dropped in favor of another method due to MultiTouchKit and the display only allowing for a limited number of tangibles to be reliably detected. Instead, each tangible was assigned to one type of block. As long as programming tasks were designed to require blocks of multiple types and the tangibles for these types were distributed among users, this meant that no user could solve tasks without any input by the other user.

It needs to create code that is executable by a robot

Thirdly, the system has to be able to translate the sequence of program blocks that users specify into code that can be conveniently executed by a robot. The robot we chose to use was the popular Mindstorms Education EV3[1], a robotics kit by the Lego company. Our software generates C code that utilizes c4ev3[2]'s EV3-API for robot-specific functions.

Lastly, it was desirable that the interface still possessed a level of functionality that allowed for more complex programs to be created so that users were not limited in their doing and that the interface's value for future use was preserved.

Based on these requirements, we implemented the system that will be described in this chapter. Figure 3.1 shows an example program on the interface in multi-touch mode.

[1]https://education.lego.com/en-us/middle-school/intro/mindstorms-ev3

[2]http://c4ev3.github.io/

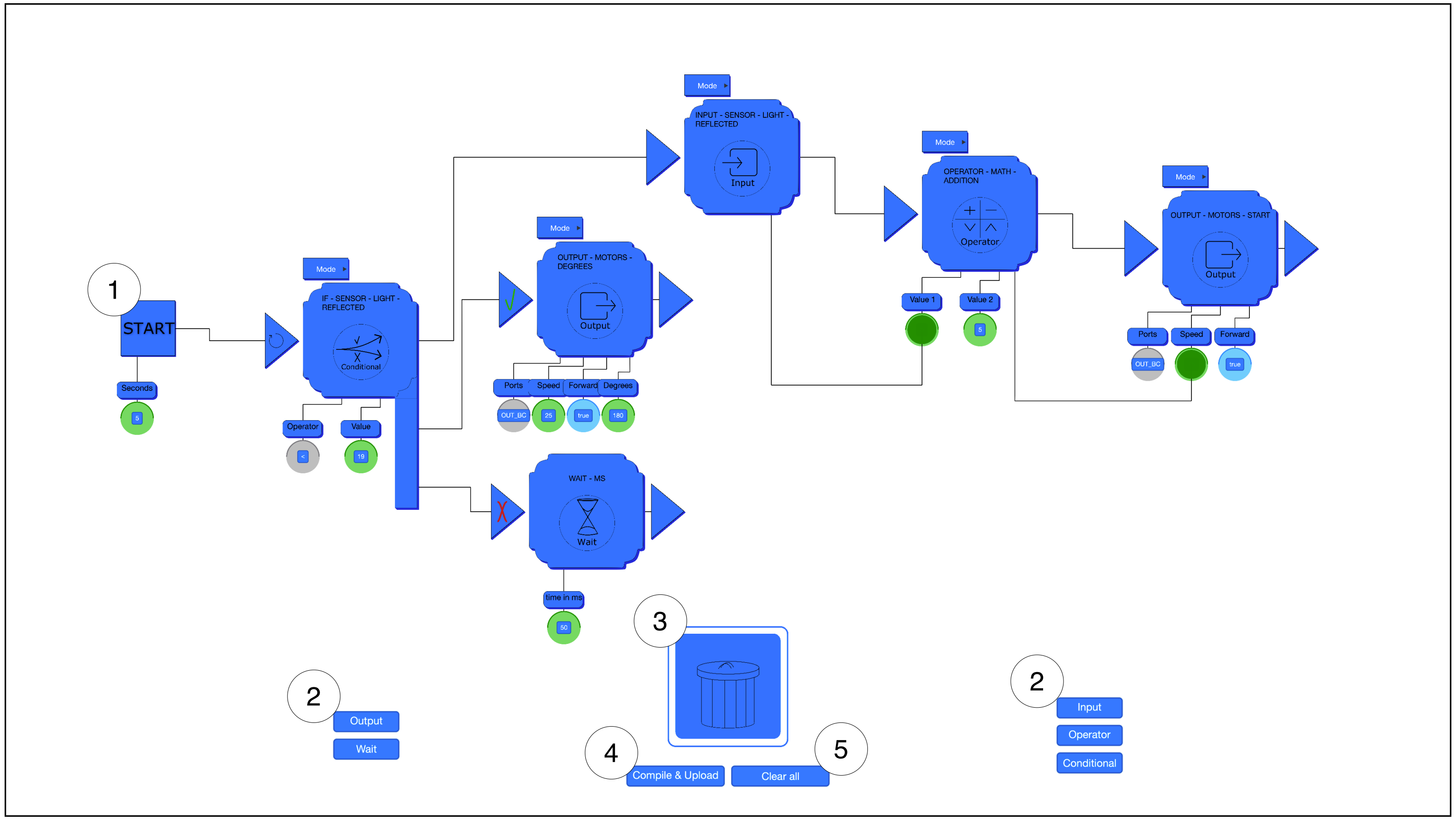


Figure 3.1: Tangible Robotics example program created in multi-touch mode. The following permanent GUI elements are depicted: 1) *start* block (entry point of each program), 2) block creation buttons, 3) trash symbol (if dragged on, blocks get deleted), 4) 'Compile & Upload' button, 5) 'Clear all' button (remove all blocks)

3.1 Hardware

The following section will describe the hardware that was used to put the Tangible Robotics software into practice.
The software runs on an Apple iMac Pro, however any device running macOS Catalina[3] can conceivably be used. The iMac is connected to an 84 inch Microsoft Surface Hub[4] over USB and HDMI in order to receive touch input and display the user interface on the Surface Hub. The Surface Hub is oriented horizontally.

We used PUCs on a Microsoft Surface Hub 84”

The tangibles that are used in Tangible Robotics are PUCs that follow the design of Voelker et al. [2013]. This means that they each have three conductive pads that are connected to each other through conductive copper foil. The copper foil also wraps around the hilt so that the user's body capacitance can be used to aid grounding them when touched. Each PUC has a unique constellation of conductive pads by which it can be identified. The tangibles are made out of plywood, feature a square base and a cylindrical hilt. A symbol representing the type of blocks that is managed by the tangible as well as the type's name are fixed to the top of the hilt. Figure 3.2 shows the five tangibles that were later used in the pre-study.

Programs made with the interface run on a Lego Mindstorms EV3 through a C API.

For our robot we chose the Lego Mindstorms EV3 for its adaptability with a wide array of possible sensor and motor configurations. The configuration that we chose for our study can be seen in figure 3.3 and consists of an 'intelligent brick', two large motors powering two front wheels, a small motor powering an arm at the front of the robot, an ultrasonic sensor, a touch sensor, a light sensor and a gyroscope. The 'intelligent brick' is a micro-controller equipped with buttons, a display, a speaker, four input and four output ports.
Furthermore, the commonness of Lego Mindstorms in the field of educational robotics (e.g., Klassner and Anderson [2003], Barnes [2002] and Kim and Jeon [2008]) and the

[3]https://www.apple.com/macos/catalina/
[4]https://www.microsoft.com/en-us/surface/business/surface-hub

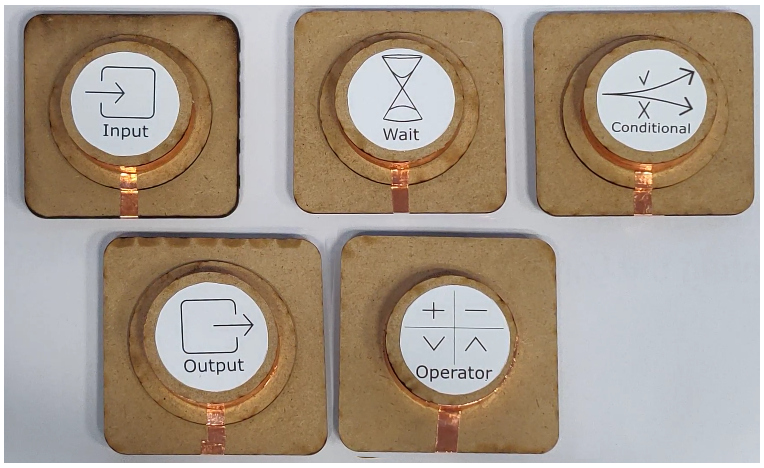


Figure 3.2: These five tangibles were used with Tangible Robotics, each representing one type of program block

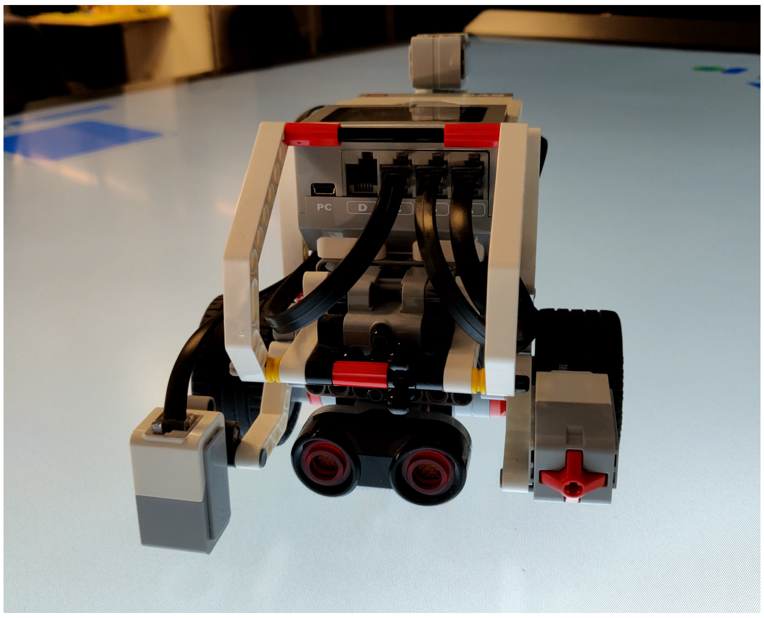


Figure 3.3: Lego Mindstorms robot in the configuration used for Tangible Robotics

EV3's product life span of 7 years at time of writing have led to a large number of programming tools for the robotics kit. It can be programmed with community-developed APIs in languages such as Python, Java, C++ or Swift and many proprietary, graphical environments.[5].
We chose c4ev3 as the programming tool for our interface, a software package specifically for the EV3 that includes an API in the programming language C as well as an uploader and was developed at the TH Aschaffenburg.

3.2 MultiTouchKit

As alluded to before, MultiTouchKit (MTK for short) is a framework that enables the detection of tangibles as well as general touch events on multi-touch displays such as the Microsoft Surface Hub. It extends Apple's SpriteKit[6], which is a framework meant for 2D game creation. As it is based around a tree-structure consisting of two-dimensional assets such as shapes, texts and graphics, it is well suited for our purposes. A program written with the help of SpriteKit has multiple scenes with each scene acting as the root node of a tree-structure. The tree can be filled with specialized nodes such as `SKSpriteNodes` which are nodes that consist of an image loaded onto the screen. Each scene also gives the programmer the ability to specify a variety of callback functions. With these functions, programmers are able to implement program logic that will be executed in between every drawn frame.

The MTK extends on this by providing touch input data from the screen, tangible data and additional callback functions.
It receives information about touch events in JSON format from the multi-touch device, interprets the data and gives access to it. This information includes the position and

[5]http://www.legoengineering.com/alternative-programming-languages/
[6]https://developer.apple.com/spritekit/

time of the touch event as well as its type (e.g., a tap or part of a dragging motion). During processing, it looks for the input pattern that would be created by a registered tangible and keeps track of known tangibles in similar fashion to regular touch events and provides the resulting data to the scene.
In addition, the MTK provides convenient multi-touch enabled graphical interface elements such as buttons, switches or sliders.

3.3 Front End

GRAPHICAL USER INTERFACE:
In the field of computer science, a graphical user interface (short GUI) is a user interface that lets humans interact with a system by manipulating graphical elements. Such elements may be but are not limited to windows, buttons or icons.
In the case of our proposed system, the GUI provides all interaction points that are not related to the tangibles or robot themselves.

Definition: *Graphical User Interface*

The following section will detail the design of the user interface that was written in Swift with the MultiTouchKit and that allows users to create the programs they want the robot to execute.
We will commence by showcasing the GUI elements that are permanently seen on screen. We will then present the seven program block types that were designed in the scope of this paper. Each block type represents one distinct action by the robot such as reading a sensor or making a decision. While seven blocks were designed, only five are featured in the version of the Tangible Robotics interface that was used for evaluation due to performance and reliability concerns with too many passive tangibles at once.

The GUI consists of permanent elements for general functions and program blocks that are dynamically added and removed and are each equivalent to an action of the robot.

3.3.1 Permanent GUI Elements

The main view of the Tangible Robotics interface is a `MTKScene` in full screen mode. The `MTKScene` class is part of the MultiTouchKit. This scene provides the physical space on which program blocks can be created and manipulated. The methods of creation and manipulation are dependent on whether the interface is set to run in *tangible mode* or in *multi-touch mode*. Figure 3.1 also details all permanent elements of the scene when in *multi-touch mode*.

Program blocks are either added through buttons or by 'stamping' them onto the screen with tangibles.

In *multi-touch mode*, new program blocks can be created through labeled, touch-activated `MTKButtons` that each correspond to one type of program block. The buttons are located at the bottom of the screen and split into two stacks that are each positioned roughly where one of two users would stand. Upon being pressed, a new program block of that type is created in the middle of the scene. Blocks can then be moved around the screen by dragging them with a finger.
In *tangible-mode* on the other hand, the role of said buttons is given to the tangibles. When a tangible is initially placed on a part of the scene that is not occupied by another GUI element, a new program block is added at the tangible's position. As each tangible represents one type of program block, the created block is of the type that the tangible represents. Program blocks are moved by placing the tangible of corresponding type on them and dragging it over the screen.

Created program blocks can be removed again by dragging them onto a *trash* symbol in the middle of the lower part of the screen. There are two additional buttons right underneath the trash symbol. The first one compiles and uploads the user-specified program to a connected robot. Feedback on the process' success is shown in a pop-up window underneath the button. The second button reverts the interface back to its initial stage, meaning that it removes all program blocks from the screen.

The last permanent GUI element is an object of the `StartBlockNode` class. While it shares some similarities with other program blocks and therefore also inherits from the same `BlockNode` superclass, there is only ever one on screen, in a fixed place on the left side at middle height. The *start* block is a rectangle with the word 'start' written on it.

Every program starts with the *start* block that is permanently on screen.

If the interface is used with all seven implemented program block types, the *start* block's only other feature is an arrow that sticks out to the right of the block and has a large, triangular head. The head (referred to as *control flow connector* from here on) can be moved around the screen. The arrow's line will constantly connect the block to its *control flow connector*. The start block serves as the entry point of each user-specified program, its *control flow connector* needs to be connected to the first block that the user wants the robot to execute. The details of connecting blocks are detailed in 3.3.2 "Commonalities & the Superclass".

In the version of Tangible Robotics that was used for evaluation however, the *start* block serves a slightly different purpose. When debating which block types to remove with minimal loss in versatility, we noticed that many common robotics use-cases used a single loop in which a sequence of actions was performed. As a consequence, we removed the *loop* block type and instead modified the *start* block to execute the connected program blocks in a loop instead of only once. In order to convey this behavior, we added a round arrow symbol that was pointing at its own tail to the *start* block's *control flow connector*.

Additionally, the user can specify for how many seconds the loop is supposed to run. This was done to prevent the program from running indefinitely. The time can be specified through an *input receptor* as described in detail in 3.3.2 "Commonalities & the Superclass". It manages the integer variable 'seconds'.

3.3.2 Program Blocks

Overview over the program blocks

As mentioned, any user-specified program consists of program blocks of various types. Each block is representative of a distinct action that is defined by its type. Since all blocks have some common requirements such as needing a way for the user to specify an action's parameters, all program blocks are subclasses of one superclass that implements shared elements and logic.
The seven program blocks presented in this study are the *input*, *output*, *wait*, *operator*, *conditional*, *loop* and *variable* block. They are realized through the separate classes `InputBlockNode`, `OutputBlockNode`, `WaitBlockNode`, `OperatorBlockNode`, `ConditionalBlockNode`, `LoopBlockNode` and `VariableBlockNode` which all inherit common features from the `BlockNode` class. The following few sections will go into a detailed description of the commonalities and their implementation in the joint superclass as well as each block's function and unique characteristics.

Commonalities & the Superclass

Features that are shared by all program blocks like parameter input methods or control flow management are implemented in a superclass

The `BlockNode` superclass includes logic that pertains to the base block, control flow management, block mode switching, block output & parameter management and general code generation. All visual elements that are inherited from the `BlockNode` class can be seen in figure 3.4 in the form of an *input* block.

The base shape of each program block is inherited from the `BlockNode` superclass and consists of a square block with triple rounded corners. The same block type graphic that is also printed onto the respective tangible can be seen in the middle of each program block's base.

Blocks connect to each other through *control flow connectors* that dictate the execution order of the implemented program.

Any program's control flow is established through *control flow connectors*, triangular sprites pointing to the right that are connected to the middle of the right side of each

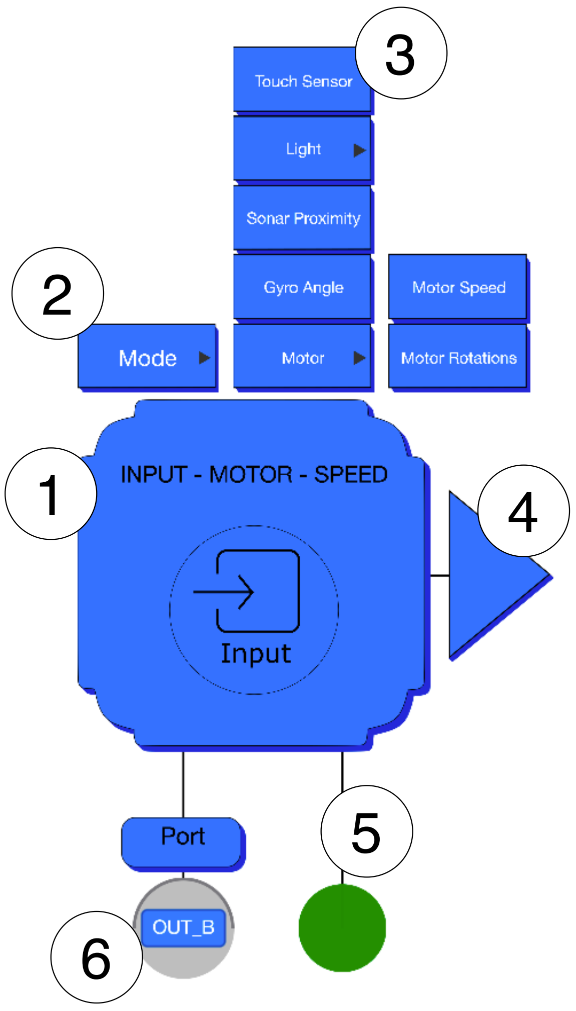


Figure 3.4: An *input* block. Shown elements: 1) base block including current mode name and block type symbol, 2) mode selector button, 3) mode context menu (appears when 2 is pressed), 4) control flow connector, 5) output connector, 6) input receptor

program block through a line.
While a *control flow connector* is dragged around the screen, a gray, transparent box appears to the left of each program block and disappears once the movement is over. The box has about the same height and width as a *control flow*

connector. If a *control flow connector* is dropped onto such a box, it is positioned in the box and the connector's parent block remembers said other block as its successor in the control flow order. Moving the *control flow connector* away from the block resolves that link.
During movement of a block's base, its *control flow connector* either stays in position if it is connected to another block or maintains its position relative to its base block if not.
If a block is removed from the scene, any *control flow connector* that was linked to it is reset to its default position relative to its base block.

Blocks have modes for different actions of the same type. Modes can be changed through a context menu on each button.

To cut down on the number of needed tangibles, similar actions were grouped together into block types and blocks were given a 'mode' that determined which specific action was executed.
A block's mode can be changed through a button labeled 'mode' that is hooked to the left side of the base block's top side. Pressing the button reveals a new context menu that unfolds to the right of the 'mode' button. It consists of a vertical stack of buttons that are each labeled with a respective mode. If a button also contains a small arrow to the right, said mode has submodes that are presented if the button is pressed. By pressing a button that has no arrow, the button's mode is selected and the context menu is hidden again. The context menu also retracts if the block is moved.
A block's mode is also written out above the block type graphic.
An example showing the different modes of an *input* block and the mode selection menu can be seen in figure 3.4.

Actions can be fine-tuned through parameters.

In order to fine-tune an action, the user has to be able to specify parameters for it. A typical example is a motor's speed when told to start turning or the amount of time for which the execution time should be halted by a *wait* block.
In Tangible Robotics, these parameters can be specified through *input receptors*. These consist of a small circle that has an outline around the top half. They are connected to the left side of the bottom of their base block through lines on which a label is situated that states the parameter's name.

In the middle of the *input receptor* there is a button that is labeled with the parameter's current name. Pressing the button brings up an input method that corresponds to the parameter's data type.
Parameters can have the following data types and corresponding input methods:

- *boolean*, which is simply toggled between 'true' and 'false' by pressing the button.
- *string*, which reveals a virtual keyboard in US-layout and input field on screen. Pressing the enter key closes the keyboard and transfers the input field's content into the parameter.
- *integer*, which reveals a virtual number pad and input field that behave analog to the virtual keyboard.
- *operator*, which reveals buttons labeled '==', '!=', '¡', etc. Pressing a button transfers its content into the parameter and removes all buttons again.
- *percentage*, which reveals a touch-sensitive slider that goes from 0 to 100 and a confirmation button.
- *motorSpeed*, which reveals a touch-sensitive slider that goes from 0 to 50 and a confirmation button.
- *colors*, which reveals a set of toggles for the colors that the light sensor can distinguish between and a confirm button. The parameter is the set of colors with an active toggle.
- *ports*, which reveals a set of toggles for the robot's output ports and a confirm button. The parameter is the set of ports with an active toggle.
- *port*, which reveals buttons for each output port. The buttons function the same as the *operator* buttons. Contrary to *ports*, a *port*'s value is only a single button instead of a collection.
- *frequency*, which reveals buttons for some predetermined frequencies that are supported by the 'intelligent brick''s speaker. They function the same as the *operator* buttons.

While the background color of *input receptors* is generally gray, some have specific colors to indicate their data type. *integer*, *percentage* and *motorSpeed* receptors are green, *string* receptors red and *boolean* receptors are blue.
If a block changes its mode, its parameters are changed with it.

Blocks can have output data that can be used as a parameter for other blocks.

For blocks whose action creates an output, such as an *input* block as it reads sensor data, the superclass also provides an optional *output connector*. *Output connectors* have the form of a circle and are connected to the right side of the bottom of their program block. They are slightly smaller than the input receptors and follow the same data type color scheme although in darker shades.
Output connectors can be dragged around the screen by touch gestures and when dropped onto an *input receptor*, they connect to it, creating a data flow from one block to another. They can only connect to blocks that are a direct or indirect successor to them in the control flow tree to ensure that blocks do not receive input from a block that has not been executed yet. The effect of connecting an *output connector* (of block a) to an *input receptor* (of block b) is that the output of block a's action will be used as block b's parameter during execution. The manually entered value of the parameter will be overwritten.
During block movement, *output connectors* behave the same way as *control flow connectors*.
If a block is deleted or its *output connector* is moved off of an *input receptor*, the *input receptor*'s parameter is reset to its prior, manual value. If a block changes modes, its *output connector* assumes its default position relative to the block. Since changing a block's mode also changes its parameters, any *output connector* connected to the block also resets to its default position.

Blocks generate the C code that corresponds to their action.

The last purpose of the superclass is to handle the generation of the actual C code that will be run on the robot. For this purpose, every block possesses the `addCode()` function that calls another `addCodeBlockSpecific()` function. Each program block type overrides the `addCodeBlockSpecific()` function and fills it with block type and mode appropriate logic for program generation as detailed in 3.4.1 "Program Generation". After calling the block specific code generation function, `addCode()` looks for whether the program block has a control flow successor and if it does, it calls the successor's `addCode()` function. The code generation is kicked off by the main scene's *Compile & Run* button which in turn calls the *start* block's `addCode()` function.

Input Block

The *input* block provides sensor data to other blocks.

The *input* block type's purpose is the provisioning of sensor data to other program blocks. The source of the data can be determined through its mode. There is one mode for each connected sensor except the light sensor. The light sensor has three modes, one each for measuring color, reflected and ambient light. Additionally, the motors connected to the 'intelligent brick"s output ports also include sensors for motor speed and rotations. These are available through their own modes.
The *input* block always has an *output connector* for the measured sensor data and only has a parameter if it is in one of the motor modes. In that case, the parameter is of the *port* type and defines the output port from which the data is to be sourced.

Output Block

The *output* block outputs data to connected motors, a speaker, etc.

Any data that is output by the 'intelligent brick' is handled through the *output* block. The present modes are four motor modes and a speaker mode. Modes for display and LED output for the 'intelligent brick' were removed due to the instability they could cause in possible programs.
The four motor modes are 'start', 'stop', 'turn for seconds' and 'turn for degrees'. They share the *ports* type 'ports' parameter for the addressed ports, the *motorSpeed* type 'speed' parameter for the speed that the motor should turn at and

the *boolean* type 'forward' parameter for the turning direction. The 'turn for seconds' and 'turn for degrees' modes also have a respective *integer* parameter for 'seconds' and 'degrees'.

Wait Block

The *wait* block halts program execution.

The *wait* block only has one mode and one parameter, an *integer* type 'time in ms' parameter. It pauses program execution for the given amount of time specified by 'time in ms'.

Operator Block

The *operator* block conducts mathematical and logical operations.

Basic mathematical and logical operations such as multiplication and conjunction can be achieved through the *operator* block. It has modes for each of the four basic mathematical operations 'division', 'multiplication', 'subtraction' and 'addition' as well as for the logical operations 'conjunction', 'disjunction' and 'negation'. In the mathematical modes, it has two *integer* parameters and an *output connector* of *integer* type. The configuration is analog for the 'conjunction' and 'disjunction' modes but with *boolean* data types. The 'negation' mode only has one *boolean* parameter.

Conditional Block

The *conditional* block can make decisions based on a multitude of conditions.

A program is capable of making decisions with the help of *conditional* blocks which check for whether a condition is met or not. They differ from any other blocks in that they have an extension to the right side of their base block as can be seen in figure 3.5. The rectangular addition allows for two additional *control flow connectors* to be connected to the block, one with a green check mark graphic on it and one with a red cross on it. Blocks that are connected to the check mark *control flow connector* will be executed if the condition is met, blocks connected to the red cross *control flow connector* will be executed if the condition is not met. Blocks

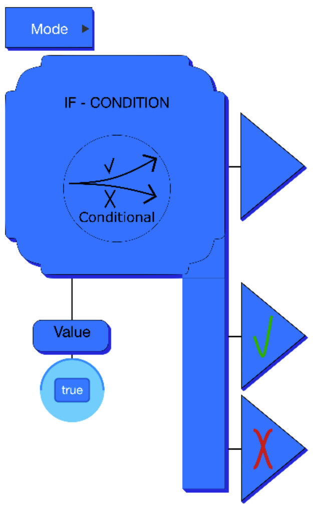


Figure 3.5: A *conditional* block, compare to other blocks, it has an extension to the base block and two additional control flow connectors.

connected to the regular *control flow connector* are executed after the chain of blocks that are connected to the applicable branch of the *conditional* block.

The type of condition that is checked against is determined by the block's mode. It has a mode for direct *boolean* input with a respective *boolean* input as well as a 'compare numbers' mode with two *integer* inputs and an *operator* input that will be used to compare the two *integer* numbers. Furthermore, the *conditional* block has modes for the same sensor data as the *input* block with additional *operator* and *integer* parameters that enable a comparison between sensor data and a value.

Loop Block

The *loop* block allows for the inclusion of loops in programs. As such, before each iteration a loop condition is checked. When a loop iteration is done and as long as said condition is still met, another loop iteration starts. The *loop* block is similar in design to the *conditional* block in that it also has

The *loop* block allows for program sections to be executed multiple times.

the same extension. However, it only has one *control flow connector* for the blocks that the loop should iterate over. Instead, the *loop* block also has an *output connector* with type *integer* that provides other blocks with the number of iterations that have been executed so far.
It has almost the same modes as the *conditional* block with additional 'time' and 'iterations' modes. The 'time' mode receives the overall execution time of the loop as an *integer* parameter in seconds. The 'iterations' mode has three *integer* parameters that determine the first and last value to iterate over and the iteration step size.

Variable Block

The *variable* block provides variable access.

Variable blocks supply the user with a way to store and subsequently access data.
The variable's data type, and whether a variable is stored or accessed, is determined by the block's mode. There are modes for 'reading' and 'writing' *integer*, *boolean*, and *string* variables.
The block has a *string* parameter for the variable's name and a parameter of the type that matches the current mode for data entry when in one of the 'write' modes. When it is in one of the 'read' modes, it has a *string* parameter for the variable's name and an *output connector* of the type that matches the current mode that outputs the variable's value.

3.4 Back End

The C code generated by the program blocks is written to a file, cross compiled and uploaded to the robot.

The back end of Tangible Robotics handles the interpretation of the program block sequence that is connected to the *start* block as well as its conversion into C code. Further, the C code needs to be compiled into an ELF executable for Linux that runs on ARM processors and uploaded to the Lego Mindstorms robot. The code that will be generated heavily utilizes the EV3-API provided by c4ev3. The reason we chose c4ev3 for our back end was that it provided both an API that could access close to all of the robot's features and a simple way of compiling and uploading code to the robot through console commands instead of an IDE. Furthermore, running C code that utilizes c4ev3 can be achieved on the EV3 robot without having to alter its firmware or operating system as is common among other programming languages for the EV3.

3.4.1 Program Generation

Each block writes its specific code to a shared variable and prompts its successor to do the same.

As mentioned in 3.3.2 "Commonalities & the Superclass", the generation of C code is mostly handled by the superclass `BlockNode` and its subclasses. Code generation is kicked off through an interface button and starts with the *start* block. Each program block type overwrites the superclass's `addCodeBlockSpecific()` function that generates the specific code that is unique to the block type. This function will be called by the `addCode()` before handing over code generation the program block's successor if it has one. The generated code is managed by a separate `CFileIO.swift` file that stores the generated C code in a string variable and provides static functions for simple management of the code. It also provides functions for writing the code to a file in order for it to be compiled.

If only five tangibles are used, the *start* block will loop its successors' actions.

As the *start* block is always the first block to execute its `addCodeBlockSpecific()` call, it adds lines to the C code that are necessary for setup, such as setting up the sensor ports with the correct sensors and sensor modes or resetting the motors' rotation count. The EV3-API provides functions for that. If only five tangibles are used and the user-specified program blocks are therefore enclosed in a loop, it also adds said loop in multiple lines of code. While the rest of the program blocks generate their lines of code in a single pass from top to bottom without having to jump up again, the *start* block has to jump up into the loop it created in order to fill it with the code of its successor(s).

Conditional and *loop* blocks call their respective branches to generate their code before handing code generation off to their successor.

Most program block classes' `addCodeBlockSpecific()` function starts off by going through all existing block parameters and checks whether they receive data from an *output connector* or from the manual entry option. It continues with a case distinction between block modes and adds a few lines to the C code that usually consist of calls to the EV3-API as many block actions map well onto actions that the API provides a simple function for.
Since they have additional *control flow connectors*, the *conditional* and *loop* block type behave differently. The *conditional* block for example constructs the condition that is meant to be checked based on the mode and using the mode's parameters.
Next, it adds the header of an 'if' or statement, with the condition integrated into it, to the C code.
If program blocks are connected to the block's 'true' or 'false' control flow connectors, their respective `addCode()` functions will also be called to create the body of the 'if' and 'else' statement. Lastly, the 'if' statement is closed.

Once all successors of the *start* block are finished generating their C code, it is written into a .c file and a set of shell commands is executed that relates to the compiling and uploading process.

3.4.2 Program Execution

The finished program is uploaded and can be executed through the robot's interface.

The finished .c file first needs to be compiled using a cross compiler for Linux on ARM. Carlson-Minot Inc.[7] provides an adaption of CodeSourcery's GNU/ARM toolchain for use with macOS X. After that, the ev3duder tool that comes with c4ev3 can be used to both upload the resulting .elf file as well as make and upload a launcher file that enables the program to be started from the 'intelligent brick''s interface. The upload can be done over USB, Bluetooth or WiFi. In our experience, hot plugging a USB cable proved to be both the most reliable and quickest method.

[7]http://www.carlson-minot.com/available-arm-gnu-linux-g-lite-builds-for-mac-os-x

Chapter 4

Evaluation

Overview

The interface was designed with the intention of testing the usefulness of tangibles as distributed resources in the process of learning to program. Based on the findings of Shi et al. [2019], it can be assumed that among other things, distributing the programming resources among both participants of the pair programming exercise may lead to more time spent planning and more collaboration between users.
Our aim is to find out to what degree these effects can also be observed if the design element of distributed resources is recreated in a very physical way of programming that combines the inherently tangible field of robotics with tangibles. Further, we are also interested in how users interact with the distributed resource depending on the nature of its representation.

The originally intended study could not be executed due to COVID-19.

While a study was designed around this aim, it could unfortunately not be executed in the scope of this Bachelor thesis due to governmental restrictions in light of the COVID-19 outbreak in late 2019 and early 2020. As the study required close interaction between two participants, it was deemed to be too risky with regard to COVID-19's infectiousness. The study's focus on physical interaction also made it impossible for it to be adapted into an online study, which is why we decided to also design and execute an online pre-study.

An online pre-study was executed instead.

The pre-study focuses on evaluating the intuitiveness of different aspects of the tangible-supported programming interface. Its goal is to both improve the interface in general but also to highlight issues that potential users might have with the tangible interaction as they may be relatively unfamiliar with tangibles. Such issues could have the potential to skew the results of the subsequent study.

4.1 Pre-Study

Overview over the pre-study's goals and methods

The pre-study was designed to investigate the intuitiveness of various elements of the interface design. As it needed to be conducted without physical interaction between participants and investigators, it was decided that the study should be conducted over the internet.
For this purpose, a video was created that showcased the interface and all its elements. The tangible-supported variant of the interface was used for the video. Participants then individually watched the video while in a voice call with an investigator. Throughout the video, the participants were instructed to stop the video at predetermined points and were then asked questions about the shown situations. The questions focused on the participants' intuition of how to perform a given task.
The answers to each question were recorded and semantically equivalent answers were grouped together into equivalence classes for analysis. In the subsequent paragraphs we showcase the results of said analysis with a focus on situations in which participants' cognitive model of the interface differed significantly from the actual design. We also reflect on some qualitative feedback given by participants and draw conclusion from the collected data with the target of improving the interface's intuitiveness.

4.1.1 Research Questions

In the pre-study, we were evaluating the intuitiveness of a list of interface elements and design choices through questions tailored to those specific elements. The following paragraph lists these elements and section 4.1.5 "Tasks" details the angle of questioning that we took to evaluate them.

The interface aspects that questioning touched on were:

- the symbols and names representing each block type that are printed onto each tangible
- the creation of a control flow between program blocks
- generic block elements such as a block's input receptors
- handling of unexpected behavior
- block-specific elements such as the three *control flow connectors* of the *conditional* block type

4.1.2 Setup

The pre-study's setup consisted of the participant sitting in front of a desktop or laptop computer and joining a voice call with an investigator through the video conferencing tools Skype or Zoom. The video used 2160p footage and was rendered in 1080p and 30 frames per second to cut down file size.

As the pre-study was conducted online, the setup unfortunately differed slightly between study runs as participants had different computer setups at home. We tried to circumvent potential connection issues by providing the 1080p video in download form rather than as a stream. Additionally, we prompted participants to view the video on a large screen if available and to check their audio levels before starting the video.

The conference tools' respective built-in recording function was used to record the entire study run.

4.1.3 Participants

Collecting demographics of participants

Overall, 13 people took part in the pre-study. They were aged 19 to 57 (M = 23, SD = 12.3), 9 male and 4 female. 6 Participants were studying or working in the field of computer science, an additional 4 had an engineering or mathematics background.

Measuring participants' prior programming knowledge

On a Likert scale of 1 to 5, with 1 'highly disagree' and 5 being 'highly agree', participants answered the question whether they possessed extensive programming knowledge in a range from 1 to 5 (M = 4, SD = 1.4). 5 participants answered with a 1 or 2. When asked to specify their programming experience, answers ranged from never having programmed before over introductory courses to experience in multiple programming language types.

Measuring participants' prior robotics knowledge

On the same Likert scale, when asked if they possessed extensive robotics knowledge, participants gave answers ranging from 1 to 3 (M = 2, SD = 0.8). 10 participants answered with a 1 or 2. When asked to specify their robotics experience, answers ranged from none to having programmed Mindstorms robots in a university course.
The responses to both Likert scale statements are shown in figure 4.1.

4.1.4 Procedure

Introducing participants to the study

Each study run began by the investigator informing the participant about the premise of the study and its connection to the full study. More specifically, the investigator told participants what role the interface would play in the full study and inform them of the purpose of the pre-study.

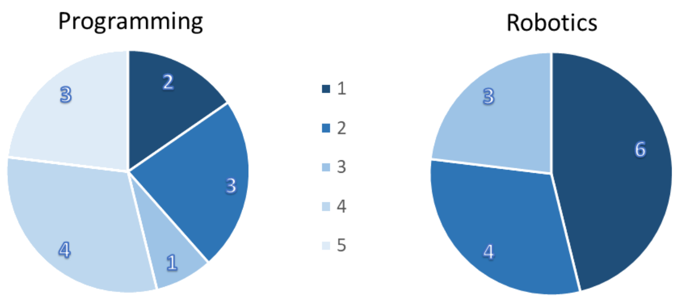


Figure 4.1: Participants' answers to being asked whether they agreed that they possessed extensive programming & robotics knowledge on a Likert scale of 1 to 5 with 1 being 'highly disagree' and 5 being "highly agree'

Consent form and demographic information

Following that, participants were asked to sign a consent form (see A "Pre-study Documents") that included information on the study such as the procedure, its risks and how the collected data would be handled.
In order to be able to contextualize the pre-study's findings, participants were asked to provide some demographic information concerning gender, age, occupation and prior experience in programming and robotics (see A "Pre-study Documents" for the used form). Prior knowledge was determined through participants ranking the statements "Before this study, I possessed extensive programming knowledge" (S1) and "Before this study, I possessed extensive robotics knowledge" (S2) on a Likert scale from 1 to 5 with 1 representing 'highly disagree' and 5 representing 'highly agree'.

Participants watch a video detailing the interface and answer questions.

During the main section of the study run, participants would watch the video and pause playback when the video prompted them to do so. During each pause, the investigator read out the questions relating to the shown situation and the participant gave their answer to them. If participants' answers were unclear, the investigator asked them to further elaborate on them. Participants were allowed to repeat parts of the video if something seemed unclear to them.

All answers to each question were recorded, coded and analyzed.

Once all study runs were completed, the resulting audio material was examined and for each question, all possible answers were collected and coded. Each resulting code represented one group of semantically equivalent answers. Questions 12 and 13 were split into subquestions that each related to one specific design element.
Afterwards, the appropriate codes for each question in each study run were determined. If participants gave answers that included elements of multiple codes, this was also noted and the distinction was made between participants giving multiple codes in one train of thought, participants taking pauses to think of alternative answers and participants withdrawing earlier answers in favor of another.

4.1.5 Tasks

List of investigated elements and why they were chosen.

As mentioned in 4.1.1, the questions that participants answered were split into five categories. These categories, why we chose to investigate them and in which way will be listed in the following paragraph. The full script to the video can be found in A "Pre-study Documents".

- **The symbols and names representing each block type that are printed onto each tangible**
  It is important that users have a clear association of actions they want the robot to take and by which block these actions are represented.
  One question regarding the block types was about asking the user to intuitively pick a tangible they would assume an action to be associated with. Further questions were asking the user to speculate on what actions they would suspect in a given block type.

- **The creation of a control flow on the canvas**
  The control flow of a program, meaning the order in which its statements are executed is key to a

program's function. As such, it should be clear to even new users how to create such a control flow.
To evaluate the interface's method of control flow creation, we asked the participants how they would intuitively manipulate the interface to create an arbitrary control flow. Furthermore, after they were shown how to create such a control flow, they were also asked how they would intuitively edit an existing control flow, specifically on how to add a block in between two other ones.

- **Generic block elements such as a block's input receptors**
  Most program blocks provided by the interface use similar or the same design elements for related functionality. An example for this are the parameters for each block's action that are always represented by circles hanging off the bottom of the block that include a button that initiates manual data entry.
  Due to elements like these being used repeatedly in the interface, users will interact with them on a regular basis, highlighting the need for these elements to be intuitive.
  Participants were asked how they would try to change an action's parameter as well as a block's mode. Additionally, they were asked to speculate on which modes a particular block may have and what effect connecting a block's output connector to another block's input receptor may have.

- **Handling of unexpected behavior**
  Since the early stages of familiarizing oneself with an unknown interface are often of a trial-and-error kind, users are likely to run into behavior that they did not expect. To avoid such situations in the future, it is critical for the user to understand how they occurred in the first place.
  During early interactions of outsiders with prototypes of the interface, it became clear that two such situations that occurred frequently were that users tried to move blocks with the wrong tangible and that users tried to connect a block's output connector to

blocks that were not descendants of said block. The fact that they did not connect usually only became apparent when one of the blocks was moved and the output connector did not move the way it would if connected.
In order to investigate whether participants would be able to understand why these unexpected behaviors occurred, we simulated both these situations in the video and asked participants to explain their cause.

- **Block-specific elements such as the three *control flow connectors* of the *conditional* block type**
  Lastly, the *start* and the *conditional* block have elements that are unique to them. As users have to inevitably interact with the *start* block and the *conditional* block is essential for any branching program, their additional elements should be easy to understand and use.
  To see if this is the case, participants were asked to explain the *conditional* block's necessity for its three *control flow connectors* and the *start* block's design, especially in regard to the symbol on its *control flow connector* and its 'seconds' parameter.

4.1.6 Results

Key figures

In total, 13 participants each answered 16 questions, making for a total of 208 answers. The answers could be categorized into 59 equivalence classes in total with 2 to 7 classes per question (M = 3, SD = 1.4).
The following paragraphs will detail the results of the coding process, present some statistical data about the impact of prior knowledge.
When asked after the study, the majority of participants expressed that they overall found the interface to be intuitive once it and the situations in each question were explained to them. However, many questions were answered in multiple, very differing ways, which is why the last section will go into conclusions that can be drawn from the coding of individual questions.

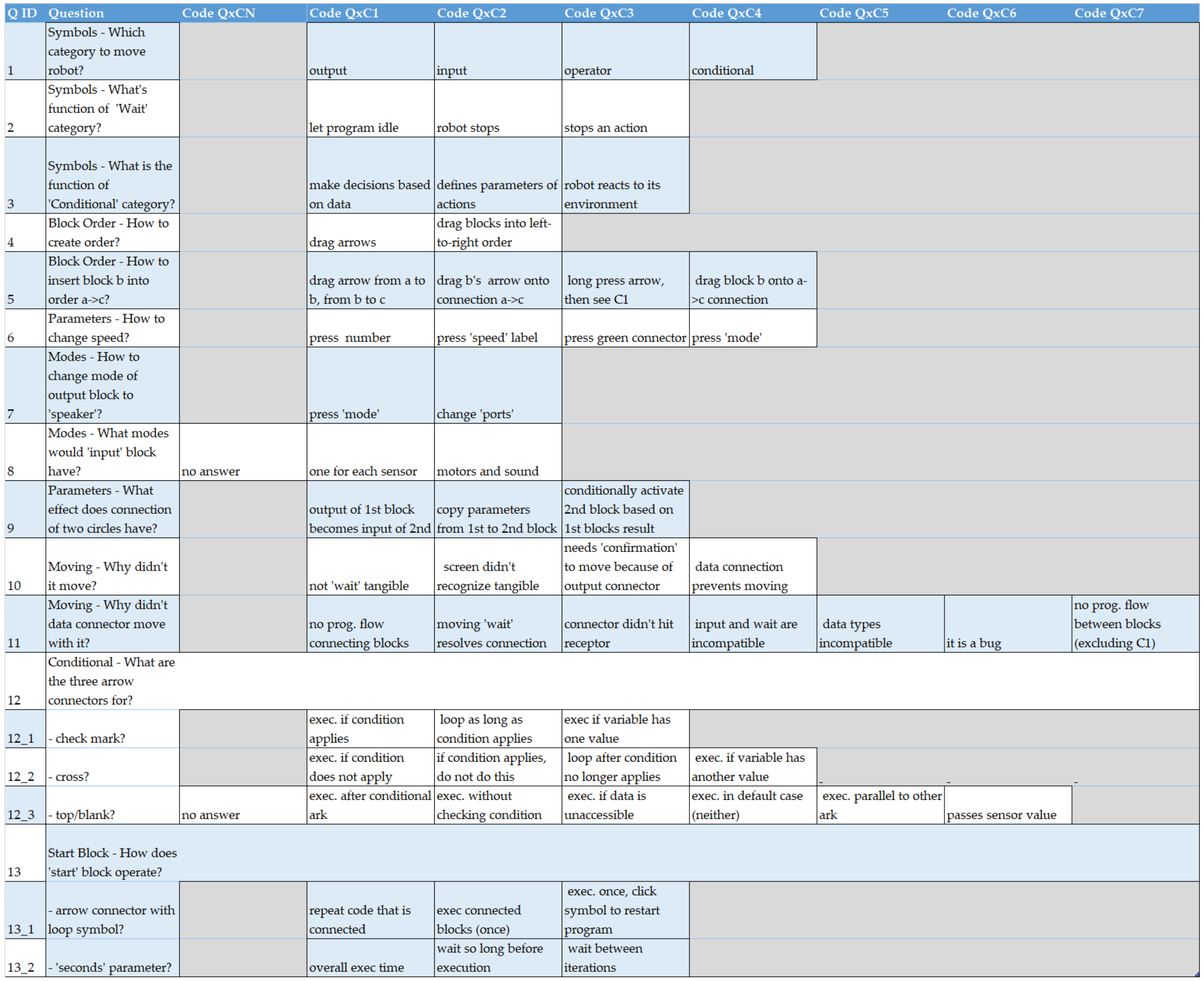
Coded Answers

Coding process

Once all answers for a particular question were examined, the codes representing the equivalence classes were created and each answer was assigned one or more codes. For each question, the code that represented the actual behavior of the interface was designated as *QxC1* with *x* standing for the question's ID, any further answers were assigned ascending codes *QxC2*, *QxC3* and so forth. In two cases, participants were unable to give any prediction as to how to manipulate the interface or what an element represents. In these cases, their answers were given the code *QxCN* for 'no answer'.
All codes can be taken from figure 4.2. The 'Q ID' column indicates the ID of each question as shown in the video script (see A "Pre-study Documents").

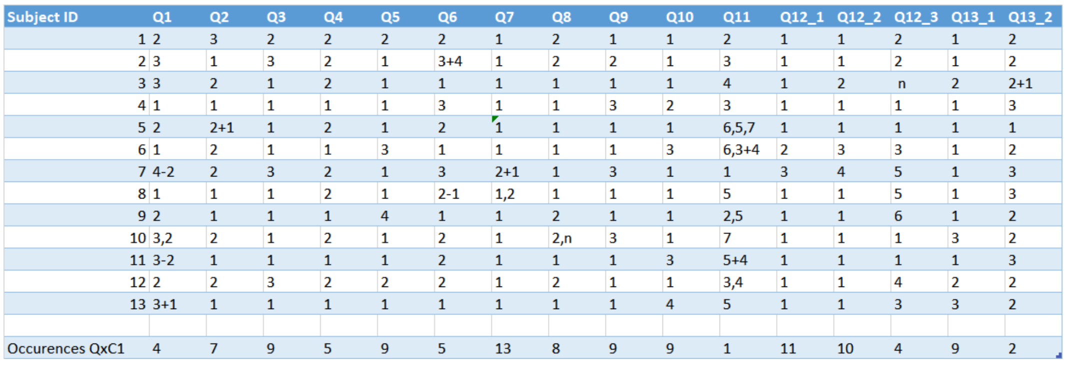
Coding results

Figure 4.3 shows the coding of the participants' answers. It also shows how many answers included *QxC1* for each question, which is the code that was always assigned to answers that described the actual interface's behavior.
Quite clearly, the results for each question differ significantly (*range:* 1-13, M = 8.5, SD = 3.2). While it is to be expected that participants will give very varied answers if asked to predict the behavior of an interface, these results clearly show that some design elements are more intuitively understood than others. Section 4.1.6 "Implications of Individual Questions" looks at individual questions with unusual coding patterns and draws conclusions from the coded answers to these questions.



Q ID	Question	Code QxCN	Code QxC1	Code QxC2	Code QxC3	Code QxC4	Code QxC5	Code QxC6	Code QxC7
1	Symbols - Which category to move robot?		output	input	operator	conditional			
2	Symbols - What's function of 'Wait' category?		let program idle	robot stops	stops an action				
3	Symbols - What is the function of 'Conditional' category?		make decisions based on data	defines parameters of actions	robot reacts to its environment				
4	Block Order - How to create order?		drag arrows	drag blocks into left-to-right order					
5	Block Order - How to insert block b into order a->c?		drag arrow from a to b, from b to c	drag b's arrow onto connection a->c	long press arrow, then see C1	drag block b onto a->c connection			
6	Parameters - How to change speed?		press number	press 'speed' label	press green connector	press 'mode'			
7	Modes - How to change mode of output block to 'speaker'?		press 'mode'	change 'ports'					
8	Modes - What modes would 'input' block have?	no answer	one for each sensor	motors and sound					
9	Parameters - What effect does connection of two circles have?		output of 1st block becomes input of 2nd	copy parameters from 1st to 2nd block	conditionally activate 2nd block based on 1st blocks result				
10	Moving - Why didn't it move?		not 'wait' tangible	screen didn't recognize tangible	needs 'confirmation' to move because of output connector	data connection prevents moving			
11	Moving - Why didn't data connector move with it?		no prog. flow connecting blocks	moving 'wait' resolves connection	connector didn't hit receptor	input and wait are incompatible	data types incompatible	it is a bug	no prog. flow between blocks (excluding C1)
12	Conditional - What are the three arrow connectors for?								
12_1	- check mark?		exec. if condition applies	loop as long as condition applies	exec if variable has one value				
12_2	- cross?		exec. if condition does not apply	if condition applies, do not do this	loop after condition no longer applies	exec. if variable has another value	-	-	-
12_3	- top/blank?	no answer	exec. after conditional ark	exec. without checking condition	exec. if data is unaccessible	exec. in default case (neither)	exec. parallel to other ark	passes sensor value	
13	Start Block - How does 'start' block operate?								
13_1	- arrow connector with loop symbol?		repeat code that is connected	exec connected blocks (once)	exec. once, click symbol to restart program				
13_2	- 'seconds' parameter?		overall exec time	wait so long before execution	wait between iterations				

Figure 4.2: Questions' short forms and all generated codes for each question



Subject ID	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12_1	Q12_2	Q12_3	Q13_1	Q13_2
1	2	3	2	2	2	2	1	2	1	1	2	1	1	2	1	2
2	3	1	3	2	1	3+4	1	2	2	1	3	1	1	2	1	2
3	3	2	1	2	1	1	1	1	1	1	4	1	2	n	2	2+1
4	1	1	1	1	1	3	1	1	3	2	3	1	1	1	1	3
5	2	2+1	1	2	1	2	1	1	1	1	6,5,7	1	1	1	1	1
6	1	2	1	1	3	1	1	1	1	3	6,3+4	2	3	3	1	2
7	4-2	2	3	2	1	3	2+1	1	3	1	1	3	4	5	1	3
8	1	1	1	2	1	2-1	1,2	1	1	1	5	1	1	5	1	3
9	2	1	1	1	4	1	1	2	1	1	2,5	1	1	6	1	2
10	3,2	2	1	2	1	2	1	2,n	3	1	7	1	1	1	3	2
11	3-2	1	1	1	1	2	1	1	1	3	5+4	1	1	1	1	3
12	2	2	3	2	2	2	1	2	1	1	3,4	1	1	4	2	2
13	3+1	1	1	1	1	1	1	1	1	4	5	1	1	3	3	2
Occurences QxC1	4	7	9	5	9	5	13	8	9	9	1	11	10	4	9	2

Figure 4.3: Coding of the participants' answers; **+** indicated same train of thought; **,** indicates pausing to think between codes; **-** indicates revision of an answer

Impact of Prior Knowledge

Since the full study will focus on learning effects in the field of programming, it is of interest how novices in the field of programming and robotics fared in comparison to more experienced participants. For this, we look at the relationship between prior knowledge and an individual's number of answers that included the code *QxC1*.

Prior programming knowledge lead to more success in predicting the interface's behavior.

As can be seen in figure 4.4, participants with more previous programming knowledge generally gave more answers that included code *QxC1*. The Pearson correlation coefficient is 0.68, indicating that these two factors are moderately, positively correlated. It is however noteworthy that participants who answered '1' on the Likert scale concerning prior programming knowledge on average achieved a higher score than those who answered '2' or '3'. Whether this is attributable to outliers could be ascertained by a study run with a larger sample size.

Prior robotics knowledge lead to more success in predicting the interface's behavior.

When looking at the relationship between prior robotics knowledge and occurrences as shown in figure 4.5, there also appears to be a correlation between prior knowledge and occurrences of code *QxC1*, although it seems slightly less pronounced (Pearson correlation coefficient is 0.62).

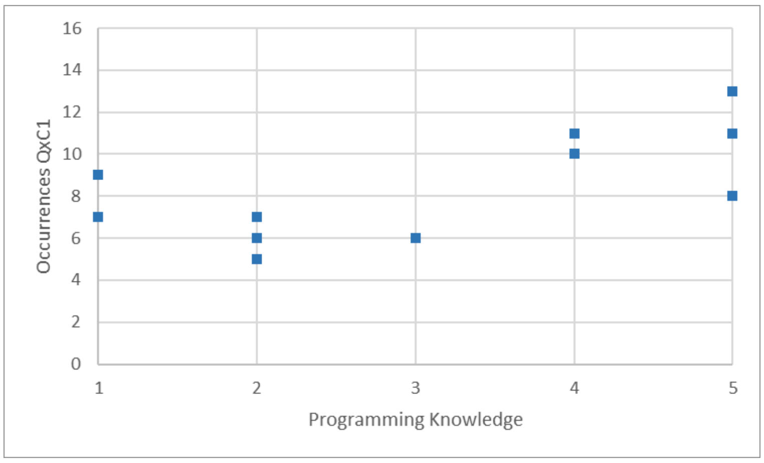


Figure 4.4: Number of answers that included code *QxC1* by self-reported programming knowledge according to S1

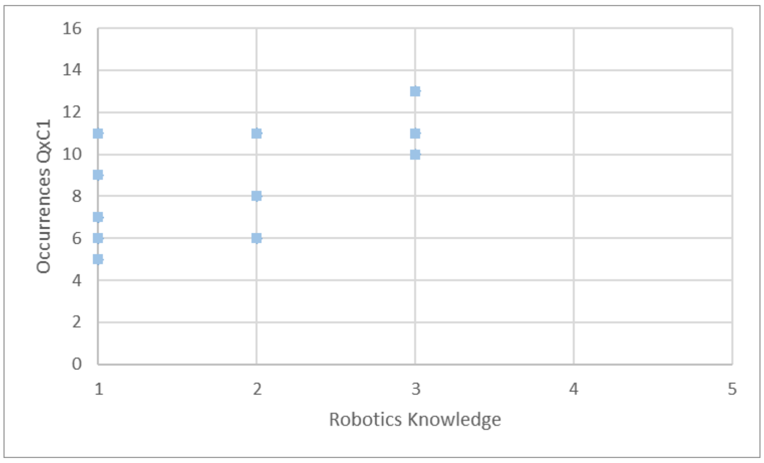


Figure 4.5: Number of answers that included code *QxC1* by self-reported robotics knowledge according to S2

Implications of Individual Questions

Participants were unclear on the meaning of 'input' and 'output'.

Question 1 asked participants to pick which tangible they would expect to use to make the robot move. As the robot's movements are motor-based and the motors are connected to the intelligent brick's output ports, the expected answer was the *output* tangible. Only 4 participants actually answered this way while 6 chose the *input* tangible instead. It is noteworthy that while some participants did give answers pertaining to multiple codes, no answer included both the *output* and *input* tangible. Upon being asked about the rationale behind their choice, the participants answering 'input' all stated that they suspected that they needed to give the motors input.
This confusion was likely caused by the terms *input* and *output* being dependent on one's viewpoint. In this context, the motor's *input* is the intelligent brick's *output*.
Another question that also highlights participants' confusion between the *input* and *output* types was question 8. In it, participants speculated on which modes an *input* block may have. At this point in the video, participants had already received the information that the *input* type of blocks read sensor information and that an *output* block had modes concerning the robot's motors and speaker. Yet still, five participants expected the *input* block to have the same modes as an *output* block. 4 of these 5 participants also scored their robotics knowledge as '1' on the Likert scale. A way of solving this issue may be to redesign the symbols for these two types in a less abstract way that also indicates that the intelligent brick's perspective is being taken. This could be achieved by replacing the rounded square symbolizing a port with a simplified image of the intelligent brick.

The *wait* type can be misinterpreted as stopping the motors.

When asked what behavior the participants expected to be represented by the *wait* tangible in question 2, about half of the participants correctly assumed it would interrupt the current program's execution. However, almost as many expected it to stop the robot for a given time as well. Since an already running motor is not affected by the executed program until it is addressed by an *output* block*output* block again, motors do not halt while the program execution is halted. As the video shown to participants did not go into the concrete semantics of each program block, participants were not expected to know this. It does show however, that the distinction between halting the program execution and the robot does not come naturally to users who are new to robotics and even those that estimated their robotics knowledge at a higher level (Pearson correlation coefficient of 0,21 for the relationship between participants' self-reported robotics knowledge and whether their answer included code *Q2C1* or not).During the interface introduction in the full study, investigators should pay special attention to explaining this behavior.

Some participants found it more intuitive to create control flow by ordering blocks.

Another question with notable results is question 4, in which participants were asked how they would instinctively try to create a control flow among blocks. In the interface, this is achieved by dragging the *control flow connector* from one block to an area to the left of another program block that is highlighted during any flow connector's movement. 5 participants said that they would drag the *control flow connector* while the remaining 8 stated that they would try to move the blocks themselves and order them from left to right by desired execution order. While this is currently not supported by the programming interface, it may be a worthwhile addition as this method of control flow creation can work alongside the existing one. The same goes for the method of dropping a program block into an existing control flow that is described by code *Q5C4*.

To change parameter values, participants would press elements other than the button that contains the value.

A similar situation is revealed by question 6, which had participants speculate on how to manually enter a value into a program block's parameter receptor. Only 5 participants intuitively would have tried to press the display of the current value which is embedded into a button that brings up an input method depending on the parameter's type. In contrast, 9 participants (including one that later revised their answer) intuitively would have pressed the receptor that encloses the button or the label that describes the parameter. One participant who stated that they would press the label mentioned that the label's shadow made it look like a button that could be pushed in. Designing a flatter label graphic may fix this. Alternatively, either or both the receptor and button could be made to also bring up the correct input method provided that no output connector is connected to the input receptor.

In cases of predictable user error, the interface needs to be more informative.

Question 11 is both the question with the least amount of participants answering in accordance with code *QxC1* and the question with the most identified codes which warrants further investigation into participants' answers.
The video first showed how someone tried to connect the output connector of a block *a* to an input receptor of another block *b* that was not a direct descendant of *a*. Due to said connection not existing, the output connector did not snap into place and connect with block *b*'s input receptor. Later, when block *b* was moved, unlike how a connected output connector was shown to behave, the output connector of block *a* did not move with the input receptor of block *b*. Participants were asked to explain this behavior.
Only one participant was able to identify the issue with other participants having very varied answers. The most commonly mentioned suspicion was that the connection did not form due to incompatible data types. It is noteworthy that this answer was only given by participants with self-reported programming knowledge of 4 or above, indicating that this theory likely draws from prior programming experience. Other somewhat common answers conjectured that the blocks themselves were not compatible with each other or that this was a bug.
This and the fact that the amount of answers that included more than one code was higher than for any other question, suggests that users should be provided further feedback when trying to establish a data connection between blocks that do not have a control flow connection. This could be achieved by drawing the user's attention to the current program's control flow, for example by letting block *a*'s *control flow connector* wiggle up and down a few times.

The purpose of a *condtional* block's regular *control flow connector* is unclear.

When confronted with the *conditional* type of blocks in question 12 and asked to explain its design, participants were generally successful in guessing the function of the two *control flow connectors* that relate to the *true* and *false* ark of the conditional. However, only 4 participants identified the *control flow connector* situated at the top of the block as the connector that dictates further program execution after the respective conditional set of actions has finished execution. There was only a weak correlation between giving answers including code *Q12_3C1* and self-reported programming knowledge (Pearson correlation coefficient of 0.24), indicating that even the more experienced participants struggled with this interface element. When asked for feedback, 2 participants reported that they would have found the *conditional* block more intuitive if said *control flow connector* was protruding more to the right than the two other connectors.

The *start* block's 'seconds' parameter is misleadingly named.

Finally, participants were asked to speculate on the nature of the *start* block. It forms the entry point of every program, has a *control flow connector* with an image of an arrow forming a loop on it and an integer parameter that is labeled 'seconds'. Participants were generally able to identify the *control flow connector*'s function successfully with only 4 participants either overlooking or misinterpreting the arrow symbol as a button that would restart a running program.

The 'seconds' parameter determines the overall runtime of a program before it gets terminated. When asked about how the parameter is to be understood, however, 8 participants answered that it was a delay before the program's execution. Only 2 participants' answers hinted at code *Q12_2C1*. This suggests that the parameter's label 'seconds' may be too ambiguous, which is supported by one participant suggesting to rename it to 'timeout'.

4.1.7 Discussion

Summary of the pre-study's conclusions

The pre-study's goal was to identify interface elements that were unintuitive to users that were completely unfamiliar with the interface. This was done in preparation for the full study. While participants in the full study will get an introduction to the programming interface beforehand, these findings will help make the interface more easily understandable and therefore minimize the risk of participants being distracted by complications with the interface.

During the pre-study, we found that some graphical elements of the interface such as labels, parameter names or symbols caused confusion among participants and may benefit from design changes. We also found possible shortcuts in control flow manipulation that could enable a more intuitive interaction.

Limitations of the pre-study.

Even though we consider the pre-study to be a success due to the many design cues derived from its data, we did run into limitations during its execution. A higher number of participants for example would improve the confidence in our findings. This is especially true with regard to unrepresented groups, such as participants with very high robotics knowledge (no participant reported their knowledge as higher than 3 on the Likert scale) or robotics knowledge that they rated higher than their programming knowledge.
Furthermore, evaluating a touch interface remotely without participants having the possibility to manipulate it, limited us to the point of only being able to evaluate how participants would initially interact with the interface. Any further interaction once participants had followed their first instinct could not be investigated. It would have been especially insightful if we would have been able to see how participants would react if their initial reactions did not lead them to their desired outcome.

4.2 Study Protocol For Full Study

Overview over the full study

As mentioned, the programming interface was designed in order to investigate the effects of using tangibles as a distributed resource in the context of a pair programming learning environment. For this, the tangible-enabled interface has to be compared to a multi-touch only interface, which is the conventional input method in the context of large multi-touch tabletop displays.
For the study, a second interface was designed that differs in not using the tangibles. It recreates their block creation functionality with on-screen buttons that create a new block in the middle of the screen and their tangible-specific block movement functionality with simple finger-dragging.

The differences that we want to quantify include how much time both subjects spend actively engaging with the interface compared to taking a supportive role for their partner. Furthermore, we will measure time spent in the conventional stages of solving a programming problem: 'problem understanding', 'planning', 'implementation' and 'testing and debugging'. Lastly, we plan to quantify the user experience and subjective advantages of one system over the other as perceived by the users and participant interaction with the distributed resources.

Unfortunately, the user study's execution was canceled in the light of the COVID-19 virus outbreak in late 2019 and governmental restrictions taking effect in early 2020. Nevertheless, the study's hypotheses and methodology are described in detail in the following section for future reference in hopes that the study can be executed at a later date.

4.2.1 Hypotheses

In the distributed-resources scenario, we will aim to disprove the following null-hypotheses:

H1 The difference in how much time each user spends actively engaging the interface will not decrease.

H2 Users will not spend more time in the 'planning' stage of programming.

H3 Users will not spend less time in the 'testing and debugging' stage of programming.

H4 There will not be a quantifiable difference in reported user experience and resulting improvement of self-efficacy.

H5 There will not be a perceivable difference in how users interact with distributed resources.

4.2.2 Methodology

Independent & Dependent Variables

Independent:

I1: The method of placing and moving a program block.

The first method will be using on-screen buttons for placing a new program block and finger-dragging on a block for its movement. The buttons will be divided in two groups. Each group will be on screen in front of one user with the users standing next to each other. The first group (placed at the bottom of the screen to the left) will include the 'Output' and 'Wait' buttons, the second group (to the right) the 'Input', 'Operator' and 'Conditional' buttons. This distribution was chosen so that for a possible solution

You either use on-screen buttons or tangibles to add blocks to the canvas. Each tangible or button represents one block type for each task, roughly the same number of blocks from each group would be required. Additionally, blocks from the right group would be required at the beginning of the program and vice versa.

The second method will be using passive tangibles that each correspond to one type of program block. New blocks will be created by placing the tangible on a free space on the table. Existing blocks will be moved by placing the tangible that corresponds to the block's type onto the block and dragging it. The tangibles will be divided into the same groups that the buttons were divided into. The first group will be placed in front of the left user and the second group in front of the right user.

We will measure engagement time for each partner and time spent in programming stages.

Dependent:

D1: The amount of time that each partner spends actively engaging the interface.

It is considered an active engagement if the user directly manipulates interface elements (using the table's touch input or a tangible to place, move or remove a program block, using touch input to modify a block's mode or using touch input to modify a block's input or output variables) or indirectly interacts with the interface (pointing, discussing interface elements or possible solutions). Time will be measured in seconds

D2: The amount of time spent in each programming stage.

The stages are 'problem understanding', 'planning', 'implementation' and 'testing and debugging'. Time will be measured in seconds.

Tasks

The users will be provided with an assembled Lego Mindstorms EV3 robot that is equipped with three motors, a sonar sensor, a touch sensor, a gyroscope sensor and a light sensor. The programming interface will provide them with five program block types either through five on-screen buttons or five tangibles. The provided blocks will be:

Participants will program the robot with actions of five types.

- an 'input' type, capable of reading the robot's sensors and passing their value along to other blocks
- an 'output' type, capable of addressing the robot's output methods such as its motors or speaker
- a 'wait' type, allowing the robot to wait for a given time
- an 'operator' type, providing basic mathematical and logical operations and outputting the result to other blocks
- a 'conditional' type, allowing the robot to make decisions based on values passed along from other blocks and sensor inputs.

While additional *variable* and *loop* blocks were implemented in software, it was decided to not include them in the study due to concerns about the MultiTouchKit's lowered tangible detection reliability and performance if used with more than five passive tangibles at once as well as concerns about overburdening users who may be completely new to programming.
Since this limitation decreases the range of writable programs, the tasks were chosen to be easily performable without these additional blocks.

The tasks that will be performed

They will use the developed programming interfaces to perform two tasks:

T1: Program the robot to move forward. Every second, it should make a sound based on its distance to an object in front of it. The closer the object is, the higher the pitch should be.

T2: Program the robot to drive on the table without falling off.

For each task, they will have 20 minutes.

Participants

The participants of the study will be students with no to little prior experience with robotics, the Lego EV3 in particular and the programming interfaces provided by Lego.

Experimental Design

The study will follow a between-subjects design. Half of the user pairs will perform T1 without tangibles and T2 with tangibles. The other half will be using multi-touch only for T1 and the tangibles for T2. It will be decided at random which pair will perform which task with tangibles.

Experiment Setup

The programming interface will be displayed on an 84-inch Microsoft Surface Hub at its native resolution of 3840x2160. The Surface Hub is oriented horizontally and supports up to 100 touches at once. Its touch inputs are transmitted to a virtual machine running Microsoft Windows 10 on an Apple iMac Pro via USB. The iMac also runs the software that provides the programming interface.

The setup is located in a corner of a shared research space. The users will be standing next to each other on one of the long sides of the display. Figure 4.6 shows two users using the interface in multi-touch mode as they would during the study.

The tangibles consist of a square baseplate and a round, centered handle on top, both made of plywood. On the underside, each tangible has three soft, conductive pads which are connected to each other with copper foil that is stuck to the underside of the baseplate. On each tangible, the positions of the pads form a relation unique to the tangible. This relation is used to identify the tangible based on the touch inputs that the pads are creating.

The robot is connected to the iMac via USB in order to export the created program onto it and then unplugged in order to run the program without the physical limitation of a USB tether.

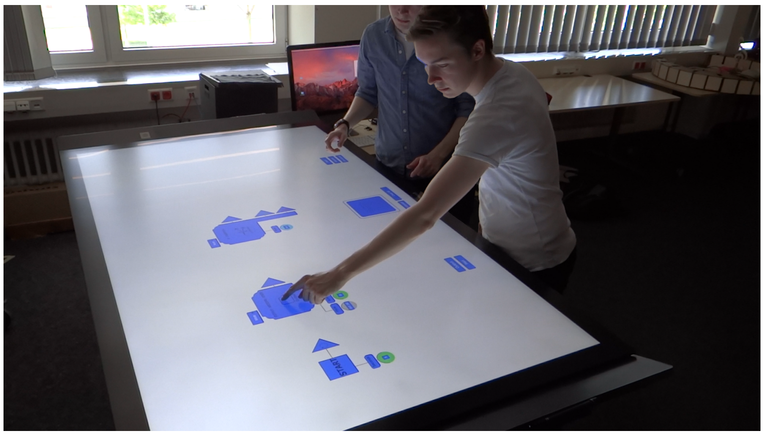


Figure 4.6: Two users at the study space in front of a Microsoft Surface Hub using Tangible Robotics in multi-touch mode.

Experiment Procedure

Introducing participants to the study and the interface

Before the experiment can begin, participants will be asked to fill out an informed consent form (see B "Study Documents"). The experiment will commence with the instructor guiding the users to their designated places and a short introduction to the robot by the instructor. In this introduction, the instructor will explain the robot's sensor and motor configuration (i.e., port connections and attached sensors).

Afterwards, the programming interface will be explained. The following points will be included in the explanation:

- Each block has a connector that connects to another program block when dragged into an area to the left of it. The second block's action will be executed after the first block's action.
- Some blocks have an output connector that can pass the output of the block's action along to another block.
- Each block can have input receptors that allow setting

parameters for the block's action. Each input receptor has a manual input method corresponding to its input type (integer number, color, etc.) and can accept another block's output if the latter's output connector is dragged onto the input receptor.

- Some blocks have different modes that determine their action and come with their own set of inputs and potentially an output.
- These is a 'start' block that marks the start of the program and will loop the user-definable part of the program. The block has a connector to mark the start of the user-defined part as well as an input receptor that determines how many seconds the program will loop for.
- The block types as mentioned under 4.2.2 "Tasks".

Depending on which input method will be used for task T1, the introduction will proceed with an example program in order to demonstrate how to add a new program block, how to move or delete an existing one as well as how to plug the USB cable into the robot and start the generated program on the robot.

Participants work on task T1 first, then switch input methods and work on T2.

Once any further questions are answered, the instructor will read out task T1 to the users and allow them to start working on it.
After the first task is completed or the specified time is up, the input method will be changed and quickly explained and the second task T2 will be read out.
Participants will be given a data sheet (see B "Study Documents") with the most important technical information about the sensors and outputs.
During the users' work on the tasks, the instructor will be available for questions regarding the robot's sensor and motor configuration, the data types of input receptors and output connectors and the description of program block modes. Questions concerning the behavior generated by possible block sequences will not be answered.

Questionnaire and demographical information

To conclude the study run, the users will answer a questionnaire (see B "Study Documents") about their experience with both systems. For each system, they will rate the following statements on a Likert scale from 1 to 5 with 1 representing "highly disagree" and 5 representing "highly agree":

- The interface was easy to understand.
- Using the interface was frustrating.
- I was able to easily translate my intentions into actions.
- My understanding of programming has improved.
- Working in a pair made me more confident in our program.

Furthermore, the questionnaire will have a field for comments.
We will also collect demographic information through a form (see B "Study Documents"):

- Age
- Gender
- Course of studies
- Prior programming knowledge ("Before this study, I possessed extensive programming knowledge" on the same Likert scale as above and a free field for a short description)
- Prior robotics knowledge ("Before this study, I possessed extensive robotics knowledge" on the same Likert scale as above and a free field for a short description)

Data Analysis Methods

Each user session at the table will be recorded both in video and audio form. The camera will be positioned in a way that allows for both users' hands to be seen at most times, especially during interactions with the display. The instructor will note whether a task is fulfilled in case the camera does not record the robot during execution.

The recordings will then be transcribed, and a coding scheme will be applied to each line of dialog. The coding scheme's codebook will consist of the four stages of programming as mentioned in 1.1. and each line will be assigned one or no code. The transcript's time stamps will then be used to identify the time spent in each programming stage.

Furthermore, a second coding scheme that identifies user interactions will be applied. Its codebook will consist of 'no active interaction', 'active interaction by user 1', 'active interaction by user 2' and 'active interaction by both users'. Again, the time spent in each state will be determined through time stamps.

For each answer to the questionnaire, the mean value and standard deviation will be calculated to allow for comparison between both interfaces.

Lastly, field notes and the video recordings will be used to analyze participants' interactions with the distributed resource. Special attention will be given to whether participants only use the resources that were implied to be 'theirs', whether they will use each others' resources or whether they will ask their partner to use them.

Chapter 5

Summary and future work

The following chapter will conclude this thesis. We commence by summarizing our work and contributions and end by giving a prospect on how our work can be extended in the future.

5.1 Summary and contributions

We began our work by studying existing methods of easing the barrier of entry into programming. The tools that we identified as being common in the field and in research were visual programming languages, robotics as a field of application, tangible user interfaces and pair programming. Pyrus[2019] showed that pair programming can benefit from the constraints of 'distinct actions', 'distributed resources' and 'enforced turn-taking' even though these restrictions may also cause frustration.

'Distinct actions', 'distributed resources' and 'enforced turn-taking' help structure pair programming better.

We then set out to design and implement our own programming interface designed for novices based on these findings. Our interface presented a visual programming language on a large tabletop multi-touch display where each element was mapped to a discrete action by a robot.

The proposed study investigates the effects of applying 'distinct actions' and 'distributed resources' through tangibles.

In our study design, we proposed to distribute the resources needed for successful task completion among both users. In the study, we would choose the types of program block and therefore the types of actions that the robot can perform as our distributed resource. We would implemented this setup through two separate testing conditions. In the first condition, we would use tangibles as physical manifestations of each block type. For the second condition, we would instead use on-screen buttons for each type. The intent of the study would be to explore whether the design aid of 'distributed resources' can benefit from an implementation using tangibles.
Further, while the resources would be distributed in both scenarios, we did not plan on enforcing strict 'possession'. Instead, we would hope to observe if there are differences in how users interact with a distributed resource if it is a real world object instead of a virtual, on-screen one.

The performed pre-study helped us uncover some shortcomings in our design.

Due to COVID-19, we were unable to perform said study and performed a pre-study instead. The goal of the pre-study was to evaluate our proposed interface's intuitiveness. We found several instances where participants' intuitive assessment of a situation or interpretation of a design element did not coincide with reality and drew conclusions from each of these occurrences as to how the interface could be further improved.

5.2 Future work

Performing the study

While the current situation prevented our planned study from being executed, this could obviously be done in the future once collaborative in-person user studies can be executed again without serious safety concerns.

Replace PUCs with active tangibles

When looking purely at the proposed interface, the results of the pre-study and the resulting, concrete solutions to the

interface's issues could be applied.
Further, the interface could benefit from replacing the passive PUCs with a variant of tangible that is less restrictive when it comes to the number of tangibles that can be tracked. That way, all seven designed program blocks could be used.
An example for such tangibles are PERCs (Voelker et al. [2015]), which are active derivatives of PERCs that use a field and a light sensor for tracking whether they are placed onto a screen and transmit said data to the application. Based on when a PERC is reporting its placement and when the screen detects touch points, it can identify tangibles uniquely. Using this technique of identification is more reliable than using the touch points' relationship to each other and therefore allows for more tangibles to be used.
Additionally, some blocks could be extended through more modes. The input block, for example, could be made to support old Mindstorms NXT 2.0 sensors or the sensor port could be specified as a parameter to support multiple sensors of the same type.

Transfer game-design restrictions to other TUIs

Provided that our proposed tangible-supported interface shows to have advantages over traditional multi-touch, especially with respect to the effect of using tangibles as distributed resources, it could be worthwhile to also investigate how well the concepts of 'distinct actions' and 'distributed resources' translate onto other tangible user interfaces. In this regard, it would also be interesting to look into an approach to tangible programming where the tangible object and therefore distributed resource would be the program blocks and not the block types.

Appendix A

Pre-study Documents

On the following pages, you will find the video script, consent form and demographics form utilized in the pre-study.

Video Script

Voice over in blue

Setup:
Screen showing a small, finished program

Introduction:

- This is a programming interface that will be used by two users
- It will be used to program a robot

  Show robot
- The robot has ways to both output and input information
- The output methods that will be important to us will be…

  Closeup speaker grill

  sound output through a small speaker…

  Closeup motor

  … and motors…

  Closeup output ports

  That are connected to the robot's brain through cables

  Closeup of front of robot, showcase each sensor quickly with finger, colored paper, hand
- Furthermore, the robot can read information about its surroundings through sensors such as a touch sensor, a light sensor or a proximity sensor
- You will piece together your program out of predetermined blocks
- Each block of your program represents one action for the robot

Fade to crop of interface in its initial state, i.e. no tangibles or blocks visible
Tangibles

- You add blocks by using the tangibles as stamps on the screen

Show 'Wait' tangible to camera, place on table and take away

- There are five tangibles, each represents its own category of program blocks
- The five tangibles are…

Cut to blank background (not table)
Place each tangible into frame upon introduction

- The input tangible
- The output tangible
- The wait tangible
- The conditional tangible
- The operator tangible

Pause
Questions:
Q1: Given these symbols, which category would you pick if you wanted to let the robot move forward?
Q2: What would you expect a 'Wait' block to do?

Q3: What would you expect a ‘Conditional’ block to do?
Continue

Fade to shot of table, show to camera and place each tangible on screen from left bottom to right bottom in half circle when talked about
Blocks:

- The input block reads sensor information
- The output block gives control the robot’s outputs, meaning its motors and speaker
- The wait block lets the robot idle for a given time
- The operator block provides basic mathematical operations
- The conditional block allows the robot to make decisions based on e.g. previous data or sensor input

Pause
To create a working program, it is not just important which actions are performed but also that the robot knows in which order to perform them.
Questions:
Q4: Looking at the current screen, how would you create such an order?
Continue

Connect the program blocks in a row except ‘wait’ block in the middle
Each block has an arrow that points to its successor which is the block that will be executed after the first block.

Pause
Questions:
Q5: If you wanted to insert the unconnected ‘wait’ block in the middle between the ‘output’ and ‘operator’, how would you do that?
Continue

Fade to closeup of output block
Depending on the block type, there are many parameters to fine tune the action. This output block for example starts one or more motors. As you can see, you can specify which motors should turn, the speed they turn at and whether they turn forwards or backwards.

Pause
Questions:
Q6: How would you try to change the speed of the motor?
As already said, the output block gives you control over both the motors and the speaker. Right now, this block only starts motors.
Q7: Do you have an idea on how to access the speaker?
Continue

Press ‘Mode’, press ‘Play sound’
Fade to closeup of input block & output block
Drag data connector to ‘Output’ block

Some blocks on the other hand provide other actions with data, for example the input block. It provides sensor data in form of another connector that can be connected to one of its successors' parameters.

Pause
Questions:
Q8: What modes would you expect the input block to have?
Q9: Can you speculate on what effect connecting the two circles of the 'Input' and 'Output' block has?
Continue

Move 'Output' block with the tangible
Blocks can be moved by placing the tangible that corresponds to their type onto them and dragging it.
Move 'Output' block onto trash, connections of 'Input' block are reset
Moving a block onto the trash symbol deletes that block from the screen.

Move 'Input' data connector onto 'Wait' block (does not connect)
Try moving 'Wait' block with 'Output' tangible (does not work)
Switch to 'Wait' tangible, move 'Wait' block (data connector stays in place)

Pause
Questions:
Q10: The first time I tried to move it, it didn't work. Do you have an idea why?
Q11: The second time, the data connector from the other block didn't move with it, why could that be?
Continue

Fade to 'Condition' block without anything around it, change mode to proximity
Let's take a closer look at the 'Condition' category of blocks. As mentioned, it gives the robot the ability to make decisions based for example on sensor input.
Pause
Questions:
It differs from the other blocks in that it has three arrow connectors.
Q12: Can you tell me why it needs them?
Continue

Fade to closeup of 'Start' block
Lastly, there is the 'Start' block.
Pause
Questions:
Q13: Based on what you see, can you tell me how the block operates?
Continue

Fin

Informed Consent Form

Evaluating the performance of a robotics programming interface that uses tangibles on a multi-touch table.

PRINCIPAL INVESTIGATOR	Till Bußmann, Media Computing Group, RWTH Aachen University, Email: till.bussmann@rwth-aachen.de

Purpose of the study: The goal of this study is to evaluate the intuitiveness of a robotics programming interface that uses tangibles on a multi-touch table. The results will help improve said interface.

Procedure: Participation in this study will involve the participant watching a video that explains the interface. The video includes markings at which the participant will stop the video and answer questions about a situation depicted in the video. During the whole participation, voice will be recorded for the purpose of later analysis.

Risks/Discomfort: Even though the study is expected to last no longer than half an hour, you may become fatigued during the course of your participation in the study. Feel free to take as many breaks as necessary during the study. There are no risk associated with the participation in the study. Should completion of the task become distressing to you, it will be terminated immediately.

Benefits: The results of this study will be useful to improve the programming interface that will later be used in another study in which it will be compared to a traditional multi-touch interface.

Alternatives to Participation: Participation in this study is voluntary. You are free to withdraw or discontinue the participation.

Cost and Compensation: Participation in this study will involve no cost to you.

Confidentiality: All information collected during the study period will be kept strictly confidential. You will be identified through identification numbers. No publications or reports from this project will include identifying information on any participant. If you agree to join this study, please sign your name below.

Audio recordings: You will be audio recorded during the study. The recordings will not be made public and only be used for study analysis. You are free to withdraw or discontinue the participation at any time, if you don't want to be recorded.

O I have read and understood the information on this form.
O I have had the information on this form explained to me.

Participant's Name	Participant's Signature	Date
	Principal Investigator	Date

If you have any questions regarding this study, please contact Till Bußmann. E-mail: till.bussmann@rwth-aachen.de

Demographic information

Age: ________

Gender: ________________

Course of studies/ field of work: ____________________________________

Please rate the following statements about yourself on a scale from 1 to 5, where 1 means “highly disagree” and 5 means “highly agree”:

Statement	Highly disagree				Highly agree
	1	2	3	4	5
Before this study, I possessed extensive programming knowledge					
Before this study, I possessed extensive robotics knowledge					

Please specify any prior programming knowledge:

Please specify any prior robotics knowledge:

Appendix B

Study Documents

On the following pages, you will find the consent form, demographics form, questionnaire and data sheet designed for the full study.

Informed Consent Form

Evaluating the performance of tangibles as a distributed resource in an educational robotics environment

PRINCIPAL INVESTIGATOR	Till Bußmann, Media Computing Group, RWTH Aachen University, Email: till.bussmann@rwth-aachen.de

Purpose of the study: The goal of this study is to find out how tangibles as a distributed resource can influence the learning process in novice robotics programming. Participants will create programs using two different interfaces and run them on a robot.

Procedure: Participation in this study will involve two tasks that will be performed after each other. Two participants will work on both tasks together. After the first task, the participants will be given a form that gathers demographic information. The participation will be closed by a closing questionnaire, which includes questions about both interaction methods. During the whole participation, voice and video will be recorded for the purpose of later analysis.

Risks/Discomfort: Even though the study is expected to last no longer than one hour, you may become fatigued during the course of your participation in the study. Feel free to take as many breaks as necessary during the study. There are no risks associated with the participation in the study. Should completion of the task become distressing to you, it will be terminated immediately.

Benefits: The results of this study will be useful to further the understanding of distributed tangibles' influence on robotics learning.

Alternatives to Participation: Participation in this study is voluntary. You are free to withdraw or discontinue the participation.

Cost and Compensation: Participation in this study will involve no cost to you. There will be snacks and drinks for you during and after the participation.

Confidentiality: All information collected during the study period will be kept strictly confidential. You will be identified through identification numbers. No publications or reports from this project will include identifying information on any participant. If you agree to join this study, please sign your name below.

Video recordings: You will be video recorded (image and sound) during the study. The recordings will not be made public and only be used for study analysis. You are free to withdraw or discontinue the participation at any time, if you don't want to be recorded.

O (Optional): I agree to a publication of a short video clip or a still photography in the written thesis, thesis presentation (usually made public on the lab's website), scientific paper or article.

O I have read and understood the information on this form.
O I have had the information on this form explained to me.

Participant's Name	Participant's Signature	Date
	Principal Investigator	Date

If you have any questions regarding this study, please contact Till Bußmann. E-mail: till.bussmann@rwth-aachen.de

Demographic information

Age: ________

Gender: ________________

Course of studies/ field of work: ____________________________________

Please rate the following statements about yourself on a scale from 1 to 5, where 1 means "highly disagree" and 5 means "highly agree":

Statement	Highly disagree				Highly agree
	1	2	3	4	5
Before this study, I possessed extensive programming knowledge					
Before this study, I possessed extensive robotics knowledge					

Please specify any prior programming knowledge:

Please specify any prior robotics knowledge:

Questionnaire

Tangibles:

Please rate the following statements about yourself on a scale from 1 to 5 in regard to using the **tangible interaction method**:

Statement	Highly disagree				Highly agree
	1	2	3	4	5
The interface was easy to understand.					
Using the interface was frustrating.					
I was able to easily translate my intentions into actions.					
My understanding of programming has improved.					
Working in a pair made me more confident in our program.					

Multi-Touch:

Please rate the following statements about yourself on a scale from 1 to 5 in regard to using the **multi-touch only interaction method**:

Statement	Highly disagree				Highly agree
	1	2	3	4	5
The interface was easy to understand.					
Using the interface was frustrating.					
I was able to easily translate my intentions into actions.					
My understanding of programming has improved.					
Working in a pair made me more confident in our program.					

Comments:

Tangible Robotics – Data Sheet

Outputs:

Motors:

speed: range: [0,50]

ports:

A:	small motor, shovel
B:	big motor, left wheel
C:	big motor, right wheel
D:	/

Sound:

frequency: range: ~ [40,4000] (Hz)

C2:	65 Hz
CS2:	69 Hz
…	
B2:	123 Hz
…	
B7:	3951 Hz

Inputs:

Motors:

degrees:

- measured since start of program

speed:

- see outputs

Gyro:

degrees:

- measured since start of program
- positive values for clockwise rotation
- negative values for anti-clockwise rotation

Proximity:

- measures distance to object in front of sensor
- range: [0,2550] (mm)

Button:

- 1 or ‘true’ - pushed
- 0 or ‘false’ - not pushed

Light:

reflected:

- shines light, measures how much is reflected into sensor
- range: [0,100]

ambient:

- measures how much light hits sensor
- range: [0,100]

color:

- measures color of object in front of it
- range: [0,7]

transparent	- 0
black	- 1
blue	- 2
green	- 3
yellow	- 4
red	- 5
white	- 6
brown	- 7

Bibliography

Saira Anwar, Nicholas Alexander Bascou, Muhsin Menekse, and Asefeh Kardgar. A systematic review of studies on educational robotics. *Journal of Pre-College Engineering Education Research (J-PEER)*, 9(2):2, 2019. URL `https://doi.org/10.7771/2157-9288.1223`.

David J. Barnes. Teaching introductory java through lego mindstorms models. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '02, page 147–151, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134738. URL `https://doi.org/10.1145/563340.563397`.

Tracey Booth and Simone Stumpf. End-user experiences of visual and textual programming environments for arduino. In Yvonne Dittrich, Margaret Burnett, Anders Mørch, and David Redmiles, editors, *International symposium on end user development*, pages 25–39, Berlin, Heidelberg, 2013. Springer, Springer Berlin Heidelberg. ISBN 978-3-642-38706-7. URL `https://doi.org/10.1007/978-3-642-38706-7_4`.

Edgar Acosta Chaparro, Aybala Yuksel, Pablo Romero, and Sallyann Bryant. Factors affecting the perceived effectiveness of pair programming in higher education. In Pablo Romero, Judith Good, S Bryant, and EA Chaparro, editors, *Psychology of Programming Interest Group 17th Workshop*, pages 5–18, January 2005. URL `http://sro.sussex.ac.uk/id/eprint/20428/`.

Brigitte Denis and Sylviane Hubert. Collaborative learning in an educational robotics environment. *Computers in Human Behavior*, 17(5):465 – 480, 2001. ISSN

0747-5632. URL https://www.doi.org/10.1016/S0747-5632(01)00018-8.

Son Do-Lenh, Patrick Jermann, Sébastien Cuendet, Guillaume Zufferey, and Pierre Dillenbourg. Task performance vs. learning outcomes: A study of a tangible user interface in the classroom. In Martin Wolpers, Paul A. Kirschner, Maren Scheffel, Stefanie Lindstaedt, and Vania Dimitrova, editors, *Sustaining TEL: From Innovation to Learning and Practice*, pages 78–92, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-16020-2. URL https://doi.org10.1007/978-3-642-16020-2_6.

Diana Franklin, Charlotte Hill, Hilary A. Dwyer, Alexandria K. Hansen, Ashley Iveland, and Danielle B. Harlow. Initialization in scratch: Seeking knowledge transfer. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, page 217–222, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450336857. URL https://doi.org/10.1145/2839509.2844569.

Michael S. Horn, Erin Treacy Solovey, R. Jordan Crouser, and Robert J.K. Jacob. Comparing the use of tangible and graphical programming languages for informal science education. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, page 975–984, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605582467. URL https://doi.org/10.1145/1518701.1518851.

For Inspiration, Recognition of Science, and Technology. At a glance, 2020. URL https://www.firstinspires.org/about/at-a-glance.

Hiroshi Ishii, Carlo Ratti, Ben Piper, Yao Wang, Assaf Biderman, and Eran Ben-Joseph. Bringing clay and sand into digital design—continuous tangible user interfaces. *BT technology journal*, 22(4):287–299, 2004. URL https://www.doi.org/10.1023/B:BTTJ.0000047607.16164.16.

Sergi Jordà. The reactable. In *International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH*, Boston, USA, 02/08/2006 2006. Proceedings of the International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH 2006, ACM, Proceedings of the International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH 2006, ACM. ISBN 1-59593-364-6.

Seung Han Kim and Jae Wook Jeon. Introduction for freshmen to embedded systems using lego mindstorms. *IEEE transactions on education*, 52(1):99–108, 2008. URL `https://www.doi.org/10.1109/TE.2008.919809`.

Frank Klassner and Scott D Anderson. Lego mindstorms: Not just for k-12 anymore. *IEEE robotics & automation magazine*, 10(2):12–18, 2003. URL `https://www.doi.org/10.1109/MRA.2003.1213611`.

Rene Linden. Multitouchkit: A software framework for touch input and tangibles on tabletops and. Master's thesis, RWTH Aachen University, Aachen, September 2015.

Timothy S McNerney. From turtles to tangible programming bricks: explorations in physical language design. *Personal and Ubiquitous Computing*, 8(5):326–337, 2004. URL `https://www.doi.org/10.1007/s00779-004-0295-6`.

Edward F. Melcer and Katherine Isbister. Bots & (main)frames: Exploring the impact of tangible blocks and collaborative play in an educational programming game. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, page 1–14, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356206. URL `https://doi.org/10.1145/3173574.3173840`.

Seymour A Papert and Daniel H Watt. Assessment and documentation of a children's computer laboratory. *AI Memos (1959 - 2004)*, 1977. URL `http://hdl.handle.net/1721.1/6286`.

Kris Powers, Paul Gross, Steve Cooper, Myles McNally, Kenneth J. Goldman, Viera Proulx, and Martin Carlisle. Tools for teaching introductory programming: What works? In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, volume 38, page 560–561, New York, NY, USA, March 2006. Association for Computing Machinery. URL https://doi.org/10.1145/1124706.1121514.

David Preston. Pair programming as a model of collaborative learning: A review of the research. *J. Comput. Sci. Coll.*, 20(4):39–45, April 2005. ISSN 1937-4771. URL https://dl.acm.org/doi/10.5555/1047846.1047852.

Thomas W. Price and Tiffany Barnes. Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, page 91–99, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336307. doi: 10.1145/2787622.2787712. URL https://doi.org/10.1145/2787622.2787712.

Mara Saeli, Jacob Perrenet, Wim MG Jochems, and Bert Zwaneveld. Teaching programming in secondary school: A pedagogical content knowledge perspective. *Informatics in education*, 10(1):73–88, 2011. ISSN 1648-5831. URL https://www.persistent-identifier.nl/urn:nbn:nl:ui:25-0841b2fb-3fec-47a7-9301-73eb70685f25.

B. Schneider, P. Jermann, G. Zufferey, and P. Dillenbourg. Benefits of a tangible interface for collaborative learning and interaction. *IEEE Transactions on Learning Technologies*, 4(3):222–232, 2011. URL https://www.doi.org/10.1109/TLT.2010.36.

Bertrand Schneider, Paulo Blikstein, and Wendy Mackay. Combinatorix: A tangible user interface that supports collaborative learning of probabilities. In *Proceedings of the 2012 ACM International Conference on Interactive Tabletops and Surfaces*, ITS '12, page 129–132, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312097. URL https://doi.org/10.1145/2396636.2396656.

Schulministerium NRW. Zentralabitur an gymnasien und gesamtschulen, 2019. URL https://www.standardsicherung.schulministerium.nrw.de/cms/upload/abitur-gost/berichte/

Zentralabitur-Gymnasiale-Oberstufe-2019.pdf.

Joshua Shi, Armaan Shah, Garrett Hedman, and Eleanor O'Rourke. Pyrus: Designing a collaborative programming game to promote problem solving behaviors. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–12, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359702. doi: 10.1145/3290605.3300886. URL https://doi.org/10.1145/3290605.3300886.

Masanori Sugimoto, Tomoki Fujita, Haipeng Mi, and Aleksander Krzywinski. Robotable2: A novel programming environment using physical robots on a tabletop platform. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, ACE '11, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308274. URL https://doi.org/10.1145/2071423.2071436.

Simon Voelker, Kosuke Nakajima, Christian Thoresen, Yuichi Itoh, Kjell Ivar Øvergård, and Jan Borchers. Pucs: Detecting transparent, passive untouched capacitive widgets on unmodified multi-touch displays. In *Proceedings of the 2013 ACM International Conference on Interactive Tabletops and Surfaces*, ITS '13, page 101–104, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322713. URL https://doi.org/10.1145/2512349.2512791.

Simon Voelker, Christian Cherek, Jan Thar, Thorsten Karrer, Christian Thoresen, Kjell Ivar Øvergård, and Jan Borchers. Percs: Persistently trackable tangibles on capacitive multi-touch displays. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, page 351–356, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450337793. URL https://doi.org/10.1145/2807442.2807466.

Christopher Watson and Frederick W.B. Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, page 39–44, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328333. doi: 10.1145/2591708.2591749. URL `https://doi.org/10.1145/2591708.2591749`.

Index

Typeset May 18, 2020